Safe Parallel Programming with Session Java

Nicholas Ng Nobuko Yoshida Olivier Pernet
Raymond Hu Yiannos Kryftis*

Imperial College London *National Technical University of Athens

COORDINATION 2011
June 6, 2011

Motivation

- Parallel programming is non-trivial and error prone (eg. deadlock)
- Session types theory guarantees communication safety between processes
- Parallel programming with a session-based programming language for **safety** (type safety, deadlock freedom) and **performance**

Contributions

1. Extended Session Java (SJ) with *multi-channel primitives* for parallel programming
2. Defined *multi-channel session calculus* with operational semantics and typing system
3. Showed the practical use of *multi-channel primitives* by implementing representative parallel algorithms in SJ
4. Evaluated performance of parallel algorithms implemented in SJ and compared against MPJ Express

Session types

- Typing system for π-calculus [Honda et al., ESOP'98]
- π-calculus models structured interactions between processes
- Communication should have a **dual**

Conventional types/sorts

- `int i = 9`
- `i` and 9 are both `int` datatype

Session types

- Program 1: `send(9)`
- Program 2: `int intValue = receive()`
- *Send int* and *Receive int* are duals

Session programming with SJ

Session Java (SJ) [Hu et al., ECOOP'08]

- An implementation of session types in Java
- Provides a socket programming interface

Session initiation	`accept()`	`request()`
Communication	`send()`	`receive()`
Iteration	`outwhile` statement	`inwhile` statement

- Prelimary work in [Bejleri et al., PLACES'09]
- But lacks efficient mechanism to synchronise multiple sessions

Session programming with SJ: Workflow

1. Declare session type (called `protocol`) in source code
2. Local session type conformance by SJ compiler
   (ie. does program implement session as declared?)
3. Duality check between communicating programs at runtime
   (ie. are `protocol`s compatible?)

```
protocol helloWorldSvr {
 sbegin. // start session
 ![          // Outwhile
   !<String> // Send
 ]*
}
```

```
protocol helloWorldClnt {
 cbegin. // Join session
 ?[          // Inwhile
   ?(String) // Recv
 ]*
}
```

Session programming with SJ: Workflow

1. Declare session type (called `protocol`) in source code
2. Local session type conformance by SJ compiler (ie. does program implement session as declared?)
3. Duality check between communicating programs at runtime (ie. are `protocol`s compatible?)

```
protocol helloWorldSvr
 { sbegin.![!<String>]* }

SJSocket s = ss.accept();
s.outwhile(i++<3) {
 s.send("Hello World");
}
```

```
protocol helloWorldClnt
 { cbegin.?[?(String)]* }

SJSocket c = cs.request();
c.inwhile {
 String str = c.receive();
}
```

inwhile and outwhile

- Powerful construct to connect two sessions
- Allow one process to control iteration of another

$$P_1 \quad \overset{s12}{\rightarrow} \quad P_2$$

s12: session between P1 and P2

P1 `s12.outwhile(true){ /*... */ }`

P2 `s12.inwhile { /*... */}`

Iteration chaining

How to synchronise multiple independent sessions?

$$P_1 \quad \stackrel{s12}{\rightarrow} \quad P_2 \quad \stackrel{s23}{\rightarrow} \quad P_3$$

Iteration chaining

How to synchronise multiple independent sessions?

$$P_1 \quad \overset{s12}{\rightarrow} \quad P_2 \quad \overset{s23}{\rightarrow} \quad P_3$$

Incorrect, non type-safe implementation of P_2:

```
s12.inwhile {
  s23.outwhile(true) {
    // ...
  }
}
```

Iteration chaining

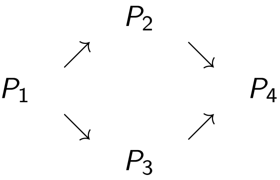
How to synchronise multiple independent sessions?

$$P_1 \overset{s12}{\rightarrow} P_2 \overset{s23}{\rightarrow} P_3$$

P_2 with *iteration chaining* syntax:

```
s23.outwhile(s12.inwhile) {
  // ...
  s12.send();
  s23.send();
}
```

