A Practical Approach To Session Types

Debjani Banerjee

March 14, 2015

Abstract

Over the course of implementing a type checker for Session Types based the important question of incorporating these concepts practically was unresolved. This write-up addresses the need for incorporating session types into a programming language and discusses the incorporation of the same for the POP3 Protocol.

1 Introduction

Concurrent interactions are the norm for most computational models these days, ranging from multiprocessing on stand-alone systems to the emerging area of cloud-computing systems. Unfortunately no practical language has been successfully adopted for representing parallel computation.

Session types are one of the way that have been proposed to structure interactions over communicating processes and their behavour. Session types support structured patterns of communications, so that the communication behaviour of agents in a distributed system can be verified by static type checking.

The central construct of session types is the channel, i.e., channel of communication. The two types of channels are linear and shared types. Linear channels are known two two interacting parties alone. They serve as ideal representations of private communication over a network. An example of the same would be a person sending an email. The process responsible for sending the email should only be aware of the sender and the recipient. Shared channels can be shared by 0 to N parties. An example of the same would be a website handling multiple users simultaneously. The paper on Fundamental Session Types uses type qualifiers to allow us to distinguish between the two and simplify the type theory for the same.

Previously work on concurrent computation used a mathematical formalism called pi calculus to represent distributed communication. Programming languages have sought to include the basic ideas and representations in pi calculus to represent protocols and communicating processes.

We will begin by introducing Pi Calculus as a means of representing concurrent communication. Post this we will informally introduce pi calculus with simple examples. The formal syntax of types and algorithmic type checking logic is presented followed by a review of how session types can be used to specify protocols, e.g., POP3.

2 The Pi Calculus

Pi calculus is a formal system for representing dynamic process communication. Functional programs can be represented in terms of the same, hence this is an ideal place to start if we plan on creating a practical implementation of session types. The following are some of the constructs of the calculus-

P :: =	Processes
0	Inaction
$\overline{x}$ v.P	Output
x(y).P	Input
P | Q	Parallel Composition
if v then P else Q	Conditional
(γ x y)P	Scope Restriction
x	Variable
true/false	Boolean

Fig 1: Syntax of process

The base types in the system are boolean values true and false. These are reserved in the grammar.

Lower case letters represent variables. A variable can represent a boolean or a channel.

Channels are the construct on which communication is based. Communication in general implies two parties, a sender and a receiver. A channel is therefore represented by two endpoints, x representing the end of the channel which receives a value and $\overline{x}$, the endpoint of the channel that outputs a value. The variables x and $\overline{x}$ (where x and $\overline{x}$ can be replaced by any lower-case letter) are called co-variables. A thread need only access one of x or $\overline{x}$ in order to send/receive a value respectively.

Upper case letters represent the processes themselves. P, Q, R are examples of processes.

The simplest process is 0, representing no action or the end of an execution.

The two main worker processes are the input and the output.

The input process x v.P represents a process that reads for a channel x. The value read from channel endpoint x is then used to replace bound variable v. Post this process P is executed.

The output process $\overline{x}$ v.P represents a process that writes a value v on a channel $\overline{x}$,. After it has completed doing the same it will proceed with process P.

Parallel Composition is represented by P | Q. Processes P and Q are asynchronous concurrently executed processes. If there are dependencies between the two processes they must be on a shared channel else there will be a unfulfilled resource dependency in one of the processes.

Scope Restriction, represented by (γ x y).P can be seen as a process allocating channels x and y within P. The constants of π-calculus are defined by their names only and are always communication channels. Creation of a new name in a process is also called restriction.

The If construct is intuitive. The if v then P else Q implies that based on whether the value of v is true or false we either execute process P or process Q.

3 Session Types With Examples

The following concise and relevant translations from type semantics to session types have been adapted from Vasconcelos' paper on IPC Session Type communication.

