Programming hybrid systems with synchronous languages

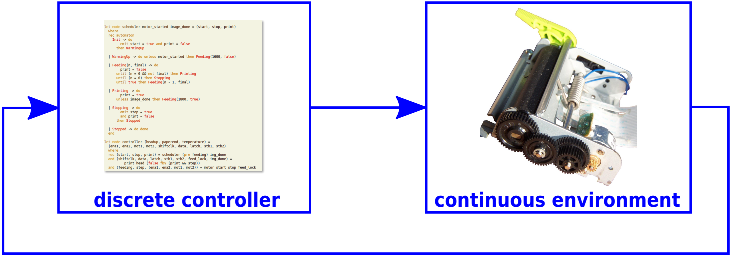
Timothy Bourke[1,2]

Albert Benveniste[1] Benoît Caillaud[1] Marc Pouzet[2,1]

1. INRIA

2. École normale supérieure (LIENS)

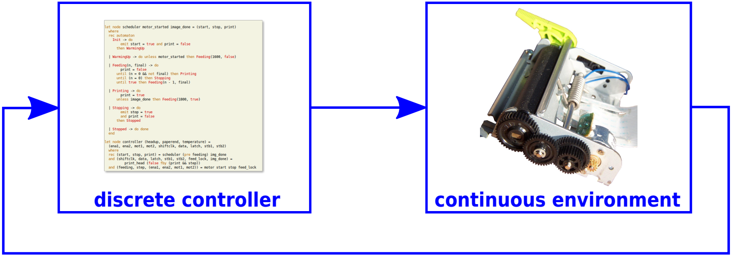
SAPLING 2011, November 18, Sydney, Australia



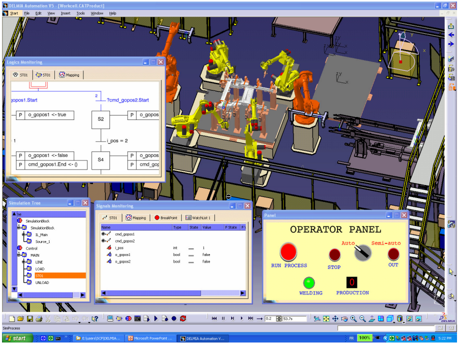
```
let node scheduler motor_started image_done = (start, stop, print)
  where
  rec automaton
    Init -> do
        emit start = true and print = false
      then WarmingUp

  | WarmingUp -> do unless motor_started then Feeding(1600, false)

  | Feeding(n, final) -> do
        print = false
      until (n = 0 && not final) then Printing
      until (n = 0) then Stopping
      until true then Feeding(n - 1, final)

  | Printing -> do
        print = true
      unless image_done then Feeding(1800, true)

  | Stopping -> do
        emit stop = true
        and print = false
      then Stopped

  | Stopped -> do done
  end

let node controller (headup, paperend, temperature) =
  (ena1, ena2, mot1, mot2, shiftclk, data, latch, stb1, stb2)
  where
  rec (start, stop, print) = scheduler (pre feeding) img_done
  and (shiftclk, data, latch, stb1, stb2, feed_lock, img_done) =
          print_head (false fby (print && step))
  and (feeding, step, (ena1, ena2, mot1, mot2)) = motor start stop feed_lock
```

discrete controller

continuous environment



```
let node scheduler motor_started image_done = (start, stop, print)
  where
  rec automaton
    Init -> do
        emit start = true and print = false
      then WarmingUp

  | WarmingUp -> do unless motor_started then Feeding(1600, false)

  | Feeding(n, final) -> do
        print = false
      until (n = 0 && not final) then Printing
      until (n = 0) then Stopping
      until true then Feeding(n - 1, final)

  | Printing -> do
        print = true
      unless image_done then Feeding(1800, true)

  | Stopping -> do
        emit stop = true
        and print = false
      then Stopped

  | Stopped -> do done
  end

let node controller (headup, paperend, temperature) =
  (ena1, ena2, mot1, mot2, shiftclk, data, latch, stb1, stb2)
  where
  rec (start, stop, print) = scheduler (pre feeding) img_done
  and (shiftclk, data, latch, stb1, stb2, feed_lock, img_done) =
          print_head (false fby (print && step))
  and (feeding, step, (ena1, ena2, mot1, mot2)) = motor start stop feed_lock
```



Name	Type	State	Value	F State
cmd_gopos1				
cmd_gopos2				
i_pos	int	—	1	
o_gopos1	bool	—	false	
o_gopos2	bool	—	false	

Dassault Systèmes Delmia and Catia http://www.3ds.com/products

Outline

Dataflow programming

Research objectives

Continuous modelling and simulation

Typing and compilation

Demonstration and conclusion

Dataflow programming: core concepts

Programming with streams

Dataflow programming: core concepts

Programming with streams

constants $1 = 1 \quad 1 \quad 1 \quad 1 \quad \cdots$

Dataflow programming: core concepts

Programming with streams

constants	`1`	$=$	1	1	1	1	$\cdots$
operators	`x + y`	$=$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$\cdots$

Dataflow programming: core concepts

Programming with streams

constants	1	=	1	1	1	1	$\cdots$
operators	x + y	=	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$\cdots$
unit delay	0 **fby** (x + y)	=	0	$x_0 + y_0$	$x_1 + x_1$	$x_2 + x_2$	$\cdots$

Dataflow programming: core concepts

Programming with streams

constants $\quad 1 = 1 \quad 1 \quad 1 \quad 1 \quad \cdots$

operators $\quad \mathsf{x} + \mathsf{y} = x_0 + y_0 \quad x_1 + y_1 \quad x_2 + y_2 \quad x_3 + y_3 \quad \cdots$

unit delay $\quad 0\ \mathbf{fby}\ (\mathsf{x} + \mathsf{y}) = 0 \quad x_0 + y_0 \quad x_1 + x_1 \quad x_2 + x_2 \quad \cdots$

Programming with iterating machines

Caspi and Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. 1998.

Dataflow programming: core concepts

Programming with streams

constants	1	$=$	1	1	1	1	$\cdots$
operators	x + y	$=$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$\cdots$
unit delay	0 **fby** (x + y)	$=$	0	$x_0 + y_0$	$x_1 + x_1$	$x_2 + x_2$	$\cdots$

Programming with iterating machines

Caspi and Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. 1998.

```
let one () = 1
```

Dataflow programming: core concepts

Programming with streams

constants	1	=	1	1	1	1	$\cdots$
operators	x + y	=	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$\cdots$
unit delay	0 **fby** (x + y)	=	0	$x_0 + y_0$	$x_1 + x_1$	$x_2 + x_2$	$\cdots$

Programming with iterating machines

Caspi and Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. 1998.

```
let one () = 1

let add x y = x + y
```

Dataflow programming: core concepts

Programming with streams

constants	1	$=$	1	1	1	1	$\cdots$
operators	x + y	$=$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$\cdots$
unit delay	0 **fby** (x + y)	$=$	0	$x_0 + y_0$	$x_1 + x_1$	$x_2 + x_2$	$\cdots$

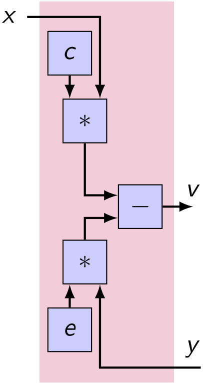
Programming with iterating machines

Caspi and Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. 1998.

```
let one () = 1

let add x y = x + y
```

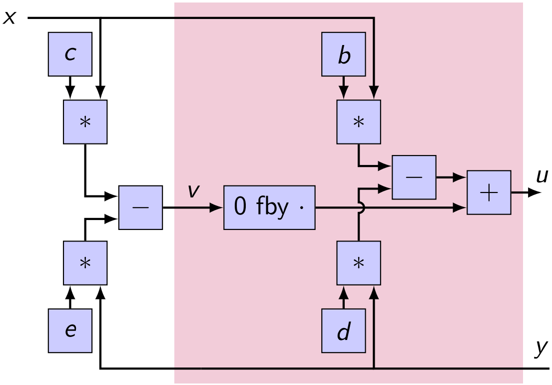
```
let delay () = { init = true; pre = nil }

let delay_step self x y =
  let result =
    if self.init then x else self.pre in
  self.pre <- y;
  self.init <- false;
  result

let delay_reset self = self.init <- true
```

