Modular Session Types for Distributed Object-Oriented Programming

Simon J. Gay
Department of Computing Science
University of Glasgow, UK
simon@dcs.gla.ac.uk

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

António Ravara *
CITI and Department of Informatics,
FCT, New University of Lisbon, Portugal
aravara@fct.unl.pt

Nils Gesbert
Department of Computing Science
University of Glasgow, UK
nils@dcs.gla.ac.uk

Alexandre Z. Caldeira
Department of Informatics
University of Lisbon, Portugal
zua@di.fc.ul.pt

Abstract

Session types allow communication protocols to be specified type-theoretically so that protocol implementations can be verified by static type-checking. We extend previous work on session types for distributed object-oriented languages in three ways. (1) We attach a session type to a class definition, to specify the possible sequences of method calls. (2) We allow a session type (protocol) implementation to be *modularized*, i.e. partitioned into separately-callable methods. (3) We treat session-typed communication channels as objects, integrating their session types with the session types of classes. The result is an elegant unification of communication channels and their session types, distributed object-oriented programming, and a form of typestates supporting non-uniform objects, i.e. objects that dynamically change the set of available methods. We define syntax, operational semantics, a sound type system, and a correct and complete type checking algorithm for a small distributed class-based object-oriented language. Static typing guarantees that both sequences of messages on channels, and sequences of method calls on objects, conform to type-theoretic specifications, thus ensuring type-safety. The language includes expected features of session types, such as delegation, and expected features of object-oriented programming, such as encapsulation of local state. We also describe a prototype implementation as an extension of Java.

* Work developed while the author was at SQIG, Instituto de Telecomunicações, and Department of Mathematics, IST, Technical University of Lisbon.

POPL'10, January 17–23, 2010, Madrid, Spain.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Classes and objects; D.3.2 [*Language Classifications*]: Object-oriented languages; D.3.1 [*Formal Definitions and Theory*]; F.3.2 [*Semantics of Programming Languages*]: Operational semantics; F.3.3 [*Studies of Program Constructs*]: Type structure; D.1.5 [*Object-oriented Programming*]

General Terms Languages, Theory, Verification

Keywords Session types, object-oriented calculus, non-uniform method availability, typestates

1. Introduction

Session types [29, 49] allow communication protocols to be specified type-theoretically so that protocol implementations can be verified by static type-checking. A session type describes a communication channel, and defines the permitted sequences and types of messages. For example, the session type $S = !\,[\mathsf{Int}]\,.\,?\,[\mathsf{Bool}]\,.\mathsf{end}$ specifies that an integer must be sent and then a boolean must be received, and there is no further communication. More generally, branching and repetition can be specified.

Session types were originally formulated for languages closely based on process calculus. Since then, the idea has been applied to functional languages [25, 26, 39, 44, 51], component-based object systems [50], object-oriented languages [10, 17–19, 31, 38], operating system services [22] and more general service-oriented systems [11]. Session types have also been generalized from two-party to multi-party systems [8, 30], although in the present paper we will only consider the two-party case.

We propose a new approach to combining session-typed communication channels and distributed object-oriented programming, which extends previous work and allows increased programming flexibility. The key idea is to allow a channel (e.g., of type S above) to be stored in a field of an object, and for separate methods to implement parts of the session. For example, method m can send the integer and method n can receive the boolean. Because the session type of the channel requires that the send occurs first, it follows that m must be called before n. We need therefore to work with *non-uniform objects*, in which the availability of methods depends on the state of the object. In order to develop a static type system for object-oriented programming with session-typed channels, we

use a form of typestates (a type safe state abstraction, according to [14, 21]) that we have previously introduced under the name of *dynamic interfaces* [52]. In this type system, the availability of a class's methods (i.e., the possible sequences of method calls) is specified in a style that itself resembles a form of session type, giving a pleasing commonality of notation at both the channel and class levels.

The result of this combination of ideas is a language that allows a very natural integration of programming with session-based channels and with non-uniform objects. In particular, the implementation of a session can be *modularized* by dividing it into separate methods that can be called in turn. In contrast, previous work on object-oriented session types, although allowing a session to be delegated to another method, does not allow separation into separately-callable blocks of code. Thus, our approach leads to a more flexible programming style than the other approaches mentioned above. Our formal language provides channels as disciplined streams, because session types are a high-level abstraction for structuring communication, and integrates this communication-based construct, without further restrictions, with the high-level object-oriented abstractions for structuring computation.

We have formalized a core *distributed class-based object-oriented* language with a static type system that combines session-typed channels and a form of typestates. We have proved that static typing guarantees two runtime safety properties: first, that the sequence of method calls on every non-uniform object follows the specification of its class's session type; second, as a consequence (because channel operations are implemented as method calls) that the sequence of messages on every channel follows the specification of its session type. We have also formalized a typechecking algorithm and proved its correctness, and implemented a prototype language as an extension of Java.

There is a substantial literature of related work, which we discuss in detail in Section 8. Very briefly, the contributions of our paper are the following.

- In contrast to other work on session types for object-oriented languages, we do not require a channel to be created and completely used (or delegated) within a single method. Several methods can operate on the same channel, thus allowing effective encapsulation of channels in objects, while retaining the usual object-oriented development practice. This is made possible by our integration of channels and non-uniform objects. This contribution was the main motivation for our work.
- In contrast to other typestate systems, we use a global specification of method availability, inspired by session types, made part of a class definition. While typestates are intensional, directly related to the object's state, we define these states with types, making use of standard type-theoretic tools to ensure client-conformance.

The remainder of the paper is structured as follows. In Section 2 we illustrate our system by introducing an example. In Section 3 we formalize a core sequential language; in Section 4 we extend it to a distributed language and in Section 5 we state the key properties of the type system. In Section 6 we present a typechecking algorithm and state results about its correctness. Section 7 describes our prototype implementation. Section 8 contains a more extensive discussion of related work; Section 9 outlines future work and concludes.

2. Example: Buyer/Seller

To illustrate the features of the formal language and of the type system, we incrementally present an example.

The Buyer/Seller Protocol. Our example is based on an e-commerce protocol between a buyer and a seller. The two parties interact on a point-to-point communication channel, each owning one endpoint. The buyer's protocol is specified by the session type

$$B = \oplus\{\mathsf{reqQuote} : !\,[\mathsf{Product}]\,.\,?\,[\mathsf{Price}]\,.\,?\,[\mathsf{Quote}]\,.\,B,\ \mathsf{accQuote} : !\,[\mathsf{Quote}]\,.\,!\,[\mathsf{Payment}]\,.\,B,\ \mathsf{quit} : \mathsf{end}\}$$

The buyer has a choice between reqQuote, accQuote and quit. If she chooses reqQuote she must send information about the desired product, and then receive the price and a reference number for the quote. After this, the session type is again B, and the buyer can choose another option. When she wants to buy a product, the buyer can select accQuote and then send a quote reference followed by payment information. It is therefore only possible to buy an item after a quote has been obtained, although this is not specified explicitly as part of the type. Selecting quit at any time, instead of accQuote or reqQuote, terminates the protocol.

The seller's protocol is specified by the dual session type

$$S = \&\{\mathsf{reqQuote} : ?\,[\mathsf{Product}]\,.\,!\,[\mathsf{Price}]\,.\,!\,[\mathsf{Quote}]\,.\,S,\ \mathsf{accQuote} : ?\,[\mathsf{Quote}]\,.\,?\,[\mathsf{Payment}]\,.\,S,\ \mathsf{quit} : \mathsf{end}\}$$

in which send (!) and receive (?) are exchanged, and the choice constructor ($\oplus$) is replaced by the branch constructor ($\&$). This means that the seller must be ready to respond to all of the three choices that the buyer can make. We express the relationship of duality between S and B by $S = \overline{B}$, or equivalently $B = \overline{S}$ as the duality operation is self-inverse.

The goal of a static type system with session types is to be able to verify, by type-checking, that the implementations of the buyer and the seller follow the specified protocol.

An API for the Buyer. We work within a model of distributed computing in which there are a number of sites, each executing an independent program. Services are accessed via typed access points n, analogous to URLs. The type $\langle S \rangle$ describes an access point for a service whose type (protocol) is S. A point-to-point bidirectional communication channel is created by the interaction of operations $n.\mathsf{request}()$ and $n.\mathsf{accept}()$ executed at separate sites. If n has type $\langle S \rangle$ then $n.\mathsf{accept}()$ yields one endpoint of the channel, with type S, and $n.\mathsf{request}()$ yields the other endpoint, with type $\overline{S}$. Given a channel c, synchronous communication occurs through the interaction of $c.\mathsf{send}$ and $c.\mathsf{receive}$ operations. An access point such as n must be announced at the top level at every site that uses it, and all such occurrences must share the same type. For simplicity, we do not allow access points to be created dynamically.

It is very natural to implement an API for buyers, by defining the class BuyerAPI in Figure 1. A program that needs to act as a buyer — for example, driven by a GUI application — can create an instance of class BuyerAPI and call methods on it, instead of working directly with the primitive operations request, send and receive. This approach has several advantages. The class abstracts from the details of the protocol, for example the exact order of messages. It also hides the Quote information by storing it in a data structure indexed by Product. As we will see, it can form the basis for an inheritance hierarchy of classes that offer more services, although we do not formalize inheritance in the present paper.

The code in Figure 1 consists of four declarations. Lines 1 and 3 define enumerated types Option and Result. Lines 5–8 define the session type S of the channel protocol; we have chosen the seller's viewpoint. Lines 10–43 define the class BuyerAPI. Because the field c will store a channel of type $\overline{S}$ the class BuyerAPI is non-uniform. We specify the availability of methods by the *session declaration* in lines 11–17. We refer to this as a *class session type* to distinguish it from *channel session types* such as S.

```
1 enum Option {reqQuote, accQuote, quit}
2
3 enum Result {ok, error}
4
5 typedef S =
6 &{Option.reqQuote:?[Product].![Price].![Quote].S
7   Option.accQuote:?[Quote].?[Payment].S,
8   Option.quit:end}
9
10 class BuyerAPI {
11   session Init
12   where Init = { init: Shop }
13         Shop = { price: Shop,
14                  buy: ⟨Result.ok: Pay,
15                        Result.error: Shop⟩,
16                  stop: end }
17         Pay  = { pay: Shop }
18
19   c; qs; // fields, initially null:Null
20
21   void init(⟨S⟩ u) {
22     c = u.request();
23     qs = new QuoteStore(); qs.init();
24   }
25   Price price(Product p) {
26     c.send(Option.reqQuote);
27     c.send(p);
28     Price pr = c.receive();
29     Quote q = c.receive();
30     qs.add(p,q);
31     return pr;
32   }
33   Result buy(Product p) {
34     Quote q = qs.get(p);
35     if (q == null)
36       return Result.error;
37     else {
38       c.send(Option.accQuote);
39       c.send(q);
40       return Result.ok;
41   }
42   void pay(Payment p) { c.send(p); }
43   void stop() { c.send(Option.quit); }
44 }
```

Figure 1. An API for the buyer.

An object of class BuyerAPI has abstract states Init, Shop, Pay and **end**. The type constructor {...} specifies the available methods and the abstract states that result when they are called. The type of an instance of class BuyerAPI is BuyerAPI[Init], BuyerAPI[Shop], BuyerAPI[Pay] or BuyerAPI[**end**]. The state **end** is a standard abbreviation for a state without available methods. Our approach to specifying method availability is similar to other systems of typestates for object-oriented languages [16, 21], except that we collect the whole specification into the class session type instead of annotating the method definitions with pre- and post-conditions. In our system, annotations are required only for recursive methods; we discuss this point at the end of Section 3.

