Modular Session Types for Distributed Object-Oriented Programming

Simon J. Gay[1], Vasco T. Vasconcelos[2], António Ravara[3], Nils Gesbert[1], and Alexandre Z. Caldeira[2]

[1] Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK. Email: `simon@dcs.gla.ac.uk`, `nils@dcs.gla.ac.uk`
[2] Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa, 1749-016 Lisboa, Portugal. Email: `vv@di.fc.ul.pt`, `zua@di.fc.ul.pt`
[3] Security and Quantum Information Group, Instituto de Telecomunicações, and Departamento de Matemática, Instituto Superior Técnico, Universidade Técnica de Lisboa, 1049-001 Lisboa, Portugal. Email: `amar@math.ist.utl.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 by allowing a session (protocol) implementation to be *modularized*: it can be separated into separately-callable methods. We achieve this by integrating session-typed communication channels into a type system for non-uniform objects, based on a form of typestates. We formalize a small distributed class-based object-oriented language in which static typing guarantees that both sequences of messages on channels, and sequences of method calls on objects, conform to type-theoretic specifications. The formalization includes both a type *system* and a typechecking *algorithm*. The language includes expected features of session types, such as delegation, and expected features of object-oriented programming, such as inheritance. We also describe a prototype implementation as an extension of Java.

1 Introduction

Session types [25, 42] allow communication protocols to be specified type-theoretically so that protocol implementations can be verified by static type-checking. A session type describes an agent's view of 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 [20, 21, 33, 38, 44], component-based object systems [43], object-oriented languages [5, 12–14, 26, 32], operating system services [15] and more general service-oriented systems [6]. Session types have also been generalized from two-party to multi-party systems [3, 24], 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 [9, 17]) that we have previously introduced under the name of *dynamic interfaces* [45]. 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 separately. 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 messages on every channel follows the specification of its session type; second, that the sequence of method calls on every non-uniform object follows the specification of its class's session type. We have also formalized a typechecking algorithm 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 6. 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. This is made possible by our integration of channels and non-uniform objects.

– In contrast to other type systems for non-uniform objects, we use a global specification of method availability, inspired by session types, which constitutes a class type. From it, we calculate pre- and post-conditions of methods during type-checking. Moreover, we allow the state of an object to depend on the result of a method call.

The remainder of the paper is structured as follows. In Section 2 we illustrate our system by introducing an example and then extending it to include inheritance. In Section 3 we formalize a core language and state the type safety results. The core language requires, in addition to the class session type, explicit pre- and post-conditions for each method. In Section 4 we present a typechecking algorithm which infers the pre- and post-conditions from the session types. Section 5 describes our prototype implementation. Section 6 contains a more extensive discussion of related work, Section 7 outlines future work and Section 8 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

$$\begin{aligned} S = \oplus_c\{\ & \mathsf{requestQuote} : ![\mathsf{Product}]\,.\,?[\mathsf{Price}]\,.\,?[\mathsf{Quote}]\,.\,S, \\ & \mathsf{acceptQuote} : ![\mathsf{Quote}]\,.\,![\mathsf{Payment}]\,.\,S, \\ & \mathsf{quit} : \mathsf{end}\} \end{aligned}$$

The buyer has a choice between requestQuote, acceptQuote and quit. If she chooses requestQuote 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 S, and the buyer can choose another option. When she wants to buy a product, the buyer can select acceptQuote 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 acceptQuote or requestQuote, terminates the protocol.

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

$$\begin{aligned} \overline{S} = \&_c\{\ & \mathsf{requestQuote} : ?[\mathsf{Product}]\,.\,![\mathsf{Price}]\,.\,![\mathsf{Quote}]\,.\,\overline{S}, \\ & \mathsf{acceptQuote} : ?[\mathsf{Quote}]\,.\,?[\mathsf{Payment}]\,.\,\overline{S}, \\ & \mathsf{quit} : \mathsf{end}\} \end{aligned}$$

in which send (!) and receive (?) are exchanged, and the choice constructor ($\oplus_c$) is replaced by the branch constructor ($\&_c$). This means that the seller must be ready to respond to all of the three choices that the buyer can make.

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 sites, some fixed and well-known (global names, analogous to URLs) and others dynamically created, execute independent programs. The primitive **service** S creates a name n where service providers and clients to services meet in order to initiate a session, *i.e.*, to run the protocol specified by S. A point-to-point communication channel is then created by the interaction of operations n.request() and n.accept() executed at separate sites. Given a channel c, synchronous communication occurs through the interaction of c.send and c.receive operations.

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 in Section 2, it can form the basis for an inheritance hierarchy of classes that offer more services.

The code in Figure 1 consists of four declarations. Lines 1 and 3 define enumerated types Option and Result. Lines 5–7 define the session type S of the channel protocol. By convention this is defined from the server's (seller's) viewpoint; in the formal language the same type is associated globally with a service name. Lines 9–42 define the class BuyerAPI. Because it has a field of type $\overline{S}$ the class BuyerAPI is non-uniform. We specify the availability of methods by the *session declaration* in lines 10–15. We refer to this as a *class session type* to distinguish it from *channel session types* such as S. An object of class BuyerAPI has abstract states Init, Shop, Pay and **end**. The 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 [11, 17], except that we collect the whole specification into the class session type instead of annotating the method definitions with pre- and post-conditions. Actually our formal system, defined in Section 3, does use pre- and post-conditions, which are calculated from the class session type during type-checking.

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 line 13, where the $\oplus$ constructor specifies a collection of states 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.

```
enum Option { requestQuote, acceptQuote, quit }

enum Result {ok, error}

typedef S = &{ Option.requestQuote: ?[Product].![Price].![Quote].S,
               Option.acceptQuote: ?[Quote].?[Payment].S,
               Option.quit: end }

class BuyerAPI {
  session Init
  where Init = &{ init: Shop }
        Shop = &{ price: Shop,
                  buy: ⊕{Result.ok: Pay, Result.error: Shop},
                  stop: end }
        Pay  = &{ pay: Shop }

  c; qs; // fields, initially null:Null

  void init(<S> n) {
    c = n.request();
    qs = new QuoteStore(); qs.init();
  }
  Price price(Product p) {
    c.send(Option.requestQuote);
    c.send(p);
    Price pr = c.receive();
    Quote q = c.receive();
    qs.add(p,q);
    return pr;
  }
  Result buy(Product p) {
    Quote q = qs.get(p);
    if (q == null)
      return Result.error;
    else {
      c.send(Option.acceptQuote);
      c.send(q);
      return Result.ok;
  }
  void pay(Payment p) { c.send(p); }
  void stop() { c.send(Option.quit); }
}
```

Fig. 1. An API for the buyer.

Method init uses n.**request**() to create a channel. The identifier n is a name (like a port address), declared as a parameter of init with channel session type S.

```
b = new BuyerAPI();
b.init();
while(b.price(myProduct) > 100) {}; // Wait until price is right
switch(b.buy(myProduct)) {
  case error: print("What have I done wrong?"); break;
  case ok: b.pay(myPayment); break;
}
b.stop();
```

Fig. 2. A buyer — code fragment.

Method init 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 the initialization part of a constructor. Likewise, we assume that after the call to QuoteStore.init(), object qs is in some state Q to each all other methods in the class lead.