Dataflow programming: composition and syntax



`v = c * x - e * y`

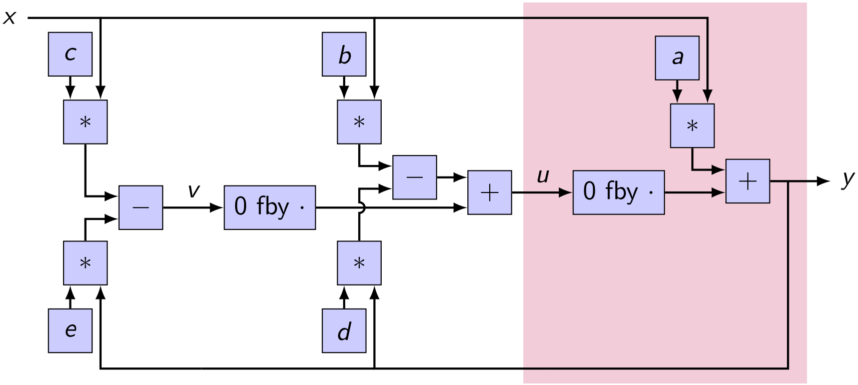
Dataflow programming: composition and syntax



u = b ∗ x − d ∗ y + (0.0 **fby** v)

and v = c ∗ x − e ∗ y

Dataflow programming: composition and syntax

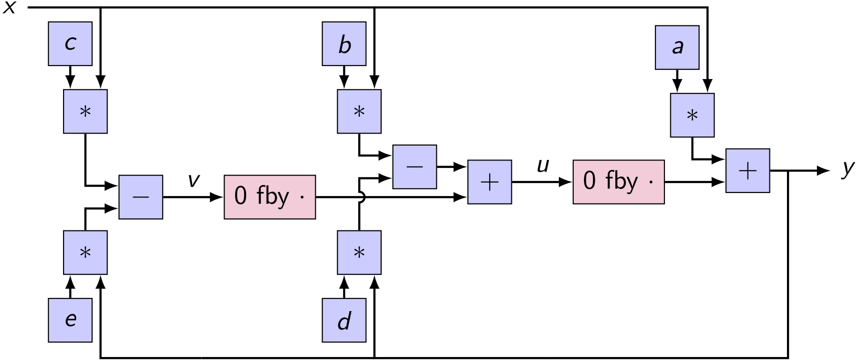


rec y = a ∗ x + (0.0 **fby** u)

and u = b ∗ x − d ∗ y + (0.0 **fby** v)

and v = c ∗ x − e ∗ y

Dataflow programming: composition and syntax

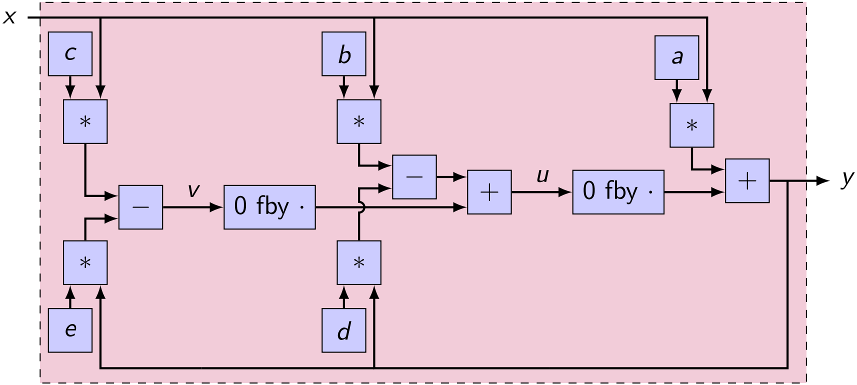


rec y = a ∗ x + (0.0 **fby** u)

and u = b ∗ x − d ∗ y + (0.0 **fby** v)

and v = c ∗ x − e ∗ y

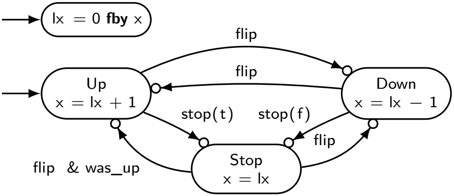
Dataflow programming: composition and syntax



```
let node iir_filter_2 x = y where
  rec y = a * x + (0.0 fby u)
  and u = b * x − d * y + (0.0 fby v)
  and v = c * x − e * y
```

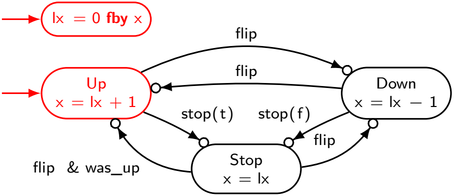
(Mixed) Dataflow programming: control structures

```
let node counter (flip, stop) = x
 where
 rec lx = 0 fby x
 and automaton
     | Up →
        do
          x = lx + 1
        until flip then Down
            | stop then Stop(true)
        done

     | Down →
        do
          x = lx − 1
        until flip then Up
            | stop then Stop(false)
        done

     | Stop(was_up) →
        do
          x = lx
        until flip & was_up then Up
            | toggle then Down
        done
     end
```



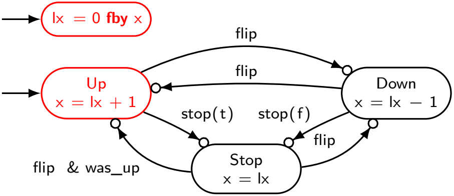
(Mixed) Dataflow programming: control structures

```
let node counter (flip, stop) = x
 where
 rec lx = 0 fby x
 and automaton
     | Up →
       do
         x = lx + 1
       until flip then Down
           | stop then Stop(true)
       done

     | Down →
       do
         x = lx − 1
       until flip then Up
           | stop then Stop(false)
       done

     | Stop(was_up) →
       do
         x = lx
       until flip & was_up then Up
           | toggle then Down
       done
     end
```



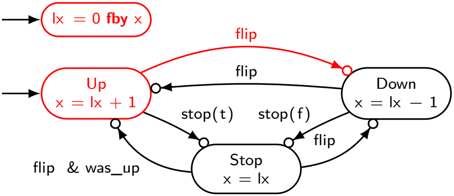
(Mixed) Dataflow programming: control structures

```
let node counter (flip , stop) = x
 where
 rec lx = 0 fby x
 and automaton
     | Up →
       do
         x = lx + 1
       until flip then Down
           | stop then Stop(true)
       done

     | Down →
       do
         x = lx − 1
       until flip then Up
           | stop then Stop(false)
       done

     | Stop(was_up) →
       do
         x = lx
       until flip & was_up then Up
           | toggle then Down
       done
     end
```



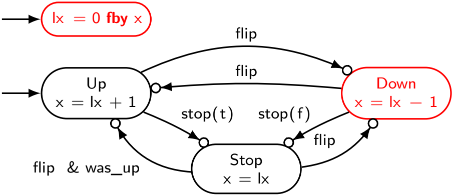
(Mixed) Dataflow programming: control structures

```
let node counter (flip, stop) = x
 where
 rec lx = 0 fby x
 and automaton
     | Up →
       do
         x = lx + 1
       until flip then Down
           | stop then Stop(true)
       done

     | Down →
       do
         x = lx − 1
       until flip then Up
           | stop then Stop(false)
       done

     | Stop(was_up) →
       do
         x = lx
       until flip & was_up then Up
           | toggle then Down
       done
     end
```



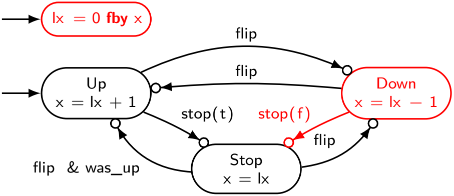
(Mixed) Dataflow programming: control structures

```
let node counter (flip, stop) = x
 where
 rec lx = 0 fby x
 and automaton
     | Up→
       do
         x = lx + 1
       until flip then Down
           | stop then Stop(true)
       done

     | Down→
       do
         x = lx − 1
       until flip then Up
           | stop then Stop(false)
       done

     | Stop(was_up)→
       do
         x = lx
       until flip & was_up then Up
           | toggle then Down
       done
     end
```



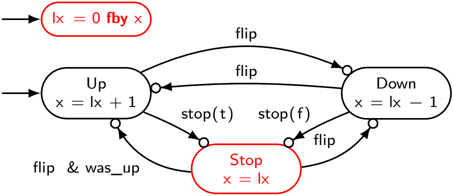
(Mixed) Dataflow programming: control structures

```
let node counter (flip, stop) = x
 where
 rec lx = 0 fby x
 and automaton
      | Up →
        do
          x = lx + 1
        until flip then Down
            | stop then Stop(true)
        done

      | Down →
        do
          x = lx − 1
        until flip then Up
            | stop then Stop(false)
        done

      | Stop(was_up) →
        do
          x = lx
        until flip & was_up then Up
            | toggle then Down
        done
      end
```



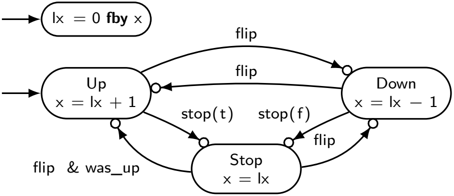
(Mixed) Dataflow programming: control structures

```
let node counter (flip , stop) = x
 where
 rec lx = 0 fby x
 and automaton
      | Up →
        do
          x = lx + 1
        until flip then Down
            | stop then Stop(true)
        done

      | Down →
        do
          x = lx − 1
        until flip then Up
            | stop then Stop(false)
        done

      | Stop(was_up) →
        do
          x = lx
        until flip & was_up then Up
            | toggle then Down
        done
      end
```