3.1 Input and Output

Consider a server that only performs addition. A protocol is described where the client sends two integers and theserver returns their sum. The client and the server are independent channels. Channels are used to describe the communication and are dyadic in nature, supporting both input and output. The addition server can be represented formally as -

$$S = ?Int.?Int.!Int.End$$

Where ? is recieve, ! is send, . represents a sequence A.B where A occurs before B and End represents the end of an execution.

In our hypothetical programming language the same can be expressed as -

```
let x = recieve c
let y = recieve c
in send x+y on c
```

If we interchange the operations we obtain the client side of the protocol -

$$S = !Int.!Int.?Int.End$$

```
send 2 on c
send 3 on c
x = recieve c
```

Sending the two integers on the channel together is represented formally by !(Int x Int).?Int.

3.2 Dual Types

The client and the server side of the expression can be obtained by exchanging ! and ?. Send and receive are two types that complement each other and are hence referred to as dual types.

A client !Int.?Int sends a number and then receives another number. The server can be represented by the dual type ?Int.!Int, which receives the number from the client and sends and integer.

3.3 Branching

Using the branching (& operator) we are able to represent selection. The server uses the branching operator to offer the choice of adding, negation or termination of the operations.

Formal Representation:

$$\&\langle \text{add: ?Int.?Int.!Int.End, neg : ?Int.!Int.End} \rangle$$

Programmatic Representation:

```
case c of {
add -> send ((receive c) + (receive c)) on c
neg ->  send (receive c) on c }
```

3.4 Choice

The client side needs to be able to choose among the options available. The same is provided by the choice operator.

Formal Representation:

$$\oplus\langle \text{add: !Int.!Int.?Int.End, neg : !Int.?Int.End} \rangle$$

Programmatic Representation:

```
select add on c
send 2, 3 on c
let x = recieve c in code
```

Choice and Branching are also dual types. To represent the client side code all we have done is replace & with $\oplus$ and exchanged the appropriate ! and ? symbols.

3.5 Parallel Execution

Session types are focused on concurrent processes. There is an environment mantained containing the session variables and types associated with them. If the variables are linear(lin) they must be used one time only; unrestricted(un) can be used N number of times.

Formal Representation:

$$\text{add: !Int.!Int.?Int.End} \mid \text{neg : !Int.?Int.End}$$

Programmatic Representation:

```
Thread t1 = select add on c; send 2, 3 on c;
Thread t2 = select neg on c; send 2 on c;
t1.start();
t2.start();
```

3.6 Recursive Types

A realistic server is one that performs multiple operations. In order to represent the same we introduce the concept of recursive types. An example of the same is

S = &<add: ?Int.?Int.!Int.S;
sub : ?Int.?Int.!Int.S;
quit: End>

Here S is a recursive type since after the operation of adding/subtracting is performed it makes a callback to itself. Quit here is a base case after which the server operations end. Unrestricted access to a particular channel is represented using recursive types.

We can represent the server programmatically as follows -

```
serve c =
case c of {
add -> send ((receive c) + (receive c)) on c
serve c
sub -> send ((receive c) - (recieve c)) on c
serve c
quit -> close c
}
```

3.7 Replication

In the previous example the servers channel C can be used multiple times by multiple operations. For the add operation the client sends 2 integers to the server, which the server adds together and yields the results on the same channel. However you could have a different client accessing the sub function and trying to subtract the two numbers respectively. Such a channel is unrestricted and is said to be replicated i.e. shared among the various clients. If we need some secure data to be transmitted over the channel and want to make sure that once the data is transferred it can never be accessed again. We do so by specifying the channel with a linear qualifier 'lin' which means that once the operation on the channel is completed the channel cannot be used again by any other process.