Another distinctive feature of our language is that the abstract state after a method call may depend on the return value of the method, if it is of an enumerated type. This is illustrated on lines 14–15, where ⟨...⟩ is a variant type indexed by values of type Result. A caller of buy must **switch** on the result in order to discover the state and hence the available methods; this is enforced by the type system. In this example, method buy returns error if a price has not yet been obtained for the specified product. It is not possible to use the session type to specify that price must be called before buy, because the product description is arbitrary data.

```
1 // sellerURL is of type ⟨S⟩
2 // with S defined in Figure 1
3 b = new BuyerAPI();
4 b.init(sellerURL);
5 // Wait until price is right
6 while(b.price(myProduct) > 100) {};
7
8 switch(b.buy(myProduct)) {
9   case error:
10    print("Something went wrong"); break;
11  case ok: b.pay(myPayment); break;
12 }
13 b.stop();
```

Figure 2. A buyer — code fragment.

Method init has a parameter u whose type ⟨S⟩ indicates that it represents an access point for a service of type S. The use of the notation ⟨...⟩ for both variant types and access point types should not cause confusion as they occur in different contexts. When init is called, the actual parameter will be a specific access point that has been announced as such with type ⟨S⟩. The method uses u.**request**() to create a channel. It also creates and initializes a QuoteStore object, which we assume allows construction of a mapping between products and quotes, in a similar way to a Java HashMap. Although our language does not include constructors as a special category, the session type of BuyerAPI specifies that init must be called first, so we can regard it as the initialization part of a constructor. Likewise, we assume that after the call to QuoteStore.init(), the object stored in qs is in some state Q in which all other QuoteStore methods are available.

Methods price, buy and pay implement parts of the buyer's protocol. Defining these operations as separate methods is the key innovation of our approach. This is what we mean by *modularity* of sessions. Other work on object-oriented session types does not allow this.

There is a consistency requirement between the channel session type S, the class session type Init, and the definitions of the methods. Consistency is checked by the type system described in Section 3 and by the type-checking algorithm described in Section 6. If we take a sequence of method calls allowed by the class session type, and look at the channel operations in the methods to obtain a sequence of channel operations, then this must be allowed by the channel session type S.

In order to support *modular type-checking* we require only the session type of a class, not the types of its fields. For example, in order to type-check classes that are clients of BuyerAPI, we do not need to know that BuyerAPI contains a channel with a session type; the class session type of BuyerAPI contains all of the necessary information about the allowed sequences of method calls. It is therefore possible to associate session types with library classes containing native methods whose source code cannot be available.

Type safety with non-uniform objects requires tight control of aliasing. When the type of an object changes, by calling a method on it or by analysing an enumeration constant returned from a method call, there must be a unique reference to it. Since we are mainly interested in exploring the key idea of modularizing session implementations by integrating session-typed channels and non-uniform objects, we have adopted a simple approach to ownership control: a linear type system. We expect to be able to ease this restrictive system by using an off-the-shelf solution to aliasing control, such as one of the approaches discussed in Section 8.

```
1  access ⟨S⟩ sellerURL; // S defined in Figure 1
2
3  class Seller {
4    session { main: end }
5
6    void main() {
7      while (true)
8        spawn SellerThread.run(sellerURL.accept());
9  } }
10
11 class SellerThread {
12   session { run: end }
13
14   void run(S x) {
15     switch (x.receive()) {
16       case reqQuote: reqQuote(x);
17       case accQuote: accQuote(x);
18       case quit: break
19   } }
20   void reqQuote(?[Product].![Price].![Quote].S x)
21   { Product p = x.receive();
22     x.send(...); // Calculate price
23     x.send(...); // Quote reference
24     run(x)
25   }
26   void accQuote(?[Quote].?[Payment].S x) {
27     Quote q = x.receive();
28     Price py = x.receive();
29     ... // Process payment
30     run(x)
31 } }
```

Figure 3. A multi-threaded seller, featuring two "private" methods and mutual recursion.

Interacting with the Buyer API. Figure 2 shows a code fragment that creates and uses an instance of class BuyerAPI. We assume that the typed access point sellerURL corresponds to the published access point of a particular seller that observes protocol S. In the remaining code, myProduct and myPayment represent, respectively, the name of a particular product and the details of a method of payment.

Figure 3 contains a schematic definition of a seller. The seller should run independently at some location, so class Seller defines a main method and the class session type specifies that main is called once. The statement **spawn** SellerThread.run(sellerURL.**accept**()) is repeatedly executed by the body of main. The semantics of this statement is as follows. The expression sellerURL.**accept**() creates a channel by interacting with a matching sellerURL.**request**() at another site (it blocks until there is a matching sellerURL.**request**()), and evaluates to the endpoint c^+ so that we have the statement **spawn** SellerThread.run(c^+). According to our very simple concurrency mechanism, this creates a new heap containing an instance of SellerThread on which run(c^+) is called, forming an independently executing expression. As will be explained in Section 4, for simplicity our formal language has no concept of separate threads within a single location; we therefore have to think of spawn as creating a new location. The run method uses mutual recursion to implement a loop that repeatedly receives and processes requests, until quit is selected. The effect is that main accepts a connection and immediately delegates the new channel endpoint to a new thread. It would also be possible for main to execute part of the protocol before delegating the channel.

Notice that the methods reqQuote and accQuote of the class SellerThread are not in the session type. Although our language does not include method qualifiers, the two methods can be regarded as *private* since the type system ensures that they cannot be called by any client of class SellerThread. Notice also that the three mutually recursive methods in SellerThread each implement a part of session type S.

We assume an external mechanism for checking that access points are announced consistently at all sites. This could be a trusted central repository of typed services, or Hu's [31] system of run-time type-checks when **request** and **accept** interact.

The code in this example differs from the formal language defined in Sections 3 and 4 in two ways. First, the methods run, reqQuote and accQuote, being mutually recursive, should be annotated with their effect on the types of the fields of SellerThread. Because SellerThread has no fields, the annotations would be vacuous and so we have omitted them. Second, the parameter types of reqQuote and accQuote should have the form Chan[S'], where Chan is a special class name representing channels and S' is a class session type derived from the channel session type S. We have used the channel session type in the example code in order to make it more readable.

```
1  enum NewResult restricts Result {ok}
2
3  class NewBuyerAPI extends BuyerAPI {
4    @Override
5    NewResult buy(Product p) {
6      if (!qs.contains(p)) price(p);
7      c.send(Option.acceptQuote);
8      c.send(qs.get(p));
9      return NewResult.ok;
10 } }
```

Figure 4. An extended buyer API — features a self call to a "public" method.

Inheritance and Subtyping. For simplicity, the formal language defined in the present paper does not include inheritance; however, it does include a subtyping relation on session types, which provides a foundation for inheritance and is also used in other ways. It is straightforward to add inheritance, along the following lines. A class C inherits from (extends) a class D in the usual way: C may define additional fields and methods, and may override methods of D. By considering the standard principle of safe substitutability, namely that an object of class C should be safely usable wherever an object of class D is expected, we can work out the appropriate subtyping relationship between the session types of C and D. In a given state, C must make at least as many methods available as D; if a given method returns an enumeration, corresponding to a variant session type, then the set of values in C must be a subset of the set in D. When a method of D is overridden by a method of C, we allow contravariant changes in the parameter types and covariant changes in the result type. Subtyping between session types is defined in Section 3, but without subtyping on variant types, which is not needed in the present paper.

To support covariant changes in the result type, we can add the **restricts** declaration for enumerated types. An example is shown in Figure 4, where class NewBuyerAPI overrides method buy in such a way that, if the quote to the product is not in the quote store, the method issues a price request first. Notice that method price is both "public" (appears in the session type for the class) and the recipient of a self-call (unlike method SellerThread.reqQuote, which is not public). Our language distinguishes these two usages of the same method, by advancing the session type of the class in the first case but not in the second. Of course the self-call of price may change the types of the fields of NewBuyerAPI, but this is included in the effect of buy. Inheritance, in the sequential setting, is described in more detail in [52].

$$
\begin{array}{rrcl}
\text{Class/Enum dec} & D & ::= & \mathsf{class}\ C\ \{S; \vec{f}; \vec{M}\} \mid \mathsf{enum}\ E\ \{\vec{l}\} \\
\text{Types} & T & ::= & \mathsf{Null} \mid C[S] \mid E\ \mathsf{link}\ r \\
\text{Method dec} & M & ::= & T\ m(T\ x)\ \{e\} \\
\text{Values} & v & ::= & \mathsf{null} \mid l \\
\text{Paths} & r & ::= & \mathsf{this} \\
\text{Expressions} & e & ::= & v \mid x \mid r.f.m(e) \mid e; e \\
 & & & \mid \mathsf{new}\ C() \mid \mathsf{switch}\ (e)\ \{l : e_l\}_{l \in E} \\
 & & & \mid r.f \mid r.f\ =\ e \\
\text{Class session types} & S & ::= & \{m_i : S_i\}_{i \in I} \mid \langle l : S_l \rangle_{l \in E} \\
 & & & \mid X \mid \mu X.S
\end{array}
$$

Figure 5. Top level syntax.

$$
\begin{array}{rrcl}
\text{Types} & T & ::= & \ldots \mid C[F] \\
\text{Field types} & F & ::= & \{T_i\ f_i\}_{i \in I} \mid \langle l : F_l \rangle_{l \in E} \mid \bot \\
\text{Values} & v & ::= & \ldots \mid o \\
\text{Paths} & r & ::= & o \mid r.f \\
\text{Expressions} & e & ::= & \ldots \mid \mathsf{return}\ e\ \mathsf{from}\ r \\
\text{Object records} & R & ::= & C[\{f_i = v_i\}_{i \in I}] \\
\text{Heaps} & h & ::= & \varepsilon \mid h :: o = R \\
\text{States} & s & ::= & (h;\ e) \\
\text{Contexts} & \mathcal{E} & ::= & [\_] \mid \mathcal{E}; e \mid r.m(\mathcal{E}) \mid \mathsf{return}\ \mathcal{E}\ \mathsf{from}\ r \\
 & & & \mid \mathsf{switch}\ (\mathcal{E})\ \{l : e_l\}_{l \in E} \mid r.f\ =\ \mathcal{E}
\end{array}
$$

Identifier this is an instance of object identifier o.

Figure 6. Extended syntax for the type system and semantics.

3. A Core Sequential Language

We now present a formal syntax, operational semantics, and type system for a core sequential language. The main simplification is that all objects are treated as non-uniform and handled linearly by the type system. Incorporating standard (non-uniform) objects is straightforward, but it complicates and obscures the formal definitions. Our prototype implementation (Section 7) includes them. Also, all methods have exactly one parameter. In terms of expressivity this is not significant, as multiple parameters can be passed within an object, and a dummy parameter can be added if necessary. Anyway, it is easy to generalize the definitions, at the expense of slightly more complex notation. The calls **request**(), **accept**() and **receive**() should be regarded as abbreviations for **request**(**null**) etc. Finally, the examples use **void** methods, which are not in the formal language but can easily be added.

Syntax. We separate the syntax into the programmer's language (Figure 5) and the extensions required by the type system and operational semantics (Figure 6). Identifiers C, E, m, f and l are taken from disjoint countable sets representing names of classes, enumerations, methods, fields and labels respectively. Class, enumerated set and method declarations have been illustrated by the examples. A class declaration does not declare types for fields because they can vary at run-time. When an object is created, its fields are initialised to null.

There are some restrictions on the syntax of expressions. The programmer can only refer to fields of the current object, this; in other terms, all fields are private. Method call is only available on a field specification, not an arbitrary expression. The examples in Section 2 omit this as the prefix to all field accesses, but they can easily be inserted by the compiler.

