Parallel Computing with the π-calculus

Frédéric Peschanski

UPMC Paris – LIP6 – RSR – Team APR

LRDE Seminar – February 2011

Outline

1. Motivations
2. The intermediate language
3. Scheduling principles
4. Garbage collection
5. Conclusion

Outline

1. Motivations
2. The intermediate language
3. Scheduling principles
4. Garbage collection
5. Conclusion

About concurrency

Why (again) ?

The Free Lunch Is Over : A Fundamental Turn Toward Concurrency in Software

Herb Sutter, Dr Dobb's Mai 2005

About concurrency

Why (again) ?

The Free Lunch Is Over : A Fundamental Turn Toward Concurrency in Software Herb Sutter, Dr Dobb's Mai 2005

- The advent of multi-core techs brings back concurrency as a central concern
- with (some) novelties :
  - new *cheap* parallel hardware
  - new techniques : *lock-free/wait-free* algorithms, transactional memories, etc.
  - and a matured **concurrency theory** : process algebras, P/T nets, etc.

Splendor and misery of the π-calculus

Splendor

- A **minimal** language
  to describe concurrent and dynamic systems
- a very **expressive** language
- A (too ?) large body of **theoretical works**

Splendor and misery of the π-calculus

Splendor

- A **minimal** language
  to describe concurrent and dynamic systems
- a very **expressive** language
- A (too ?) large body of **theoretical works**

Misery

- Many variants (early, late, open, asynchronous, etc.)
- Lack of modelling/verification tools
- Lack of implementations (stable, faithful, efficient, etc.)

From λ to π

λ-calculus
$\Rightarrow$ functions

π-calculus
$\Rightarrow$ processes, channels

From λ to π

λ-calculus	π-calculus
$\Rightarrow$ functions	$\Rightarrow$ processes, channels
Abstract machine	Abstract machine
$\Rightarrow$ SECD, ZAM, etc.	$\Rightarrow$ π-threads

From λ to π

λ-calculus	π-calculus
$\Rightarrow$ functions	$\Rightarrow$ processes, channels
Abstract machine	Abstract machine
$\Rightarrow$ SECD, ZAM, etc.	$\Rightarrow$ π-threads
Virtual machine	Virtual machine
$\Rightarrow$ Zinc, etc.	$\Rightarrow$ PCM(Parallel Commitment Machine)

From λ to π

λ-calculus	π-calculus
$\Rightarrow$ functions	$\Rightarrow$ processes, channels
Abstract machine	Abstract machine
$\Rightarrow$ SECD, ZAM, etc.	$\Rightarrow$ π-threads
Virtual machine	Virtual machine
$\Rightarrow$ Zinc, etc.	$\Rightarrow$ PCM(Parallel Commitment Machine)
Compiler	Compiler
$\Rightarrow$ Ocaml, etc.	$\Rightarrow$ TODO ...

From λ to π

λ-calculus	π-calculus
$\Rightarrow$ functions	$\Rightarrow$ processes, channels
Abstract machine	Abstract machine
$\Rightarrow$ SECD, ZAM, etc.	$\Rightarrow$ π-threads
Virtual machine	Virtual machine
$\Rightarrow$ Zinc, etc.	$\Rightarrow$ PCM(Parallel Commitment Machine)
Compiler	Compiler
$\Rightarrow$ Ocaml, etc.	$\Rightarrow$ TODO ...

Goal : an abstract machine for π and its implementation(s)

- **Faithful** to (a variant of) the π-calculus
- **Efficient** parallel implementation (scheduling, GC, etc.)

Outline

1 Motivations

2 The intermediate language

3 Scheduling principles

4 Garbage collection

5 Conclusion

The language : syntax

Definition **def** $D(x_1,\ldots,x_n) = P$

Process	P	$::=$	**end**	Termination
			$\mid \sum_i [g_i]\alpha_i, P_i$	Guarded choice
			$\mid D(v_1,\ldots,v_n)$	Tail call
Prefix	α	$::=$	**tau**	Internal step
			$\mid c\,!v$	Emission
			$\mid c\,?(x)$	Reception
			$\mid$ **new**(c)	Channel creation
			$\mid$ **spawn**$\{P\}$	Thread creation

The language : syntax

Definition **def** $D(x_1,\ldots,x_n) = P$

Process	P	$::=$		
			end	Termination
			$\mid \sum_i [g_i]\alpha_i, P_i$	Guarded choice
			$\mid D(v_1,\ldots,v_n)$	Tail call

Prefix	α	$::=$		
			tau	Internal step
			$\mid c\,!v$	Emission
			$\mid c\,?(x)$	Reception
			$\mid$ **new**(c)	Channel creation
			$\mid$ **spawn**$\{P\}$	Thread creation

+ derived forms : **if-then-else**, parallel operator $||$, etc.

The language : syntax

Definition **def** $D(x_1,\ldots,x_n) = P$

Process

P	$::=$	**end**	Termination
		$\mid \sum_i [g_i]\alpha_i, P_i$	Guarded choice
		$\mid D(v_1,\ldots,v_n)$	Tail call

Prefix

α	$::=$	**tau**	Internal step
		$\mid c\,!v$	Emission
		$\mid c\,?(x)$	Reception
		$\mid$ **new**(c)	Channel creation
		$\mid$ **spawn**$\{P\}$	Thread creation

+ derived forms : **if-then-else**, parallel operator $||$, etc.

+ type system

The language : syntax

Definition **def** $D(x_1,\ldots,x_n) = P$

Process

P	$::=$	**end**	Termination
		$\mid \sum_i [g_i]\alpha_i, P_i$	Guarded choice
		$\mid D(v_1,\ldots,v_n)$	Tail call

Prefix

α	$::=$	**tau**	Internal step
		$\mid c\,!v$	Emission
		$\mid c\,?(x)$	Reception
		$\mid$ **new**(c)	Channel creation
		$\mid$ **spawn**$\{P\}$	Thread creation

+ derived forms : **if-then-else**, parallel operator $||$, etc.

+ type system

+ evaluator for expressions of basic types (bool, int, etc.)

Example : tail recursion

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
new(c :chan<int>), spawn{Fib(5,0,1,c)},c?(n),print(n)
```

Example : tail recursion

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
new(c :chan<int>), spawn{Fib(5,0,1,c)},c?(n),print(n)
```

Example : tail recursion

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
spawn{Fib(5,0,1,ĉ)},ĉ?(n),print(n)
```

Example : tail recursion

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

---

spawn{Fib(5,0,1,$\hat{c}$)},$\hat{c}$?(n),print(n)

Example : tail recursion

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
Fib(5,0,1,ĉ),
|| ĉ?(n),print(n)
```

Example : tail recursion

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
[5=0] ĉ!0,end + [true] tau,Fib(4,1,0,ĉ)
|| ĉ?(n),print(n)
```

Example : tail recursion

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
[5=0] ĉ!0,end + [true] tau,Fib(4,1,0,ĉ)
|| ĉ?(n),print(n)
```

Example : tail recursion

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
Fib(4,1,0,ĉ)
|| ĉ?(n),print(n)
```

Example : tail recursion

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
[4=0] ĉ!1,end + [true] tau,Fib(3,1,1,ĉ)
|| ĉ?(n),print(n)
```

Example : tail recursion

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
Fib(3,1,1,ĉ)
|| ĉ?(n),print(n)
```

Example : tail recursion

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
Fib(2,2,1,ĉ)
|| ĉ?(n),print(n)
```

Example : tail recursion

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
Fib(1,3,2,ĉ)
|| ĉ?(n),print(n)
```

Example : tail recursion

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
Fib(0,5,3,ĉ)
|| ĉ?(n),print(n)
```

