Programming Parallelism with Futures in Lustre

Albert Cohen and Léonard Gérard and Marc Pouzet

PARKAS Team
ENS Paris, France

Synchron 2012 (EMSOFT 2012)

Introduction

Application domain

- Embedded control system programming
- Block-diagram languages: StateCharts, Simulink, Scade ...
- More precisely synchronous languages: Esterel, Signal, Lustre

Introduction

Application domain

- Embedded control system programming
- Block-diagram languages: StateCharts, Simulink, Scade ...
- More precisely synchronous languages: Esterel, Signal, Lustre

Goal

- Generate efficient *parallel* code
- from *explicit* source annotations
- without changing
  - the *semantics* of the program
  - usual properties like *static and bounded memory*
- nor changing the existing sequential compilation.

Introduction, Heptagon a LUSTRE/SCADE-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  y = x + m;
  m = 0 fby y;
tel
```

Heptagon in short:

- Functional synchronous
- Declarative data-flow

Introduction, Heptagon a Lustre/Scade-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- Functional synchronous
- Declarative data-flow

Introduction, Heptagon a Lustre/Scade-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- Functional synchronous
- Declarative data-flow
- Values are streams
- Types and operators are lifted pointwise
- The synchronous register `fby`

m	0
x	
y	

Introduction, Heptagon a LUSTRE/SCADE-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- Functional synchronous
- Declarative data-flow
- Values are streams
- Types and operators are lifted pointwise
- The synchronous register `fby`

m	0
x	0
y	

Introduction, Heptagon a Lustre/Scade-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- Functional synchronous
- Declarative data-flow
- Values are streams
- Types and operators are lifted pointwise
- The synchronous register `fby`

m	0
x	0
y	0

Introduction, Heptagon a Lustre/Scade-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- Functional synchronous
- Declarative data-flow
- Values are streams
- Types and operators are lifted pointwise
- The synchronous register `fby`

m	0	0
x	0	
y	0	

Introduction, Heptagon a Lustre/Scade-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- Functional synchronous
- Declarative data-flow
- Values are streams
- Types and operators are lifted pointwise
- The synchronous register `fby`

m	0	0
x	0	1
y	0	

Introduction, Heptagon a Lustre/Scade-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- Functional synchronous
- Declarative data-flow
- Values are streams
- Types and operators are lifted pointwise
- The synchronous register `fby`

m	0	0
x	0	1
y	0	1

Introduction, Heptagon a Lustre/Scade-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- Functional synchronous
- Declarative data-flow
- Values are streams
- Types and operators are lifted pointwise
- The synchronous register `fby`

m	0	0	1
x	0	1	0
y	0	1	

Introduction, Heptagon a Lustre/Scade-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- Functional synchronous
- Declarative data-flow
- Values are streams
- Types and operators are lifted pointwise
- The synchronous register `fby`

m	0	0	1
x	0	1	0
y	0	1	1

Introduction, Heptagon a Lustre/Scade-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- Functional synchronous
- Declarative data-flow
- Values are streams
- Types and operators are lifted pointwise
- The synchronous register `fby`

m	0	0	1	1
x	0	1	0	2
y	0	1	1	

Introduction, Heptagon a Lustre/Scade-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- Functional synchronous
- Declarative data-flow
- Values are streams
- Types and operators are lifted pointwise
- The synchronous register `fby`

m	0	0	1	1
x	0	1	0	2
y	0	1	1	3

Introduction, Heptagon a LUSTRE/SCADE-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Translate into JAVA syntax:

```
class Sum {
  int m;
  void reset(){ m = 0; }
  int step(int x){
    int y;
    y = x + m;
    m = y;
    return y;
  }
}
```

- Modular compilation, each node is compiled into a class.

Introduction, Heptagon a LUSTRE/SCADE-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Translate into JAVA syntax:

```
class Sum {
  int m;
  void reset(){ m = 0; }
  int step(int x){
    int y;
    y = x + m;
    m = y;
    return y;
  }
}
```

- Modular compilation, each node is compiled into a class.
- Synchronous registers are instance variables.

Introduction, Heptagon a LUSTRE/SCADE-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Translate into JAVA syntax:

```
class Sum {
  int m;
  void reset(){ m = 0; }
  int step(int x){
    int y;
    y = x + m;
    m = y;
    return y;
  }
}
```

- Modular compilation, each node is compiled into a class.
- Synchronous registers are instance variables.
- Initialisation (and reinitialisation) method.

Introduction, Heptagon a Lustre/Scade-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Translate into Java syntax:

```
class Sum {
  int m;
  void reset(){ m = 0; }
  int step(int x){
    int y;
    y = x + m;
    m = y;
    return y;
  }
}
```

- Modular compilation, each node is compiled into a class.
- Synchronous registers are instance variables.
- Initialisation (and reinitialisation) method.
- Step method, with in place update of the state.

Introduction, Heptagon a Lustre/Scade-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Translate into Java syntax:

```
class Sum {
  int m;
  void reset(){ m = 0; }
  int step(int x){
    int y;
    y = x + m;
    m = y;
    return y;
  }
}
```

- Modular compilation, each node is compiled into a class.
- Synchronous registers are instance variables.
- Initialisation (and reinitialisation) method.
- Step method, with in place update of the state.

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- `when`: the sampling operator

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- `when`: the sampling operator
- `merge`: the (lazy) complementing operator

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- `when`: the sampling operator
- `merge`: the (lazy) complementing operator

`x`	0
`big = period3()`	*true*
`xt = x when big`	
`xf = x whenot big`	
`y = merge big xt xf`	

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- `when`: the sampling operator
- `merge`: the (lazy) complementing operator

`x`	0
`big = period3()`	*true*
`xt = x when big`	0
`xf = x whenot big`	.
`y = merge big xt xf`	

- `whenot` = `when not`
- (.) = absence of value

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- `when`: the sampling operator
- `merge`: the (lazy) complementing operator

`x`	0
`big = period3()`	*true*
`xt = x when big`	0
`xf = x whenot big`	.
`y = merge big xt xf`	0

- `whenot` = `when not`
- (.) = absence of value

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- `when`: the sampling operator
- `merge`: the (lazy) complementing operator

`x`	0	1
`big = period3()`	*true*	*false*
`xt = x when big`	0	
`xf = x whenot big`	.	
`y = merge big xt xf`	0	

- `whenot` = `when not`
- (.) = absence of value

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- `when`: the sampling operator
- `merge`: the (lazy) complementing operator

`x`	0	1
`big = period3()`	*true*	*false*
`xt = x when big`	0	.
`xf = x whenot big`	.	1
`y = merge big xt xf`	0	

- `whenot` = `when not`
- (.) = absence of value

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- `when`: the sampling operator
- `merge`: the (lazy) complementing operator

`x`	0	1
`big = period3()`	*true*	*false*
`xt = x when big`	0	.
`xf = x whenot big`	.	1
`y = merge big xt xf`	0	1

- `whenot` = `when not`
- (.) = absence of value
- `merge` is *lazy*, its inputs have to arrive only when needed.

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- `when`: the sampling operator
- `merge`: the (lazy) complementing operator

x	0	1	2
`big = period3()`	*true*	*false*	*false*
`xt = x when big`	0	.	.
`xf = x whenot big`	.	1	2
`y = merge big xt xf`	0	1	2

- `whenot` = `when not`
- (.) = absence of value
- `merge` is *lazy*, its inputs have to arrive only when needed.

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- `when`: the sampling operator
- `merge`: the (lazy) complementing operator

x	0	1	2	3	4	...
big = period3()	*true*	*false*	*false*	*true*	*false*	...
xt = x when big	0	.	.	3	.	...
xf = x whenot big	.	1	2	.	4	...
y = merge big xt xf	0	1	2	3	4	...

- `whenot` = `when` not
- (.) = absence of value
- `merge` is *lazy*, its inputs have to arrive only when needed.
- The compiler computes correct rhythm for every stream.

The slow_fast classical exemple