Types are separated into object types and non-object types. The type of an object is $C[S]$, where C is a class name and S is a class session type. The type $C[S]$ is the view of an object from outside: the session type S shows which methods can be called, but the fields are not visible. The type Null has the single value null. The type E link r describes a label from the enumerated set E whose value will be used to resolve a variant type associated with object path r. For simplicity, the core language does not allow other uses of labels, hence E is not by itself a type.

Session types have been discussed in relation to the example. We refer to $\{m_i : S_i\}_{i \in I}$ as a branch type and to $\langle l : S_l \rangle_{l \in E}$ as a variant type. Session type end abbreviates the empty branch type $\{\}$. In contrast to variant types in functional languages, values are not tagged; instead the tag is stored in a value of type E link r, where r refers to the variantly typed object. The core language does not include named session types, or **typedef** or the **session** and **where** clauses from the examples; we just work with recursive session type expressions of the form $\mu X.S$, which are required to be *contractive*, i.e. containing no subexpression of the form $\mu X_1.\cdots\mu X_n.X_1$. We adopt the equi-recursive approach [43, Chapter 21] and regard $\mu X.S$ and $S\{^{(\mu X.S)}/_X\}$ as equivalent, using them interchangeably in any mathematical context.

It is worth noting that the type system, which we will describe later, enforces the following restrictions: nested variants are not permitted and in a class declaration, the initial session type is always a branch. They reflect the fact that variant types are tied to the result of a method call.

Figure 6 defines additional syntax needed for the formal system, not available to the programmer. Identifier o is taken from a set of *object identifiers* which includes this, the only identifier allowed in the programmer's language. There is an alternative form of object type, $C[F]$, which has a field typing instead of a session type. It represents the view of an object from within its own class and is used when typing method definitions. A field typing F can either be a record type associating one type to each field of the object or a variant field typing $\langle l : F_l \rangle_{l \in E}$, indexed by the values of an enumerated set E, similar to a variant session type. Type $\bot$ represents the uninhabited field typing which no object can have and can appear in variant types along with records, representing an impossible case.

Object records, heaps and states are used to define the operational semantics. A heap h ties object identifiers o to object records R. The identifiers are values, which may occur in expressions. The operation $h :: o = R$ represents adding a record for identifier o to the heap h and we consider it to be associative and commutative, that is, h is essentially an unordered set of bindings. *Paths* r, that occur in expressions to indicate where an object is, are extended to allow a toplevel object identifier followed by an arbitrary number of field specifications and serve to represent a location in the heap. In the toplevel syntax, the only known location is this, the current object. We use the following notation with respect to records, heaps and paths:

DEFINITION 1 (Heap locations).

- *If $R = C[\{f_i = v_i\}_{i \in I}]$, we define $R.f_i = v_i$ (for all i) and $R.\mathsf{class} = C$. For any value v and any $j \in I$, we also define $R\{f_j \mapsto v\} = C[\{f_i = v'_i\}_{i \in I}]$ where $v'_i = v_i$ for $i \neq j$ and $v'_j = v$.*

$$\text{(R-NEW)}\ \frac{o \text{ fresh} \qquad C.\mathsf{fields} = \vec{f}}{(h;\ \mathsf{new}\ C()) \longrightarrow (h :: o = C[\vec{f} = \overrightarrow{\mathsf{null}}];\ o)}$$

$$\text{(R-ACCESS)}\ \frac{h(r).f = v}{(h;\ r.f) \longrightarrow (h\{r.f \mapsto \mathsf{null}\};\ v)}$$

$$\text{(R-ASSIGN)}\ (h;\ r.f\ =\ v) \longrightarrow (h\{r.f \mapsto v\};\ \mathsf{null})$$

$$\frac{\_\, m(\_\, x)\ \{e\} \in h(r.f).\mathsf{class}}{(h;\ r.f.m(v)) \longrightarrow (h;\ \mathsf{return}\ e\{^{r.f}/_{\mathsf{this}}\}\{^{v}/_{x}\}\ \mathsf{from}\ r.f)}\ \text{(R-CALL)}$$

$$\text{(R-RETURN)}\ (h;\ \mathsf{return}\ v\ \mathsf{from}\ r) \longrightarrow (h;\ v)$$

$$\text{(R-SWITCH)}\ \frac{l_0 \in E}{(h;\ \mathsf{switch}\ (l_0)\ \{l : e_l\}_{l \in E}) \longrightarrow (h;\ e_{l_0})}$$

$$\text{(R-SEQ)}\ (h;\ v; e) \longrightarrow (h;\ e)$$

$$\text{(R-CONTEXT)}\ \frac{(h;\ e) \longrightarrow (h';\ e')}{(h;\ \mathcal{E}[e]) \longrightarrow (h';\ \mathcal{E}[e'])}$$

Figure 7. Reduction rules for states.

- *If $h = (h' :: o = R)$, we define $h(o) = R$, and for any field f of R, $h\{o.f \mapsto v\} = (h' :: o = R\{f \mapsto v\})$.*
- *If $r = r'.f$ and $h(r').f = o$, then we also define $h(r) = h(o)$ and $h\{r.f' \mapsto v\} = h\{o.f' \mapsto v\}$.*
- *In any other case, these operations are not defined. Note in particular that $h(r)$ is not defined if r is a path that exists in h but does not point to an object identifier.*

Finally, the return expression is used to represent an ongoing method call; a state consists of a heap and an expression; $\mathcal{E}$ are evaluation contexts in the style of Wright and Felleisen [53].

Programs. A program consists of a collection of class and enum declarations D. The semantic and typing rules we will present next are implicitly parameterized by the set of these declarations. It is assumed that the whole set is available at any point and that any class, enum or label is declared only once. We consider that enum declarations define sets of labels, and use the notation $l \in E$ accordingly. As opposed to labels, we do not require the sets of method or field names to be disjoint from one class to another. We will use the following notation: if class $C\ \{S; \vec{f}; \vec{M}\}$ is one of the declarations, $C.\mathsf{session}$ means S and $C.\mathsf{fields}$ means $\vec{f}$, and if $T\ m(T'\ x)\ \{e\} \in \vec{M}$ then $C.m$ is e.

Operational Semantics. Figure 7 defines an operational semantics on states $(h;\ e)$ consisting of a heap and an expression. All rules have the implicit premise that the expressions appearing in them must be defined, for example $r.f$ only reduces if $h(r)$ is an object record containing a field named f. An example of reduction, together with typing, will be presented at the end of the section in Figure 10.

R-NEW creates a new object in the heap, with null fields. R-ACCESS extracts the value of a field from an object in the heap. Linear control of objects requires that the field be nullified. R-ASSIGN updates the value of a field. The value of the assignment, as an expression, is null; linearity means that it cannot be v as in Java. R-CALL wraps the method body, with the full path to the object instance substituted for this and the actual parameter substituted for the formal one, in a return expression that is used for type preservation. R-RETURN then unwraps the resulting value.

R-SWITCH is standard. R-SEQ discards the result of the first part of a sequential composition. R-CONTEXT is the usual rule for reduction in contexts.

$$\frac{S <: S'}{C[S] <: C[S']} \qquad \frac{\forall i \in I \qquad T_i <: T_i'}{C[\{T_i\ f_i\}_{i \in I}] <: C[\{T_i'\ f_i\}_{i \in I}]}\ \text{(S-SESS,S-FIELD)}$$

Figure 8. Definition of subtyping.

Subtyping. The source of subtyping in our language is the sub-session relation, coinductively defined as follows:

DEFINITION 2 (Sub-session). $<:$ *is the largest relation on class session types such that:*

- *If $\{m_i : S_i\}_{i \in I} <: S'$ then $S' = \{m_j : S_j'\}_{j \in J}$ with $J \subseteq I$ and $\forall j \in J, S_j <: S_j'$.*
- *If $\langle l : S_l \rangle_{l \in E} <: S'$ then $S' = \langle l : S_l' \rangle_{l \in E}$ with $\forall l.\ S_l <: S_l'$.*

Like the definition of subtyping for channel session types [24], the type that allows a choice to be made (the branch type here, the $\oplus$ type in [24]) has contravariant subtyping in the set of choices. Further details, including the proof that subtyping is reflexive and transitive and an algorithm for checking subtyping, can be adapted from [24].

Figure 8 defines subtyping between types of our language. There is no subtyping between classes; the sub-session relation induces subtyping between session-typed objects (S-SESS), and for field-typed objects, subtyping on the fields propagates to the records (S-FIELD).

Type System. The type system for the toplevel language is defined by the rules in Figure 9 and by Definition 4 below. They use typing environments of the form $\Gamma = y_1 : T_1, \ldots, y_n : T_n$ where we use y to stand for either object identifiers o or variables x. As for heaps, we consider environments an unordered set of bindings; in other words, the comma is associative and commutative. Similarly to heaps also, we use the following notation to access arbitrary paths in an environment:

DEFINITION 3 (Locations in environments).

- *If $\Gamma = \Gamma', y : T$ then we define $\Gamma(y) = T$ and $\Gamma\{y \mapsto T'\} = \Gamma', y : T'$*
- *Inductively, if $r = r'.f_j$, and if $\Gamma(r') = C[\{T_i\ f_i\}_{i \in I}]$ and $j \in I$, then we define $\Gamma(r) = T_j$ and $\Gamma\{r \mapsto T'\} = \Gamma\{r' \mapsto C[\{T_i'\ f_i\}_{i \in I}]\}$ where $T_i' = T_i$ for $i \neq j$ and $T_j' = T'$.*
- *In any other case, in particular if $\Gamma(r')$ is of the form $C[S]$, these operations are not defined.*

We also write $\Gamma <: \Gamma'$ if for every y in $dom(\Gamma')$ we have $y \in dom(\Gamma)$ and $\Gamma(y) <: \Gamma'(y)$. We say a type is *simple* if it is either a base (non-object) type or an object with a branch session type.