(Mixed) Dataflow programming: control structures

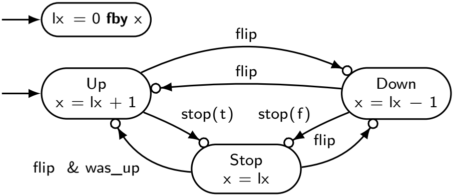
```
let node counter (flip , stop) = x
 where
 rec lx = 0 fby x
 and automaton
     | Up →
       do
         x = lx + 1
       until flip then Down
           | stop then Stop(true)
       done

     | Down →
       do
         x = lx − 1
       until flip then Up
           | stop then Stop(false)
       done

     | Stop(was_up) →
       do
         x = lx
       until flip & was_up then Up
           | toggle then Down
       done
     end
```



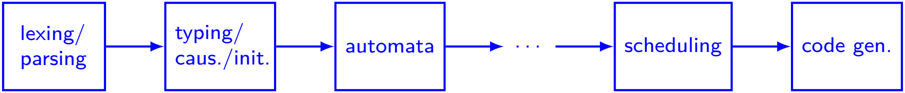
- (Parameterized) modes contain definitions, incl. automata
- **until**: weak preemption (test after)
- **unless**: strong preemption (test before)
- **then**: enter-with-reset
- **continue**: entry-by-history

(Mixed) Dataflow programming: control structures

```
let node counter (flip, stop) = x
 where
 rec lx = 0 fby x
 and automaton
     | Up →
       do
         x = lx + 1
       until flip then Down
           | stop then Stop(true)
       done

     | Down →
       do
         x = lx - 1
       until flip then Up
           | stop then Stop(false)
       done

     | Stop(was_up) →
       do
         x = lx
       until flip & was_up then Up
           | toggle then Down
       done
     end
```



- (Parameterized) modes contain definitions, incl. automata
- **until**: weak preemption (test after)
- **unless**: strong preemption (test before)
- **then**: enter-with-reset
- **continue**: entry-by-history

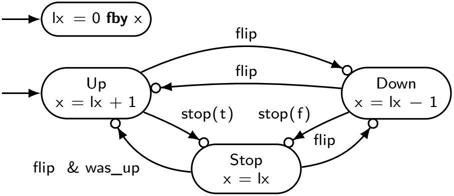


lexing/ parsing → typing/ caus./init. → automata → · · · → scheduling → code gen.

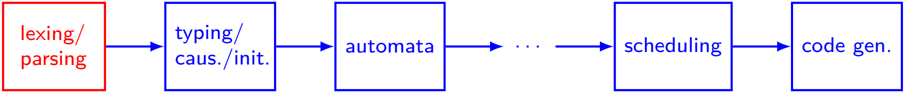
Colaço, Pagano and Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. 2005.

(Mixed) Dataflow programming: control structures

```
let node counter (flip, stop) = x
 where
 rec lx = 0 fby x
 and automaton
     | Up →
       do
         x = lx + 1
       until flip then Down
           | stop then Stop(true)
       done

     | Down →
       do
         x = lx − 1
       until flip then Up
           | stop then Stop(false)
       done

     | Stop(was_up) →
       do
         x = lx
       until flip & was_up then Up
           | toggle then Down
       done
     end
```



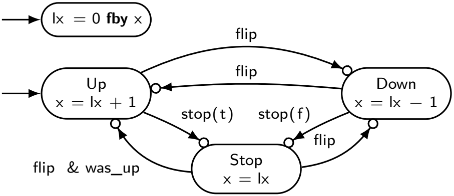
- (Parameterized) modes contain definitions, incl. automata
- **until**: weak preemption (test after)
- **unless**: strong preemption (test before)
- **then**: enter-with-reset
- **continue**: entry-by-history



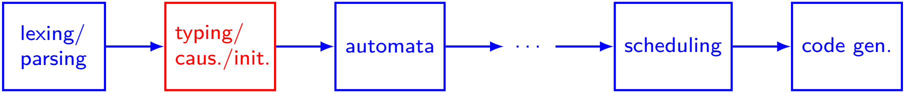
Colaço, Pagano and Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. 2005.

(Mixed) Dataflow programming: control structures

```
let node counter (flip, stop) = x
 where
 rec lx = 0 fby x
 and automaton
     | Up →
       do
         x = lx + 1
       until flip then Down
           | stop then Stop(true)
       done

     | Down →
       do
         x = lx − 1
       until flip then Up
           | stop then Stop(false)
       done

     | Stop(was_up) →
       do
         x = lx
       until flip & was_up then Up
           | toggle then Down
       done
     end
```



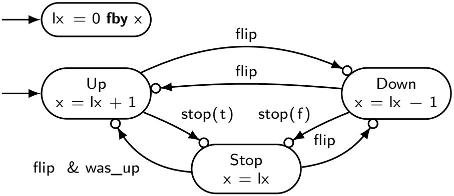
- (Parameterized) modes contain definitions, incl. automata
- **until**: weak preemption (test after)
- **unless**: strong preemption (test before)
- **then**: enter-with-reset
- **continue**: entry-by-history



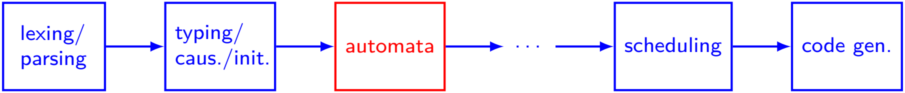
Colaço, Pagano and Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. 2005.

(Mixed) Dataflow programming: control structures

```
let node counter (flip, stop) = x
 where
 rec lx = 0 fby x
 and automaton
      | Up →
        do
          x = lx + 1
        until flip then Down
            | stop then Stop(true)
        done

      | Down →
        do
          x = lx − 1
        until flip then Up
            | stop then Stop(false)
        done

      | Stop(was_up) →
        do
          x = lx
        until flip & was_up then Up
            | toggle then Down
        done
    end
```



- (Parameterized) modes contain definitions, incl. automata
- **until**: weak preemption (test after)
- **unless**: strong preemption (test before)
- **then**: enter-with-reset
- **continue**: entry-by-history

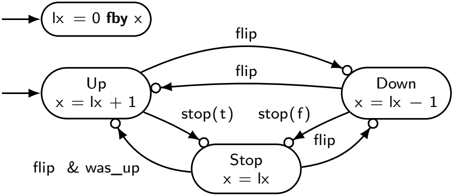


lexing/ parsing → typing/ caus./init. → automata → · · · → scheduling → code gen.

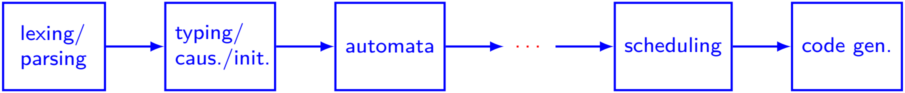
Colaço, Pagano and Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. 2005.

(Mixed) Dataflow programming: control structures

```
let node counter (flip, stop) = x
 where
 rec lx = 0 fby x
 and automaton
     | Up →
       do
         x = lx + 1
       until flip then Down
           | stop then Stop(true)
       done

     | Down →
       do
         x = lx − 1
       until flip then Up
           | stop then Stop(false)
       done

     | Stop(was_up) →
       do
         x = lx
       until flip & was_up then Up
           | toggle then Down
       done
     end
```



- (Parameterized) modes contain definitions, incl. automata
- **until**: weak preemption (test after)
- **unless**: strong preemption (test before)
- **then**: enter-with-reset
- **continue**: entry-by-history

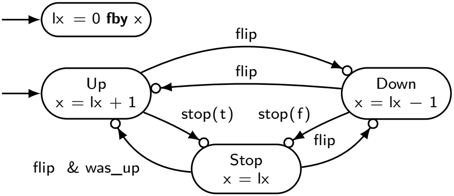


lexing/ parsing → typing/ caus./init. → automata → · · · → scheduling → code gen.

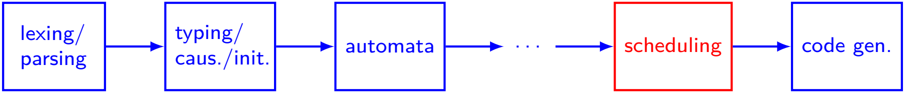
Colaço, Pagano and Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. 2005.

(Mixed) Dataflow programming: control structures

```
let node counter (flip, stop) = x
 where
 rec lx = 0 fby x
 and automaton
      | Up →
        do
          x = lx + 1
        until flip then Down
            | stop then Stop(true)
        done

      | Down →
        do
          x = lx − 1
        until flip then Up
            | stop then Stop(false)
        done

      | Stop(was_up) →
        do
          x = lx
        until flip & was_up then Up
            | toggle then Down
        done
    end
```



- (Parameterized) modes contain definitions, incl. automata
- **until**: weak preemption (test after)
- **unless**: strong preemption (test before)
- **then**: enter-with-reset
- **continue**: entry-by-history

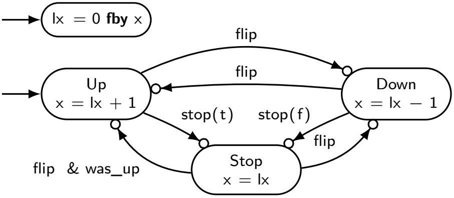


lexing/ parsing → typing/ caus./init. → automata → · · · → scheduling → code gen.