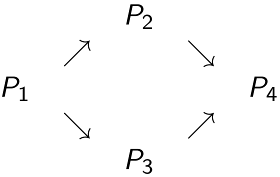
Multi-channel primitives



How to write P_1 (again, incorrect and non type-safe):

```
e = true;
s12.outwhile( e ) {
  s13.outwhile( e ) {
    // ...
  }
}
```

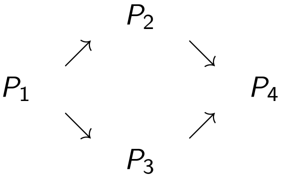
Multi-channel primitives



Multi-channel `outwhile`:

```
<s12, s13>.outwhile(true) {
  // ...
}
```

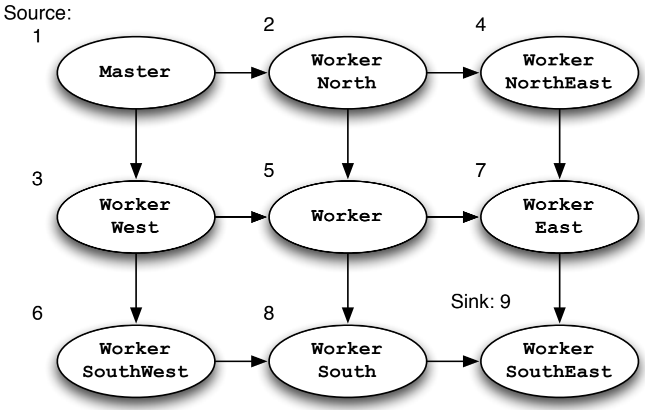
Multi-channel primitives



Similarly for P_4, multi-channel `inwhile`:

```
<s24, s34>.inwhile {
  // ...
}
```

Multi-channel primitives example: Jacobi solution



Multi-channel primitives example: Jacobi solution

```
/** Master */
<right, down>.outwhile(e)
      {
  // ...
}
```

```
/** North */
<right, down>.outwhile(
  left.inwhile) {
  // ...
}
```

```
/** NorthEast */
down.outwhile(
  left.inwhile) {
  // ...
}
```

```
/** West */
<down, right>.outwhile(
  up.inwhile) {
  // ...
}
```

```
/** Worker */
<right, down>.outwhile(
  <left, up>.inwhile) {
  // ...
}
```

```
/** East */
down.outwhile(
  <left, up>.inwhile) {
  // ...
}
```

```
/** SouthWest */
right.outwhile(
  up.inwhile) {
  // ...
}
```

```
/** South */
right.outwhile(
  <left, up>.inwhile) {
  // ...
}
```

```
/** SouthEast */
<left, up>.inwhile {
  // ...
}
```

Multi-channel primitives example: Jacobi solution

Worker process, chained multi-channel inwhile and outwhile

```
<right, down>.outwhile(<left, up>.inwhile) {
  // ... calculation ...

  up.send(topRow);
  topRow = up.receive();
  right.send(rightCol);
  rightCol = right.receive();

  bottomRow_rcvd = down.receive();
  down.send(bottomRow);
  leftCol_rcvd = left.receive();
  left.send(leftCol);
}
```

Multi-channel primitives in SJ: summary

- More topologies can be expressed
- More intuitive to program and reason about
- Synchronises multiple sessions

Multi-channel session types: intuition

- Formalisation of multi-channel primitives
  - Correctness
  - Deadlock freedom
- `outwhile` multicasts loop condition to all channels
- `inwhile` collects loop conditions from all channels

Multi-channel session types: reduction rules (1)

Outwhile (true)

$$E[\langle k_1 \dots\ k_n \rangle.\texttt{outwhile}(e)\{\ P\ \}] \qquad (E[e] \rightarrow^{*} E'[\texttt{true}])$$
$$\rightarrow\ E[P; \langle k_1 \dots\ k_n \rangle.\texttt{outwhile}(e')\{\ P\ \}] \mid k_1 \dagger [\texttt{true}] \mid \dots \mid k_n \dagger [\texttt{true}]$$