The typing judgement for expressions is $\Gamma \triangleright e : T \triangleleft \Gamma'$. Here Γ and Γ' are the initial and final type environments when typing e. Γ' may differ from Γ either because identifiers disappear (due to linearity) or because their types change (if they are non-uniform objects). These judgements are constructed by rules T-LINVAR to T-SUBENV; we comment on them later but first explain how a session type can be checked against a class. We use the following coinductive definition to relate the views of an object from inside (fields) and from outside (session):

DEFINITION 4. *For any class C, we define the relation $F \vdash C : S$ between field typings F and session types S as the largest relation such that $F \vdash C : S$ implies either $F = \bot$ or:*

- *If $S = \{m_i : S_i\}_{i \in I}$, then for all i in I there is a definition $T_i\ m_i(T_i'\ x_i)\ \{e_i\}$ in the declaration of class C such that we*

$$\text{(T-LinVar)}\ \Gamma, x : C[S] \triangleright x : C[S] \triangleleft \Gamma \qquad \text{(T-Var)}\ \frac{T \neq C[\_]}{\Gamma, x : T \triangleright x : T \triangleleft \Gamma, x : T} \qquad \text{(T-Assign)}\ \frac{\Gamma \triangleright e : T \triangleleft \Gamma' \qquad \Gamma'(r.f) \text{ is simple}}{\Gamma \triangleright r.f \ = \ e : \mathsf{Null} \triangleleft \Gamma'\{r.f \mapsto T\}}$$

$$\text{(T-Call)}\ \frac{\Gamma \triangleright e : T' \triangleleft \Gamma' \qquad \Gamma'(r) = C[\{m_i : S_i\}_{i\in I}] \qquad j \in I \qquad T\ m_j(T'\ x)\ \{\_\} \in C}{\Gamma \triangleright r.m_j(e) : T\{^r/_{\mathsf{this}}\} \triangleleft \Gamma'\{r \mapsto C[S_j]\}} \qquad \text{(T-Null)}\ \Gamma \triangleright \mathsf{null} : \mathsf{Null} \triangleleft \Gamma$$

$$\text{(T-New)}\ \Gamma \triangleright \mathsf{new}\ C() : C[C.\mathsf{session}] \triangleleft \Gamma \qquad \text{(T-InjF)}\ \frac{\Gamma(r) = C[F_{l_0}] \qquad l_0 \in E \qquad \text{no } F_l \text{ contains a variant}}{\Gamma \triangleright l_0 : E\ \mathsf{link}\ r \triangleleft \Gamma\{r \mapsto C[\langle l : F_l\rangle_{l\in E}]\}}$$

$$\text{(T-Access)}\ \frac{\Gamma(r.f) = T \qquad T \text{ is simple}}{\Gamma \triangleright r.f : T \triangleleft \Gamma\{r.f \mapsto \mathsf{Null}\}} \qquad \text{(T-Seq)}\ \frac{\Gamma \triangleright e : T \triangleleft \Gamma'' \qquad \Gamma'' \triangleright e' : T' \triangleleft \Gamma' \qquad T \neq E\ \mathsf{link}\ r}{\Gamma \triangleright e; e' : T' \triangleleft \Gamma'}$$

$$\text{(T-Switch)}\ \frac{\Gamma \triangleright e : E\ \mathsf{link}\ r \triangleleft \Gamma'' \qquad \Gamma''(r) = C[\langle l : S_l\rangle_{l\in E}] \qquad \forall l \in E, \Gamma''\{r \mapsto C[S_l]\} \triangleright e_l : T \triangleleft \Gamma'}{\Gamma \triangleright \mathsf{switch}\ (e)\ \{l : e_l\}_{l\in E} : T \triangleleft \Gamma'}$$

$$\text{(T-Sub)}\ \frac{\Gamma \triangleright e : T \triangleleft \Gamma' \qquad T <: T'}{\Gamma \triangleright e : T' \triangleleft \Gamma'} \qquad \text{(T-SubEnv)}\ \frac{\Gamma \triangleright e : T \triangleleft \Gamma' \qquad \Gamma' <: \Gamma''}{\Gamma \triangleright e : T \triangleleft \Gamma''} \qquad \text{(T-Class)}\ \frac{\overrightarrow{\mathsf{Null}}\ \vec{f} \vdash C : S}{\vdash \mathsf{class}\ C\ \{S; \vec{f}; \vec{M}\}}$$

Figure 9. Typing rules for the toplevel language

$$\begin{array}{rcll}
o : C[C'[\{m_i : S_i\}_{i\in I}]\ f, T\ g] & \triangleright & o.g = o.f.m_j(); \mathsf{switch}\ (o.g)\ \{l\colon e_l\}_{l\in E} & \to \text{(R-Call)} \\
o : C[C'[F]\ f, T\ g] & \triangleright & o.g = \mathsf{return}\ e\{^{o.f}/_{\mathsf{this}}\}\ \mathsf{from}\ o.f; \mathsf{switch}\ (o.g)\ \{l\colon e_l\}_{l\in E} & \to^* \\
o : C[C'[F_{l_0}]\ f, T\ g] & \triangleright & o.g = \mathsf{return}\ l_0\ \mathsf{from}\ o.f; \mathsf{switch}\ (o.g)\ \{l\colon e_l\}_{l\in E} & \to \text{(R-Return)} \\
o : C[C'[S_{l_0}]\ f, T\ g] & \triangleright & o.g = l_0; \mathsf{switch}\ (o.g)\ \{l\colon e_l\}_{l\in E} & \to \text{(R-Assign, R-Seq)} \\
o : C[C'[\langle l : S_l\rangle_{l\in E}]\ f, (E\ \mathsf{link}\ o.f)\ g] & \triangleright & \mathsf{switch}\ (o.g)\ \{l\colon e_l\}_{l\in E} & \to \text{(R-Access)} \\
o : C[C'[S_{l_0}]\ f, \mathsf{Null}\ g] & \triangleright & \mathsf{switch}\ (l_0)\ \{l\colon e_l\}_{l\in E} & \to \text{(R-Switch)} \\
o : C[C'[S_{l_0}]\ f, \mathsf{Null}\ g] & \triangleright & e_{l_0} &
\end{array}$$

Figure 10. Example of the interplay between method call, switch and link types (heaps and rightmost typing environment omitted).

have $x_i : T_i'$, $\mathsf{this} : C[F] \triangleright e_i : T_i \triangleleft \mathsf{this} : C[F_i]$ *with* F_i *such that* $F_i \vdash C : S_i$ *and if* $F_i = \langle l : \_\rangle_{l\in E}$ *then* $T_i = E\ \mathsf{link}\ \mathsf{this}$.

- *If* $S = \langle l : S_l\rangle_{l\in E}$, *then* $F = \langle l : F_l\rangle_{l\in E}$ *and for any* l *in* E *we have* $F_l \vdash C : S_l$.

The relation $F \vdash C : S$ represents the fact that an object with internal type $C[F]$ can be safely viewed from outside as having type $C[S]$. First note that $\bot$ can only be used as a component of a variant field typing and represents a case that never occurs, hence its particular status in the definition: any session type at all is compatible with it, because it is internally known that the label will never have the corresponding value. The second point accounts for correspondence between variant types. The main point is the first: if the object has internal type $C[F]$ and its session type allows a certain method to be called, then it means that the method body is typable with an initial type of $C[F]$ for this and the declared type for the parameter. Furthermore, the type of the expression must match the declared return type and the final type of this must be compatible with the subsequent session type. In the particular case where the final type is a variant, the returned value must be the tag of that variant, hence have the corresponding link type.

We now comment on the rules of Figure 9. The last rule T-Class requires consistency between the declared session type of a class and the initial null field typing. The others are for expressions. T-Var and T-LinVar are used to access a method's parameter, removing it from the environment if it has an object type (linear). For simplicity, this is the only way to use a parameter, in particular we do not allow calling methods directly on them: to call a method on a parameter, it must first be assigned to a field. T-Access types field access, nullifying the field because its value has moved into the expression part of the judgement. T-Assign types field update; the type of the field changes, and the type of the expression is Null, again because of linearity. In both rules, the restriction to simple types (either a base, non-object, type or an object with a branch session type) is to avoid invalidating link types. T-New types a new object, giving it the initial session type from the class declaration. T-Seq accounts for the effects of the first expression on the environment and checks that a label is not discarded, which would leave the associated variant unusable.

T-Call requires an environment in which method $r.m_j$ is available. The type of the parameter is checked as usual, and the final environment Γ' is updated to contain the new session type of the object sitting at location r. The substitution occurring in the type of the call expression is only relevant when the return type of the method is of the form E link this, meaning that the type of the object after the call is a variant session whose tag is the value returned. In that case, the type becomes E link r to indicate that the result really describes the state of the object at r.

T-InjF constructs a variant type. More precisely, it is used to give a variant field typing to an object from within; the literal label which constitutes the expression is the tag of the variant type, thus the variant case corresponding to that particular label is the actual type of the object and the others are arbitrary. Note that when typing method bodies, r is always this, as there cannot be anything else in the environment which has a type of the form $C[F]$. It is also the only rule for typing labels, as they are only used in association with variants.

T-Switch types a switch expression; the type of the argument must be a link to a location with a variant session type. All branches must have the same final environment Γ', so that it is a consistent final environment for the switch expression. An interesting particular case is if T is of the form E' link this: then the different expressions may return different labels and modify the fields' types in different ways, and T-InjF allows those cases to be unified into a single variant type.

T-Sub is a standard subsumption rule, and T-SubEnv allows subsumption in the final environment. The main use of the latter

rule is to enable the branches of a switch to be given the same final environments.

Example of reduction and typing. Figure 10 illustrates the operational semantics and the way in which the environment used to type an expression changes as the expression reduces (see Theorem 1, Section 5).

The initial expression is

$$o.g = o.f.m_j(); \mathsf{switch}\ (o.g)\ \{\mathsf{case}\ l : e_l\}_{l \in E}$$

where for simplicity we have ignored the parameter of m_j. The initial typing environment is

$$o : C[C'[\{m_i : S_i\}_{i \in I}]\ f, T\ g]$$

where $S_j = \langle l : S_l \rangle_{l \in E}$. The body of method m_j is e with the typing

$$\mathsf{this} : C'[F] \triangleright e : E\ \mathsf{link}\ \mathsf{this} \triangleleft \mathsf{this} : C'[\langle l : F_l \rangle_{l \in E}]$$

and we suppose that m_j returns $l_0 \in E$. According to Definition 4 and the typing of the declaration of class C' we have $F_{l_0} \vdash C' : S_{l_0}$ and $F \vdash C' : \{m_i : S_i\}_{i \in I}$.

The figure shows the environment in which each expression is typed; the environment changes as reduction proceeds, for several reasons explained below. The typing of an expression is $\Gamma \triangleright e : T \triangleleft \Gamma'$ but we only show Γ because Γ' does not change and T is not the interesting part of this example. We also omit the heap, showing the typing of expressions instead of states. Calling $o.f.m_j()$ changes the type of field f to $C'[F]$ because we are now inside the object. As e reduces to l_0 the type of f may change, finally becoming $C'[F_{l_0}]$ so that it has the component of the variant field typing $\langle l : F_l \rangle_{l \in E}$ corresponding to l_0. The reduction by R-RETURN changes the type of f to $C'[S_{l_0}]$ because we are now outside the object again, but the type is still the component of a variant typing corresponding to l_0. The assignment changes the type of f again, to $C'[\langle l : S_l \rangle_{l \in E}]$, which is $C'[S_j]$, the type we were expecting after the method call. At this point the information about which component of the variant typing we have is stored in $o.g$. The type of the expression $o.f.m_j()$ is E link $o.f$, which appears as the type of $o.g$ after the assignment is executed. Extracting the value of $o.g$, in order to switch on it, nullifies $o.g$ and so the type E link $o.f$ disappears from the environment and becomes the type of the subexpression $o.g$, at the same time resolving the variant type of f according to the particular enumerated value l_0.

Extension: self-calls and recursive methods. The rules in Figure 11 extend the language to include method calls on this and recursive methods. Recursive calls are also self-calls. To simplify the formal system, self-calls have their own syntax, which is not necessary in the implementation. Self-calls do not check or advance the session type. A method that is only self-called does not appear in the session type. A method that is self-called and called from outside appears in the session type, and calls from outside do check and advance the session type. The reason why it is safe to not check the session type for self-calls is that the effect of the self-call on the field typing is included in the effect of the method that calls it. All of the necessary checking of session types is done because of the original outside call that eventually leads to the self-call.

Because they are not in the session type, self-called methods must be explicitly annotated with their initial (req) and final (ens) field typings. The annotations are used to type self-calls and method definitions.

If a method is in the session type then its body is checked by the first hypothesis of T-CLASS, but the annotations (if present) are ignored except when they are needed to check recursive calls. If a method has an annotation then its body is checked by the second hypothesis of T-CLASS. If both conditions apply then the body is checked twice. The implementation can optimize this.