```
ys = 0 fby slow(1, ys);
```

ys	0	3.14	6.28	9.42	12.56	...

- `slow`: step integration with horizon of 1 second.

The slow_fast classical exemple

```
ys = 0 fby slow(1, ys);
yf = 0 fby fast(1, yf);
```

ys	0	3.14	6.28	9.42	12.56	...
yf	0	3	6	9	12	...

- `slow`: step integration with horizon of 1 second.
- `fast`: fast approximate

The `slow_fast` classical exemple

```
ys = 0 fby slow(1, ys);
yf = 0 fby fast(1/3, yf);
```

ys	0	3.14	6.28	9.42	12.56	...					
yf	0	1	2	3	4	5	6	7	8	9	10...

- `slow`: step integration with horizon of 1 second.
- `fast`: fast approximate with horizon of 1/3 second.

The `slow_fast` classical exemple

```
ys = 0 fby slow(1, ys);
yf = 0 fby fast(1/3, yf);
big = period3();
y = merge big ys (yf whenot big);
```

big	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	...
ys	0	.	.	3.14	.	.	6.28	.	...
yf	0	1	2	3	4	5	6	7	...
y	0	1	2	3.14	4	5	6.28	7	...

- `slow`: step integration with horizon of 1 second.
- `fast`: fast approximate with horizon of 1/3 second.
- We use the correct value when possible.

The slow_fast classical exemple

```
ys = 0 fby slow(1, ys);
yf = 0 fby fast(1/3, yf);
big = period3();
y = merge big ys (yf whenot big);
```

big	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	...
ys	0	.	.	3.14	.	.	6.28	.	...
yf	0	1	2	3	4	5	6	7	...
y	0	1	2	3.14	4	5	6.28	7	...

- slow: step integration with horizon of 1 second.
- fast: fast approximate with horizon of 1/3 second.
- We use the correct value when possible.
- And complement with the approximate one.

The slow_fast classical exemple

```
ys = 0 fby slow(1, ys);
yf = 0 fby fast(1/3, y);
big = period3();
y = merge big ys (yf whenot big);
```

big	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	...
ys	0	.	.	3.14	.	.	6.28	.	...
yf	0	1	2	3	4.14	5.14	6.14	7.28	...
y	0	1	2	3.14	4.14	5.14	6.28	7.28	...

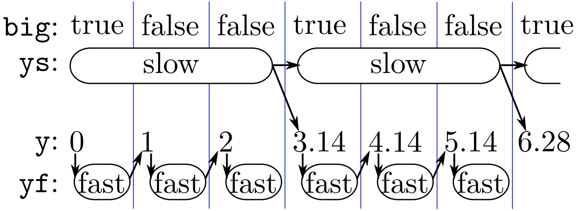
- slow: step integration with horizon of 1 second.
- fast: fast approximate with horizon of 1/3 second.
- We use the correct value when possible.
- And complement with the approximate one.

The slow_fast classical exemple

```
ys = 0 fby slow(1, ys);
yf = 0 fby fast(1/3, y);
big = period3();
y = merge big ys (yf whenot big);
```

big	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	...
ys	0	.	.	3.14	.	.	6.28	.	...
yf	0	1	2	3	4.14	5.14	6.14	7.28	...
y	0	1	2	3.14	4.14	5.14	6.28	7.28	...

We would like to run them in parallel:

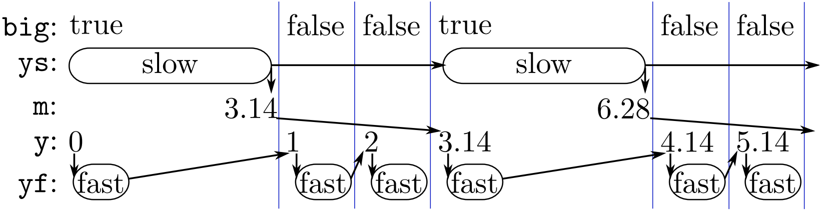


The `slow_fast` classical exemple

```
ys = 0 fby slow(1, ys);
yf = 0 fby fast(1/3, y);
big = period3();
y = merge big ys (yf whenot big);
```

big	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	...
ys	0	.	.	3.14	.	.	6.28	.	...
yf	0	1	2	3	4.14	5.14	6.14	7.28	...
y	0	1	2	3.14	4.14	5.14	6.28	7.28	...

This is what happens, unfortunately:



Synchronous register are synchronous

```
class Slow_fast {
  Fast fast;
  Slow slow;
  Period3 period3;
  float m;
  float m2;
  void reset () {
    period3.reset();
    slow.reset();
    fast.reset();
    m = 0.f;
    m2 = 0.f;
  }
  float step () {
    float y;
    boolean big;
    big = period3.step();
    if (big) {
      y = m;
      m = slow.step(1.f, y);
    } else {
      y = m2;
    }
    m2 = fast.step(0.3f,y);
    return y;
  }
}
```

Reminder:

- `y` gets the value of the register `m`.
- During the same step, `m` is *updated for the next time*.

Synchronous register are synchronous

```
class Slow_fast {
  Fast fast;
  Slow slow;
  Period3 period3;
  float m;
  float m2;
  void reset () {
    period3.reset();
    slow.reset();
    fast.reset();
    m = 0.f;
    m2 = 0.f;
  }
  float step () {
    float y;
    boolean big;
    big = period3.step();
    if (big) {
      y = m;
      m = slow.step(1.f, y);
    } else {
      y = m2;
    }
    m2 = fast.step(0.3f,y);
    return y;
  }
}
```

Reminder:

- y gets the value of the register m.
- During the same step, m is *updated for the next time*.

This sequential compilation is:

- very efficient and simple
- traceable
- used and certified in Scade 6

But it *prevents parallelization across step boundaries*.

Synchronous register are synchronous

```
class Slow_fast {
  Fast fast;
  Slow slow;
  Period3 period3;
  float m;
  float m2;
  void reset () {
    period3.reset();
    slow.reset();
    fast.reset();
    m = 0.f;
    m2 = 0.f;
  }
  float step () {
    float y;
    boolean big;
    big = period3.step();
    if (big) {
      y = m;
      m = slow.step(1.f, y);
    } else {
      y = m2;
    }
    m2 = fast.step(0.3f,y);
    return y;
  }
}
```

Reminder:

- y gets the value of the register m.
- During the same step, m is *updated for the next time*.

This sequential compilation is:

- very efficient and simple
- traceable
- used and certified in Scade 6

But it *prevents parallelization across step boundaries*.

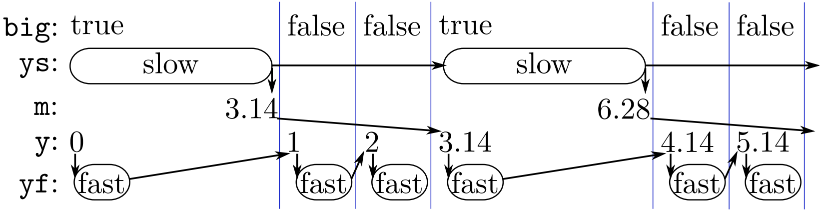
OCREP by A. Girault

The distributed imperative code is *optimized to bypass* the synchronous register.

Decoupling slow_fast with *futures*

