Dynamic Interfaces

Vasco T. Vasconcelos
Departamento de Informática
Faculdade de Ciências da
Universidade de Lisboa, Portugal
vv@di.fc.ul.pt

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

António Ravara
SQIG at Instituto de Telecomunicações
Department of Mathematics, IST,
Technical University of Lisbon, Portugal
amar@math.ist.utl.pt

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

Alexandre Z. Caldeira
Departamento de Informática
Faculdade de Ciências da
Universidade de Lisboa, Portugal
zua@di.fc.ul.pt

Abstract

We define a small class-based object-oriented language in which the availability of methods depends on an object's abstract state: objects' interfaces are *dynamic*. Each class has a *session type* which provides a global specification of the availability of methods in each state. A key feature is that the abstract state of an object may depend on the result of a method whose return type is an enumeration. Static typing guarantees that methods are only called when they are available. We present both a type *system*, in which the typing of a method specifies pre- and post-conditions for its object's state, and a typechecking *algorithm*, which infers the pre- and post-conditions from the session type, and prove type safety results. Inheritance is included; a subtyping relation on session types, related to that found in previous literature, characterizes the relationship between method availability in a subclass and in its superclass. We illustrate the language and its type system with example based on a Java-style iterator and a hierarchy of classes for accessing files, and conclude by outlining several ways in which our theory can be extended towards more practical languages.

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

FOOL '09 24 January, Savannah, Georgia, USA.
ACM .

1. Introduction

Standard class-based object-oriented languages present the programmer with the following view: an object is declared to belong to a certain type, and the type defines various methods; it is therefore possible at any time to call any of the methods described in the type. However, there are often semantic reasons why it is not appropriate to call a particular method when the object is in a particular internal state. For example: with a stack, one should not attempt to pop if the stack is empty; with a finite buffer, one should not attempt to write if the buffer is full; with a file, one should not attempt to read data until the file has been opened. We will refer to the set of available methods as the *interface* of an object, and use the term *dynamic interfaces* in connection with objects for which method availability depends on state. Objects with dynamic interfaces are also referred to in the literature as *non-uniform objects* (active objects with non-uniform service availability (36)), although usually in a concurrent setting.

Consider the `java.util.Iterator` interface.

```
interface Iterator {
  boolean hasNext()
  Object next()
  void remove()
}
```

One correct pattern for using an iterator `it` is as follows:

```
while (it.hasNext())
  {... it.next() ... }
```

while common programming errors, familiar to any teacher, include failing to call `hasNext()`:

```
for (int i = 0; i < 5; i++) {
  ... it.next() ... }
```

calling `next()` too often:

```
while (it.hasNext())
  if (it.next().equals(x))
    y.add(it.next());
```

and omitting an initial call of `hasNext()`:

```
b.append(it.next());
while (it.hasNext())
  b.append(",").append(it.next());
```

There are two problems with the above interface: a) it provides no clue on how its methods should be used; in fact it is not obvious from the English text that accompanies java.util.Iterator when one is supposed to call method remove; b) all of the code snippets above compile, errors being caught only at runtime, if ever.

We propose to discipline the order of method calls by annotating the above interface with a *session type*, inspired by work on type-theoretic specifications of communication protocols (44; 28), as follows.

```
session Init
where Init   = &{hasNext: Result}
      Result = ⊕{true: Next,
                 false: end}
      Next   = &{next: &{hasNext: Result,
                         remove: Init},
                 hasNext: Result}
```

The session type defines certain abstract states of iterator objects and specifies which methods are available in each state. For example, in state Init only the method hasNext() is available, and after calling it the abstract state becomes Result. The form of this state indicates that the method hasNext() returns a result of type **boolean** and that the subsequent state depends on the result. In general, & indicates available methods and ⊕ indicates possible results. In state Next, the method next() is available, and after calling it, the method remove() is also available. The keyword **end** is an abbreviation for an empty &, indicating that no methods are available, hence that the object cannot be used further and may be subject to garbage collection. The session type captures the specification of an iterator: hasNext() must be called in order to find out whether or not next() can be called, and remove() can be called at most once for each call of next(). Our type system makes sure that, not only methods are called by the specified order, but also that client code actually tests methods' results and proceeds accordingly, rendering the above programming errors untypable.

In the present paper we propose, in the sequential setting, a type system to support static checking of correctness of method calls in the presence of objects with dynamic interfaces. The key features of our approach are as follows.

- Each class declares a *session type* which provides an abstract view of the allowed sequences of method calls. In the simplest case, a session type defines a directed graph of abstract state transitions labelled by method names.
- A session type can also specify that the abstract state after a method call depends on the result of the call, where this result comes from an enumerated type. In this case the caller must perform a case-analysis on the result before calling further methods.
- We allow inheritance between classes, characterizing the conditions for inheritance by means of a subtyping relation on session types.
- We formalize the operational semantics and typing rules of a core language with these features, enabling us to prove a type safety result. Objects with dynamic interfaces are handled linearly in order to avoid aliasing problems.
- We define a typechecking algorithm, whose success guarantees not only type safety but also consistency between method definitions and session types, so that every sequence of methods calls in the session type is realizable in a typable program.