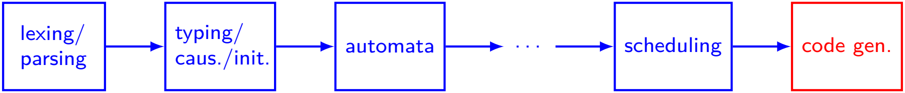
Colaço, Pagano and Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. 2005.

(Mixed) Dataflow programming: control structures

```
let node counter (flip , stop) = x
 where
 rec lx = 0 fby x
 and automaton
     | Up →
       do
         x = lx + 1
       until flip then Down
           | stop then Stop(true)
       done

     | Down →
       do
         x = lx − 1
       until flip then Up
           | stop then Stop(false)
       done

     | Stop(was_up) →
       do
         x = lx
       until flip & was_up then Up
           | toggle then Down
       done
     end
```



- (Parameterized) modes contain definitions, incl. automata
- **until**: weak preemption (test after)
- **unless**: strong preemption (test before)
- **then**: enter-with-reset
- **continue**: entry-by-history


lexing/ parsing → typing/ caus./init. → automata → · · · → scheduling → code gen.

Colaço, Pagano and Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. 2005.

Dataflow programming languages

- Kahn Networks Kahn. The Semantics of a Simple Language for Parallel Programming 1974.
- Lucid Wadge and Ashcroft. LUCID, the dataflow programming language. 1985.

Asynchronous

- Lustre Caspi, Pilaud, Halbwachs, and Plaice. Lustre: A Declarative Language for Programming Synchronous Systems. 1987.
  - Clock calculus
  - Deterministic, bounded memory, bounded execution time
- SCADE 6 http://www.esterel-technologies.com/products/scade-suite/
  - Industrial (extended) version of Lustre
  - Used in critical systems (DO-178B certified)
  - Airbus flight control; Train braking; Nuclear safety
- Lucid Synchrone Caspi and Pouzet. A Functional Extension to Lustre. 1995.
  - Higher-order dataflow
  - Hierarchical automata
  - Signals

Synchronous

- Ptolemy http://ptolemy.eecs.berkeley.edu/
- (subsets of) Simulink (and Stateflow) http://www.mathworks.com/products/simulink/

Dataflow programming languages

- Kahn Networks Kahn. The Semantics of a Simple Language for Parallel Programming 1974.
- Lucid Wadge and Ashcroft. LUCID, the dataflow programming language. 1985.

Asynchronous

- Lustre Caspi, Pilaud, Halbwachs, and Plaice. Lustre: A Declarative Language for Programming Synchronous Systems. 1987.
  - Clock calculus
  - Deterministic, bounded memory, bounded execution time
- SCADE 6 http://www.esterel-technologies.com/products/scade-suite/
  - Industrial (extended) version of Lustre
  - Used in critical systems (DO-178B certified)
  - Airbus flight control; Train braking; Nuclear safety
- Lucid Synchrone Caspi and Pouzet. A Functional Extension to Lustre. 1995.
  - Higher-order dataflow
  - Hierarchical automata
  - Signals

Synchronous

- Ptolemy http://ptolemy.eecs.berkeley.edu/
- (subsets of) Simulink (and Stateflow) http://www.mathworks.com/products/simulink/

Dataflow programming languages

- Kahn Networks Kahn. The Semantics of a Simple Language for Parallel Programming 1974.
- Lucid Wadge and Ashcroft. LUCID, the dataflow programming language. 1985.

Asynchronous

- Lustre Caspi, Pilaud, Halbwachs, and Plaice. Lustre: A Declarative Language for Programming Synchronous Systems. 1987.
  - Clock calculus
  - Deterministic, bounded memory, bounded execution time
- SCADE 6 http://www.esterel-technologies.com/products/scade-suite/
  - Industrial (extended) version of Lustre
  - Used in critical systems (DO-178B certified)
  - Airbus flight control; Train braking; Nuclear safety
- Lucid Synchrone Caspi and Pouzet. A Functional Extension to Lustre. 1995.
  - Higher-order dataflow
  - Hierarchical automata
  - Signals

Synchronous

- Ptolemy http://ptolemy.eecs.berkeley.edu/
- (subsets of) Simulink (and Stateflow) http://www.mathworks.com/products/simulink/

So, what's to do?

- We want a language for programming **complex discrete systems** and modelling their **physical environments**
- (Also: embedded software that includes physical models)

So, what's to do?

- We want a language for programming **complex discrete systems** and modelling their **physical environments**
- (Also: embedded software that includes physical models)

- Something like Simulink/Stateflow, but
    - Simpler and more consistent semantics and compilation
    - Better understand interactions between discrete and continuous
    - Simpler treatment of automata
    - Certifiability for the discrete parts

  Understand and improve the design of such modelling tools

Approach

- Add Ordinary Differential Equations to an existing synchronous language
- Two concrete reasons:
  - Increase modelling power (hybrid programming)
  - Exploit existing compiler (target for code generation)
- Simulate with an external off-the-shelf numerical solver (Sundials CVODE, Hindmarsh et al. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. 2005.)
- Conservative extension: synchronous functions are compiled, optimized, and executed as per usual.

Approach

- Add Ordinary Differential Equations to an existing synchronous language
- Two concrete reasons:
  - Increase modelling power (hybrid programming)
  - Exploit existing compiler (target for code generation)
- Simulate with an external off-the-shelf numerical solver (Sundials CVODE, Hindmarsh et al. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. 2005.)
- Conservative extension: synchronous functions are compiled, optimized, and executed as per usual.

Approach

- Add Ordinary Differential Equations to an existing synchronous language
- Two concrete reasons:
  - Increase modelling power (hybrid programming)
  - Exploit existing compiler (target for code generation)
- Simulate with an external off-the-shelf numerical solver (Sundials CVODE, Hindmarsh et al. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. 2005.)
- Conservative extension: synchronous functions are compiled, optimized, and executed as per usual.

Outline

Dataflow programming

Research objectives

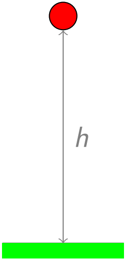
Continuous modelling and simulation

Typing and compilation

Demonstration and conclusion

Bouncing ball

model



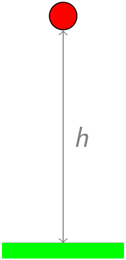
$$F = m \cdot a$$

$$-g = m \cdot \frac{d^2 h(t)}{dt^2}$$

$$\frac{d^2 h(t)}{dt^2} = -g/m$$

Bouncing ball

model



$$F = m \cdot a$$

$$-g = m \cdot \frac{d^2 h(t)}{dt^2}$$

$$\frac{d^2 h(t)}{dt^2} = -g/m$$

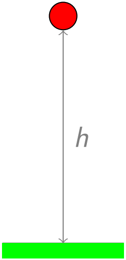
$$\dot{v} = -g/m \qquad v(0) = v_0$$

$$\dot{h} = v \qquad h(0) = h_0$$

Causal first-order ODEs

Bouncing ball

model



$$F = m \cdot a$$
$$-g = m \cdot \frac{d^2 h(t)}{dt^2}$$
$$\frac{d^2 h(t)}{dt^2} = -g/m$$

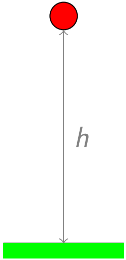
$$\dot{v} = -g/m \qquad v(0) = v_0$$
$$\dot{h} = v \qquad h(0) = h_0$$

Causal first-order ODEs

$$v(t) = v_0 + \int_0^t (-g/m)\,.d\tau$$
$$h(t) = h_0 + \int_0^t v(\tau)\,.d\tau$$

