Calculating communication costs with Sessions Types and Sizes*

Juliana Franco, Sophia Drossopoulou, and Nobuko Yoshida

Imperial College of London, Department of Computing
London, United Kingdom
`{j.vicente-franco, s.drossopoulou, n.yoshida}@imperial.ac.uk`

Abstract

We present a small object-oriented language with communication primitives. The language allows the assignment of binary session types to communication channels in order to govern the interaction between different objects and to statically calculate communication costs. Class declarations are annotated with size information in order to determine the cost of sending and receiving objects. This paper describes our first steps in the creation of a session-based, object-oriented language for communication optimization purposes.

1998 ACM Subject Classification Languages and compilers

Keywords and phrases Session types, communication, object-oriented, multicore

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.50

1 Introduction

In the near future, parallel machines with thousands of cores are expected and programming languages that easily scale for such machines are needed. Furthermore, it is possible to have concurrent computations with a large number of messages exchanged and develop programs with such communications can be a hard task and liable to several errors. Session types [9, 16] are a well-established mechanism to describe message-passing computations that allows to ensure communication safety and the absence of race conditions and deadlocks. When assigned to communication channels they can be used to express the kind of messages exchanged among the different partners of a communication and their order as well as to statically verify if the communication proceeds as specified by session types. There are two variants of session types: binary (or two-party) session types to describe the communication between two parties and multiparty session types to govern the communication among multiple participants.

In this work, we are interested in the design of a new object-oriented programming language that uses session types to govern the communication between different objects and to statically calculate the respective communication costs. We start only with two-party sessions and then when we achieve our goals we will expand our approach to multiparty session types. We present a Java-like syntax with communication primitives where objects may communicate using channels with session types assigned.

The contributions of this paper can be summarized as our first steps in the design of a session-based, object-oriented language that allows to express the size of data structures and to statically calculate communication costs, using session types, for optimization purposes. We believe that in the future we will be able to use this information to apply optimizations

* This work was supported by Upscale Project.

$$\begin{array}{rrcl}
\textbf{Class declarations:} & D & ::= & \mathsf{class}\ C[\langle N^+\rangle]\{F^*; M^*\} \\
\textbf{Method declarations:} & M & ::= & (\mathsf{@set}\ sa\ (\mathsf{this}.f), v;)^* \quad \tau\ m((x:\rho)^*)\{e\} \\
\textbf{Field declarations:} & F & ::= & [\mathsf{@}sa\ v\ :]T\ f; \\
\textbf{Size annotations:} & sa & ::= & \mathsf{has} \quad | \quad \mathsf{has\ up\_to} \\
\textbf{Values:} & v & ::= & \text{constant} \quad | \quad N \quad | \quad v+v \quad | \quad v*v \\
\textbf{Types:} & T & ::= & \mathsf{boolean} \quad | \quad \mathsf{integer} \quad | \quad \mathsf{char} \quad | \quad \mathsf{string}[v] \quad | \quad C[\langle v^+\rangle] \\
\textbf{Return types:} & \tau & ::= & T \quad | \quad \mathsf{void} \\
\textbf{Parameter types:} & \rho & ::= & T \quad | \quad ST
\end{array}$$

Figure 1 Syntax of classes and types.

such as, to change the location of the objects—given that the location of objects has impact in the communication cost (if two objects are distant the communication cost is worst than if they are close) we can change the objects topology in order to obtain a better performance.

Structure of the document: The document is organised as follows. Section 2 describes our syntax of classes and types. Section 3 shows the syntax of session types followed by Section 4 with the syntax of session operations. The calculation of communication costs is described in Section 5. We present Section 6 and we finish this document with conclusion and future work in Section 7.

2 Syntax of classes and types

Our base language follows the syntax of Java; each program is a set of class declarations and each class declaration contains sets of fields and method declarations. However, we annotate our class declarations with information about the size of the objects (the number of elements of a data structure). Moreover we also annotate field and method declarations and the type parameters of our method declarations may be session types. The syntax of classes is shown in Figure 1, with the identifiers conventions: C for class identifiers, f for field identifiers, m for method identifiers, and N for size parameter identifiers.

Size parameters are annotated in class declarations and used in field declarations to describe (exactly or in maximum) how many elements a data structure has. For some class declaration C_3, consider, for instance, the class declarations:

$$\mathsf{class}\ C_1\langle N\rangle\ \{\quad \mathsf{@has}\ N:\ C_3\ c_3; \quad\} \qquad \mathsf{class}\ C_2\langle N\rangle\ \{\quad \mathsf{@has\ up\_to}\ N:\ C_3\ c_3; \quad\}$$

The type $C_1\langle 5\rangle$ represents an object that has exactly 5 elements of type C_3 while the type $C_2\langle 5\rangle$ represents an object that maximally has 5 elements of class C_3.