There is a substantial literature of related work, which we discuss in more detail in Section 7. Type systems for non-uniform concurrent objects have been proposed by several authors including Nierstrasz (36), Ravara *et al.* (42; 43) and Puntigam and Peter (40; 41). The type systems of the imperative languages *Cyclone* (24; 25), *Vault* (10; 18) and *Fugue* (19; 11) address similar issues in sequential programming. The related topic of type-theoretic specifications of protocols on communication channels has been studied for concurrent object-oriented languages by Dezani-Ciancaglini *et al.* (12; 14; 15; 16) and implemented in the *Sing#* language (17).

Contributions

Our work makes the following new contributions. In contrast with *Cyclone* and *Vault*, we define an object-oriented language with inheritance. In contrast with *Sing#* and other work on session types for object-oriented languages, we consider objects with dynamic interfaces in a setting more general than communication channels. In contrast with *Fugue*, we use a session type as a global specification of method availability, instead of pre- and post-conditions on methods. In contrast with work on non-uniform concurrent objects, we work with a Java-like language, not a mobile process calculus; and we force the caller to perform a case-analysis when necessary, meaning that our session types are richer than simply sequences of available methods.

The remainder of the paper is structured as follows. In Section 2 we illustrate our system by means of a more extensive example, extending it in Section 3 to include inheritance. In Section 4 we formalize a core language and prove a type safety result. The core language requires, in addition to the session type, explicit pre- and post-conditions for each method. In Section 5 we present a type-checking algorithm which infers the pre- and post-conditions from the session types. Section 6 describes our prototype implementation. Section 7 contains a more extensive discussion of related work, Section 8 outlines future work and Section 9 concludes.

2. Programming with Dynamic Interfaces

This section and the next section contain more extensive examples of programming with dynamic interfaces. From now on, we do not use the term *interface* in the technical Java sense; for us, an interface is simply a class definition without the code of the methods. Of course it includes a session type, making it dynamic.

Our next example involves the class FileReadToEnd, representing part of an API for using a file system. A file has a dynamic interface: it must first be opened, then can be read repeatedly, and must finally be closed. Before reading, a test for end of file must be carried out, in a way similar to the iterator. A key feature of FileReadToEnd is that the file cannot be closed until all of the data has been read. We will relax this restriction later. The example also contains a class FileReader, representing application code which uses FileReadToEnd to access a file and read its data into a string.

Figures 1 and 2 contain the code for the example. Figure 1 consists of three declarations. Lines 1 and 3 define enumerated types Res and Bool. Lines 5–19 define the interface FileReadToEnd. For technical reasons we use an enumerated type Bool rather than the primitive type **boolean**, and type **Null** rather than **void**. The session type is represented diagrammatically in Figure 7(a).

Our language does not include constructor methods as a special category, but the method open must be called first and can therefore be regarded as a constructor.

Figure 2 defines the class FileReader, which uses an object of class FileReadToEnd. The class diagram for the example is in Figure 8. This class has a session type of its own, defined on lines 2–5. It specifies that methods must be called in the sequence init, read, toString, toString, ... Line 7 defines the fields of FileReader. The core language does not require a type declaration for f, since this is a linear field, and linear fields always start with type **Null**. Lines 14–23 illustrate the **switch** construct. Unlike *Sing#* (17), we allow arbitrary code between the method call and the **switch**.

```
enum Res {OK, NOT_FOUND, DENIED;}                                  1
                                                                   2
enum Bool {FALSE, TRUE;}                                           3
                                                                   4
interface FileReadToEnd {                                          5
  session Init                                                     6
  where Init  = &{open: ⊕{Res.OK: Open,                            7
                          Res.NOT_FOUND: end,                      8
                          Res.DENIED: end}}                        9
        Open  = &{eof: ⊕{Bool.TRUE: Close,                         10
                        Bool.FALSE: Read}}                         11
        Read  = &{read:Open}                                       12
        Close = &{close:end}                                       13
                                                                   14
  Res open()                                                       15
  Bool eof()                                                       16
  String read()                                                    17
  Null close()                                                     18
}                                                                  19
```

Figure 1. The interface of a file that must be read to the end-of-file.

```
class FileReader {                                                 1
  session Init                                                     2
  where Init = &{init: Read}                                       3
        Read = &{read: Final}                                      4
        Final = &{toString: Final}                                 5
                                                                   6
  f; String s;                                                     7
                                                                   8
  Null init() {                                                    9
    f = new FileReadToEnd();                                       10
    s = "";                                                        11
  }                                                                12
  Null read() {                                                    13
    switch(f.open()) {                                             14
      case NOT_FOUND:                                              15
      case DENIED:                                                 16
        break;                                                     17
      case OK:                                                     18
        while (!f.eof())                                           19
          s = s + f.read();                                        20
        f.close();                                                 21
        break;                                                     22
  } }                                                              23
  String toString() { return s; }                                  24
}                                                                  25
```

Figure 2. A client that reads from a FileReadToEnd.