Outwhile (false)

$$E[\langle k_1 \dots\ k_n \rangle.\texttt{outwhile}(e)\{\ P\ \}] \qquad (E[e] \rightarrow^{*} E'[\texttt{false}])$$
$$\rightarrow\ E[\mathbf{0}] \mid k_1 \dagger [\texttt{false}] \mid \dots \mid k_n \dagger [\texttt{false}]$$

- Multichannel `outwhile` forwards loop condition to all session channels

Multi-channel session types: reduction rules (2)

Inwhile (true)

$$E[\langle k_1 \ldots\ k_n\rangle.\texttt{inwhile}\{\ \ P\ \ \}] \mid k_1 \dagger [\texttt{true}] \mid \ldots \mid k_n \dagger [\texttt{true}]$$
$$\rightarrow\ E[P; \langle k_1 \ldots\ k_n\rangle.\texttt{inwhile}\{\ \ P\ \ \}]$$

Inwhile (false)

$$E[\langle k_1 \ldots\ k_n\rangle.\texttt{inwhile}\{\ \ P\ \ \}] \mid k_1 \dagger [\texttt{false}] \mid \ldots \mid k_n \dagger [\texttt{false}]$$
$$\rightarrow\ E[\mathbf{0}]$$

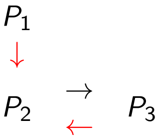
- Multichannel `inwhile` collects loop conditions from all session channels
- Proceeds if conditions match
- Mismatch of conditions: runtime error

Well-formed topology: Example

$$
\begin{array}{ccc}
P_1 & & \\
\downarrow & & \\
P_2 & \begin{array}{c}\rightarrow \\ \leftarrow\end{array} & P_3
\end{array}
$$

- `s23.outwhile(<s12, s23>.inwhile)`
- Valid `inwhile` `outwhile` topology construction

Well-formed topology: Example



- `s23.outwhile(<s12, s23>.inwhile)`
- Valid `inwhile` `outwhile` topology construction
- Cycle in the flow of control messages: **deadlock**

Well-formed topology

- Governs how multi-channel `outwhile` and `inwhile` are connected
- Well-formed iff topology constructed as **uni-directed acyclic graph**
- All examples in paper conforms to well-formed topology:
  - *n*-Body simulation: ring topology
  - Jacobi solution of the discrete Poission equation: mesh topology
  - Linear equation solver: wraparound mesh topology

Well-formed topology

Theorem (Subject reduction)

Multi-channel `outwhile` and `inwhile` will not reduce to error

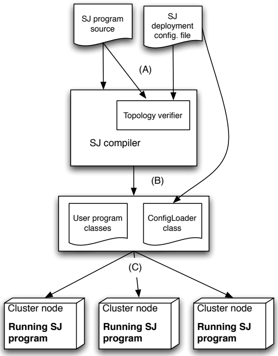
Theorem (Type and communication safety)

A typable process which forms a well-formed topology is type and communication safe.

Theorem (Deadlock freedom)

If P forms a well-formed topology and P is well-typed, then P is deadlock free.

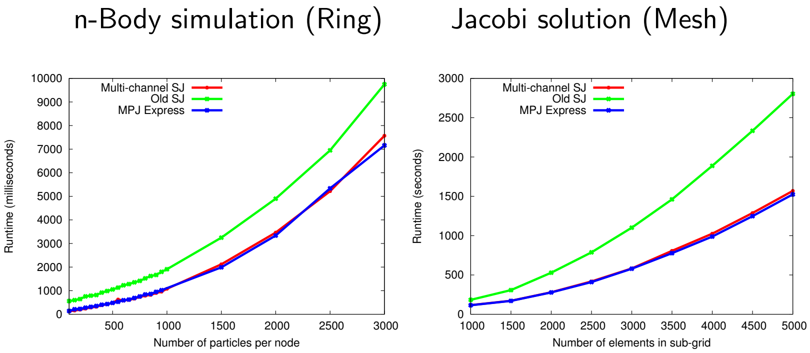
Multi-channel session types and SJ programming



