Session Types for Linear Multithreaded Functional Programming [*]

Vasco T. Vasconcelos
Department of Informatics, Faculty of Sciences, University of Lisbon, Portugal
vv@di.fc.ul.pt

Abstract

The construction of reliable concurrent and distributed systems is an extremely difficult endeavour. For complex systems, it requires modular development strategies based on precise interface specifications that allow the various modules to interact properly. In this extended abstract we are concerned with message passing systems where partners engage in long and complex interactions, as opposed to, say, remote procedure calls composed of a pair of simple interactions.

Session types allow for the description of continuous series of interactions between several partners. In the simpler case, they detail protocols between two partners [Honda et al. 1998]; recently the original setting was widened to encompass multiple partners [Honda et al. 2008].

In this paper we deal with binary sessions only. Through a running example we visit session types and a functional concurrent language equipped with buffered semantics. Apart from the traditional "well typed programs do not go wrong", the semantics proposed allows for two extra interesting results: the ability to predict the required buffer size, and that of anticipating an output with respect to an input operation.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages, Theory

Keywords Session types, Functional programming, Linear type systems, Concurrency

1. Describing Protocols

Our running example consists of a simplified distributed auction system with three kinds of players, taken from [Vallecillo et al. 2006]:

- Sellers that want to sell items,
- Auctioneers that sell items on their behalf,
- Bidders that bid for an item being auctioned.

Protocols as types The protocol for the seller looks like this.

$$\begin{aligned}\mathsf{Seller} = \oplus\{&\mathit{selling}\colon !\,\mathsf{Item}\,.\,!\,\mathsf{Price}\,.\\ &\&\{\mathit{sold}\colon ?\,\mathsf{Price}\,.\,\mathbf{end}\,,\ \mathit{notSold}\colon \mathbf{end}\}\}\end{aligned}$$

The seller starts by choosing the *selling* operation on the auctioneer (*selling* is one of the various operations the auctioneer provides); *choosing an alternative* is indicated by $\oplus$. It then sends an item (the item to be sold), followed by the initial price; *value sending* is indicated by $!$. Following that, the seller expects an answer from the auctioneer; it can come in one of two forms, *sold* or *notSold*; *offering alternatives* is indicated by $\&$. Together with a sold confirmation comes the value the item was sold for; *value reception* is indicated by $?$. *Protocol completion* is indicated by **end**.

The auctioneer, while interacting with a seller, follows the below protocol.

$$\begin{aligned}&\mathsf{AuctioneerWithSeller} = \&\{\mathit{selling}\colon ?\,\mathsf{Item}\,.\,?\,\mathsf{Price}\,.\\ &\quad\oplus\{\mathit{sold}\colon !\,\mathsf{Price}\,.\,\mathbf{end}\,,\ \mathit{notSold}\colon \mathbf{end}\}\}\end{aligned}$$

It starts by offering the *selling* option, after which it receives an item and a price, following which it chooses one of two alternatives: *sold* or *notSold*. Selling and buying are complementary activities, and so are the types that govern them; we call them *dual*. We can easily see that when the seller says $\oplus$, the auctioneer says $\&$, and similarly for $?$ and $!$. Type **end** is dual of itself.

Now the protocol that bidders are supposed to follow.

$$\begin{aligned}\mathsf{Bidder} = \oplus\{&\mathit{register}\colon !\,\mathsf{Name}\,.\,?\,\mathsf{Item}\,.\,?\,\mathsf{Price}\,.\\ &\oplus\{\mathit{buy}\colon \mathbf{end}\,,\ \mathit{notInterested}\colon \mathbf{end}\}\}\end{aligned}$$

Bidders start by selecting option *register*, then send their name and wait for an item on sale and its price. They finally reply to the auctioneer manifesting their interest in the product, selecting the *buy* or the *notInterested* option. The type of auctioneers while interacting with bidders is the dual of the bidders' type.

$$\begin{aligned}&\mathsf{AuctioneerWithBidder} = \&\{\mathit{register}\colon ?\,\mathsf{Name}\,.\,!\,\mathsf{Item}\,.\\ &\quad !\,\mathsf{Price}\,.\,\&\{\mathit{buy}\colon \mathbf{end}\,,\ \mathit{notInterested}\colon \mathbf{end}\}\}\end{aligned}$$

One might wonder where the actual bidding is happening. Well, it does not. This is version 0.0 of our system. Bidders are given a price and must take it or leave it.