The **while** loop (lines 19–20) is similar. The result of f.eof() must be a constant from enumeration Bool. Line 24 defines the method toString which simply accesses a field.

Clearly, correctness of this code requires that the sequence of method calls on field f within class FileReader matches the &s in the session type of class FileReadToEnd, and that the appropriate **switch** or **while** loops are performed when prescribed by the ⊕s in the session type. Our static type system, defined in Section 4, enables this consistency to be checked at compile-time. In order to check statically that an object with a dynamic interface such as FileReader.f is used correctly, our type system treats the reference linearly so that aliases to it cannot be created.

In order to support *separate compilation* we require only the interface of a class. For example, in order to typecheck classes that are clients of FileReader, we rely on its interface, as defined in Figure 3. Similarly, to typecheck class FileReader, which is a client of FileReadToEnd, it suffices to use the interface in Figure 1, thus

```
interface FileReader {                                             1
  session &{init: &{read: Final}}                                  2
  where Final = &{toString: Final}                                 3
                                                                   4
  Null init()                                                      5
  Null read()                                                      6
  String toString()                                                7
}                                                                  8
```

Figure 3. Interface for FileReader.

```
interface FileRead extends FileReadToEnd {                         1
  session Init                                                     2
  where Init = ... // As in Figure 1                               3
        Open = &{eof: ⊕{Bool.TRUE: Close,                          4
                        Bool.FALSE:Read},                          5
                  close:end}                                       6
        Read = &{read:Open, close:end}                             7
        Close = &{close:end}                                       8
}                                                                  9
```

Figure 4. A file that can be closed before the end-of-file.

effectively supporting typing clients of classes containing *native methods*.

3. Subclassing Dynamic Interfaces

We now extend our example to illustrate inheritance and subtyping in the presence of session types. 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 ⊕ session type, then the set of values in C must be a subset of the set in D. When a method of class D is overridden by a method of class C, we allow contravariant changes in the parameter types and covariant changes in the result type.

Class FileRead (Figure 4) extends FileReadToEnd (Figure 1). The extension is that method close is now also available from states Open and Read. A new client FileBoundedReader (Figure 5) takes advantage of this possibility to read the first string in the file. Because method FileReader.init (the constructor) creates a object of class FileReadToEnd, the new client also *overrides* method init to create an object of the correct type. The class diagram is in Figure 8.

The original FileReader can safely use an object of class FileRead instead of the expected FileReadToEnd; it does not take advantage of the additional availability of close. The diagram in Figure 7(b) highlights the key idea: the session type of a subclass can add extra branches to a & type. Similarly, clients of class FileReader can use the subclass FileBoundedReader; they do not use the new method readFirst.

Class FileRead also illustrates a self-call (to method tryRead). This method does not show up in the session type of the class. Although our language does not include method qualifiers, method tryRead can be regarded as *private* since the type system ensures that it cannot be called by any client of class FileBoundedReader. There are other important features of the language that the example does not show: methods can be recursive, methods can be both "public" and "private" in the above sense, and methods can have

```
class FileBoundedReader extends FileReader {
  session Init
  where ... // see Figure 3
        Read = &{read: Final,
                 toString: Final,
                 readFirst: Final}
  @Override
  Null init() {
    f = new FileRead();
    s = "";
  }
  Null readFirst() {
    switch(f.open()) {
      case NOT_FOUND:
      case DENIED:
        break;
      case OK:
        tryRead();
  } }
  Null tryRead() {
    switch(f.eof()) {
      case TRUE: break;
      case FALSE: f.read(); break;
    }
    f.close();
} }
```

Figure 5. A client that reads strings from a FileRead.

```
enum KindRes restricts Res {OK;}

interface KindFileRead extends FileRead {
  session Init
  where Init = &{open: ⊕{KindRes.OK: Open}
        ... // see Figure 5
  @Override
  KindRes open()
}
```

Figure 6. A file that can always be opened for reading.

parameters. Because of this last feature, the language supports passing objects, hence *passing sessions*: a class may create an object, use it according to the initial part of its session type, and then send the object to another class that continues using the object according to the session type. Reference (23) contains examples of these kinds.

Classes can also be subtyped by restricting the range of the possible results of method calls. For example, class KindFileRead (Figure 6) extends class FileRead (Figure 5). The method open is overridden and the new version never returns NOT_FOUND or DENIED. The diagram in Figure 7(c) illustrates the idea of removing two branches from a ⊕ type in this way. The return type of open is now an enumeration containing only the value OK. To observe the usual requirement of covariant changes to the return type of a method, we define a new enumeration KindRes which is a subtype of Res. The syntax **enum** KindRes **restricts** Res {OK;} is analogous to the **extends** form for classes, but instead of specifying additional members it specifies the remaining values.

The original FileReader can safely use a KindFileRead; the NOT_FOUND and DENIED branches of the **switch** statment will never be called. In the new KindFileBoundedReader class (not shown), the **switch** statement has just one branch, for OK. Had we defined KindFileRead and KindBoundedReader in isolation there would be no need to use an enumeration at all, but it is necessary for compatibility with the rest of the class hierarchy.