The method declaration is very similar to the Java method declaration where the returning type can be a type T or the void type and it has a set of parameters (each one with a type assigned). Note the possibility to have channel end-points as parameters (a variable assigned to a session type). The set annotation is used to change the size of an object. For instance if we add or remove elements of a data structure we need to indicate the new number of elements. A value v (a natural number or the result of an arithmetic expression) may denote either the number of elements of the object or the repetitions of a given communication

Session type declaration: ST ::= session $S[\langle N^* \rangle] = SB$
Session type body: SB ::= $!T.SB$ | $?T.SB$ | end |
$+\{l_i : SB_i\}_{i \in I}$ | $\&\{l_i : SB_i\}_{i \in I}$ | rec $a.SB$ | $a[v]$ | $SB_1; SB_2$
Session type invocation: SI ::= $S\langle v^* \rangle$ | dualof $S\langle v^* \rangle$

Figure 2 Syntax of session types.

pattern (recursive session types). We omit the syntax of expressions because it should be similar to that of Java expressions. However, note that our language supports also session operations. Session operations and session types are explained later. Our syntax allows five different types: the primitives boolean, integer, char and the string$[v]$ type (representing an array of chars of length v), and the object types, represented by a class identifier and the respective annotation for size.

In order to complement the explanation of our language we can use an example where we have a professor communicating with an administrative system. The professor should authenticate her credentials and after that she can provide an identifier of a course and ask to the system a list of students, or she can give a particular student identifier and obtain the respective student. Below, we show four class declarations useful to this example.

```
class Login {
  string[50] username;
  string[20] password;
}
```

```
class Student {
  string[150] name;
  integer age;
}
```

```
class StudentList<N> {
  @has up_to N;
  Node head;
}
```

```
class Node {
  Node next;
  Student student;
}
```

The class Login is a class with two fields only, username and password. Each instance of the class Student has a name (which in maximum has 150 characters) and an age. The class StudentList represents a list with a maximum of N students. In fact, we have that the list has N nodes, however each node has 1 student and therefore we may say that there are N students in the list. The class Node has a reference to the next node in the list and a reference to its student. If we want to add operations to change the list (for instance, add a new student) we can write the following method declaration in the class StudentList.

```
@set has(this.head), N + 1; void addStudent(Student n) { ... }
```

This means that after the invocation of this method, the list will have one more Student.

3 Syntax of session types

We intend to integrate this language with the theory of session types to describe the communication during one session [9, 16]. We consider bidirectional channels, where each channel is composed of two end-points; when two processes/objects communicate, each one possesses one of the channel ends and each channel end has a session type assigned. For I some index set and with the identifiers conventions $N \in \mathsf{Id}$, for size and repetition parameters, and $S \in \mathsf{Id}$, for session identifiers, Figure 2 shows the syntax of session types.

A session type declaration, ST assigns an identifier and size information to a session type, the session type body. It binds the size variables, denoted by N, that may be used in the session type body. We may use the session type $!T.SB$ to describe a channel end that should send a value of type T and then proceed as SB, or the type $?T.SB$ to receive a value of type T and continue behaving as described in SB. To describe an end-point that selects an option from a fixed range of options, we use the type $+\{l_i : SB_i\}_{i\in I}$. The session type to describe an end-point that offers this menu of options is $\&\{l_i : SB_i\}_{i\in I}$. The type end should be assigned to end-points where no further interaction may occur. The syntax presented is very similar to other languages with support for binary session types, such as [6, 16]. However we introduce two constructs that we believe to be different from the other approaches: the limitation of recursion, through the type variable annotation (usually the type/recursion variable is not annotated with information about repetition), and the sequence of session types. Using recursive types of the form rec $a.SB$ and the annotated type variable $a[v]$, it is also possible to write a session type that maximally allows the repetition of a given behaviour v times. For instance, the type rec $a.$!integer$.a[3]$ when assigned to an end-point, means that it can send 3 integers. We can consider that rec $a.$!integer$.a[3]$ is equivalent to !integer.!integer.!integer.end. The type rec $a + \{l_1 :!T_1.a[3], l_2 :!T_2.a[2]\}$ governs a channel end that selects between l_1 and l_2 a total of 5 times, sends 3 values of type T_1 and 2 values of type T_2. An end-point assigned to $SB_1; SB_2$ will first behave as defined by SB_1 and then as SB_2. We may say that the session type $SB_1; SB_2$ is an abbreviation for SB_1 where we replace end by SB_2, which is $[SB_2/\text{end}]SB_1$ [1]. In order to use a declared session type our syntax provides a constructor for the session type invocation composed of the name assigned to the session type, S, and the value or the result of the arithmetic expression, v, to be used in the session type SB. It also provides for a dual type invocation, that give the dual session type—it is important to ensure that when one of the channel ends sends a value of type T, the other must be ready to receive a value of type T and when one of the channel ends selects an option from a menu, the other must offer a menu that contains this option. This means that the end-points of a channel must have *dual* behaviours [2].