Compatibility The auctioneer seems to require two types: one for interacting with sellers, the other for interacting with bidders. The type of the auctioneer proper conjoins the two types.

$$\begin{aligned}&\mathsf{Auctioneer} =\\ &\quad\&\{\mathit{register}\colon ?\,\mathsf{Name}\,.\,!\,\mathsf{Item}\,.\,!\,\mathsf{Price}\,.\\ &\qquad\quad\&\{\mathit{buy}\colon \mathbf{end}\,,\ \mathit{notInterested}\colon \mathbf{end}\}\,,\\ &\qquad\mathit{selling}\colon ?\,\mathsf{Item}\,.\,?\,\mathsf{Price}\,.\\ &\qquad\quad\oplus\{\mathit{sold}\colon !\,\mathsf{Price}\,.\,\mathbf{end}\,,\ \mathit{notSold}\colon \mathbf{end}\}\}\end{aligned}$$

PPDP'09, September 7–9, 2009, Coimbra, Portugal.
Copyright © 2009 ACM 978-1-60558-568-0/09/09...$10.00

The particular types used to interact with sellers and with bidders are *subtypes* of this type. Advantages include:

- Bidders do not need to know the protocol for sellers,
- The code for bidders may be developed before the introduction of (online) sellers in the auction system.

The bidder-auctioneer *compatibility* rests assured: the type for the auctioneer, Auctioneer, is a supertype of AuctioneerWithBidder which is dual of the type Bidder of the bidder.

System evolution Evidence collected during the operation of the auctioneer system version 0.0 suggests that the most common complaint come from sellers and was related to the inability of lowering the initial price after an unsuccessful auction. After an upgrade, the new auctioneer now provides a third choice, in addition to *sold* and *notSold*. The new option, *lowerYourPrice*, reads "We are very excited about your item; would you consider lowering the price?".

```
&{selling: ? Item . ? Price . Selling ,  register: . . . }
where  Selling  =
  ⊕{ sold : ! Price . end ,
     notSold: end ,
     lowerYourPrice:&{ ok : ? Price . Selling , noWay: end } }
```

The obvious question arises: is compatibility still assured? The old seller still works with the new auctioneer, it just does not use the new functionality. The new type is far more complex than the original: additional recursion and one more ⊕ choice. Expanding recursion we see that all there remains is really one more choice: the new type is a *subtype* of the old type.

```
&{selling: ? Item . ? Price .
  ⊕{ sold : ! Price . end ,
     notSold: end ,
     lowerYourPrice:&{ ok : ? Price . Selling ,
                       noWay: end } } }
```

How does subtyping works on session types? Guided by the intuition sketched above, subtypes offer less alternatives (&) and choose more options (⊕). For value input (?), subtypes may take a subtype as parameter, and for value output (!), supertypes. In short, input is covariant on the argument and on the set choices, and output contravariant. In all cases continuations are covariant. Recursion is "unfolded away" as needed, a co-inductive definition.

Sessions Protocols such as the seller-auctioneer-bidder run between exactly two partners at a time:

- seller-auctioneer, or
- auctioneer-bidder.

Each such run is called a *session*.

Channels An auctioneer must be able to conduct multiple sessions in parallel, with different sellers and with different bidders. And it must not mix sessions, e.g., announcing *sold* to bidder A while sending the corresponding Price to bidder B. Each session is conducted on a different bi-directional communication medium called channel.

How are such channels created? By using other channels known to all participants potentially interested on online auctions, e.g., distributed on the world-wide-web. We could distinguish linear channels known by one partner, from shared channels known by any number of partners, but we prefer to work with a single kind of channel and *distinguish linear from unrestricted (shared) channel operations*. This gives us greater flexibility and a simplified theory. The annotated type of a buyer is as follows.[1]

[1] The only interesting **end** is unrestricted.

```
Buyer  =  lin⊕{selling: lin ! Item . lin ! Price .
                      lin&{ sold : lin ? Price . un  end ,
                            notSold: un  end } }
```

To lighten the syntax in examples, we henceforth omit all unrestricted qualifiers and only annotate linear types.

Now we can go back to session establishment. The common knowledge between the three kinds of partners is a shared channel, used to establish linear sessions. Recall the type Auctioneer of the auctioneer's session above. Then, the type of the shared channel, as seen by the auctioneer, is

```
T  =  un ! Auctioneer . T
```

where the !Auctioneer part is responsible for establishing *one* session and the T part for establishing *more* sessions. This idiom is so common that we abbreviate it to ∗!Auctioneer.

