Compiling the π-calculus into a Multithreaded Typed Assembly Language

Tiago Cogumbreiro
cogumbreiro@di.fc.ul.pt

Joint work with
Francisco Martins Vasco T. Vasconcelos

Universidade de Lisboa

Workshop on Places '08

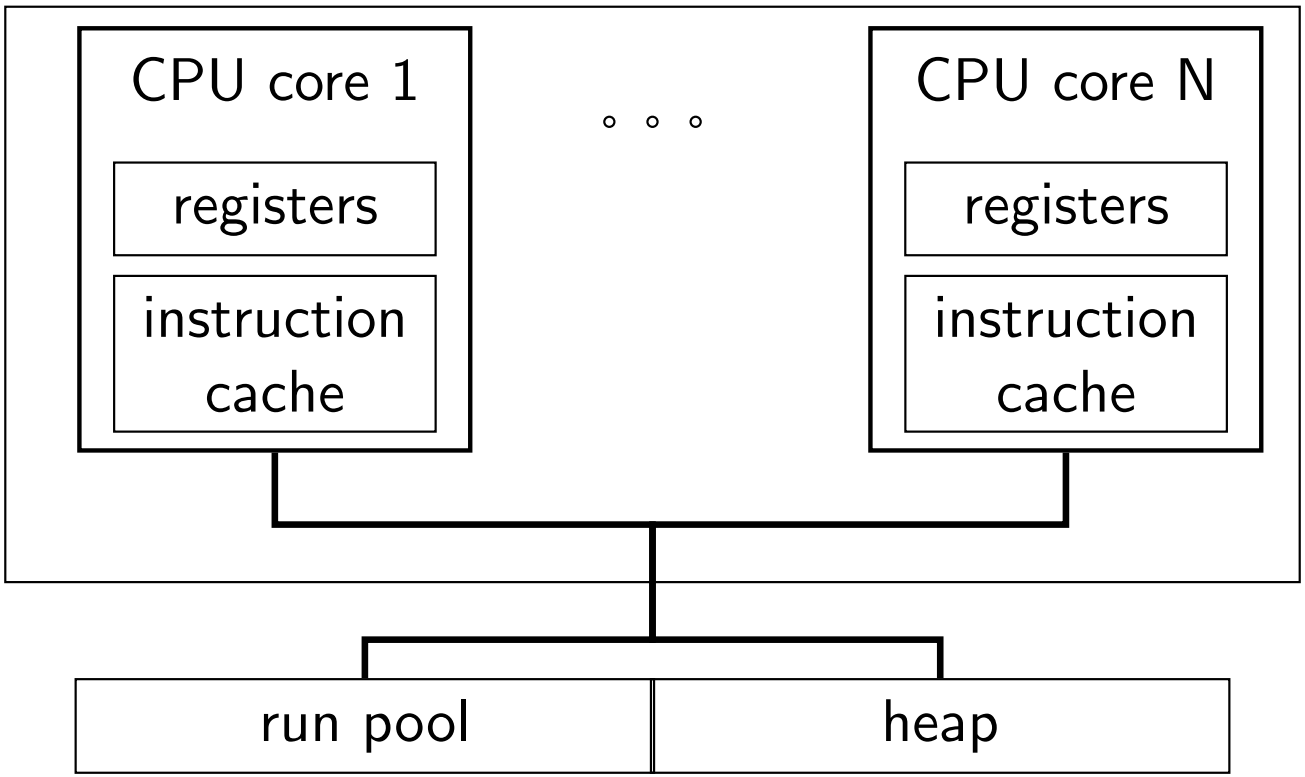
Goal Compilation from the π-calculus into a multithreaded typed assembly language (MIL)

Result Type-preserving translation

Source language: typed π-Calculus

$P ::=$		**Processes**	$v ::=$		**Values**
	$\mathbf{0}$	nil		$\ldots \mid 0 \mid \ldots$	integer
$\mid$	$\overline{x}\langle v \rangle$	output	$\mid$	x	name
$\mid$	$x(y).P$	input			
$\mid$	$P \mid Q$	parallel	$T ::=$		**Types**
$\mid$	$!\, x(y).P$	replicated input		int	integer type
$\mid$	$(\nu\, x : (T))P$	restriction	$\mid$	(T)	channel type

Target language: MIL (architecture)



Target language: MIL (features)

