Integer clocks, a way to efficient distribution and compilation of synchronous programs

Léonard Gérard[1 2] Marc Pouzet[2] Albert Cohen[3]

[1] ENSL, Lyon France

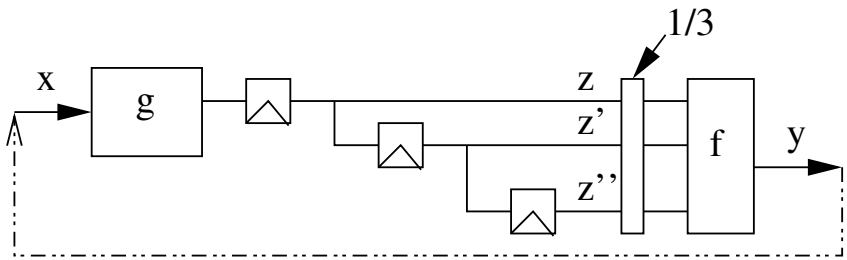
[2] LRI, Univ. Paris-Sud 11 Orsay France

[3] ALCHEMY, INRIA Saclay France

November 30, 2008

f and *g*'s story

```
let filter x = y where
    rec z = 0 fby g(x)
    and z' = z fby z
    and z'' = z' fby z'
  and y = f((z'',z',z) when three)
```



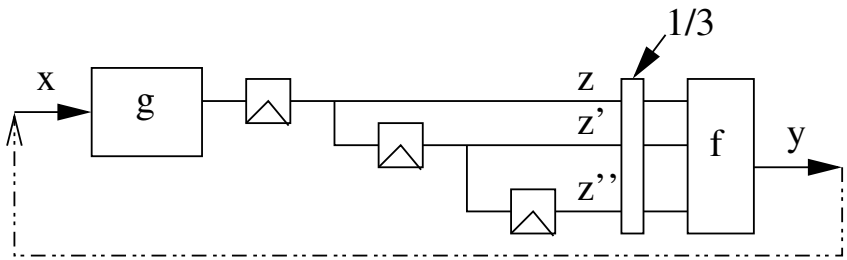
three	t	f	f	t	f	f	t
x	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$g(x)$	$g(x_1)$	$g(x_2)$	$g(x_3)$	$g(x_4)$	$g(x_5)$	$g(x_6)$	$g(x_7)$
$f(\ldots)$	$f(0,0,0)$	.	.	$f(g(x_1),g(x_2),g(x_3))$	.	.	$f(g(x_4),g(x_5$

Long tasks

`f` can be seen as a *long task*
Feedback restrictions : $(x_i, x_{i+1}, x_{i+2}) \nleftarrow y_i$

f and *g*'s story

```
let filter x = y where
    rec z = 0 fby g(x)
    and z' = z fby z
    and z'' = z' fby z'
  and y = f((z'',z',z) when three)
```



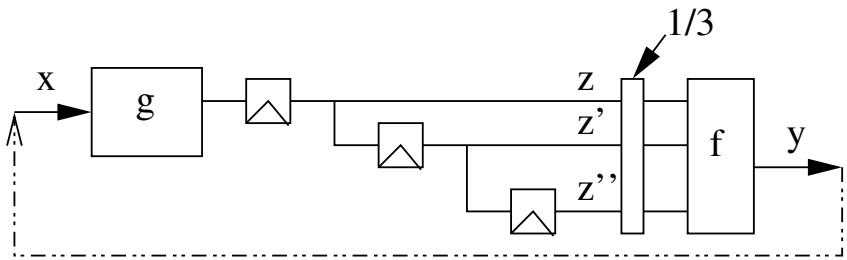
three	*t*	*f*	*f*	*t*	*f*	*f*	*t*
x	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$g(x)$	$g(x_1)$	$g(x_2)$	$g(x_3)$	$g(x_4)$	$g(x_5)$	$g(x_6)$	$g(x_7)$
$f(\ldots)$	$f(0,0,0)$	.	.	$f(g(x_1),g(x_2),g(x_3))$	.	.	$f(g(x_4),g(x_5$

Long tasks

`f` can be seen as a *long task*
Feedback restrictions : $(x_i, x_{i+1}, x_{i+2}) \nleftarrow y_i$

f and *g*'s story

```
let filter x = y where
    rec z = 0 fby g(x)
    and z' = z fby z
    and z'' = z' fby z'
  and y = f((z'',z',z) when three)
```



three	*t*	*f*	*f*	*t*	*f*	*f*	*t*
x	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$g(x)$	$g(x_1)$	$g(x_2)$	$g(x_3)$	$g(x_4)$	$g(x_5)$	$g(x_6)$	$g(x_7)$
$f(\ldots)$	$f(0,0,0)$	.	.	$f(g(x_1),g(x_2),g(x_3))$	.	.	$f(g(x_4),g(x_5$

Long tasks

`f` can be seen as a *long task*
Feedback restrictions : $(x_i, x_{i+1}, x_{i+2}) \nleftarrow y_i$

Efficient long task management

Manual solution

Split f down to f_1, f_2 and f_3

$g(x)$	$g(x_1)$	$g(x_2)$	$g(x_3)$	$g(x_4)$	$g(x_5)$	$g(x_6)$
$f(\dots)$	$f_1(0,0,0)$	$f_2(\dots)$	$f_3(\dots)$	$f_1(g(x_1), g(x_2), g(x_3))$	$f_2(\dots)$	$f_3(\dots)$

Desynchronization solution

Under feedback delay conditions

z	$[g(x_1)\ g(x_2)\ g(x_3)]$	$[g(x_4)\ g(x_5)\ g(x_6)]$	$[g(x_7)\ g(x_8)\ g(x_9)]$
$f(\dots)$	$f(0,0,0)$	$f(g(x_1), g(x_2), g(x_3))$	$f(g(x_4), g(x_5), g(x_6))$

Desynchronization allows

- Distribution $\Longrightarrow$ Concurrent computation
- Loop generation $\Longrightarrow$ Computational efficiency

Efficient long task management

Manual solution

Split f down to f_1, f_2 and f_3

$g(x)$	$g(x_1)$	$g(x_2)$	$g(x_3)$	$g(x_4)$	$g(x_5)$	$g(x_6)$
$f(\dots)$	$f_1(0,0,0)$	$f_2(\dots)$	$f_3(\dots)$	$f_1(g(x_1), g(x_2), g(x_3))$	$f_2(\dots)$	$f_3(\dots)$

Desynchronization solution

Under feedback delay conditions

z	$[g(x_1)\ g(x_2)\ g(x_3)]$	$[g(x_4)\ g(x_5)\ g(x_6)]$	$[g(x_7)\ g(x_8)\ g(x_9)]$
$f(\dots)$	$f(0,0,0)$	$f(g(x_1), g(x_2), g(x_3))$	$f(g(x_4), g(x_5), g(x_6))$

Desynchronization allows

- Distribution $\Longrightarrow$ Concurrent computation
- Loop generation $\Longrightarrow$ Computational efficiency

Efficient long task management

Manual solution

Split f down to f_1, f_2 and f_3

$g(x)$	$g(x_1)$	$g(x_2)$	$g(x_3)$	$g(x_4)$	$g(x_5)$	$g(x_6)$
$f(\ldots)$	$f_1(0,0,0)$	$f_2(\ldots)$	$f_3(\ldots)$	$f_1(g(x_1), g(x_2), g(x_3))$	$f_2(\ldots)$	$f_3(\ldots)$

