

Implementation of a Linear Session Type System

Second assignment of the Languages for
Concurrency and Distribution, A.Y. 2023/2024

Christian Micheletti
July 1, 2024



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

1 Structure

- Session Types
- Software features

2 Language

- Syntax
- Runtime

The paper *Fundamentals of Session Types* presents a π -calculus equipped with a (linear) type system to check against errors that the system can incur into.

Such language features linear types, which allow checking whether a given channel is used by exactly one process.

This π -calculus' features are incrementally presented through five sections:

- **basics and fundamentals** features, as scope restriction, in sections 2 and 3;
- **recursive types**, needed to write useful unrestricted types, in section 4;
- **unbounded computations**, in section 5, that enable infinite behaviours;
- **branching and selection**, in section 6 (not implemented).

The implementation process follows this very same organization, and will be presented accordingly.

The software is written in Haskell and it features:

- a **code interpreter**, that executes the code in Concurrent Haskell;
- a **type checker**, parallelized using the Eval monad;
- a type inferrer.

The paper has several examples of the expected behaviour of the type checker, that are implemented here as unit tests to avoid regressions.

Chapter 2's original syntax

Implementation

```
P ::=  $\bar{x} v . P$   
       $x(x) . P$   
       $P \mid P$   
      if  $v$  then  $P$  else  $P$   
      0  
       $(\nu x x) P$ 
```

```
P ::= x << v . P  
      | x >> x . P  
      | P | P  
      | if v then P else P  
      | 0  
      | x << x . P  
      | { P }
```

```
v ::= x  
      true  
      false
```

```
v ::= x  
      | true  
      | false
```

After parsing, the program is precomputed lifting all bindings over all parallel compositions, as expressed in the following **structural congruence**:

$$(\nu xy)P|Q \equiv (\nu xy)(P|Q)$$

The code interpreter prints debug information about the program executed, along with a timestamp and a description of what the process did at that time.

Output format

```
[TIMESTAMP | ThreadId THREAD_ID]: MESSAGE
```

Program behaviour is defined according to the **operational semantics** presented in chapter 2.

The language runtime is implemented in concurrent Haskell, using:

- the IO monad, along with a local state that maps variables to channels and literals, to model threads;
- and the MVar type to model channels.

Each thread is created with the

```
forkIO :: IO () -> IO ThreadId
```

function, that simply creates a new thread, returning its id.

Plain MVars can await for value insertion with the function

```
takeMVar :: MVar a -> IO a
```

but the dual operation

```
putMVar :: MVar a -> a -> IO ()
```

does not await the variable to be ready to accept a new value.

Channels are represented as tuples of `MVar`, meaning `(value, idle)`:

- `value :: MVar v`, containing the passed value, which needs to be evaluated;
- `idle :: MVar ()`, which has a `()` value in it when the channel is ready to receive.

Then, the `Channel` datatype further distinguishes whether it is a read end or a write end. Read ends and write ends share the same `MVars`.

Inaction

The process 0 just prints STOP and ends the thread.

Branching

The process if v then P_1 else P_2 prints two debug messages, both starting with BRANCHING:

- the guard before evaluation;
- and then, after evaluation in the local state.

After that, the run continues as the appropriate process.

Binding

The process $(\nu xy).P$ just prints BINDING followed by the two bounded variables. Then, it creates two MVars, one for the value and one for the lock and associates in the local state both variables with the respective ends.

The run proceeds in the same thread as prescribed by P .

Fork

The process $P_1|P_2$ prints FORK followed by the two new processes ids.

To prevent a concurrent program to end before all forked threads terminate, this code constructs two MVars that are notified when the two new threads finish. This process doesn't do anything more than awaiting for both threads to finish.

Sending

The process $\bar{x}v.P$ prints SENDING followed by the value:

- before evaluation;
- and after evaluation.

Then the process sends the value over the channel x and proceeds as prescribed by P .

Receiving

The process $x(v).P$ prints RECEIVING followed by:

- the newly bound variable name;
- and the value received.

Then the process proceeds as prescribed by P .