Dynamic Interfaces

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

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

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*, 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 the interface 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 (40; 25), as follows.

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


FOOL 2009 24 January, 2009, Savannah, Georgia, USA.

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 $\oplus$ 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 6. Type systems for non-uniform concurrent objects have been proposed by several authors including Nierstrasz (31), Ravara *et al.* (38; 39) and Puntigam and Peter (35; 36). The type systems of the imperative languages *Cyclone* (21; 22), *Vault* (9; 16) and *Fugue* (17; 10) 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.* (11; 12; 13; 14) and implemented in the *Sing#* language (15).

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;

```
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
                                                                     16
  Bool eof()                                                         17
                                                                     18
  String read()                                                      19
                                                                     20
  Null close()                                                       21
}                                                                    22
```

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

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 typechecking algorithm which infers the pre- and post-conditions from the session types. Section 6 contains a more extensive discussion of related work, Section 7 outlines future work and Section 8 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 5–22 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

```
class FileReader {                                          1
  session Init&{init: &{read: Final}}                       2
  where Init = &{init: Read}                                3
        Read = &{read:Final}                                4
        Final = &{toString: Final}                          5
                                                            6
  Null f;                                                   7
  Res g;                                                    8
  Bool b;                                                   9
  String s;                                                 10
                                                            11
  Null init() {                                             12
    f = new FileReadToEnd();                                13
    s = ""; }                                               14
                                                            15
  Null read() {                                             16
    g = f.open();                                           17
    switch(g) {                                             18
      case Res.NOT_FOUND:                                   19
      case Res.DENIED:                                      20
        break;                                              21
      case Res.OK:                                          22
        b = f.eof();                                        23
        while (!b) {                                        24
          s = s + f.read();                                 25
          b = f.eof();                                      26
        };                                                  27
        f.close();                                          28
        break; } }                                          29
                                                            30
  String toString() { return s; }                           31
}                                                           32
```

Figure 2. A client that reads from a FileReadToEnd.

and 3. It specifies that methods must be called in the sequence init, read, toString, toString, ...

Lines 5–8 define 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 15–30 define the method read. Lines 16–17 illustrate the point that **switch** must be applied to a field, not an arbitrary expression, because of the details of the semantics and type system, which will be explained in Section 4. Unlike *Sing#* (15), we allow arbitrary code between the method call and the **switch**.

The **while** loop (lines 22–29) is similar. Our language forces separating the method call from the loop test, thus requiring an extra method call at the end of the loop; however, forgetting to call the method inside the loop will be detected by the type system. The result of f.eof() must be a constant from enumeration Bool. Line 32 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 effectively supporting typing clients of classes containing *native methods*.

```
interface FileReader {                                      1
  session &{init: &{read: Final}}                           2
  where Final = &{toString: Final}                          3
                                                            4
  Null init()                                               5
                                                            6
  Null read()                                               7
                                                            8
  String toString()                                         9
}                                                           10
```

Figure 3. Interface for FileReader.

```
interface FileRead extends FileReadToEnd {                 1
  session Init                                              2
  where Init = ... // As in Fig. 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.

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 D is overridden by a method of C, we allow covariant changes in the result type.

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.

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

```
class FileBoundedReader extends FileReader {
  session &{init: &{read: Final,
                    readNext: Final}}
  where ... // see Fig. 3
  @Override
  Null init() {
    f = new FileRead(); s = ""; }

  Null readFirst() {
    g = f.open();
    switch(g) {
      case Res.NOT_FOUND:
      case Res.DENIED: break;
      case Res.OK:
        b = f.eof();
        switch(b) {
          case Bool.TRUE: break;
          case Bool.FALSE:
            f.read();
            f.close();
            break;
        }
        break; } }
}
```

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

```
enum KindRes restricts Res {OK;}

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

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

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.

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 prove a type safety result. The main simplification is that methods must return enumerated types, which means that all session types alternate between & and ⊕ constructors. 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. We also omit method parameters, which complicate the system but don't require new conceptual techniques. The formal language only includes classes; the interfaces in Sections 2 and 3 should be interpreted as classes without method bodies.