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 4. 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 (class Buyer in Figure 2, for example), 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 unique object references (in particular, aliasing should be tightly controlled). 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. This restrictive approach may be eased by using an off-the-shelf solution to aliasing control; in Section 6 we present several candidates.

```
class Seller {
  session &{ main: end }

  void main(<S> n) { // S defined in BuyerAPI
    while (true)
      spawn SellerThread.run(n.accept());
} }

class SellerThread {
  session &{ run: end }

  void run(S x) {
    switch (x.receive()) {
      case requestQuote: reqQuote(x);
      case acceptQuote: accQuote(x);
      case quit: break
  } }
  void reqQuote(?[Product].![Price].![Quote].S x) {
    Product p = x.receive();
    x.send(...); // Calculate price
    x.send(...); // Quote reference
    run(x)
  }
  void accQuote(?[Quote].?[Payment].S x) {
    Quote q = x.receive();
    Price py = x.receive();
    ... // Process payment
    run(x)
} }
```

Fig. 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. 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 body of main repeatedly executes **spawn** SellerThread.run(n.**accept**()). The semantics of this statement is as follows. The expression n.**accept**() creates a channel by interacting with a matching n.**request**() at another site, and evaluates to the endpoint c^+ so that we have **spawn** SellerThread.run(c^+). According to our threads mechanism, this creates a new thread with its own heap containing an instance of SellerThread on which run(c^+) is called. This 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.

```
enum NewResult restricts Result {ok}

class NewBuyerAPI extends BuyerAPI {
  @Override
  NewResult buy(Product p) {
    if (!qs.contains(p)) price(p);
    c.send(Option.acceptQuote);
    c.send(qs.get(p));
    return NewResult.ok;
} }
```

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

Notice that the methods reqQuote and accQuote of 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. The formal language of the present paper does not include **while** loops, but they are defined in our previous work [45].

Seller.main is parameterized on the service name n, so a suitably typed name must be provided when the seller is launched. We assume an external mechanism for checking that names are typed consistently across all sites. This could be a trusted central repository of typed services, or Hu's [26] system of run-time type-checks when **request** and **accept** interact.

Inheritance and Subtyping. We allow a class C to inherit from (extend) 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 allowed 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 $\oplus$ 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.

To support covariant changes in the result type, our language includes 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.

Class dec	D	$::=$	$\mathsf{class}\ C\ \{S; \vec{f}; \vec{M}\}\ \mid\ \mathsf{enum}\ E\ L$
Constant sets	L	$::=$	$\{l_i\}_{i\in I}$
Method dec	M	$::=$	$T\ m(T\ x)\ \{e\}$
Values	v	$::=$	$\mathsf{null}\ \mid\ E.l\ \mid\ o$
Value references	r	$::=$	$v\ \mid\ x\ \mid\ o.f$
Expressions	e	$::=$	$r\ \mid\ o.f\ =\ e\ \mid\ \mathsf{new}\ C()\ \mid\ r.m(e)\ \mid\ e;e\ \mid$
			$\mathsf{switch}\ (e)\ \{\mathsf{case}\ l_i\colon e_i\}_{i\in I}\ \mid\ \mathsf{spawn}\ C.m(e)$
Non-object types	B	$::=$	$\mathsf{Null}\ \mid\ E\ \mid\ \Sigma\ \mid\ \langle\Sigma\rangle$
Types	T	$::=$	$B\ \mid\ C[S]$
Class session types	S	$::=$	$X\ \mid\ \mu X.S\ \mid\ \&\{m_i\colon S_i\}_{i\in I}\ \mid\ \oplus\{E.l_i\colon S_i\}_{i\in I}$
Channel session types	Σ	$::=$	$X\ \mid\ \mu X.\Sigma\ \mid\ \&_c\{E.l_i : \Sigma_i\}_{i\in I}\ \mid\ \oplus_c\{E.l_i : \Sigma_i\}_{i\in I}\ \mid$
			$?[B]\,.\,\Sigma\ \mid\ ![B]\,.\,\Sigma$

The only object reference o available to the programmer is this. The keywords send, receive, request and accept are method names m.

Fig. 5. Programmer's syntax.

3 A Core Distributed OO Language with Session Types

We now present a formal syntax, operational semantics and type system for a core language containing the essential features of the examples in Section 2, and state type safety results. 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 5) 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.

In order to simplify the presentation of the type system and the proof of type safety, the formal language requires every method definition to be annotated with pre- and post-conditions, expressed as a requirement (req) and a guarantee (ens, for "ensures") on the type of the object on which it is called. These annotations are in the style of the Fugue system [11] but stated in terms of session types. The typechecking algorithm presented in Section 4 infers the pre- and post-conditions.

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).

$$
\begin{array}{rrcl}
\text{Values} & v & ::= & \ldots \mid c^{+} \mid c^{-} \mid n \\
\text{Method dec} & M & ::= & \mathsf{req}\ T\ \mathsf{ens}\ T\ \mathsf{for}\ T\ m(T\ x)\ \{e\} \\
\text{Field types} & F & ::= & \vec{T}\,\vec{f} \\
\text{Runtime types} & T & ::= & \ldots \mid C[S;F] \mid T\ \mathsf{link}\ r \mid \langle E.l_i : T_i\rangle_{i\in I} \\
\text{Field values} & V & ::= & \vec{f} = \vec{v} \\
\text{Heaps} & h & ::= & \varepsilon \mid h :: o = C[V] \\
\text{States} & s & ::= & h;\ e \\
\text{Contexts} & \mathcal{E} & ::= & [\_] \mid o.f\ =\ \mathcal{E} \mid \mathcal{E}; e \mid r.m(\mathcal{E}) \mid \mathsf{spawn}\ C.m(\mathcal{E}) \\
\text{Configurations} & K & ::= & \langle h;\ e\rangle \mid K \,||\, K \mid (\nu c)\, K
\end{array}
$$

Fig. 6. Syntax used only in the type system and semantics.

Class, enum and method declarations, including the forms for inheritance, have been illustrated by the examples. We write *session*(C), *fields*(C), *methods*(C) to access the components of a class, and *constants*(E) for the set of values in an enum. 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. We assume that class and enum identifiers in a sequence of declarations $\vec{D}$ are all distinct, and that method names, field names and labels in $\vec{M}$, $\vec{f}$ and $\{l_i\}_{i\in I}$ are distinct as well.

There are some restrictions on the syntax of expressions. Field access and assignment are only available for a field of an object reference, not a field of an arbitrary expression; moreover, the only object reference available to the programmer is this. Method call is only available on object references and parameters, not on arbitrary expressions. All fields are private: this.$f.g$ and this.$f.g.m()$ are not syntactically correct. The examples in Section 2 omit this as the prefix to all field accesses, but they can easily be inserted by the compiler. The primitive operations involving channels, namely **send**, **receive**, **request** and **accept**, are treated as method names in the syntax.

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 Null has the single value null. Messages on channels must have non-object types, as can be seen from the presence of B in the definition of channel session types; we do not send objects as messages. This is not really a fundamental restriction, but sending an object as a message would also require some form of serialization, which we have avoided formalizing.

Session types (both class and channel ones) have been discussed in relation to the example. Session type end abbreviates both $\&\{\}$ and $\&_c\{\}$. In $\oplus\{E.l_i\colon S_i\}_{i\in I}$, the identifier E is an enum with values $\{l_i\}_{i\in I}$. In the session type of a class declaration, the top-level constructor, apart from recursion, must be $\&$. The core language does not include named session types, nor **typedef** or the **session** and **where** clauses from the examples; we just work with recursive session type ex-

$$h;\ \mathsf{new}\ C() \longrightarrow h :: o = C[\mathit{fields}(C) = \vec{\mathsf{null}}];\ o \qquad \text{(R-New)}$$

$$\frac{h(o).f = v}{h;\ o.f \longrightarrow h\{o.f \mapsto \mathsf{null}\};\ v} \qquad h;\ o.f\ =\ v \longrightarrow h\{o.f \mapsto v\};\ \mathsf{null}$$

$$\text{(R-Field,R-Assign)}$$

$$\frac{(\_\ m(\_\ x)\ \{e\}) \in \mathit{methods}(h(o).\mathsf{class})}{h;\ o.m(v) \longrightarrow h;\ e\{o/\mathsf{this}\}\{v/x\}} \qquad \text{(R-DirectCall)}$$

$$\frac{h(o).f = o' \qquad (\_\ m(\_\ x)\ \{e\}) \in \mathit{methods}(h(o').\mathsf{class})}{h;\ o.f.m(v) \longrightarrow h;\ e\{o'/\mathsf{this}\}\{v/x\}} \qquad \text{(R-IndirCall)}$$

$$h;\ \mathsf{switch}\ (E.l_j)\ \{\mathsf{case}\ l_i \colon e_i\}_{i \in I} \longrightarrow h;\ e_j \quad (j \in I) \qquad \text{(R-Switch)}$$

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

Fig. 7. Reduction rules for states.

pressions 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 [37, Chapter 21] and regard $\mu X.S$ and $S\{(\mu X.S)/X\}$ as equivalent, using them interchangeably in any mathematical context.