Desynchronization solution

Under feedback delay conditions

z	$[g(x_1)\ g(x_2)\ g(x_3)]$	$[g(x_4)\ g(x_5)\ g(x_6)]$	$[g(x_7)\ g(x_8)\ g(x_9)]$
$f(\ldots)$	$f(0,0,0)$	$f(g(x_1), g(x_2), g(x_3))$	$f(g(x_4), g(x_5), g(x_6))$

Desynchronization allows

- Distribution $\Longrightarrow$ Concurrent computation
- Loop generation $\Longrightarrow$ Computational efficiency

Lustre Distribution : Ocrep [Girault et al. 94]

$$\text{Lustre} \xrightarrow{\text{classiqueclassical}} \text{OC} \xrightarrow{\text{Ocrep}} \text{OC distributed}$$

OC format : (Object Code) imperative automata

$$\begin{cases} x = in + 1 & \text{at A} \\ y = f(x) & \text{at B} \\ z = g(x+y) & \text{at A} \end{cases} \xrightarrow{\text{duplication and deletion}} A : \begin{cases} x = & in + 1 \\ z = & g(x+y) \end{cases} \quad B : \begin{cases} y = & f(x) \end{cases}$$

$$\xrightarrow{\text{communication}} \quad A : \begin{cases} x = & in + 1 \\ & send(x, B) \\ y = & receive(B) \\ z = & g(x+y) \end{cases} \quad B : \begin{cases} x = & receive(A) \\ y = & f(x) \\ & send(y, A) \end{cases}$$

Back-pressure and synchronization through blind variables.

Long task distribution in Ocrep

Desynchronization in Ocrep

- Complex optimization
- Acts at low-level without coder control
- Needs distribution
- Needs thus different clocks

The other way around

Desynchronization allows

- good distribution
- efficient local computations

Long task distribution in Ocrep

Desynchronization in Ocrep

- Complex optimization
- Acts at low-level without coder control
- Needs distribution
- Needs thus different clocks

The other way around

Desynchronization allows

- good distribution
- efficient local computations

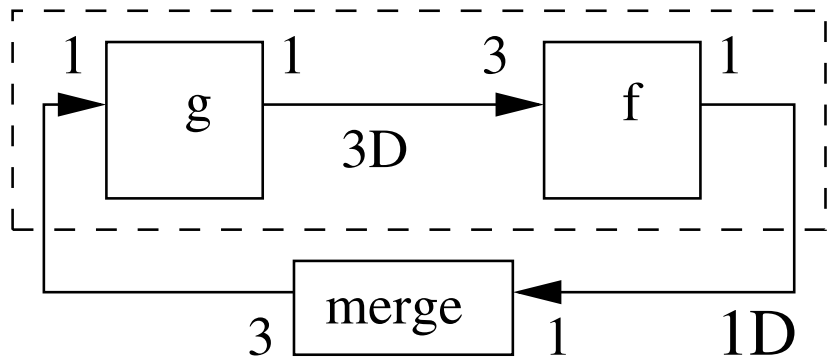
Synchronous Data-Flow (SDF) [Lee et al 87]

- Kahn Semantic
- Periodic systems
- Popular in video streaming : StreamIt (MIT, 02-07), Xstream (ST-Micro 06)

f and *g* in SDF

Close the system to periodic

```
x = merge three (pre y) 0
y = filter(x)
```



- Initialized delay
- Atomic consumption and production

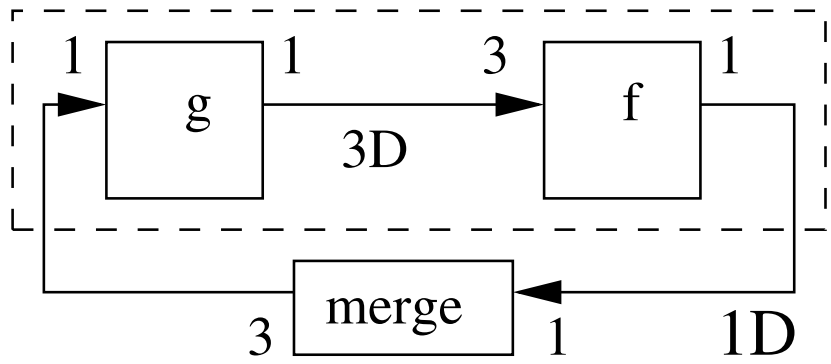
Synchronous Data-Flow (SDF) [Lee et al 87]

- Kahn Semantic
- Periodic systems
- Popular in video streaming : StreamIt (MIT, 02-07), Xstream (ST-Micro 06)

f and *g* in SDF

Close the system to periodic

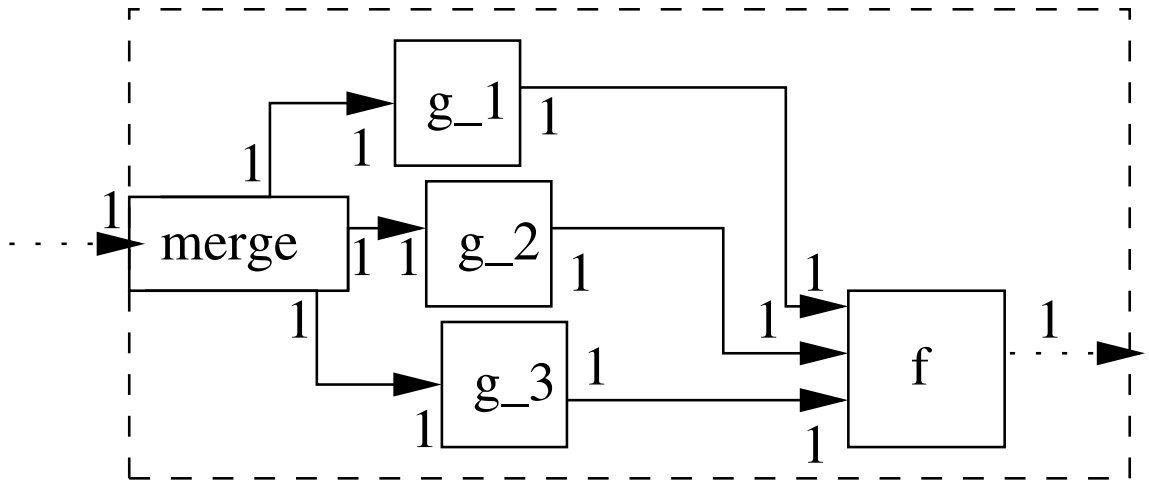
```
x = merge three (pre y) 0
y = filter(x)
```



