# Implementation of a Linear Session Type System

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

Christian Micheletti

July 8, 2024


Università degli Studi di Padova

# Outline

# Session Types



The paper *Fundamentals of Session Types* presents a $\pi$-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 $\pi$-calculus' features are incrementtally 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.

# Software features



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.

# Language - Process Syntax



Section 2's original syntax

$$
\begin{aligned}
P ::= & \ \overline{x}\, v.P \\
& \ x(x).P \\
& \ P \mid P \\
& \ \texttt{if}\ v\ \texttt{then}\ P\ \texttt{else}\ P \\
& \ 0 \\
& \ (\nu\, x\, x)P
\end{aligned}
$$

$$
\begin{aligned}
v ::= & \ x \\
& \ \texttt{true} \\
& \ \texttt{false}
\end{aligned}
$$

Implementation

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

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

# Language - Precomputation



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

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

# Language - Operational Semantics



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 section 2.

# Runtime - Concurrent Haskell



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.

# Runtime - Channels (1/2)



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.

# Runtime - Channels (2/2)



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`.

# Runtime - Program Behaviour (1/3)



## 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.

# Runtime - Program Behaviour (2/3)



## 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.

# Runtime - Program Behaviour (3/3)



## Sending

The process $\overline{x}v.P$ 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.

# Basic Types - Syntax



Implementation of qualifier, pretypes and types are fully compliant to section 3's original syntax:

$$
\begin{aligned}
q &::= lin \\
&\quad\ un \\
p &::= ?T.T \\
&\quad\ !T.T
\end{aligned}
\qquad\qquad
\begin{aligned}
T &::= bool \\
&\quad\ end \\
&\quad\ qp \\
\Gamma &::= \emptyset \\
&\quad\ \Gamma, x : T
\end{aligned}
$$

Contexts are implemented as hash maps of types

# Basic Types - Duality



Duality is partially defined as follows:

```
dualType :: SpiType -> SpiType
dualType End = End
dualType Boolean = error "..."
dualType (Qualified q (Receiving t1 t2)) =
    Qualified q (Sending t1 (dualType t2))
dualType (Qualified q (Sending t1 t2)) =
    Qualified q (Receiving t1 (dualType t2))
dualType (Recursive a p) = Recursive a (dualType p)
dualType (TypeVar x) = TypeVar
```

# Basic Types - Context



Contexts support the following operations:

- (nondeterministic) **context split** $\Gamma = \Gamma_1 \circ \Gamma_2$: the function `ndsplit :: Context -> [(Context, Context)]` creates all possible combinations of dividing `linear` variables, mantaining `unrestricted` variables;
- **update** $\Gamma + (x : T)$: the function `update k t` inserts $k : t$ in the context only if the variable $k$ was not present, or it was yet defined `unrestricted` with type $t$;
- **override** $\Gamma, (x : T)$: this represents newly bounded variables, possibly shadowing preexisting definitions.

# Basic Types - Typing Rules



Sequent calculus rules and operations over contexts are modeled as instances of the context transition **monad**:

```
newtype CT a = CT
    (Context -> TypeErrorBundle TypeError (a, Context))

instance Monad CT where
    return :: a -> CT a
    (>>=) :: CT a -> (a -> CT b) -> CT b
```

Rules can be composed and propagate context side effects, and can yield an output

```
class TypeCheck a where
    type Output a
    check :: a -> CT (Output a)
```

# Basic Types - Modelling Rules



## Unrestricted requirement

un(Γ) holds when all entries in the context are unrestricted:

```
unGamma :: CT ()
unGamma = CT (\context -> if all unrestricted context
    then Right ((), context)
    else Left "Error message...")
```

# Basic Types - Modelling Operations



## Context update

$\Gamma + (x : T)$ throws an error if the conditions arent met:

```
update :: String -> SpiType -> CT ()
update k t = do
    may <- liftPure (M.lookup k)
    case may of
        Just found -> unless (predicate Un t && found ≈ t)
            (throwError "Error message..")
        Nothing    -> sideEffect (M.insert k t)
```



## Context update

Context transitions can even return useful values:

```
extract :: String -> CT SpiType
extract k = do
    t <- get k
    unless (predicate Un t) (delete k)
    return t
```

This is used to optimize some context splits in the [T-Rec] and [T-Send] rules

# Basic Types - Type Checking



The paper shows two sets of rules to type check a process:

- the rules based on **context split**;
  - naive approach, requires to check for all possible context splits for parallel composition (other rules are optimizable);
  - can be done without using the `Output` of the algorithm;
- the rules presented in section 8 (Algorithmic Type Checking):
  - more efficient, doesn't rely on context split;
  - requires to track all used variables along the program and return them as output.

# Basic Types - Context Split (1/7)



Context split rules are modelled as `CT ()`, i.e. transitions that yield no output.