Locks

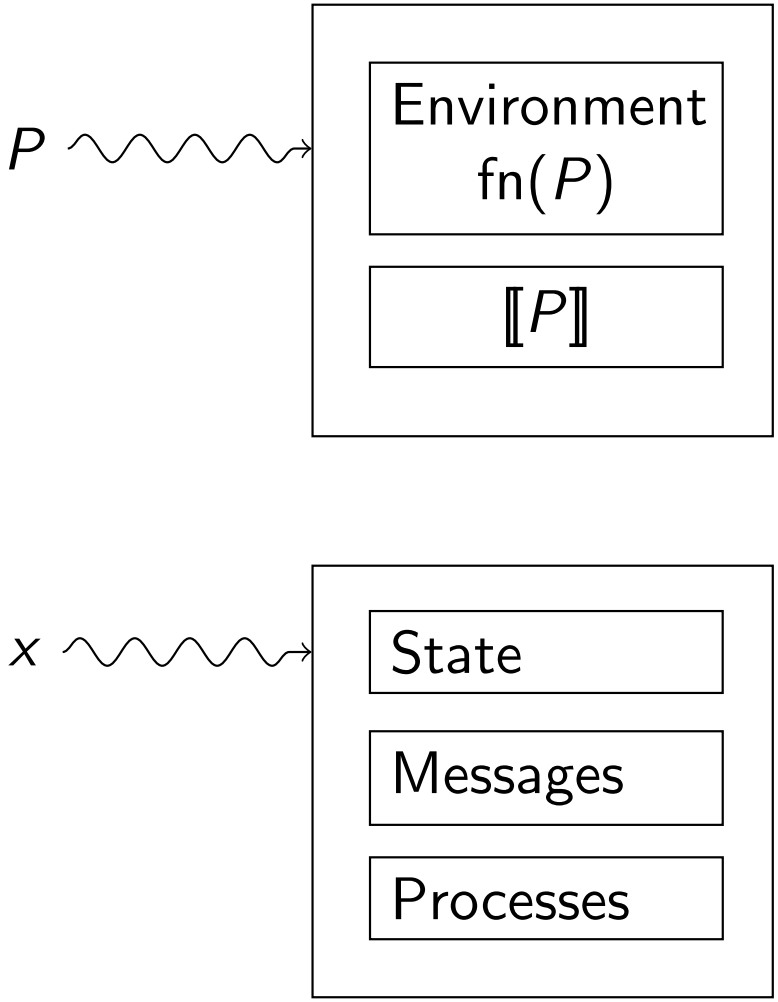
Create	$\alpha, r :=$ newLock b
Acquire	$r :=$ testSetLock v
Release	unlock v