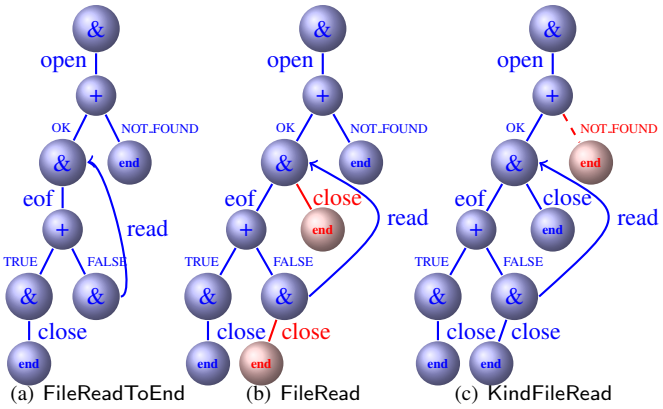

Figure 7. The diagrammatic representation of the session types for the classes in Figures 1, 5 and 6 (branch DENIED omitted).

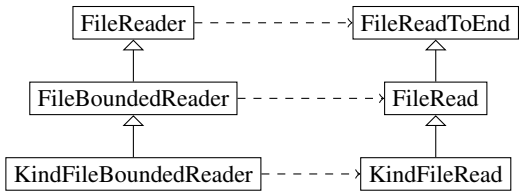

Figure 8. Class diagram for the classes in the example.

4. A Core Language for Dynamic Interfaces

We now present a formal syntax, operational semantics and type system for a core language containing the essential features of the examples in Sections 1–3, and state a type safety result. All objects have dynamic interfaces, meaning that all objects are handled linearly by the type system, whereas a practical language would also contain standard (non-dynamic) objects. 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 formal language only includes classes; the interfaces in Sections 2 and 3 are used for illustration purposes only.

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 5 infers the pre- and post-conditions for each "public" method.

4.1 Syntax

We separate the syntax into the programmer's language (Figure 9) and the extensions required by the type system and operational semantics (Figure 10). 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.

$$
\begin{array}{rrcl}
\text{Class dec} & D & ::= & \mathsf{class}\ C\ \{S; \vec{f}; \vec{M}\} \mid \mathsf{enum}\ E\ L \mid \\
& & & \mathsf{class}\ C\ \mathsf{extends}\ C\ \{S; \vec{f}; \vec{M}\} \mid \\
& & & \mathsf{enum}\ E\ \mathsf{restricts}\ E\ L \\
\text{Constant sets} & L & ::= & \{l_i\}_{i \in I} \\
\text{Method dec} & M & ::= & T\ m(T\ x)\ \{e\} \\
\text{Values} & v & ::= & \mathsf{null} \mid E.l \mid o \\
\text{Value references} & r & ::= & v \mid x \mid o.f \\
\text{Expressions} & e & ::= & r \mid \mathsf{new}\ C() \mid o.f\ =\ e \mid \\
& & & r.m(e) \mid e; e \mid \mathsf{while}\ (e)\ \{e\} \mid \\
& & & \mathsf{switch}\ (e)\ \{\mathsf{case}\ l_i\colon e_i\}_{i \in I} \\
\text{Types} & T & ::= & \mathsf{Null} \mid E \mid C[S] \\
\text{Session types} & S & ::= & \&\{m_i\colon S_i\}_{i \in I} \mid \oplus\{E.l_i\colon S_i\}_{i \in I} \mid \\
& & & \mu X.S \mid X
\end{array}
$$

The only object reference o available to the programmer is this.

Figure 9. Programmer's syntax

$$
\begin{array}{rrcl}
\text{Method dec} & M & ::= & \mathsf{req}\ T\ \mathsf{ens}\ T\ \mathsf{for}\ T\ m(T\ x)\ \{e\} \\
\text{Field types} & F & ::= & \vec{T}\,\vec{f} \\
\text{Types} & T & ::= & \ldots \mid C[S; F] \mid T\ \mathsf{link}\ r \mid \langle E.l_i\colon T_i\rangle_{i \in I} \\
\text{Field values} & V & ::= & \vec{f} = \vec{v} \\
\text{Heaps} & h & ::= & \varepsilon \mid h :: o = C[V] \\
\text{States} & s & ::= & h; e \\
\text{Contexts} & \mathcal{E} & ::= & [\_] \mid o.f\ =\ \mathcal{E} \mid \mathcal{E}; e \mid r.m(\mathcal{E}) \mid \\
& & & \mathsf{switch}\ (\mathcal{E})\ \{\mathsf{case}\ l_i\colon e_i\}_{i \in I}
\end{array}
$$

Figure 10. Syntax used only in the type system and semantics.

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.

$C[S]$ is the type of an object of class C in state (session type) S. E is an enumerated type, and the type Null has the single value null. The type system uses type environments Γ, which are functions assigning types to object references o.

Session types have been discussed in relation to the example. Session type end abbreviates $\&\{\}$. 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 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 (39, Chapter 21) and regard $\mu X.S$ and $S\{(\mu X.S)/X\}$ as equivalent, using them interchangeably in any mathematical context.