Figure 6 defines additional syntax needed for the formal system. It is not available to the programmer. In the values, n is a name on which **request** and **accept** can be called, and c^+ and c^- are the two endpoints of a communication channel, on which **send** and **receive** can be called. In the internal types, $C[S; F]$ is a form of object type that includes field types, and $\langle E.l_i \colon T_i \rangle_{i \in I}$ is a variant type, indexed by the values of an enumerated type E. In contrast to variant types in functional languages, values are not tagged; instead the tag is stored in a field of type E link r, where r has variant type. These types are used in the type system but do not appear in top-level programs.

Field values, heaps, states and configurations are used to define the operational semantics. A heap is a function and an entry in the heap maps an object reference o to an object: $o = C[\vec{f} = \vec{v}]$, where C is the class and the fields $\vec{f}$ have values $\vec{v}$. $\mathcal{E}$ are evaluation contexts in the style of Wright and Felleisen [47]. A configuration is a parallel collection of (heap, expression) pairs, representing separate locations in a distributed system. In $(\nu c)K$, νc binds endpoints c^+ and c^- of channel c.

Operational Semantics. We define the operational semantics in two stages. Figure 7 defines reductions of states $h; e$ consisting of a heap and an expression. In Figure 8, rule R-Thread lifts these reductions, in contexts, to basic reductions on configurations; rules R-Par, R-NewChan and R-Struct define general reductions on configurations. Structural congruence, used in R-Struct, is the smallest congruence (with respect to parallel and ν) closed under rules E-Comm, E-Assoc and E-Scope. Rules R-Init, R-Com and R-Spawn define interactions between components and creation of new components. All of the

$$K_1 \parallel K_2 \;\equiv\; K_2 \parallel K_1 \qquad K_1 \parallel (K_2 \parallel K_3) \;\equiv\; (K_1 \parallel K_2) \parallel K_3 \quad \text{(E-Comm,E-Assoc)}$$

$$K_1 \parallel (\nu c)K_2 \;\equiv\; (\nu c)(K_1 \parallel K_2) \;\text{ if } c^+, c^- \text{ not free in } K_q \qquad \text{(E-Scope)}$$

$$\frac{h;\; e \longrightarrow h';\; e'}{\langle h;\; \mathcal{E}[e]\rangle \longrightarrow \langle h';\; \mathcal{E}[e']\rangle} \qquad \frac{K \longrightarrow K'}{K \,||\, K'' \longrightarrow K' \,||\, K''} \qquad \text{(R-Thread,R-Par)}$$

$$\frac{K \longrightarrow K'}{(\nu c)\, K \longrightarrow (\nu c)\, K'} \qquad \frac{K \equiv K' \quad K' \longrightarrow K'' \quad K'' \equiv K'''}{K \longrightarrow K'''}$$
$$\text{(R-NewChan,R-Struct)}$$

$$\frac{r = n \text{ or } \left\{\begin{smallmatrix} r=o.f \text{ and} \\ h(o).f=n \end{smallmatrix}\right. \qquad r' = n \text{ or } \left\{\begin{smallmatrix} r'=o'.f' \text{ and} \\ h'(o').f'=n \end{smallmatrix}\right. \qquad (c \text{ fresh})}{\langle h;\; \mathcal{E}[r.\mathsf{accept}()]\rangle \,||\, \langle h';\; \mathcal{E}'[r'.\mathsf{request}()]\rangle \longrightarrow (\nu c)\,(\langle h;\; \mathcal{E}[c^+]\rangle \,||\, \langle h';\; \mathcal{E}'[c^-]\rangle)} \quad \text{(R-Init)}$$

$$\frac{r = c^p \text{ or } \left\{\begin{smallmatrix} r=o.f \text{ and} \\ h(o).f=c^p \end{smallmatrix}\right. \qquad r' = c^{\overline{p}} \text{ or } \left\{\begin{smallmatrix} r'=o'.f' \text{ and} \\ h'(o').f'=c^{\overline{p}} \end{smallmatrix}\right.}{\langle h;\; \mathcal{E}[r.\mathsf{send}(v)]\rangle \,||\, \langle h';\; \mathcal{E}'[r'.\mathsf{receive}()]\rangle \longrightarrow \langle h;\; \mathcal{E}[\mathsf{null}]\rangle \,||\, \langle h';\; \mathcal{E}'[v]\rangle} \quad \text{(R-Com)}$$

$$\langle h;\; \mathcal{E}[\mathsf{spawn}\ C.m(v)]\rangle \longrightarrow \langle h;\; \mathcal{E}[\mathsf{null}]\rangle \,||\, \langle o = C[\mathit{fields}(C) = \vec{\mathsf{null}}];\; o.m(v)\rangle \;\; (o \text{ fresh})$$
$$\text{(R-Spawn)}$$

Fig. 8. Structural congruence and reduction rules for configurations.

rules are implicitly parameterized by $\vec{D}$, the list of declarations constituting the program. When we say an identifier should be fresh, we mean distinct from all others used in any mathematical context in which that identifier appears.

The operation $h :: o = C[\vec{f} = \vec{v}]$ denotes adding an entry to the heap h, and it is only defined if o is not in h and all object references in $\vec{v}$ are in the domain of h. If $h(o) = C[\vec{f} = \vec{v}]$ then $h(o).\mathsf{class}$ means C and $h(o).f_i$ means v_i. If $h(o)$ is defined (this is an implicit hypothesis) then the notation $h\{o.f \mapsto v\}$ means the heap obtained by changing the value of field f in object o to v.

R-New creates a new object in the heap, with null fields. R-Field extracts the value of a field from an object. 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.

There are two rules for method call. R-DirectCall is for calls directly on an object reference, which arise from calls on this and calls on method parameters. R-IndirCall is for calls on fields of objects. In both cases, appropriate substitutions are made for this and the formal parameter. R-Switch is standard. R-Seq discards the result of the first part of a sequential composition.

The rules in Figure 8 are as expected for a distributed language with point-to-point channels. R-Init and R-Com allow the operations to be called both on names/channels and on fields that contain names/channels.

To complete the definition of the semantics we need to define the initial state. We assume that a class C and main method m are specified. For a given class C and main method m, one would expect an initial state of the form $\emptyset;\, \mathsf{new}\ C().m(\ldots)$. Because we cannot call methods on arbitrary expressions, the initial state is actually $o = C[\mathit{fields}(C) = \vec{\mathsf{null}}];\; o.m(\ldots)$.

$$T <: T \qquad \frac{T <: T'' \quad T'' <: T'}{T <: T'} \qquad \text{(S-Id,S-Trans)}$$

$$\frac{\mathsf{enum}\ E\ \mathsf{restricts}\ E'\ L \in \vec{D} \quad L \subseteq \mathit{constants}(E')}{E <: E'} \qquad \text{(S-Enum)}$$

$$\frac{T <: T'}{T\ \mathsf{link}\ r <: T'\ \mathsf{link}\ r} \qquad \frac{I \subseteq J \quad E <: E' \quad T_i <: T'_i \quad (\forall i \in I)}{\langle E.l_i : T_i\rangle_{i \in I} <: \langle E'.l_j : T'_i\rangle_{j \in J}} \qquad \text{(S-Link,S-Variant)}$$

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ C'\ \{\_;\_;\_\} \in \vec{D} \qquad S <: S' \qquad F <: F'}{C[S;F] <: C'[S';F']} \qquad \text{(S-Class)}$$

$$\frac{T <: T' \quad W' <: W}{T\ m(Wx)\ \{\_\} <: T'\ m(W'x)\ \{\_\}} \qquad \text{(S-Method)}$$

$$\frac{T <: T' \quad U <: U' \quad V <: V' \quad W' <: W}{\mathsf{req}\ T\ \mathsf{ens}\ U\ \mathsf{for}\ V\ m(Wx)\ \{\_\} <: \mathsf{req}\ T'\ \mathsf{ens}\ U'\ \mathsf{for}\ V'\ m(W'x)\ \{\_\}} \qquad \text{(S-AnnotMethod)}$$

Fig. 9. Subtyping rules for types and method signatures

Subtyping. The foundation for inheritance and subtyping is the definition of subtyping between class session types. Let $\mathcal{S}$ be the set of class session types. Define $\mathit{unfold}(\mu X.S) = \mathit{unfold}(S\{(\mu X.S)/X\})$, and $\mathit{unfold}(S) = S$ for non-recursive session types S; contractivity guarantees that this definition terminates.