Subtyping in session types was introduced by [Gay and Hole 2005]. The **lin**/**un** session type qualifiers where introduced in [Vasconcelos 2009] (inspired by [Walker 2005]), which discusses the apparent variance inversion of subtyping with respect to the usual practice in the lambda-calculus.

2. Programming

The next step is to program our example. In which language shall we do it? Functional, imperative, object-oriented? You'll find all flavours in the literature (see references in [Vasconcelos 2009]). The language must however must incorporate concurrency and threads communicating via message passing; we shall use a call-by-value concurrent functional language.

Let us start with the code for a particular seller. Suppose that c is the name of the channel leading to the auctioneer.

```
sellerSession  ::  Seller → unit
sellerSession  c  =
  select  selling  on  c ;
  send  "psp"  on  c ;
  send  100  on  c ;
  case  c  of  {
    sold ⇒  receive  x  from  c  in
        print  ("made "  ^  x  ^  "euros!") ,
    notSold ⇒  print  ("next  time  I'll  ask  99.9!")
  }
```

The auctioneer starts by choosing the *selling* alternative: choosing an alternative is indicated by the **select**-**on** primitive operation. It then sends the item to be sold and the initial price, using two **send**-**on** expressions. The protocol continues with the seller accepting two alternatives, *sold* and *notSold*, expressed by the **case**-**of** operation. The *sold* branch starts with the reception of the actual price the item was sold for, and indicated by the **receive**-**from** expression. Notice the one-to-one correspondence between the operations on channel c and the type Seller above.

The code for a bidder, below, should be easy to understand.

```
bidderSession  ::  Bidder → unit
bidderSession  c   =
  select  register  on  c ;  send  "vasco"  on  c ;
  receive  item  from  c  in
  receive  price  from  c  in
  if  item  =  "psp"  and  price  <  100
  then  select  buy  on  c
  else  select  notInterested  on  c
```

Again, collecting all the operations on channel c yields the type Bidder above. Notice that in both pieces of code channel c is consumed to the **end**, as expected: the initial part of the type for the channel is linear and must be consumed; the last part (**end**) is unrestricted and may remain.

Finally the auctioneer, the most sophisticated piece of code; to be continued below.

```
AuctioneerSession :: Auctioneer → ∗?Repository
  → unit
AuctioneerSession c =
  case c of {
    selling ⇒  −− handle sellers ' requests
    register ⇒  −− handle bidders ' requests
  }
```

Bootstrapping How do sellers and bidders start sessions? By requesting a new, private, channel on the auctioneer's public, shared, name. Notice that by subtyping both functions seller and bidder may be given arguments of a type dual to that of the auctioneer, which we denote by **dual**(∗!Auctioneer) = ∗?(**dual** Auctioneer).

```
seller :: ∗?Seller → unit
seller a = receive c from a in sellerSession c
bidder :: ∗?Bidder → unit
bidder a = receive c from a in bidderSession c
```

How do auctioneers start sessions? By creating a fresh channel and sending it to clients. We shall distinguish two identifiers for the same underlying channel, two channel endpoints: d is the endpoint to be used by the client (bidder or seller), c is the endpoint used by the auctioneer itself.

```
auctioneer :: ∗!Auctioneer → ∗?Repository → unit
auctioneer a r =
  channel c,d in send d on a;
  fork (case c of {selling ⇒ sell c r,
                   register ⇒ regist c r});
  auctioneer a r
```

The shared repository Concentrate on the selling option; some pseudo-code first.

```
receive item from c in
receive price from c in (put item price);
if (wasSold item)
then select sold on c; send (getPrice item) on c
else select notSold on c
```

We have to program functions put, wasSold, and getPrice. But these three functions deal with a *shared* repository, hence its accesses must be governed by a protocol. For example the put function becomes

```
put :: Item → Price → ∗?Repository → unit
put item price r =
  receive d from r in select put on d;
  send item on d; send price on d
```

where r is the shared name for the repository. Operations wasSold and getPrice must be dealt together to take advantage of the case construct.

```
checkIfSold :: Item → lin⊕{sold:...} ⊸
  ∗?Repository ⊸ unit
checkIfSold item c r =
  receive d from r in
  select wasItSold on d; send item on d;
  case d of {
    notSold ⇒ select notSold on c,
    sold ⇒ select sold on c;
      receive price from d in send price on c}
```

Sessions c and d are now mixed, but the types remain apart. The type of d is obtained by gathering the operations on channel d (and forgetting those on c). All our types are (lin or un) qualified: $T \multimap U$ is an abbreviation for lin$\,T \rightarrow U$. Function checkIfSold would not type check under type Item →**lin**⊕{*sold*:...} → ... for the lambda abstraction λr. **receive** ... is typed under an environment containing a linear entry c: **lin**⊕{*sold*:...}. Hence the corresponding linear function constructor ⊸.

