Oversampling in a Dataflow Synchronous Language (Heptagon)

Léonard Gérard[1]

[1]PARKAS team
ENS

Synchron'11

2011-12-14

Oversampling in a Dataflow Synchronous Language (Heptagon)

Oversampling in a Dataflow Synchronous Language (Heptagon)

Léonard Gérard[1]

[1]PARKAS 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...

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
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
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);
tel
```


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

Classic oversampling example

```
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
```

c	*true*	*false*	*true*	*false*	*true*	...
x	x_0		x_1		x_2	...
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		$f^2(x_0)$		$f^2(x_1)$		...

Classic oversampling example

Oversampling with clock given as argument.

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:: . -> .
```

x	x_0	x_1	x_2	$\dots$
c	$[true \quad false]$	$[true \quad false]$	$[true \quad false]$	$\dots$
t	$[x_0 \quad f(x_0)]$	$[x_1 \quad f(x_1)]$	$[x_2 \quad f(x_2)]$	$\dots$
cpt	$[1 \quad 2]$	$[3 \quad 4]$	$[5 \quad 6]$	$\dots$
y	$[f(x_0) \quad f^2(x_0)]$	$[f(x_1) \quad f^2(x_1)]$	$[f(x_2) \quad f(f(x_2))]$	$\dots$
out	$f^2(x_0)$	$f^2(x_1)$	$f^2(x_2)$	$\dots$

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:: . -> .

Local Hiding of Oversampling in Heptagon

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

```
val n:: . on c -> . on c
```

is transformed into a node with signature

```
val n:: . -> .
```

with a simple transformation in the generated sequential code:

```
step_n(x) {                      step_n(x) {
  [vars_n]                         [vars_n]
  [code_n]           ⟹            do {
  return y;                          [code_n]
}                                  } while (!c);
                                   return y;
                                 }
```


Local Hiding of Oversampling in Heptagon
Any node which would be given the usually illegal signature
val n:: . on c -> . on c
is transformed into a node with signature
val n:: . -> .
with a simple transformation in the generated sequential code:

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) {                    step_n(x) {
  [vars_n]                         [vars_n]
  [code_n]              ⟹          do {
  return y;                          [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`.

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.

First attempt to use LHO, before LHO transformation

```
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:: . on c -> . on c
```

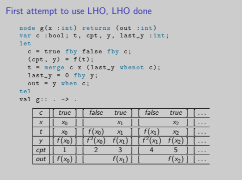
c	*true*	*false*	*true*	*false*	*true*	...
x	x_0		x_1		x_2	...
t	x_0	$f(x_0)$	x_1	$f(x_1)$	x_2	...
y	$f(x_0)$	$f^2(x_0)$	$f(x_1)$	$f^2(x_1)$	$f(x_2)$	...
cpt	1	2	3	4	5	...
out	$f(x_0)$		$f(x_1)$		$f(x_2)$	...

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:: . -> .
```

c	[*true*]	[*false*	*true*]	[*false*	*true*]	[...
x	[x_0]	[	x_1]	[	x_2]	[...
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	[$f(x_0)$]	[	$f(x_1)$]	[	$f(x_2)$]	[...



- The square brackets are used to display the oversampling : from the outside of the node, the signature hide the inner steps of these brackets.
- Nothing new, to be able to do oversampling, we need to loose 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:: . -> .
```

c	[*true*]	[*false*	*true*]	[*false*	*true*]	[…
x	[x_0]	[	x_1]	[	x_2]	[…
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	[0]	[	$f^2(x_0)$]	[	$f^2(x_1)$]	[…

Correct use of LHO

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;
tel
val g:: . -> .
```

c	[*true*]	[*false*	*false*	*true*]	[*false*	*false*	*true*]	...
x	[x_0]	[		x_1]	[		x_2]	...
t	[x_0]	[$f(x_0)$	$f^2(x_0)$	x_1]	[$f(x_1)$	$f^2(x_1)$	$f^3(x_1)\ldots$]	
y	[$f(x_0)$]	[$f^2(x_0)$	$f^3(x_0)$	$f(x_1)$]	[$f^2(x_1)$	$f^3(x_1)$	$f(x_2)$]	...
cpt	[1]	[2	3	4]	[5	6	7]	...
out	[0]	[		$f^3(x_0)$]	[		$f^3(x_1)$]	...

It is now easy to do any number of oversampling steps.

But can't we do it without delay!?