Example : tail recursion

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
[0=0] ĉ!5,end + [true] tau,Fib(-1,8,5,ĉ)
|| ĉ?(n),print(n)
```

Example : tail recursion

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
[0=0] ĉ!5,end + [true] tau,Fib(-1,8,5,ĉ)
|| ĉ?(n),print(n)
```

Example : tail recursion

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)

---

end
|| print(5)
```

Non-tail recursion : encoding

```
def Ack(n p : int, r : chan<int>) =
  [n=0] r!(p+1)
  + [p=0] Ack(n-1,1,r)
  + new(r1 :chan<int>), [ r1?(pp),Ack(n-1,pp,r) || Ack(n,p-1,r1) ]
```

Non-tail recursion : encoding

```
def Ack(n p : int, r : chan<int>) =
  [n=0] r!(p+1)
  + [p=0] Ack(n-1,1,r)
  + new(r1 :chan<int>), [ r1?(pp),Ack(n-1,pp,r) || Ack(n,p-1,r1) ]
```

Inefficient ?

Non-tail recursion : encoding

```
def Ack(n p : int, r : chan<int>) =
  [n=0] r!(p+1)
  + [p=0] Ack(n-1,1,r)
  + new(r1 :chan<int>), [ r1?(pp),Ack(n-1,pp,r) || Ack(n,p-1,r1) ]
```

Inefficient ? not so sure (“chain reactions”, linear channels . . .)

A concurrent example

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
                || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\mathsf{TaskPool}(2,\hat{e},\hat{l}) \parallel \mathsf{Task}(\hat{e}) \parallel \mathsf{Task}(\hat{e}) \parallel \mathsf{Task}(\hat{e}) \parallel \mathsf{Task}(\hat{e})$

A concurrent example

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
           || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{l}$?, TaskPool(3,$\hat{e}$,$\hat{l}$)
+ [2>0] $\hat{e}$?(release), Permit(release,$\hat{l}$) || TaskPool(1,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), */* task behavior */*, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), */* task behavior */*, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), */* task behavior */*, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), */* task behavior */*, rel !

A concurrent example

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
           || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{l}$?, TaskPool(3,$\hat{e}$,$\hat{l}$)
+ [2>0] $\hat{e}$?(release), Permit(release,$\hat{l}$) || TaskPool(1,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), */* task behavior */*, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), */* task behavior */*, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), */* task behavior */*, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), */* task behavior */*, rel !

A concurrent example

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
              || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{l}$?, TaskPool(3,$\hat{e}$,$\hat{l}$)
+ [2>0] $\hat{e}$?(release), Permit(release,$\hat{l}$) || TaskPool(1,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !
|| $\hat{e}$!($\hat{r1}$), /* task behavior */, $\hat{r1}$!
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !

A concurrent example

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
           || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{l}$?, TaskPool(3,$\hat{e}$,$\hat{l}$)
+ [2>0] $\hat{e}$?(release), Permit(release,$\hat{l}$) || TaskPool(1,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !
|| $\hat{e}$!($\hat{r1}$), /* task behavior */, $\hat{r1}$!
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !

A concurrent example

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
              || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\mathsf{Permit}(\hat{r1},\hat{l}) \parallel \mathsf{TaskPool}(1,\hat{e},\hat{l})$
$\parallel$ **new**(rel :chan<>),$\hat{e}$!(rel), */* task behavior */*, rel !
$\parallel$ */* task behavior */*, $\hat{r1}$!
$\parallel$ **new**(rel :chan<>),$\hat{e}$!(rel), */* task behavior */*, rel !
$\parallel$ **new**(rel :chan<>),$\hat{e}$!(rel), */* task behavior */*, rel !

A concurrent example

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
                 || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}$?,$\hat{l}$! || TaskPool(1,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), ** task behavior **/, rel !
|| ** task behavior **/, $\hat{r1}$!
|| **new**(rel :chan<>),$\hat{e}$!(rel), ** task behavior **/, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), ** task behavior **/, rel !

A concurrent example

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
              || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}$?,$\hat{l}$! || $\hat{r2}$?,$\hat{l}$! || TaskPool(0,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), ** task behavior **/, rel !
|| /* task behavior */, $\hat{r1}$!
|| **new**(rel :chan<>),$\hat{e}$!(rel), ** task behavior **/, rel !
|| /* task behavior */, $\hat{r2}$!

A concurrent example

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
              || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}$?,$\hat{l}$! || $\hat{r2}$?,$\hat{l}$! || $\hat{l}$?, TaskPool(1,$\hat{e}$,$\hat{l}$) + [0>0] ...
|| **new**(rel :chan<>),$\hat{e}$!(rel), ** task behavior */*, rel !
|| ** task behavior */*, $\hat{r1}$!
|| **new**(rel :chan<>),$\hat{e}$!(rel), ** task behavior */*, rel !
|| ** task behavior */*, $\hat{r2}$!

A concurrent example

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
              || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}\,?,\hat{l}\,!\;||\;\hat{r2}\,?,\hat{l}\,!\;||\;\hat{l}\,?,\;\mathrm{TaskPool}(1,\hat{e},\hat{l})\;+\;[0>0]\;\ldots$
$||\;\hat{e}\,!(\hat{r3}),\;/*\;\textit{task behavior}\;*/,\;\hat{r3}\,!$
$||\;\boxed{/*\;\textit{task behavior}\;*/},\;\hat{r1}\,!$
$||\;\mathbf{new}(\mathrm{rel}:\mathrm{chan}<>),\hat{e}\,!(\mathrm{rel}),\;/*\;\textit{task behavior}\;*/,\;\mathrm{rel}\,!$
$||\;\boxed{/*\;\textit{task behavior}\;*/},\;\hat{r2}\,!$

A concurrent example

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
                || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}\,?,\hat{l}\,!\;||\;\hat{r2}\,?,\hat{l}\,!\;||\;\hat{l}\,?,\;\mathrm{TaskPool}(1,\hat{e},\hat{l})\;+\;[0>0]\;\ldots$
$||\;\hat{e}\,!(\hat{r3}),\;/*\;\textit{task behavior}\;*/,\;\hat{r3}\,!$