- Initialized delay
- Atomic consumption and production

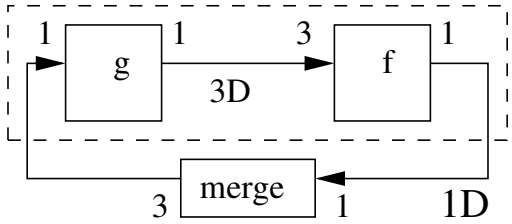
Stationary state

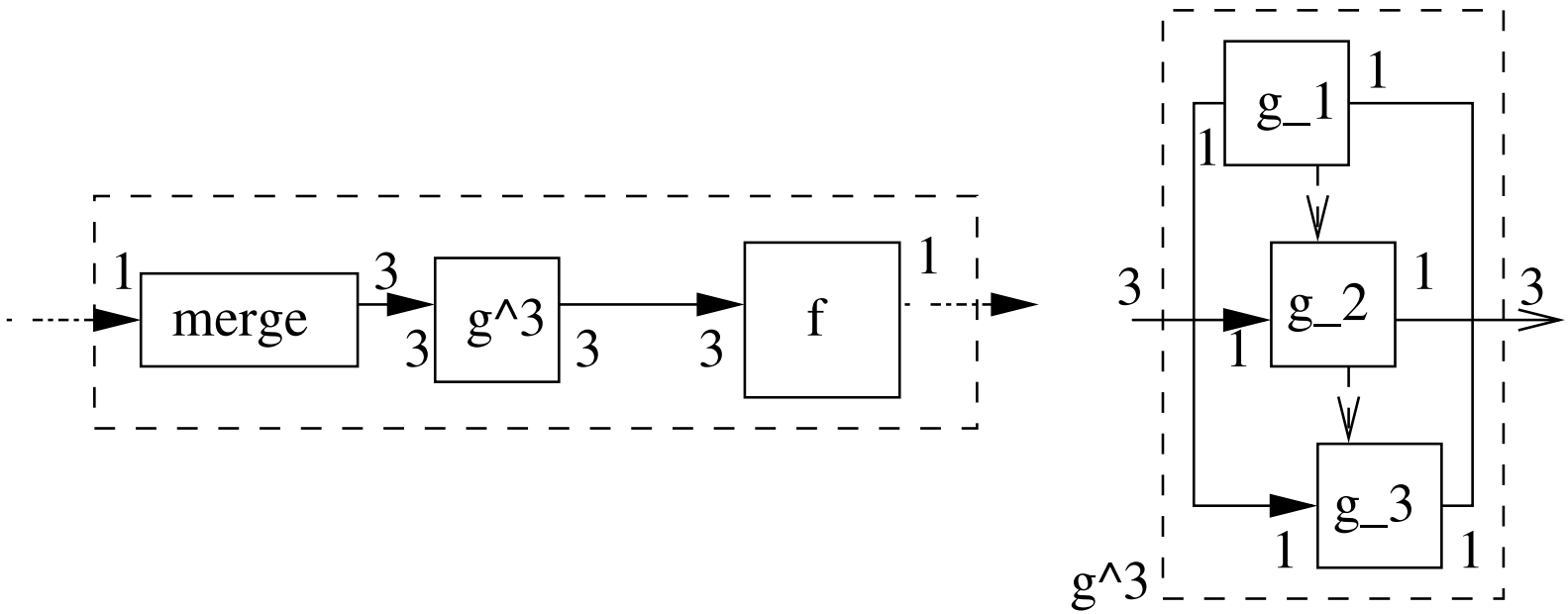
SDF compute the equilibrium (if it exists).



SDF granularity abstraction, micro-schedules

Compilation through micro-schedules



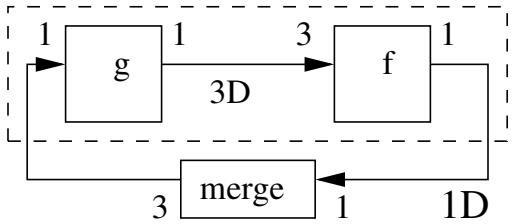


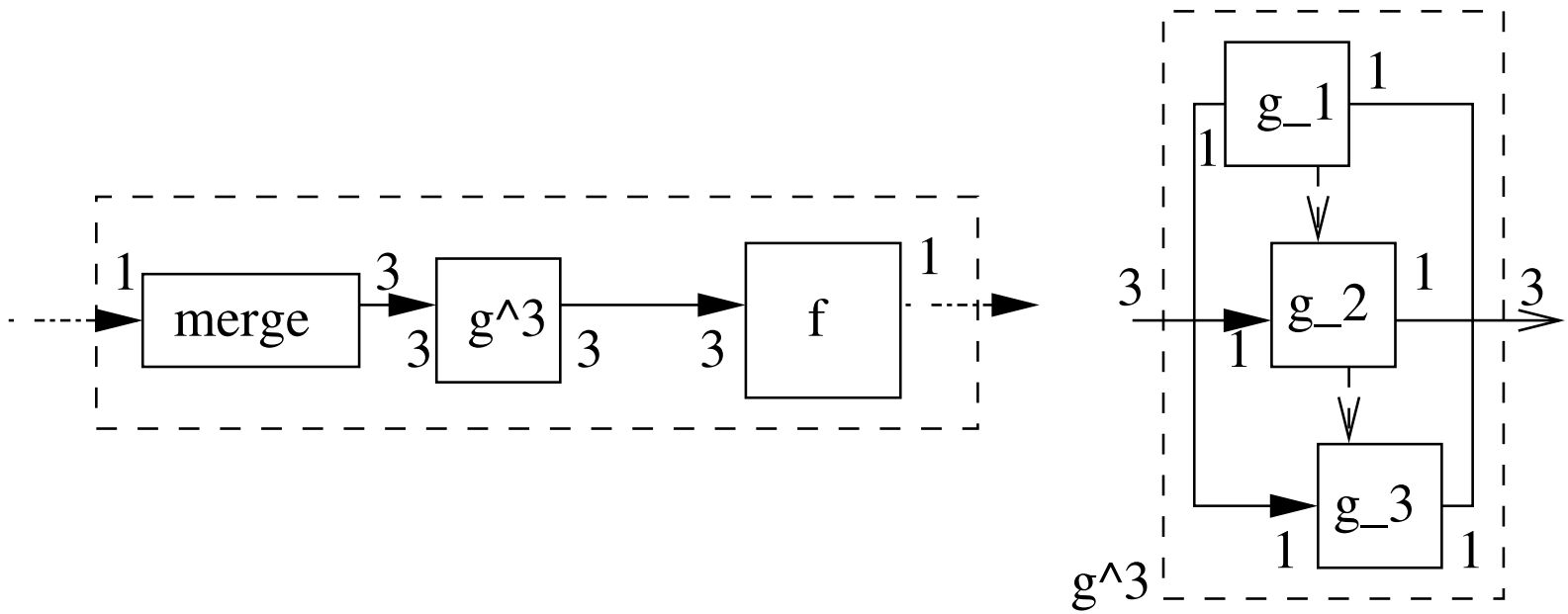
To draw from this

- Express Micro-schedule
- Synchronization at micro-schedules borders
- Fine to coarse grain synchronization

SDF granularity abstraction, micro-schedules

Compilation through micro-schedules





To draw from this

- Express Micro-schedule
- Synchronization at micro-schedules borders
- Fine to coarse grain synchronization

Parenthesized boolean clocks

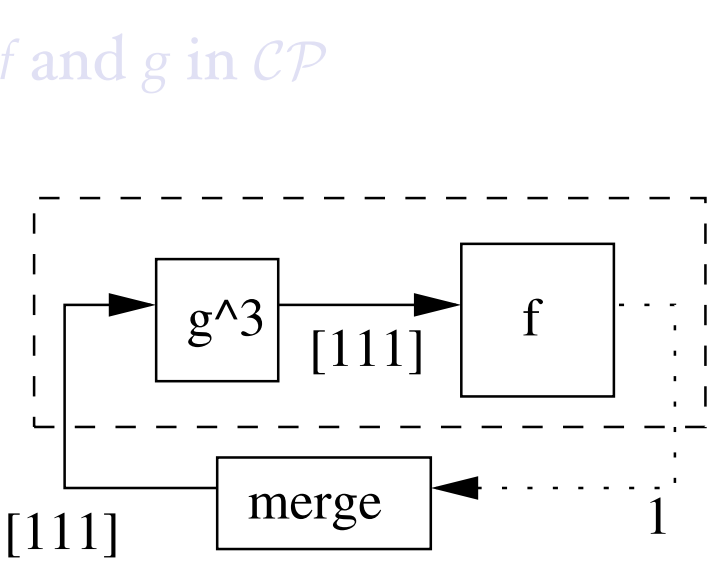
The burst : clock abstraction of micro-schedules

