# A concurrent programming language with refined session types

Juliana Franco and Vasco T. Vasconcelos

LaSIGE, University of Lisbon, Portugal

September 23, 2013

# Motivation

- Session types are by now a well-established methodology for typed, message-passing concurrent computations
- Session types were originally proposed for the pi-calculus
- There is no pi-based implementation on which one may
    - exercise examples
    - test program idioms
    - experiment with type systems

# SePi _ SEssions on PI

- An exercise in the design and implementation of a concurrent programming language based on the pi calculus, where process interaction is governed by linearly refined session types
- Allows to explore the practical applicability of new (and old) works on session-based type systems
- Provides a tool where new program idioms and type developments may be tested and eventually incorporated

# Running example _ An online donation service

- Four sorts of participants: bank, server, clients and benefactors
- **Clients** create donation campaigns and send the campaign link to benefactors
- **Benefactors** donate by providing a credit card number and an amount to be charged
- The **server** provides for the creation of campaigns and forwards the donations to the bank
- The **bank** charges the donations on credit cards

# SePi _ communication channels

- Bi-directional synchronous channels
- Each channel is defined by two end-points: one to write, the other to read
- Each end-point is governed by a session type

# Types _ input/output and termination

?**integer**.T

represents a channel end ready to receive an integer; continues as prescribed by T.

!**integer**.T

sends an integer and continues as T.

**end**

a channel where no further interaction is possible.

# SePi _ channel creation

```
new r w: ?integer.end
```

- r has type ?integer.end
- w has type !integer.end
- dualof ?integer.end is !integer.end
- Equivalent: new w r: !integer.end

# SePi _ channel read/write

```
new w r: !integer.end
w!2013 |
r?x.printInteger!x
```

- The *output* process, !, writes the value 2013 on the newly created channel
- The *input* process, ?, reads from the channel and stores the value on x
- printInteger is a builtin channel end
- The vertical bar, |, denotes parallel composition

## Reduction

- The process

```
new w r: !integer.end
w!2013 |
r?x.printInteger!x
```

- reduces in one step to

```
new w r: end
printInteger!2013
```

- which (prints 2013 on the console and) reduces in one step to

```
new w r: end
{}
```

- The terminated process is denoted by {}, the parallel composition of zero processes

# Types _ choice

- Type

  $$\&\{\text{setDate:T1, commit:T2}\}$$

  represents a channel end offering two choices: setDate and commit. If setDate is chosen then behaves as T1; if commit is chosen then behaves as T2.
- Type

  $$+\{\text{setDate:T3, commit:T4}\}$$

  selects one of the choices.
- **dualof** &{setDate: **end**, commit: **end**} is +{setDate: **end**, commit: **end**}

# SePi _ select and case processes

```
new w r: +{setDate: end, commit: end}
w select setDate |
case r of setDate → printString!”Got setDate”
          commit  → printString!”Got commit”
```

- **select** chooses an option on a menu
- **case** offers a menu of options

# Exchanging an unbounded number of messages

- Clients want to upload the campaign information (setDate) until satisfied and then press the commit button.
- We would like to write:

  $+\{\text{setDate: } !\textbf{integer}.\text{go}-\text{back}-\text{to}-\text{the}-\text{begin, commit: } \textbf{end}\}$

- After the setDate choice is taken the whole menu is again available. Use a recursive type:

  $\textbf{rec}\ \texttt{a}.\ +\{\text{setDate: } !\textbf{integer}.\texttt{a},\ \text{commit: } \textbf{end}\}$

# SePi _ Type abbreviations

- Declare

```
type Donation = {setDate: !integer.Donation, commit: end}
```

- and use the type name Donation in place of

```
rec a. +{setDate: !integer.a, commit: end}
```

# SePi _ unbounded behaviour

w **select** setDate. w!2012. w **select** setDate. w!2013. w **select** commit

- The client may now upload the date two times before committing.

```
def setup r: Donation =
    case r of setDate → r?x. setup!r
              commit → ...
```

- The server recurs after serving the setDate option

# SePi _ process definitions

```
def setup r: Donation = P
RestOfTheProgram
```

- is short for

```
new setup _setupReader: *!Donation
_setupReader*?r.P |
RestOfTheProgram
```

- where _setupReader*?r.P is a replicated input: reduces against zero or more output processes

# Types _ linear and unrestricted

- Donation is a linear type: during the setup phase only one client may share the communication channel. Donation in its full glory:

  ```
  rec a. lin+{setDate: lin!integer.a, commit: end}
  ```

- But channel setup may be shared by multiple processes in parallel. Type

  ```
  rec b. un!Donation. b
  ```

  abbreviated to $*$!Donation

- Type abbreviations allow to omit the **lin**/**un** qualifiers in most cases

# Honest servers

- Benefactors donate by providing the server with a credit card number and a donation amount
- The donation server forwards these values to the bank
- A session with bank process has the following type

$$!\mathsf{CreditCard}\,.\,!\,\textbf{integer}\,.\,\textbf{end}$$

- What guarantees that
  1. the server forwards the correct amount?
  2. the server charges the right amount only once?

# Types _ refinements

- The idea is that the bank is not interested in arbitrary (ccard, amount) pairs but else on pairs for which a charge(ccard, amount) capability has been granted
- We may refine type

  !CreditCard. !**integer**.**end**

  into

  !ccard:CreditCard. !amount:{x: **integer**|charge(ccard, x)}. **end**

# SePi _ assuming and asserting capabilities

- The capability of charging a given amount on a specific credit card is usually granted by the benefactor, by *assuming* an instance of the charge predicate:

```
assume charge("2345", 10) | w!"2345". w!10
```

- In turn, the bank makes sure the capability to charge the card was granted by the client, by *asserting* the same predicate:

```
r?ccard. r?amount. assert charge(ccard, amount)
```

- The server must forward the values received, exactly once

# SePI _ Formulae are treated linearly

- Formulae:
  - Uninterpreted predicates: charge(ccard, amount)
  - Joining: charge(ccard, amount)$*$charge(ccard, amount)
  - Unit: **unit**
- In a valid program
  - each assumption is asserted exactly once and
  - each assert is assumed exactly once

# Demo _ Eclipse plugin

- Syntax highlight
- Validation (type checking)
- Run, interpreter based on Turner's abstract machine
- Code completion, refactoring, . . .

## Summing up

- SePi is a new concurrent programming language based on the monadic pi-calculus where
  - communication between processes is governed by session types
  - refinement types allow the specification of properties about the values exchanged.
- SePi includes a few abbreviations and derived constructs, such as
  - the **dualof** operator
  - input/output of multiples values
  - mutually recursive process definitions and type declarations.
- An Eclipse plugin for SePi facilitates code development. Try it at `http://gloss.di.fc.ul.pt/sepi`

# Future work

- New constructs:
  - an **import** clause
  - an abbreviation for session initiation
- Predicates over expressions, using a SMT solver
- What about your future work on top of SePi?
  - Type systems for progress
  - Polymorphism
  - Subtyping
  - ...