Only a single function authenticate is defined, which accepts a linear channel c1. The client sends a password and recieves a boolean response. This can be expressed formally as channel

Formal Representation -

lin (?string).(!bool)

Programmatic Representation -

```
authenticate -> send((recieve c1)) on c1
```

4 Typing

The types of the previous system are described formally below -

q ::=	Qualifiers
lin	linear
un	unrestricted
p ::=	Pretypes:
?T:T	receive
!T:T	send
T ::=	Types:
bool	boolean
int	integer
string	string
end	termination
q p	qualified pretype
Γ ::=	Contexts:
ϕ	empty context
Γ;x: T	assumption

A majority of these types have been defined in the previous section. The only new introduction is the context. A context is a key value mapping between values and their associated types. Given a particular context we can evaluate if the language is well-typed or not. Evaluation of the context was the basis of the algorithmic type-checker implemented for fundamental session types. Contexts have different rules defined to handle linear and shared channels.

$$\Gamma \div \phi = \Gamma$$

On removing null from a given context we obtain the previous context itself. The other two however, are crucial to our understanding of linear and shared channels.

$$\frac{\Gamma_1 \div L = \Gamma_2, x{:}T, un(T)}{\Gamma_1 \div (L{,}x) = \Gamma_2} \quad \Bigg| \quad \frac{\Gamma_1 \div L = \Gamma_2, x \notin \Gamma_2}{\Gamma_1 \div (L{,}x) = \Gamma_2}$$

Given a particular context with Γ_1, if we try to update the context after using a list of linear variables L along with a variable x we get a new contect Γ_2. If x is unrestricted then it is possible to reuse x and the type T of x is unrestricted. Else, x is removed from the context and we only get Γ_2. This understanding forms the basis of the algorithmic type system we will develop for the same.

Typing rules for values $\Gamma \vdash v : T; \Gamma$

$$\Gamma \vdash true : bool; \Gamma \qquad \Gamma \vdash false : bool; \Gamma$$

$$\frac{un(T)}{\Gamma_1, x : T, \Gamma_2 \vdash x : T; (\Gamma_1, x : T, \Gamma_2} \qquad \Gamma_1, x : linp, \Gamma_2 \vdash x : linp; (\Gamma_1, \Gamma_2)$$

The boolean values are constants in our language and leave the context unchanged. You can interpret the rule as saying that given a context with a set of key, value pairs, we evaluate the context for boolean values. As boolean values are constants, the resultant context Γ on the right of the statement remains unchanged.

For unrestricted and linear values the context update rules apply, i.e., the context remains unmodified for unrestricted variables and for linear variables the context is updated by removing the used linear variable.

$$\Gamma \vdash 0 : \Gamma; \phi \text{ [A-INACT]}$$

For an inactive process, the context remains unchanged after the process is executed.

$$\frac{\Gamma_1 \vdash P : \Gamma_2; L_1 \;\; \Gamma_2 \div L_1 \vdash Q : \Gamma_3; L_2}{\Gamma_1 \vdash P|Q : \Gamma_3; L_2} \text{ [A-POS]}$$

Parallel composition is important as the context must be split between processes P and Q. The process P's context Γ_1 is evaluated and the resultant context Γ_2 is then fed to process Q. We must make sure that context Γ_2 must not contain any of the free linear variables consumed by Γ_1. The resultant rule [A-PAR] defines an output context Γ_3 with a set of consumed linear variables L_2

$$\frac{\Gamma_1, x : T, y : \overline{T} \vdash P : \Gamma_2; L}{\Gamma_1 \vdash (\gamma xy : T)P : \Gamma_2 \div x, y; L - \{x, y\}} \text{ [A-RES]}$$

Scope Resolution requires the variables to be dual as defined by T and $\overline{T}$. The variables can only be used within the scope. The resultant context Γ_2 removes the x and y variables.

$$\frac{\Gamma_1 \vdash \text{v :q bool ; } \Gamma_2 \quad \Gamma_2 \vdash P : \Gamma_3, L \quad \Gamma_2 \vdash Q : \Gamma_3, L}{\Gamma_3 \vdash \text{if v then P else Q} : \Gamma_3; L} \text{ [A-IF]}$$

