Oversampling in a Dataflow Synchronous Language (Heptagon)

Léonard Gérard

1PARKAS team
ENS

Synchron’11
Heptagon

A small Scade v6
- Automaton
- Arrays and iterators
- Modular reset
- Static parameters

Novelties
- Memory optimization for arrays
- Controller synthesis
- and WIP
  - asynchronous computations
  - oversampling
  - lucy-n generation
  - ...

Soon to be released as open source...
Classic oversampling example

node f(x : int) returns (cpt, y : int)
let
  y = x + 1
  cpt = (0 fby cpt) + 1
tel

dnode g(x : int; c : bool) returns (out : int)
var t, cpt, y, last_y : int;
let
  (cpt, y) = f(t);
  t = merge c x (last_y whenot c);
  last_y = 0 fby y;
  out = y whenot c;
tel

node main() returns (out : int; c : bool) var x : int;
let
  x = 0 fby (x+10);
  c = true fby false fby c;
  out = g(x,c);
Classic oversampling example

```plaintext
node g(x : int ; c : bool ) returns ( out : int )
var t, cpt , y, last_y : int ;
let
  (cpt , y) = f(t);
  t = merge c x ( last_y whenot c);
  last_y = 0 fby y;
  out = y whenot c;

tel
val g:: (. on c, c : .) -> . on not c
```

Oversampling with clock given as argument.

| c  | true | false | true | false | true | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x_0</td>
<td>x_1</td>
<td>x_2</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
| t  | x_0  | f(x_0)| x_1  | f(x_1)| x_2  | ...
| y  | f(x_0)| f^2(x_0)| f(x_1)| f^2(x_1)| f(x_2)| ...
| cpt| 1    | 2     | 3    | 4     | 5    | ...
| out| f^2(x_0)| f^2(x_1)| ... |
Why hiding the oversampling clock?

- It is strange to define the clock outside of $g$.
- The node $g$ communicate at each of its steps, even if no value for $x$ and $out$ is meaningful.
- From the outside, the clocks of $x$ and $out$ are needlessly complex.

We would like

```
val g :: . -> .
```

<table>
<thead>
<tr>
<th>$x$</th>
<th>$x_0$</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>$t$</td>
<td>$x_0 f(x_0)$</td>
<td>$x_1 f(x_1)$</td>
<td>$x_2 f(x_2)$</td>
<td>...</td>
</tr>
<tr>
<td>$cpt$</td>
<td>1 2</td>
<td>3 4</td>
<td>5 6</td>
<td>...</td>
</tr>
<tr>
<td>$y$</td>
<td>$f(x_0)$</td>
<td>$f^2(x_0)$</td>
<td>$f(x_1)$</td>
<td>$f^2(x_1)$</td>
</tr>
<tr>
<td>$out$</td>
<td>$f^2(x_0)$</td>
<td>$f^2(x_1)$</td>
<td>$f^2(x_2)$</td>
<td>...</td>
</tr>
</tbody>
</table>
Local Hiding of Oversampling in Heptagon

Any node which would be given the *usually illegal* signature

\[
\text{val } n :: . \text{ on } c \rightarrow . \text{ on } c
\]

is transformed into a node with signature

\[
\text{val } n :: . \rightarrow .
\]

with a simple transformation in the generated sequential code:

\[
\text{step}_n(x) \{ \\
\text{[vars}_n] \\
\text{do} \{ \\
\text{[code}_n] \\
\text{return } y; \\
\} \}
\]

\[
\Rightarrow \\
\text{step}_n(x) \{ \\
\text{[vars}_n] \\
\text{do} \{ \\
\text{[code}_n] \\
\text{while } (!c); \\
\text{return } y; \\
\} \\
\} 
\]
Local Hiding of Oversampling in Heptagon (bis)

val n:: (c : . on e on d, . on e on d on c) -> . on e on d on c

is transformed into a node with signature

val n:: (c : . , . on c) -> . on c

step_n (c,x) {
  [vars_n]
  [code_n]
  return y;
}

⇒

step_n (x) {
  [vars_n]
  do {
    [code_n]
    while (!(d && e));
    return y;
  }
}

PS: The common root of the clocks of the signature is the local oversampling. Here . on e on d.
We are asked to give the same sampling to the input and the output. So naively we do so.
First attempt to use LHO, LHO done

node g(x : int) returns (out : int)
var c : bool; t, cpt, y, last_y : int;
let
    c = true fby false fby c;
    (cpt, y) = f(t);
    t = merge c x (last_y whenot c);
    last_y = 0 fby y;
out = y when c;
tel
val g:: . -> .