Definition 1. *A relation $R \subseteq \mathcal{S} \times \mathcal{S}$ is a* class session type simulation *if $(S, S') \in R$ implies the following conditions.*

1. *If $\mathit{unfold}(S) = \&\{m_i : S_i\}_{i \in I}$ then $\mathit{unfold}(S') = \&\{m_j : S'_j\}_{j \in J}$, $J \subseteq I$ and $\forall j \in J.(S_j, S'_j) \in R$.*
2. *If $\mathit{unfold}(S) = \oplus\{E.l_i : S_i\}_{i \in I}$ then $\mathit{unfold}(S') = \oplus\{E'.l_j : S'_j\}_{j \in J}$, $\forall i \in I.(S_i, S'_i) \in R$ and $\mathit{constants}(E) \subseteq \mathit{constants}(E')$.*

The subtyping relation *on class session types is defined by $S <: S'$ if there exists a class session type simulation R such that $(S, S') \in R$.*

The direction of subtyping is opposite to that defined for channel session types in [19], because we make a choice by selecting a method from a session of & type instead of by sending a label on a channel of $\oplus$ type. However, in both cases, the type allowing a choice to be made has contravariant subtyping in the set of choices. This reversal of the subtyping relation for session types also occurs in [6]. Further details, including the proof that subtyping is reflexive and transitive and an algorithm for checking subtyping, can be adapted from [19].

Figure 9 defines subtyping between types of our language. The relation is as expected for object types viewed as records of fields, with the addition of subtyping between the session types. We could also include subtyping on channel session types, using the standard definitions [19], and the more refined treatment of subtyping on access types $\langle \Sigma \rangle$ from [21], but we have omitted them for simplicity.

It turns out that both the **requires** and **ensures** types behave covariantly. For **ensures** this is because the type is really part of the result type of the method, describing the implicitly returned **this**. For **requires** it is because the type is the true type of the object on which the method is called.

Type System. The type system is defined by the rules in Figures 10, 11 and 12. 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 channels or non-uniform objects). We regard an environment Γ as a function from object references o to object types $C[S; \vec{T}\,\vec{f}]$ and names to message types. If $\Gamma(o) = C[S; \vec{T}\,\vec{f}]$ then $\Gamma(o).f_i$ means T_i. If r is a value reference such that $\Gamma(r)$ is defined (i.e. either $r = o$ and $o \in \mathrm{dom}(\Gamma)$ or $r = o.f$ and $o \in \mathrm{dom}(\Gamma)$ and $\Gamma(o)$ has field f) then the notation $\Gamma\{r \mapsto T\}$ means the environment obtained by changing the type of o or $o.f$, as appropriate, to T.

First consider Figure 10, which defines typing of expressions. T-Chan and T-LinVar take account of linearity by removing identifiers from the environment. T-Field 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. The restriction on variant types is to avoid invalidating link types. T-New types a new object, giving it the initial session type from the class declaration and giving all the fields type Null.

T-Accept and T-Request type the creation of channels with dual session types. T-Send, T-Receive, T-Select and T-Branch type communication operations according to the session type of the channel. Note that in T-Select, the label $E.l_j$ must be a literal value.

T-Call requires an environment in which method $r.m$ is available. In the signature of m, the req type must match the type of $o.f$, the ens type gives the final type of $o.f$, and the result type gives the type of the expression as usual. The rule covers two cases, depending on whether the method returns an object or an enumerated value. In the latter case, which corresponds to a variant type in the ens of the method, the expression acquires the type T' link r, indicating that r has a variant type that will be resolved by a switch on the result of the method. The same mechanism is used by T-Branch. The other rule for method call, T-SelfCall, does not check the req annotation; self-calls do not change the session type of the object.

T-Switch types a switch on $o.f$, whose type must have a link to a field with a variant type. All branches must have the same final environment Γ'', so that it is a consistent final environment for the switch expression.

Now consider Figure 11. The most interesting rules are T-Meth and T-MethVar, which check that a method body has the effect specified by the req and ens declarations. There are two rules because the typing of a method has different forms depending on whether or not the class session type is $\oplus$. If it is, then the method must produce a variant type for this.

$$\Gamma \triangleright \mathsf{null} : \mathsf{Null} \triangleleft \Gamma \qquad \frac{T \text{ is not a variant type}}{\Gamma, o : T \triangleright o : T \triangleleft \Gamma} \qquad \text{(T-NULL, T-REF)}$$

$$\frac{T \text{ is not a variant type}}{\Gamma, c^p : T \triangleright c^p : T \triangleleft \Gamma} \qquad \frac{l \in \mathit{constants}(E)}{\Gamma \triangleright E.l : E \triangleleft \Gamma} \qquad \text{(T-CHAN, T-CONST)}$$

$$\frac{T = \mathsf{Null} \text{ or } E \text{ or } \langle \Sigma \rangle}{\Gamma, x : T \triangleright x : T \triangleleft \Gamma, x : T} \qquad \frac{T = C[\ldots] \text{ or } \Sigma}{\Gamma, x : T \triangleright x : T \triangleleft \Gamma} \qquad \text{(T-VAR, T-LINVAR)}$$

$$\Gamma, n : T \triangleright n : T \triangleleft \Gamma, n : T \qquad \text{(T-NAME)}$$

$$\frac{j \in I}{\Gamma, o : C[\oplus\{E.l_i : S_i\}_{i \in I}; F_j] \triangleright E.l_j : E \text{ link } o \triangleleft \Gamma, o : \langle E.l_i : C[S_i; F_i] \rangle_{i \in I}} \qquad \text{(T-INJ)}$$

$$\frac{\Gamma(o).f = T \quad T \text{ is not a variant type}}{\Gamma \triangleright o.f : T \triangleleft \Gamma\{o.f \mapsto \mathsf{Null}\}} \qquad \frac{\Gamma \triangleright e : T \triangleleft \Gamma' \quad \Gamma'(o).f \text{ is not a variant type}}{\Gamma \triangleright o.f = e : \mathsf{Null} \triangleleft \Gamma'\{o.f \mapsto T\}}$$

