HAL open science

Modular session types for objects

Simon J. Gay, Nils Gesbert, António Ravara, Vasco Thudichum Vasconcelos

To cite this version:

Simon J. Gay, Nils Gesbert, António Ravara, Vasco Thudichum Vasconcelos. Modular session types for objects. Logical Methods in Computer Science, 2015, 4 (12), pp.76. 10.2168/LMCS-11(4:12)2015 . hal-00700635v4

HAL Id: hal-00700635

https://hal.science/hal-00700635v4

Submitted on 8 Oct 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NoDerivatives 4.0 International License

MODULAR SESSION TYPES FOR OBJECTS

SIMON J. GAY, NILS GESBERT, ANTÓNIO RAVARA, AND VASCO T. VASCONCELOS

University of Glasgow
e-mail address: Simon.Gay@glasgow.ac.uk

Grenoble INP – Ensimag
e-mail address: Nils.Gesbert@grenoble-inp.fr

Universidade Nova de Lisboa
e-mail address: aravara@fct.unl.pt

Universidade de Lisboa
e-mail address: vmvasconcelos@ciencias.ulisboa.pt

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

1. Introduction

Computing infrastructure has become inherently concurrent and distributed, from the internals of machines, with the generalisation of many- and multi-core architectures, to data storage and sharing solutions, with "the cloud". Both hardware and software systems are now not only distributed but also collaborative and communication-centred. Therefore, the precise specification of the protocols governing the interactions, as well as the rigorous verification of their correctness, are critical factors to ensure the reliability of such infrastructures.

Software developers need to work with technologies that may provide correctness guarantees and are well integrated with the tools usually used. Since Java is one of the most widely used programming languages, the incorporation of support to specify and implement

LOGICAL METHODS
IN COMPUTER SCIENCE
DOI:10.2168/LMCS-???
© Simon J. Gay, Nils Gesbert, António Ravara, and Vasco T. Vasconcelos
Creative Commons

correct software components and their interaction protocols would be a step towards more reliable systems.

Behavioural types represent abstractly and concisely the interactive conduct of software components. These kind of types are simple yet expressive languages, that characterise the permitted interaction within a distributed system. The key idea is that some aspects of dynamic behaviour can be verified statically, at compile-time rather than at run-time. In particular, when working with a programming language equipped with a behavioural type system, one can statically ensure that an implementation of a distributed protocol, specified with types, is not only safe in the standard sense ("will not go wrong"), but also that the sequences of interactions foreseen by the protocol are realizable by its distributed implementation. Thorough descriptions of the state-of-the-art of research on the topic have been prepared by COST Action IC1201 (Behavioural Types for Reliable Large-Scale Software Systems, "BETTY") [3, 46].

The problem we address herein is the following: can one specify the full interactive behaviour of a given protocol as a collection of types and check that a Java implementation of that protocol realises safely such behaviour?

The solution we present works like this: first, specify as a session type (a particular idiom of behavioural types) the behaviour of each party involved in the protocol; second, using each such term to type a class, implement the protocol as a (distributed, using channel-based communication where channels are objects) Java program; third, simply compile the code and if the type checker accepts it, then the code is safe and realises the protocol. Details follow.

Session types [43, 70] allow communication protocols to be specified type-theoretically, so that protocol implementations can be verified by static type checking. The underlying assumption is that we have a concurrent or distributed system with bi-directional point-to-point communication channels. These are implemented in Java by TCP/IP socket connections. A session type describes the protocol that should be followed on a particular channel; that is to say, it defines the permitted sequences, types and directions of messages. For example, the session type $S = !\,[\mathsf{Int}]\,.\,?\,[\mathsf{Bool}]\,.\,\mathsf{end}$ specifies that an integer must be sent and then a boolean must be received, and there is no further communication. More generally, branching and repetition can also be specified. A session type can be regarded as a finite-state automaton whose transitions are annotated with types and directions, and whose language defines the protocol.

Session types were originally formulated for languages closely based on process calculus. Since then, the idea has been applied to functional languages [38, 39, 58, 64, 73], the process-oriented programming language Erlang [55], component-based object systems [72], object-oriented languages [17, 27, 28, 29, 30, 45], operating system services [31], and more general service-oriented systems [18, 23]. Session types have also been generalised from two-party to multi-party systems [11, 42], although in the present paper we will only consider the two-party case.