Syntax (top-level) :

$$\begin{aligned} M &::= \ \ldots \ \mid \mathsf{req}\ F\ \mathsf{ens}\ F\ \mathsf{for}\ T\ m(T\ x)\ \{e\} \\ e &::= \ \ldots \ \mid r\#m(e) \end{aligned}$$

Reduction rule :

$$\text{(R-SELFCALL)}\ \frac{\_\ m(\_\ x)\ \{e\} \in h(r).\mathsf{class}}{(h;\ r\#m(v)) \longrightarrow (h;\ e\{^r/_{\mathsf{this}}\}\{^v/_x\})}$$

Typing rule (expressions) :

$$\text{(T-SELFCALL)}\ \frac{\Gamma \triangleright e : T' \triangleleft \Gamma' \qquad \Gamma'(r) = C[F] \qquad \mathsf{req}\ F\ \mathsf{ens}\ F'\ \mathsf{for}\ T\ m(T'\ x)\ \{e\} \in C}{\Gamma \triangleright r\#m(e) : T \triangleleft \Gamma'\{r \mapsto C[F']\}}$$

Typing rule (annotated method definitions) : T-ANNOTMETH :

$$\frac{x : T', \mathsf{this} : C[F] \triangleright e : T \triangleleft \mathsf{this} : C[F'] \qquad F' \neq \langle\_\rangle}{\vdash_C \mathsf{req}\ F\ \mathsf{ens}\ F'\ \mathsf{for}\ T\ m(T'\ x)\ \{e\}}$$

Replacement for T-CLASS :

$$\frac{\overrightarrow{\mathsf{Null}}\ \vec{f} \vdash C : S \qquad \forall m \in \vec{M}.\ (m\ \mathsf{has}\ \mathsf{req/ens} \Rightarrow\ \vdash_C m)}{\vdash \mathsf{class}\ C\ \{S; \vec{f}; \vec{M}\}}$$

Figure 11. Rules for recursive methods and other self-calls

$$\begin{array}{rrcl} \text{Declarations} & D & ::= & \ldots \mid \mathsf{access}\ \langle\Sigma\rangle\ n \\ \text{Values} & v & ::= & \ldots \mid c^+ \mid c^- \mid n \\ \text{Expressions} & e & ::= & \ldots \mid \mathsf{spawn}\ C.m(e) \\ \text{Contexts} & \mathcal{E} & ::= & \ldots \mid \mathsf{spawn}\ C.m(\mathcal{E}) \\ \text{Types} & T & ::= & \ldots \mid \langle\Sigma\rangle \\ \text{Message types} & B & ::= & \mathsf{Null} \mid \langle\Sigma\rangle \mid \mathsf{Chan}[S] \\ \text{Channel session types} & \Sigma & ::= & ?[B]\,.\,\Sigma \mid \&\{l : \Sigma_l\}_{l \in E} \\ & & & \mid\ ![B]\,.\,\Sigma \mid \oplus\{l : \Sigma_l\}_{l \in E} \\ & & & \mid\ X \mid \mu X.\Sigma \\ \text{States} & s & ::= & \ldots \mid s \parallel s \mid (\nu c)\,s \end{array}$$

Figure 12. Additional syntax for channels and states

An annotated method cannot produce a variant field typing or have a link type, because T-SWITCH can only analyze a variant session type.

Extension: while loops. The language can easily be extended to include while loops. The reduction rule defines while recursively in terms of switch, and the typing rule is derived straightforwardly from T-SWITCH.

4. A Core Distributed Language

We now define a distributed language based on the idea of a *configuration*, which is a parallel collection of threads (heap-expression pairs) representing separate locations. States, which represented a single thread in Figure 6, are extended in Figure 12 to represent such a parallel configuration. The expressions in different locations can communicate via synchronous messages on point-to-point channels. The new syntax and reduction rules are defined in Figures 12 and 13; they have already been illustrated by the examples in Section 2. The primitive operations **send** and **receive** are treated

Structural congruence: E-COMM, E-ASSOC, E-SCOPE

$$s_1 \parallel s_2 \equiv s_2 \parallel s_1 \qquad s_1 \parallel (s_2 \parallel s_3) \equiv (s_1 \parallel s_2) \parallel s_3 \qquad s_1 \parallel (\nu c)s_2 \equiv (\nu c)(s_1 \parallel s_2) \text{ if } c^+, c^- \text{ not free in } s_1$$

Additional reduction rules and typing rules for expressions:

$$\text{(R-INIT)}\ \frac{h(r).f = n \qquad h'(r').f' = n \qquad c \text{ fresh}}{(h;\ \mathcal{E}[r.f.\mathsf{accept}()]) \parallel (h';\ \mathcal{E}'[r'.f'.\mathsf{request}()]) \longrightarrow (\nu c)\left((h;\ \mathcal{E}[c^+]) \parallel (h';\ \mathcal{E}'[c^-])\right)} \qquad \text{(R-PAR)}\ \frac{s \longrightarrow s'}{s \parallel s'' \longrightarrow s' \parallel s''}$$

$$\text{(R-COM)}\ \frac{h(r).f = c^p \qquad h'(r').f' = c^{\overline{p}}}{(h;\ \mathcal{E}[r.f.\mathsf{send}(v)]) \parallel (h';\ \mathcal{E}'[r'.f'.\mathsf{receive}()]) \longrightarrow (h;\ \mathcal{E}[\mathsf{null}]) \parallel (h';\ \mathcal{E}'[v])} \qquad \text{(R-STR)}\ \frac{s \equiv s' \quad s' \longrightarrow s'' \quad s'' \equiv s'''}{s \longrightarrow s'''}$$

$$\text{(R-SPAWN)}\ \frac{o \text{ fresh} \qquad C.\mathsf{fields} = \vec{f} \qquad \_\, m(\_\, x)\ \{e\} \in C}{(h;\ \mathcal{E}[\mathsf{spawn}\ C.m(v)]) \longrightarrow (h;\ \mathcal{E}[\mathsf{null}]) \parallel (o = C[\vec{f} = \vec{\mathsf{null}}];\ e\{^o/_{\mathsf{this}}\}\{^v/_x\})} \qquad \text{(R-NEWCHAN)}\ \frac{s \longrightarrow s'}{(\nu c)\, s \longrightarrow (\nu c)\, s'}$$

$$\text{(T-SPAWN)}\ \frac{\Gamma \triangleright e : B \triangleleft \Gamma' \qquad C.\mathsf{session} = \{m_i : \_\}_{i \in I} \qquad j \in I \quad \_\, m_j(B\, x)\ \{\_\} \in C}{\Gamma \triangleright \mathsf{spawn}\ C.m_j(e) : \mathsf{Null} \triangleleft \Gamma'} \qquad \text{(T-NAME)}\ \frac{n.\mathsf{protocol} = \Sigma}{\Gamma \triangleright n : \langle \Sigma \rangle \triangleleft \Gamma}$$

$$\text{(T-ACCEPT)}\ \frac{\Gamma(r.f) = \langle \Sigma \rangle}{\Gamma \triangleright r.f.\mathsf{accept}() : \mathsf{Chan}[[\![\Sigma]\!]] \triangleleft \Gamma} \qquad \text{(T-REQUEST)}\ \frac{\Gamma(r.f) = \langle \Sigma \rangle}{\Gamma \triangleright r.f.\mathsf{request}() : \mathsf{Chan}[[\![\overline{\Sigma}]\!]] \triangleleft \Gamma}$$

Figure 13. Reduction and typing rules for concurrency and channels

Given a channel session type Σ, define a class session type $[\![\Sigma]\!]$:

$$\begin{aligned}
[\![X]\!] &= X \\
[\![\mu X.\Sigma]\!] &= \mu X.\, [\![\Sigma]\!] \\
[\![?\,[T]\,.\,\Sigma]\!] &= \{\mathsf{receive}_T : [\![\Sigma]\!]\} \\
[\![!\,[T]\,.\,\Sigma]\!] &= \{\mathsf{send}_T : [\![\Sigma]\!]\} \\
[\![\&\,\{l : \Sigma_l\}_{l \in E}]\!] &= \{\mathsf{receive}_E : \langle l : [\![\Sigma_l]\!] \rangle_{l \in E}\} \\
[\![\oplus\,\{l : \Sigma_l\}_{l \in E}]\!] &= \{\mathsf{send}_l : [\![\Sigma_l]\!]\}_{l \in E}
\end{aligned}$$

and method signatures:

$$\begin{aligned}
T\ \mathsf{receive}_T() \qquad & \qquad \mathsf{Null}\ \mathsf{send}_T(T\ x) \\
(E\ \mathsf{link\ this})\ \mathsf{receive}_E() \qquad & \qquad \mathsf{Null}\ \mathsf{send}_l()
\end{aligned}$$

Figure 14. Translation of a channel session type into a class session type.

as method names m. A channel has two endpoints, c^+ and c^-, on which **send** and **receive** can be called; each endpoint has a session type Σ.

We write $\overline{\Sigma}$ for the *dual* of Σ, obtained by exchanging $\&/\oplus$ and $?/!$. The two endpoints of a channel have dual types, just as in previous work [24]. In $(\nu c)s$, νc binds c^+ and c^-. We write c^p for an unspecified endpoint and $c^{\overline{p}}$ for its partner. The other new value is n, which ranges over access points (service names) that can be used to initialize channels by interaction between $\mathsf{request}()$ and $\mathsf{accept}()$. These access points are announced in a way similar to class and enum declarations, and we use the notation $n.\mathsf{protocol}$ to mean the protocol (session type) Σ associated to access point n, similarly to $C.\mathsf{session}$. Access points must be announced with the same type in all locations; we assume some mechanism to enforce or check this restriction. The type of an access point is $\langle \Sigma \rangle$, and the new channel endpoints will have types Σ and $\overline{\Sigma}$. The definitions at the level of configurations are similar to previous work on session types for functional languages [25, 51]. As well as R-INIT, the crucial rule is R-COM for synchronous communication. In the definition of channel session types, messages have non-object types. In the core language this means that messages can only be channel endpoints, access points or null, but we could easily add non-object base types. The reason for not allowing objects as messages is to avoid the complication of defining the transfer of an object and all of its subobjects from one heap to another. It is not a fundamental restriction.

As explained in Section 2, $\mathsf{spawn}\ C.m(e)$ creates a new component of the configuration, with a new local heap containing an instance of class C on which $m(e)$ is called.

Figure 14 defines a class session type for each channel session type Σ. A channel with type Σ is treated as an object with type $\mathsf{Chan}[[\![\Sigma]\!]]$ where Chan is a distinguished class name. Environments Γ are extended to allow channel endpoints c^p in addition to object identifiers and variables. The operations send and $\mathsf{receive}$ are typed as method calls, and the channel remains available for further communication. Figure 14 defines different send_T and $\mathsf{receive}_T$ methods for each type T, but the implementation omits the T and uses the session type and/or the parameter type to disambiguate. Rule R-COM ignores the subscript. Also, R-COM treats send_l as $\mathsf{send}(l)$; the message must be a literal label.

Access points do not behave like objects; new typing rules are needed for them (Figure 13, bottom line). T-NAME types an access point as a literal value, and T-ACCEPT and T-REQUEST are used to type channel creation.

5. Properties of the Type System

In order to state a type preservation theorem, we first need to extend the type system to states. This is done in Figure 15. First of all there are a few more rules for expressions: T-REF allows typing an object identifier and T-CHAN a literal channel endpoint. T-INJS complements T-INJF by allowing a literal label to be the tag of a variant session type as well as of a variant field typing (at top level, variant session types can only come from method calls). T-RETURN serves to type a return expression, representing an ongoing method call in the object at r. The expression e is typed in an environment where this object's fields are accessible, but the return has the effect of 'closing' the object by reverting its type to the outside view of a session. The technical substitution in $\Gamma'(r)$ only applies to the link types which may be contained in F; it is due to the fact that $F \vdash C : S$ (Definition 4) uses judgements in an environment where the object is this.