$||\;\boxed{/*\;\textit{task behavior}\;*/},\;\hat{r1}\,!$
$||\;\hat{e}\,!(\hat{r4}),\;/*\;\textit{task behavior}\;*/,\;\hat{r4}\,!$
$||\;\boxed{/*\;\textit{task behavior}\;*/},\;\hat{r2}\,!$

A concurrent example

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
              || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}\,?,\hat{l}\,! \parallel \underline{\hat{r2}\,?},\hat{l}\,! \parallel \hat{l}\,?,\ \mathrm{TaskPool}(1,\hat{e},\hat{l}) + [0>0]\ \ldots$
$\parallel \hat{e}\,!(\hat{r3}),\ /*\ \textit{task behavior}\ */,\ \hat{r3}\,!$
$\parallel \boxed{/*\ \textit{task behavior}\ */},\ \hat{r1}\,!$
$\parallel \hat{e}\,!(\hat{r4}),\ /*\ \textit{task behavior}\ */,\ \hat{r4}\,!$
$\parallel \underline{\hat{r2}\,!}$

A concurrent example

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
              || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}\,?,\hat{l}\,! \parallel \underline{\hat{l}\,!} \parallel \underline{\hat{l}\,?}$, TaskPool$(1,\hat{e},\hat{l})$ + [0>0] ...
$\parallel \hat{e}\,!(\hat{r3})$, /* task behavior */, $\hat{r3}\,!$
$\parallel$ /* task behavior */, $\hat{r1}\,!$
$\parallel \hat{e}\,!(\hat{r4})$, /* task behavior */, $\hat{r4}\,!$

A concurrent example

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
              || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}\,?,\hat{l}\,!$ || TaskPool(1,$\hat{e}$,$\hat{l}$)
|| $\hat{e}\,!(\hat{r3})$, *task behavior*, $\hat{r3}\,!$
|| *task behavior*, $\hat{r1}\,!$
|| $\hat{e}\,!(\hat{r4})$, *task behavior*, $\hat{r4}\,!$

A concurrent example

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
              || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}\,?,\hat{l}\,!\;\|\;\hat{r4}\,?,\hat{l}\,!\;\|\;\text{TaskPool}(0,\hat{e},\hat{l})$
$\|\;\hat{e}\,!(\hat{r3}),\ /*\ \textit{task behavior}\ */,\ \hat{r3}\,!$
$\|\;\boxed{/*\ \textit{task behavior}\ */},\ \hat{r1}\,!$
$\|\;\boxed{/*\ \textit{task behavior}\ */},\ \hat{r4}\,!$

A concurrent example

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
                 || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}\,?,\hat{l}\,! \parallel \hat{r4}\,?,\hat{l}\,! \parallel \text{TaskPool}(0,\hat{e},\hat{l})$
$\parallel \hat{e}\,!(\hat{r3}),\ /*\ task\ behavior\ */,\ \hat{r3}\,!$
$\parallel$ /* task behavior */ , $\hat{r1}\,!$
$\parallel$ /* task behavior */ , $\hat{r4}\,!$

etc ...

Outline

1 Motivations

2 The intermediate language

3 Scheduling principles

4 Garbage collection

5 Conclusion

The π-threads and the PCM

Goals

- Faithfulness (e.g. : no + in Pict)
- Efficiency (CML, GHC, ...)
- Parallelism (GPH, Erlang, ...)

The π-threads and the PCM

Goals

- Faithfulness (e.g. : no + in Pict)
- Efficiency (CML, GHC, ...)
- Parallelism (GPH, Erlang, ...)

The π-threads (abstract machine) and the PCM (virtual machine)

- Stackless architecture
- Fast, decentralized scheduling
- Parallel GC

Concurrency and the stack

Common approaches

heavyweight 1 stack per thread (ex. Java threads)

lightweight 1 stack for all threads (ex. Go threads)

Concurrency and the stack

Common approaches

heavyweight 1 stack per thread (ex. Java threads)

- ... but *heavyweight* indeed

lightweight 1 stack for all threads (ex. Go threads)

Concurrency and the stack

Common approaches

heavyweight 1 stack per thread (ex. Java threads)

- ... but *heavyweight* indeed

lightweight 1 stack for all threads (ex. Go threads)

- ... but (very) complex (splitting/migration, etc.)

Concurrency and the stack

Common approaches

heavyweight 1 stack per thread (ex. Java threads)

- ... but *heavyweight* indeed

lightweight 1 stack for all threads (ex. Go threads)

- ... but (very) complex (splitting/migration, etc.)

Our approach : stackless !

Concurrency and the stack

Common approaches

heavyweight 1 stack per thread (ex. Java threads)

- ... but *heavyweight* indeed

lightweight 1 stack for all threads (ex. Go threads)

- ... but (very) complex (splitting/migration, etc.)

Our approach : stackless !

By the way . . . A stack for what ?

Concurrency and the stack

Common approaches

heavyweight 1 stack per thread (ex. Java threads)

- ... but *heavyweight* indeed

lightweight 1 stack for all threads (ex. Go threads)

- ... but (very) complex (splitting/migration, etc.)

Our approach : stackless !

By the way . . . A stack for what ?

- non-tail calls
- unbounded, tree-structured lexical environments

Concurrency and the stack

Common approaches

heavyweight 1 stack per thread (ex. Java threads)

- ... but *heavyweight* indeed

lightweight 1 stack for all threads (ex. Go threads)

- ... but (very) complex (splitting/migration, etc.)

Our approach : stackless !

By the way . . . A stack for what ?

- non-tail calls
- unbounded, tree-structured lexical environments

$\Rightarrow$ none of these in the π-threads

Structure of a π-thread

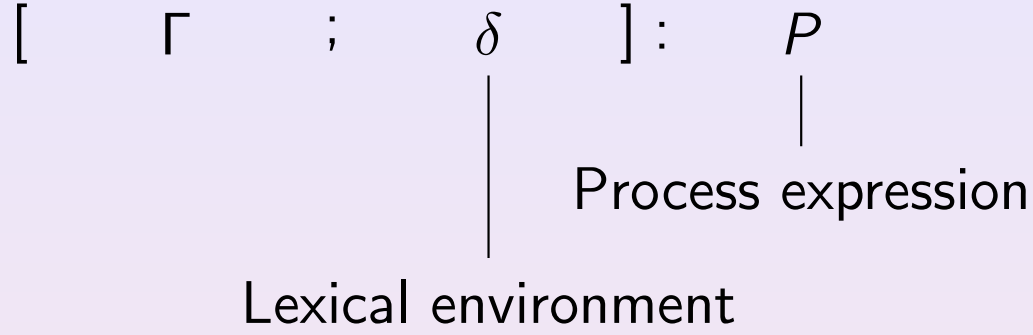
$$[\quad \Gamma \quad ; \quad \delta \quad] : \quad P$$

Structure of a π-thread

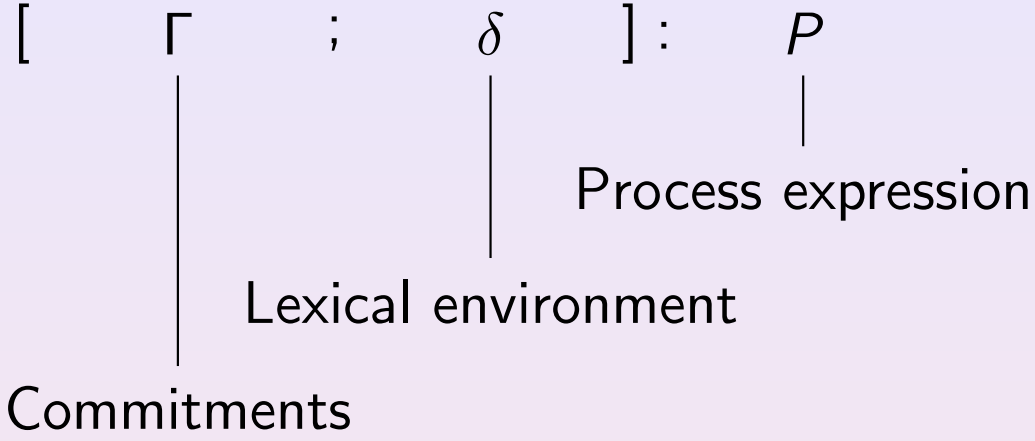
$$[\quad \Gamma \quad ; \quad \delta \quad] : \quad P$$

Process expression

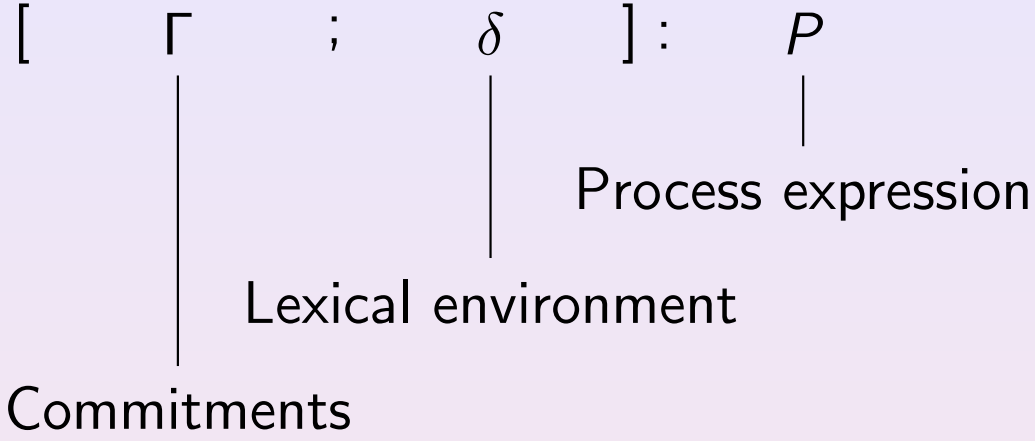
Structure of a π-thread



Structure of a π-thread

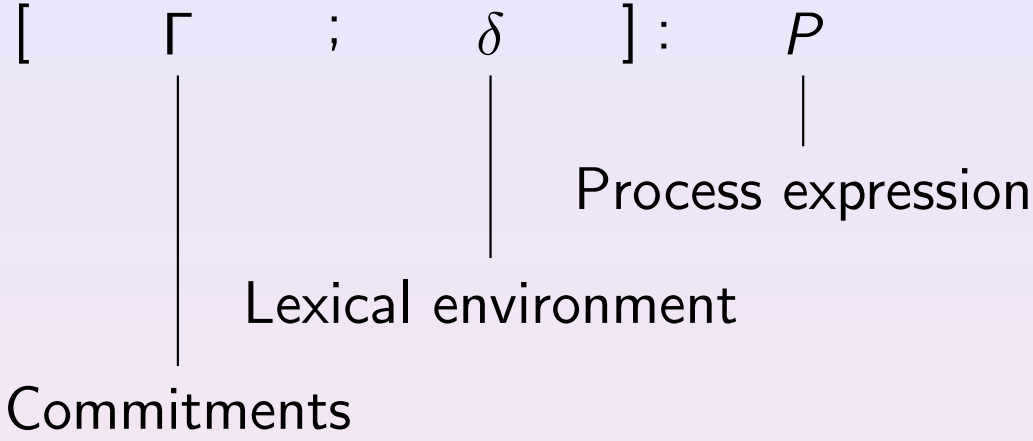


Structure of a π-thread



Remark : "Flat" Lex. env. (+ tail calls) $\Longrightarrow$ stackless [VEE06]

Structure of a π-thread



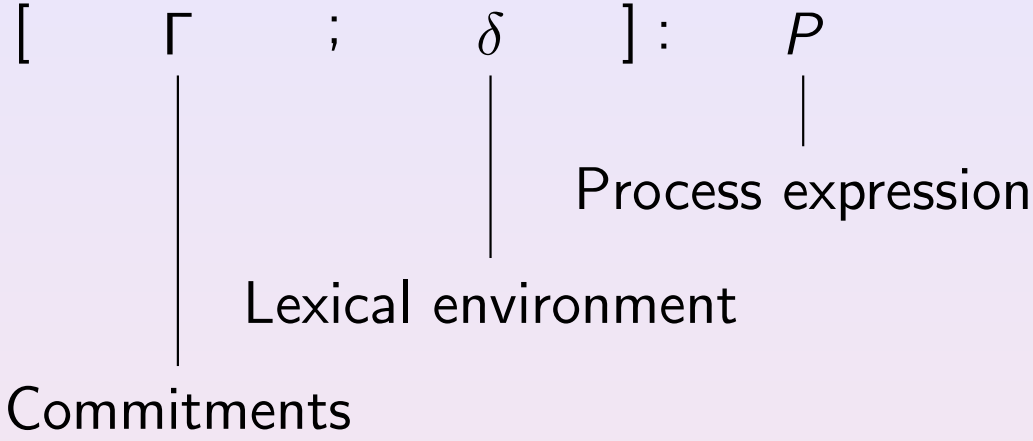
Remark : "Flat" Lex. env. (+ tail calls) $\Longrightarrow$ stackless [VEE06]

Commitments

Emission $\hat{c} \Leftarrow v : Q$ channel $\hat{c}$ value v (or expression $e \approx \mathit{lazy}$) and continuation Q

Reception $\hat{c} \Rightarrow x : Q$ channel $\hat{c}$ variable x and continuation Q

Structure of a π-thread



Remark : "Flat" Lex. env. (+ tail calls) $\Longrightarrow$ stackless [VEE06]

Commitments

Emission $\hat{c} \Leftarrow v : Q$ channel $\hat{c}$ value v (or expression $e \approx lazy$) and continuation Q

Reception $\hat{c} \Rightarrow x : Q$ channel $\hat{c}$ variable x and continuation Q

Explicit commitments $\Longrightarrow$ fast scheduling

A glimpse of the PCM

