Linearity and the Pi Calculus, Revisited

Luca Padovani
with Tzu-Chun Chen and Andrea Tosatto

Dipartimento di Informatica – Università di Torino – Italy

CoCo:PoPS – 8 June 2015

Type reconstruction for the linear π-calculus

Uses

- infer automatically properties of communicating processes
- explore some (possibly unconventional) meanings for "linear"

How it operates

- `INPUT` an untyped process P (π-calculus with data types)
- `OUTPUT` either fail or a type environment Γ such that $\Gamma \vdash P$

References for the type systems

- Padovani, **Type reconstruction for the linear π-calculus with composite and equi-recursive types**, FoSSaCS 2014
- Padovani, **Deadlock and lock freedom in the linear π-calculus**, CSL-LICS 2014

Programming languages and linearity

Dardha, Giachino, Sangiorgi, **Session types revisited**, PPDP 2012

Consequences

- binary sessions encodable in the (linear) π-calculus with CPS

Programming languages and linearity

Dardha, Giachino, Sangiorgi, **Session types revisited**, PPDP 2012

Consequences

- binary sessions encodable in the (linear) π-calculus with CPS
- any reasonable type system can express protocols
- the key missing ingredient is linearity

Reasonably typed PL	**Linear π-calculus**
channel types	**linear** channel types
product types	–
sum types	–
(equi-)recursive types	–

Tracking the use of channels

Channel types

$$^{\kappa_1,\kappa_2}[t]$$

Uses

$$\kappa \quad ::= \quad 0 \;\mid\; 1 \;\mid\; \omega$$

- 0: not used
- 1: used once
- ω: possibly used

Example

succ1.hy

```
*succ?(x,r).r!(x + 1)
```

Combining type environments

$$\frac{\Gamma; \Delta_1 \vdash P \qquad \Gamma; \Delta_2 \vdash Q}{\Gamma; \Delta_1, \Delta_2 \vdash P \mid Q}$$

Combining type environments

$$\frac{\Gamma; \Delta_1 \vdash P \qquad \Gamma; \Delta_2 \vdash Q}{\Gamma; \Delta_1, \Delta_2 \vdash P \mid Q} \qquad\qquad \frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 + \Gamma_2 \vdash P \mid Q}$$

Type combination

$$\begin{aligned} t + t &= t && \text{if } \mathsf{un}(t) \\ {}^{\kappa_1,\kappa_2}[t] + {}^{\kappa_3,\kappa_4}[t] &= {}^{\kappa_1+\kappa_3,\kappa_2+\kappa_4}[t] \end{aligned}$$

Use combination

$$\begin{aligned} 0 + \kappa = \kappa + 0 &= \kappa \\ 1 + 1 &= \omega \\ \omega + \kappa = \kappa + \omega &= \omega \end{aligned}$$

Examples

- Input + Output `io.hy`

  ```
  a?x | a!3
  ```

- Output + Output `oo.hy`

  ```
  a!3 | a!4
  ```

- Output + Extrusion (and odd channels) `oe.hy`

  ```
  new a in { a!3 | b!a }
  ```

- Output + Extrusions `oee.hy`

  ```
  new a in { a!3 | b!a | c!a }
  ```

More examples

- recursive processes `fibonacci.hy`
- recursive types (sync/async/odd) `from.hy`
- recursive protocols (also odd) `from.hy`

Compiling pattern matching

The successor service succ1.hy

```
*succ?(x,r).r!(x + 1)
```

The successor service, compiled succ2.hy

```
*succ?p.(snd p)!((fst p) + 1)
```

Combining composite types

$$\begin{aligned} t + t &= t && \text{if } \mathrm{un}(t) \\ {}^{\kappa_1,\kappa_2}[t] + {}^{\kappa_3,\kappa_4}[t] &= {}^{\kappa_1+\kappa_3,\kappa_2+\kappa_4}[t] \end{aligned}$$

Two options

1. leave $+$ as it is $\Rightarrow$ **syntactic** linearity

Combining composite types

$$\begin{aligned} t + t &= t && \text{if } \mathsf{un}(t) \\ {}^{\kappa_1,\kappa_2}[t] + {}^{\kappa_3,\kappa_4}[t] &= {}^{\kappa_1+\kappa_3,\kappa_2+\kappa_4}[t] \\ (t_1 \times t_2) + (s_1 \times s_2) &= (t_1 + s_1) \times (t_2 + s_2) \end{aligned}$$

Two options

1. leave $+$ as it is $\Rightarrow$ **syntactic** linearity
2. distribute $+$ over composite types $\Rightarrow$ **operational** linearity

Example: combining pairs

$$(\texttt{int} \times {}^{0,1}[\texttt{int}]) + (\texttt{int} \times {}^{0,0}[\texttt{int}])$$

Example: combining pairs

$$(\texttt{int} \times {}^{0,1}[\texttt{int}]) + (\texttt{int} \times {}^{0,0}[\texttt{int}])$$

$$\Downarrow$$

$$(\texttt{int} + \texttt{int}) \times ({}^{0,1}[\texttt{int}] + {}^{0,0}[\texttt{int}])$$

