Session Types in Functional Languages

Vasco T. Vasconcelos
Universidade de Lisboa

Bernardo Toninho
Universidade Nova de Lisboa

Betty meeting
March 24, 2013

Session types in programming languages

- Session developed around the pi calculus
- Later transferred to different realms:
  - Object-oriented programming
  - Functional programming
  - Operating systems
  - Software services
  - Object broker systems

Session types in programming languages

- Used as descriptions for communication media in general

Session types on functional programing languages

- We distinguish three approaches:
  1. Session types in Haskell
  2. Functional language + channel primitives
  3. Functional language + process language
- and briefly address the last two

An Implementation of Session Types

Matthias Neubauer and Peter Thiemann*

Universität Freiburg
Georges-Köhler-Allee 079
D-79110 Freiburg, Germany

Abstract. A session type is an abstraction of a set of sequences of heterogeneous values sent and received over a communication channel. Session types can be used for specifying stream-based Internet protocols. Typically, session types are attached to communication-based program calculi, which renders them theoretical tools which are not readily usable in practice. To transfer session types into practice, we propose an embedding of a core calculus with session types into the functional programming language Haskell. The embedding preserves typing. A case study (a client for SMTP, the Simple Mail Transfer Protocol) demonstrates the feasibility of our approach.

PADL 2004

Haskell Session Types with (Almost) No Class

Riccardo Pucella Jesse A. Tov
Northeastern University
{riccardo,tov}@ccs.neu.edu

Haskell 2008

Session Types in Haskell

Updating Message Passing for the 21st Century

Matthew Sackman Susan Eisenbach

Imperial College, London

{ms, sue}@doc.ic.ac.uk

A full implementation of Session Types in Haskell

Keigo Imai Shoji Yuen Kiyoshi Agusa
Graduate School of Information Science, Nagoya University
imai@nagoya-u.jp, yuen@is.nagoya-u.ac.jp, agusa@is.nagoya-u.ac.jp

Call-by-value functional multi-threaded programming

- Lambda: basic values, variables, abstraction, application and pairs
- Communication channels: creation, sending/receiving/selecting/branching on a channel
- Forking new threads

Typechecking a Multithreaded Functional Language with Session Types ⋆

Vasco T. Vasconcelos

Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa, 1749-016 Lisboa, Portugal

Simon J. Gay

Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK

António Ravara

CLC and Departamento de Matemática, Instituto Superior Técnico, 1049-001 Lisboa, Portugal

TCS 2006

JFP **20** (1): 19–50, 2010. 19
doi:10.1017/S0956796809990268 First published online 8 December 2009

Linear type theory for asynchronous session types

SIMON J. GAY
Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK
(*e-mail:* `simon@dcs.gla.ac.uk`)

VASCO T. VASCONCELOS
Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa, 1749-016 Lisboa, Portugal
(*e-mail:* `vv@di.fc.ul.pt`)

I&C 2010

Propositions as Sessions

Philip Wadler
University of Edinburgh
wadler@inf.ed.ac.uk

ICFP 2012

Example: The petition server

- The type governing the interaction with the petition server, as seen from the side of the client
- First, "interactively" set up the title and the closing date for the reception of the signatures

$$\text{Petition} = \oplus\{\textit{setTitle}\text{: !string.Petition}, \\ \textit{setDate}\text{: !date.Petition}, \\ \textit{submit}\text{: ...}\}$$

Submitting a proposal

- Once happy, the petiton writer commits the title+date.
- If the petition proposal is accepted by the server, then the promotion phase begins

Petition = ⊕{...,
submit: &{*accepted*: Promotion,
denied: ?string.**end**}}

The promotion phase

- During the promotion phase all one can do is to sign the petition by sending a signature

Promotion = !string .Promotion

The linear and the unrestricted phases

- The set up part is linear, we want no interferences

Petition = lin⊕{*setTitle*: lin!string.Petition,
setDate: lin!date.Petition,
submit: lin&{*accepted*: Promotion,
denied: lin?string.end}}

- The promotion is unrestricted, we seek as many signatories as possible

Promotion = un!string.Promotion

The well-known type of the petition channel

- The channel as seen from the client's side

$$\text{PetitionServer} = \text{un}?\text{Petition}.\ \text{PetitionServer}$$

- Abbreviated to

$$*?\text{Petition}$$

Creating and distributing the petition channel

```
main :: unit → unit
main _ =
   split new *!Petition as ps1, ps2 in
   fork (petitionServer ps1);
   fork (saveTheWolf ps2)
```

Code for the server

```
petitionServer :: *!Petition → unit
petitionServer ps =
   split new Petition as p1, p2 in
   ps!p1;
   fork (setup p2 (1,1,1970) "Save me");
   petitionServer ps
setup :: dual(Petition) → date →
   string → unit
setut p d t =
   p ▷ {setDate: setup p (p?) t,
        setTitle: setup p d (p?),
        submit: p ◁ accepted;
                promotion p []
        }
promotion :: *?string →
   stringList → unit
promotion p l =
   promotion p ((p?):: l)
```

Functions and Processes

- A monadic integration of functions and (session-typed) processes:

{c:S <- d:T} :: Functional type for a proc. c:S, using d:T
{ c <- P <- d } :: Functional term for a proc c:S, using d:T

- A linear extension to a general functional PL
- Processes can communicate functional terms so...
- Higher-order, mobile (open) processes!

Streams as Processes

- Output an infinite sequence of integers, starting at n.
- A recursive session type:

  stype intStream = !int.intStream

- Write a recursive session using a recursive function:

```
nats : int -> {c:intStream}
  c <- nats n =
    { _ <- output c n
        c <- nats (n+1) }
```

Streams as Processes

- Output an infinite sequence of integers, starting at n.
- A recursive session type:

  stype intStream = !int.intStream

- Write a recursive session using a recursive function:

```
filter : (int -> bool) -> { d:intStream <- c:intStream }
d <- filter q <- c =
{ x <- input c
  case (q x) of
     true => _ <- output d x
             d <- filter q <- c
   | false => d <- filter q <- c}
```

Higher-Order Processes

- Monadic values can be communicated by processes.
- An App Store Session:

```
stype AppStore = Choice{
                    weather:
                        !{c:Weather <- d:API, e:GPS}.end
                    travel:
                        !{c:Travel <- d:API}.end
                    game:
                        !{c:Game <- d:API}.end}
```

Higher-Order Processes

- The App Store code:

```
c <- Store W  Tr G =
{  case c of
      weather =>     _ <- output c W
                     close c
      travel =>      _ <- output c  Tr
                     close c
      game =>        _ <- output c G
                     close c}
```

Higher-Order Processes

- The App Store Client, running the Weather App:

```
c <- WeatherClient() <- a:AppStore, d:API =
{ _ <- a.weather
  w <- input a
  g  <- ActivateGPS()
  c <- w <- d, g }
```