```
record Scheduler {
  run : Queue[PiThread]
  ready : Queue[PiThread]
  wait : Queue[PiThread]
  old : Queue[PiThread]
  date : Int
}
record PiThread {
  env : Array[Value]
  knows : Set[Channel]
  state : { R, Y, W, O }
  commits : Array[Commit]
  clock, date : Int
}
```

```
record Channel {
  taken : Lock
  globalrc, waitrc : Int
  commits : Set[Commit]
}

record Commit {
  kind : { IN, OUT }
  thread : PiThread
  chan : Channel
  val : Value
  clock : Int
}
```

A scheduling example

Skip

Thread	clock
Pool	0
Task_1	0
Task_2	0
Task_3	0
Task_4	0

Channel	commits
$\hat{e}$	[]
$\hat{l}$	[]

$\text{TaskPool}(2,\hat{e},\hat{l}) \parallel \text{Task}(\hat{e}) \parallel \text{Task}(\hat{e}) \parallel \text{Task}(\hat{e}) \parallel \text{Task}(\hat{e})$

A scheduling example

Skip

Thread	clock
Pool	0
Task_1	0
Task_2	0
Task_3	0
Task_4	0

Channel	commits
$\hat{e}$	[]
$\hat{l}$	[]

$\hat{l}\,?,\ \text{TaskPool}(3,\hat{e},\hat{l})$
$+\ [2>0]\ \hat{e}\,?(\text{release}),\ \text{Permit}(\text{release},\hat{l})\ ||\ \text{TaskPool}(1,\hat{e},\hat{l})$
$||$ **new**(rel :chan<>),$\hat{e}\,!(\text{rel}),$ /* *task behavior* */, rel !
$||$ **new**(rel :chan<>),$\hat{e}\,!(\text{rel}),$ /* *task behavior* */, rel !
$||$ **new**(rel :chan<>),$\hat{e}\,!(\text{rel}),$ /* *task behavior* */, rel !
$||$ **new**(rel :chan<>),$\hat{e}\,!(\text{rel}),$ /* *task behavior* */, rel !

A scheduling example

Skip

Thread	clock
Pool	0
$Task_1$	0
$Task_2$	0
$Task_3$	0
$Task_4$	0

Channel	commits
$\hat{e}$	[?Pool@0]
$\hat{l}$	[?Pool@0]

$\hat{l}$?, TaskPool(3,$\hat{e}$,$\hat{l}$)
+ [2>0] $\hat{e}$?(release), Permit(release,$\hat{l}$) || TaskPool(1,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* *task behavior* */, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* *task behavior* */, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* *task behavior* */, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* *task behavior* */, rel !

A scheduling example

Skip

Thread	clock
Pool	0
$Task_1$	0
$Task_2$	0
$Task_3$	0
$Task_4$	0

Channel	commits
$\hat{e}$	[?Pool@0]
$\hat{l}$	[?Pool@0]

$\hat{l}$?, TaskPool(3,$\hat{e}$,$\hat{l}$)
+ [2>0] $\hat{e}$?(release), Permit(release,$\hat{l}$) || TaskPool(1,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* *task behavior* */, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* *task behavior* */, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* *task behavior* */, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* *task behavior* */, rel !

A scheduling example

Skip

Thread	clock
Pool	0
$Task_1$	0
$Task_2$	0
$Task_3$	0
$Task_4$	0

Channel	commits
$\hat{e}$	[?Pool@0]
$\hat{l}$	[?Pool@0]

$\hat{l}$?, TaskPool(3,$\hat{e}$,$\hat{l}$)
+ [2>0] $\hat{e}$?(release), Permit(release,$\hat{l}$) || TaskPool(1,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* *task behavior* */, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* *task behavior* */, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* *task behavior* */, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* *task behavior* */, rel !

A scheduling example

Skip

Thread	clock
Pool	0
$Task_1$	0
$Task_2$	0
$Task_3$	0
$Task_4$	0

Channel	commits
$\hat{e}$	[?Pool@0]
$\hat{l}$	[?Pool@0]
$\hat{r1}$	[]

$\hat{l}$?, TaskPool(3,$\hat{e}$,$\hat{l}$)
+ [2>0] $\hat{e}$?(release), Permit(release,$\hat{l}$) || TaskPool(1,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* *task behavior* */, rel !
|| $\hat{e}$!($\hat{r1}$), /* *task behavior* */, $\hat{r1}$!
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* *task behavior* */, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* *task behavior* */, rel !

A scheduling example

Skip

Thread	clock
Pool	0
$Task_1$	0
$Task_2$	0
$Task_3$	0
$Task_4$	0

Channel	commits
$\hat{e}$	[?Pool@0]
$\hat{l}$	[?Pool@0]
$\hat{r1}$	[]

$\hat{l}$?, TaskPool(3,$\hat{e}$,$\hat{l}$)
+ [2>0] $\hat{e}$?(release), Permit(release,$\hat{l}$) || TaskPool(1,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel!