Figure 10 defines additional syntax needed for the formal system. It is not available to the programmer. In the 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 (45).

4.2 Operational Semantics

Figure 11 defines an operational semantics on configurations $h; e$ consisting of a heap and an expression. The rules are implicitly parameterized by $\vec{D}$, the list of declarations constituting the program. We regard a heap h as a function from object references o to objects $C[\vec{f} = \vec{v}]$. 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.

The rules central to our proposal—method call and switch—and their interplay with the novel link types are better described via an example. The reduction of the expression

$$o.g = o.f.m_j(); \mathsf{switch}\ (o.g)\ \{\mathsf{case}\ l_k\colon e_k\}_{k \in K}$$

is illustrated in Figure 12. The figure shows the environment in which each expression is typed; the environment changes as reduction proceeds, because a method call advances the session type of the object (and because of the link type). The typing of an expression is $\Gamma \triangleright e : T \triangleleft \Gamma'$ but we only show Γ because Γ' does not change. We also omit the heap, showing the typing of expressions instead of states, and the type of the expression, which is not the interesting part of this example. Calling $o.f.m_j()$ introduces a variant type for field f. 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. Finally, the switch resolves the variant type of $o.f$ according to the particular enumerated value $E.l_p$.

R-NEW creates a new object in the heap, with null fields. R-FIELD 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.

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-WHILE defines the behaviour of a while expression by rewriting to an appropriate switch. R-SWITCH itself 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.

To complete the definition of the semantics we need to define the initial state. 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

$$o = C[\mathit{fields}(C) = \vec{\mathsf{null}}];\ o.m(\ldots).$$

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

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

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

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

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

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

$$h;\ \mathsf{while}\ (e)\ \{e'\} \longrightarrow h;\ \mathsf{switch}\ (e)\ \{\mathsf{case\ F}\ :\ \mathsf{null}, \mathsf{case\ T}\ :\ e'; \mathsf{while}\ (e)\ \{e'\}\} \qquad \text{(R-WHILE)}$$

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

Figure 11. Reduction rules.

$$\begin{array}{rrcl}
 & o : C[S; C'[\&\{m_i\colon S_i\}_{i\in I}]\ f, T\ g] & \triangleright & o.g = o.f.m_j(); \mathsf{switch}\ (o.g)\ \{\mathsf{case}\ l_k\colon e_k\}_{k\in K} \\
\longrightarrow \text{(R-INDIRCALL)} & o : C[S; \langle E.l_k\colon C'[S_k]\rangle_{k\in K}\ f, T\ g] & \triangleright & o.g = e\{h(o.f)/\mathsf{this}\}; \mathsf{switch}\ (o.g)\ \{\mathsf{case}\ l_k\colon e_k\}_{k\in K} \\
\longrightarrow^* & o : C[S; \langle E.l_k\colon C'[S_k]\rangle_{k\in K}\ f, T\ g] & \triangleright & o.g = E.l_p; \mathsf{switch}\ (o.g)\ \{\mathsf{case}\ l_k\colon e_k\}_{k\in K} \\
\longrightarrow \text{(R-ASSIGN, R-SEQ)} & o : C[S; \langle E.l_k\colon C'[S_k]\rangle_{k\in K}\ f, (E\ \mathsf{link}\ o.f)\ g] & \triangleright & \mathsf{switch}\ (o.g)\ \{\mathsf{case}\ l_k\colon e_k\}_{k\in K} \\
\longrightarrow \text{(R-FIELD)} & o : C[S; \langle E.l_k\colon C'[S_k]\rangle_{k\in K}\ f, \mathsf{Null}\ g] & \triangleright & \mathsf{switch}\ (E.l_p)\ \{\mathsf{case}\ l_k\colon e_k\}_{k\in K} \\
\longrightarrow \text{(R-SWITCH)} & o : C[S; C'[S_p]\ f, \mathsf{Null}\ g] & \triangleright & e_p
\end{array}$$

Figure 12. Example of the interplay between method call, switch and link types.

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

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

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

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

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

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

Figure 13. Subtyping rules for types and method signatures.

4.3 Subtyping

The foundation for the theory of inheritance and subtyping is the definition of subtyping between session types. Let $\mathcal{S}$ be the set of 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* session type simulation *if $(S, S') \in R$ implies the following conditions.*

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

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

The direction of subtyping is opposite to that defined in (21), 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, the point is that 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 easily be adapted from (21).

Figure 13 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.

It turns out that both the **requires** and **ensures** types behave covariantly. For **ensures** this is because the type is really part of the

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

(T-HEMPTY,T-HWEAK,T-HADD)

$$\frac{\Gamma \vdash h \quad \Gamma(r) = E \text{ link } o \quad h(r') = o}{\Gamma\{r \mapsto E \text{ link } r'\} \vdash h} \qquad \text{(T-HLINK)}$$

$$\frac{\vdash \vec{D} \quad \Gamma \vdash h \quad \Gamma \triangleright e : T \triangleleft \Gamma'}{\Gamma \triangleright h;\ e : T \triangleleft \Gamma'} \qquad \text{(T-STATE)}$$

$$\frac{\Gamma, o : T_0 \vdash h :: o = C[f = o', \ldots] \quad T_0.f = T_1 \quad \Gamma, o' : T_1 \triangleright h;\ e : T \triangleleft \Gamma', o' : T_2 \quad T' = \left\{ \begin{smallmatrix} E \text{ link } o.f \text{ if } T = E \text{ link } o' \\ T \text{ otherwise} \end{smallmatrix} \right.}{\Gamma, o : T_0 \triangleright h :: o = C[f = o', \ldots];\ e : T' \triangleleft \Gamma', o : T_0\{f \mapsto T_2\}}$$

(T-EXPRSTATE)

Figure 16. Typing rules for heaps and states.