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 (10) but stated in terms of session types. The typechecking algorithm presented in Section 5 infers the pre- and post-conditions.

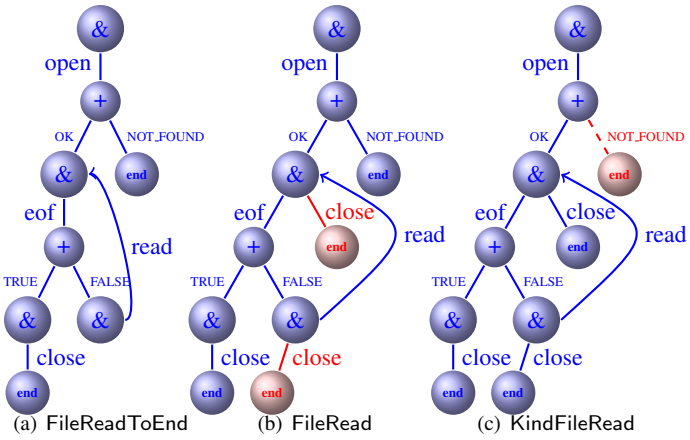

Figure 7. The diagrammatic representation of the session types for the classes in Figures 1, 5 and 6 (branch DENIED omitted).

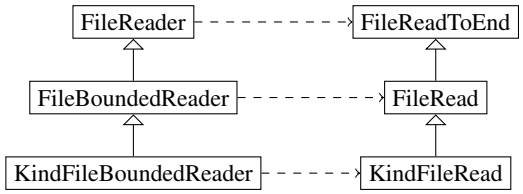

Figure 8. Class diagram for the classes in the example.

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

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

Figure 9. Programmer's syntax

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.

Method dec	M	$::= \mathsf{req}\ T\ \mathsf{ens}\ T\ \mathsf{for}\ T\ m()\ \{e\}$
Values	v	$::= \ldots \mid v\ \mathsf{then}\ f = o$
Expressions	e	$::= \ldots \mid e\ \mathsf{then}\ f = o$
Field types	F	$::= \vec{T}\,\vec{f} \mid \langle E.l_i\colon F_i\rangle_{i\in I}$
Types	T	$::= \ldots \mid C[S] \mid C[S;F] \mid T\ \mathsf{then}\ T\ f$
Heaps	h	$::= \varepsilon \mid h :: o = C[S;V]$
Field values	V	$::= \vec{f} = \vec{v}$
States	s	$::= h; e$
Contexts	E	$::= [\_] \mid o.f = E \mid E; e \mid E\ \mathsf{then}\ f = o$

Figure 10. Runtime (and type system) syntax

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.

In the syntax of expressions, field access, assignment and method call are only available on an object reference, not an arbitrary expression; moreover, the only object reference available to the programmer is this. 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 could be easily be inserted by the compiler. A more unusual point is that we only switch on a field, not an arbitrary expression; similarly the condition of a while loop tests a field. Section 2 illustrates the corresponding programming style, which we will discuss further during the rest of this section.

In the internal syntax (Figure 10), the expression e then $f = o$ and the corresponding type T then U f are essential to allow the enum result of a method to be stored and tested later.

Field values, heaps and states are used to define the operational semantics. An entry in the heap maps an object reference to an object: $o = C[S; \vec{f} = \vec{v}]$, where C is the class, S is the session type and the fields $\vec{f}$ have values $\vec{v}$. The session type is used as an abstraction of the state, to indicate which methods are available. It is only needed in order to state type safety, namely that only available methods may be called. An implementation would not need that information: once typechecked, a program can be executed using a simplified version of heaps, without session types.

Session types have been discussed in relation to the example. Session type end abbreviates $\&\{\}$. In $\oplus\{C.l_i\colon S_i\}_{i\in I}$, C 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 or the **session** and **where** clauses from the examples; we just work with recursive session type expressions of the form $\mu X.S$, which are required to be *contractive*, i.e. containing no subexpression of the form $\mu X_1.\cdots\mu X_n.X_1$. We adopt the equi-recursive approach (34, Chapter 21) and regard $\mu X.S$ and $S\{(\mu X.S)/X\}$ as equivalent, using them interchangeably in any mathematical context.

