Synchronous Retrogames in HTML5

Alexandre Doussot

Université Pierre et Marie Curie

Introduction

Context

- Game programming
- Speed is not that important
- Reactive limitations by the canvas API
- Hence control-command system
- Synchronous Data Flow language

The language

DataFlow

$$(1 + x) * y + (1 + x) * z$$

```
1 --->-----+           +---<--- y
           |           |
x ---->---[+]---->----[*]---->----[+]---->--
           |                       |
           |                       |
           +----->----[*]---->-----+
                       |
         z ------->----+
```

SAP node - (1 + x) * y + (1 + x) * z

```
node calc(x : int, y : int, z : int) -> (d : int)
    with
  a = plus(1, x);
  b = times(a, y);
  c = times(a, z);
  d = plus(b, c)
```

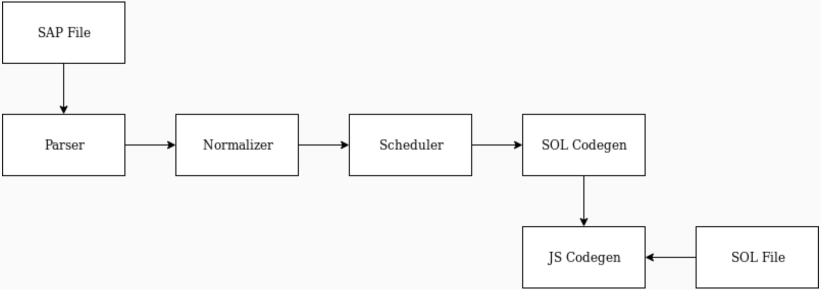
Each equation describes its own stream.

```
node clk() -> (a: int) with
  half = True fby not(half) :: base;
  y = 3 when True(half) :: base on True(half);
  x = 2 when False(half) :: base on False(half);
  a = merge half (True -> y) (False -> x) :: base
```

```
type action = Add(n: int, x: int) | Id(n: int)

interface node test(a : action) -> (x: int) with
  x = 0 fby y;
  y = merge a (Add -> add(a.n, a.x)) (Id -> a.n)
```

The compiler

- Multi-pass compiler
- Built with OCaml and Menhir



Normalization

1. Demux equations
2. Extract stateful computations

Normalization - Demux

```
node complex_demux(a : int) ->
     (x : int) with
  (a, (b, c)) = (2, (3, 4));
  (x, y) = @dup(a, b);
  f = True;
  (d, e) = merge f
    (True -> (2, 3))
    (False -> (4, 5))
```

```
node complex_demux(a : int) ->
     (x : int) with
  a = 2;
  b = 3;
  c = 4;
  (x, y) = @dup(a, b);
  f = True;
  d = merge f (True -> 2)
              (False -> 4);
  e = merge f (True -> 3)
              (False -> 5)
```

Normalization - Extraction

```
node id(x: int) -> (x : int)
    with

node simple_fby(a : int) -> (x
    : int) with
  y = 1 fby 2;
  x = @id(2 fby 3)
```

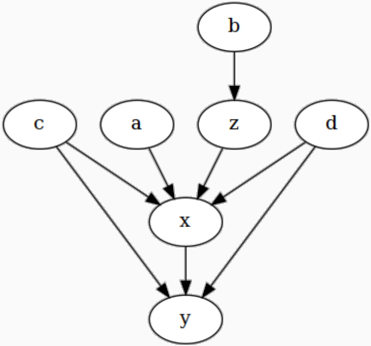
```
node id(x : int) -> (x : int)
    with

node simple_fby(a : int) -> (x
    : int) with
  t2 = @id(t3);
  t3 = 2 fby 3;
  t1 = 1 fby 2;
  y = t1;
  x = t2
```

Scheduling

1. Check each equation for dependecies
2. Build a dependency graph
3. Reverse `fby` edges
4. Schedule node

Scheduling

```
node example() -> (x: int)
    with
  a = @node_call(x);
  z = 3 fby plus(b, 1);
  b = 2 fby 1;
  x = plus(z, y);
  (c, d) = (x, y);
  y = 2
```