Using functions put and checkIfSold, we can easily write the *selling* branch of the auctioneer,

```
sell :: lin?Item.lin?Price.lin⊕{sold:...} ⊸
  ?∗Repository ⊸ unit
sell c r =
  receive item from c in
  receive price from c in
  put item price r;
  checkIfSold item c r
```

yielding the following type for the session conducted by the shared repository.

```
Repository =
  lin⊕{ wasItSold: lin!Item.
         lin&{sold: lin?Price.end, notSold: end},
       put: lin!Item.lin!Price.end}
```

Session delegation Noticed the copy-cat in function checkIfSold? Option *notSold* is forwarded from the repository (d) to the client (c), and so is option *notSold*, and so is price. Why not trust the seller's channel to the repository? The repository takes care of replying directly to the client.

```
checkIfSold item c r =
  receive d from r in select wasItSold on d;
  send item on d; send c on d
```

How does the type of the repository with delegation look like? The linear part, the part that governs the session the public name establishes, is

```
Repository = lin⊕{ wasItSold: lin!Item.!U.end}
                   put: lin!Item.lin!Price.end,
where U = lin&{sold: lin?Price.end, notSold: end}
```

where one can easily see the delegation of a channel (of type U) during the run of the protocol with the repository. Notice that the seller is not aware of the delegation; it need not change its type nor its code.

To complete our example, suppose that we would like function put to read the price from channel c, rather than receiving the price as a parameter. We need to pass c to the function; we may be tempted to write the following code.

```
sell c r =
  receive item from c in
  put item r c;
  checkIfSold item c r −− not typable
```

Channel c is a linear value; once passed (on a function or on another channel), it cannot be further used. In order for function checkIfSold to be able to use c, function put has to return the channel. The code becomes:

```
sell c r =
  receive item from c in
  checkIfSold item (put item c r) r −− ok
```

where the revised version of function put receives a channel at type **lin**?Price. **lin**⊕{*sold*:...} and returns the same channel, now at type **lin**⊕{*sold*:...}.

Functional languages with session types go back to [Vasconcelos et al. 2004, 2006] and [Neubauer and Thiemann 2004a]. The language here presented is inspired in [Gay and Vasconcelos 2008, Vasconcelos et al. 2006].

3. Syntax and Type Assignment

The syntax of our language is summarized in Figure 1. For simplicity in this section we do not consider choice nor the usual fix operator used in recursive functions. Variables describing channels

v ::=	Values:
x	variable
true | false	booleans
$()$	unit
fork	fork
$\lambda x\colon T.t$	abstraction
t ::=	Terms:
$t\,t$	application
if t then t else t	conditional
send t on $x; t$	output
receive x from x in t	input
channel $x, x\colon T$ in t	channel creation

Figure 1. The syntax of terms

q ::=	Qualifiers:
lin	linear
un	unrestricted
p ::=	Pretypes:
bool	booleans
unit	unit
$T \to T$	functions
end	termination
$?T.T$	receive
$!T.T$	send
T ::=	Types:
$q\,p$	qualified pretype
$\mu a.T$ | a	recursion
Γ ::=	Contexts:
$\emptyset$	empty context
$\Gamma, x\colon T$	assumption

Figure 2. The syntax of types and contexts

come in pairs, called co-variables, each attached to one channel endpoint. Interacting threads do not share variables for communication; instead, each thread owns one variable describing one endpoint. This mechanism allows a precise control of resources via a linear type system. For example, if x is a variable of an arbitrarily qualified type, a is a variable of an unrestricted type, and c a variable of a linear type, then the first two terms are valid whereas the last is not.

$$\mathsf{send\ true\ on}\ x; \mathsf{receive}\ y\ \mathsf{from}\ x$$
$$\mathsf{fork}\ (\mathsf{send\ true\ on}\ a); \mathsf{fork}\ (\mathsf{send\ true\ on}\ a); \mathsf{send\ false\ on}\ a$$
$$\mathsf{fork}\ (\mathsf{send\ true\ on}\ c); \mathsf{send\ true\ on}\ c$$