Bouncing ball

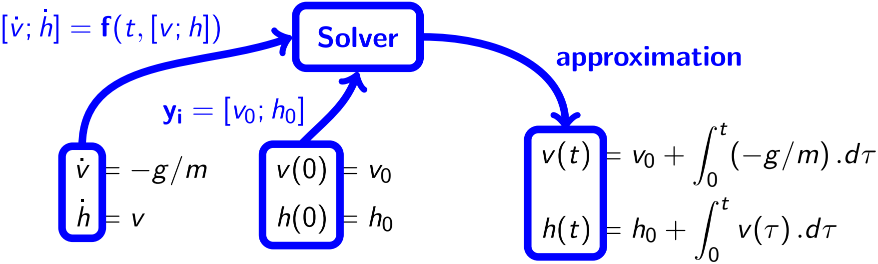
model



$$F = m \cdot a$$

$$-g = m \cdot \frac{d^2 h(t)}{dt^2}$$

$$\frac{d^2 h(t)}{dt^2} = -g/m$$



$$[\dot{v}; \dot{h}] = \mathbf{f}(t, [v; h])$$

Solver

$$\mathbf{y_i} = [v_0; h_0]$$

approximation

$$\dot{v} = -g/m$$
$$\dot{h} = v$$

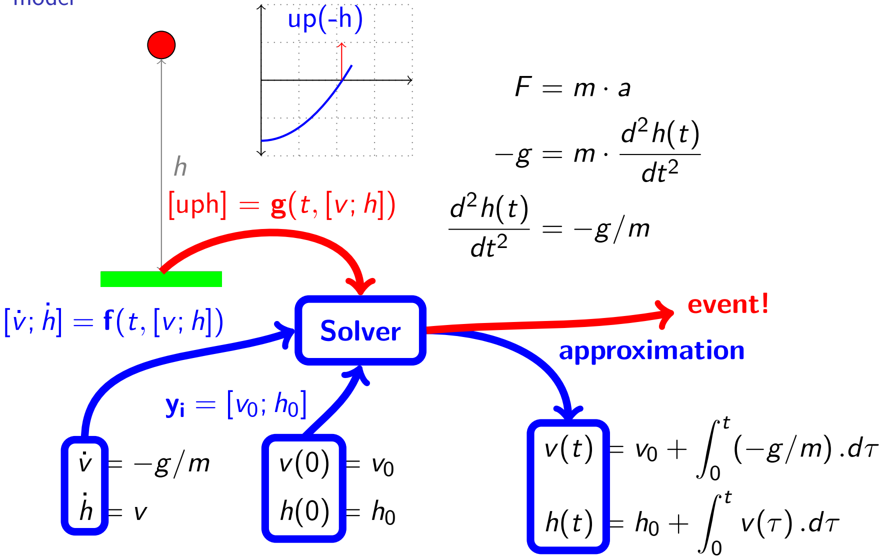
$$v(0) = v_0$$
$$h(0) = h_0$$

$$v(t) = v_0 + \int_0^t (-g/m)\,.d\tau$$
$$h(t) = h_0 + \int_0^t v(\tau)\,.d\tau$$

Causal first-order ODEs

Bouncing ball

model



$$F = m \cdot a$$

$$-g = m \cdot \frac{d^2 h(t)}{dt^2}$$

$$\frac{d^2 h(t)}{dt^2} = -g/m$$

$$[\mathrm{uph}] = \mathbf{g}(t, [v; h])$$

$$[\dot{v}; \dot{h}] = \mathbf{f}(t, [v; h])$$

$$\mathbf{y_i} = [v_0; h_0]$$

$$\boxed{\begin{array}{c} \dot{v} \\ \dot{h} \end{array}} \begin{array}{l} = -g/m \\ = v \end{array} \qquad \boxed{\begin{array}{c} v(0) \\ h(0) \end{array}} \begin{array}{l} = v_0 \\ = h_0 \end{array}$$

$$\boxed{\begin{array}{c} v(t) \\ h(t) \end{array}} \begin{array}{l} = v_0 + \int_0^t (-g/m)\,.d\tau \\ = h_0 + \int_0^t v(\tau)\,.d\tau \end{array}$$

Causal first-order ODEs

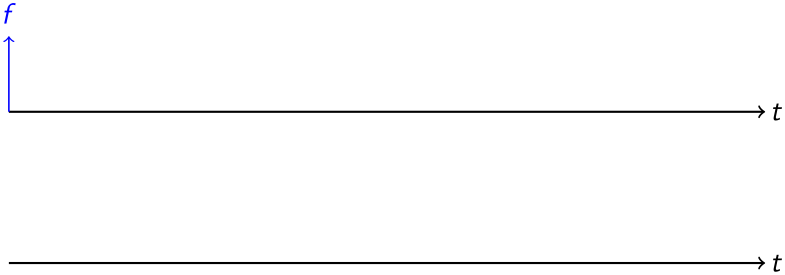
Solver execution

Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$

- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f or g

Solver execution

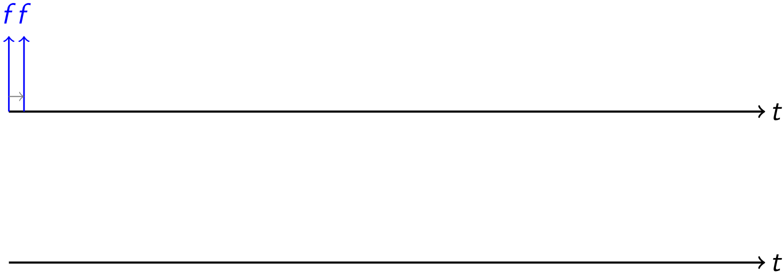
Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f or g

Solver execution

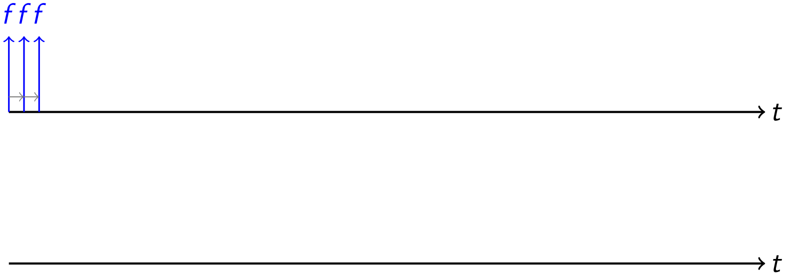
Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f or g

Solver execution

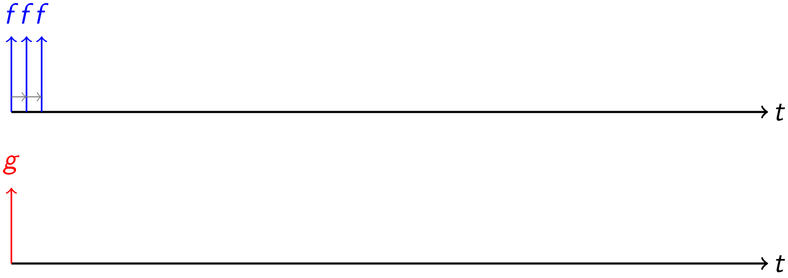
Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f or g

Solver execution

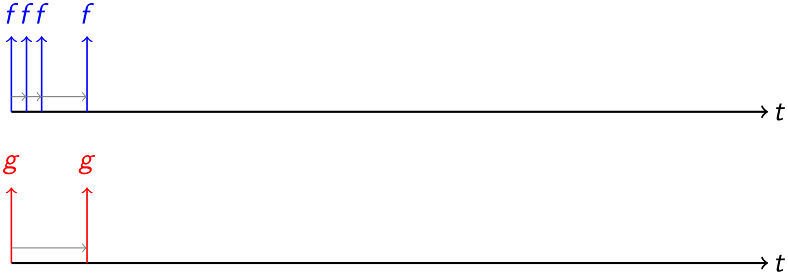
Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f or g

Solver execution

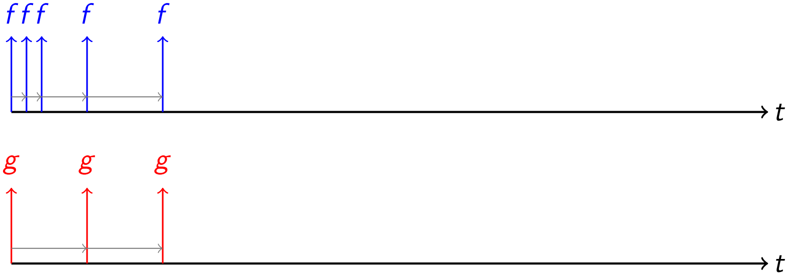
Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
    - No side-effects within f or g

Solver execution

Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$

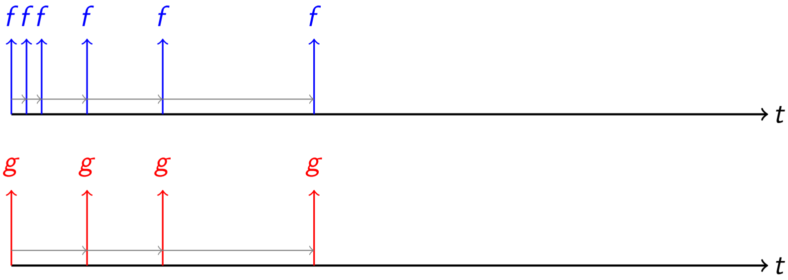


- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
    - No side-effects within f or g

Solver execution

Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$

1. approximation error too large

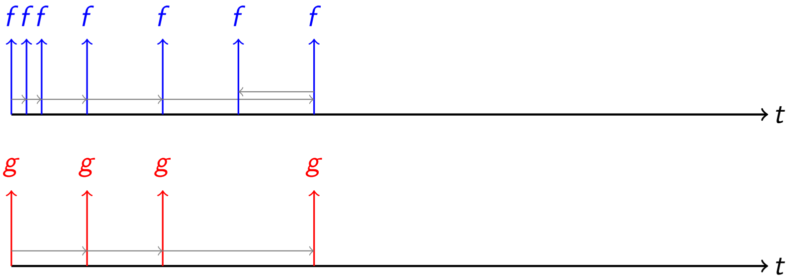


- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f or g

Solver execution

Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$

1. approximation error too large

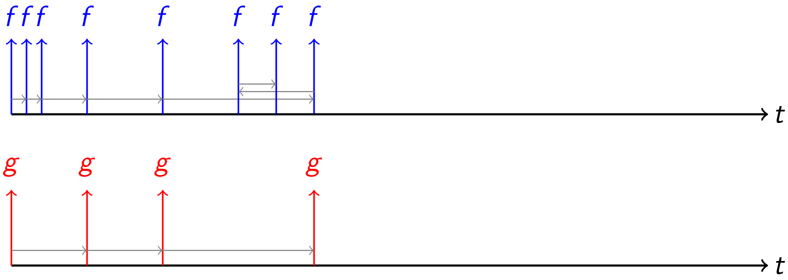


- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
    - No side-effects within f or g