Scheduling

```
node example() -> (x: int)
    with
  a = @node_call(x);
  z = 3 fby plus(b, 1);
  b = 2 fby 1;
  x = plus(z, y);
  (c, d) = (x, y);
  y = 2
```

```
node example() -> (x: int)
    with
  y = 2;
  x = 3 fby plus(b, 2)
  (c, d) = (x, y)
  z = 3 fby plus(b, 1);
  b = 2 fby 1;
  a = @node_call(x)
```

Intermediate language

Intermediate language needs:

- Imperative
- State and state modifiers
- → OOP

```
machine example =
  memory /* Instance variables */
  instances /* Node instances */
  reset () = skip
  step() returns () =
    /* Instructions */
```

```
node example(x: int) -> (y:
    int) with
  a = True fby not(a);
  b = @id(a);
  c = 1 when True(a) :: base
      on True(a);
  y = plus(3, 2)
```

```
machine example =
  memory t1 : undefined
  instances t3 : id
  reset () =
    t3.reset();
  state(t1) = True
  step(x : int) returns (y :
       int) =
    var a : undefined, c :
        undefined, y :
        undefined, t2 :
        undefined, b :
        undefined in
    a = state(t1);
    case (a) {
      True: c = 1
    };
```

Javascript Code Generation - small node

```
node small(a: int) -> (x: int)
     with
  b = @node_call(a);
  x = 0 fby b
```

```
machine small =
  memory t2 : undefined
  instances t3 : node_call
  reset () =
    t3.reset();
  state(t2) = 0
  step(a : int) returns (x :
       int) =
    var t1 : undefined, x :
          undefined, b :
          undefined in
    t1 = t3.step(a);
    x = state(t2);
    b = t1;
    state(t2) = b
```

Javascript Code Generation - small node

```
machine small =
  memory t2 : undefined
  instances t3 : node_call
  reset () =
    t3.reset();
  state(t2) = 0
  step(a : int) returns (x :
      int) =
    var t1 : undefined, x :
        undefined, b :
        undefined in
    t1 = t3.step(a);
    x = state(t2);
    b = t1;
    state(t2) = b
```

```
function small() {
  this.t2 = undefined;
  this.t3 = new node_call();
}
small.prototype.reset =
    function() {
  this.t3.reset();
  this.t2 = 0;
}
small.prototype.step =
    function(a) {
  /* Omitting vardecs */
  t1 = this.t3.step(a);
  x = this.t2;
  b = t1;
  this.t2 = b;
  return x;
}
```

Types

```
type action = Add(n: int, x:
    int) | Id(n: int)

interface node test(a :
    action) -> (x: int) with
  x = 0 fby y;
  y = merge a (Add -> add(a.n,
      a.x)) (Id -> a.n)
```

```
var action_enum = Object.
    freeze({
  Add: 1,
  Id: 2
});

function action_type() {}

action_type.Add = function(n,
     x) {
  return {id: action_enum.Add
      , n:n, x:x}
}

action_type.Id = function() {
  return {id: action_enum.Id,
       n:n}
}
```

Interfacing with engine code

```
type action = Add(n: int, x:
    int) | Id(n: int)

interface node test(a :
    action) -> (x: int) with
  x = 0 fby y;
  y = merge a (Add -> add(a.n,
       a.x)) (Id -> a.n)
```

```
test.prototype.add = function
     (n, x) {
   this.step(action_type.Add(
       n, x));
   return this;
 }

test.prototype.nothing =
    function () {
  this.step(action_type.
      Nothing());
  return this;
}
```

```
var node = new test().reset();
var result = node.add(2, 3).get_x();
assert(result == 5);
var result = node.id(5).get_x();
assert(result == 5);
```

Conclusion

Past & Future Work

Accomplished work

- Working SAP compiler
  - [Parser|Normalizer|Scheduler|SOL Codegen|JS Codegen]
- Runtime Javascript Library
- Snake clone

Future work

- Type inference & Checking
- Clock inference & Checking
- Automata
- Optimisation

Questions?