The syntax of types is summarized in Figure 2. For the lambda calculus, linearly qualified types describe resources that must be used exactly once; for channel operations such types describe variables that occur in exactly one thread. In either case, the unrestricted (or shared, un) qualifier indicates a value that can occur

$$\emptyset\cdot\emptyset=\emptyset \qquad \frac{\Gamma=\Gamma_1\cdot\Gamma_2}{\Gamma, x\colon \mathsf{un}\,p = (\Gamma_1, x\colon \mathsf{un}\,p)\cdot(\Gamma_2, x\colon \mathsf{un}\,p)}$$
$$\frac{\Gamma=\Gamma_1\cdot\Gamma_2}{\Gamma, x\colon \mathsf{lin}\,p = (\Gamma_1, x\colon \mathsf{lin}\,p)\cdot\Gamma_2} \qquad \frac{\Gamma=\Gamma_1\cdot\Gamma_2}{\Gamma, x\colon \mathsf{lin}\,p = \Gamma_1\cdot(\Gamma_2, x\colon \mathsf{lin}\,p)}$$

Figure 3. Context splitting

$$\frac{\mathsf{un}(\Gamma)}{\Gamma\vdash \mathsf{false}\colon q\,\mathsf{bool}} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma\vdash \mathsf{fork}\colon \mathsf{un}\,p \multimap \mathsf{un}\,\mathsf{unit}} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma, x\colon T\vdash x\colon T}$$
$$\frac{q(\Gamma) \qquad \Gamma, x\colon T\vdash t\colon U}{\Gamma\vdash \lambda x\colon T.t\colon q\,T\to U} \qquad \frac{\Gamma_1\vdash t_1\colon q\,T\to U \qquad \Gamma_2\vdash t_2\colon T}{\Gamma_1\cdot\Gamma_2\vdash t_1 t_2\colon U}$$
$$\frac{\Gamma_1\vdash t\colon q\,\mathsf{bool} \qquad \Gamma_2\vdash t_1\colon T \qquad \Gamma_2\vdash t_2\colon T}{\Gamma_1\cdot\Gamma_2\vdash \mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\colon T}$$
$$\frac{\Gamma_1\vdash x\colon q\,?T.U \qquad (\Gamma_2, y\colon T)\cdot x\colon U\vdash t\colon V}{\Gamma_1\cdot\Gamma_2\vdash \mathsf{receive}\ y\ \mathsf{from}\ x\ \mathsf{in}\ t\colon V}$$
$$\frac{\Gamma_1\vdash t_1\colon T \qquad \Gamma_2\vdash x\colon q\,!T.U \qquad \Gamma_3\cdot x\colon U\vdash t_2\colon V}{\Gamma_1\cdot\Gamma_2\cdot\Gamma_3\vdash \mathsf{send}\ t_1\ \mathsf{on}\ x; t_2\colon V}$$
$$\frac{\Gamma, x\colon T, y\colon \overline{T}\vdash t\colon U}{\Gamma\vdash \mathsf{channel}\ x, y\colon T\ \mathsf{in}\ t\colon U} \qquad \frac{\Gamma\vdash t\colon T \qquad T <\colon U}{\Gamma\vdash t\colon U}$$

Figure 4. Typing rules

multiple times in multiple threads. Type $\mathsf{lin}!(\mathsf{lin\ bool}).\mathsf{un\ end}$ describes a channel endpoint that can be used once to output a boolean value (that can be used once by the receiver) and then behaves as a shared channel on which no further operation is possible.

Duality is a notion central to session types. It is defined on session types only,

$$\overline{q\,?T.U} = q\,!T.\overline{U} \qquad \overline{q\,!T.U} = q\,?T.\overline{U} \qquad \overline{q\,\mathsf{end}} = q\,\mathsf{end}$$

to be complemented with recursion: $\overline{\mu a.T} = \mu a.\overline{T}$ and $\overline{a} = a$.

For example, if x_1 and x_2 are two co-variables, then the first two processes only are valid.

$$\mathsf{fork}\ (\mathsf{send\ true\ on}\ x_1); \mathsf{receive}\ z\ \mathsf{from}\ x_2$$
$$\mathsf{fork}\ (\mathsf{send\ true\ on}\ x_1; \mathsf{receive}\ w\ \mathsf{from}\ x_1);$$
$$\mathsf{receive}\ z\ \mathsf{from}\ x_2\ \mathsf{in\ send\ true\ on}\ x_2$$
$$\mathsf{fork}\ (\mathsf{send\ true\ on}\ x_1); \mathsf{send\ false\ on}\ x_2$$
$$\mathsf{fork}\ (\mathsf{send\ true\ on}\ x_1; \mathsf{send\ true\ on}\ x_1);$$
$$\mathsf{receive}\ z\ \mathsf{from}\ x_2\ \mathsf{in\ send\ true\ on}\ x_2$$

Context splitting (Figure 3) is a notion central to linear typing systems. When type checking processes with two sub-processes we pass the unrestricted part of the context to both processes, while splitting the linear part in two and passing a different part to each process. To ensure that linear objects are used exactly once, and that communication channels are used according to their protocols, our type system maintains important invariants.