Evaluating A-IF is straight-forward. One must note that the v variable used must be a boolean value. The same context Γ_2 is fed to processes P and Q, which will update the context to Γ_3 and update the list of free linear types.

$$\frac{\Gamma_1 \vdash x : q!T.U; \Gamma_2 \quad \Gamma_2 \vdash U : T; \Gamma_3 \quad \Gamma_3 + x : U \vdash P : \Gamma_4; L}{\Gamma_1 \vdash \bar{x}v.P : \Gamma_4; L \cup (\text{ if q} = \text{lin then } \{\text{x}\} \text{ else } \phi)} \text{ [A-OUTCHAN]}$$

For an output channel we check if the qualifier q of the channel is linear. If so we add it to the list of consumed linear variables, else we maintain the list as is.

$$\frac{\Gamma_1 \vdash x : q?T.U; \Gamma_2 \quad (\Gamma_2, y : T) + x : U \vdash P : \Gamma_3; L \quad q = un \Rightarrow L = \phi}{\Gamma_1 \vdash qxy.P : \Gamma_3 \div y; L \cup (\text{ if q} = \text{lin then } \{\text{x}\} \text{ else } \phi)}$$
[A-INCHAN]

The input channel is similar to the output as both of them are co-variables. If the qualifier for the linear channel is unrestricted it can be used as many times as possible and the list of free linear variables will be null.

I will not be elaborating on type checking for branching and selection as this was implemented during typechecking. However the basic idea of choice and branching holds. The variables introduced in the appropriate branch/choice statement will be consumed while the rest of them will remain unmodified in the context.

5 Session Types and POP3

POP3 is a standard protocol for sending email messages over a TCP-IP network. It begins with the client issuing a START message to the server. The server then sends an AUTH request to the server to which the client responds with a USERNAME and PASSWORD. Note that AUTH can be modelled on a linear channel. After authorization the client and server go through a set of transactions. The transactions mentioned are the STATUS, RETRIEVE and QUIT commands.

$$\begin{aligned} \text{START} &= <!\text{String.AUTH} > \\ \text{AUTH} &= \text{lin } (?\text{Password}).(\oplus(!\text{Success.}!\text{Failiure })) \\ \text{TRANSACTION} &= \&< \text{STATUS: } ?\text{Message.}!\text{String.TRANSACTION}; \\ &\quad \text{RETRIEVE: } ?\text{Message.}!\text{String.TRANSACTION; QUIT: End} > \end{aligned}$$

Note that the format that I have used is informal. Using this syntax I aim at specifying an informal way of representing the state transitions that occur on the server side. The steps are as follows. On calling START on the server the server returns String saying that it is ready. AUTH is started on a linear channel which accepts a password and returns success/failure accordingly. On Success we can request from a choice of transactions. The server expects a message from the client. The TRANSACTION state is recursive and can be represented by an unrestricted channel. We can get the status of a message or retrieve metadata related to the same. This is a simplistic version of the types mentioned in the paper.

References

Dezani-ciancaglini, Mariangiola, Elena Giachino, and Luca Padovani. *General Session Types.*

Gay, Simon J., and Vasco T. Vasconcelos. *Linear Type Theory for Asynchronous Session Types*, 2008.

Gay, Simon, Vasco Vasconcelos, and Antonio Ravara. *Session Types for Inter-Process Communication.* Technical report. 2003.

Neubauer, Matthias, and Peter Thiemann. "An Implementation of Session Types." In *In PADL, volume 3057 of LNCS*, 56–70. Springer, 2004.

Pucella, Riccardo, and Jesse A. Tov. "Haskell Session Types with (Almost) No Class." *Proc. ACM SIGPLAN 2008 Haskell Symposium* 13 (2008): 25–36.

Vasconcelos, Vasco T. "Fundamentals of session types." *Information and Computation* 21 (2012): 52–70.