```
node 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;
  out = y whenot c;
tel

val g:: . on c -> . on not c
```

c	[*true* *false*]	[*true* *false*]	[*true* *false*] ...
x	[x_0]	[x_1]	[x_2] ...
t	[x_0 $f(x_0)$]	[x_1 $f(x_1)$]	[x_2 $f(x_2)$] ...
y	[$f(x_0)$ $f^2(x_0)$]	[$f(x_1)$ $f^2(x_1)$]	[$f(x_2)$ $f(f(x_2))$] ...
cpt	[1 2]	[3 4]	[5 6] ...
out	[$f^2(x_0)$]	[$f^2(x_1)$]	[$f(f(x_2))$] ...

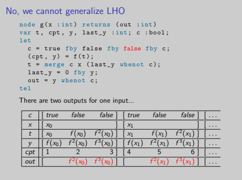
Even if this seems to generate correct code with the LHO transformation, the compiler rejects this program... It is not able to recognize the interleaving of the clock.

No, we cannot generalize LHO

```
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;
tel
```

There are two outputs for one input...

c	[*true*	*false*	*false*]	[*true*	*false*	*false*]	[...
x	[x_0		]	[x_1		]	[...
t	[x_0	$f(x_0)$	$f^2(x_0)$]	[x_1	$f(x_1)$	$f^2(x_1)$]	[...
y	[$f(x_0)$	$f^2(x_0)$	$f^3(x_0)$]	[$f(x_1)$	$f^2(x_1)$	$f^3(x_1)$]	[...
cpt	[1	2	3]	[4	5	6]	[...
out	[	$f^2(x_0)$	$f^3(x_0)$]	[	$f^2(x_1)$	$f^3(x_1)$]	[...



The compiler rejects this program rightfully.

Enumerated clocks are equivalent, but insightful

```
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)
```

c	[In	C	C	Out]	[In	C	...
x	[x_0			]	[x_1		...
y	[$f(x_0)$	$f^2(x_0)$	$f^3(x_0)$	$f^4(x_0)$]	[$f(x_1)$	$f^2(x_1)$	...
cpt	[1	2	3	4]	[5	6	...
out	[			$f^4(x_0)$]	[		...

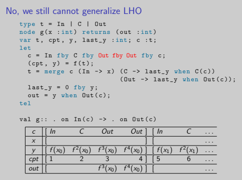
The compiler rejects this program...

No, we still cannot generalize LHO

```
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 Out 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)
```

c	[In	C	Out	Out]	[In	C	...
x	[			]	[		...
y	[$f(x_0)$	$f^2(x_0)$	$f^3(x_0)$	$f^4(x_0)$]	[$f(x_1)$	$f^2(x_1)$	...
cpt	[1	2	3	4]	[5	6	...
out	[		$f^3(x_0)$	$f^4(x_0)$]	[		...



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)
```

c	[In	C	C	C]	[In	C		...
x	[x_0			]	[x_1			...
y	[$f(x_0)$	$f^2(x_0)$	$f^3(x_0)$	$f^4(x_0)$]	[$f(x_1)$	$f^2(x_1)$		...
cpt	[1	2	3	4]	[5	6		...
out	[			]	[			...

No, we still cannot generalize LHO (ter)

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.

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.

Back to Heptagon:

Note that `on c` $\equiv$ `on true(c)` and `on not c` $\equiv$ `on false(c)`

- `. on C(b) -> . 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) -> . on C2(b)`: rejected by LHO
  - Only the perfect interleaving of the two constructors is possible.

Back to Heptagon:

Note that on c ≡ on true(c) and on not c ≡ on false(c)

- . on C(b) -> . 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) -> . on C2(b): rejected by LHO
  - Only the perfect interleaving of the two constructors is possible.

Proposal and questions

Iterator primitive:

- Static: `b = iter [In; C; C; Out]`
- Dynamic: `b = iter list`

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

U

se a restricted

Proposal and questions
Iterator primitive:
- Static: b = iter [In; C; C; Out]
- Dynamic: b = iter list
- How much do we need dynamic iteration?
- What should be dynamic (in increasing difficulty)
  - the size?
  - the order?
  - the type?
U
se a restricted