Our first session type declaration is the one to govern the administrative system communication side:

```
session Admin<u1, u2> =
  rec a. ?Login.
  +{authentication_denied: !string[u1].
    &{try_again: a[2], close: end},
    authentication_accepted:
      rec b. &{all_students: ?integer: !StudentList<u2>.end,
                   single_student: ?integer. !Student.b[u2]}
  }
```

First the system receives an object Login with the credentials of the professor, then it should select between authentication_denied and authentication_accepted. If it selects the first option, then it must send the string with the failure reason to the professor and offer a choice between try_again and close. The professor may only try to login 3 times; after that, the communication channel is closed. When the system selects the option authentication_accepted, it offers two options to the professor: all_students, where the system expects to receive an integer with course identifier to respond with a StudentList, or single_student where the system

[1] We still need to verify if the session type is well-formed. For instance the type rec a. end; a is allowed by our syntax however it is not contractive. Recursive types are required to be contractive, that is, it cannot be of the form or has any sub-type of the form rec a1, ... rec an. a1

[2] The duality function can be found in https://wp.doc.ic.ac.uk/jvicent1/files

Session operations: $O ::= x\ y = \mathsf{new}\ (SI_1@L_1, SI_2@L_2) \mid z.\mathsf{send}(e) \mid z.\mathsf{receive}() \mid z.\mathsf{select}(l) \mid \mathsf{switch}(z)\{\mathsf{case}\ l_i : e_i\}_{i\in I} \mid \mathsf{startSession}\{e.m(e^*), e.m(e^*)\}$

where x, y, z are channel names, l denotes labels and L location identifiers.

Figure 3 Syntax of session operations.

expects an **integer** with the student identifier and replies with a Student. When the option single_student is selected, the menu will be available again, after the sending of the student, in a maximum of u2 times.

4 Session operations

In this section we present the syntax of session operations shown in Figure 3. We start with the channel creation of the form $x\ y = \mathsf{new}\ (SI_1@L_1, SI_2@L_2)$ to create a new channel defined by two end-points x and y. The end-point x is assigned to the resulting session type of the type invocation SI_1, while y is assigned to SI_2, which should be dual of SI_1. The channel end x will be used by an object in some location L_1 while y will be used from location L_2. An end-point can be used to send the result of an expression with $z.\mathsf{send}(e)$, to receive a value, $z.\mathsf{receive}()$, to select an option labelled as l, from a fixed range of options, with $z.\mathsf{select}(l)$ and to offer a menu of options, using $\mathsf{switch}(z)\{\mathsf{case}\ l_i : e_i\}_{i\in I}$, for I some index set. Our syntax support also a primitive to start the session between two participants of the communication by invoking two methods—we assume that the communication happens between those methods. Note that the channel ends must be passed as parameters of the method, so that the communication may proceed. For instance in order to start the communication in our example we can use the following code [3].

```
// declaration of maxStudents and failureReason variables
Administrator a = new Administrator(maxStudents);
Professor p = new Professor(maxStudents);
x y = new (Admin<failureReason.length, maxStudents> @ L1,
       dualof Admin<failureReason.length, maxStudents> @ L2);
startSession{a.communicate(x), p.communicate(y)}
```

5 Calculation of communication cost

In this section we present the methodology to calculate the communication cost of a given channel using its session type. We assume that we have: a function sending_cost that expects two abstract locations L_1 and L_2 and returns the cost of sending 1 word from the concrete location of L_1 to the concrete location of L_2; a function receiving_cost that also expects two abstract locations L_1 and L_2 and returns the cost of receiving 1 word from the concrete location of L_2 to the concrete location of L_1. We also consider that a label of a choice type fits in 1 byte (for instance, the selection of an option is equivalent to the output of a label and we can use the function sending_cost). Furthermore, we assume that we have a function sizeOf that expects a type T and returns the number of bytes of a value or object of this type[4].