A typing of the fields of an object is either a list $\vec{T}\,\vec{f}$ of typed fields or a variant type over typed fields. In the variant type $\langle C.l_i\colon \vec{T_i}\,\vec{f}\rangle_{i\in I}$, the labels l_i come from an enum C. The type of an object, $C[S;F]$, consists of a class, a session type, and a typing of the fields. The type Null has the single value null.

Finally, the type system uses type environments Γ, which are functions assigning types to object references o; the operational semantics uses reduction contexts $E[\ ]$ in the style of Wright and Felleisen (41).

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 $C[S; \vec{f} = \vec{v}]$. If $h(o) = C[S; \vec{f} = \vec{v}]$ then $h(o)$.class means C, $h(o)$.session means S, and $h(o).f_i$ means v_i. If $h(o)$ is defined (this is an implicit hypothesis) then the notation $h + o.f = v$ means the heap obtained by changing the value of field f in object o to v. Similarly $h + o.\mathsf{session} = S$.

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

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

is illustrated in Figure 12. Calling $o.f.m_j()$ nullifies $o.f$ and introduces the delayed assignment then $f = o'$. Once the then expression reduces to a value, the method returns, and the then value is stored in field g. The subsequent switch deconstructs the then-value, selects the appropriate case and exercises the delayed assignment, storing o' back in field f.

R-NEW creates a new object in the heap, with an initial session type taken from the class declaration and with its fields set to null. 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.

R-WHILEF/R-WHILET define the behaviour of a while expression to be equivalent to a switch with two branches, one which terminates immediately and one which executes e and then loops. 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 $\emptyset; \mathsf{new}\ C().m()$. Given that we can call methods on fields only, we set up a dummy class C', with a completed session type (end), a single field f, and no methods. The initial state is then

$$o = C'[\mathsf{end}; f = \mathsf{null}];\ (o.f = \mathsf{new}\ C(); o.f.m())$$

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

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

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

$$\frac{h(o).f = o' \quad h(o').\mathsf{session} = \&\{m\colon S, \ldots\} \quad \_m()\ \{e\} \in methods(h(o').\mathsf{class})}{h;\ o.f.m() \longrightarrow h + o.f = \mathsf{null} + o'.\mathsf{session} = S;\ e\{o'/\mathsf{this}\}\ \mathsf{then}\ f = o'} \qquad \text{(R-CALL)}$$

$$\frac{h(o).f = (E.l_j\ \mathsf{then}\ g = o') \quad h(o').\mathsf{session} = \oplus\{E.l_j\colon S_j, \ldots\} \quad j \in I}{h;\ \mathsf{switch}\ (o.f)\ \{\mathsf{case}\ l_i\colon e_i\}_{i\in I} \longrightarrow h + o.f = \mathsf{null} + o.g = o' + o'.\mathsf{session} = S_j;\ e_j} \qquad \text{(R-SWITCH)}$$

$$\frac{h(o).f = (\mathsf{Bool.F}\ \mathsf{then}\ g = o') \qquad h(o').\mathsf{session} = \oplus\{\mathsf{Bool.F}\colon S, \ldots\}}{h;\ \mathsf{while}\ (o.f)\ \{e\} \longrightarrow h + o.f = \mathsf{null} + o.g = o' + o'.\mathsf{session} = S;\ \mathsf{null}} \qquad \text{(R-WHILEF)}$$

$$\frac{h(o).f = (\mathsf{Bool.T}\ \mathsf{then}\ g = o') \qquad h(o').\mathsf{session} = \oplus\{\mathsf{Bool.T}\colon S, \ldots\}}{h;\ \mathsf{while}\ (o.f)\ \{e\} \longrightarrow h + o.f = \mathsf{null} + o.g = o' + o'.\mathsf{session} = S;\ (e; \mathsf{while}\ (o.f)\ \{e\})} \qquad \text{(R-WHILET)}$$