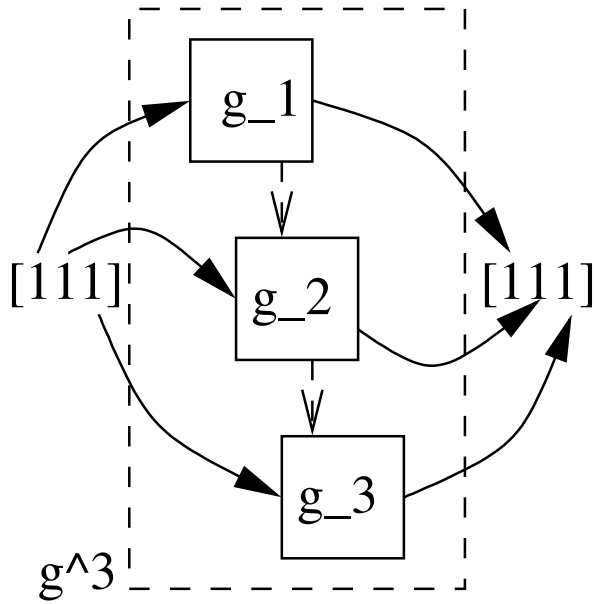
- Burst of tics inside an activation
- Activation as a logical time tic

$\mathcal{CP}$ clocks

- Boolean clocks extension
- Burst abstraction given by parenthesis
- Burst should be finite

f and g in $\mathcal{CP}$





Parenthesized boolean clocks

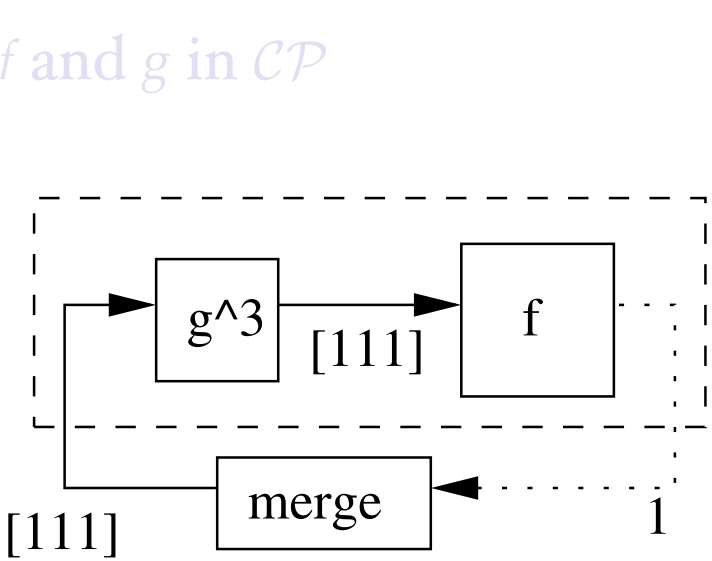
The burst : clock abstraction of micro-schedules

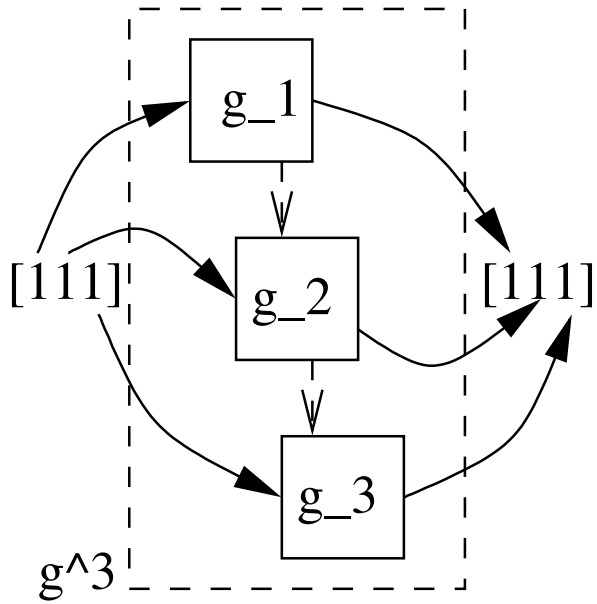
- Burst of tics inside an activation
- Activation as a logical time tic

$\mathcal{CP}$ clocks

- Boolean clocks extension
- Burst abstraction given by parenthesis
- Burst should be finite

f and g in $\mathcal{CP}$





Parenthesized boolean clocks

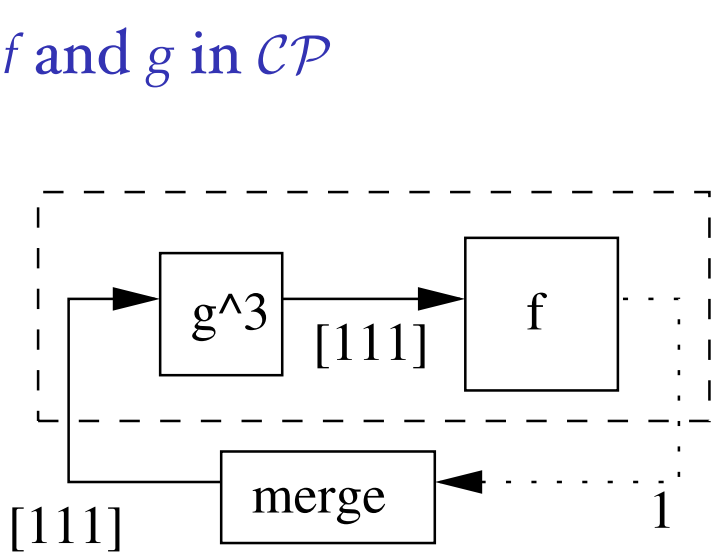
The burst : clock abstraction of micro-schedules

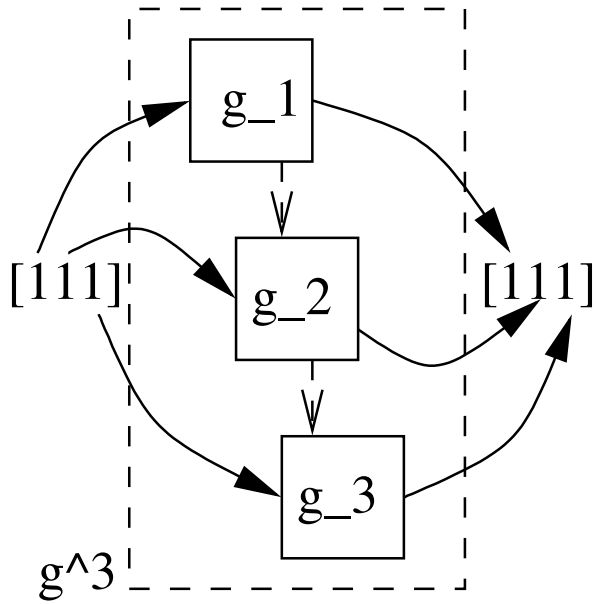
- Burst of tics inside an activation
- Activation as a logical time tic

$\mathcal{CP}$ clocks

- Boolean clocks extension
- Burst abstraction given by parenthesis
- Burst should be finite

f and g in $\mathcal{CP}$





Parenthesized boolean clocks

Newness

- Desynchronization through clock type
- Clocks give micro-schedules and synchronization points
- Generic clocks (no stationary state)
- Easy integration of boolean branching

Clock calculus, `on` operator

ck_1 ticks activate ck_2 :

ck_1	1	0	1	1		[1 1]	[1 0 1]	
ck_2	0		1	[1 0 1]	0 1		[1 1]	[1 1 1]
ck_1 on ck_2	0	0	1	[1 0 1]	[0 1]		[[1 1] 0	[1 1 1]]

Parenthesized boolean clocks

Newness

- Desynchronization through clock type
- Clocks give micro-schedules and synchronization points
- Generic clocks (no stationary state)
- Easy integration of boolean branching

Clock calculus, `on` operator

ck_1 ticks activate ck_2 :

ck_1	1	0	1	1		[1 1]	[1 0 1]	
ck_2	0		1	[1 0 1]	0 1		[1 1]	[1 1 1]
ck_1 on ck_2	0	0	1	[1 0 1]	[0 1]		[[1 1] 0	[1 1 1]]

Micro-schedule inobservability

Micro-schedule : local loop *atomically* executed.