Solver execution

Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$

1. approximation error too large

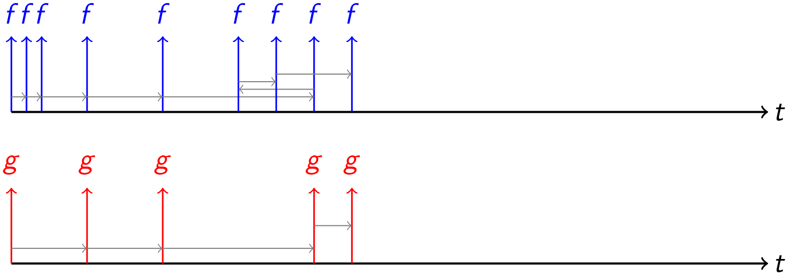


- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f or g

Solver execution

Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$

1. approximation error too large

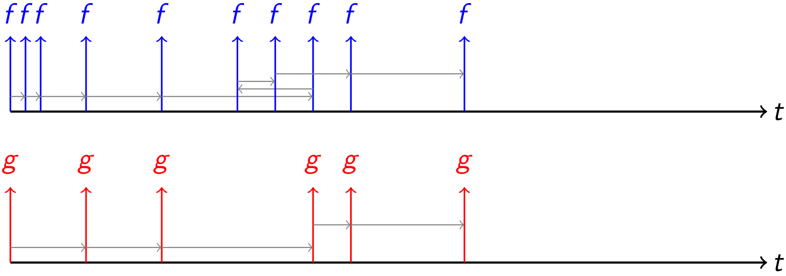


- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f or g

Solver execution

Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$

1. approximation error too large

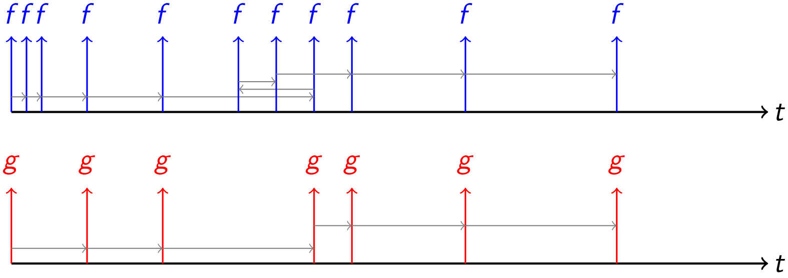


- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f or g

Solver execution

Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$

1. approximation error too large

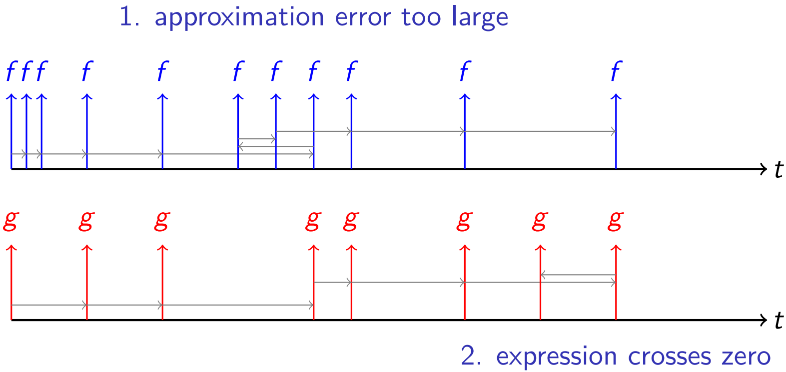


2. expression crosses zero

- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f or g

Solver execution

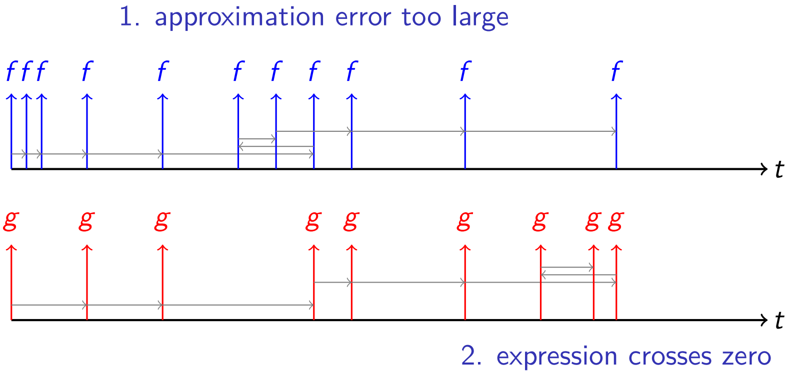
Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f or g

Solver execution

Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



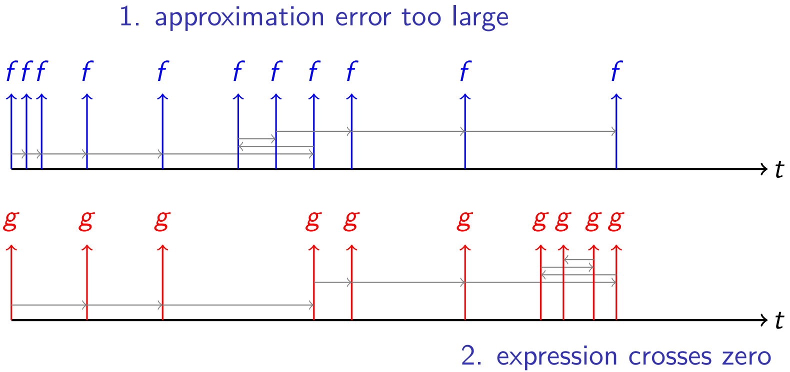
1. approximation error too large

2. expression crosses zero

- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f or g

Solver execution

Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



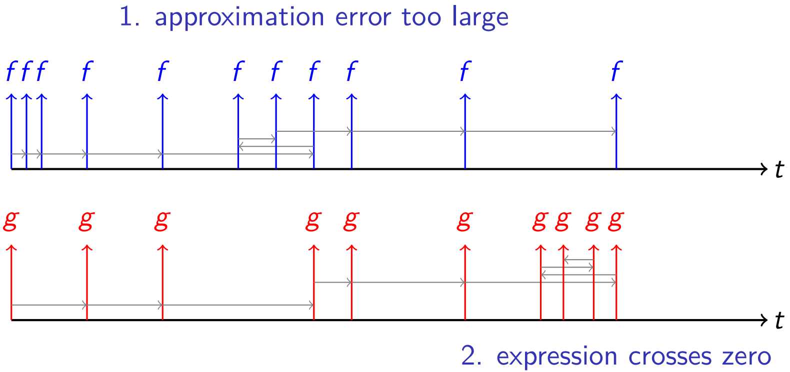
1. approximation error too large

2. expression crosses zero

- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f or g

Solver execution

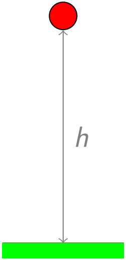
Give solver two functions: $dy = f_\sigma(t,y)$, $upz = g_\sigma(t,y)$



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
    - No side-effects within f or g

Bouncing ball

program



$$\dot{v} = -g/m \qquad v(0) = v_0$$

$$\dot{h} = v \qquad h(0) = h_0$$

reset v to $-0.8 \cdot v$ when h becomes 0

```
let hybrid ball () =
  let
  rec der v = (-. g / m) init v0
                 reset (-. 0.8 *. last v) every up(-. h)
  and der h = v init h0
  in (v, h)
```

Outline

Dataflow programming

Research objectives

Continuous modelling and simulation

Typing and compilation

Demonstration and conclusion

Which programs make sense?

Given:

```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

Which programs make sense?

Given:

```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

Evaluate:

```
der time = 1.0 init 0.0
and
y = sum(time)
```

Which programs make sense?

Given:

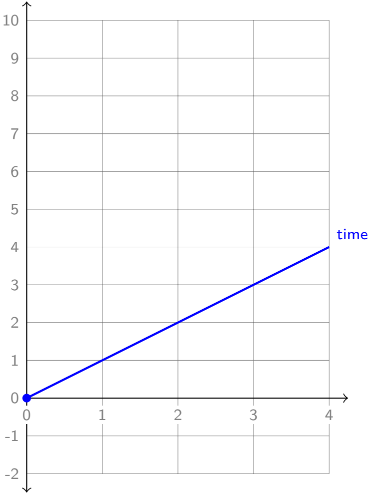
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

Evaluate:

```
der time = 1.0 init 0.0
and
y = sum(time)
```

Interpretation:

- Option 1: $\mathbb{N} \subseteq \mathbb{R}$
- Option 2: depends on solver
- Option 3: infinitesimal steps
- Option 4: type and reject



Which programs make sense?