```
node slow_fast() = (y :float)
var big :bool; yf :float; ys :float
let
  ys = 0 fby slow(1, ys);
  yf = 0 fby fast(1/3, y);
  big = period3();
  y = merge big ys (yf whenot big);
tel
```

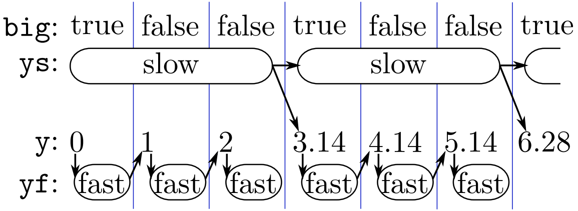
We had this:



Decoupling slow_fast with *futures*

```
node slow_fast() = (y :float)
var big :bool; yf :float; ys :float
let
  ys = 0 fby slow(1, ys);
  yf = 0 fby fast(1/3, y);
  big = period3();
  y = merge big ys (yf whenot big);
tel
```

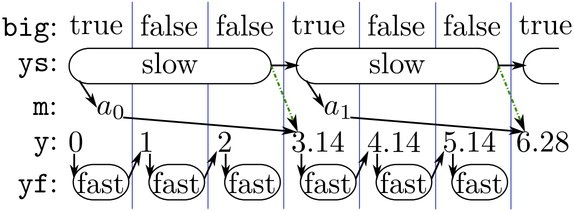
We want this:



Decoupling `slow_fast` with *futures*

```
node slow_fast() = (y :float)
var big :bool; yf :float; ys :float
let
  ys = 0 fby slow(1, ys);
  yf = 0 fby fast(1/3, y);
  big = period3();
  y = merge big ys (yf whenot big);
tel
```

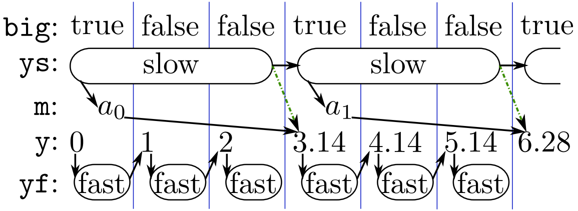
We can use *futures* as proxies:



Decoupling `slow_fast` with *futures*

```
node slow_fast_a() = (y :float)
var big :bool; yf :float; ys :float
let
  ys = 0 fby (async slow(1, ys));
  yf = 0 fby fast(1/3, y);
  big = period3();
  y = merge big ys (yf whenot big);
tel
```

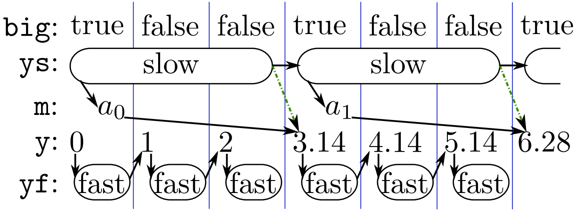
We can use *futures* as proxies:



Decoupling slow_fast with *futures*

```
node slow_fast_a() = (y :float)
var big :bool; yf :float; ys :future float
let
  ys = (async 0) fby (async slow(1, ys));
  yf = 0 fby fast(1/3, y);
  big = period3();
  y = merge big ys (yf whenot big);
tel
```

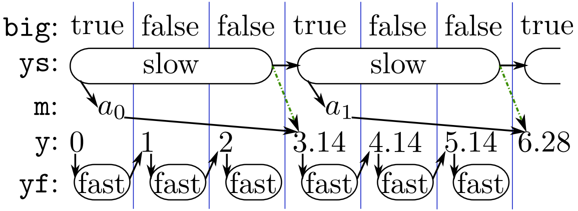
We can use *futures* as proxies:



Decoupling slow_fast with *futures*

```
node slow_fast_a() = (y :float)
var big :bool; yf :float; ys :future float
let
  ys = (async 0) fby (async slow(1, ys));
  yf = 0 fby fast(1/3, y);
  big = period3();
  y = merge big !ys (yf whenot big);
tel
```

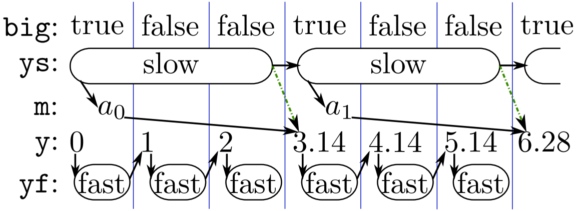
We can use *futures* as proxies:



Decoupling `slow_fast` with *futures*

```
node slow_fast_a() = (y :float)
var big :bool; yf :float; ys :future float
let
  ys = (async 0) fby (async slow(1, !ys));
  yf = 0 fby fast(1/3, y);
  big = period3();
  y = merge big !ys (yf whenot big);
tel
```

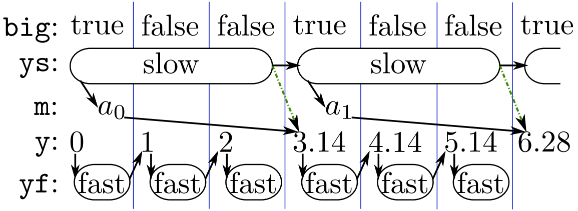
We can use *futures* as proxies:



Decoupling slow_fast with *futures*

```
node slow_fast_a() = (y :float)
var big :bool; yf :float; ys :future float
let
  ys = (async 0) fby (async slow(1, !ys));
  yf = 0 fby fast(1/3, y);
  big = period3();
  y = merge big !ys (yf whenot big);
tel
```

We can use *futures* as proxies:



Futures

- Futures appears in MultiLisp by Halstead 1985.
- Are now in most functional languages and Java, C++, etc.
- Depending on language integration, it can be a mere library.

Futures

- Futures appears in MultiLisp by Halstead 1985.
- Are now in most functional languages and Java, C++, etc.
- Depending on language integration, it can be a mere library.

What is the future?

Futures

- Futures appears in MultiLisp by Halstead 1985.
- Are now in most functional languages and Java, C++, etc.
- Depending on language integration, it can be a mere library.

What is a future?

Futures

- Futures appears in MultiLisp by Halstead 1985.
- Are now in most functional languages and Java, C++, etc.
- Depending on language integration, it can be a mere library.

What is a future?
It is *a value*, which will hold the result of a *closed term*.

Futures

- Futures appears in MultiLisp by Halstead 1985.
- Are now in most functional languages and Java, C++, etc.
- Depending on language integration, it can be a mere library.

What is a future?

It is *a value*, which will hold the result of a *closed term*.
Intuitively, it is a *promise* of result that is *bound to come*.

Futures

- Futures appears in MultiLisp by Halstead 1985.
- Are now in most functional languages and Java, C++, etc.
- Depending on language integration, it can be a mere library.

What is a future?

It is *a value*, which will hold the result of a *closed term*.
Intuitively, it is a *promise* of result that is *bound to come*.

To guarantee futures integrity in Heptagon:

- `future t` is an *abstract type*, with `t` being the result type.
- A future may *only* be created from:
  - Constants: `async 42`
  - Asynchronous function calls: `async f(x,y)`
- `!x` "get" the result held by the future `x` — it is *blocking*.

Unchanged compilation: find the 4 differences