- Guarantee coherence with grainy dependancy as in SDF
- Ensure modular and correct *Desynchronization*

Simplifications to burst and clocks

- Burst inside burst (parentheses inside parentheses)
- Absence inside burst (0s inside parentheses)

α a complete clock abstraction to $\mathcal{CN}$

ck	[100]	0	[10[11]]	[1[011]01]	0	[[10][10]]
$\alpha(ck)$	1	0	[111]	[1111]	0	[11]
$\alpha(ck)$	1	0	3	4	0	2

Micro-schedule inobservability

Micro-schedule : local loop *atomically* executed.

- Guarantee coherence with grainy dependancy as in SDF
- Ensure modular and correct *Desynchronization*

Simplifications to burst and clocks

- Burst inside burst (parentheses inside parentheses)
- Absence inside burst (0s inside parentheses)

α a complete clock abstraction to $\mathcal{CN}$

ck	[100]	0	[10[11]]	[1[011]01]	0	[[10][10]]
$\alpha(ck)$	1	0	[111]	[1111]	0	[11]
$\alpha(ck)$	1	0	3	4	0	2

Micro-schedule inobservability

Micro-schedule : local loop *atomically* executed.

- Guarantee coherence with grainy dependancy as in SDF
- Ensure modular and correct *Desynchronization*

Simplifications to burst and clocks

- Burst inside burst (parentheses inside parentheses)
- Absence inside burst (0s inside parentheses)

α a complete clock abstraction to $\mathcal{CN}$

ck	[100]	0	[10[11]]	[1[011]01]	0	[[10][10]]
$\alpha(ck)$	1	0	[111]	[1111]	0	[11]
$\alpha(ck)$	1	0	3	4	0	2

$\mathcal{CN}$ complete abstraction of $\mathcal{CP}$

Preservation of on operator

ck_1	1	0	1	1		[1 1]	[1 0 1]	
ck_2	0		1	[1 0 1]		0 1	[1 1]	[1 1 1]
ck_1 on ck_2	0	0	1	[1 0 1]		[0 1]	[[1 1] 0	[1 1 1]]

ck_1	1	0	1	1	2		2	
ck_2	0		1	2	0	1	2	3
ck_1 on ck_2	0	0	1	2	1		5	

Inductive definition of on

$$\forall g \in \mathcal{CN},\ h \in \mathcal{CN},\ (c_1, \ldots, c_g) \in \mathcal{CN}^g,\ d \in \mathcal{CN}$$

$$(g.h) \text{ on } (c_1 \ldots c_g.d) = \left(\sum_{i=1}^{g} c_i\right).(h \text{ on } d)$$

$\mathcal{CN}$ complete abstraction of $\mathcal{CP}$

Preservation of on operator

ck_1	1	0	1	1		[1 1]	[1 0 1]
ck_2	0		1	[1 0 1]	0 1	[1 1]	[1 1 1]
ck_1 on ck_2	0	0	1	[1 0 1]	[0 1]	[[1 1] 0	[1 1 1]]

ck_1	1	0	1	1	2	2
ck_2	0		1	2	0 1	2 3
ck_1 on ck_2	0	0	1	2	1	5

Inductive definition of on

$$\forall g \in \mathcal{CN},\ h \in \mathcal{CN},\ (c_1, \ldots, c_g) \in \mathcal{CN}^g,\ d \in \mathcal{CN}$$

$$(g.h) \text{ on } (c_1 \ldots c_g.d) = \left(\sum_{i=1}^{g} c_i\right).(h \text{ on } d)$$

$\mathcal{CN}$ complete abstraction of $\mathcal{CP}$

Preservation of `on` operator

ck_1	1	0	1	1		[1 1]	[1 0 1]
ck_2	0		1	[1 0 1]	0 1	[1 1]	[1 1 1]
ck_1 on ck_2	0	0	1	[1 0 1]	[0 1]	[[1 1] 0	[1 1 1]]

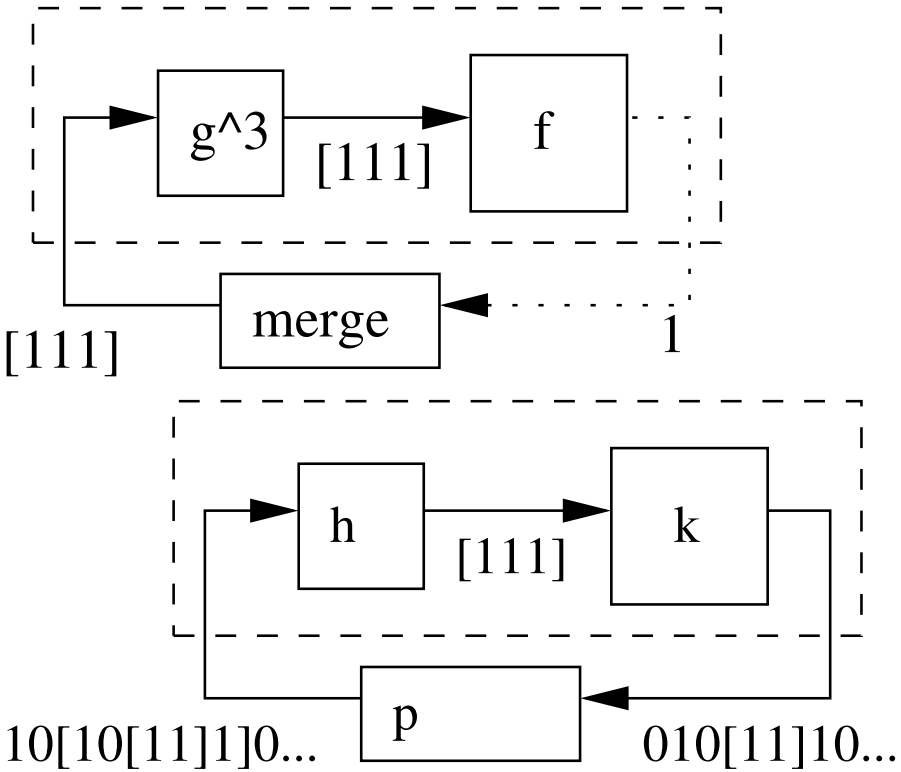
ck_1	1	0	1	1	2	2
ck_2	0		1	2	0 1	2 3
ck_1 on ck_2	0	0	1	2	1	5