|| $\hat{e}$!($\hat{r1}$), /* task behavior */, $\hat{r1}$!
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel!
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel!

A scheduling example

Skip

Thread	clock
Pool	1
Permit_1	0
Task_1	0
Task_2	0
Task_3	0
Task_4	0

Channel	commits
$\hat{e}$	[]
$\hat{l}$	[?Pool@0]
$\hat{r1}$	[]

Permit($\hat{r1}$,$\hat{l}$) || TaskPool(1,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !
|| /* task behavior */, $\hat{r1}$!
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !

A scheduling example

Skip

Thread	clock
Pool	1
Permit_1	0
Task_1	0
Task_2	0
Task_3	0
Task_4	0

Channel	commits
$\hat{e}$	[]
$\hat{l}$	[?Pool@0]
$\hat{r1}$	[?Permit_1@0]

$\hat{r1}$?,$\hat{l}$! || TaskPool(1,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* *task behavior* */, rel !
|| /* *task behavior* */, $\hat{r1}$!
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* *task behavior* */, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* *task behavior* */, rel !

A scheduling example

Skip

Thread	clock
Pool	2
Permit_1	0
Permit_2	0
Task_1	0
Task_2	0
Task_3	0
Task_4	0

Channel	commits
$\hat{e}$	[]
$\hat{l}$	[?Pool@0, ?Pool@1]
$\hat{r1}$	[?Permit_1@0]
$\hat{r2}$	[?Permit_2@0]

$\hat{r1}$?,$\hat{l}$! || $\hat{r2}$?,$\hat{l}$! || TaskPool(0,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !
|| /* task behavior */, $\hat{r1}$!
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !
|| /* task behavior */, $\hat{r2}$!

A scheduling example

Skip

Thread	clock
Pool	2
Permit_1	0
Permit_2	0
Task_1	0
Task_2	0
Task_3	0
Task_4	0

Channel	commits
$\hat{e}$	[]
$\hat{l}$	[?Pool@2]
$\hat{r1}$	[?Permit_1@0]
$\hat{r2}$	[?Permit_2@0]

$\hat{r1}$?,$\hat{l}$! || $\hat{r2}$?,$\hat{l}$! || $\hat{l}$?, TaskPool(1,$\hat{e}$,$\hat{l}$) + [0>0] ...
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !
|| /* task behavior */, $\hat{r1}$!
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !
|| /* task behavior */, $\hat{r2}$!

A scheduling example

Skip

Thread	clock
Pool	2
Permit_1	0
Permit_2	0
Task_1	0
Task_2	0
Task_3	0
Task_4	0

Channel	commits
$\hat{e}$	[$!\text{Task}_1@0$]
$\hat{l}$	[?Pool@2]
$\hat{r1}$	[$?\text{Permit}_1@0$]
$\hat{r2}$	[$?\text{Permit}_2@0$]
$\hat{r3}$	[]

$\hat{r1}\,?,\hat{l}\,!\;||\;\hat{r2}\,?,\hat{l}\,!\;||\;\hat{l}\,?,\;\text{TaskPool}(1,\hat{e},\hat{l})\;+\;[0{>}0]\;\ldots$
$||\;\hat{e}\,!(\hat{r3}),\;/*\;\textit{task behavior}\;*/,\;\hat{r3}\,!$
$||\;\boxed{/*\;\textit{task behavior}\;*/},\;\hat{r1}\,!$
$||\;\textbf{new}(\text{rel}:\text{chan}{<>}),\hat{e}\,!(\text{rel}),\;/*\;\textit{task behavior}\;*/,\;\text{rel}\,!$
$||\;\boxed{/*\;\textit{task behavior}\;*/},\;\hat{r2}\,!$

A scheduling example

Skip

Thread	clock
Pool	2
Permit_1	0
Permit_2	0
Task_1	0
Task_2	0
Task_3	0
Task_4	0

Channel	commits
$\hat{e}$	$[!\text{Task}_1@0, !\text{Task}_3@0]$
$\hat{l}$	$[?\text{Pool}@2]$
$\hat{r1}$	$[?\text{Permit}_1@0]$
$\hat{r2}$	$[?\text{Permit}_2@0]$
$\hat{r3}$	[]
$\hat{r4}$	[]

$\hat{r1}?,\hat{l}! \parallel \hat{r2}?,\hat{l}! \parallel \hat{l}?, \text{TaskPool}(1,\hat{e},\hat{l}) + [0>0] \ldots$
$\parallel \hat{e}!(\hat{r3}), /* \text{ task behavior } */, \hat{r3}!$
$\parallel \boxed{/* \text{ task behavior } */}, \hat{r1}!$
$\parallel \hat{e}!(\hat{r4}), /* \text{ task behavior } */, \hat{r4}!$
$\parallel \boxed{/* \text{ task behavior } */}, \hat{r2}!$

A scheduling example

Skip

Thread	clock
Pool	2
Permit_1	0
Permit_2	0
Task_1	0
Task_2	0
Task_3	0
Task_4	0

Channel	commits
$\hat{e}$	$[!\text{Task}_1@0, !\text{Task}_3@0]$
$\hat{l}$	$[?\text{Pool}@2]$
$\hat{r1}$	$[?\text{Permit}_1@0]$
$\hat{r2}$	$[?\text{Permit}_2@0]$
$\hat{r3}$	[]
$\hat{r4}$	[]

$\hat{r1}\,?,\hat{l}\,! \parallel \hat{r2}\,?,\hat{l}\,! \parallel \hat{l}\,?,\ \text{TaskPool}(1,\hat{e},\hat{l}) + [0>0]$ ...
$\parallel \hat{e}\,!(\hat{r3}),$ /* task behavior */, $\hat{r3}\,!$
$\parallel$ /* task behavior */, $\hat{r1}\,!$
