Implementation of a Linear Session Type System

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

Christian Micheletti

July 2, 2024

UNIVERSITÀ DEGLI STUDI DI PADOVA

Outline

Session Types

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.

Session Types - The π-calculus

This π-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

Chapter 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 chapter 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 `MVar`s.

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 Chapter 3's original syntax:

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

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 (1)

Sequent calculus rules are modeled as instnces 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

A check rule can either hold and return () or fail and return an error:

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

Basic Types - Typing Rules (2)

Unrestricted requirement

$\mathrm{un}(\Gamma)$ holds when all entries in the context are unrestricted:

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

Basic Types - Typing Rules (3)

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)
```

Basic Types - Typing Rules (4)

Context update

Context transitions can even return useful values:

```haskell
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