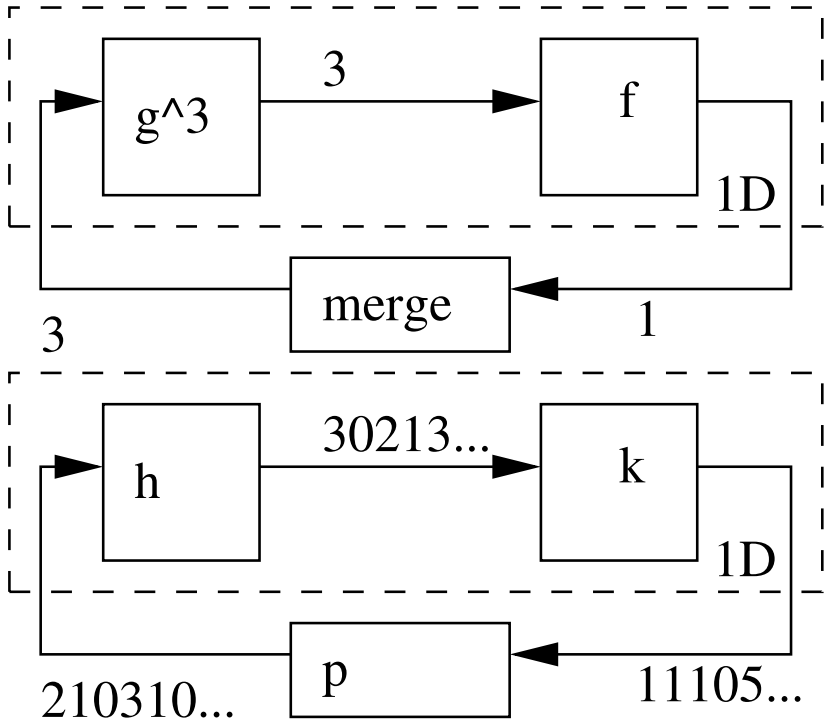
Inductive definition of `on`

$$\forall g \in \mathcal{CN},\ h \in \mathcal{CN},\ (c_1, \ldots, c_g) \in \mathcal{CN}^g,\ d \in \mathcal{CN}$$

$$(g.h) \text{ on } (c_1 \ldots c_g.d) = \left(\sum_{i=1}^{g} c_i\right).(h \text{ on } d)$$

Quick overview : balanced SDF and more





Quick overview : easy N-synchronous extension

Clocks use corollary

N-synchronous

"Asynchronous" composition system with automatic register and delay computation.

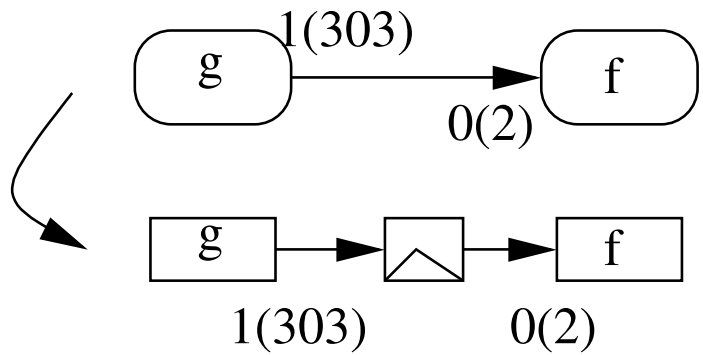
Synchronizability : $ck = v(w)$ compose with $ck' = v'(w')$ if

Original : binary periodic clocks : $\{0+1\}^n(\{0+1\}^m)$

$$|w|_1/|w| = |w'|_1/|w'|$$

Here : integer periodic clocks : $\{\mathbb{N}\}^n(\{\mathbb{N}\}^m)$

$$\sum_i w_i/|w| = \sum_i w'_i/|w'|$$



Quick overview : easy N-synchronous extension

Clocks use corollary

N-synchronous

"Asynchronous" composition system with automatic register and delay computation.

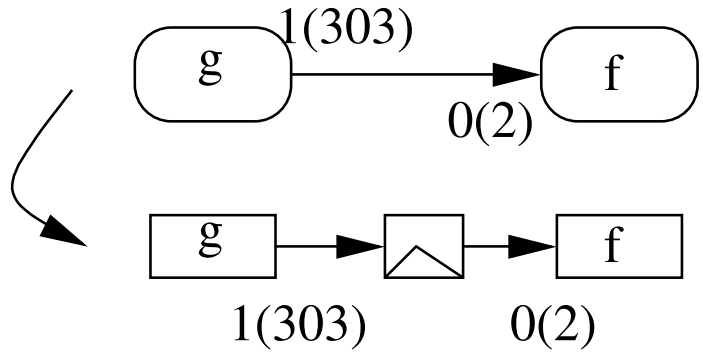
Synchronizability : $ck = v(w)$ compose with $ck' = v'(w')$ if

Original : binary periodic clocks : $\{0+1\}^n(\{0+1\}^m)$

$$|w|_1/|w| = |w'|_1/|w'|$$

Here : integer periodic clocks : $\{\mathbb{N}\}^n(\{\mathbb{N}\}^m)$

$$\sum_i w_i/|w| = \sum_i w'_i/|w'|$$



Quick overview : easy N-synchronous extension

Clocks use corollary

N-synchronous

"Asynchronous" composition system with automatic register and delay computation.

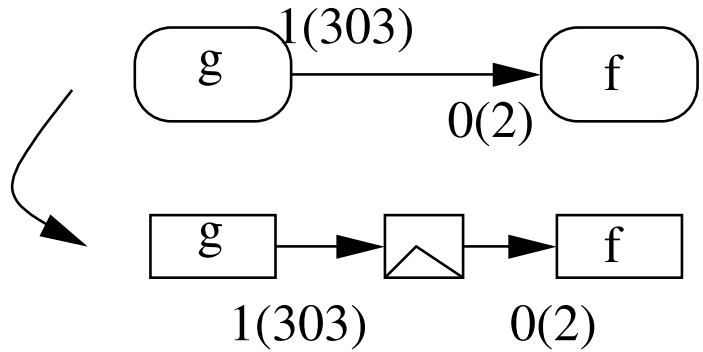
Synchronizability : $ck = v(w)$ compose with $ck' = v'(w')$ if

Original : binary periodic clocks : $\{0+1\}^n(\{0+1\}^m)$

$$|w|_1/|w| = |w'|_1/|w'|$$

Here : integer periodic clocks : $\{\mathbb{N}\}^n(\{\mathbb{N}\}^m)$

$$\sum_i w_i/|w| = \sum_i w'_i/|w'|$$



A bursty synchronous language

Conservative extension of classical operators

The `fby` operator in $\mathcal{CP}$

Stay a one slot register

ck	$[1\ 1]$	0	1	$[1\ 1\ 1]$	$[1\ 0\ 0\ 1]$
$x :: ck$	$[x_0\ x_1]$	.	x_2	$[x_3\ x_4\ x_5]$	$[x_6\ .\ .\ x_7]$
v `fby` $x :: ck$	$[v\ x_0]$	.	x_1	$[x_2\ x_3\ x_4]$	$[x_5\ .\ .\ x_6]$
m	x_1		x_2	x_5	x_7

The `when` operator in $\mathcal{CP}$

Convention

A boolean bursty stream = a clock in $\mathcal{CP}$

x `when` c is the point wise extension

Usual sampling

ck	1	0	1	[0 0 1 1]	1	[1 1 1 1]
$c :: ck$	t	.	f	[. . t f]	f	[t f t f]
$x :: ck$	x_1	.	x_2	[. . x_3 x_4]	x_5	[x_6 x_7 x_8 x_9]
x `when` c	x_1	.	.	[. . x_3 .]	.	[x_6 . x_8 .]