(T-FIELD,T-ASSIGN)

$$\Gamma \triangleright \mathsf{new}\ C() : C[\mathit{session}(C); \vec{\mathsf{Null}}\,\mathit{fields}(C)] \triangleleft \Gamma \qquad \text{(T-NEW)}$$

$$\frac{\Gamma(r) = \langle \Sigma \rangle}{\Gamma \triangleright r.\mathsf{accept}() : \Sigma \triangleleft \Gamma} \qquad \text{(T-ACCEPT)}$$

$$\frac{\Gamma(r) = \langle \Sigma \rangle}{\Gamma \triangleright r.\mathsf{request}() : \overline{\Sigma} \triangleleft \Gamma} \qquad \text{(T-REQUEST)}$$

$$\frac{\Gamma \triangleright e : T \triangleleft \Gamma' \qquad T'\ m(T\ x)\ \{\_\} <: M \qquad M \in \mathit{methods}(\Gamma'(o).\mathsf{class})}{\Gamma \triangleright o.m(e) : T' \triangleleft \Gamma'} \qquad \text{(T-SELFCALL)}$$

$$\frac{\begin{array}{c} T_1 = C[\ldots] \quad \mathsf{req}\ T_1\ \mathsf{ens}\ T_2\ \mathsf{for}\ T'\ m(T\ x)\ \{\_\} <: M \quad M \in \mathit{methods}(C) \\ r \neq \mathsf{this} \quad \Gamma \triangleright e : T \triangleleft \Gamma' \quad \Gamma'(r) = T_1 \quad T'' = \begin{cases} E \text{ link } r & \text{if } T' = E \text{ link this} \\ T' & \text{otherwise} \end{cases} \end{array}}{\Gamma \triangleright r.m(e) : T'' \triangleleft \Gamma'\{r \mapsto T_2\}} \qquad \text{(T-CALL)}$$

$$\frac{\Gamma \triangleright e : T \triangleleft \Gamma' \quad \Gamma'(r) = {!}[T]\,.\,\Sigma}{\Gamma \triangleright r.\mathsf{send}(e) : \mathsf{Null} \triangleleft \Gamma'\{r \mapsto \Sigma\}} \qquad \frac{\Gamma(r) = {?}[T]\,.\,\Sigma}{\Gamma \triangleright r.\mathsf{receive}() : T \triangleleft \Gamma\{r \mapsto \Sigma\}}$$

(T-SEND,T-RECEIVE)

$$\frac{\Gamma(r) = \oplus_c \{E.l_i : \Sigma_i\}_{i \in I} \quad j \in I}{\Gamma \triangleright r.\mathsf{send}(E.l_j) : \mathsf{Null} \triangleleft \Gamma\{r \mapsto \Sigma_j\}} \qquad \text{(T-SELECT)}$$

$$\frac{\Gamma(r) = \&_c \{E.l_i : \Sigma_i\}_{i \in I}}{\Gamma \triangleright r.\mathsf{receive}() : E \text{ link } r \triangleleft \Gamma\{r \mapsto \langle E.l_i : \Sigma_i \rangle_{i \in I}\}} \qquad \text{(T-BRANCH)}$$

$$\frac{\begin{array}{c} \Gamma \triangleright e : E \text{ link } r \triangleleft \Gamma' \quad \Gamma'(r) = \langle E.l_i : T_i \rangle_{i \in I} \\ \forall i \in I.(l_i \in \mathit{constants}(E) \quad \Gamma'\{r \mapsto T_i\} \triangleright e_i : T \triangleleft \Gamma'') \end{array}}{\Gamma \triangleright \mathsf{switch}\ (e)\ \{\mathsf{case}\ l_i : e_i\}_{i \in I} : T \triangleleft \Gamma''} \qquad \text{(T-SWITCH)}$$

$$\frac{\Gamma \triangleright e : T \triangleleft \Gamma'' \quad \Gamma'' \triangleright e' : T' \triangleleft \Gamma' \quad T \neq E \text{ link } r}{\Gamma \triangleright e; e' : T' \triangleleft \Gamma'} \qquad \frac{\Gamma \triangleright e : T \triangleleft \Gamma' \qquad T <: U}{\Gamma \triangleright e : U \triangleleft \Gamma'}$$

(T-SEQ, T-SUB)

$$\frac{\begin{array}{c} (\mathsf{req}\ T\ \mathsf{ens}\ \_\ \mathsf{for}\ \_\ m(T\ x)\ \{\_\}) \in \mathit{methods}(C) \quad T \text{ is not an object type} \\ T = C[\mathit{session}(C); \vec{\mathsf{Null}}\,\mathit{fields}(C)] \quad \Gamma \triangleright e : T \triangleleft \Gamma' \end{array}}{\Gamma \triangleright \mathsf{spawn}\ C.m(e) : \mathsf{Null} \triangleleft \Gamma'} \qquad \text{(T-SPAWN)}$$

Fig. 10. Typing rules for expressions

$$\frac{\begin{array}{c} x : T, \mathsf{this} : C[S_k; F] \triangleright e : T' \triangleleft \Gamma', \mathsf{this} : C[S_k; F'] \\ S_k \neq \oplus\{\ldots\} \quad k \in I \quad \Gamma' = \begin{cases} \emptyset \text{ if } T \text{ is linear} \\ x{:}T \text{ otherwise} \end{cases} \end{array}}{\vdash \mathsf{req}\ C[\&\{m_i\colon S_i\}_{i\in I}; F]\ \mathsf{ens}\ C[S_k; F']\ \mathsf{for}\ T'\ m_k(T\ x)\ \{e\}} \quad \text{(T-METH)}$$

$$\frac{\begin{array}{c} S_k = \oplus\{E.l_j\colon S'_j\}_{j\in J} \quad k \in I \quad \Gamma' = \begin{cases} \emptyset \text{ if } T \text{ is linear} \\ x{:}T \text{ otherwise} \end{cases} \\ x : T, \mathsf{this} : C[S_k; F] \triangleright e : E\ \mathsf{link\ this} \triangleleft \Gamma', \mathsf{this} : \langle E.l_j : C[S'_j; F_j]\rangle_{j\in J} \end{array}}{\vdash \mathsf{req}\ C[\&\{m_i\colon S_i\}_{i\in I}; F]\ \mathsf{ens}\ \langle E.l_j : C[S'_j; F_j]\rangle_{j\in J}\ \mathsf{for}\ E\ m_k(T\ x)\ \{e\}} \quad \text{(T-METHVAR)}$$

$$\frac{\begin{array}{c} \forall i. \vdash M_i \quad S <: \mathit{session}(C) \\ \forall M' \in \mathit{methods}(C'), M \in \vec{M}.(M <: M' \ \wedge\ \mathit{name}(M) = \mathit{name}(M')) \end{array}}{\vdash \mathsf{class}\ C\ \mathsf{extends}\ C'\ \{S; \vec{f}; \vec{M}\}} \quad \text{(T-EXTENDS)}$$

$$\frac{\vdash M_i \quad (\forall i)}{\vdash \mathsf{class}\ \_\ \{\_; \_; \vec{M}\}} \qquad \vdash \mathsf{enum}\ E\ L \qquad \frac{E <: E'}{\vdash \mathsf{enum}\ E\ \mathsf{restricts}\ E'\ \_} \qquad \frac{\vdash D_i \quad (\forall i)}{\vdash \vec{D}}$$
$$\text{(T-CLASS,T-ENUM,T-RESTRICTS,T-PROG)}$$

Fig. 11. Typing rules for programs

$$\emptyset \vdash \emptyset \qquad \frac{\Gamma, o\colon T \vdash h}{\Gamma \vdash h} \qquad \frac{\Gamma \vdash h \quad \Gamma \triangleright \vec{v} : \vec{T} \triangleleft \Gamma' \quad \mathit{fields}(C) = \vec{f}}{\Gamma', o : C[S'; \vec{T}\ \vec{f}] \vdash h :: o = C[\vec{f} = \vec{v}]}$$
$$\text{(T-HEMPTY,T-HWEAK,T-HADD)}$$

$$\frac{\Gamma \vdash h \quad \Gamma(r) = E\ \mathsf{link}\ o \quad h(r') = o}{\Gamma\{r \mapsto E\ \mathsf{link}\ r'\} \vdash h} \qquad \frac{\vdash \vec{D} \quad \Gamma \vdash h \quad \Gamma \triangleright e : T \triangleleft \Gamma'}{\Gamma \triangleright h;\ e : T \triangleleft \Gamma'}$$
$$\text{(T-HLINK, T-STATE)}$$

$$\frac{\begin{array}{c} \Gamma, o : T_0 \vdash h :: o = C[f = o', \ldots] \quad T_0.f = T_1 \quad \Gamma, o' : T_1 \triangleright h;\ e : T \triangleleft \Gamma', o' : T_2 \\ T' = \begin{cases} E\ \mathsf{link}\ o.f \text{ if } T{=}E\ \mathsf{link}\ o' \\ T \text{ otherwise} \end{cases} \end{array}}{\Gamma, o : T_0 \triangleright h :: o = C[f = o', \ldots];\ e : T' \triangleleft \Gamma', o : T_0\{f \mapsto T_2\}}$$
$$\text{(T-EXPRSTATE)}$$

$$\frac{\Gamma \triangleright h;\ e : \_ \triangleleft \_}{\Gamma \vdash \langle h;\ e\rangle} \qquad \frac{\Gamma \vdash K \quad \Gamma' \vdash K'}{\Gamma + \Gamma' \vdash K \,||\, K'} \qquad \frac{\Gamma, c^+ : \Sigma, c^- : \overline{\Sigma} \vdash K}{\Gamma \vdash (\nu c)\, K}$$
$$\text{(T-THREAD, T-PAR, T-NEWCHAN)}$$

$\Gamma + \Gamma'$ is defined if and only if the only identifiers Γ and Γ' have in common are names n and those names have the same type in both environments. In that case, $\Gamma + \Gamma' = \Gamma \cup \Gamma'$.