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

Figure 11. Reduction rules

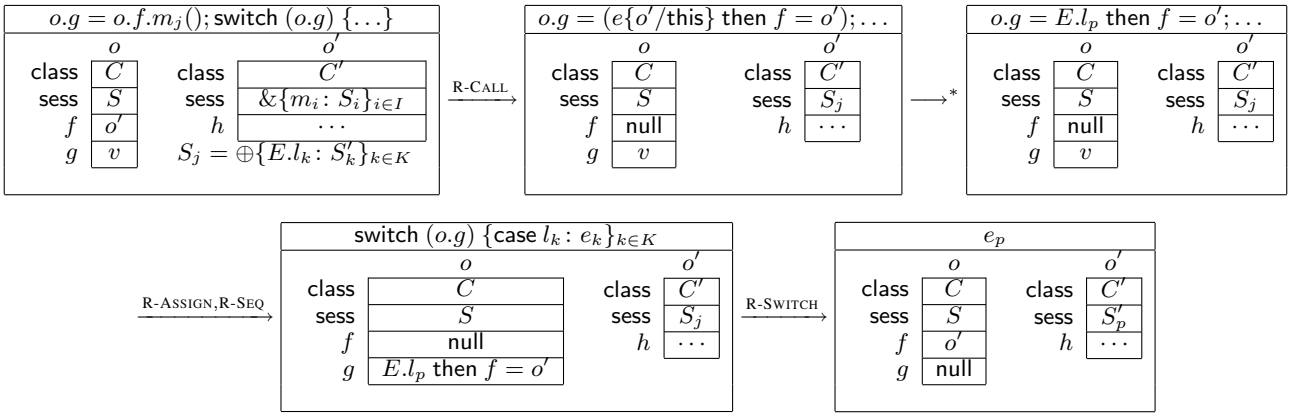

Figure 12. Example of the interplay between method call, switch and the then expression

The direction of subtyping is opposite to that defined 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, 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 (5). Further details, including the proof that subtyping is reflexive and transitive and an algorithm for checking subtyping, can easily be adapted from (19).

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 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 $\vdash \Gamma \triangleright e : T \triangleleft \Gamma'$. 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 $h(o) = C[S; \vec{T}\ \vec{f}]$ then $h(o).f_i$ means T_i. $\Gamma + o : T$ means disjoint union. If $\Gamma(o)$ is defined and has field f (this is an implicit hypothesis) then the notation $\Gamma + o.f : T$ means the environment obtained by changing the type of field f in object o 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.

T-CALL requires an environment in which method $o.f.m$ 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.

T-SWITCH types a switch on $o.f$, whose type must have a $\oplus$ session type and a variant field typing indexed by the same enum E. In the typing of branch e_i, the index i selects from both the session type and the variant type. All branches must have the same

$$\vdash \Gamma \triangleright \mathsf{null} : \mathsf{Null} \triangleleft \Gamma \qquad \vdash \Gamma + o : T \triangleright o : T \triangleleft \Gamma \tag{T-NULL,T-REF}$$

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

$$\frac{\vdash \Gamma \triangleright e : T \triangleleft \Gamma' + o : U}{\vdash \Gamma \triangleright e \;\mathsf{then}\; f = o : T \;\mathsf{then}\; U\; f \triangleleft \Gamma'} \qquad \vdash \Gamma \triangleright \mathsf{new}\; C() : C[\mathit{session}(C); \vec{\mathsf{Null}}\, \mathit{fields}(C)] \triangleleft \Gamma \tag{T-THEN,T-NEW}$$

$$\frac{\vdash \Gamma \triangleright o : T \triangleleft \Gamma'}{\vdash \Gamma \triangleright o.f : T.f \triangleleft \Gamma' + o.f : \mathsf{Null}} \qquad \frac{\vdash \Gamma \triangleright e : T \triangleleft \Gamma'}{\vdash \Gamma \triangleright o.f = e : \mathsf{Null} \triangleleft \Gamma' + o.f : T} \tag{T-FIELD,T-ASSIGN}$$