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.

4.4 Type System

The type system is defined by the rules in Figures 14, 15, and 16. 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 (due to their dynamic interfaces). We regard an environment Γ as a function from object references o to object types $C[S; \vec{T}\,\vec{f}]$. 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 14, which defines typing of expressions. T-NEW types a new object, giving it the initial session type from the class declaration and giving all the fields type Null. 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-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 e, 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.

T-WHILE is derived from T-SWITCH and the fact that a while expression reduces to a switch with two branches. The terminating branch is null and does not appear as a hypothesis. The looping branch, which is the expression e', must be typable and must preserve the environment, so that the loop makes sense. T-SEQ and T-SUB are standard.

Now consider Figure 15. 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.

Figure 16 defines rules for typing heaps and states (runtime 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.

4.5 Results

By standard techniques (45) adapted to typing judgements with initial and final environments (22) we can prove a type preservation theorem of the usual kind.

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

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

DEFINITION 2 (Call Traces). *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 call trace for every object. Self-calls are excluded from call traces. 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.

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

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

5. Typechecking Algorithm

Figure 17 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 req $C[\&\{m_i\colon S_i\}_{i \in I}]$ ens $C[S_i]$ for $T\ m_i\{e\}$. 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 form of overloading which is useful in the example of Figure 4, allowing method close to be called in three different states. In this case, the req type should be used to disambiguate methods 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, 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.

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

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

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

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

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

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

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

$$\frac{\Gamma \triangleright e : \mathsf{Bool}\ \mathsf{link}\ r \triangleleft \Gamma' \quad \Gamma'(r) = \langle \mathsf{Bool.F} : T_f, \mathsf{Bool.T} : T_t\rangle \quad \Gamma'\{r \mapsto T_t\} \triangleright e' : T \triangleleft \Gamma}{\Gamma \triangleright \mathsf{while}\ (e)\ \{e'\} : \mathsf{Null} \triangleleft \Gamma'\{r \mapsto T_f\}} \tag{T-WHILE}$$

$$\frac{\Gamma \triangleright e : T \triangleleft \Gamma'' \quad \Gamma'' \triangleright e' : T' \triangleleft \Gamma' \quad T \neq E \ \mathsf{link}\ r}{\Gamma \triangleright e; e' : T' \triangleleft \Gamma'} \qquad \frac{\Gamma \triangleright e : T \triangleleft \Gamma' \qquad T <: U}{\Gamma \triangleright e : U \triangleleft \Gamma'} \tag{T-SEQ,T-SUB}$$

Figure 14. Typing rules for expressions.

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

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

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

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

Figure 15. Typing rules for programs.

The definition of $\mathcal{B}$ follows the typing rules (Figure 14) 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 FileReadToEnd in Figure 1 produces the following annotated methods.

```
req FileReadToEnd[Init]
ens FileReadToEnd[⊕{Res.OK: Open,...}]
Res open()

req FileReadToEnd[Open]
ens FileReadToEnd[⊕{Bool.True: Close,...}]
Bool eof()

req FileReadToEnd[Read]
ens FileReadToEnd[Open]
String read()

req FileReadToEnd[Close]
ens FileReadToEnd[end]
Null close()
```

These annotations are used when algorithm $\mathcal{A}$ is applied to class FileReader in Figure 2 producing the following annotated methods, that include information about the field typing within FileReader, but no information about fields within FileReadToEnd, for which we do not have the source code.

```
req FileReader[Init; Null f]
ens FileReader[Read; FileReadToEnd[Init] f]
Null init()

req FileReader[Read; FileReadToEnd[Init] f]
ens FileReader[Final; FileReadToEnd[end] f]
Null read()
```