Given:

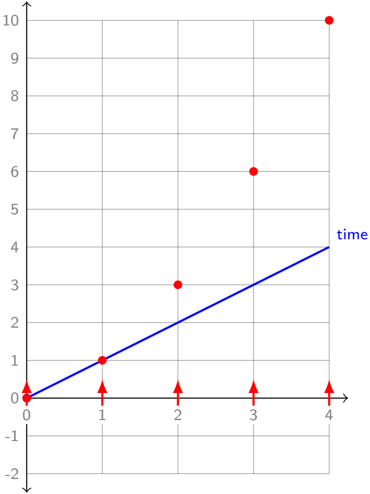
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

Evaluate:

```
der time = 1.0 init 0.0
and
y = sum(time)
```

Interpretation:

- Option 1: $\mathbb{N} \subseteq \mathbb{R}$
- Option 2: depends on solver
- Option 3: infinitesimal steps
- Option 4: type and reject



Which programs make sense?

Given:

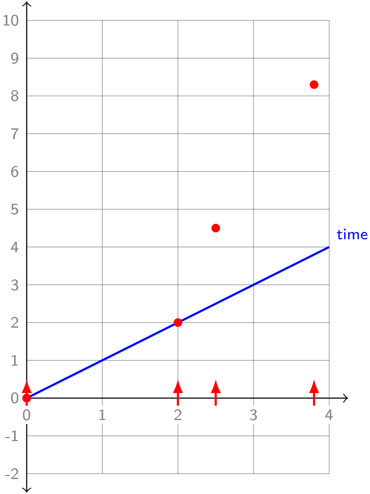
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

Evaluate:

```
der time = 1.0 init 0.0
and
y = sum(time)
```

Interpretation:

- Option 1: $\mathbb{N} \subseteq \mathbb{R}$
- Option 2: depends on solver
- Option 3: infinitesimal steps
- Option 4: type and reject



Which programs make sense?

Given:

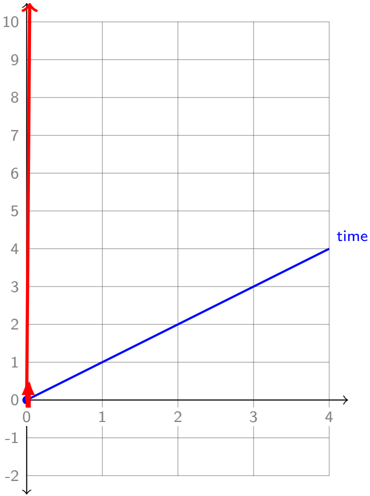
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

Evaluate:

```
der time = 1.0 init 0.0
and
y = sum(time)
```

Interpretation:

- Option 1: $\mathbb{N} \subseteq \mathbb{R}$
- Option 2: depends on solver
- Option 3: infinitesimal steps
- Option 4: type and reject



Which programs make sense?

Given:

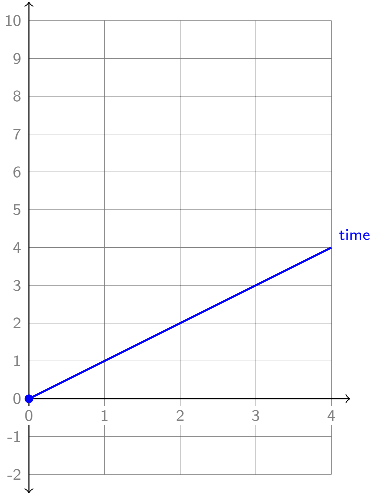
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

Evaluate:

```
der time = 1.0 init 0.0
and   ✗
y = sum(time)
```

Interpretation:

- Option 1: $\mathbb{N} \subseteq \mathbb{R}$
- Option 2: depends on solver
- Option 3: infinitesimal steps
- Option 4: type and reject



Which programs make sense?

Given:

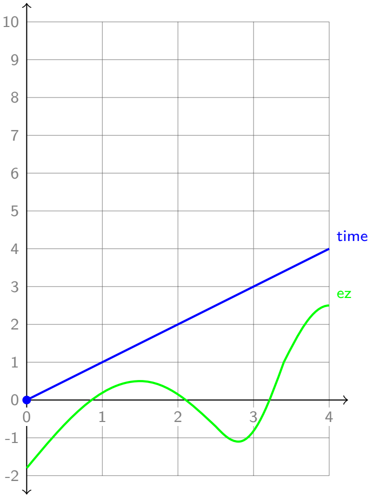
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

Evaluate:

```
der time = 1.0 init 0.0
and
y = sum(time) every up(ez) init 0.0
```

Interpretation:

- Option 1: $\mathbb{N} \subseteq \mathbb{R}$
- Option 2: depends on solver
- Option 3: infinitesimal steps
- Option 4: type and reject



Which programs make sense?

Given:

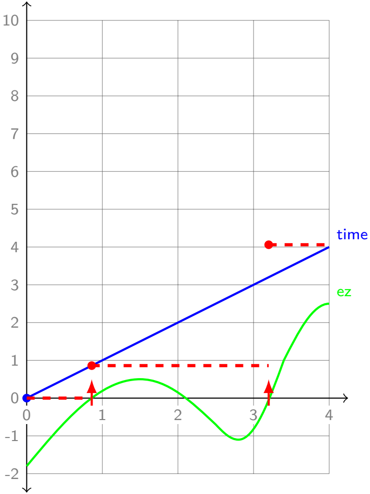
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

Evaluate:

```
der time = 1.0 init 0.0
and
y = sum(time) every up(ez) init 0.0
```

Interpretation:

- Option 1: $\mathbb{N} \subseteq \mathbb{R}$
- Option 2: depends on solver
- Option 3: infinitesimal steps
- Option 4: type and reject



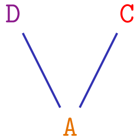
Explicitly relate simulation and logical time (using zero-crossings)

Try to minimize the effects of solver parameters and choices

Basic typing

The type language

$$\begin{array}{lcl}
bt & ::= & \mathtt{float} \mid \mathtt{int} \mid \mathtt{bool} \mid \mathtt{zero} \\
t & ::= & bt \mid t \times t \mid \beta \\
\sigma & ::= & \forall \beta_1, ..., \beta_n . t \xrightarrow{k} t \\
k & ::= & \mathtt{D} \mid \mathtt{C} \mid \mathtt{A}
\end{array}$$

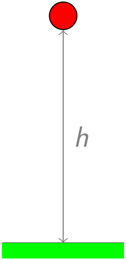

D C

A

Initial conditions

$$\begin{array}{lcl}
(+) & : & \mathtt{int} \times \mathtt{int} \xrightarrow{\mathtt{A}} \mathtt{int} \\
(=) & : & \forall \beta . \beta \times \beta \xrightarrow{\mathtt{A}} \mathtt{bool} \\
\mathtt{if} & : & \forall \beta . \mathtt{bool} \times \beta \times \beta \xrightarrow{\mathtt{A}} \beta \\
\cdot\, \mathtt{fby} \,\cdot & : & \forall \beta . \beta \times \beta \xrightarrow{\mathtt{D}} \beta \\
\mathtt{up}(\cdot) & : & \mathtt{float} \xrightarrow{\mathtt{C}} \mathtt{zero}
\end{array}$$

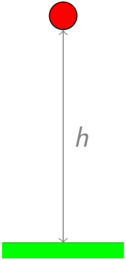
Compilation



```
let hybrid ball () =
  let
  rec der v = (-. g / m) init v0
              reset (-. 0.8 *. last v) every up(-. h)
  and der h =    v init h0
  in (v, h)
```

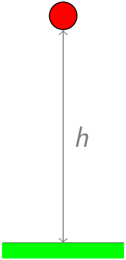
```
let node ball (z1, (lh, lv), ()) =
  let rec i = true fby false

  and dv = (-. g / m)
  and v = if i then v0
          else if z1 then -. 0.8 *. lv
          else lv

  and dh = v
  and h  = if i then h0 else lh

  and upz1 = -. h

  in ((v, h), upz1, (h, v), (dh, dv))
```

Compilation



```
let hybrid ball () =
  let
  rec der v = (-. g / m) init v0
                 reset (-. 0.8 *. last v) every up(-. h)
  and der h =       v init h0
  in (v, h)

let node ball (z1, (lh, lv), ()) =
  let rec i = true fby false

  and dv = (-. g / m)
  and v = if i then v0
          else if z1 then -. 0.8 *. lv
          else lv

  and dh = v
  and h  = if i then h0 else lh

  and upz1 = -. h

  in ((v, h), upz1, (h, v), (dh, dv))
```