[3] What we envision as the full code of our example can be found in `https://wp.doc.ic.ac.uk/jvicent1/files`.

[4] We are currently working in this function and we do not include it in the document due space limitation.

In this section we show how to calculate the communication cost of a channel with the Admin session type assigned. The first step is to normalise the session type, that is, for a session type SB_1 we need to obtain a different session type SB_2 that has the same communication cost as SB_1 and respects the following properties:

- In a session type of the form rec $a.SB$ the type variable a appears free in SB only once;
- In a choice type (selection or branching) if there is an option SB_i which has a free type variable, then the menu of options is composed only by SB_i.

This step is important when we have a branching type with recursion. A naive approach would be to calculate the cost of all options and choose the worst case (most expensive communication). However if we analyse, for instance, our session type Admin we can conclude that it would be an optimistic cost given that the authentication can be denied twice before to be accepted. After the normalisation, we are able to calculate the communication cost. The intuition behind the communication cost function is very simple. For the input/output types we only need to calculate the size of the parameter and sum to the cost of the continuation type. And given that the type is normalised, the cost of a branch type is the maximum cost of all branches. Both definitions of the functions normalisation and communication cost can be found in `https://wp.doc.ic.ac.uk/jvicent1/files`.

The normalisation

Informally, we may say that in the worst case, an end-point governed by the Admin session type receives three objects of type Login, meaning that the professor failed the first two authentication attempts. For each failed authentication the server sends the failure reason and offers the option to try to authenticate again. Then it selects the label authentication_accepted. After that, the end-point should be used to send b objects of type Student after receive b values of type **integer** (when the option single_student is selected by the professor). Finally the channel end should be used to receive an **integer** and send the complete list of students, called StudentList. If we normalise the Admin session type, we obtain:

$$\begin{aligned}&\mathsf{rec}\ a.?\mathsf{Login}.+\{\mathsf{authentication\_denied}: !\mathsf{string}[u_1].\&\{\mathsf{try\_again}: a[2]\}\};\\&?\mathsf{Login}.+\{\mathsf{authentication\_denied}: \&\{\mathsf{close}: \mathsf{end}\},\\&\qquad\qquad \mathsf{authentication\_accepted}: \mathsf{rec}\ b.\&\{\mathsf{single\_student}: ?\mathsf{integer}.!\mathsf{Student}.b[u_2]\};\\&\qquad\qquad\quad \&\{\mathsf{all\_students}: ?\mathsf{integer}.!\mathsf{StudentList}.\mathsf{end}\}\}\end{aligned}$$

The result is a session type that governs an end-point that performs the operations described above. As we can see the above session type includes two subtypes that terminate the communication in the channel end—the types $\&\{\mathsf{all\_students}: ?\mathsf{integer}.!\mathsf{StudentList}.\mathsf{end}\}$ and $!\mathsf{string}[u_1].\&\{\mathsf{close}: \mathsf{end}\}$— and we only want to consider the worst case, that is, the first subtype. Therefore this should be taken into account in the communication cost function.

The communication cost calculation

Now that we have our normalised type we can calculate the communication cost of the first subtype (the calculation for the rest of the types is similar).

$$\begin{aligned}&\mathsf{costc}(\mathsf{rec}\ a.?\mathsf{Login}.+\{\mathsf{authentication\_denied}: !\mathsf{string}[u_1].\&\{\mathsf{try\_again}: a[2]\}\}; L_1, L_2) =\\&2 * g(L_1, L_2) * (\mathsf{sizeOf}(\mathsf{Login}) + 1) + 2 * f(L_1, L_2) * (1 + \mathsf{sizeOf}(\mathsf{string}[u_1]))\end{aligned}$$

6 Related work

There are already several papers about the incorporation of session types in object-oriented languages. Dezani et al. proposed the first integration of a class-based object-oriented programming language with session types [5]. This work was followed by the programming language Moose [4], a multi-threaded object-oriented core language augmented with session types, and by an extension of Moose [3], with bounded session types for object oriented languages. Hu et al. introduced an extension of Java, called SJ [10], a session-based distributed programming language. Following a different approach, Gay et al. [7] presented a distributed object-oriented programming language where it is possible to specify the possible sequences of method calls attaching session types to class definitions. Based on this work, Bica [1] is an extension to Java5 that allows the verification of Java code against session types, and Mool [2] is a minimal object-oriented language with support for concurrency. There is also an integration of multiparty session types with a Java extension, inspired on SJ, presented by Sivaramakrishnan et al. [14, 15].