Workflow of a SJ program:

1. Declare session type (called `protocol`) in source code
2. Local session type conformance by SJ compiler
3. Well-formed topology verification on deployment config file
4. Program instantiated with verified config file
5. Duality check between communicating programs at runtime

Benchmark results



- Significant improvement over non multi-channel version
- Performs competitively against MPJ Express (MPI in Java)

Conclusions

- Multi-channel primitives increased the expressiveness of Session Java
- *multi-channel session type theory* and *well-formed topology* guarantees *communication safety* and *deadlock freedom*
- Benchmark result shows competitive performance against industry standard
- Parallel programming in multi-channel SJ is both **safe** and **efficient**

Future work

- Session-based low level, natively compiled language (eg. C) for low overhead HPC and systems programming
- Incorporate `outwhile` and `inwhile` primitives into *multiparty session types*

Full version:

`http://www.doc.ic.ac.uk/~cn06/pub/2011/sj_parallel/`

Syntax

(Values)			
v	$::=$	a, b, x, y	shared names
	$\mid$	true, false	boolean
	$\mid$	n	integer
(Expressions)			
e	$::=$	$v \mid e + e \mid \texttt{not}(e) \ldots$	value, sum, not
	$\mid$	$\langle k_1 \ldots k_n \rangle.\texttt{inwhile}$	inwhile
(Processes)			
P	$::=$	0	inaction
	$\mid$	T	prefixed
	$\mid$	$P\ ;\ Q$	sequence
	$\mid$	$P \mid Q$	parallel
	$\mid$	$(\nu u)\ P$	hiding
(Declaration)			
D	$::=$	$X(xk) = P$	

(Prefixed processes)			
T	$::=$	$\texttt{request}\ a(k)\ \texttt{in}\ P$	request
	$\mid$	$\texttt{accept}\ a(k)\ \texttt{in}\ P$	accept
	$\mid$	$k![\tilde{e}]$	sending
	$\mid$	$k?(\tilde{x})\ \texttt{in}\ P$	reception
	$\mid$	$\texttt{throw}\ k[k']$	sending
	$\mid$	$\texttt{catch}\ k(k')\ \texttt{in}\ P$	reception
	$\mid$	$X[\tilde{e}\tilde{k}]$	variables
	$\mid$	$\texttt{def}\ D\ \texttt{in}\ P$	recursion
	$\mid$	$k \lhd l$	selection
	$\mid$	$k \rhd \{l_1 : P_1 \| \cdots \| l_n : P_n\}$	branch
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$	conditional
	$\mid$	$\langle k_1 \ldots\ k_n \rangle.\texttt{inwhile}\{\ Q\ \}$	inwhile
	$\mid$	$\langle k_1 \ldots\ k_n \rangle.\texttt{outwhile}(e)\{\ P\ \}$	outwhile
	$\mid$	$k \dagger [b]$	message

Operational Semantics

$$\texttt{accept}\ a(k)\ \texttt{in}\ P_1 \mid \texttt{request}\ a(k)\ \texttt{in}\ P_2 \rightarrow (\nu k)(P_1 \mid P_2) \qquad k![c] \mid k?(x)\ \texttt{in}\ P_2 \rightarrow P_2[c/x] \qquad \text{[LINK] [COM]}$$

$$k \triangleright \{l_1 : P_1 \| \cdots \| l_n : P_n\} \mid k \triangleleft l_i; \rightarrow P_i \quad (1 \le i \le n) \qquad \texttt{throw}\ k[k'] \mid \texttt{catch}\ k(k')\ \texttt{in}\ P_2 \rightarrow P_2 \qquad \text{[LBL] [PASS]}$$

$$\texttt{if}\ \texttt{true}\ \texttt{then}\ P\ \texttt{else}\ Q \rightarrow P \qquad \texttt{if}\ \texttt{false}\ \texttt{then}\ P\ \texttt{else}\ Q \rightarrow Q \qquad \text{[IF]}$$