Threads

Create	fork v
Finish	yield

- Type system enforces race-condition freedom

Translating processes and values

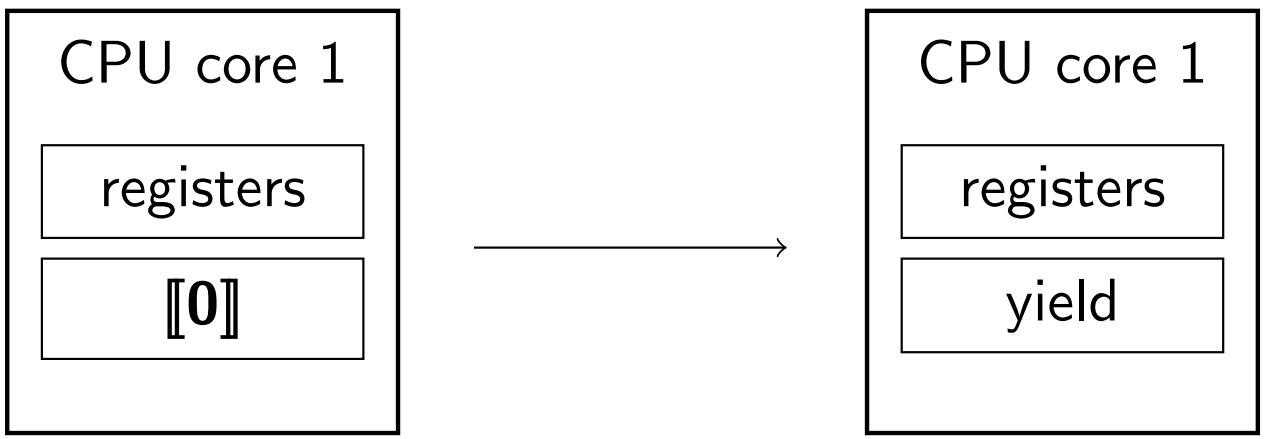


Translation of nil processes

$$[\![\mathbf{0}]\!] \rightsquigarrow \text{yield}$$

Translation of nil processes

$[\![\mathbf{0}]\!] \rightsquigarrow$ yield

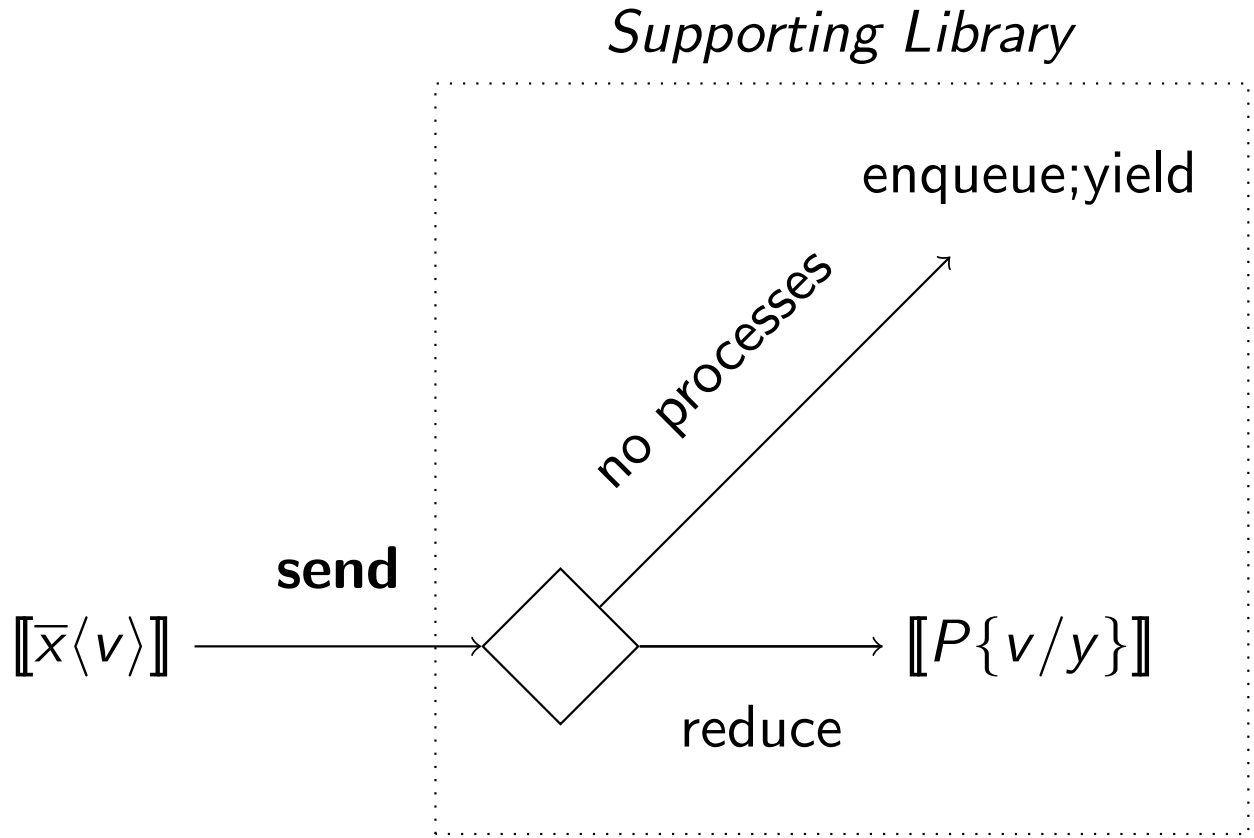


Translation of output processes

$$[\![\overline{x}\langle v\rangle]\!] \rightsquigarrow \text{jump } \mathit{send}(x, v)$$

Translation of output processes

$$[\![\overline{x}\langle v\rangle]\!] \rightsquigarrow \text{jump } \mathit{send}(x, v)$$