Fig. 12. Typing rules for heaps, states and configurations.

Figure 12 defines rules for typing heaps, states and configurations. The typing of a heap, $\Gamma \vdash h$, means that Γ gives types to the usable objects in h. Because of linearity, Γ only contains types for top-level objects, i.e. those that are not stored in fields of other objects. Weakening of the heap typing (T-HWEAK) is needed in order to prove type preservation, because assignment can discard an object. T-ExprState and T-LinkState are not used for top-level programs, but are needed in the proof of type preservation to type the state resulting from reduction of a method call.

The typing of a top-level system, to which our type safety results apply, has the form $\Gamma \vdash K_1 \parallel \ldots \parallel K_n$ where Γ gives global typings for service names.

Results. We have a type preservation theorem of the usual kind; the change from Γ to Γ' reflects changes in (class or channel) session types. The proof is by induction on the derivation of the reduction.

Theorem 1. *If $\Gamma \vdash K$ and $K \longrightarrow K'$ then there exists Γ' such that $\Gamma' \vdash K'$.*

To obtain a type safety theorem we define *traces* and *call traces*, enabling us to extract more information from the type preservation proof.

Definition 2 (Traces). *A* trace *on a channel endpoint c^p is a sequence $a_1 a_2 \ldots$ where each a_i is either $v_i\!\uparrow$ or $v_i\!\downarrow$ and v_i is either a primitive value or an enumeration label. A* call trace *on an object o is a sequence $m_1 \alpha_1 m_2 \alpha_2 \ldots$ where each m_i is a method name and each α_i is either an enumeration label or nothing.*

The operational semantics defines a trace for every channel endpoint and a call trace for every object. Self-calls are excluded from call traces.

A channel session type defines a set of traces, in which items $v_i\!\uparrow$ come from send and select types and items $v_i\!\downarrow$ come from receive and branch types. A class session type defines a set of call traces.

Theorem 2 (Type Safety). *When executing a typed program, the trace of every channel endpoint is one of the traces of its initial channel session type and the call trace of every object is one of the traces of its initial class session type.*

Given a mapping from channel endpoints and objects to traces and call traces, the safety property is an invariant of reduction, and type safety becomes a corollary of type preservation.

4 Typechecking Algorithm

Figure 13 defines a typechecking algorithm. It is used in two steps. First, for each class C with declared session type S, $\mathcal{P}_C(S, \emptyset)$ is called. This returns annotations for the methods of C, in the form $\mathsf{req}\ C[\&\{m_i \colon S_i\}_{i \in I}]\ \mathsf{ens}\ C[S_j]\ \mathsf{for}\ T\ m_j()\{e\}$ with $j \in I$. Algorithm $\mathcal{P}$ is very simple, and just translates the session type of a class into explicit pre- and post-conditions for its methods. A particular method can receive several different annotations, giving a useful form of overloading [45]. In this case, the req type should be used to disambiguate method calls.

The second step is to call $\mathcal{A}_C(F, S, \emptyset)$ for each class C, where S is the declared session type of C and F is the initial field typing (with all fields having type Null) of C. Algorithm $\mathcal{A}$ has two purposes. (1) It calls algorithm $\mathcal{B}$ to typecheck the method bodies of C, in the order corresponding to S. While typechecking,

Algorithm $\mathcal{P}$

$\mathcal{P}_C(\&\{m_i\colon S_i\}_{i\in I}, \Delta) = \{$ req $\&\{m_i\colon S_i\}_{i\in I}$ ens $C[S_i]$ for $T\ m_i()\ \{e\}$ $|$
$T\ m_i()\{e\} \in \mathit{methods}(C), i \in I\} \cup \bigcup_{j\in I} \mathcal{P}_C(S_j, c\Delta)$
$\mathcal{P}_C(\oplus\{E.l_i\colon S_i\}_{i\in I}, \Delta) = \bigcup_{i\in I} \mathcal{P}_C(S_i, \Delta)$
$\mathcal{P}_C(\mu X.S, \Delta) = \text{if } \mu X.S \in \Delta \text{ then } \emptyset \text{ else } \mathcal{P}_C(\mathit{unfold}(S), \Delta \cup \{\mu X.S\})$

Algorithm $\mathcal{A}$

$\mathcal{A}_C(\&\{m_i\colon S_i\}_{i\in I}, F, \Delta) = \bigcup_{j\in I}\{\{\mathsf{req}\ C[\&\{m_i\colon S_i\}_{i\in I}; F]\ \mathsf{ens}\ C[S_j; F_j]\ \mathsf{for}\ T_j\ m_j()\ \{e_j\}\}\cup$
$\mathcal{A}_C(S_j, F_j, \Delta) \mid T_j\ m_j()\ \{e_j\} \in \mathit{methods}(C), (F_j, T_j) = \mathcal{B}^{S_j}_C(e_j, F, \emptyset), j \in I\}$
$\mathcal{A}_C(\oplus\{E.l_i\colon S_i\}_{i\in I}, F, \Delta) = \bigcup_{j\in I} \mathcal{A}_C(S_j, F, \Delta)$
$\mathcal{A}_C(\mu X.S, F, \Delta) = \text{if } \mu X.S \in \text{dom}(\Delta) \text{ then } \Delta(\mu X.S) \text{ else } \mathcal{A}_C(\mathit{unfold}(S), F, \Delta \cup \{\mu X.S \mapsto F\})$