```
class Slow_fast {
  Fast fast;
  Slow slow;
  Period3 period3;
  float m; float m2;
  void reset () {
    period3.reset();
    slow.reset();
    fast.reset();
    m = 0.f;
    m2 = 0.f;
  }
  float step () {
    float y;
    boolean big = period3.step();
    if (big) {
      y = m;
      m = slow.step(1.f, y);
    } else {
      y = m2;
    }
    m2 = fast.step(0.3f,y);
    return y;
  }
}
```

```
class Slow_fast_a {
  Fast fast;
  Async<Slow> slow;
  Period3 period3;
  Future<float> m; float m2;
  void reset () {
    period3.reset();
    slow.reset();
    fast.reset();
    m = new Future(0.f);
    m2 = 0.f;
  }
  float step () {
    float y;
    boolean big = period3.step();
    if (big) {
      y = m.get();
      m = slow.step(1.f, y);
    } else {
      y = m2;
    }
    m2 = fast.step(0.3f,y);
    return y;
  }
}
```

Unchanged compilation: find the 4 differences

```
class Slow_fast {
  Fast fast;
  Slow slow;
  Period3 period3;
  float m; float m2;
  void reset () {
    period3.reset();
    slow.reset();
    fast.reset();
    m = 0.f;
    m2 = 0.f;
  }
  float step () {
    float y;
    boolean big = period3.step();
    if (big) {
      y = m;
      m = slow.step(1.f, y);
    } else {
      y = m2;
    }
    m2 = fast.step(0.3f,y);
    return y;
  }
}
```

```
class Slow_fast_a {
  Fast fast;
  Async<Slow> slow;
  Period3 period3;
  Future<float> m; float m2;
  void reset () {
    period3.reset();
    slow.reset();
    fast.reset();
    m = new Future(0.f);
    m2 = 0.f;
  }
  float step () {
    float y;
    boolean big = period3.step();
    if (big) {
      y = m.get();
      m = slow.step(1.f, y);
    } else {
      y = m2;
    }
    m2 = fast.step(0.3f,y);
    return y;
  }
}
```

Unchanged compilation: find the 4 differences

```
class Slow_fast {
  Fast fast;
  Slow slow;
  Period3 period3;
  float m; float m2;
  void reset () {
    period3.reset();
    slow.reset();
    fast.reset();
    m = 0.f;
    m2 = 0.f;
  }
  float step () {
    float y;
    boolean big = period3.step();
    if (big) {
      y = m;
      m = slow.step(1.f, y);
    } else {
      y = m2;
    }
    m2 = fast.step(0.3f,y);
    return y;
  }
}
```

```
class Slow_fast_a {
  Fast fast;
  Async<Slow> slow;
  Period3 period3;
  Future<float> m; float m2;
  void reset () {
    period3.reset();
    slow.reset();
    fast.reset();
    m = new Future(0.f);
    m2 = 0.f;
  }
  float step () {
    float y;
    boolean big = period3.step();
    if (big) {
      y = m.get();
      m = slow.step(1.f, y);
    } else {
      y = m2;
    }
    m2 = fast.step(0.3f,y);
    return y;
  }
}
```

Unchanged compilation: find the 4 differences

```
class Slow_fast {
  Fast fast;
  Slow slow;
  Period3 period3;
  float m; float m2;
  void reset () {
    period3.reset();
    slow.reset();
    fast.reset();
    m = 0.f;
    m2 = 0.f;
  }
  float step () {
    float y;
    boolean big = period3.step();
    if (big) {
      y = m;
      m = slow.step(1.f, y);
    } else {
      y = m2;
    }
    m2 = fast.step(0.3f,y);
    return y;
  }
}
```

```
class Slow_fast_a {
  Fast fast;
  Async<Slow> slow;
  Period3 period3;
  Future<float> m; float m2;
  void reset () {
    period3.reset();
    slow.reset();
    fast.reset();
    m = new Future(0.f);
    m2 = 0.f;
  }
  float step () {
    float y;
    boolean big = period3.step();
    if (big) {
      y = m.get();
      m = slow.step(1.f, y);
    } else {
      y = m2;
    }
    m2 = fast.step(0.3f,y);
    return y;
  }
}
```

Unchanged compilation: find the 4 differences

```
class Slow_fast {
  Fast fast;
  Slow slow;
  Period3 period3;
  float m; float m2;
  void reset () {
    period3.reset();
    slow.reset();
    fast.reset();
    m = 0.f;
    m2 = 0.f;
  }
  float step () {
    float y;
    boolean big = period3.step();
    if (big) {
      y = m;
      m = slow.step(1.f, y);
    } else {
      y = m2;
    }
    m2 = fast.step(0.3f,y);
    return y;
  }
}
```

```
class Slow_fast_a {
  Fast fast;
  Async<Slow> slow;
  Period3 period3;
  Future<float> m; float m2;
  void reset () {
    period3.reset();
    slow.reset();
    fast.reset();
    m = new Future(0.f);
    m2 = 0.f;
  }
  float step () {
    float y;
    boolean big = period3.step();
    if (big) {
      y = m.get();
      m = slow.step(1.f, y);
    } else {
      y = m2;
    }
    m2 = fast.step(0.3f,y);
    return y;
  }
}
```

The `async` wrapper

The `async` wrapper

- runs asynchronously a node in a worker thread.
- behaves like a node:
  - `step`
    - At each input a future is returned.
    - Inputs are fed to the wrapped node through a buffer.
  - `reset` is done so as to allow data-parallelism.

The `async` wrapper

The `async` wrapper

- runs asynchronously a node in a worker thread.
- behaves like a node:
  - `step`
    - At each input a future is returned.
    - Inputs are fed to the wrapped node through a buffer.
  - `reset` is done so as to allow data-parallelism.

Two exemples to illustrate

- the need of an input buffer to allow *decoupling*
- the use of reset to enable *data-parallelism*

Partial Decoupling

```
c = period3();
y0 = sum(1);
y1 = sum(2);
y = (y0 when c) + (y1 when c);
```

Partial Decoupling

```
c = period3();
y0 = sum(1);
y1 = sum(2);
y = (y0 when c) + (y1 when c);
```

c	*true*
y0	1
y1	2
y	3

Partial Decoupling

```
c = period3();
y0 = sum(1);
y1 = sum(2);
y = (y0 when c) + (y1 when c);
```

c	*true*	*false*
y0	1	2
y1	2	4
y	3	.

Partial Decoupling

```
c = period3();
y0 = sum(1);
y1 = sum(2);
y = (y0 when c) + (y1 when c);
```

c	*true*	*false*	*false*
y0	1	2	3
y1	2	4	6
y	3	.	.

Partial Decoupling