Typing

$$\texttt{when} :: ck \to (c :: ck) \to ck \texttt{ on } \texttt{unwrap}\,(c)$$

ck	1	0	1	[0 0 1 1]	1	[1 1 1 1]
$c :: ck$	t	.	f	[. . t f]	f	[t f t f]
`unwrap` (c)	t	.	f	. . t f	f	t f t f
ck `on unwrap` (c)	1	0	0	[0 0 1 0]	0	[1 0 1 0]

The `when` operator in $\mathcal{CP}$

Convention

A boolean bursty stream = a clock in $\mathcal{CP}$

x `when` c is the point wise extension

Usual sampling

ck	1	0	1	[0 0 1 1]	1	[1 1 1 1]
$c :: ck$	t	.	f	[. . t f]	f	[t f t f]
$x :: ck$	x_1	.	x_2	[. . x_3 x_4]	x_5	[x_6 x_7 x_8 x_9]
x `when` c	x_1	.	.	[. . x_3 .]	.	[x_6 . x_8 .]

Typing

$$\texttt{when} :: ck \to (c :: ck) \to ck \texttt{ on } \texttt{unwrap}\,(c)$$

ck	1	0	1	[0 0 1 1]	1	[1 1 1 1]
$c :: ck$	t	.	f	[. . t f]	f	[t f t f]
unwrap (c)	t	.	f	. . t f	f	t f t f
ck on unwrap (c)	1	0	0	[0 0 1 0]	0	[1 0 1 0]

The `when` operator in $\mathcal{CP}$

Convention

A boolean bursty stream = a clock in $\mathcal{CP}$

x `when` c is the point wise extension

Usual sampling

ck	1	0	1	[0 0 1 1]	1	[1 1 1 1]
$c :: ck$	t	.	f	[. . t f]	f	[t f t f]
$x :: ck$	x_1	.	x_2	[. . x_3 x_4]	x_5	[x_6 x_7 x_8 x_9]
x `when` c	x_1	.	.	[. . x_3 .]	.	[x_6 . x_8 .]

Typing

$$\texttt{when} :: ck \to (c :: ck) \to ck \texttt{ on unwrap } (c)$$

ck	1	0	1	[0 0 1 1]	1	[1 1 1 1]
$c :: ck$	t	.	f	[. . t f]	f	[t f t f]
`unwrap` (c)	t	.	f	. . t f	f	t f t f
ck `on unwrap` (c)	1	0	0	[0 0 1 0]	0	[1 0 1 0]

The `merge` operator in $\mathcal{CP}$

Usual exclusive streams merging

$c :: ck$	$\mathtt{f}$	$\mathtt{t}$	$[\mathtt{t}\ .\ \mathtt{t}]$	$[\mathtt{f}\ .\ [\mathtt{f}\ \mathtt{t}\ .\ \mathtt{f}]]$	$\mathtt{t}$	$\mathtt{f}$	$[\mathtt{t}\ \mathtt{f}]$
x	.	x_1	$[x_2\ .\ x_3]$	$[.\ .\ [.\ x_4\ .\ .]]$	x_5	.	$[x_6\ .]$
y	y_1	.	$[.\ .\ .]$	$[y_2\ .\ [y_3\ .\ .\ y_4]]$	.	y_5	$[.\ y_6]$
$merge\ c\ x\ y :: ck$	y_1	x_1	$[x_2\ .\ x_3]$	$[y_2\ .\ [y_3\ x_4\ .\ y_4]]$	x_5	y_5	$[x_6\ y_6]$

Typing

$$\mathtt{merge} :: (c :: ck) \to (ck\ \mathtt{dr}\ c) \to (ck\ \mathtt{dr}\ \mathtt{not}\ c) \to ck$$

With $\mathtt{dr} = \mathtt{on}\ \mathtt{unwrap}$

ck	1	1	[1 0 1]	[1 0 [1 1 0 1]]	1	1	[1 1]
$c :: ck$	$\mathtt{f}$	$\mathtt{t}$	$[\mathtt{t}\ .\ \mathtt{t}]$	$[\mathtt{f}\ .\ [\mathtt{f}\ \mathtt{t}\ .\ \mathtt{f}]]$	$\mathtt{t}$	$\mathtt{f}$	$[\mathtt{t}\ \mathtt{f}]$
$\mathtt{unwrap}\ (c)$	$\mathtt{f}$	$\mathtt{t}$	$\mathtt{t}\ .\ \mathtt{t}$	$\mathtt{f}\ .\ \mathtt{f}\ \mathtt{t}\ .\ \mathtt{f}$	$\mathtt{t}$	$\mathtt{f}$	$\mathtt{t}\ \mathtt{f}$
$ck\ \mathtt{dr}\ c$	0	1	[1 0 1]	[0 0 [0 1 0 0]]	1	0	[1 0]

The `merge` operator in $\mathcal{CP}$

Usual exclusive streams merging

$c :: ck$	$\mathtt{f}$	$\mathtt{t}$	$[\mathtt{t}\ .\ \mathtt{t}]$	$[\mathtt{f}\ .\ [\mathtt{f}\ \mathtt{t}\ .\ \mathtt{f}]]$	$\mathtt{t}$	$\mathtt{f}$	$[\mathtt{t}\ \mathtt{f}]$
x	.	x_1	$[x_2\ .\ x_3]$	$[.\ .\ [.\ x_4\ .\ .]]$	x_5	.	$[x_6\ .]$
y	y_1	.	$[.\ .\ .]$	$[y_2\ .\ [y_3\ .\ .\ y_4]]$	.	y_5	$[.\ y_6]$
$merge\ c\ x\ y :: ck$	y_1	x_1	$[x_2\ .\ x_3]$	$[y_2\ .\ [y_3\ x_4\ .\ y_4]]$	x_5	y_5	$[x_6\ y_6]$

Typing

$$\mathtt{merge} :: (c :: ck) \to (ck\ \mathtt{dr}\ c) \to (ck\ \mathtt{dr}\ \mathtt{not}\ c) \to ck$$

With $\mathtt{dr} = \mathtt{on}\ \mathtt{unwrap}$

ck	1	1	[1 0 1]	[1 0 [1 1 0 1]]	1	1	[1 1]
$c :: ck$	$\mathtt{f}$	$\mathtt{t}$	$[\mathtt{t}\ .\ \mathtt{t}]$	$[\mathtt{f}\ .\ [\mathtt{f}\ \mathtt{t}\ .\ \mathtt{f}]]$	$\mathtt{t}$	$\mathtt{f}$	$[\mathtt{t}\ \mathtt{f}]$
$\mathtt{unwrap}\ (c)$	$\mathtt{f}$	$\mathtt{t}$	$\mathtt{t}\ .\ \mathtt{t}$	$\mathtt{f}\ .\ \mathtt{f}\ \mathtt{t}\ .\ \mathtt{f}$	$\mathtt{t}$	$\mathtt{f}$	$\mathtt{t}\ \mathtt{f}$
$ck\ \mathtt{dr}\ c$	0	1	[1 0 1]	[0 0 [0 1 0 0]]	1	0	[1 0]

The `when` operator in $\mathcal{CN}$

The `when` operator

ck	2	3	0
$x :: ck$	$[x_1\ x_2]$	$[x_3\ x_4\ x_5]$	.
$c :: ck$	$[t\ f]$	$[t\ f\ t]$	.
$ck\ \texttt{dr}\ c$	1	2	0
$x\ \texttt{when}\ c :: ck\ \texttt{dr}\ c$	x_1	$[x_3\ x_5]$	.