Algorithm $\mathcal{P}$

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

Algorithm $\mathcal{A}$

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

Algorithm $\mathcal{B}$

$\mathcal{B}_C^S(\mathsf{null}, F, \_) = (F, \mathsf{Null})$
$\mathcal{B}_C^S(E.l, F, \_) = (\langle E.l\colon C[S; F]\rangle, E)$
$\mathcal{B}_C^S(\mathsf{new}\ C(), F, \_) = (F, C[\mathit{session}(C)])$
$\mathcal{B}_C^S(\mathsf{this}.f, F, \_) = (F + f : \mathsf{Null}, F(f))$
$\mathcal{B}_C^S(x, F, \_) = (F + x : \mathsf{Null}, F(x))$
$\mathcal{B}_C^S(\mathsf{this}.f\ =\ e, F, \Delta) = (F' + \mathsf{this}.f : T, \mathsf{Null})$
  where $(F', T) = \mathcal{B}_C^S(e, F, \Delta)$
$\mathcal{B}_C^S(\mathsf{this}.f.m_j(e), F, \_) = (F + f : \mathsf{Null}, T')$
  where $(F', T_x) = \mathcal{B}_C^S(e, F, \Delta)$
  and $F'(f) = C'[\&\{m_i\colon S_i\}_{i\in I}]$ and $j \in I$
  and $\mathsf{req}\ C'[\&\{m_i\colon S_i\}_{i\in I}]\ \mathsf{ens}\ \_\ \mathsf{for}\ T\ m_j(T_x\ x)\ \{\_\}$
    $\in \mathit{methods}(\mathcal{P}_{C'}(\mathit{session}(C'), \emptyset))$
  and $T' =$ if $S_j = \oplus\{\dots\}$ then T $\mathsf{link}\ \mathsf{this}$ else T
$\mathcal{B}_C^S(\mathsf{this}.m(e), F, \Delta) =$ if $m \in \mathrm{dom}(\Delta)$ then $(\Delta(m), T)$
    else $\mathcal{B}_C^S(e, F', \Delta \cup \{m \mapsto (F')\})$
  where $(F', T_x) = \mathcal{B}_C^S(e, F, \Delta)$
  and $T\ m(T_x\ x)\{e\} \in \mathit{methods}(C)$
$\mathcal{B}_C^S(\mathsf{switch}\ (e)\ \{\mathsf{case}\ l_i\colon e_i\}_{i\in I}, F, \Delta) = (\biguplus F_i, T)$
  where $(F', E\ \mathsf{link}\ \mathsf{this}) = \mathcal{B}_C^S(e, F, \Delta)$
  and $(F_i, T) = \mathcal{B}_C^S(e_i, F', \Delta)$ and $l_i \in \mathit{constants}(E)$
$\mathcal{B}_C^S(\mathsf{while}\ (e)\ \{e'\}, F, \Delta) = (F_f, \mathsf{Null})$
  where $(\langle \mathsf{Bool.F} : C[\_; F_f], \mathsf{Bool.T} : C[\_; F_t]\rangle, E\ \mathsf{link}\ \mathsf{this}) =$
    $\mathcal{B}_C^S(e, F, \Delta)$
  and $(F, \_) = \mathcal{B}_C^S(e', F_f, \Delta)$
$\mathcal{B}_C^S((e; e'), F, \Delta) = \mathcal{B}_C^S(e', F', \Delta)$
  where $(F', \_) = \mathcal{B}_C^S(e, F, \Delta)$

Combining partial variants

$T \uplus T = T$ if T is not a variant typing
$\langle E.l_i\colon T_i\rangle_{i\in I} \uplus \langle E.m_j\colon T'_j\rangle_{j\in J} = \langle E.l_k\colon T''_k\rangle_{k\in I\cup J}$
  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 $T_i = T'_j$

Figure 17. Algorithm

```
req FileReader[Final; FileReadToEnd[end] f]
ens FileReader[Final; FileReadToEnd[end] f]
String toString()
```

Notice that both algorithm $\mathcal{P}$ and algorithm $\mathcal{A}$ are driven by the session type of the class, hence, e.g., method tryRead in Figure 5 will not be annotated, although its code will be analysed triggered by the call in method readFirst .

The main results about the algorithm are: 1) it always terminates; and 2) it produces only well-typed class declarations. To prove the latter we need to check consistency between the session type and the pre- and post-conditions of the methods, as information about the allowed sequences of method calls is now specified in two places: the session type of a class, and the req and ens clauses of the methods.

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 $\vec{D}$, $\mathcal{P}_C(S, \emptyset)$ and $\mathcal{A}_C(\mathsf{Null}\vec{f}, S, \emptyset)$ terminate.*
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 $\mathcal{A}_C(\mathsf{Null}\vec{f}, S, \emptyset)$, 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. View the session type of C as a (possibly infinite) tree, with branching at & and + nodes.
2. Make it into a finite tree by replacing some & nodes by end.
3. For each & node, remove all except one of the branches; call the resulting 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.

6. Implementation

We have used the Polyglot (37) 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. 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 5. It is available from
`http://gloss.di.fc.ul.pt/bica/`.

Calling a method out of the order specified by the session type will be detected and reported as an error. For the FileReader example (Figure 2), if we try to read from the file while it is not opened, (by including a call `f.read()` before the **switch** statement in line 14), the compiler will exit with an error message:

```
FileReader.java:14:
 Cannot proceed with call to f.read()
 in state &{open:...}
```

If we omit the call `f.close()` in line 21, Figure 2, the compiler detects that the three branches (NOT_FOUND, DENIED, and OK) do not end with f in the same state.

```
FileReader.java:14-23:
  Switch cases end in different states:
    NOT_FOUND: [f: end]
    DENIED: [f: end]
    OK: [f: &{close: end}]
```

7. Related Work

Previous work on session types for object-oriented languages. Several recent papers by Dezani-Ciancaglini, Yoshida *et al.* (14; 29; 5; 13; 15; 35) 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 have recently done (23), 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 (35) add sessions to Abadi and Cardelli's object calculus. Our approach is substantially different: we use session types as global specifications of class behaviour, supporting thus a disciplined use of non-uniform objects, not (only) to discipline communication channels.