One could think of an intuitive way to parallelize the [T-Par]:

**[T-Par]**

$$\frac{\Gamma_1 \vdash P_1 \qquad \Gamma_2 \vdash P_2}{\Gamma_1 \circ \Gamma_2 \vdash P_1 | P_2} \text{ [T-Par]}$$

# Basic Types - Context Split (2/7)



The algorithm checks all possible splits of $\Gamma_1 \circ \Gamma_2$

```
check (Par p1 p2) = do
    splits <- liftPure ndsplit
    -- Compute all possible splits
    runs <- return () -< (candidate <$> splits)
    liftEither $ foldChoice runs
        {- `using` parList rdeepseq -}
    where
        candidate (c1, c2) = (return () -<
            [ c1 |> check p1
            , c2 |> check p2
            ] {- <&> (`using` parList rdeepseq) -}
            ) >- return ()
```

It seems natural to desire to parallelize this code in the points with comments...

# Basic Types - Context Split (3/7)

... but this would imply loss in performance. Consider the following program:

## Example: assets/well-formed-ill-typed/multiple.spi

```
a1 >< a2: lin?bool.end .
b1 >< b2: lin?bool.end .
c1 >< c2: lin?bool.end .
d1 >< d2: lin?bool.end .
e1 >< e2: lin?bool.end .
f1 >< f2: lin?bool.end .
x >< y: rec x. ?bool.x .
    x1 >< y1: lin?bool.lin!bool.end .
        x2 >< y2: lin?bool.lin!bool.end .
            { x << true . y >> z . if z then 0 else 0
            | y >> z . if z
                then x << false . 0
                else 0
            | x1 << true . x1 >> n . y2 >> n . y2 << n .0
            | y1 >> n . y1 << false . x2 << false . x2 >> n .0
            | a1 << true . b1 << true . c1 << true
                . d1 << true . e1 << true . f1 << true . 0
            | a2 >> e . b2 >> e . c2 >> e . d2 >> e . e2 >> e .
                0
            }
```

# Basic Types - Context Split (4/7)



Running this program with one CPU, it takes roughly 7 seconds to terminate.

Running this program with more CPUs makes it even worse!

Other rules are optimizied to not resort to context split.

## Example: T-In

$$\frac{\Gamma_1 \vdash x : q?T.U \qquad (\Gamma_2 + x : U), y : T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x(y).P}\ [\text{T-In}]$$

In this case the rule holds following those observations:

- if $x$ is not present in $\Gamma_1 \circ \Gamma_2$ the left assumption can never be verified;
- if $\Gamma_1 \circ \Gamma_2$ contains the claim $x : \mathtt{un}\, T.U$, then both $\Gamma_1, \Gamma_2$ contain such claim;
- if $\Gamma_1 \circ \Gamma_2$ contains the claim $x : \mathtt{lin}\, T.U$, then $\Gamma_1$ must contain that claim and $\Gamma_2$ must not.

# Basic Types - Context Split (6/7)



Notice that in order to type $\Gamma_1 \vdash x : q?T.U$, as for the rule [T-Var], $\Gamma_1$ must not contain any linear claim.

Hence, $\Gamma_2$ must contain all linear claims in $\Gamma_1 \circ \Gamma_2$, of course except for $x$ if it was linear.

# Basic Types - Context Split (7/7)



Hence the algorithm for this rule is as follows:

## Example: T-In

```
check (Rec x y p) = do
    xType <- extract x -- If not present, the monadic
        -- bind will make the whole rule fail.
        -- The function extract will preserve
        --
    (t, u) = case xType of
            Qualified _ (Receiving t u) ->
                return (t, u)
            _ -> throwError "..."
    update x u
    replace y t
    check p
```

# Algorithmic Type Checking (1/3)



The problem with the parallel composition rule is that a linear channel cannot be in both contexts

However, this problem can be easily overcame by keeping track of **variables** used in **subject position**, such as channel send or receive.

For processes, output context $\Gamma_2$ contains all free variables "consumed" by process $P$, $L$ contains all linear free variable used in process $P$

$$\Gamma_1 \vdash P : \Gamma_2; L \qquad\qquad \Gamma_1 \vdash x : T; \Gamma_2$$

For variables, the output is a context without the variable, if it was linear

# Algorithmic Type Checking (2/3)



Then **context difference** $\Gamma \div L$ removes all variables in $L$ from $\Gamma$ and yields an error if there was a linear variable in $L$.

Then, parallel composition rule is easily defined:

$$\frac{\Gamma_1 \vdash P : \Gamma_2; L_1 \qquad \Gamma_2 \div L_1 \vdash Q : \Gamma_3; L_2}{\Gamma_1 \vdash P|Q : \Gamma_3; L_2}\ [\text{A-Par}]$$

The implementation is straight forward:

```
check (Par p1 p2) = do
    l1 <- check p1
    contextDiff l1
    check p2
```

# Algorithmic Type Checking (3/3)



Other rules have similar implementation. The optimizations for direct context split check of [T-In] and [T-Out] are related.