Abstraction α

When $c :: ck$, $ck\ \texttt{dr}\ c = \alpha(c)$, so that

$$\texttt{when} :: ck \to (c :: ck) \to \alpha(c)$$

Possible extension to integers

ck	1	3	1
$x :: ck$	x_3	$[x_4\ x_5\ x_6]$	x_7
$c :: ck$	2	[1 2 0]	3
$ck\ \texttt{dr}\ c$	2	3	3
$x\ \texttt{when}\ c :: ck\ \texttt{dr}\ c$	$[x_3\ x_3]$	$[x_4\ x_5\ x_5]$	$[x_7\ x_7\ x_7]$

The `when` operator in $\mathcal{CN}$

The `when` operator

ck	2	3	0
$x :: ck$	$[x_1\ x_2]$	$[x_3\ x_4\ x_5]$	.
$c :: ck$	$[t\ f]$	$[t\ f\ t]$	.
$ck\ \texttt{dr}\ c$	1	2	0
$x\ \texttt{when}\ c :: ck\ \texttt{dr}\ c$	x_1	$[x_3\ x_5]$	.

Abstraction α

When $c :: ck$, $ck\ \texttt{dr}\ c = \alpha(c)$, so that

$$\texttt{when} :: ck \to (c :: ck) \to \alpha(c)$$

Possible extension to integers

ck	1	3	1
$x :: ck$	x_3	$[x_4\ x_5\ x_6]$	x_7
$c :: ck$	2	[1 2 0]	3
$ck\ \texttt{dr}\ c$	2	3	3
$x\ \texttt{when}\ c :: ck\ \texttt{dr}\ c$	$[x_3\ x_3]$	$[x_4\ x_5\ x_5]$	$[x_7\ x_7\ x_7]$

The `when` operator in $\mathcal{CN}$

The `when` operator

ck	2	3	0
$x :: ck$	$[x_1\ x_2]$	$[x_3\ x_4\ x_5]$	.
$c :: ck$	$[t\ f]$	$[t\ f\ t]$	.
$ck\,\texttt{dr}\,c$	1	2	0
$x\,\texttt{when}\,c :: ck\,\texttt{dr}\,c$	x_1	$[x_3\ x_5]$	.

Abstraction α

When $c :: ck$, $ck\,\texttt{dr}\,c = \alpha(c)$, so that

$$\texttt{when} :: ck \to (c :: ck) \to \alpha(c)$$

Possible extension to integers

ck	1	3	1
$x :: ck$	x_3	$[x_4\ x_5\ x_6]$	x_7
$c :: ck$	2	[1 2 0]	3
$ck\,\texttt{dr}\,c$	2	3	3
$x\,\texttt{when}\,c :: ck\,\texttt{dr}\,c$	$[x_3\ x_3]$	$[x_4\ x_5\ x_5]$	$[x_7\ x_7\ x_7]$

The `merge` operator in $\mathcal{CN}$

The `merge` operator

$c :: ck$	`f`	`t`	`[t t]`	`[f f t f]`	`t`	`f`	`[t f]`
ck `dr` c	0	1	2	1	1	0	1
$x :: ck$ `dr` c	.	x_1	$[x_2\ x_3]$	x_4	x_5	.	x_6
`not` $c :: ck$	`t`	`f`	`[f f]`	`[t t f t]`	`f`	`t`	`[f t]`
ck `dr not` c	1	0	0	3	0	1	1
$y :: ck$ `dr not` c	y_1	.	.	$[y_2\ y_3\ y_4]$	.	y_5	y_6
merge $c\ x\ y$	y_1	x_1	$[x_2\ x_3]$	$[y_2\ y_3\ x_4\ y_4]$	x_5	y_5	$[x_6\ y_6]$

The new operator `by`

`x by 3` gives bursts of size 3

$x :: ck$	x_1	x_2	x_3	$[x_4\ x_5]$	x_6	$[x_7\ x_8\ x_9]$
$x \texttt{ by } 3 :: ck_r$	.	.	$[x_1\ x_2\ x_3]$	.	$[x_4\ x_5\ x_6]$	$[x_7\ x_8\ x_9]$
$3 :: ck_n$	3	.	.	3	.	3
$ck_n = ck \texttt{ on } (10^{n_i-1})_{i\in\mathbb{N}}$	1	0	0	1	0	3
$ck_r = ck \texttt{ on } (0^{n_i-1}n_i)_{i\in\mathbb{N}}$	0	0	3	0	3	3

Complex example

ck	2	1	3	0	2
$x :: ck$	$[x_1\ x_2]$	x_3	$[x_4\ x_5\ x_6]$	.	$[x_7\ x_8]$
$n :: ck_n$	2	3	2	.	1
$x \texttt{ by } n :: ck_r$	$[x_1\ x_2]$	.	$[x_3\ x_4\ x_5]$	.	$[x_6\ x_7\ x_8]$
$ck'_n = (10^{n_i-1})_{i\in\mathbb{N}}$	1 0	1	0 0 1		0 1
$ck_n = ck \texttt{ on } ck'_n$	1	1	1	0	1
$ck'_r = (0^{n_i-1}n_i)_{i\in\mathbb{N}}$	0 2	0	0 3 0		2 1
$ck_r = ck \texttt{ on } ck'_r$	2	0	3	0	3

The new operator by

x by 3 gives bursts of size 3

$x :: ck$	x_1	x_2	x_3	$[x_4\ x_5]$	x_6	$[x_7\ x_8\ x_9]$
x by $3 :: ck_r$	.	.	$[x_1\ x_2\ x_3]$	.	$[x_4\ x_5\ x_6]$	$[x_7\ x_8\ x_9]$
$3 :: ck_n$	3	.	.	3	.	3
$ck_n = ck$ on $(10^{n_i-1})_{i\in\mathbb{N}}$	1	0	0	1	0	3
$ck_r = ck$ on $(0^{n_i-1}n_i)_{i\in\mathbb{N}}$	0	0	3	0	3	3

Complex example

ck	2	1	3	0	2
$x :: ck$	$[x_1\ x_2]$	x_3	$[x_4\ x_5\ x_6]$	.	$[x_7\ x_8]$
$n :: ck_n$	2	3	2	.	1
x by $n :: ck_r$	$[x_1\ x_2]$	.	$[x_3\ x_4\ x_5]$	.	$[[x_6\ x_7]\ x_8]$
$ck'_n = (10^{n_i-1})_{i\in\mathbb{N}}$	1 0	1	0 0 1		0 1
$ck_n = ck$ on ck'_n	1	1	1	0	1
$ck'_r = (0^{n_i-1}n_i)_{i\in\mathbb{N}}$	0 2	0	0 3 0		2 1
$ck_r = ck$ on ck'_r	2	0	3	0	3

Array encoding in $\mathcal{CN}$

```
let f x = z where
  x1 = x when ([1000])
  and x2 = x when ([0100])
  and x3 = x when ([0010])
  and x4 = x when ([0001])
  and z1 = x1 + x3
  and z2 = x2 + x4
  and c = 1 -> not c
  and y = merge (c by 2) z1 z2
```

Types and computation

ck_x	4	4
$x :: ck_x$	$[x_1\ x_2\ x_3\ x_4]$	$[x_5\ x_6\ x_7\ x_8]$
ck_x dr ([0100])	1	1
$x2 :: ck_x$ on ([0100])	x_2	x_6
$z1$	$x_1 + x_3$	$x_5 + x_7$
c by 2	[1 0]	[1 0]
y	$[(x_1 + x_3)\ (x_2 + x_4)]$	$[(x_5 + x_7)\ (x_6 + x_9)]$

Conclusion

- Very powerful and generic
- High level semantic
- Desynchronization

In the near future

- Finish type inference
- Compilation with loops

Conclusion

- Very powerful and generic
- High level semantic
- Desynchronization

In the near future

- Finish type inference
- Compilation with loops