Compiling the $\pi$-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 $\pi$-calculus into a multithreaded typed assembly language (MIL)

Result  Type-preserving translation
Source language: typed $\pi$-Calculus

\[
P ::= \begin{array}{llll}
0 & \text{nil} & \ldots & 0 \\
\overline{x}(v) & \text{output} & x \\
x(y).P & \text{input} \\
P | Q & \text{parallel} \\
!x(y).P & \text{replicated input} \\
(\nu x : (T))P & \text{restriction} \\
\end{array}
\]

$\pi$-Calculus

Processes

\[
v ::= \begin{array}{ll}
\ldots & 0 \\
x & \text{name} \\
\end{array}
\]

Values

$\pi$-Calculus

Types

\[
T ::= \begin{array}{ll}
\text{int} & \text{channel type} \\
(T) & \text{integer type} \\
\end{array}
\]
Target language: MIL (architecture)
Locks

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

Threads

Create fork $v$
Finish yield

- Type system enforces race-condition freedom
Translating processes and values

\[ P \sim \text{Environment} \text{ fn}(P) \sim [P] \sim \text{State} \sim \text{Messages} \sim \text{Processes} \]
Translation of nil processes

\[ \begin{array}{c}
[0] \quad \Rightarrow \quad \text{yield}
\end{array} \]
Translation of nil processes

\[ [0] \xrightarrow{\text{yield}} \]

CPU core 1

- registers
  - [0]

CPU core 1

- registers
- yield
Translation of output processes

\[
\overline{x}(v) \rightarrow \text{jump send}(x, v)
\]
Translation of output processes

\[
\begin{align*}
\overline{\langle v \rangle} & \xrightarrow{\text{jump}} \text{send}(x, v)
\end{align*}
\]
Translation of parallel processes

\[ [P \mid Q] \sim\sim\rightarrow \text{fork } [Q]; \text{jump } [P] \]
Translation of parallel processes

\[ [P | Q] \leadsto \text{fork} [Q]; \text{jump} [P] \]

Diagram:

- CPU core 1
  - registers
  - \([P | Q]\)
- Thread pool

- CPU core 1
  - registers
  - \([P]\)
- Thread pool
  - \([Q]\)
Translation of input processes

\[ \lbrack x(y).P \rbrack \sim \sim \rightarrow \text{jump } receive(x, P) \]
Translation of input processes

\[ [x(y).P] \sim \sim \text{jump receive}(x, P) \]

Supporting Library

enqueue;yield

no messages

receive: 

\[ [x(y).P] \rightarrow \text{reduce} \rightarrow [P\{v/y\}] \]
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 := tslS 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 $\pi$-process, then $\llbracket P \rrbracket$ 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?