$$\texttt{def}\ X(xk) = P\ \texttt{in}\ X[ck] \rightarrow \texttt{def}\ X(xk) = P\ \texttt{in}\ P\{c/x\} \qquad \text{[DEF]}$$

$$\langle k_1 \ldots k_n \rangle.\texttt{inwhile}\{\ P\ \} \mid \Pi_{i\in\{1..n\}} k_i \dagger [\texttt{true}] \rightarrow P; \langle k_1 \ldots k_n \rangle.\texttt{inwhile}\{\ P\ \} \qquad \text{[IW1]}$$

$$\langle k_1 \ldots k_n \rangle.\texttt{inwhile}\{\ P\ \} \mid \Pi_{i\in\{1..n\}} k_i \dagger [\texttt{false}] \rightarrow \mathbf{0} \qquad \text{[IW2]}$$

$$E[\langle k_1 \ldots k_n \rangle.\texttt{inwhile}] \mid \Pi_{i\in\{1..n\}} k_i \dagger [\texttt{true}] \rightarrow E[\texttt{true}] \qquad \text{[IWE1]}$$

$$E[\langle k_1 \ldots k_n \rangle.\texttt{inwhile}] \mid \Pi_{i\in\{1..n\}} k_i \dagger [\texttt{false}] \rightarrow E[\texttt{false}] \qquad \text{[IWE2]}$$

$$E[e] \rightarrow^* E'[\texttt{true}] \Rightarrow E[\langle k_1 \ldots k_n \rangle.\texttt{outwhile}(e)\{\ P\ \}] \rightarrow E'[P; \langle k_1 \ldots k_n \rangle.\texttt{outwhile}(e)\{\ P\ \}] \mid \Pi_{i\in\{1..n\}} k_i \dagger [\texttt{true}] \qquad \text{[OW1]}$$

$$E[e] \rightarrow^* E'[\texttt{false}] \Rightarrow E[\langle k_1 \ldots k_n \rangle.\texttt{outwhile}(e)\{\ P\ \}] \rightarrow E'[\mathbf{0}] \mid \Pi_{i\in\{1..n\}} k_i \dagger [\texttt{false}] \qquad \text{[OW2]}$$

$$P \equiv P' \text{ and } P' \rightarrow Q' \text{ and } Q' \equiv Q \Rightarrow P \rightarrow Q \qquad \text{[STR]}$$

$$e \rightarrow e' \Rightarrow E[e] \rightarrow E[e'] \qquad P \rightarrow P' \Rightarrow E[P] \rightarrow E[P']$$

$$P \mid Q \rightarrow P' \mid Q' \Rightarrow E[P] \mid Q \rightarrow E[P'] \mid Q' \qquad \text{[EVAL]}$$

In [OW1] and [OW2], we assume $E = E' \mid \Pi_{i\in\{1..n\}} k_i \dagger [b_i]$

Type system

Outwhile

$$\frac{\Gamma;\ \Delta \vdash e \triangleright \mathsf{bool} \qquad \Gamma \vdash P \triangleright \Delta \cdot k_1 : \tau_1.\mathsf{end} \ldots k_n : \tau_n.\mathsf{end}}{\Gamma \vdash \langle k_1 \ldots\ k_n \rangle.\mathtt{outwhile}(e)\{\ P\ \} \triangleright \Delta \cdot k_1 : ![\tau_1]^*.\mathsf{end} \ldots k_n : ![\tau_n]^*.\mathsf{end}}$$

Inwhile

$$\frac{\Gamma;\ \Delta \vdash Q \triangleright \Delta \cdot k_1 : \tau_1.\mathsf{end} \ldots k_n : \tau_n.\mathsf{end}}{\Gamma \vdash \langle k_1 \ldots\ k_n \rangle.\mathtt{inwhile}\{\ \ Q\ \ \} \triangleright \Delta \cdot k_1 : ?[\tau_1]^*.\mathsf{end} \ldots k_n : ?[\tau_n]^*.\mathsf{end}}$$