$$\frac{\Gamma(o).f = T_1 \quad M \in \mathit{methods}(T_1.\mathsf{class}) \quad \mathsf{req}\; T_1\; \mathsf{ens}\; T_2\; \mathsf{for}\; T\; m()\; \{e\} <: M}{\vdash \Gamma \triangleright o.f.m() : (T \;\mathsf{then}\; T_2\; f) \triangleleft \Gamma + o.f : \mathsf{Null}} \tag{T-CALL}$$

$$\frac{\begin{array}{r}\Gamma(o).f <: E \;\mathsf{then}\; C[\oplus\{E.l_i : S_i\}_{i\in I}; \langle E.l_i : F_i\rangle_{i\in I}]\; g \\ \forall i.l_i \in \mathit{constants}(E) \quad \vdash \Gamma + o.f : \mathsf{Null} + o.g : C[S_i; F_i] \triangleright e_i : T \triangleleft \Gamma' \quad (\forall i \in I)\end{array}}{\vdash \Gamma \triangleright \mathsf{switch}\; (o.f)\; \{\mathsf{case}\; l_i : e_i\}_{i\in I} : T \triangleleft \Gamma'} \tag{T-SWITCH}$$

$$\frac{\Gamma(o).f <: \mathsf{Bool} \;\mathsf{then}\; C[\oplus\{\mathsf{Bool.F} : S_f, \mathsf{Bool.T} : S_t\}; \langle \mathsf{Bool.F} : F_f, \mathsf{Bool.T} : F_t\rangle]\; g \quad \vdash \Gamma + o.f : \mathsf{Null} + o.g : C[S_t; F_t] \triangleright e : \_ \triangleleft \Gamma}{\vdash \Gamma \triangleright \mathsf{while}\; (o.f)\; \{e\} : \mathsf{Null} \triangleleft \Gamma + o.f : \mathsf{Null} + o.g : C[S_f; F_f]} \tag{T-WHILE}$$

$$\frac{\vdash \Gamma \triangleright e : \_ \triangleleft \Gamma'' \qquad \vdash \Gamma'' \triangleright e' : T \triangleleft \Gamma'}{\vdash \Gamma \triangleright e; e' : T \triangleleft \Gamma'} \qquad \frac{\vdash \Gamma \triangleright e : T \triangleleft \Gamma' \qquad T <: U}{\vdash \Gamma \triangleright e : U \triangleleft \Gamma'} \tag{T-SEQ,T-SUB}$$

Figure 14. Typing rules for expressions

$$\frac{\vdash \mathsf{this} : C[S_j; F] \triangleright e : E \triangleleft \mathsf{this} : C[S_j; \langle E.l_i : F_i\rangle_{i\in I}] \qquad j \in I}{\vdash \mathsf{req}\; C[\&\{m_i : S_i\}_{i\in I}; F]\; \mathsf{ens}\; C[S_j; \langle E.l_i : F_i\rangle_{i\in I}]\; \mathsf{for}\; E\; m_j()\; \{e\}} \qquad \frac{\vdash M_i \quad (\forall i)}{\vdash \mathsf{class}\; \_\; \{\_; \_; \vec{M}\}} \tag{T-METH,T-CLASS}$$

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

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

Figure 15. Typing rules for programs

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

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

$$\frac{T <: T' \qquad U <: U'}{T \;\mathsf{then}\; U\; f <: T' \;\mathsf{then}\; U'\; f} \tag{S-THEN}$$

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

$$\frac{T_i <: T_i' \quad (\forall i)}{F, \vec{T}\vec{f} <: \vec{T}'\vec{f}} \qquad \frac{I \subseteq J \quad E <: E' \quad F_i <: F_i' \quad (\forall i \in I)}{\langle E.l_i : F_i\rangle_{i\in I} <: \langle E'.l_j : F_i'\rangle_{j\in J}} \tag{S-FIELDS,S-VARIANT}$$

$$\frac{T <: T' \quad U <: U' \quad V <: V'}{\mathsf{req}\; T\; \mathsf{ens}\; U\; \mathsf{for}\; V\; m()\; \{\_\} <: \mathsf{req}\; T'\; \mathsf{ens}\; U'\; \mathsf{for}\; V'\; m()\; \{\_\}} \tag{S-METHOD}$$