- Linear variables describing channels occur in exactly one thread;
- Other variables are used exactly once;
- Co-variables have dual types;
- Unrestricted data structures may not contain linear data structures.

Let $\mathsf{lin} \sqsubseteq \mathsf{un}$. The predicate q is true of types $q'p$ when $q' \sqsubseteq q$, and is extended point-wisely to typing contexts. The type assignment system is in Figure 4. The typing rules make sure that linear values (including abstraction) are not discarded without being used, by checking that there is no linear variable in context. Function application crucially takes advantage of context splitting. Notice that there is no context splitting in the two branches of a conditional, since only one of them will be executed. The rule for channel creation captures the essence of co-variables: they must have dual types. The rule for channel reading splits the context into two parts: one to type check the channel x, the other to type check continuation t. If x is of type $q\,?T.U$ in receive y from x in t, then we use $y\colon T$ to type check t. Term receive y from x in t uses x at type $q\,?T.U$, whereas the continuation t may use x at the continuation type U.

[Wadler 1990] is one of the first references on linear types for functional programming. The type system here presented is inspired in [Gay and Vasconcelos 2008, Vasconcelos 2009, Walker 2005].

Operational Semantics and Main Results To talk about results we must define an operational semantics. We do not expound the details for the sake of brevity. The reader may nevertheless expect the runtime of our language to be composed of expressions running in parallel, created by the **fork** expression,

$$\mathsf{fork}\ t_1; t_2 \ \rightarrow\ t_1 \mid t_2$$

running under the scope of channel endpoints, created by the **channel** expression,

$$\mathsf{channel}\ x, y\ \mathsf{in}\ t \ \rightarrow\ (\nu xy)t$$

evolving by beta-reduction, and communicating via **send**-**receive** or **select**-**case**:

$$(\nu xy)(\mathsf{send}\ v\ \mathsf{on}\ x; t_1 \mid \mathsf{receive}\ z\ \mathsf{from}\ y\ \mathsf{in}\ t_2 \mid t_3) \ \rightarrow \\ (\nu xy)(t_1 \mid t_2[v/z] \mid t_3)$$

What can go wrong with programs? The obvious case: the value in the condition part of an if-term is neither true nor false. More interesting cases encompass different communication patterns on the same channel end, or channels ends with incompatible communication patterns.

$$(\nu x_1 x_2)(\mathsf{send\ true\ on}\ x_1 \mid \mathsf{receive}\ z\ \mathsf{from}\ x_1)$$
$$(\nu x_1 x_2)(\mathsf{send\ true\ on}\ x_1 \mid \mathsf{send\ true\ on}\ x_2)$$
$$(\nu x_1 x_2)(\mathsf{receive}\ z\ \mathsf{from}\ x_1 \mid \mathsf{receive}\ w\ \mathsf{from}\ x_2)$$

The main result of our language states that well-typed programs do not reduce to wrong runtime terms.

4. Buffered Semantics

The runtime system in the previous section is *unbuffered*. If x_1 and x_2 are the two endpoints of a same channel, then the sending party send v on x blocks waiting for a receiving partner receive z from y, and conversely. The idea of distinguishing the two channels endpoints is particularly appealing in a distributed setting where one usually finds *buffered* semantics. In fact, in distributed environments channels are usually uni-directional and buffered, for ease of implementation and performance. Our channels are conceptually bi-directional, but technical reasons compelled us to distinguish the two ends of a same channel. We now take this idea one step further and associate to each such channel endpoint a distinguished buffer to hold the messages in transit (that is, the values sent and the labels selected), thus making send a non-blocking operation.