- The square brackets are used to display the oversampling: from the outside of the node, the signature hides the inner steps of these brackets.
- Nothing new, to be able to do oversampling, we need to lose one instant. See the Lucid V3 manual page 24.
Correct use of LHO

node g(x : int) returns (out : int)
var c : bool; t, cpt, y, last_y : int;
let
  c = true fby false fby c;
  (cpt, y) = f(t);
  t = merge c x (last_y whenot c);
  last_y = 0 fby y;
  out = last_y when c;

tel
val g:: . -> .
Correct use of LHO (bis)

node g(x : int) returns (out : int)
var c : bool; t, cpt, y, last_y : int;
let
  c = true fby false fby false fby c;
  (cpt, y) = f(t);
  t = merge c x (last_y whenot c);
  last_y = 0 fby y;
  out = last_y when c;
labeled
val g:: . -> .

It is now easy to do any number of oversampling steps.
But can’t we do it without delay!?  

```plaintext
def g(x : int) returns (out : int):
  var t, cpt, y, last_y : int; c : bool;
  let
      c = true fby false fby c;
      (cpt, y) = f(t);
      t = merge c x (last_y whenot c);
      last_y = 0 fby y;
  in
      out = y whenot c;
  in
  val g:: . on c -> . on not c
```

Even if this seems to generate correct code with the LHO transformation, the compiler rejects this program... It is not able to recognize the interleaving of the clock.
No, we cannot generalize LHO

```plaintext
node g(x : int) returns (out : int)
var t, cpt, y, last_y : int; c : bool;
let
    c = true fby false fby false fby c;
    (cpt, y) = f(t);
    t = merge c x (last_y whenot c);
    last_y = 0 fby y;
    out = y whenot c;
end
```

There are two outputs for one input...

<table>
<thead>
<tr>
<th>c</th>
<th>true</th>
<th>false</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x₀</td>
<td>x₁</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t</th>
<th>x₀</th>
<th>x₁</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>f(x₀)</td>
<td>f²(x₀)</td>
<td>f(x₁)</td>
<td>f²(x₁)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>y</th>
<th>f(x₀)</th>
<th>f²(x₀)</th>
<th>f³(x₀)</th>
</tr>
</thead>
<tbody>
<tr>
<td>f(x₁)</td>
<td>f²(x₁)</td>
<td>f³(x₁)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cpt</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>out</th>
<th>f²(x₀)</th>
<th>f³(x₀)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>f²(x₁)</td>
<td>f³(x₁)</td>
</tr>
</tbody>
</table>

Oversampling in a Dataflow Synchronous Language (Heptagon)

The compiler rejects this program rightfully.
type t = In | C | Out

node g(x : int) returns (out : int)

var t, cpt, y, last_y : int; c : t;

let
    c = In fby C fby C fby Out fby c;
    (cpt, y) = f(t);
    t = merge c (In -> x) (C -> last_y when C(c))
    (Out -> last_y when Out(c));

    last_y = 0 fby y;
    out = y when Out(c);

tel

val g :: . on In(c) -> . on Out(c)

---

Oversampling in a Dataflow Synchronous Language (Heptagon)

Enumerated clocks are equivalent, but insightful

The compiler rejects this program...
No, we still cannot generalize LHO

The compiler reject this program rightfully.
No, we still cannot generalize LHO (ter)

type t = In | C | Out

node g(x : int) returns (out : int)

var t, cpt, y, last_y : int; c : t;

let

  c = In fby C fby C fby C fby c;
  (cpt, y) = f(t);
  t = merge c (In -> x) (C -> last_y when C(c))
      (Out -> last_y when Out(c));

  last_y = 0 fby y;
  out = y when Out(c);

tel

val g:: . on In(c) -> . on Out(c)

The compiler reject this program rightfully.
Bursts must be well formed

Observed instant
An observed instant of a burst is an instant accessed, from someone which is observing the burst as one instant.
—In and Out are observed, C isn’t.

Sufficient and necessary condition to apply LHO
During one burst, every observed instant must appear one and only one time.

Burst boundaries
- The boundaries of bursts are constrained by the causality.
- In the case of causal functions with outputs depending on all inputs, the end of the burst is aligned with the last output.
Note that on \( c \equiv \) on true(\( c \)) and on not \( c \equiv \) on false(\( c \))

- on \( C(b) \rightarrow . on C(b) \): accepted by LHO
  - Reactivity requires \( C(b) \) to be true an infinite amount of time.
  - Should/how can we ensure it?
  - Right now our prototype in Heptagon doesn’t check it.
- on \( C(b) \rightarrow . on C2(b) \): rejected by LHO
  - Only the perfect interleaving of the two constructors is possible.
Proposal and questions

Iterator primitive:

- Static: \( b = \text{iter} \ [\text{In}; C; C; \text{Out}] \)
- Dynamic: \( b = \text{iter} \ \text{list} \)

- How much do we need dynamic iteration?
- What should be dynamic (in increasing difficulty)
  - the size?
  - the order?
  - the type?

Use a restricted