Carnegie Mellon University

Refinements for Session-typed Concurrency

{ By: Josh Acay   Advisor: Frank Pfenning }   Carnegie Mellon University

Introduction

Session types regulate the communication behaviour along channels between concurrent processes in a typed setting with message-passing concurrency. Recently, a connection between session-types and linear logic have been presented (through the lens of the Curry-Howard correspondence), which gave rise to languages such as SILL [1]. Languages incorporating linear session-types enjoy many desirable properties such as global progress, absence of deadlock, and race freedom.

However, vanilla session-types are not strong enough to describe more interesting behavioral properties of concurrent processes. Here, we present a refinement type system in the style of [2]. Our system combines intersections and unions with equirecursive types, which is strong enough to specify and automatically verify many properties of concurrent programs. We have implemented a type-checker for our system.

Language

Session types are described by the following grammar:

$$
\begin{array}{rcll}
A, B, C & ::= & \mathbf{1} & \text{send } \texttt{end} \text{ and terminate} \\
& | & A \otimes B & \text{send channel of type } A \text{ and continue as } B \\
& | & \oplus\{lab_k : A_k\}_{k\in I} & \text{send } lab_i \text{ and continue as } A_i \\
& | & A \multimap B & \text{receive channel of type } A \text{ and continue as } B \\
& | & \&\{lab_k : A_k\}_{k\in I} & \text{receive } lab_i \text{ and continue as } A_i \\
& | & A \sqcap B & \text{act as both } A \text{ and } B \\
& | & A \sqcup B & \text{act as either } A \text{ or } B
\end{array}
$$

Below is a summary of the process expressions, with the sending construct followed by the matching receiving construct.

$$
\begin{array}{rcll}
P, Q, R & ::= & x \leftarrow P_x \,;\, Q_x & \text{cut (spawn)} \\
& | & c \leftarrow d & \text{id (forward)} \\
& | & \text{close } c \mid \text{wait } c \,;\, P & \mathbf{1} \\
& | & \text{send } c\ (y \leftarrow P_y) \,;\, Q \mid x \leftarrow \text{recv } c \,;\, R_x & A \otimes B,\ A \multimap B \\
& | & c.lab \,;\, P \mid \text{case } c \text{ of } \{lab_k \to Q_k\}_{k\in i} & \\
& & \&\{lab_k : A_k\}_{k\in I}, \oplus\{lab_k : A_k\}_{k\in I} &
\end{array}
$$

Note that intersections and unions have no matching constructs since they specify behavioral properties and not structure. Any well-typed program can be given an intersection or a union type. We have three main judgements:

$[A \leq B]$ A is a subtype of B

$[\Psi \vdash P :: (c : A)]$ Process P offers along channel c the session A in the context Ψ

$[\models \Omega :: \Psi]$ Process configuration Ω provides all the channels in Ψ

Intersections and Unions

Typing rules for intersections and unions are given below.

Intersections

$$\frac{\Psi \vdash P :: (c : A) \quad \Psi \vdash P :: (c : B)}{\Psi \vdash P :: (c : A \sqcap B)}\ \sqcap\mathrm{R}$$

$$\frac{\Psi, c : A \vdash P :: (d : D)}{\Psi, c : A \sqcap B \vdash P :: (d : D)}\ \sqcap\mathrm{L}_1 \qquad \frac{\Psi, c : B \vdash P :: (d : D)}{\Psi, c : A \sqcap B \vdash P :: (d : D)}\ \sqcap\mathrm{L}_2$$

Unions

$$\frac{\Psi \vdash P :: (c : A)}{\Psi \vdash P :: (c : A \sqcup B)}\ \sqcup\mathrm{R}_1 \qquad \frac{\Psi \vdash P :: (c : B)}{\Psi \vdash P :: (c : A \sqcup B)}\ \sqcup\mathrm{R}_2$$

$$\frac{\Psi, c : A \vdash P :: (d : D) \quad \Psi, c : B \vdash P :: (d : D)}{\Psi, c : A \sqcup B \vdash P :: (d : D)}\ \sqcup\mathrm{L}$$

Example

We can define strings of bits using processes. Below, `Bits` is the type of bit strings. `emp` represents the empty string, and `zero` and `one` append the corresponding bit to the least significant position.

```
module BitString where
  {- eps is the empty string, zero and one append a least
     significant bit -}
  type Bits = +{eps : 1, zero : Bits, one : Bits}

  {- Bit strings in standard form (no leading zeros) -}
  type Std = +{eps : 1} or StdPos
  type StdPos = +{one : Std, zero : StdPos}

  eps : Bits and Std
  `c <- eps = do
    `c.eps
    close `c

  zero : Bits -o Bits
  one : Bits -o Bits and Std -o Std

  {- Increment a bit string by one. Note the hypotheses that
    we added manually -}
  succ : Std -o Std and {- hypotheses -} StdPos -o StdPos and
    +{eps : 1} -o StdPos
  `c <- succ `d =
    case `d of
      eps -> do wait `d; `c.one; `c.eps; close `c
      zero -> do `c.one; `c <- `d
      one -> do
        `c.zero
        `sd <- succ `d
        `c <- `sd
```

Bidirectional Type Checking

Due to the non-structural nature of intersections and unions, there is no obvious way to go from the declarative typing rules to a type-checking algorithm. We therefore designed an algorithmic system, on which the implementation is based.

We write the algorithmic typing judgement as $\Psi \Vdash P :: (c : \alpha)$ where α is a multiset of types, and Ψ maps a channel to a multiset. On the right, a multiset is interpreted as a disjunction, and on the left, it it interpreted as a conjunction. Additionally, subtyping is handled exclusively at a forward (meaning subsumption cannot be applied freely). Whenever we see an intersection on the left, we break it up into two. We do the same for unions on the right.

The two systems are related through the following soundness and completeness results:

Theorem 1 (Soundness). *If $\Psi \Vdash P :: (c : \alpha)$, then $\bigsqcap \Psi \vdash P :: (c : \bigsqcup \alpha)$.*

Theorem 2 (Completeness). *If $\Psi \vdash P :: (c : A)$, then $\Psi \Vdash P :: (c : A)$.*

Metatheory

We give the standard progress and preservation theorems.

Theorem 3 (Progress). *If $\models \Omega :: \Psi$ then either*

1. *$\Omega \longrightarrow \Omega'$ for some Ω', or*
2. *Ω is poised.*

Intuitively, a process configuration Ω is poised if every process in Ω is waiting on its client.

Theorem 4 (Preservation). *If $\models \Omega :: \Psi$ and $\Omega \longrightarrow \Omega'$ then $\models \Omega' :: \Psi$.*

Proof of preservation is currently in progress.

References

[1] Frank Pfenning and Dennis Griffith. Polarized substructural session types. Draft, March 2015.

[2] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*, pages 268–277, Toronto, Ontario, June 1991. ACM Press.

[3] Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In M.Felleisen and P.Gardner, editors, *Proceedings of the European Symposium on Programming (ESOP'13)*, pages 350–369, Rome, Italy, March 2013. Springer LNCS 7792.