Rule [A-If] requires that the outputs for both branches are the same:

$$\frac{\Gamma_1 \vdash v : q\ \mathsf{bool}; \Gamma_2 \qquad \Gamma_2 \vdash P : \Gamma_3; L \qquad \Gamma_2 \vdash Q : \Gamma_3; L}{\Gamma_1 \vdash \mathsf{if}\ v\ \mathsf{then}\ P\ \mathsf{else}\ Q : \Gamma_3; L}\ [\text{A-If}]$$

# Other features



Other features include:

- **recursive types** (described in section 4);
- **tuple syntax** (described in section 4);
- **replicated behaviours** (described in section 5);
- **branching and selection** (described in sectin 6 but not implemented).

# Recursive Types (1/4)



Context update doesn't allow [T-In] and [T-Out] rules (and algorithmic respectives) to type any unbounded type:

**Rule T-In**

$$\frac{\Gamma_1 \vdash x : q?T.U \qquad (\Gamma_2 + x : U), y : T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x(y).P}\ [\text{T-In}]$$

Tt requires the $U$ type to be the same as un$?T.U$. Same holds for [T-Out]

# Recursive Types (2/4)



To overcome this, we would like to consider the solution of an equation like

$$x = \mathsf{un?}\, T.x$$

In order to do this, the paper introduces two new syntax constructs:

- **type variables**;
- a type-level binder $\mu$ that define types as **infinite regular trees**.

This is due to the fact that we need a way to determine type equality which not syntactic congruence

# Recursive Types (3/4)



The paper omits the co-inductive defintion, the code implements this equivalence as a **bisimulation**

```
class (Eq a, Ord a) => Bisimulation a where
    behave :: a -> (a, a)

bisim :: (Bisimulation a) => S.Set (a, a) -> a -> a -> Bool
bisim _   a b | a == b = True
bisim rel a b | S.member (a, b) rel = True
bisim rel a b | S.member (b, a) rel = True
bisim rel a b =
    let (a', oa) = behave a
        (b', ob) = behave b
    in (oa == ob && bisim (S.insert (a, b) rel) a' b')

(≈) :: (Bisimulation a) => a -> a -> Bool
(≈) = bisim S.empty
```

# Recursive Types (4/4)



Recursive types $\mu a.T$ implement the behave function required, as specified in the paper, to never consider the type as it is but another type in the same equivalence class, namely $T[\mu a.T/a]$

```
instance Bisimulation SpiType where
    behave :: SpiType -> (SpiType, SpiType)
    behave (Qualified q (Sending v t)) =
        (t, Qualified q (Sending v End))
    behave (Qualified q (Receiving v t)) =
        (t, Qualified q (Receiving v End))
    behave t@(Recursive x t') = behave (subsType x t t')
    behave t = (t, t)
```

Other types resort to syntactic equivalence

# Tuple Syntax



The paper presents a syntactic sugar for tuple passing:

$\overline{x_1}\langle u, v\rangle.P$ abbreviates $(\nu y_1 y_2)\overline{x_1} y_2.\overline{y_1} u.\overline{y_1} v.P$

$x_2(w, t).P$ abbreviates $x_2(z).z(w).z(t).P$

Hidden channels $y_1, y_2$ need to be typed. There are two ways to achieve this:

- type annotation on the binding $(\nu x_1 x_2)$ (long and tedious);
- type inference (not described in the paper, hence details omitted).

# Replicated Behaviour (1)



Section 5 modifies process syntax annotating receiving channels with `lin` and `un` modifiers:

- if a channel is annotated with `lin` then it is consumed;
- if a channel is annotated with `un` then it is replicated.

Reduction rules (and runtime implementation) are updated accordingly:

$$(\nu xy)\overline{x}v.P|\mathsf{lin}y(z).Q \rightarrow (\nu xy)P|Q[v/z] \quad [\text{R-LinCom}]$$

$$(\nu xy)\overline{x}v.P|\mathsf{un}y(z).Q \rightarrow (\nu xy)(P|Q[v/z]|\mathsf{un}y(z).Q) \quad [\text{R-UnCom}]$$

(In practice, only the `un` modifier is introduced)

# Replicated Behaviour (2)



Types are updated following a third invariant:

inv. (iii) Unrestricted input processes may not contain free linear variables.

That implies:

- there is no replication of linear variables, mantaining their transient behaviour;
- there could be *bounded* linear variables.

# Replicated Behaviour (3)



Hence the new rules for receiving channels are:

## Rule T-In

$$\frac{\Gamma_1 \vdash x : q?T.U \qquad (\Gamma_2 + x : U), y : T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x(y).P} \text{ [T-In]}$$

## Rule A-In

$$\frac{\begin{array}{l}\Gamma_1 \vdash x : q_2?T.U; \Gamma_2 \\ (\Gamma_2, y : T) + x : U \vdash P : \Gamma_3; L\end{array} \qquad q_1 = \text{un} \Longrightarrow L \setminus \{y\} = \emptyset}{\Gamma_1 \vdash q_1 x(y).P : \Gamma_3 \div \{y\}; L \setminus \{y\} \cup (\text{if } q_2 = \text{lin then } \{x\} \text{ else } \emptyset)} \text{ [A-In]}$$