Example: combining pairs

$$(\mathtt{int} \times {}^{0,1}[\mathtt{int}]) + (\mathtt{int} \times {}^{0,0}[\mathtt{int}])$$

$$\Downarrow$$

$$(\mathtt{int} + \mathtt{int}) \times ({}^{0,1}[\mathtt{int}] + {}^{0,0}[\mathtt{int}])$$

$$\Downarrow$$

$$\mathtt{int} \times {}^{0,1}[\mathtt{int}]$$

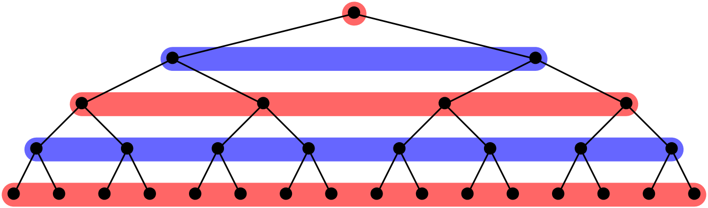
```
*succ?p.(snd p)!((fst p) + 1)
```

Example: trees

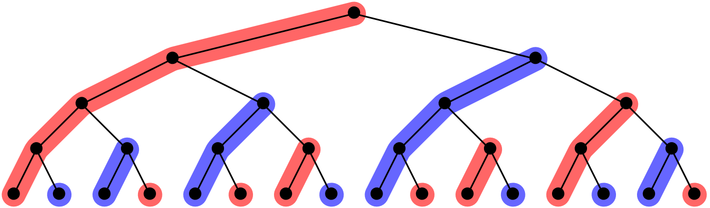
- even + odd `tree_even_odd.hy`

- flip + flop `tree_flip_flop.hy`

Channels used by even and odd



Channels used by `flip` and `flop`



Enforcing stronger properties

- a **linear** channel is a **promise** of communication

Enforcing stronger properties

- a **linear** channel is a **promise** of communication

Theorem (Kobayashi, Pierce, Turner, 1999)

A well-typed process performs at most one communication on each of its linear channels

Example

abba.hy

```
a?x.b!true | b?y.a!3
```

Deadlock and lock freedom

Definition

P is **deadlock free** if $P \rightarrow^* \texttt{new}\ \tilde{a}\ \texttt{in}\ Q \not\rightarrow$ implies $\neg\mathsf{wait}(a, Q)$

P is **lock free** if $P \rightarrow^* \texttt{new}\ \tilde{a}\ \texttt{in}\ Q$ and $\mathsf{wait}(a, Q)$ implies

$$Q \rightarrow^* R \text{ such that } \mathsf{sync}(a, R)$$

Note

- lock freedom implies deadlock freedom
- the converse also holds, for **finite** processes only

Example `later.hy`

```
new a in { *c?x.c!x | c!a | a!3 }
```

Tracking the use of channels, refined

Annotated channel types

$$^{\kappa_1,\kappa_2}[t]_k^h$$

Annotations

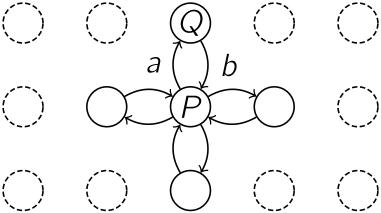
- level $h \in \mathbb{Z}$ $\Rightarrow$ **order** in which channel is used
- $k \in \mathbb{N}$ tickets $\Rightarrow$ max number of **delegations**

Examples

- deadlock `abba.hy`
- lock `later.hy`
- parallel recursive function (level polymorphism) `fibonacci.hy`

Example: half- and full-duplex communications

duplex.hy

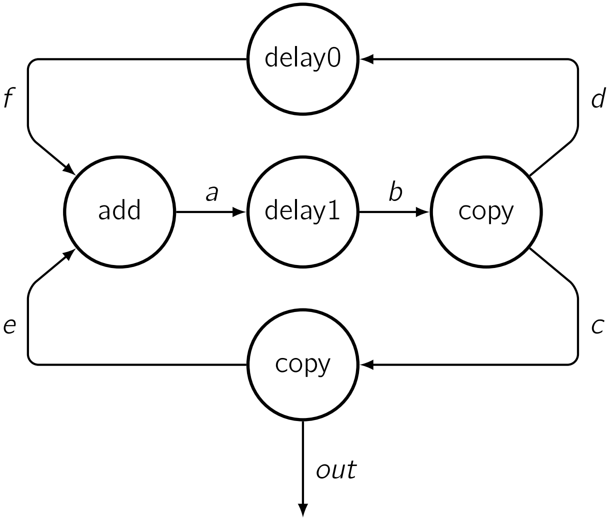



```
*full?(a,b).
 new c in
 { a!c
 | b?d.full!(c,d) }
```

```
*half?(a,b).
 b?d.
 new c in
 { a!c | half!(c,d) }
```

Example: Kahn process network

network.hy



Counter-examples

- some multiparty sessions `multiparty1.hy`
- sieve of Eratosthenes `sieve.hy`

Note: lock freedom can be unreasonable

- recursion is well-founded (Fibonacci)
- there exist infinitely many prime numbers (sieve)

Paperware and software

- Kobayashi, Pierce, Turner, **Linearity and the Pi Calculus**, TOPLAS 1999
- Igarashi and Kobayashi, **Type Reconstruction for Linear π-Calculus with I/O Subtyping**, Inf. & Comp. 2000
- Padovani, **Type reconstruction for the linear π-calculus with composite and equi-recursive types**, FoSSaCS 2014
- Padovani, **Deadlock and lock freedom in the linear π-calculus**, CSL-LICS 2014
- Padovani, Chen, Tosatto, **Type reconstruction algorithms for deadlock-free and lock-free linear π-calculi**, COORD. 2015
- **Hypha** (available at `http://di.unito.it/hypha`)

`Slides, papers, links on my home page`