Compilation

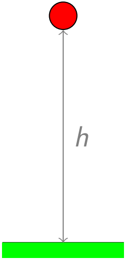


```
let hybrid ball () =
  let
  rec der v = (-. g / m) init v0
                   reset (-. 0.8 *. last v) every up(-. h)
  and der h =      v init h0
  in (v, h)
```

```
let node ball (z1, (lh, lv), ()) =
  let rec i = true fby false

  and dv = (-. g / m)
  and v = if i then v0
          else if z1 then -. 0.8 *. lv
          else lv

  and dh = v
  and h  = if i then h0 else lh

  and upz1 = -. h

  in ((v, h), upz1, (h, v), (dh, dv))
```

transform into discrete subset

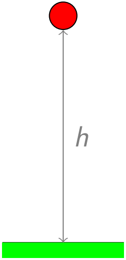
Compilation



```
let hybrid ball () =
  let
  rec der v = (−. g / m) init v0
                 reset (−. 0.8 *. last v) every up(−. h)
  and der h =       v init h0
  in (v, h)

let node ball (z1, (lh, lv), ()) =
  let rec i = true fby false

  and dv = (−. g / m)
  and v = if i then v0
          else if z1 then −. 0.8 *. lv
          else lv

  and dh = v
  and h  = if i then h0 else lh

  and upz1 = −. h

  in ((v, h), upz1, (h, v), (dh, dv))
```

transform continuous variables

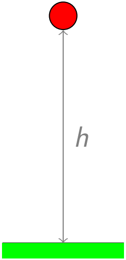
Compilation



```
let hybrid ball () =
  let
  rec der v = (-. g / m) init v0
                 reset (-. 0.8 *. last v) every up(-. h)
  and der h =     v init h0
  in (v, h)

let node ball (z1, (lh, lv), ()) =
  let rec i = true fby false

  and dv = (-. g / m)
  and v = if i then v0
          else if z1 then -. 0.8 *. lv
          else lv

  and dh = v
  and h  = if i then h0 else lh

  and upz1 = -. h

  in ((v, h), upz1, (h, v), (dh, dv))
```

transform zero-crossings

Compilation

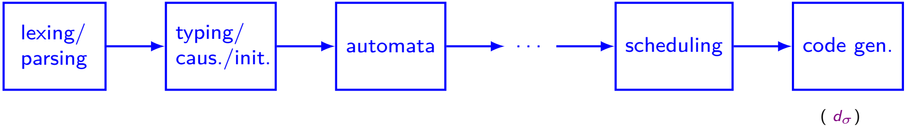


```
let hybrid ball () =
  let
  rec der v = (-. g / m) init v0
                reset (-. 0.8 *. last v) every up(-. h)
  and der h =     v init h0
  in (v, h)

let node ball (z1, (lh, lv), ()) =
  let rec i = true fby false

  and dv = (-. g / m)
  and v = if i then v0
          else if z1 then -. 0.8 *. lv
          else lv


  and dh
  and h


  and upz

  in ((v, h), upz1, (h, v), (dh, dv))
```

Careful mixing of discrete and continuous

- control discrete changes to respect invariant
- branching (i.e. automata) is tricky

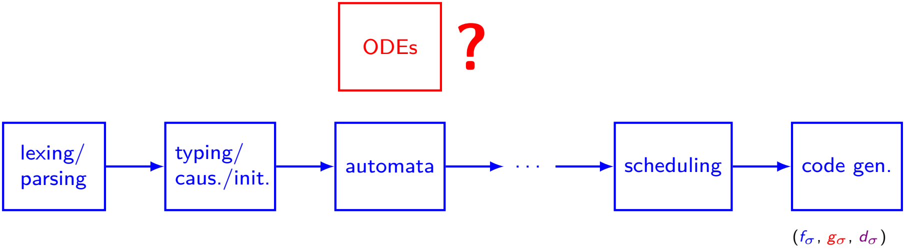
Source-to-source transformation



lexing/ parsing → typing/ caus./init. → automata → · · · → scheduling → code gen.

(d_σ)

Source-to-source transformation



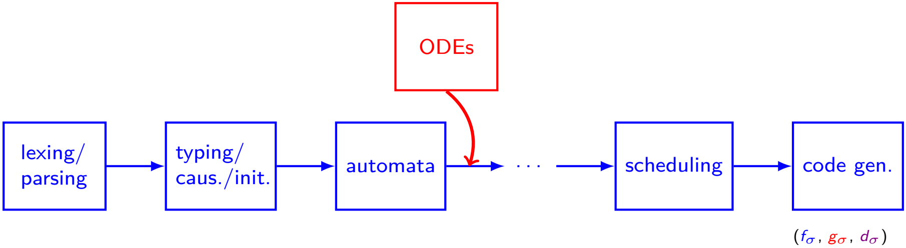
ODEs ?

lexing/ parsing → typing/ caus./init. → automata → ··· → scheduling → code gen.

$(f_\sigma, g_\sigma, d_\sigma)$

$$\begin{tikzcd}
\text{Data-flow} + \text{Auto.} + \text{ODE} \arrow[r, "ode"] \arrow[d, "auto"'] & \text{Data-flow} + \text{Auto.} \arrow[d, "auto"] & \\
\text{Data-flow} + \text{ODE} \arrow[r, "ode"] & \text{Data-flow} \arrow[r, "codegen"] & \text{Imperative code}
\end{tikzcd}$$

Source-to-source transformation



ODEs

lexing/ parsing

typing/ caus./init.

automata

$\cdots$

scheduling

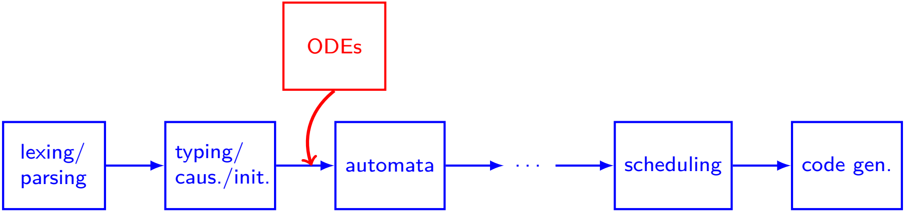
code gen.

$(f_\sigma, g_\sigma, d_\sigma)$

$$\begin{tikzcd}
\text{Data-flow} + \text{Auto.} + \text{ODE} \arrow[r, "\textit{ode}"] \arrow[d, "\textit{auto}"'] & \text{Data-flow} + \text{Auto.} \arrow[d, "\textit{auto}"] & \\
\text{Data-flow} + \text{ODE} \arrow[r, "\textit{ode}"] & \text{Data-flow} \arrow[r, "\textit{codegen}"] & \text{Imperative code}
\end{tikzcd}$$

- Pro: simpler definition of ODE
- Con: subtle invariant over intermediate language

Source-to-source transformation


ODEs

lexing/ parsing

typing/ caus./init.

automata

· · ·

scheduling

code gen.

$(f_\sigma, g_\sigma, d_\sigma)$

$$\begin{tikzcd}
\text{Data-flow} + \text{Auto.} + \text{ODE} \arrow[r, "ode"] \arrow[d, "auto"'] & \text{Data-flow} + \text{Auto.} \arrow[d, "auto"] & \\
\text{Data-flow} + \text{ODE} \arrow[r, "ode"] & \text{Data-flow} \arrow[r, "codegen"] & \text{Imperative code}
\end{tikzcd}$$

- Pro: intermediate result is well-typed
- Pro/Con: ODE code must include cases for automata

Outline

Dataflow programming

Research objectives

Continuous modelling and simulation

Typing and compilation

Demonstration and conclusion

Demonstrations

- Bouncing ball (standard)
- Bang-bang temperature controller (Simulink/Stateflow)
- Sticky Masses (Ptolemy)

Conclusion

Conclusion

- Synchronous languages should and can properly treat hybrid systems
- There are three good reasons for doing so:
  1. To exploit existing compilers and techniques
  2. For programming the discrete subcomponents
  3. To clarify underlying principles and guide language design/semantics
- Our approach
  - Hybrid dataflow language with hierarchical automata
  - System of kinds for rejecting unreasonable programs
  - Relate discrete to continuous via zero-crossings
  - Compilation via source-to-source transformations
  - Simulation using off-the-shelf numerical solvers
- Prototype compiler in OCaml using Sundials CVODE solver

Conclusion

- Synchronous languages should and can properly treat hybrid systems
- There are three good reasons for doing so:
  1. To exploit existing compilers and techniques
  2. For programming the discrete subcomponents
  3. To clarify underlying principles and guide language design/semantics
- Our approach
  - Hybrid dataflow language with hierarchical automata
  - System of kinds for rejecting unreasonable programs
  - Relate discrete to continuous via zero-crossings
  - Compilation via source-to-source transformations
  - Simulation using off-the-shelf numerical solvers
- Prototype compiler in OCaml using Sundials CVODE solver

Conclusion

Conclusion

- Synchronous languages should and can properly treat hybrid systems
- There are three good reasons for doing so:
  1. To exploit existing compilers and techniques
  2. For programming the discrete subcomponents
  3. To clarify underlying principles and guide language design/semantics
- Our approach
  - Hybrid dataflow language with hierarchical automata
  - System of kinds for rejecting unreasonable programs
  - Relate discrete to continuous via zero-crossings
  - Compilation via source-to-source transformations
  - Simulation using off-the-shelf numerical solvers
- Prototype compiler in OCaml using Sundials CVODE solver

Conclusion

- Synchronous languages should and can properly treat hybrid systems
- There are three good reasons for doing so:
  1. To exploit existing compilers and techniques
  2. For programming the discrete subcomponents
  3. To clarify underlying principles and guide language design/semantics
- Our approach
  - Hybrid dataflow language with hierarchical automata
  - System of kinds for rejecting unreasonable programs
  - Relate discrete to continuous via zero-crossings
  - Compilation via source-to-source transformations
  - Simulation using off-the-shelf numerical solvers
- Prototype compiler in OCaml using Sundials CVODE solver