The next four rules define a relation $\Theta; \Gamma \vdash h$ between a channel environment Θ, a typing environment Γ and a heap h. Θ is similar to a regular typing environment but only contains types for channel endpoints; thus in the purely sequential setting it is always empty.

$$\text{(T-REF)}\ \frac{T \text{ is simple}}{\Gamma, o : T \triangleright o : T \triangleleft \Gamma} \quad \text{(T-CHAN)}\ \Gamma, c^p : T \triangleright c^p : T \triangleleft \Gamma \quad \text{(T-INJS)}\ \frac{\Gamma(r) = C[S_{l_0}] \qquad l_0 \in E \qquad \text{All } S_l \text{ are branches}}{\Gamma \triangleright l_0 : E \ \mathsf{link}\ r \triangleleft \Gamma\{r \mapsto C[\langle l : S_l \rangle_{l \in E}]\}}$$

$$\text{(T-RETURN)}\ \frac{\Gamma \triangleright e : T \triangleleft \Gamma' \quad \text{If } T = E \ \mathsf{link}\ r' \text{ then } r' = r \qquad \Gamma'(r) = C[F\{^{r}/_{\mathsf{this}}\}] \qquad F \vdash C : S}{\Gamma \triangleright \mathsf{return}\ e\ \mathsf{from}\ r : T \triangleleft \Gamma'\{r \mapsto C[S]\}} \quad \text{(T-HEMPTY)}\ \Theta; \Theta \vdash \varepsilon$$

$$\text{(T-HADD)}\ \frac{\Theta; \Gamma_0 \vdash h \qquad \forall i \in \{1 \dots n\}, \begin{cases} \Gamma_{i-1} \triangleright v_i : T_i \triangleleft \Gamma_i & \text{if } T_i \text{ is simple, or} \\ \Gamma_{i-1} = \Gamma_i, v_i : T_i & \text{if it is not} \end{cases} \qquad C.\mathsf{fields} = (f_i)_{1 \leqslant i \leqslant n}}{\Theta; (\Gamma_n, o : C[\{T_i\ f_i\}_{1 \leqslant i \leqslant n}])\{^{\mathsf{link}\ o.f_i.\vec{\varphi}}/_{\mathsf{link}\ v_i.\vec{\varphi}}\} \vdash h :: o = C[\{f_i = v_i\}_{1 \leqslant i \leqslant n}]}$$

$$\text{(T-HIDE)}\ \frac{\Theta; \Gamma, o : C[F] \vdash h \qquad F\{^{\mathsf{this}}/_{o}\} \vdash C : S}{\Theta; \Gamma, o : C[S] \vdash h} \qquad \text{(T-STATE)}\ \frac{\Theta; \Gamma \vdash h \qquad \Gamma \triangleright e : T \triangleleft \Gamma'}{\Theta; \Gamma \triangleright (h;\ e) : T \triangleleft \Gamma'}$$

$$\text{(T-THREAD)}\ \frac{\Theta; \Gamma \triangleright (h;\ e) : T \triangleleft \Gamma'}{\Theta \vdash (h;\ e)} \quad \text{(T-PAR)}\ \frac{\Theta \vdash s \qquad \Theta' \vdash s'}{\Theta + \Theta' \vdash s \parallel s'} \quad \text{(T-NEWCHAN)}\ \frac{\Theta, c^+ : \mathsf{Chan}[[\![\Sigma]\!]], c^- : \mathsf{Chan}[[\![\overline{\Sigma}]\!]] \vdash s}{\Theta \vdash (\nu c)\, s}$$

Figure 15. Additional typing rules for the proofs

The rules are technical, but essentially they enforce restrictions on the contents of Γ: a channel endpoint can only appear in it if it is also in Θ with the same type (T-HEMPTY). An object can only appear in it if it is in the heap and either has a field typing consistent with its field values (T-HADD) or has a session type consistent with this field typing (T-HIDE). T-HADD also takes care of removing from the top level environment whatever goes into the fields of the objects; this includes channel endpoints, thus Γ can end up containing fewer channel endpoints than Θ even though the starting point of the derivation is always T-HEMPTY. Finally, T-STATE defines that a typing judgement holds for a given single-threaded program state if it holds for the corresponding expression *and* the initial typing environment is compatible with the heap and the channel environment. The remaining rules are for distributed configurations and we comment on them later.

By standard techniques [53] adapted to typing judgements with initial and final environments [26] we can prove the expected results about an individual thread. Assume that we are working relative to a set of well-typed declarations.

THEOREM 1 (Type Preservation). *If* $\Theta; \Gamma \triangleright (h;\ e) : T \triangleleft \Gamma'$ *and* $(h;\ e) \longrightarrow (h';\ e')$ *then there exists* Γ'' *such that* $\Theta; \Gamma'' \triangleright (h';\ e') : T \triangleleft \Gamma'$.

THEOREM 2 (No Stuck States). *If* $\emptyset; \Gamma \triangleright (h;\ e) : T \triangleleft \Gamma'$ *then either* e *is a value or there exists* h' *and* e' *such that* $(h;\ e) \longrightarrow (h';\ e')$.

Notice that in Theorem 2 the channel environment must be empty. Otherwise, the thread might be waiting for a communication and thus unable to reduce by itself.

We also have *conformance* of sequences of method calls to session types.

DEFINITION 5 (Call Traces). *A* call trace *on an object* o *is a sequence* $m_1\alpha_1 m_2\alpha_2 \dots$ *where each* m_i *is a method name and each* α_i *is either an enumeration label or nothing.*

Following the operational semantics, it is possible to define a call trace for every object, excluding self-calls. A session type defines a set of call traces, which is simply the set of paths through the session type regarded as a labelled directed graph. We state the result informally to avoid presenting a sequence of very technical definitions.

THEOREM 3 (Conformance). *When executing a typed program, the call trace of every object is one of the traces of the initial session type of its class.*

PROOF (Sketch): Similar to the proof of Theorem 1, with a stronger invariant. Rule T-CALL shows that every method call conforms to the current session type of the target object. The most interesting case is a reduction by R-RETURN when the value is a label l: the call trace is extended by l and the session type of the object advances to the corresponding option which will eventually be selected by a switch on l. □

Distributed setting. The three last rules of Figure 15 describe how distributed configurations are typed. T-THREAD extracts the channel environment from the typing of a single thread. T-PAR merges two environments ($+$ represents disjoint union; it is not defined if the domains overlap). T-NEWCHAN checks for duality. We use $\vdash s$ as an abbreviation for $\emptyset \vdash s$; this represents well-typedness of a closed configuration. We have the following result:

THEOREM 4. *If* $\vdash s$ *and* $s \longrightarrow s'$ *then* $\vdash s'$.

PROOF (Sketch): In order to do an inductive proof we need to state a similar result for configurations with free channels. It relies on the concept of a *balanced* channel environment, similarly to previous work on session types in π-calculus [24], which is roughly defined as follows: Θ is *balanced* if whenever $\Theta(c^+) = \mathsf{Chan}[[\![\Sigma]\!]]$ and $\Theta(c^-) = \mathsf{Chan}[[\![\Sigma']\!]]$ then $\Sigma' = \overline{\Sigma}$.

We argue that if s is typed in a balanced environment and communication takes place on channel c, then the endpoints have dual session types and the communication advances both of them, so they remain dual and the resulting environment, which types s', is also balanced. Reduction internal to a thread does not affect Θ as stated in Theorem 1, R-SPAWN does not affect channels either, and R-INIT introduces a bound channel whose endpoints' types are dual because access points have the same type in all locations. □

In the distributed language, call traces can also be defined for channel endpoints. Because of the translation from channel session types to class session types, these call traces correspond to the sequence and type of messages. We therefore have, stated informally:

COROLLARY 1 (to Theorem 3). *When executing a typed configuration, the sequence of communication operations on every channel endpoint conforms to its session type.*

Furthermore, we have the following safety result:

THEOREM 5 (No Communication Errors). *Suppose that we have* $s \equiv (\nu \vec{c})(s' \parallel (h;\ \mathcal{E}[r.f.m(v)]) \parallel (h';\ \mathcal{E}[r'.f'.m'(v')]))$ *with* $h(r).f = c^+$ *and* $h'(r').f' = c^-$.

If $\vdash s$, *then there exists* s'' *such that* $s \longrightarrow s''$.

Note that by setting s' to something which cannot reduce (e.g. $(\varepsilon;\ \mathsf{null})$) we obtain more precisely that the particular reduction

which consists of R-COM applied to the two rightmost components is always possible. This means in particular that if m is send then m' is receive and conversely. This theorem complements Theorem 2 in the case of communication: if the reducible part of the expression in a thread is a method call on a *channel endpoint*, which was not allowed in Theorem 2, then it is still able to reduce provided another thread calls a method on the other endpoint.

6. Typechecking Algorithm

Figure 16 defines a typechecking algorithm for the language. Algorithm $\mathcal{A}$ is used to check the relation $F \vdash C : S$; it uses internally, for recursive calls, a set Δ of assumptions which is needed because of the coinductive definition of this relation. If typing succeeds, then this set is returned, else nothing is returned. The actual contents of the set returned are not relevant at the top level.

The algorithm for checking subtyping is not described here but is similar to the one defined for channel session types in [24]. We write $\mathsf{sup}(S, S')$ for the least upper bound of S and S' with respect to subtyping, and extend it to $\mathsf{sup}(C[S], C[S'])$, requiring the same C in both types. It is defined by taking the intersection of sets of methods and the least upper bound of their continuations. Details of a similar definition (greatest lower bound of channel session types) can be found in [36].

A program is typechecked by checking, for every class C, that $\mathcal{A}_C(C.\mathsf{session}, \vec{\mathsf{Null}}\, C.\mathsf{fields}, \emptyset)$ returns something. This corresponds to checking T-CLASS. Algorithm $\mathcal{A}$ uses algorithm $\mathcal{B}$ to check method definitions. The definition of $\mathcal{B}$ follows the typing rules (Figure 9) except for one point: T-INJF means that the rules are not syntax-directed. To compensate, clause l produces a *partial* variant field typing with an incomplete set of labels, and clause switch uses the $\uplus$ operator to combine partial variants and check for consistency. Then the operation $\mathsf{comp}(F)$ used in algorithm $\mathcal{A}$ transforms a partial variant into a true variant by adding $\bot$ in the missing cases. The various "where" and "if" clauses should be interpreted as conditions for the functions to be defined; cases in which the functions are undefined should be interpreted as typing errors.

The typechecking algorithm is modular in the sense that to check class C we only need to know the session types of other classes, not their method definitions.

THEOREM 6. *Algorithm $\mathcal{A}$ always terminates, either with an error (and then the function $\mathcal{A}$ is undefined) or with a result.*

PROOF: Similar to proofs about algorithms for coinductively-defined subtyping relations [43]. □

THEOREM 7. *$\mathcal{A}_C(S,\, F,\, \emptyset)$ is defined if and only if $F \vdash C : S$.*

PROOF (Sketch): 'If' direction: we prove by induction on the number of recursive calls the more general result that if Δ_0 is such that $(F, S) \in \Delta_0$ implies $F \vdash C : S$ and if $F_0 \vdash C : S_0$ holds, then $\mathcal{A}_C(S_0,\, F_0,\, \Delta_0)$ is defined.