Figure 13. Subtyping rules for types and method signatures

final environment Γ', so that it is a consistent final environment for the switch expression.

$$\vdash \emptyset \triangleright \emptyset \qquad \frac{\vdash h \triangleright \Gamma + o : T}{\vdash h \triangleright \Gamma} \tag{T-HEMPTY,T-HWEAK}$$

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

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

Figure 16. Typing rules for heaps and states

T-WHILE is based on the idea that a while loop is a switch with two branches. The terminating branch is effectively 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 rule is T-METH, which checks that a method body has the effect specified by the req and ens declarations.

Figure 16 defines rules for typing heaps and states (runtime configurations). The typing of a heap, $\vdash\ h \triangleright \Gamma$, 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.

4.5 Results

By standard techniques (41) adapted to typing judgements with initial and final environments (20) we can prove the usual results: Type Preservation and Type Safety.

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

DEFINITION 2 (Stuck States). *If* e *is neither a value nor a variable and* $\neg\exists h', e'.(h; e \longrightarrow h'; e')$ *then the state* $h; e$ *is* stuck.

A state is stuck if either the left-hand side of the reduction rule for its syntactic category is undefined, or if some premise does not hold. To prove that the language is type safe, we prove the following by induction on the structure of e.

THEOREM 2 (Type Safety). *If* $\vdash \Gamma \triangleright h; e : T \triangleleft \Gamma'$ *then* $h; e$ *is* not stuck.

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 $\mathsf{req}\ C[\&\{m_i\colon S_i\}_{i\in I}]\ \mathsf{ens}\ C[\Delta(S_i)]\ \mathsf{for}\ T\ m_i\{e\}$. 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 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 another set of annotations for the methods of C, in the form $\mathsf{req}\ C[\&\{m_i\colon S_i\}_{i\in I}; F]\ \mathsf{ens}\ C[S_j; F_j]\ \mathsf{for}\ T\ m_j()\ \{e\}$. 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 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]
```

Algorithm $\mathcal{P}$

$\mathcal{P}(C, \&\{m_i\colon S_i\}_{i\in I}, \Delta) =$
$\quad \{\mathsf{req}\ C[\&\{m_i\colon S_i\}_{i\in I}]\ \mathsf{ens}\ C[\Delta(S_i)]\ \mathsf{for}\ T\ m_i\{e\}$
$\quad \mid T\ m_i()\{e\} \in methods(C)\} \cup \bigcup_{j\in I} \mathcal{P}(C, S_j, \Delta)$
$\mathcal{P}(C, \oplus\{E.l_i\colon S_i\}_{i\in I}, \Delta) = \bigcup_{i\in I} \mathcal{P}(C, S_i, \Delta)$
$\mathcal{P}(C, \mu X.S, \Delta) = \mathcal{P}(C, S, \Delta \cup \{X \mapsto \mu X.S\})$

Algorithm $\mathcal{A}$

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

Algorithm $\mathcal{B}$

$\mathcal{B}(F, \mathsf{null}) = (F, \mathsf{Null})$
$\mathcal{B}(F, E.l) = (\langle E.l\colon F\rangle, E)$
$\mathcal{B}(F, \mathsf{new}\ C()) = (F, C[session(C)])$
$\mathcal{B}(F, \mathsf{this}.f) = (F + f : \mathsf{Null}, F(f))$
$\mathcal{B}(F, \mathsf{this}.f\ =\ e) = (F' + \mathsf{this}.f : T, \mathsf{Null})$
$\quad$ where $(F', T) = \mathcal{B}(F, e)$
$\mathcal{B}(F, \mathsf{this}.f.m()) = (F + f : \mathsf{Null}, T\ \mathsf{then}\ C[S']\ f)$
$\quad$ where $F(f) = C[S]$ and
$\quad \mathsf{req}\ C[S]\ \mathsf{ens}\ C[S']\ \mathsf{for}\ T\ m_j()\ \{e\} \in methods(\mathcal{P}(C))$
$\mathcal{B}(F, \mathsf{switch}\ (\mathsf{this}.f)\ \{\mathsf{case}\ l_i\colon e_i\}_{i\in I}) = (\biguplus_{i\in I} F_i', T)$
$\quad$ where $F(f) = (T\ \mathsf{then}\ C[\oplus\{E.l_i\colon S_i\}_{i\in I}]\ g)$ and
$\quad \forall i \in I.((F_i', T) = \mathcal{B}(F + f : \mathsf{Null} + g : C[S_i], e_i))$
$\mathcal{B}(F, \mathsf{while}\ (\mathsf{this}.f)\ \{e\}) = (F_f + f : \mathsf{Null} + g : C[S_f], \mathsf{Null})$
$\quad$ where $F(f) = (\mathsf{Bool}\ \mathsf{then}\ C[\oplus\{\mathsf{Bool.F} : S_f, \mathsf{Bool.T} : S_t\}])$
$\quad$ and $(F, T) = \mathcal{B}(F_t, e)$
$\mathcal{B}(F, e; e') = \mathcal{B}(\vec{m}, F', e')$ where $(F', \_) = \mathcal{B}(\vec{C}, F, e)$

Combining partial variants

$F \uplus F = F$ if F is not a variant typing
$\langle E.l_i\colon F_i\rangle_{i\in I} \uplus \langle E.m_j\colon G_j\rangle_{j\in J} = \langle E.l_k\colon H_k\rangle_{k\in I\cup J}$
$\quad$ where $\forall i \in I.(H_i = F_i), \forall j \in J.(H_j = G_j)$,
$\quad$ and whenever $l_i = l_j$ we have $F_i = G_j$

Figure 17. Algorithm