$\parallel \hat{e}\,!(\hat{r4}),$ /* task behavior */, $\hat{r4}\,!$
$\parallel \hat{r2}\,!$

A scheduling example

Skip

Thread	clock
Pool	2
Permit_1	0
Permit_2	1
Task_1	0
Task_2	0
Task_3	0

Channel	commits
$\hat{e}$	$[!\text{Task}_1@0, !\text{Task}_3@0]$
$\hat{l}$	$[?\text{Pool}@2]$
$\hat{r1}$	$[?\text{Permit}_1@0]$
$\hat{r2}$	[]
$\hat{r3}$	[]
$\hat{r4}$	[]

$\hat{r1}\,?,\hat{l}\,! \;||\; \hat{l}\,! \;||\; \hat{l}\,?,\ \text{TaskPool}(1,\hat{e},\hat{l}) + [0>0]$ ...
$||\ \hat{e}\,!(\hat{r3}),$ /* task behavior */, $\hat{r3}\,!$
$||\ \boxed{\text{/* task behavior */}},\ \hat{r1}\,!$
$||\ \hat{e}\,!(\hat{r4}),$ /* task behavior */, $\hat{r4}\,!$

A scheduling example

Skip

Thread	clock
Pool	3
Permit_1	0
Task_1	0
Task_2	0
Task_3	0

Channel	commits
$\hat{e}$	[$!\text{Task}_1@0$, $!\text{Task}_3@0$]
$\hat{l}$	[]
$\hat{r1}$	[$?\text{Permit}_1@0$]
$\hat{r2}$	[]
$\hat{r3}$	[]
$\hat{r4}$	[]

$\hat{r1}\,?,\hat{l}\,!$ || $\text{TaskPool}(1,\hat{e},\hat{l})$
|| $\hat{e}\,!(\hat{r3})$, /* task behavior */, $\hat{r3}\,!$
|| /* task behavior */, $\hat{r1}\,!$
|| $\hat{e}\,!(\hat{r4})$, /* task behavior */, $\hat{r4}\,!$

A scheduling example

Skip

Thread	clock
Pool	3
Permit_1	0
Permit_3	0
Task_1	0
Task_2	0
Task_3	0

Channel	commits
$\hat{e}$	$[!\text{Task}_1@0]$
$\hat{l}$	[]
$\hat{r1}$	$[?\text{Permit}_1@0]$
$\hat{r2}$	[]
$\hat{r3}$	[]
$\hat{r4}$	$[?\text{Permit}_3@0]$

$\hat{r1}?,\hat{l}! \parallel \hat{r4}?,\hat{l}! \parallel \text{TaskPool}(0,\hat{e},\hat{l})$
$\parallel \hat{e}!(\hat{r3}),$ ** task behavior */*, $\hat{r3}!$
$\parallel$ ** task behavior */*, $\hat{r1}!$
$\parallel$ ** task behavior */*, $\hat{r4}!$

A scheduling example

Skip

Thread	clock
Pool	3
Permit_1	0
Permit_3	0
Task_1	0
Task_2	0
Task_3	0

Channel	commits
$\hat{e}$	$[!\text{Task}_1@0]$
$\hat{l}$	[]
$\hat{r1}$	$[?\text{Permit}_1@0]$
$\hat{r2}$	[]
$\hat{r3}$	[]
$\hat{r4}$	$[?\text{Permit}_3@0]$

$\hat{r1}?,\hat{l}! \parallel \hat{r4}?,\hat{l}! \parallel \text{TaskPool}(0,\hat{e},\hat{l})$
$\parallel \hat{e}!(\hat{r3})$, ** task behavior **/, $\hat{r3}!$
$\parallel$ ** task behavior **/, $\hat{r1}!$
$\parallel$ ** task behavior **/, $\hat{r4}!$

etc ...

Outline

1 Motivations

2 The intermediate language

3 Scheduling principles

4 Garbage collection

5 Conclusion

Garbage collection

Parallel Garbage Collectors

Reference counting "easy" parallelization (decentralization)

Tracing (very) complex parallelization

Garbage collection

Parallel Garbage Collectors

Reference counting "easy" parallelization (decentralization)

- ... but (very) complex cycle collection
- and efficiency issues (counting overhead)

Tracing (very) complex parallelization

Garbage collection

Parallel Garbage Collectors

Reference counting "easy" parallelization (decentralization)

- ... but (very) complex cycle collection
- and efficiency issues (counting overhead)

Tracing (very) complex parallelization

- ... but "efficient" in practice

Garbage collection

Parallel Garbage Collectors

Reference counting "easy" parallelization (decentralization)

- ... but (very) complex cycle collection
- and efficiency issues (counting overhead)

Tracing (very) complex parallelization

- ... but "efficient" in practice

Our approach : global reference counting on channels

Garbage collection

Parallel Garbage Collectors

Reference counting "easy" parallelization (decentralization)

- ... but (very) complex cycle collection
- and efficiency issues (counting overhead)

Tracing (very) complex parallelization

- ... but "efficient" in practice

Our approach : global reference counting on channels

- "Easy" parallelization (decentralization)

Garbage collection

Parallel Garbage Collectors

Reference counting "easy" parallelization (decentralization)

- ... but (very) complex cycle collection
- and efficiency issues (counting overhead)

Tracing (very) complex parallelization

- ... but "efficient" in practice

Our approach : global reference counting on channels

