# On Lazy Sessions and Productive Futures

Paula Severi

University of Leicester

Betty Meeting, 17 April 2015

joint work with Mariangiola Dezani, Luca Padovani and Emilio Tuosto

# Overview

- Aim and Motivation
- A typed lazy functional language with communication primitives and co-inductive data types.
  - Lazy evaluation
  - Input/Output in Lazy Programming Languages
  - Potentially Infinite Data and Productivity
  - Modal operator for ensuring productivity
- Contributions
- Related work

# Aim

A *functional programming language* that has

- communication primitives for sending and receiving *infinite data*
- *stream* processing
- building infinite data structures *interleaving input/output*
- the property of *productivity*

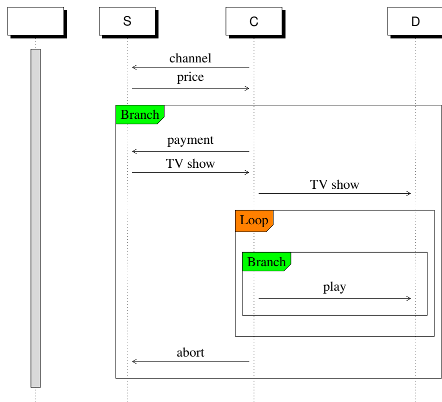
# Motivation

## Case Study: Pay-per-view

A customer C

1. buys TV channels streamed by S
2. watches them on her device D.

# Case Study: Pay-per-view




Session between client and device get stuck if TV show (potentially infinite list) not productive.

# Operational semantics for infinite data?

## Operational semantics for infinite data

~~Call-by-value~~

LAZY EVALUATION!!!

$$(\lambda x.e)\, f \longrightarrow e\{f/x\}$$

The argument is evaluated only when needed.

So far calculi with session types have been call-by-value.

# Is (communication primitives + lazy evaluation) possible?

## Separation of pure part from the input/output with side-effects

Type constructor `IO` from Haskell

Functions for communication are tagged with `IO` .

$$\begin{aligned} \texttt{send} &: \; !t.T \to t \multimap \texttt{IO}\; T \\ \texttt{recv} &: \; ?t.T \to \texttt{IO}\; (t \times T) \end{aligned}$$

# Type constructor IO

- The type constructor $\texttt{IO}$ does not have an elimination rule:

$$\xcancel{\mathsf{elim} : \texttt{IO}\ t \to t}$$

  a program contaminated with input/output remains contaminated.

- Only way to combine programs of type $\texttt{IO}$ is to use:

$$\texttt{bind} \quad : \quad \texttt{IO}\ t \to (t \multimap \texttt{IO}\ s) \multimap \texttt{IO}\ s$$

  for sequential composition.

  **Notation**. $\texttt{bind}\ e\ f$ abbreviated as $e >>= f$.

$$\texttt{bind}\ (\texttt{return}\ e)\ e' \quad \longrightarrow \quad e'\ e$$

- Canonical element.

$$\texttt{return} \quad : \quad t \to \texttt{IO}\ t$$

# Example

$$(\texttt{send}\ c^+\ 4)\ \ \texttt{>>=}\ \ f \mid (\texttt{recv}\ c^-)$$

$$\longrightarrow\ \ (\texttt{return}\ c^+)\ \ \texttt{>>=}\ \ f \mid \texttt{return}\ (\texttt{pair}\ 4\ c^-)$$

$$\longrightarrow\ \ f\ c^+ \mid \texttt{return}\ (\texttt{pair}\ 4\ c^-)$$

# Productivity = Infinitary Normalization

$$
\begin{aligned}
\mathsf{zeros} &\rightarrow (\mathsf{cons}\ 0\ \mathsf{zeros}) \\
&\rightarrow (\mathsf{cons}\ 0\ (\mathsf{cons}\ 0\ \mathsf{zeros})) \\
&\rightarrow (\mathsf{cons}\ 0\ (\mathsf{cons}\ 0\ (\mathsf{cons}\ 0\ \mathsf{zeros}))) \\
&\rightarrow \ldots \\
&\vdots \\
&\quad (\mathsf{cons}\ 0\ (\mathsf{cons}\ 0\ (\mathsf{cons}\ 0\ \ldots)))
\end{aligned}
$$

We are always producing some output.

# Syntactic Criteria to ensure productivity

- Guardedness Condition.
  - Used in the proof assistant Coq.
  - Recursive calls should be protected by constructors (Coquand,Types 1993).
  - Example.

    interleave xs ys = (head xs): ( interleave ys (tail xs))

- Pebbles
  - Based on infinitary rewriting systems (Endrullis et al TCS 2010).
  - Decidable.
  - More general than guardedness condition.
  - Example.

    zerosprime = 0: (interleave zerosprime zerosprime)

# Can we ensure infinitary normalization via typing?

- Temporal Modal Operator

  $\bullet A$ represents information that will be displayed in the next time (in the future).

  *H. Nakano. LICS 2000. Krishnaswami and Benton. LICS 2011.*

- Typing fixed point operator:

$$\texttt{fix} : (\bullet t \to t) \to t$$

An argument of fix is $f : (\bullet t \to t)$

1. $f$ is a "contractive function" (metric space semantics).
2. the recursive call $r$ in the expression $e$ of $f = \lambda r.e$ occurs at depth greater or equal than 1.

# Future

Way of postponing an IO action $e$

- Operational Semantics.

$$x \Leftarrow C[\mathtt{future}\ e] \quad \longrightarrow \quad \nu y.(x \Leftarrow C[\mathtt{return}\ y] \mid y \Leftarrow e)$$

- Typing.

$$\mathtt{future} : \bullet^n(\mathtt{IO}\ t) \to \mathtt{IO}\ (\bullet^n t)$$

# Programming Example

**Webcam storing a video**

$\texttt{store}\ x = \ \texttt{recv}\ x \ \texttt{>>=}$
$\lambda y.\texttt{split}\ y\ \texttt{as}\ y_1, y_2\ \texttt{in}\ \texttt{future}\ (\texttt{store}\ y_2)\ \texttt{>>=}$
$\lambda z.\texttt{<}(\text{cons}\ y_1\ z)\texttt{>}$

**Type of** $\texttt{store}$ **is** $(\texttt{SIS Nat}) \rightarrow \texttt{IO}\ (\texttt{Stream Nat})$

| | | |
|---|---|---|
| $(\texttt{Stream}\ t)$ | $= t \times \bullet(\texttt{Stream}\ t)$ | **Data Structure for Streams** |
| $(\texttt{SIS}\ t)$ | $= ?t.(\texttt{SIS}\ t)$ | **Session Type for Stream Process** |

>>= shorthand for bind

# Contributions

1. Exchange data include *infinite objects* (big data)
2. Lazy evaluation for sessions (communication on demand)
3. Treat `IO` as a linear type
4. Modal operator to ensure productivity of data that contains I/O
5. Properties:
    1. Productivity of data
    2. Processes are always successful:
       *every well-typed process eventually produces some data.*

# Related Work

*Gay and Vasconcelos JFP 2010*

- Similarity:

  primitive *functions* for { sending, receiving, opening a session

- Differences:
  1. Gay and Vasconcelos JFP 2010 is call-by-value
  2. For us, exchange values can be infinite

# Related Work

*Tonhino, Caires and Pfenning TGC 2014*

- Calculus for data types is independent and not presented
  we have a calculus where we mix data with communication
- Types do not ensure productivity
- their notion of productivity refers to processes (not data)

Similar differences with draft by Morris, Lindley and Wadler.