'Only if' direction: if $\mathcal{A}_C(S_0,\, F_0,\, \emptyset)$ is defined, we look at its evaluation and define the following relation: $F\ \mathcal{R}\ S$ iff $\mathcal{A}_C$ gets called with parameters F and S at some point. We then prove that $\mathcal{R}$ satisfies the hypotheses of Definition 4, hence is included in the largest such relation, and conclude by noticing that we have $F_0\ \mathcal{R}\ S_0$. □

7. Implementation

We have used the Polyglot [41] system to implement the ideas of this paper as a prototype extension to Java 1.4, which we call Bica. This includes type-checking method calls against the class session

$\mathcal{A}_C(S,\, \bot,\, \Delta) = \Delta$

$\mathcal{A}_C(S,\, F,\, \Delta) = \Delta$ if $(F, S) \in \Delta$

$\mathcal{A}_C(\mu X.S,\, F,\, \Delta) = \mathcal{A}_C(S\{^{\mu X.S}/_X\},\, F,\, \Delta \cup \{(F, \mu X.S)\})$

$\mathcal{A}_C(\{m_i : S_i\}_{1 \leqslant i \leqslant n},\, F,\, \Delta_0) = \Delta_n$
where for $i = 1$ to n, $\Delta_i = \mathcal{A}_C(S_i,\, \mathsf{comp}(F_i),\, \Delta_{i-1})$
where
$T_i\ m_i(U_i\ x_i)\ \{e_i\} \in C$ and
$(T_i',\, F_i,\, \_) = \mathcal{B}_C(e_i,\, F,\, x_i : U_i)$ and
$T_i' <: T_i$ and
if $\mathsf{comp}(F_i) = \langle l : ... \rangle_{l \in E}$ then $U_i = E$ link this

$\mathcal{A}_C(\langle l : S_l \rangle_{l \in E},\, \langle l : F_l \rangle_{l \in E},\, \Delta_0) = \Delta_n$
where $E.\mathsf{labels} = \{l_1 \ldots l_n\}$ and
for $i = 1$ to n, $\Delta_i = \mathcal{A}_C(S_{l_i},\, F_{l_i},\, \Delta_{i-1})$

$\mathcal{B}_C(\mathsf{null},\, F,\, \Gamma) = (\mathsf{Null},\, F,\, \Gamma)$

$\mathcal{B}_C(n,\, F,\, \Gamma) = (\langle n.\mathsf{protocol} \rangle,\, F,\, \Gamma)$

$\mathcal{B}_C(x,\, F,\, x : T) = (T,\, F,\, \Gamma)$
where $\Gamma = \emptyset$ if T is linear or $x : T$ otherwise

$\mathcal{B}_C(\mathsf{this}.f,\, F,\, \Gamma) = (T,\, F\{f \mapsto \mathsf{Null}\},\, \Gamma)$
where $T = F(f)$ and T is simple

$\mathcal{B}_C(l,\, F,\, \Gamma) = (E\ \mathsf{link\ this},\, \langle l : F \rangle,\, \Gamma)$
where $l \in E$

$\mathcal{B}_C(\mathsf{new}\ C'(),\, F,\, \Gamma) = (C'[C'.\mathsf{session}],\, F,\, \Gamma)$

$\mathcal{B}_C(\mathsf{this}.f = e,\, F,\, \Gamma) = (\mathsf{Null},\, F'\{f \mapsto T\},\, \Gamma')$
where $(T,\, F',\, \Gamma') = \mathcal{B}_C(e,\, F,\, \Gamma)$ and $F(f)$ is simple

$\mathcal{B}_C(\mathsf{this}.f.m_j(e),\, F,\, \Gamma) = (T\{^f/_{\mathsf{this}}\},\, F'\{f \mapsto C'[S_j]\},\, \Gamma')$
where $(U',\, F',\, \Gamma') = \mathcal{B}_C(e,\, F,\, \Gamma)$ and
$F'(f) = C'[\{m_i : S_i\}_{i \in I}]$ and $j \in I$ and
$T\ m_j(U\ x)\ \{\_\} \in C'$ and $U' <: U$

$\mathcal{B}_C(\mathsf{switch}\ (e)\ \{l : e_l\}_{l \in E},\, F,\, \Gamma) = (T,\, \biguplus_{l \in E} F_l',\, \Gamma'')$
where $(E\ \mathsf{link}\ f,\, F',\, \Gamma') = \mathcal{B}_C(e,\, F,\, \Gamma)$ and
$F'(f) = C'[\langle l : S_l \rangle_{l \in E}]$ and
$\forall l \in E, (T,\, F_l',\, \Gamma_l) = \mathcal{B}_C(e_l,\, F'\{r \mapsto C'[S_l]\},\, \Gamma')$ and
$\Gamma'' = \bigcap_{l \in E} \Gamma_l$

$\mathcal{B}_C(e; e',\, F,\, \Gamma) = \mathcal{B}_C(e',\, F',\, \Gamma')$
where $(\_,\, F',\, \Gamma') = \mathcal{B}_C(e,\, F,\, \Gamma)$

$\mathcal{B}_C(\mathsf{this}.f.\mathsf{accept}(),\, F,\, \Gamma) = (\mathsf{Chan}[[\![\Sigma]\!]],\, F,\, \Gamma)$
where $F(f) = \langle \Sigma \rangle$

$\mathcal{B}_C(\mathsf{this}.f.\mathsf{request}(),\, F,\, \Gamma) = (\mathsf{Chan}[\overline{[\![\Sigma]\!]}],\, F,\, \Gamma)$
where $F(f) = \langle \Sigma \rangle$

$\mathcal{B}_C(\mathsf{spawn}\ C'.m_j(e),\, F,\, \Gamma) = (\mathsf{Null},\, F',\, \Gamma')$
where $(B',\, F',\, \Gamma') = \mathcal{B}_C(e,\, F,\, \Gamma)$ and
$C'.\mathsf{session} = \{m_i : S_i\}_{i \in I}$ and $j \in I$ and
$T\ m_j(B\ x)\ \{\_\} \in C'$ and $B' <: B$

Combining partial variants

$\{T_i\ f_i\}_{i \in I} \uplus \{T_i'\ f_i\}_{i \in I} = \{\mathsf{sup}(T_i, T_i')\ f_i\}_{i \in I}$

$\langle l : F_l \rangle_{l \in I} \uplus \langle l : F_l' \rangle_{l \in J} = \langle l : F_l'' \rangle_{l \in I \cup J}$
where $F_l'' = F_l \uplus F_l'$ if $l \in I \cap J$, F_l if $l \notin J$, F_l' if $l \notin I$

$\mathsf{comp}(F) = F$ if F is not a partial variant

$\mathsf{comp}(\langle l : F_l \rangle_{l \in I}) = \langle l : F_l \rangle_{l \in E}$ if $I \subseteq E$, where $F_l = \bot$ for $l \notin I$

Figure 16. Typechecking algorithm.

types of non-uniform objects, and inheritance as outlined in Section 2, but not yet generating class session types from channel session types. Bica supports shared as well as linear objects, standard as well as session-related conditionals, switch, while-loops, and return values. Bica is implemented on top of the JL5 Polyglot extension in order to cater to enumerated types as well as to allow Java 5 features to be added later. The semantics of Bica is standard Java. It is available from `http://gloss.di.fc.ul.pt/bica/`.

We have begun to experiment with defining session types for iterators and collections from the Java 1.4.2 `java.util` package. For iterators this is a straightforward process. Other cases are not so easy; API documentation is not always explicit about the protocol for sequences of calls. It will be necessary to experiment with naturally-occurring client code in order to determine the most suitable session types. Further results about Bica and annotation of APIs will be reported in future publications.

8. Related Work

Previous work on session types for object-oriented languages. Several recent papers by Dezani-Ciancaglini, Yoshida *et al.* [10, 17–19, 31, 38] have combined session types, as specifications of protocols on communication channels, with the object-oriented paradigm. A characteristic of all of these works is that a channel is always created and used within a single method call. It is possible for a method to delegate a channel by passing it to another method, but it is not possible to modularize session implementations as we do, by storing a channel in a field of an object and allowing several methods to use it. We are also able to interleave sessions on different channels. The most recent work in this line [10] unifies sessions and methods, and continues the idea that a session is a complete entity. Mostrous and Yoshida [38] add sessions to Abadi and Cardelli's object calculus.

Non-uniform concurrent objects / active objects. Another related line of research was started by Nierstrasz [40], aimed at describing the behaviour of non-uniform *active* objects in concurrent systems, whose behaviour (including the set of available methods) may change dynamically. He defined subtyping for active objects, but did not formally define a language semantics or a type system. The topic has been continued, in the context of process calculi, by several authors [9, 45–47]. Caires [9] is the most relevant work; it uses an approach based on spatial logic to give very fine-grained control of resources, and Militão [37] has implemented a Java prototype based on this idea. Damiani *et al.* [14] define a concurrent Java-like language incorporating inheritance and subtyping and equipped with a type-and-effect system, in which method availability is made dependent on the state of objects.

The distinctive feature of our approach to non-uniform objects, in comparison with all of the above work, is that we allow an object's abstract state to depend on the result of a method call. This gives a very nice integration with the branching structure of channel session types, and with subtyping.

Typestates. Based on the fact that method availability depends on an object's internal state (the situation identified by Nierstrasz, as mentioned above), Strom and Yemini propose *typestates* [48]. The concept consists of identifying the possible states of an object and defining pre- and post-conditions that specify in which state an object should be so that a given method would be available, and in which state the method execution would leave the object.

Vault [15, 20] follows the typestates approach. It uses linear types to control aliasing, and uses the *adoption and focus* mechanism [20] to re-introduce aliasing in limited situations. *Fugue* [16, 21] extends similar ideas to an object-oriented language, and uses explicit pre- and post-conditions.

Bierhoff and Aldrich [5] also work on a typestates approach in an object-oriented language, defining a sound modular automated static protocol checking setting. They define a state and method refinement relation achieving a behavioural subtyping relation. The work is extended with access permissions, that combines typestate with aliasing information about objects [4], and with concurrency, via the atomic bloc synchronization primitive used in transactional memory systems [3]. Like us, they allow the typestate to depend on the result of a method call. *Plural* is a prototype tool that embodies their approach, providing automated static analysis in a concurrent object-oriented language [6]. To evaluate their approach they annotated and verified several standard Java APIs [7].

Finally, *Sing#* [22] is an extension of C# which has been used to implement Singularity, an operating system based on message-passing. It incorporates session types to specify protocols for communication channels, and introduces typestate-like *contracts* The published paper [22] does not discuss the relationship between channel contracts and non-uniform objects or typestates, and does not define a formal language. A technical point is that *Sing#* uses a single construct switch receive to combine receiving an enumeration value and doing a case-analysis, whereas our system allows a switch on an enumeration value to be separated from the method call that produces it.

Session types and typestates are related approaches, but there are stylistic and technical differences. With respect to the former, session types are like labelled transition systems or finite-state automata, capturing the behaviour of an object. When developing an application, one may start from session types and then implement the classes. Typestates take each transition of a session type and attach it to a method as pre- and post-conditions. With respect to technical differences, the main ones are: (a) session types unify types and typestates in a single class type as a global behavioural specification; (b) our subtyping relation is structural, while the typestates refinement relation is nominal; (c) *Plural* uses a software transactional model as concurrency control mechanism (thus, shared memory), which is lighter and easier than locks, but one has to mark atomic blocks in the code, whereas our communication-centric model (using channels) is simpler and allows us to use the same type abstraction (session types) instead of a new programming construct; moreover, channel-based communication also allows us to specify the client-server communication protocol as the channel session type, and to implement it modularly, in several methods which may even be in different classes; (d) typestates approaches allow flexible aliasing control, whereas our approach uses only linear objects (to add better alias/access control is simple and an orthogonal issue).