- "Easy" parallelization (decentralization)
- efficiency (reference counting up-to aliasing)

Garbage collection

Parallel Garbage Collectors

Reference counting "easy" parallelization (decentralization)

- ... but (very) complex cycle collection
- and efficiency issues (counting overhead)

Tracing (very) complex parallelization

- ... but "efficient" in practice

Our approach : global reference counting on channels

- "Easy" parallelization (decentralization)
- efficiency (reference counting up-to aliasing)
- cycle collection (as a by-product)

Garbage collector principles

Elementary predicate

$$\text{knows}(\hat{c}, [\Gamma; \delta] : P) \stackrel{\text{def}}{=} \exists x \in \text{dom}(\delta),\ \delta(x) = \hat{c}$$

Garbage collector principles

Elementary predicate

$$\text{knows}(\hat{c}, [\Gamma; \delta] : P) \stackrel{\text{def}}{=} \exists x \in \text{dom}(\delta),\ \delta(x) = \hat{c}$$

Global references

$$\text{globalrc}(\hat{c}, \Delta \vdash \prod_i [\Gamma_i; \delta_i] : P_i) \stackrel{\text{def}}{=} \sum_i \begin{cases} 1 \text{ if knows}(\hat{c}, [\Gamma_i; \delta_i] : P_i) \\ 0 \text{ otherwise} \end{cases}$$

Garbage collector principles

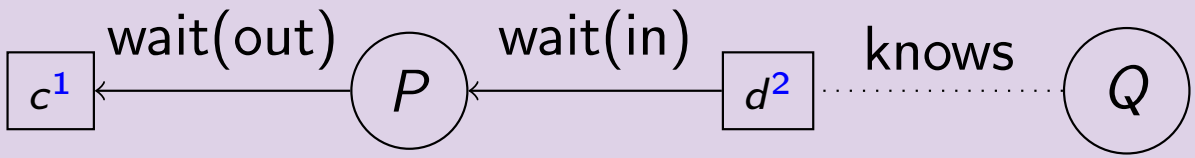
Elementary predicate

$$\text{knows}(\hat{c}, [\Gamma; \delta] : P) \stackrel{\text{def}}{=} \exists x \in \text{dom}(\delta),\ \delta(x) = \hat{c}$$

Global references

$$\text{globalrc}(\hat{c}, \Delta \vdash \prod_i [\Gamma_i; \delta_i] : P_i) \stackrel{\text{def}}{=} \sum_i \begin{cases} 1 \text{ if knows}(\hat{c}, [\Gamma_i; \delta_i] : P_i) \\ 0 \text{ otherwise} \end{cases}$$

Representation



Collecting channels

Règle

$$\frac{\mathrm{globalrc}(\hat{c}, A) = 0}{\Delta, \hat{c} \vdash A \rightarrow \Delta \vdash A} \ (\mathit{reclaim})$$

Collecting channels

Règle

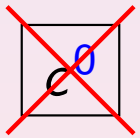
$$\frac{\mathrm{globalrc}(\hat{c}, A) = 0}{\Delta, \hat{c} \vdash A \to \Delta \vdash A} \ (\mathit{reclaim})$$

$\boxed{c^0}$

Collecting channels

Règle

$$\frac{\mathrm{globalrc}(\hat{c}, A) = 0}{\Delta, \hat{c} \vdash A \to \Delta \vdash A} \ (\textit{reclaim})$$



Collecting threads

Règle

$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j} \ (\mathit{stuck})$$

Collecting threads

Règle

$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j} \ (\mathit{stuck})$$

Principles

Collecting threads

Règle

$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j} \ (\mathit{stuck})$$

Principles

- Detection of a *clique* of threads waiting ...

Collecting threads

Règle

$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j} \ (\mathit{stuck})$$

Principles

- Detection of a *clique* of threads waiting ...
- ... on channels only known to the clique itself

Collecting threads

Règle

$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j} \ (\mathit{stuck})$$

Principles

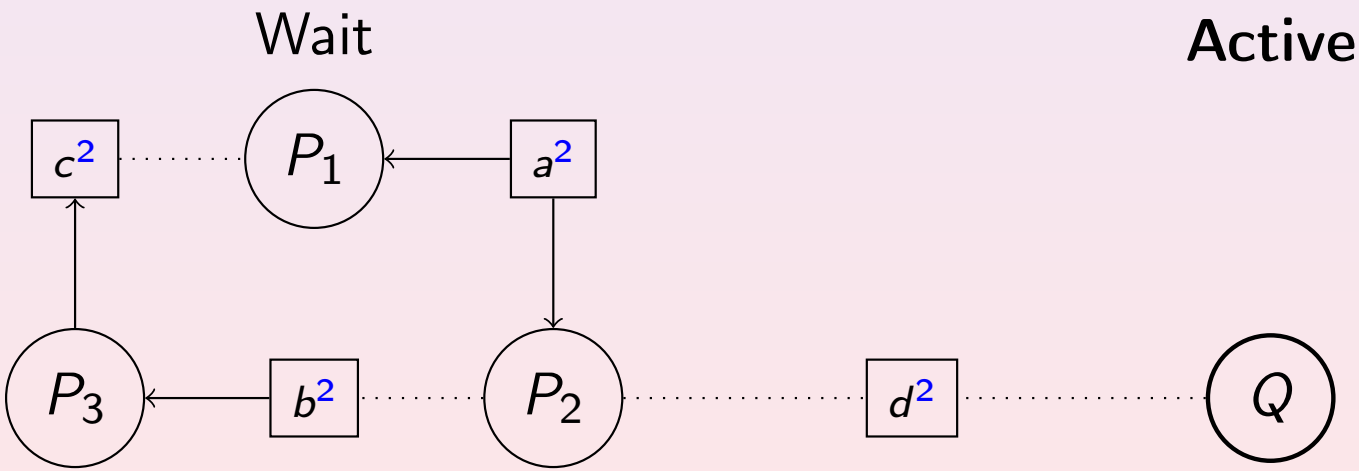
- Detection of a *clique* of threads waiting ...
- ... on channels only known to the clique itself

Remark : no explicit cycle detection

Thread collection : positive case

Rule

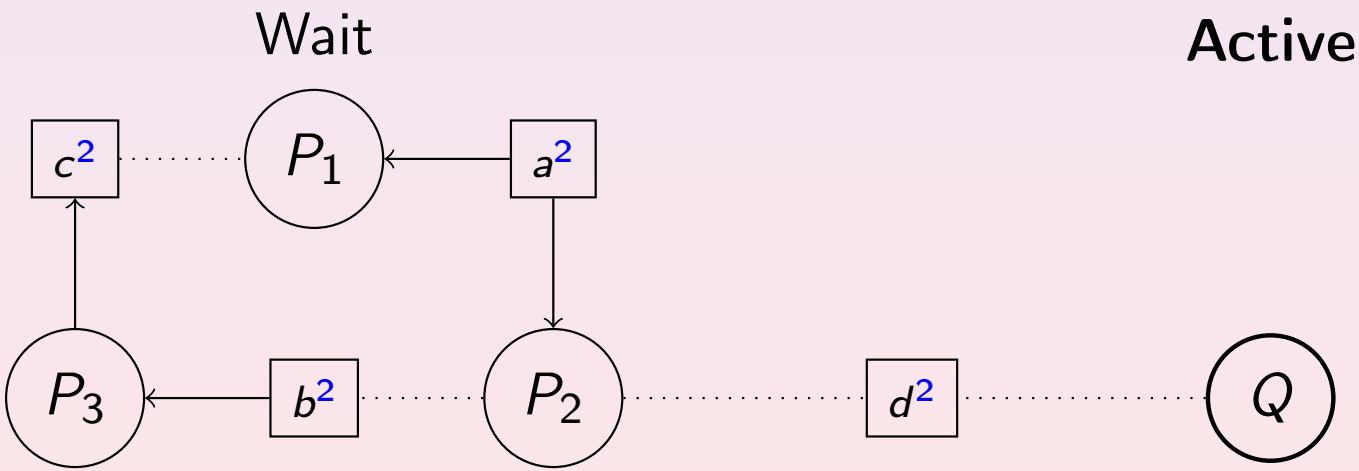
$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j} \ (\mathit{stuck})$$



Thread collection : positive case

Rule

$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j} \ (\mathit{stuck})$$

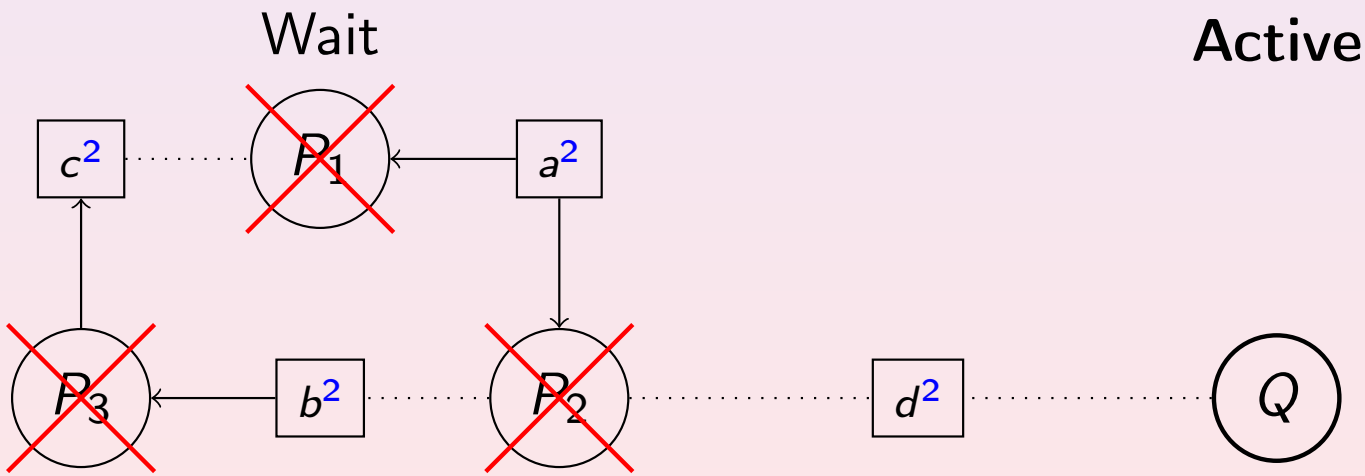


Thread collection : positive case

Rule

$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j} \ (\mathit{stuck})$$

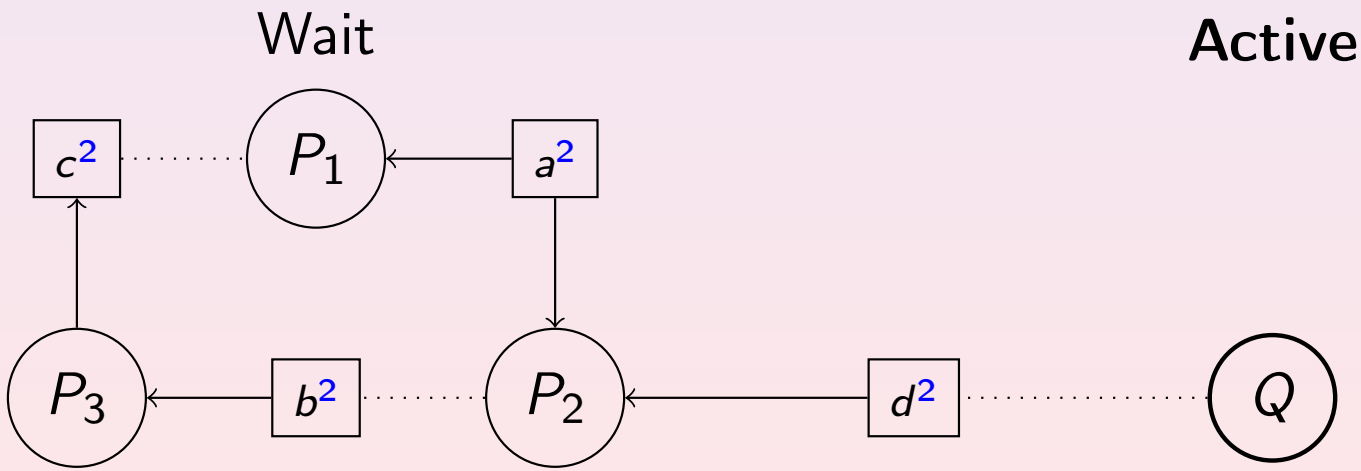
$$\longrightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j$$



Thread collection : negative case

Rule

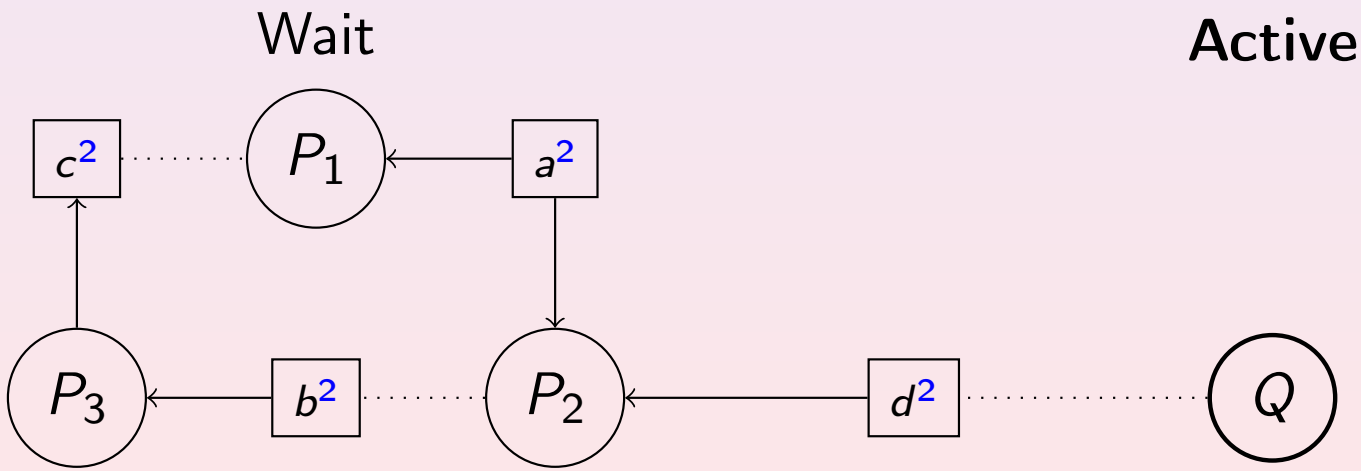
$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j} \ (stuck)$$



Thread collection : negative case

Rule

$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) \neq 0}{\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j} \ (stuck)$$



GC algorithm