```
c = period3();
y0 = sum(1);
y1 = sum(2);
y = (y0 when c) + (y1 when c);
```

c	*true*	*false*	*false*	*true*
y0	1	2	3	4
y1	2	4	6	8
y	3	.	.	12

Partial Decoupling

```
c = period3();
y0 = sum(1);
y1 = sum(2);
y = (y0 when c) + (y1 when c);
```

c	*true*	*false*	*false*	*true*	*false*	*false*	*true*	...
y0	1	2	3	4	5	6	7	...
y1	2	4	6	8	10	12	14	...
y	3	.	.	12	.	.	21	...

Partial Decoupling

```
c = period3();
y0 = sum(1);
y1 = sum(2);
y = (y0 when c) + (y1 when c);
```

c	*true*	*false*	*false*	*true*	*false*	*false*	*true*	...
y0	1	2	3	4	5	6	7	...
y1	2	4	6	8	10	12	14	...
y	3	.	.	12	.	.	21	...

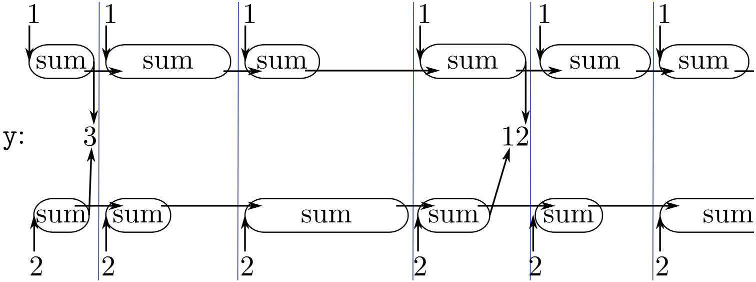
Consider `sum` takes a variable time to execute

Partial Decoupling

```
c = period3();
y0 = sum(1);
y1 = sum(2);
y = (y0 when c) + (y1 when c);
```

c	*true*	*false*	*false*	*true*	*false*	*false*	*true*	...
y0	1	2	3	4	5	6	7	...
y1	2	4	6	8	10	12	14	...
y	3	.	.	12	.	.	21	...

Consider `sum` takes a variable time to execute, then:

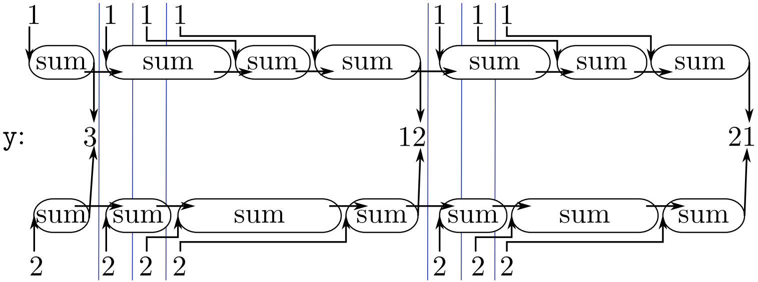


Partial Decoupling

```
c = period3();
y0 = sum(1);
y1 = sum(2);
y = (y0 when c) + (y1 when c);
```

c	*true*	*false*	*false*	*true*	*false*	*false*	*true*	...
y0	1	2	3	4	5	6	7	...
y1	2	4	6	8	10	12	14	...
y	3	.	.	12	.	.	21	...

We would like to smooth out this variability:

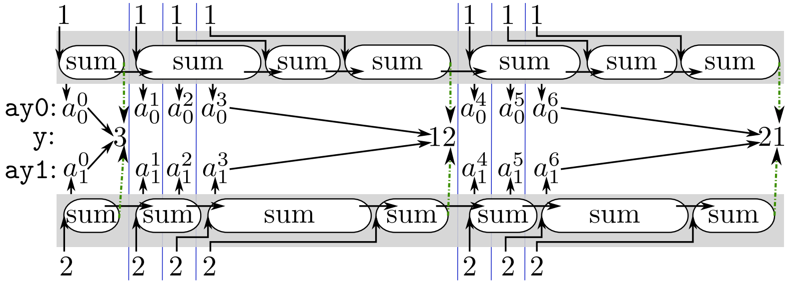


Partial Decoupling

```
c = period3();
y0 = sum(1);
y1 = sum(2);
y = (y0 when c) + (y1 when c);
```

c	*true*	*false*	*false*	*true*	*false*	*false*	*true*	...
y0	1	2	3	4	5	6	7	...
y1	2	4	6	8	10	12	14	...
y	3	.	.	12	.	.	21	...

We would like to smooth out this variability with futures:

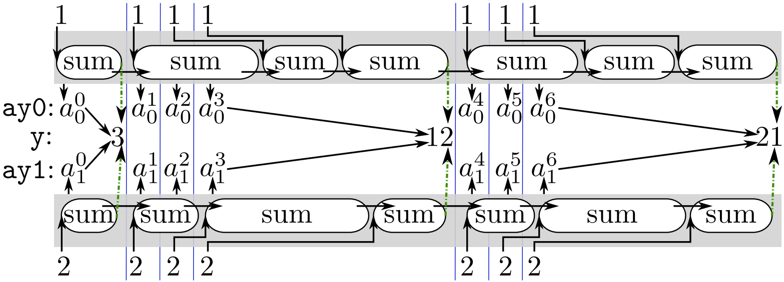


Partial Decoupling

```
c = period3();
y0 = sum(1);
y1 = sum(2);
y = (y0 when c) + (y1 when c);
```

c	*true*	*false*	*false*	*true*	*false*	*false*	*true*	...
y0	1	2	3	4	5	6	7	...
y1	2	4	6	8	10	12	14	...
y	3	.	.	12	.	.	21	...

We would like to smooth out this variability with futures:

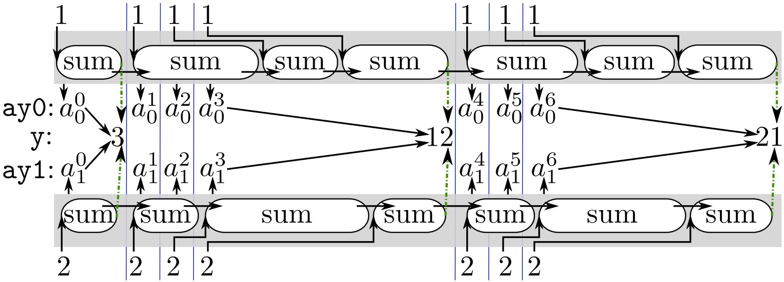


Partial Decoupling

```
c = period3();
ay0 = async sum(1);
ay1 = async sum(2);
y = !(ay0 when c) + !(ay1 when c);
```

c	*true*	*false*	*false*	*true*	*false*	*false*	*true*	...
ay0								
ay1								
y								

We would like to smooth out this variability with futures:

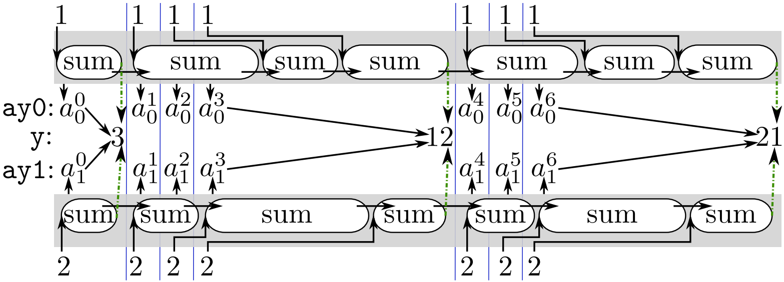


Partial Decoupling