Operational Semantics Once again, we are concise on the description of the unbuffered semantics. Threads now read from their own buffer and write on the other endpoint buffer. Buffers are entities running in parallel with threads. A buffer for endpoint x is a pair containing the address of the other endpoint y, and the actual elements in the buffer $\vec{b}$. We write all this as $x\colon (y, \vec{b})$. The channel creation expression now creates two empty buffers.

$$\mathsf{channel}\ x, y\ \mathsf{in}\ t \ \rightarrow\ (\nu xy)(x\colon (y, \varepsilon) \mid y\colon (x, \varepsilon) \mid t)$$

Writing becomes a non-blocking operation provided the two buffers are available. Notice that the send operation reads from its buffer x the address y of the other endpoint, on which it writes.

$$\mathsf{send}\ v\ \mathsf{on}\ x; t \mid x\colon (y, \vec{b}) \mid y\colon (x, \vec{b}') \ \rightarrow\ t \mid x\colon (y, \vec{b}) \mid y\colon (x, \vec{b}'v)$$

The receive operation reads from its own buffer provided that it is not empty.

$$\mathsf{receive}\ z\ \mathsf{from}\ x\ \mathsf{in}\ t \mid x\colon (y, v\vec{b}) \ \rightarrow\ t[v/z] \mid x\colon (y, \vec{b})$$

The proof of the main result is now more elaborate for it is no more the case that, in the parallel composition of two threads (t_1 and t_2) each using one endpoint (x_1 and x_2) of a same channel, the two threads see the types of the endpoints at dual types. Intuitively, the initial part of the type associated with x_1 is in its buffer, while the rest is read from thread t_1, and similarly for x_2 and t_2.

Buffered semantics in session types go back to [Neubauer and Thiemann 2004b]; they are used in [Fähndrich et al. 2006]. Reference [Gay and Vasconcelos 2008] contains the details for the semantics here presented.

Bounded Buffers Buffered semantics comes equipped with an interesting result: one can read from a session type the size of the buffer required to hold all data that will eventually be written on the buffer, that is we can statically prevent buffer overflows, under certain conditions. Two caveats are in order. We assume all data is of fixed size, so that for, e.g., variable length strings we place in the buffer a reference to an heap location where we allocate the actual string. Some session types may require an infinite bound, that is, the size of the buffer is taken from the set $\mathsf{Nat} \cup \{\infty\}$.

As an example, the size of the buffer required for a channel of type Seller at the very beginning of this paper is 2. Why? Because in the worst case, the buffer will contain a label *sold* and a value of type Price. Conversely, the size of the buffer for the dual type, AuctioneerWithSeller is 3, for the buffer may contain label *selling*, and two values of type Item and Price, before the reader thread starts reading.

For recursive types, we have to reason co-inductively. Interesting enough the bound for recursive type of the Auctioneer after the upgrade is still 3. How can we find out this value? Consider the infinite tree obtained by unfolding recursion endlessly. Count the maximal sequence of input operations (? and &); the counting terminates because the tree as only finitely many different subtrees. In our case the largest sequence of contiguous input operations in the whole type happens in the stretch **lin** &{*selling*: **lin**?Item.**lin**?Price.⊕{...}}.

Not all channels can be endowed with finite buffers. Take for example a channel governed by type **lin** !**unit**.μa?**lin bool**.a, which says "I am ready" before embarking on an infinite loop reading boolean values. A compiler may nevertheless suggest the introduction of a "sync" point after a certain number of reads, as in μa. **lin** !**unit**. **lin** ?**bool**. **lin** ?**bool**. **lin** ?**bool**.a, to require a buffer of finite length (of size 3, in this case).

How can we incorporate this technology in a compiler? Whenever the compiler finds an expression channel $x, y\colon T$ in t it computes the bound of T and generates the buffer for x accordingly, and similarly for $\overline{T}$ and y. If one or more of the bounds are not finite then the compiler may refuse to proceed or else use, say, linked lists in place of fixed-sized arrays for holding the buffers. Furthermore, information available during typechecking can be used to generate code to reduce the size of a buffer and ultimately to deallocate the buffer of a channel of type **end**. The formal proof that the session type of a channel can provide a static upper bound on the size of its buffer can be found in [Gay and Vasconcelos 2008], and was observed informally by [Fähndrich et al. 2006].