```
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. These annotations 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() {...}

req FileReader[Final; FileReadToEnd[end] f]
ens FileReader[Final; FileReadToEnd[end] f]
```

```
String toString() {...}
```

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 $\mathcal{A}(C, \vec{\mathsf{Null}}\vec{f}, S, \emptyset)$ 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 $\mathcal{A}(C, \vec{\mathsf{Null}}\vec{f}, S, \emptyset)$, 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. Related Work

Cyclone, Vault, CQual, Fugue, Sing#. *Cyclone* (21; 22), *Vault* (9; 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* (17; 10) 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. The main novelty of our work is the use of the session type of a class as a global specification, to introduce a dependency between the result of a method and the subsequent abstract state of the object, and to characterize the subtyping relation.

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. Like our system, *Sing#* is object-oriented, but it uses dynamic interfaces only in relation to communication channels, and not with classes in general. Although inheritance of contracts is mentioned very briefly in (15), details do not seem to have been published; in our system the use of the session type as a global specification is significant in describing inheritance and subtyping. 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.

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. Albeit quite expressive, their system does not seem to allow expressing branching on method results.

Non-Uniform Concurrent Objects / Active Objects. Another related line of research was started by Nierstrasz (31), 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 including Ravara *et al.* (38; 39), Puntigam and Peter (35; 36), and Caires (4). Damiani, Giachino, et.al. (8), propose a Java-like concurrent language incorporating inheritance and subtyping and equipped with a type-and-effect system, where method availability is made dependent on the state of objects. Our language is sequential, but we go beyond all existing work on non-uniform objects referred above by describing client branching on method results. Caires' work has finer-grained control of resources.

Unique Ownership of Objects. Our system treats objects with dynamic interfaces linearly in a straightforward way to prevent the creation of aliases and thus avoid the problem of maintaining consistency between different views of an object. This idea goes back at least as far as (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), Noble *et al.*'s *flexible alias protection* (32), Clarke *et al.*'s *ownership types* (7), Fähndrich and DeLine's *adoption and focus* (16), Östlund *et al.*'s Joe_3 (33) among others.

We have developed our type system for dynamic interfaces, and the theory of inheritance, using the simplest possible approach to alias control. We hope that it will be possible to integrate a more sophisticated alias analysis, such as those mentioned above, into our system. The key 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.

Session Types for Object-Oriented Languages. Several recent papers by Dezani-Ciancaglini, Yoshida *et al.* (11; 12; 13; 14; 26) have combined session types, as specifications of protocols on communication channels, with the object-oriented paradigm. This work focuses on communication channels and does not address the more general question of dynamic interfaces. Their approach is also different from that of *Sing#*. Instead of defining a contract for a channel, with methods for sending and receiving, a channel is created and used entirely within a single method. Bounded polymorphism, which includes subtyping, is studied in (14) but it seems orthogonal to the question of inheritance and subclasses.

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. Rajamani *et al.*'s *Behave* (6; 37) uses CCS to describe properties of π-calculus programs, and verify the validity of temporal formulae via a combination of type-checking and model-checking techniques, thereby going beyond static analysis. However, all of this work is based on π-calculus; it is not obvious how to use these results to directly tackle the problem of analysis of object-oriented programs.

7. 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 general methods. It should not be difficult to allow methods to return arbitrary values, not just enumerated values, meaning that session types would not have to alternate between $\&$ and $\oplus$. Supporting method parameters is very straightforward.