```
c = period3();
ay0 = async sum(1);
ay1 = async sum(2);
y = !(ay0 when c) + !(ay1 when c);
```

c	*true*	*false*	*false*	*true*	*false*	*false*	*true*	...
ay0	a_0^0							
ay1	a_1^0							
y	$!a_0^0 + !a_1^0$							

We would like to smooth out this variability with futures:

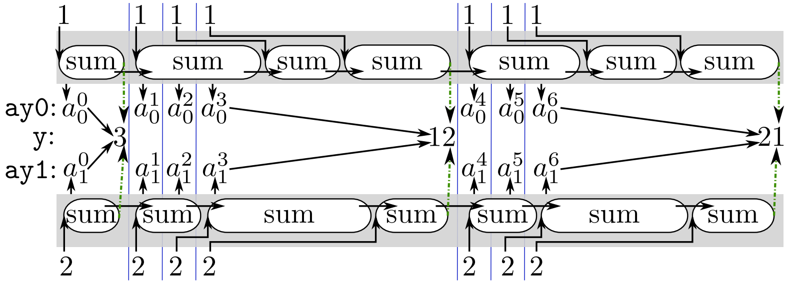


Partial Decoupling

```
c = period3();
ay0 = async sum(1);
ay1 = async sum(2);
y = !(ay0 when c) + !(ay1 when c);
```

c	*true*	*false*	*false*	*true*	*false*	*false*	*true*	...
ay0	a_0^0							
ay1	a_1^0							
y	3							

We would like to smooth out this variability with futures:

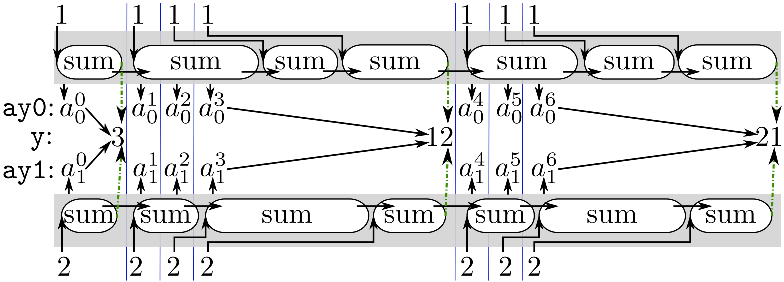


Partial Decoupling

```
c = period3();
ay0 = async sum(1);
ay1 = async sum(2);
y = !(ay0 when c) + !(ay1 when c);
```

c	*true*	*false*	*false*	*true*	*false*	*false*	*true*	...
ay0	a_0^0	a_0^1						
ay1	a_1^0	a_1^1						
y	3	.						

We would like to smooth out this variability with futures:

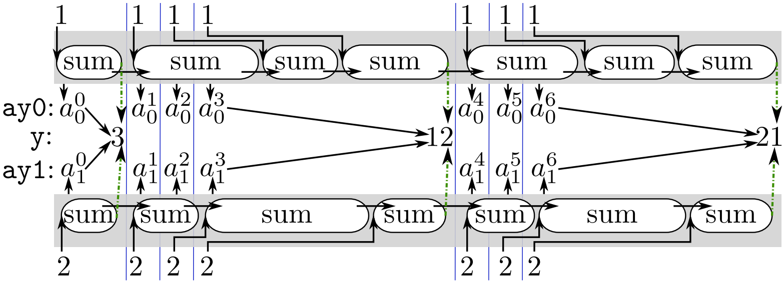


Partial Decoupling

```
c = period3();
ay0 = async sum(1);
ay1 = async sum(2);
y = !(ay0 when c) + !(ay1 when c);
```

c	*true*	*false*	*false*	*true*	*false*	*false*	*true*	...
ay0	a_0^0	a_0^1	a_0^2					
ay1	a_1^0	a_1^1	a_1^2					
y	3	.	.					

We would like to smooth out this variability with futures:

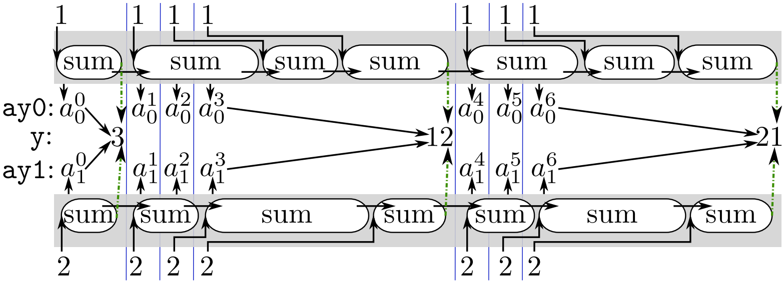


Partial Decoupling

```
c = period3();
ay0 = async sum(1);
ay1 = async sum(2);
y = !(ay0 when c) + !(ay1 when c);
```

c	*true*	*false*	*false*	*true*	*false*	*false*	*true*	...
ay0	a_0^0	a_0^1	a_0^2	a_0^3				
ay1	a_1^0	a_1^1	a_1^2	a_1^3				
y	3	.	.	$!a_0^3 + !a_1^3$				

We would like to smooth out this variability with futures:

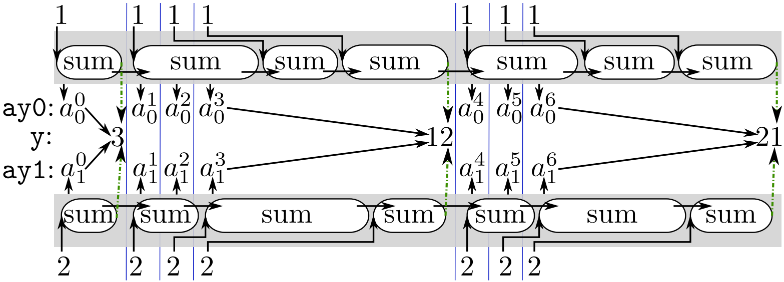


Partial Decoupling

```
c = period3();
ay0 = async sum(1);
ay1 = async sum(2);
y = !(ay0 when c) + !(ay1 when c);
```

c	*true*	*false*	*false*	*true*	*false*	*false*	*true*	...
ay0	a_0^0	a_0^1	a_0^2	a_0^3				
ay1	a_1^0	a_1^1	a_1^2	a_1^3				
y	3	.	.	12				

We would like to smooth out this variability with futures:

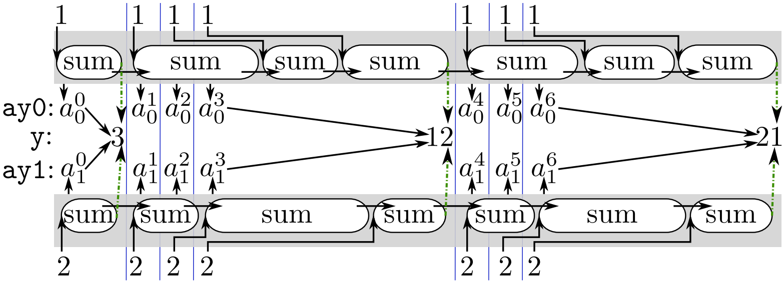


Partial Decoupling