Anticipating Outputs Suppose that our auctioneer sells a single product. The code for the regist operation at the auctioneer can be written as follows.

```
regist :: lin ?Name. lin ! Item . lin ! Price .
  lin&{ buy : . . } ⊸ ?∗ Repository ⊸ unit
regist c r =
  receive name on c in
  send a_very_large_product_description on c;
  send 100 on c; ...
```

Since the auctioneer sells a single product it cannot employ aggressive marketing techniques based on the nature of the bidder. Then, knowing that product descriptions are rather lengthy, in order to maximize throughput, the auctioneer may want to send the product description before actually reading the name of the bidder.

```
regist c r =
  send a_very_large_product_description on c;
  send 100 on c;
  receive name on c in ...
```

Operationally, communication still proceeds smoothly: the auctioneer writes its two values on the bidder's buffer, yielding:

$$d\colon (c, \mathsf{a\_very\_large\_product\_description}\ 100)$$

while the bidder writes is name on the auctioneer's buffer:

$$c\colon (d, \text{"vasco"})$$

and then the auctioneer may receive a name on channel endpoint c, while the bidder receives an item and a price on channel endpoint d. Because there are distinct buffers for the two channel endpoints no deadlock arises.

What other commutations are in order? Suppose the auctioneer always terminates a session with a bidder by issuing a thank-you message. Further suppose that the message is the same in both branches.

```
regist c r = ...
  receive name on c in
  case c of {
    {sold ⇒ send a−long−thank−you on c; t1
     notSold ⇒ send a−long−thank−you on c; t2}
```

Once again, if the message is lengthy, the auctioneer may anticipate its sending by writing instead

```
regist c r = ...
  send a−long−thank−you on c;
  receive name on c in
  case c of {sold ⇒ t1 , notSold ⇒ t2}
```

The new program still runs smoothly; the problem is at type level: the auctioneer declares a type that does not match that of the bidder. The stretch of the new type related with the piece of code above is

```
!Item . ! Price . ! Message . ?Name.&{ buy : end , . . . }
```

(**lin** qualifiers omitted) whereas that of the bidder remains

```
!Name. ? Item . ? Price .⊕{ buy : ? Message . end , . . . }
```

The new type for the auctioneer is a refinement of the old type where we anticipate the output with respect to the input, or dually, postpone the output with respect to the input. We capture this requirement with an extension of the subtyping relation. If we denote by I an input operation (that is, ? or &) and by O an output operation (! or ⊕), then the mnemonic is

$$\mathsf{OI} <: \mathsf{IO}$$

The subsumption rule (in Figure 4) allows to type the new auctioneer at its old type

```
?Name . ! Item . ! Price .&{ buy : ! Message . end , . . . }
```

which is compatible with that for the bidder, as we have seen.

It should be easy to see that all other possible permutations either violate type safety or introduce potential deadlocks. The above subtyping relation was introduced by [Mostrous and Yoshida 2009], in conjunction with a language that also includes (pi calculus) process passing.

Acknowledgments

The incorporation of session types in functional programming, the buffered semantics, and the bounded buffers ideas where developed jointly with Simon Gay [Gay and Vasconcelos 2008]; the concrete formulation here presented is however new. The author was partially supported by the EU IST proactive initiative FET-Global Computing, project Sensoria, IST–2005–16004.

References

Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity OS. *SIGOPS Oper. Syst. Rev.*, 40(4):177–190, 2006.

Simon Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. Subsumes Technical Report 2007–251, University of Glasgow, 2008.

Simon J. Gay and Malcolm J. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *SIGPLAN Not.*, 43(1):273–284, 2008.

Dimitris Mostrous and Nobuko Yoshida. Session-based communication optimisation for higher-order mobile processes. In *Typed Lambda Calculi and Applications (TLCA'09)*, LNCS. Springer-Verlag, 2009. To appear.

Matthias Neubauer and Peter Thiemann. An implementation of session types. In *Proceedings of PADL'04*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004a.

Matthias Neubauer and Peter Thiemann. Session types for asynchronous communication. Unpublished, 2004b.

Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the behavior of objects and components using session types. *Fundamenta Informaticæ*, 73(4):583–598, 2006.

Vasco T. Vasconcelos. *9th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Web Services (SFM 2009)*, volume 5569 of *LNCS*, chapter Fundamentals of Session Types, pages 158–186. Springer-Verlag, 2009.

Vasco T. Vasconcelos, António Ravara, and Simon Gay. Session types for functional multithreading. In *CONCUR'04*, volume 3170 of *LNCS*, pages 497–511. Springer-Verlag, 2004.

Vasco T. Vasconcelos, Simon Gay, and António Ravara. Typechecking a multithreaded functional language with session types. *TCS*, 368(1–2): 64–87, 2006.

Philip Wadler. Linear types can change the world. In M. Broy and C. B. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581. North-Holland, 1990.

David Walker. *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems. MIT Press, 2005.