Static verification of protocols. *Cyclone* [27] and *CQual* [23] are systems based on the C programming language that allow protocols to be statically enforced by a compiler. *Cyclone* adds many benefits to C, but its support for protocols is limited to enforcing locking of resources. Between acquiring and releasing a lock, there are no restrictions on how a thread may use a resource. In contrast, our system uses types both to enforce locking of objects (via linearity) and to enforce the correct sequence of method calls. *CQual* expects users to annotate programs with type qualifiers; its type system, simpler and less expressive than the above, provides for type inference.

Unique ownership of objects. In order to demonstrate the key idea of modularizing session implementations by integrating session-typed channels and non-uniform objects, we have taken the simplest possible approach to ownership control: strict linearity of non-uniform objects. This idea goes back at least to the work of Baker [2] and has been applied many times. However, linearity causes problems of its own: linear objects cannot be stored in

shared data structures, and this tends to restrict expressivity. There is a large literature on less extreme techniques for static control of aliasing: Hogg's *Islands* [28], Almeida's *balloon types* [1], Clarke *et al.*'s *ownership types* [13], Fähndrich and DeLine's *adoption and focus* [20], Östlund *et al.*'s Joe$_3$ [42] among others. In future work we intend to use an off-the-shelf technique for more sophisticated alias analysis. The property we need is that when changing the type of an object (by calling a method on it or by performing a switch or a while on an enumeration constant returned from a method call) there must be a unique reference to it.

Resource usage analysis. Igarashi and Kobayashi [32] define a general resource usage analysis problem for an extended λ-calculus, including a type inference system, that statically checks the order of resource usage. Although quite expressive, their system only analyzes the sequence of method *calls* and does not consider branching on method *results* as we do.

Analysis of concurrent systems using pi-calculus. Some work on static analysis of concurrent systems expressed in pi-calculus is also relevant, in the sense that it addresses the question (among others) of whether attempted uses of a resource are consistent with its state. Kobayashi *et al.* have developed a generic framework [33] including a verification tool [34] in which to define type systems for analyzing various behavioural properties including sequences of resource uses [35]. In some of this work, types are themselves abstract processes, and therefore in some situations resemble our session types. Chaki *et al.* [12] use CCS to describe properties of pi-calculus programs, and verify the validity of temporal formulae via a combination of type-checking and model-checking techniques, thereby going beyond static analysis.

All of this pi-calculus-based work follows the approach of modelling systems in a relatively low-level language which is then analyzed. In contrast, we work directly with the high-level abstractions of session types and objects.

9. Conclusions

We have extended existing work on session types for object-oriented languages by allowing the implementation of a session to be divided between several methods which can be called independently. This supports a modular approach which is absent from previous work. Technically, it is achieved by integrating session types for communication channels and a static type system for non-uniform objects. A session-typed channel is one kind of non-uniform object, but objects whose fields are non-uniform are also, in general, non-uniform. Typing guarantees that the sequence of messages on every channel, and the sequence of method calls on every non-uniform object, satisfy specifications expressed as session types.

We have formalized the syntax, operational semantics and static type system of a core distributed class-based object-oriented language incorporating these ideas. Soundness of the type system is expressed by type preservation, conformance and correct communication theorems. The type system includes a form of typestates and uses simple linear type theory to guarantee unique ownership of non-uniform objects. Somewhat unusually, it allows the state of an object after a method call to depend on the result of the call, if this is of an enumerated type.

We have illustrated our ideas with an example based on e-commerce, and described a prototype implementation. By incorporating further standard ideas from the related literature, it should be straightforward to extend the implementation to a larger and more practical language.

In the future we intend to work on the following topics. (1) More flexible control of aliasing. The mechanism for controlling aliasing should be orthogonal to the theory of how operations affect uniquely-referenced objects. We intend to adapt existing work to relax our strictly linear control and obtain a more flexible language. (2) Java-style interfaces. If class C **implements** interface I then we should have $\mathsf{session}(C) <: \mathsf{session}(I)$, interpreting the interface as a specification of minimum method availability. (3) Specifications involving several objects. Multi-party session types [8, 30] specify protocols with more than two participants. It would be interesting to adapt that theory into a type system for more complex patterns of object usage.

Acknowledgments

Gay was partially supported by the UK EPSRC (EP/E065708/1 "Engineering Foundations of Web Services" and EP/F037368/1). He thanks the University of Glasgow for the sabbatical leave during which part of this research was done. Gay and Ravara were partially supported by the Security and Quantum Information Group at Instituto de Telecomunicações, Portugal. Caldeira, Ravara, and Vasconcelos were partially supported by the EU IST proactive initiative FET-Global Computing (project Sensoria, IST–2005–16004). Vasconcelos was partially supported by the Large-Scale Informatics Systems Laboratory, Portugal. Ravara was partially supported the Portuguese Fundação para a Ciência e a Tecnologia FCT (SFRH/BSAB/757/2007), and by the UK EPSRC (EP/F037368/1 "Behavioural Types for Object-Oriented Languages"). Gesbert was supported by the UK EPSRC (EP/E065708/1). We thank Jonathan Aldrich and Luís Caires for helpful discussions.

References

[1] P. S. Almeida. Balloon types: Controlling sharing of state in data types. *ECOOP, Springer LNCS*, 1241:32–59, 1997.

[2] H. G. Baker. 'Use-once' variables and linear objects — storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, 1995.

[3] N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *OOPSLA '08*, pages 227–244. ACM Press, 2008. ISBN 978-1-60558-215-3. doi: http://doi.acm.org/10.1145/1449764.1449783.

[4] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA '07*, pages 301–320. ACM Press, 2007. ISBN 978-1-59593-786-5. doi: http://doi.acm.org/10.1145/1297027.1297050.

[5] K. Bierhoff and J. Aldrich. Lightweight object specification with typestates. In *13th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE '05)*, pages 217–226. ACM Press, 2005.

[6] K. Bierhoff and J. Aldrich. PLURAL: checking protocol compliance under aliasing. In *ICSE Companion '08*, pages 971–972. ACM Press, 2008. ISBN 978-1-60558-079-1. doi: http://doi.acm.org/10.1145/1370175.1370213.

[7] K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *ECOOP '09*, pages 195–219, 2009.

[8] E. Bonelli and A. Compagnoni. Multipoint session types for a distributed calculus. *TGC, Springer LNCS*, 4912:240–256, 2007.

[9] L. Caires. Spatial-behavioral types for concurrency and resource control in distributed systems. *Theoret. Comp. Sci.*, 402(2–3):120–141, 2008.

[10] S. Capecchi, M. Coppo, M. Dezani-Ciancaglini, S. Drossopoulou, and E. Giachino. Amalgamating sessions and methods in object-oriented languages with generics. *Theoret. Comp. Sci.*, 410:142–167, 2009.

[11] M. Carbone, K. Honda, and N. Yoshida. Structured global programming for communication behaviour. *ESOP, Springer LNCS*, 4421:2–17, 2007.

[12] S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. *POPL, ACM SIGPLAN Notices*, 37(1):45–57, 2002.

[13] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. *OOPSLA, ACM SIGPLAN Not.*, 33(10):48–64, 1998.

[14] F. Damiani, E. Giachino, P. Giannini, and S. Drossopoulou. A type safe state abstraction for coordination in Java-like languages. *Acta Informatica*, 45(7–8):479–536, 2008. ISSN 0001-5903. URL `http://pubs.doc.ic.ac.uk/stateAbstrCoordJava/`.

[15] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. *PLDI, ACM SIGPLAN Notices*, 36(5):59–69, 2001.

[16] R. DeLine and M. Fähndrich. The Fugue protocol checker: is your software Baroque? Technical Report MSR-TR-2004-07, Microsoft Research, 2004.

[17] M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopolou. A distributed object-oriented language with session types. *TGC, Springer LNCS*, 3705:299–318, 2005.

[18] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopolou. Session types for object-oriented languages. *ECOOP, Springer LNCS*, 4067:328–352, 2006.

[19] M. Dezani-Ciancaglini, S. Drossopoulou, E. Giachino, and N. Yoshida. Bounded session types for object-oriented languages. *FMCO, Springer LNCS*, 4709:207–245, 2007.

[20] M. Fähndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. *PLDI, ACM SIGPLAN Notices*, 37 (5):13–24, 2002.

[21] M. Fähndrich and R. DeLine. Typestates for objects. *ESOP, Springer LNCS*, 3086:465–490, 2004.

[22] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*. ACM, 2006.

[23] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. *PLDI, ACM SIGPLAN Notices*, 37(5):1–12, 2002.

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

[25] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 2009. URL `http://www.dcs.gla.ac.uk/~simon/publications/Lin-Async.pdf`. To appear.

[26] S. J. Gay, A. Ravara, and V. T. Vasconcelos. Session types for inter-process communication. Technical Report TR-2003-133, Comp. Sci., Univ. Glasgow, 2003.

[27] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. *PLDI, ACM SIGPLAN Notices*, 37(5):282–293, 2002.

[28] J. Hogg. Islands: aliasing protection in object-oriented languages. *OOPSLA, ACM SIGPLAN Notices*, 26(11):271–285, 1991.

[29] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. *ESOP, Springer LNCS*, 1381:122–138, 1998.

[30] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *POPL, ACM SIGPLAN Notices*, 43(1):273–284, 2008.

[31] R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in Java. *ECOOP, Springer LNCS*, 5142:516–541, 2008.

[32] A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Trans. on Programming Languages and Systems*, 27(2):264–313, 2005.

[33] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004.

[34] N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4–5):291–347, 2005.

[35] N. Kobayashi, K. Suenaga, and L. Wischik. Resource usage analysis for the π-calculus. *Logical Methods in Comp. Sci.*, 2(3:4):1–42, 2006.

[36] L. G. Mezzina. *Typing Services*. PhD thesis, IMT Institute for Advanced Studies, Lucca, Italy, 2009.

[37] F. Militão. Design and implementation of a behaviorally typed programming system for web services. Master's thesis, New University of Lisbon, 2008.

[38] D. Mostrous and N. Yoshida. A session object calculus for structured communication-based programming. Submitted, 2008.

[39] M. Neubauer and P. Thiemann. An implementation of session types. *PADL, Springer LNCS*, 3057:56–70, 2004.

[40] O. Nierstrasz. Regular types for active objects. In *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.

[41] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for Java. *Compiler Construction, Springer LNCS*, 2622:138–152, 2003.

[42] J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, uniqueness and immutability. In *IWACO (ECOOP workshop)*, 2007.

[43] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[44] R. Pucella and J. A. Tov. Haskell session types with (almost) no class. In *Proceedings, 1st ACM SIGPLAN symposium on Haskell*, pages 25–36. ACM, 2008.

[45] F. Puntigam. State inference for dynamically changing interfaces. *Computer Languages*, 27:163–202, 2002.

[46] F. Puntigam and C. Peter. Types for active objects with static deadlock prevention. *Fundamenta Informaticæ*, 49:1–27, 2001.

[47] A. Ravara and V. T. Vasconcelos. Typing non-uniform concurrent objects. *CONCUR, Springer LNCS*, 1877:474–488, 2000.

[48] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986. ISSN 0098-5589.

[49] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. *PARLE, Springer LNCS*, 817:398–413, 1994.

[50] A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of software components using session types. *Fundamenta Informaticæ*, 73(4):583–598, 2006.

[51] V. T. Vasconcelos, S. J. Gay, and A. Ravara. Typechecking a multithreaded functional language with session types. *Theoret. Comp. Sci.*, 368(1–2):64–87, 2006. URL `http://www.di.fc.ul.pt/~vv/papers/vasconcelos.gay.ravara:tychecking-session-types.pdf`.

[52] V. T. Vasconcelos, S. J. Gay, A. Ravara, N. Gesbert, and A. Z. Caldeira. Dynamic interfaces. FOOL, 2009.

[53] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.