Algorithm $\mathcal{B}$

$\mathcal{B}^S_C(\mathsf{null}, F, \_) = (F, \mathsf{Null})$
$\mathcal{B}^S_C(E.l, F, \_) = (\langle E.l\colon C[S; F]\rangle, E)$
$\mathcal{B}^S_C(\mathsf{new}\ C(), F, \_) = (F, C[\mathit{session}(C)])$
$\mathcal{B}^S_C(\mathsf{this}.f, F, \_) = (F + f : \mathsf{Null}, F(f))$
$\mathcal{B}^S_C(\mathsf{this}.f\ =\ e, F, \Delta) = (F' + \mathsf{this}.f : T, \mathsf{Null}) \text{ where } (F', T) = \mathcal{B}^S_C(e, F, \Delta)$
$\mathcal{B}^S_C(\mathsf{this}.f.m_j(), F, \_) = (F + f : \mathsf{Null}, T') \text{ where } F(f) = C'[\&\{m_i\colon S_i\}_{i\in I}] \text{ and } j \in I \text{ and}$
$\mathsf{req}\ C[\&\{m_i\colon S_i\}_{i\in I}]\ \mathsf{ens}\ \_\ \mathsf{for}\ T\ m_j()\ \{\_\} \in \mathit{methods}(\mathcal{P}_{C'}(\mathit{session}(C'), \emptyset)) \text{ and}$
$T' = \text{if } S_j = \oplus\{\dots\} \text{ then } T\ \mathsf{link}\ \mathsf{this} \text{ else } T$
$\mathcal{B}^S_C(\mathsf{this}.m(), F, \Delta) = \text{if } m \in \text{dom}(\Delta) \text{ then } (\Delta(m), T) \text{ else } \mathcal{B}^S_C(e, F, \Delta \cup \{m \mapsto F\})$
$\text{where } T\ m()\{e\} \in \mathit{methods}(C)$
$\mathcal{B}^S_C(\mathsf{switch}\ (\mathsf{this}.f)\ \{\mathsf{case}\ l_i\colon e_i\}_{i\in I}, F, \Delta) = (\biguplus F_i, T)$
$\text{where } (F', E\ \mathsf{link}\ \mathsf{this}) = \mathcal{B}^S_C(e, F, \Delta) \text{ and } (F_i, T) = \mathcal{B}^S_C(e_i, F, \Delta) \text{ and } l_i \in \mathit{constants}(E)$
$\mathcal{B}^S_C((e; e'), F, \Delta) = \mathcal{B}^S_C(e', F', \Delta) \text{ where } (F', \_) = \mathcal{B}^S_C(e, F, \Delta)$
$\mathcal{B}^S_C(\mathsf{service}\ \Sigma, F, \_) = (\Sigma, F)$
$\mathcal{B}^S_C(\mathsf{this}.f.\mathsf{request}(), F, \_) = (F, \Sigma) \text{ where } F(f) = \langle\Sigma\rangle$
$\mathcal{B}^S_C(\mathsf{this}.f.\mathsf{accept}(), F, \_) = (F, \overline{\Sigma}) \text{ where } F(f) = \langle\Sigma\rangle$
$\mathcal{B}^S_C(\mathsf{this}.f.\mathsf{send}(e), F, \Delta) = (F' + f : \Sigma, \mathsf{Null}) \text{ where } (F', T) = \mathcal{B}^S_C(e, F, \Delta) \text{ and } F'(f) = !\,[T]\,.\Sigma$
$\mathcal{B}^S_C(\mathsf{this}.f.\mathsf{send}(E.l_j), F, \_) = (F + f : \Sigma_j, \mathsf{Null}) \text{ where } F(f) = \oplus_c\{E.l_i : \Sigma_i\}_{i\in I} \text{ and } j \in I$
$\mathcal{B}^S_C(\mathsf{this}.f.\mathsf{receive}(), F, \_) = (F + f : \Sigma, T) \text{ where } F(f) = ?\,[T]\,.\Sigma$
$\mathcal{B}^S_C(\mathsf{this}.f.\mathsf{receive}(), F, \_) = (F + f : \langle E.l_i\colon \Sigma_i\rangle_{i\in I}, E\ \mathsf{link}\ f) \text{ where } F(f) = \&_c\{E.l_i : \Sigma_i\}_{i\in I}$

Combining partial variants

$T \uplus T = T$ if T is not a variant typing
$\langle E.l_i\colon T_i\rangle_{i\in I} \uplus \langle E.m_j : T'_j\rangle_{j\in J} = \langle E.l_k : T''_k\rangle_{k\in I\cup J}$
where $\forall i \in I.(T''_i = T_i)$, $\forall j \in J.(T''_j = T'_j)$, and whenever $l_i = m_j$ we have $T_i = T'_j$

Fig. 13. Algorithm

the annotations calculated by algorithm $\mathcal{P}$ are used to check the effect of method calls. (2) It calculates a more comprehensive set of annotations for the methods of C, in the form req $C[\&\{m_i\colon S_i\}_{i\in I}; F]$ ens $C[S_j; F_j]$ for $T\ m_j()\ \{e\}$. These are used in the proof of type safety, to show that a typable program in the top-level language yields a typable program in the runtime language.

The definition of $\mathcal{B}$ follows the typing rules (Figure 10) except for one point: T-Inj means that the rules are not syntax-directed. To compensate, clause $E.l$ of $\mathcal{B}$ 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. 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.

For example, applying algorithm $\mathcal{P}$ to class BuyerAPI in Figure 1 produces the following annotated methods.

```
req BuyerAPI[Init] ens BuyerAPI[Shop] for Null init(<S> n);
req BuyerAPI[Shop] ens BuyerAPI[Shop] for Price price(Product p);
req BuyerAPI[Shop] ens BuyerAPI[Shop] for Result buy(Product p);
req BuyerAPI[Shop] ens BuyerAPI[Shop] for Null pay(Payement p);
req BuyerAPI[Shop] ens BuyerAPI[end] for Null stop ();
```

These are the "public" annotations, and are used when algorithm $\mathcal{A}$ is applied to class Buyer in Figure 2. For "internal" type checking we need the types of the fields as well. The more comprehensive annotations include information about the fields in BuyerAPI, but no information about the types of the fields within QuoteStore, for which we do not have the source code.

```
req BuyerAPI[Init; Null c, Null qs]
  ens BuyerAPI[Shop; S c; QuoteStore[Q] qs] for Null init(<S> n);
req BuyerAPI[Shop; S c, QuoteStore[Q] qs]
  ens BuyerAPI[Shop; S c; QuoteStore[Q] qs] for Price price(Product p);
...
req BuyerAPI[Shop; S c, QuoteStore[Q] qs]
  ens BuyerAPI[Shop; End c; QuoteStore[Q] qs] for Null stop();
```

We conclude this section with the main results of the algorithm. They are: 1) it always terminates; and 2) it produces only well-typed class declarations.

Theorem 3. *Let $\vec{D}$ be a program, i.e. a sequence of declarations.*

1. *For every* class $C\ \{S; \vec{f}; \vec{M}\}$ *or* class C extends $C'\ \{S; \vec{f}; \vec{M}\}$ *in* $\vec{D}$, $\mathcal{P}_C(S, \emptyset)$, *the call* $\mathcal{A}_C(\vec{\mathsf{Null} f}, S, \emptyset)$ *terminates.*
2. *For every* class $C\ \{S; \vec{f}; \vec{M}\}$ *or* class C extends $C'\ \{S; \vec{f}; \vec{M}\}$ *in* $\vec{D}$, *replace* $\vec{M}$ *by* $\mathcal{A}_C(\vec{\mathsf{Null} f}, S, \emptyset)$, *and let* $\vec{D}'$ *be the resulting declarations. Then* $\vdash \vec{D}'$.

The session type of a class has two interpretations. The first is as a limit on the allowed sequences of method calls, a kind of *safety* property, and this is always guaranteed by our type safety result. The second interpretation is that every sequence of method calls in the session type should be realizable in a typable program. Given a class definition C in the internal syntax, with explicit req and ens annotations, construct an expression e_C as follows:

1. The session type of C is a (possibly infinite) tree, branching at & and $\oplus$.
2. Make it into a finite tree by replacing some & nodes by end.
3. For each & node, remove all but one of the branches; obtain tree T.
4. e_C constructs an object of class C and contains a sequence of method calls and **switch** statements corresponding to the structure of T.

Typability of C in the internal system does not guarantee that e_C is typable, because it is possible for the req and ens clauses to contain spurious constraints such that the ens of one method does not match the req of the next method in the session type. But the typechecking algorithm, applied to a program in the programmer's syntax, produces definitions such that every e_C is typable.

5 Implementation

We have used the Polyglot [35] system to implement the ideas of this paper as a prototype extension to Java 5, which we call Bica. This includes type-checking method calls against the class session types of non-uniform objects, and type-checking communication operations on streams against their channel session types. The implementation also includes standard classes without session types, which are not linearly controlled. The definitions of class session types, and the **restricts** declaration of enumerated types, are implemented as Java annotations @session and @restricts rather than syntactic extensions. The semantics of the language is standard Java.

The implementation of Bica follows the Polyglot framework and is structured as a number of visitors which process session type declarations and implement the type-checking algorithms defined in Section 4. It is available from `http://gloss.di.fc.ul.pt/bica/`.