```
c = period3();
ay0 = async sum(1);
ay1 = async sum(2);
y = !(ay0 when c) + !(ay1 when c);
```

c	*true*	*false*	*false*	*true*	*false*	*false*	*true*	...
ay0	a_0^0	a_0^1	a_0^2	a_0^3	a_0^4	a_0^5	a_0^6	...
ay1	a_1^0	a_1^1	a_1^2	a_1^3	a_1^4	a_1^5	a_1^6	...
y	3	.	.	12	.	.	21	...

We would like to smooth out this variability with futures:

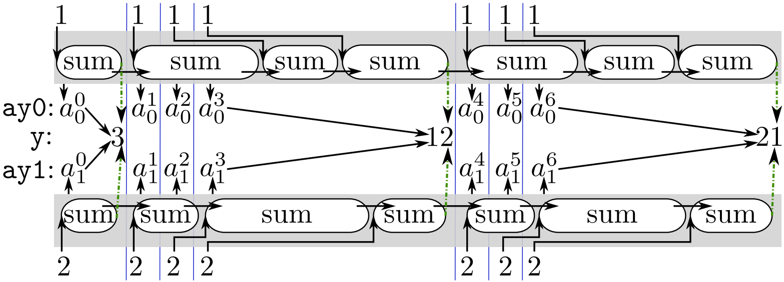


Partial Decoupling

```
c = period3();
ay0 = async<<3>> sum(1);
ay1 = async<<3>> sum(2);
y = !(ay0 when c) + !(ay1 when c);
```

c	*true*	*false*	*false*	*true*	*false*	*false*	*true*	...
ay0	a_0^0	a_0^1	a_0^2	a_0^3	a_0^4	a_0^5	a_0^6	...
ay1	a_1^0	a_1^1	a_1^2	a_1^3	a_1^4	a_1^5	a_1^6	...
y	3	.	.	12	.	.	21	...

We would like to smooth out this variability with futures:



Decoupling

Decoupling is dictated by:

- the size of the `async` input buffer

Decoupling

Decoupling is dictated by:

- the size of the `async` input buffer
- the moment we get the result

Decoupling

Decoupling is dictated by:

- the size of the `async` input buffer
- the moment we get the result

Last exemple was partial decoupling because periodically we required the result of the *current* instant.

Decoupling

Decoupling is dictated by:

- the size of the `async` input buffer
- the moment we get the result

Last exemple was partial decoupling because periodically we required the result of the *current* instant.

Decoupling of `n` instants requires:

- an input buffer of size `n`

This is written:

```
async<<n>> f(x)
```

Decoupling

Decoupling is dictated by:

- the size of the `async` input buffer
- the moment we get the result

Last exemple was partial decoupling because periodically we required the result of the *current* instant.

Decoupling of `n` instants requires:

- an input buffer of size `n`
- a *delay* of `n` instant on the result

This is written:

```
async 0 fby<<n>> (async<<n>> f(x))
```

Decoupling

Decoupling is dictated by:

- the size of the `async` input buffer
- the moment we get the result

Last exemple was partial decoupling because periodically we required the result of the *current* instant.

Decoupling of `n` instants requires:

- an input buffer of size `n`
- a *delay* of `n` instant on the result

This is written:

```
!(async 0 fby<<n>> (async<<n>> f(x)))
```

Decoupling

Decoupling is dictated by:

- the size of the `async` input buffer
- the moment we get the result

Last exemple was partial decoupling because periodically we required the result of the *current* instant.

Decoupling of `n` instants requires:

- an input buffer of size `n`
- a *delay* of `n` instant on the result

This is written:

```
!(async 0 fby<<n>> (async<<n>> f(x)))
```

Decoupling is always statically bounded

Resetting a node: the every keyword

```
c = period2();
y = sum(x) every c;
```

Resetting a node: the `every` keyword

```
c = period2();
y = sum(x) every c;
```

Resetting is done *prior* to the call to the node:

x	1
c	*true*
m	0
y	

Resetting a node: the every keyword

```
c = period2();
y = sum(x) every c;
```

Resetting is done *prior* to the call to the node:

x	1
c	*true*
m	0
y	1

Resetting a node: the every keyword

```
c = period2();
y = sum(x) every c;
```

Resetting is done *prior* to the call to the node:

x	1
c	*true*
m	0
y	1

Resetting a node: the every keyword

```
c = period2();
y = sum(x) every c;
```

Resetting is done *prior* to the call to the node:

x	1	2
c	*true*	*false*
m	0	1
y	1	

Resetting a node: the every keyword

```
c = period2();
y = sum(x) every c;
```

Resetting is done *prior* to the call to the node:

x	1	2
c	*true*	*false*
m	0	1
y	1	3

Resetting a node: the every keyword

```
c = period2();
y = sum(x) every c;
```

Resetting is done *prior* to the call to the node:

x	1	2	
c	*true*	*false*	
m	0	↗ 1	↗ 3
y	1	3	

Resetting a node: the every keyword

```
c = period2();
y = sum(x) every c;
```

Resetting is done *prior* to the call to the node:

x	1	2	3
c	*true*	*false*	*true*
m	0	1	0
y	1	3	

Resetting a node: the every keyword

```
c = period2();
y = sum(x) every c;
```

Resetting is done *prior* to the call to the node:

x	1	2	3
c	*true*	*false*	*true*
m	0	1	0
y	1	3	

Resetting a node: the `every` keyword

```
c = period2();
y = sum(x) every c;
```

Resetting is done *prior* to the call to the node:

x	1	2	3
c	*true*	*false*	*true*
m	0	1	0
y	1	3	3

Resetting a node: the `every` keyword

```
c = period2();
y = sum(x) every c;
```

Resetting is done *prior* to the call to the node:

x	1	2	3	4
c	*true*	*false*	*true*	*false*
m	0	1	0	
y	1	3	3	

Resetting a node: the `every` keyword

```
c = period2();
y = sum(x) every c;
```

Resetting is done *prior* to the call to the node:

x	1	2	3	4
c	*true*	*false*	*true*	*false*
m	0	↗ 1	0	↗ 3
y	1	3	3	

Resetting a node: the `every` keyword