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.

Self calls. For technical reasons, the present version of our work does not allow self calls ($\mathsf{this}.m(..)$). One solution could be to distinguish between *private* and *public* session types, related by hiding some methods; self calls would be allowed for private methods.

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 $\mathsf{implements}$ interface I then we should have $session(C) <: session(I)$, interpreting the interface as a specification of minimum method availability.

Concurrency. We believe that by extending our system to a concurrent language with channel-based communication, we can unify dynamic interfaces, the *Sing#* (15) version of session types, and other work on object-oriented session types (12; 13; 14; 26). It will also be interesting to look at RMI for dynamic interfaces.

Specifications involving several objects. Multi-party session types (3; 24) 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.

8. 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

Simon Gay was partially supported by the Security and Quantum Information Group, Instituto de Telecomunicações, Portugal, and by the UK EPSRC grants "Engineering Foundations of Web Services: Theories and Tool Support" (EP/E065708/1) and "Behavioural Types for Object-Oriented Languages" (EP/F037368/1). He thanks the University of Glasgow for the sabbatical leave during which part of this research was done.

António Ravara and Vasco T. Vasconcelos were partially supported by FEDER, 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).

António Ravara was also partially supported by the UK EPSRC grant "Behavioural Types for Object-Oriented Languages" (EP/F037368/1).

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), 1995.

[3] E. Bonelli and A. Compagnoni. Multipoint Session Types for a Distributed Calculus. In G. Barthe and C. Fournet, editors, *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, distributed services, and resources. *TGC, Springer LNCS*, 4661:98–115, 2006.

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

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

[7] 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.

[8] F. Damiani, E. Giachino, P. Giannini, and S. Drossopoulou. A type safe state abstraction for coordination in Java -like languages. *Acta Informatica*, October 2008.

[9] 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.

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

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

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

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

[14] S. Drossopoulou, M. Dezani-Ciancaglini, and M. Coppo. Amalgamating the session types and the object oriented programming paradigms. Manuscript, 2007.

[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 Press, 2006.

[16] 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.

[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. 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.

[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, Department of Computing Science, University of Glasgow, March 2003.

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

[22] 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.

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

[24] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In G. C. Necula and P. Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, pages 273–284. ACM, 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] O. Nierstrasz. Regular types for active objects. In *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.

[32] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. *ECOOP, Springer LNCS*, 1445:158–185, 1998.

[33] 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.

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

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

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

[37] S. K. Rajamani and J. Rehof. A behavioral module system for the pi-calculus. *SAS, Springer LNCS*, 2126:375–394, 2001.

[38] 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.

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

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

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