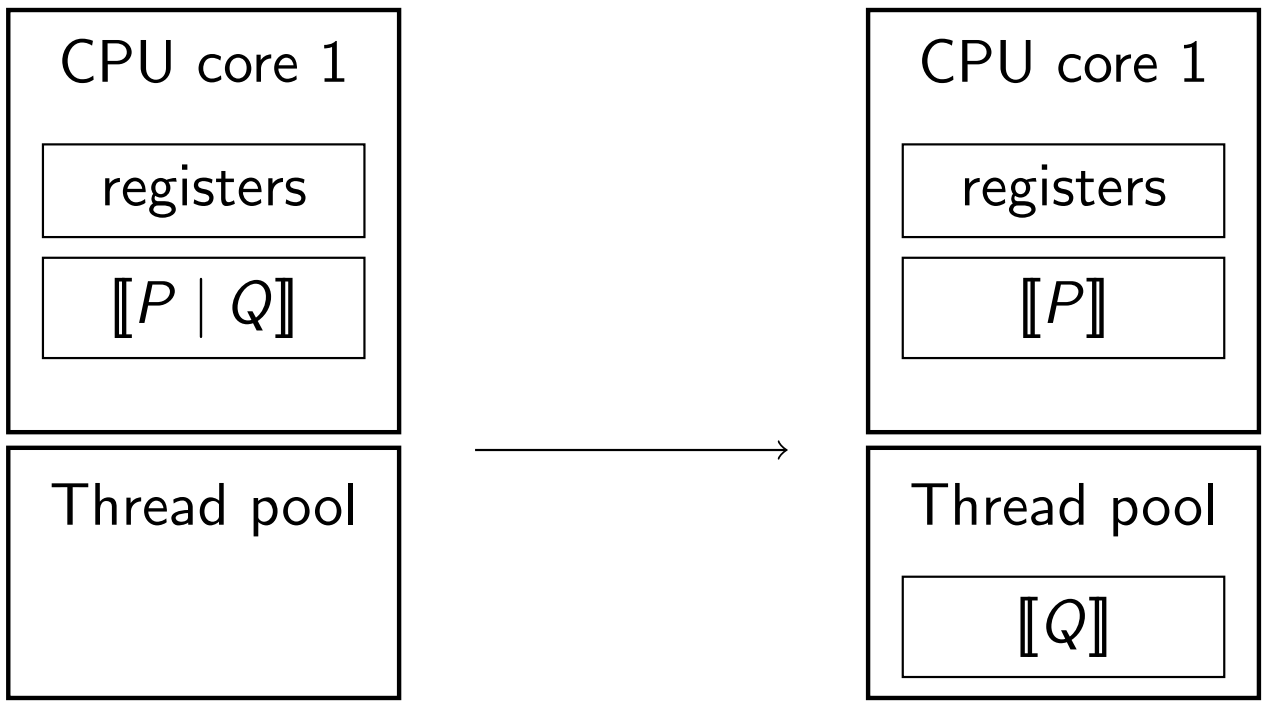


Translation of parallel processes

$$[\![P \mid Q]\!] \rightsquigarrow \text{fork } [\![Q]\!]; \text{jump } [\![P]\!]$$

Translation of parallel processes

$$[\![P \mid Q]\!] \rightsquigarrow \text{fork } [\![Q]\!]; \text{jump } [\![P]\!]$$