6 Related Work

Previous work on session types for object-oriented languages. Several recent papers by Dezani-Ciancaglini, Yoshida *et al.* [5, 12–14, 26, 32] have combined session types, as specifications of protocols on communication channels, with the object-oriented paradigm. A characteristic of all of this work 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. The most recent work in this line [5] unifies sessions and methods, and continues the idea that a session is a complete entity. Mostrous and Yoshida [32] add sessions to Abadi and Cardelli's object calculus.
Non-uniform concurrent objects / active objects. Another related line of research was started by Nierstrasz [34], 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 by several authors [4, 9, 39–41].The last two are the most relevant. Damiani *et al.* [9] 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. Caires [4] uses an approach based on spatial logic to give very fine-grained control of resources, and Militão [31] has implemented a prototype based on this idea. 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.

Cyclone, Vault, CQual, Fugue, Sing#. *Cyclone* [22],*Vault* [10, 16], and CQual [18] 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.

Vault is much closer to our system, allowing abstract states to be defined for resources, with pre- and post-conditions for each operation, and checking statically that operations occur in the correct sequence. It uses linear types to control aliasing, and uses the *adoption and focus* mechanism [16] to re-introduce aliasing in limited situations. *Fugue* [11, 17] extends similar ideas to an object-oriented language, and uses explicit pre- and post-conditions that are somewhat similar to our req/ens annotations. *CQual* expects users to annotate programs with type qualifiers; its type system, simpler and less expressive than the above, provides for type inference.

Sing# [15] 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 *contracts* which are analogous to our req and ens clauses. The published paper [15] does not discuss the relationship between channel contracts and non-uniform objects or typestates, and does not define a formal language.

The main novelties of our work are the integration of session-typed channels, the use of the session type of a class as a global specification, the dependency between the result of a method and the subsequent abstract state of the object, and the characterization of the subtyping relation. 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.

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* [23], Almeida's *balloon types* [1], Clarke *et al.*'s *ownership types* [8], Fähndrich and DeLine's *adoption and focus* [16], Östlund *et al.*'s Joe_3 [36] 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 [28] 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 [27] including a verification tool [29] in which to define type systems for analyzing various behavioural properties including sequences of resource uses [30]. In some of this work, types are themselves abstract processes, and therefore in some situations resemble our session types. Chaki *et al.* [7] 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.

7 Future Work

Shared classes. In the present formal language, all classes are linear. It is straightforward to add shared classes, whose objects do not have to be uniquely referenced; indeed, the implementation includes them. The behaviour of shared objects is largely orthogonal to that of linear objects, except for the condition that a shared object's fields cannot contain linear objects.

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.

Complete use of sessions. Some systems based on session types guarantee that sessions are completely used, finishing in state end. To achieve this property we only need to change the rules for assignment so that an incompletely-used object cannot be discarded, and adapt the type-checking algorithm to check that when an object has session type End all of its fields have session type End. This can be specified independently for each class.

Java-style interfaces. If class C **implements** interface I then we should have $\mathit{session}(C) <: \mathit{session}(I)$, interpreting the interface as a specification of minimum method availability.

Specifications involving several objects. Multi-party session types [3, 24] 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.

8 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. Correctness of the type system is expressed by a type safety theorem. 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.

Acknowledgements. Caldeira, Ravara, and Vasconcelos were partially supported by the EU IST proactive initiative FET-Global Computing (project Sensoria, IST–2005–16004), and by the Portuguese FCT (via SFRH/BSAB/757/2007, and project Space–Time–Types, POSC/EIA/55582/2004). Ravara was also partially supported by the UK EPSRC (EP/F037368/1 "Behavioural Types for Object-Oriented Languages"). Gay was partially supported by the Security and Quantum Information Group, Instituto de Telecomunicações, Portugal, and 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. Gesbert was supported by the UK EPSRC (EP/E065708/1).

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. E. Bonelli and A. Compagnoni. Multipoint session types for a distributed calculus. *TGC, Springer LNCS*, 4912:240–256, 2007.
4. L. Caires. Spatial-behavioral types for concurrency and resource control in distributed systems. *Theoretical Computer Science*, 402(2–3):120–141, 2008.
5. S. Capecchi, M. Coppo, M. Dezani-Ciancaglini, S. Drossopoulou, and E. Giachino. Amalgamating sessions and methods in object-oriented languages with generics. *Theoretical Computer Science*, 2008. To appear.
6. M. Carbone, K. Honda, and N. Yoshida. Structured global programming for communication behaviour. *ESOP, Springer LNCS*, 4421:2–17, 2007.
7. 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.
8. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. *OOPSLA, ACM SIGPLAN Notices*, 33(10):48–64, 1998.
9. 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.
10. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. *PLDI, ACM SIGPLAN Notices*, 36(5):59–69, 2001.
11. R. DeLine and M. Fähndrich. The Fugue protocol checker: is your software Baroque? Technical Report MSR-TR-2004-07, Microsoft Research, 2004.
12. M. Dezani-Ciancaglini, S. Drossopoulou, E. Giachino, and N. Yoshida. Bounded session types for object-oriented languages. *FMCO, LNCS*, 4709:207–245, 2007.
13. M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopolou. Session types for object-oriented languages. *ECOOP, Springer LNCS*, 4067:328–352, 2006.
14. M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopolou. A distributed object-oriented language with session types. *TGC, LNCS*, 3705:299–318, 2005.
15. 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.
16. M. Fähndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. *PLDI, ACM SIGPLAN Notices*, 37(5):13–24, 2002.
17. M. Fähndrich and R. DeLine. Typestates for objects. *ESOP, Springer LNCS*, 3086:465–490, 2004.
18. J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. *PLDI, ACM SIGPLAN Notices*, 37(5):1–12, 2002.
19. S. J. Gay and M. J. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
20. 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.
21. S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. `www.dcs.gla.ac.uk/~simon/publications/Lin-Async.pdf`. Submitted, 2008.
22. 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.

23. J. Hogg. Islands: aliasing protection in object-oriented languages. *OOPSLA, ACM SIGPLAN Notices*, 26(11):271–285, 1991.
24. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *POPL, ACM SIGPLAN Notices*, 43(1):273–284, 2008.
25. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. *ESOP, Springer LNCS*, 1381:122–138, 1998.
26. R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in Java. *ECOOP, Springer LNCS*, 5142:516–541, 2008.
27. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004.
28. A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Trans. on Programming Languages and Systems*, 27(2):264–313, 2005.
29. N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4–5):291–347, 2005.
30. N. Kobayashi, K. Suenaga, and L. Wischik. Resource usage analysis for the π-calculus. *Logical Methods in Computer Science*, 2(3:4):1–42, 2006.
31. F. Militão. Yak programming language. `ctp.di.fct.unl.pt/yak`, 2008.
32. D. Mostrous and N. Yoshida. A session object calculus for structured communication-based programming. `www.doc.ic.ac.uk/~mostrous/sesobj.pdf`. Submitted, 2008.
33. M. Neubauer and P. Thiemann. An implementation of session types. *PADL, Springer LNCS*, 3057:56–70, 2004.
34. O. Nierstrasz. Regular types for active objects. In *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.
35. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for Java. *CC, Springer LNCS*, 2622, 2003.
36. J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, uniqueness and immutability. In *IWACO (ECOOP workshop)*, 2007.
37. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
38. 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.
39. F. Puntigam. State inference for dynamically changing interfaces. *Computer Languages*, 27:163–202, 2002.
40. F. Puntigam and C. Peter. Types for active objects with static deadlock prevention. *Fundamenta Informaticae*, 49:1–27, 2001.
41. A. Ravara and V. T. Vasconcelos. Typing non-uniform concurrent objects. *CONCUR, Springer LNCS*, 1877:474–488, 2000.
42. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. *PARLE, Springer LNCS*, 817, 1994.
43. A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of software components using session types. *Fundamenta Informaticae*, 73(4):583–598, 2006.
44. 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.
45. V. T. Vasconcelos, S. J. Gay, A. Ravara, and N. Gesbert. Dynamic interfaces. `www.dcs.gla.ac.uk/~simon/publications/Dyn-Int.pdf`. FOOL, 2009.
46. V. T. Vasconcelos, A. Ravara, and S. J. Gay. Session types for functional multithreading. *CONCUR, Springer LNCS*, 3170:497–511, 2004.
47. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.