```
c = period2();
y = sum(x) every c;
```

Resetting is done *prior* to the call to the node:

x	1	2	3	4
c	*true*	*false*	*true*	*false*
m	0	↗ 1	0	↗ 3
y	1	3	3	7

Resetting a node: the every keyword

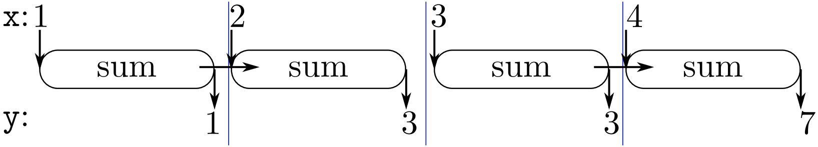
```
c = period2();
y = sum(x) every c;
```

Resetting is done *prior* to the call to the node:

x	1	2	3	4	5	6	7	...
c	*true*	*false*	*true*	*false*	*true*	*false*	*true*	...
m	0	1	0	3	0	5	0	...
y	1	3	3	7	5	3	7	...

Data-parallelism

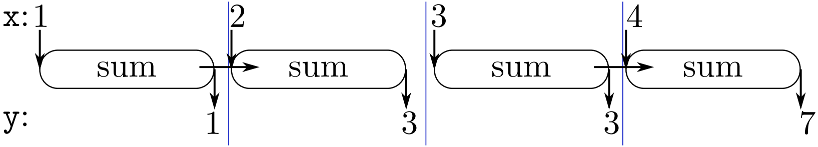
```
c = period2();
y = sum(x) every c;
```



- Reset removes dependencies, enabling data-parallelism.

Data-parallelism

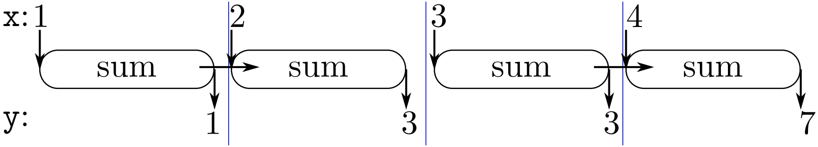
```
c = period2();
ay = async<<2>> sum(x) every c;
```



- Reset removes dependencies, enabling data-parallelism.
- Decoupling of at least 2 instants is required.

Data-parallelism

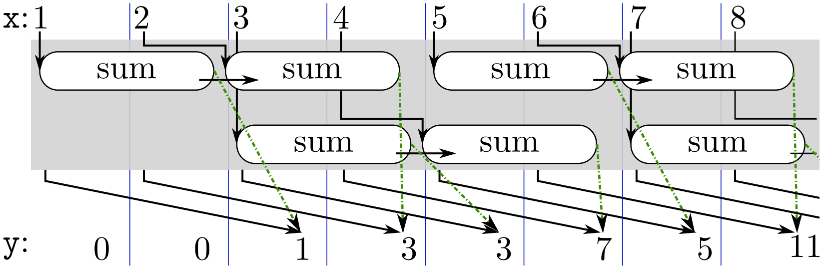
```
c = period2();
ay = async<<2>> sum(x) every c;
y = !(async 0 fby<<2>> ay);
```



- Reset removes dependencies, enabling data-parallelism.
- Decoupling of at least 2 instants is required.

Data-parallelism

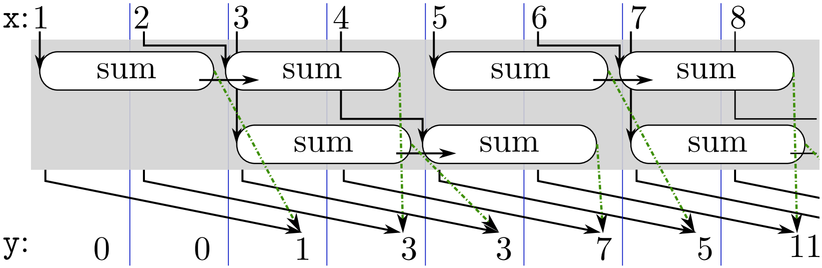
```
c = period2();
ay = async<<2>> sum(x) every c;
y = !(async 0 fby<<2>> ay);
```



- Reset removes dependencies, enabling data-parallelism.
- Decoupling of at least 2 instants is required.

Data-parallelism

```
c = period2();
ay = async<<2,2>> sum(x) every c;
y = !(async 0 fby<<2>> ay);
```



- Reset removes dependencies, enabling data-parallelism.
- Decoupling of at least 2 instants is required.

Thread number is static and explicit, here 2 are needed.

Memory management

A future

- is a shared object with one producer, multiple consumers
- may be stored and used later on
- may not be used
- may not depend on previous ones

Memory management

A future

- is a shared object with one producer, multiple consumers
- may be stored and used later on
- may not be used
- may not depend on previous ones

Without restrictions, the live-range of a future is *undecidable* and a *concurrent gc* is needed, as the one of java.

Memory management

A future

- is a shared object with one producer, multiple consumers
- may be stored and used later on
- may not be used
- may not depend on previous ones

Without restrictions, the live-range of a future is *undecidable* and a *concurrent gc* is needed, as the one of java.

Memory boundedness

Alive futures are bounded by the number of synchronous registers.
A slab allocator is possible with *static allocation* and reuse.

Memory management

A future

- is a shared object with one producer, multiple consumers
- may be stored and used later on
- may not be used
- may not depend on previous ones

Without restrictions, the live-range of a future is *undecidable* and a *concurrent gc* is needed, as the one of java.

Memory boundedness

Alive futures are bounded by the number of synchronous registers. *A slab allocator* is possible with *static allocation* and reuse.

Scope restriction for node level memory management

Preventing futures to be returned or passed to an `async` call, allows gc and slab to be *synchronous* and *node local*.

Backends

Existing Java backend

Everything shown was generated with our Java backend.

- Futures are the ones of Java
- Static queues and worker threads
- But dynamic allocation of futures

Backends

Existing Java backend

Everything shown was generated with our Java backend.

- Futures are the ones of Java
- Static queues and worker threads
- But dynamic allocation of futures

Existing C backend, aiming embedded systems

- Hand tailored futures, queues and threads
- SLAB local to each node
- Futures have scope restrictions

Backends

Existing Java backend

Everything shown was generated with our Java backend.

- Futures are the ones of Java
- Static queues and worker threads
- But dynamic allocation of futures

Existing C backend, aiming embedded systems

- Hand tailored futures, queues and threads
- SLAB local to each node
- Futures have scope restrictions

WIP OpenMP Stream backend

- Data-flow parallel runtime
- No thread burden
- May be used to handle large number of `async`

Code generation for embedded systems

`async` may be *annotated* with any needed static arguments.

Code generation for embedded systems

`async` may be *annotated* with any needed static arguments.

Location annotations for distribution without scheduler:

- One thread per computing unit
- No surprise
- Usually not efficient

Code generation for embedded systems

`async` may be *annotated* with any needed static arguments.

Location annotations for distribution without scheduler:

- One thread per computing unit
- No surprise
- Usually not efficient

Priority annotations for EDF scheduling:

- Well known
- May be optimal
- Existing tools need to be adapted

Code generation for embedded systems

`async` may be *annotated* with any needed static arguments.

Location annotations for distribution without scheduler:

- One thread per computing unit
- No surprise
- Usually not efficient

Priority annotations for EDF scheduling:

- Well known
- May be optimal
- Existing tools need to be adapted

Real time period annotations, etc.

Conlusions

Semantics

Same semantics as the sequential program without `async` and `!`.

Expressivness

- Synchronous language: *time programming*
- Futures: decouple and make explicit *beginning* and *end* of computations
- Together they allow for *programing parallelism*:
  - decoupling, partial-decoupling,
  - data-parallelism,
  - fork-join, temporal fork-join,
  - pipeline, etc.

Safety

- No deadlocks: futures in a pure language
- No dynamic memory allocation or thread creation

Synchronization on inputs

```
c = period3();
ay0 = async<<0>> sum(1);
ay1 = async<<0>> sum(2);
y = !(ay0 when c) + !(ay1 when c);
```