```
procedure gc2(th :PiThread) {
  var clique :Set[PiThread] := ∅
  var candidates :Set[PiThread] := { th }
  do {
    var candidate :PiThread := choose(candidates)
    for(ch :Channel ∈ chans(candidate.commits)) {
      if ch.waitrc = ch.globalrc then {
        candidates := candidates
                      ∪ procs(ch.inCommits∪ch.outCommits)
                      \ clique
      } else return
    }
    clique := clique ∪ { candidate }
  } while(candidates≠ ∅)

  for stuck :PiThread ∈ clique { reclaim(stuck) }
}
```

Outline

1 Motivations

2 The intermediate language

3 Scheduling principles

4 Garbage collection

5 Conclusion

Implementations

Available

- CubeVM : *stackless* interpreter from VEE06
- LuaPi : library base on the Lua coroutines
  (invited researcher at Puc/Rio)
  - Explicit commitments, O(1) scheduling
    + extensions : broadcast, join patterns
- JavaPi : library base on the Java Threads
  - Parallel implementation, *lock-free* algorithms, deadlock detection (simplified GC)

Implementations

Available

- CubeVM : *stackless* interpreter from VEE06
- LuaPi : library base on the Lua coroutines
  (invited researcher at Puc/Rio)
  - Explicit commitments, O(1) scheduling
    + extensions : broadcast, join patterns
- JavaPi : library base on the Java Threads
  - Parallel implementation, *lock-free* algorithms, deadlock detection (simplified GC)

Ongoing work

- Native parallel compiler
  - "inlined" PCM
  - "automatic" threading
  - **Availability** : hopefully 2011

Implementations

Available

- CubeVM : *stackless* interpreter from VEE06
- LuaPi : library base on the Lua coroutines
  (invited researcher at Puc/Rio)
  - Explicit commitments, O(1) scheduling
    + extensions : broadcast, join patterns
- JavaPi : library base on the Java Threads
  - Parallel implementation, *lock-free* algorithms, deadlock detection (simplified GC)

Ongoing work

- Native parallel compiler
  - "inlined" PCM
  - "automatic" threading
  - **Availability** : hopefully 2011

Future works

Front end

- Language extensions : broadcast, join patterns, datatypes, etc.
- Experiment : type systems for the π-calculus
- Automatic "stack injection" (inference of "chain reactions")

Future works

Front end

- Language extensions : broadcast, join patterns, datatypes, etc.
- Experiment : type systems for the π-calculus
- Automatic "stack injection" (inference of "chain reactions")

Backend(s)

- GC with I/O channels
- Support for structured datatypes (records etc.)
- Benchmarks, etc.

Publications

VEE'06 a Stackless Runtine Environment for a π-calculus (with Samuel Hym)

JFLA'10 Principes et Pratiques de la Programmation en π-calcul

DAMP'11 Parallel Computing with the π-calculus

Publications

VEE'06 a Stackless Runtine Environment for a π-calculus (with Samuel Hym)

JFLA'10 Principes et Pratiques de la Programmation en π-calcul

DAMP'11 Parallel Computing with the π-calculus

$\Rightarrow$ Questions ?