Non-uniform concurrent objects / active objects. Another related line of research was started by Nierstrasz (36), aimed at describing the *active* objects in concurrent systems, whose non-uniform 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 (43; 40; 41; 4; 9).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 (34) 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* (25),*Vault* (10; 18), and CQual (20) 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 (18) to re-introduce aliasing in limited situations. *Fugue* (19; 11) 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# (17) 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 (17) 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* (26), Almeida's *balloon types* (1), Clarke *et al.*'s *ownership types* (8), Fähndrich and DeLine's *adoption and focus* (18), Östlund *et al.*'s Joe$_3$ (38) 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 (31) 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 (30) including a verification tool (32) in which to define type systems for analyzing various behavioural properties including sequences of resource uses (33). In some of this work, types are themselves abstract processes, and therefore situations resemble our session types. Chaki *et al.* (7) follow Kobayashi's approach and 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, as they are interested in liveness properties.

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. It is not clear how to lift the generic types pproach from process algebra to a programming language, as we have done with session types.

8. Future Work

There are several topics for future work.

Shared classes. In the present system, all classes are linear. It is straightforward to add shared classes, whose objects do not have to be uniquely referenced. 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. Our system does not have this property, but to achieve it we only need to change the rules for assignment so that an incompletely-used object cannot be discarded. In a practical language this condition could be specified independently for each object.

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

Concurrency. We have extended our system to a concurrent language with channel-based communication (23), unifying dynamic interfaces, the *Sing#* (17) version of session types, and other work on object-oriented session types (14; 15; 16; 29). It will also be interesting to look at RMI for dynamic interfaces.

Specifications involving several objects. Multi-party session types (3; 27) allow to describe specifications that involve more than two objects. The introduction of such types would allow disciplining the usage of more sophisticated patterns of object usage.

9. Conclusions

We have defined a core class-based object-oriented language with a static type system which is able to check correctness of method calls in a setting in which method availability depends on an object's state. We have proved the correctness of the type system with respect to a formal operational semantics. Key features of the language are: firstly, allowing a dependency between the result of a method and the subsequent abstract state of its object, thereby forcing the client of such a method to branch accordingly; secondly, the use of a session type as a global representation of the abstract states of an object, enabling us to define rules for inheritance in terms of a subtyping relation on session types.

Acknowledgments

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. In *ECOOP*, volume 1241 of *LNCS*, pages 32–59. Springer, 1997.

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

[3] E. Bonelli and A. Compagnoni. Multipoint Session Types for a Distributed Calculus. In *Proceedings of the Third International Symposium on Trustworthy Global Computing*, volume 4912 of *Lecture Notes in Computer Science*, pages 240–256. Springer-Verlag, 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. In *ESOP*, number 4421 in LNCS, pages 2–17. Springer, 2007.

[7] S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *POPL*, pages 45–57. ACM Press, 2002.

[8] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, volume 33(10) of *SIGPLAN Notices*. ACM Press, 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. In *PLDI*, volume 36(5) of *SIGPLAN Notices*, pages 59–69. ACM Press, 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. In F. de Boer and M. Bonsangue, editors, *FMCO'06*, volume 4709 of *LNCS*, pages 207–245. Springer, 2007.

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

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

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

[16] S. Drossopoulou, M. Dezani-Ciancaglini, and M. Coppo. Amalgamating the Session Types and the Object Oriented Programming Paradigms. In *MPOOL'07*, 2007.

[17] 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 Press, 2006.

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

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

[20] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2002. ACM.

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

[22] S. J. Gay, A. Ravara, and V. T. Vasconcelos. Session types for inter-process communication. Technical Report TR-2003-133, Department of Computing Science, University of Glasgow, March 2003.

[23] S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-oriented programming. Submitted.

[24] D. Grossman. Type-safe multithreading in Cyclone. In *TLDI*, volume 38(3) of *SIGPLAN Notices*, pages 13–25. ACM Press, 2003.

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

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

[27] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, pages 273–284. ACM, 2008.

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

[29] R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in java. In *ECOOP*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.

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

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

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

[33] 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.

[34] F. Militão. Yak programming language. `ctp.di.fct.unl.pt/yak`, 2008.

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

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

[37] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for Java. In *12th International Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 138–152. Springer-Verlag, 2003.

[38] J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, uniqueness and immutability. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2007.

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

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

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

[42] A. Ravara and L. Lopes. Programming and implementation issues in non-unifom TyCO. Technical report, Department of Computer Science, Faculty of Sciences, University of Porto, Portugal, 1999.

[43] A. Ravara and V. T. Vasconcelos. Typing non-uniform concurrent objects. In *CONCUR*, volume 1877 of *LNCS*, pages 474–488. Springer, 2000.

[44] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE*, volume 817 of *LNCS*. Springer, 1994.

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