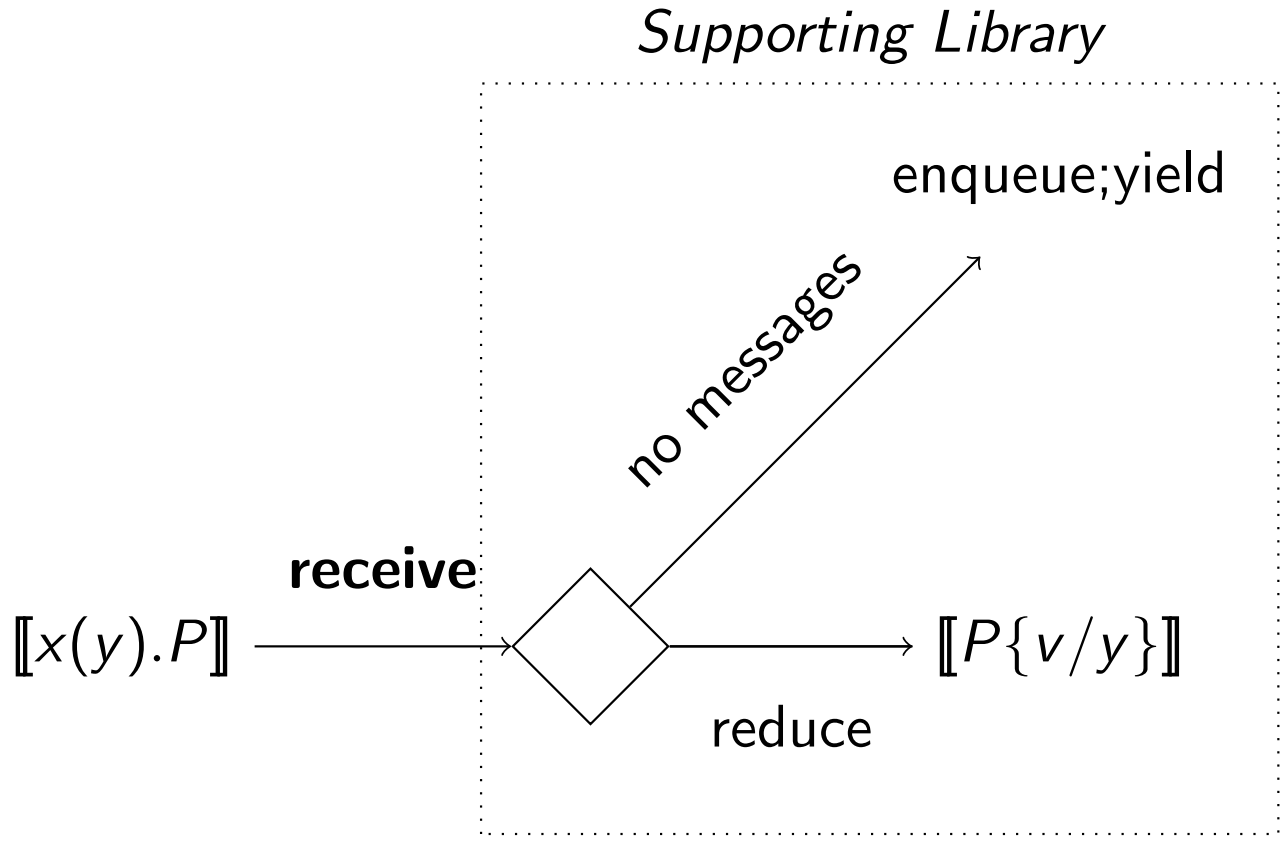


Translation of input processes

$$[\![x(y).P]\!] \rightsquigarrow \text{jump } \mathit{receive}(x, P)$$

Translation of input processes

$$[\![x(y).P]\!] \rightsquigarrow \text{jump } \mathit{receive}(x, P)$$



Supporting library

- Queue operations (creating, enqueueing, dequeueing)
- Three public operations send, receive, and create channel
- 19 code blocks
- 34 type definitions
- 322 lines of MIL code
- 8 registers needed
- Locks are abstracted from the translation function

Lock usage

- Multiple readers on environments (no contention)
- One lock per channel

Spin lock

```
send [alpha,tau] (r1: tau,
                  r4: ChannelType(tau,alpha),
                  r5: <lock(alpha)>^alpha) {
  -- spin lock to acquire the global lock alpha
  r2 := tsIS r5
  if r2 = 0
    -- acquired the lock, unpack the channel
    jump sendUnpack[tau][alpha]
  -- try again
  jump send[tau][alpha]
}
```

Trying to reduce

```
sendMessage [alpha,tau]
                    (r1: tau,
                     r2: ChannelQueueType(tau,alpha),
                     r3: <lock(alpha)>^alpha)
                     requires (alpha;;) {
  -- get the state of the channel
  r4 := ChannelQueueState(r2)
  if r4 = CHANNEL_QUEUE_WITH_PROCS
    -- when there are, deliver the message:
    jump sendMessageReduce[tau][alpha]
```

```
    -- flag the channel as containing messages:
    ChannelQueueState(r2) := CHANNEL_QUEUE_WITH_MSGS
    -- get the queue of messages
    r2 := ChannelQueueMsgs(r2)
    -- put the message (r1) in the queue
    QueueAdd(r2,r1,r4,r1,tau,alpha)
    unlockE r3
    yield
}
```

Conclusions

- Formalized translation in a multithreaded architecture
- Type-preservation: If P is a well typed π-process, then $[\![P]\!]$ is a well typed MIL program

Future work

- Simplifying the supporting library
- Extend MIL (lock-free channels)
- Correctness

- For publications and implementation, please refer to `http://gloss.di.fc.ul.pt/mil`

Thank you.

Any questions?