In the scope of high-performance computing, Ng et al. created Session C, a programming framework for message-passing parallel programming that combines multiparty session types with the C programming language and its respective runtime libraries [12]. Based on Session C, Ng et al. presented a programming framework for safe and reconfigurable parallel designs and Pable a parametrised protocol description language [13, 11]. Honda et al. also proposed a methodology for MPI programs based on session types [8]. As in Session C, the idea is to check MPI programs against local protocols (obtained from multiparty session types), in polynomial time, to ensure type safety, communication safety and deadlock freedom.

At the best of our knowledge, all of these languages, and other which we do not mention, do not use session types to calculate communication costs. Some of them consider some optimizations using session types, such as [14, 15], however none of them do it as we intend to do in the future: use these costs to optimise the communication.

7 Conclusion and further work

Conclusion. We present our first steps towards the creation of a new object oriented programming language with primitives for communication over bidirectional channels. Session types will be assigned to Channels allowing to statically verify if the exchanged messages respect the expected order and type and in addition to calculate the communication cost of an end-point when this is located at some location. We believe that this communication cost information can be used to improve the communication in a concurrent computation, even that it only gives us the communication cost for the worst case.

Further Work. This is work is still in the beginning and there are yet several questions that we have to answer and several points to improve, such as, about the correctness of our functions and how we can check that the number of repetitions of session types and the number of elements of a data structure are correct. Moreover we want to integrate the sizeOf function and find a better solution for the annotated type variable in the duality function. Furthermore, we started our work with binary session types however our main goal is to have a object-oriented programming language with multiparty session types. Given that multiparty session types can be projected into local session types, we decided to start our work with binary session types and after we achieve our goals, we can extend our ideas to multiparty session types. In addition, we intend to study what optimizations we can do using this cost information.

References

1 Alexandre Caldeira and Vasco T. Vasconcelos. Bica. `http://gloss.di.fc.ul.pt/bica`.

2 Joana Campos and Vasco T. Vasconcelos. Channels as objects in concurrent object-oriented programming. In *PLACES 2010*, volume 69 of *EPTCS*, pages 12–28, 2011.

3 Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou, and Nobuko Yoshida. Bounded session types for object oriented languages. In *Formal Methods for Components and Objects*, volume 4709 of *LNCS*, pages 207–245. Springer Berlin Heidelberg, 2007.

4 Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *European Conference on Object-Oriented Programming*, 2006.

5 Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alexander Ahern, and Sophia Drossopoulou. A distributed object-oriented language with session types. In *Trustworthy Global Computing*, pages 299–318, 2005.

6 Juliana Franco and VascoThudichum Vasconcelos. A concurrent programming language with refined session types. In *Software Engineering and Formal Methods*, LNCS, pages 15–28. Springer International Publishing, 2014.

7 Simon Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *Principles of Programming Languages*, pages 299–312. ACM Press, 2010.

8 Kohei Honda, EduardoR.B. Marques, Francisco Martins, Nicholas Ng, Vasco T. Vasconcelos, and Nobuko Yoshida. Verification of mpi programs using session types. In *Recent Advances in the Message Passing Interface*, LNCS, pages 291–293. Springer Berlin Heidelberg, 2012.

9 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *European Symposym on Programming*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

10 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *European Conference on Object-Oriented Programming*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.

11 Nicholas Ng and Nobuko Yoshida. Pabble: Parameterised scribble for parallel programming. In *PDP 2014*, pages 707–714. IEEE Computer Society, 2014.

12 Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty session C: safe parallel programming with message optimisation. In *Conference on Objects, Models, Components, Patterns*, volume 7304 of *LNCS*, pages 202–218. Springer, 2012.

13 Nicholas Ng, Nobuko Yoshida, Xinyu Niu, and Kuen Hung Tsoi. Session types: towards safe and fast reconfigurable programming. *SIGARCH Computer Architecture News*, 40(5):22–27, 2012.

14 K. C. Sivaramakrishnan, Mohammad Qudeisat, Lukasz Ziarek, Karthik Nagaraj, and Patrick Eugster. Efficient sessions. *Sci. Comput. Program.*, pages 147–167, 2013.

15 K.C. Sivaramakrishnan, Karthik Nagaraj, Lukasz Ziarek, and Patrick Eugster. Efficient session type guided distributed interaction. In *Coordination Models and Languages*, LNCS, pages 152–167. Springer Berlin Heidelberg, 2010.

16 Vasco T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, 2012.