In previous work [36] we proposed a new approach to combining session-typed communication channels and distributed object-oriented programming. Our approach extends earlier work and allows increased programming flexibility. We adopted the principle that it should be possible to store a channel in a field of an object and allow the object's methods to use the field like any other; we then followed the consequences of this idea. For example, consider a field containing a channel of type S above, and suppose that method m sends the integer and method n receives the boolean. Because the session type of the channel requires

the send to occur first, it follows that m must be called before n. We therefore need to work with *non-uniform objects*, in which the availability of methods depends on the state of the object: method n is not available until after method m has been called. In order to develop a static type system for object-oriented programming with session-typed channels, we use a form of typestate [69] that we have previously presented under the name of *dynamic interfaces* [74]. In this type system, the availability of a class's methods (i.e., the possible sequences of method calls) is specified in a style that itself resembles a form of session type, giving a pleasing commonality of notation at both the channel and class levels.

The result of this combination of ideas is a language that allows a natural integration of programming with session-based channels and with non-uniform objects. In particular, the implementation of a session can be *modularized* by dividing it into separate methods that can be called in turn. This is not possible in SJ [45], the most closely related approach to combining sessions and objects (we discuss related work thoroughly in Section 9). We believe that we have achieved a smooth and elegant combination of three important high-level abstractions: the object-oriented abstraction for structuring computation and data, the typestate abstraction for structuring state-dependent method availability, and the session abstraction for structuring communication.

Contributions In the present paper we formalize a core *distributed class-based object-oriented* language with a static type system that combines session-typed channels and a form of typestate. The language is intended to model programming with TCP/IP sockets in Java. The formal language differs from that introduced in our previous work [36] by using structural rather than nominal types. This allows several simplifications of the type system. We have also simplified the semantics, and revised and extended the presentation. We prove that static typing guarantees two runtime safety properties: first, that the sequence of method calls on every non-uniform object follows the specification of its class's session type; second, as a consequence (because channel operations are implemented as method calls), that the sequence of messages on every channel follows the specification of its session type. This paper includes full statements and proofs of type safety, in contrast to the abbreviated presentation in our conference paper. We also formalize a type checking algorithm and prove its correctness, again with a revised and expanded presentation in comparison with the conference paper.

There is a substantial literature of related work, which we discuss in detail in Section 9. Very briefly, the contributions of our paper are the following.

- In contrast to other work on session types for object-oriented languages, we do not require a channel to be created and completely used (or delegated) within a single method. Several methods can operate on the same channel, thus allowing effective encapsulation of channels in objects, while retaining the usual object-oriented development practice. This is made possible by our integration of channels and non-uniform objects. This contribution was the main motivation for our work.
- In contrast to other typestate systems, we use a global specification of method availability, inspired by session types, as part of a class definition. This replaces pre- and post-condition annotations on method definitions, except in the particular case of recursive methods.
- When an object's typestate depends on the *result* (in an enumerated type) of a method call, meaning that the result must be case-analyzed before using the object further, we do not force the case-analysis to be done immediately by using a combined

```
class File {
  session Init
  where Init  = { {OK, ERROR} open(String): ⟨OK: Open, ERROR: Init⟩}
        Open  = { {TRUE, FALSE} hasNext(): ⟨TRUE: Read, FALSE: Close⟩,
                  Null close(): Init}
        Read  = {String read(): Open, Null close(): Init}
        Close = {Null close(): Init}
  open(filename) {...}
  hasNext() {...}
  read() {...}
  close() {...}
}
```

Figure 1: A class describing a file in some API

"switch-call" primitive. Instead, the method result can be stored in a field and the case-analysis can happen at any subsequent point. Although this feature significantly increases the complexity of the formal system and could be omitted for simplicity, it supports a natural programming style and gives more options to future programming language designers.

- Our structural definition of subtyping provides a flexible treatment of relationships between typestates, which can also support inheritance; this is discussed further in Section 4.7.

The remainder of the paper is structured as follows. In Section 2 we illustrate the concept of dynamic interfaces by means of a sequential example. In Section 3 we formalize a core sequential language and in Section 4 we describe some extensions. In Section 5 we extend the sequential example to a distributed setting and in Section 6 we extend the formal language to a core distributed language. In Section 7 we state and prove the key properties of the type system. In Section 8 we present a type checking algorithm and prove its soundness and completeness, and describe a prototype implementation of a programming language based on the ideas of the paper. Section 9 contains a more extensive discussion of related work; Section 10 outlines future work and concludes.

2. A Sequential Example

A file is a natural example of an object for which the availability of its operations depends on its state. The file must first be opened, then it can be read repeatedly, and finally it must be closed. Before reading from the file, a test must be carried out in order to determine whether or not any data is present. The file can be closed at any time.

There is a variety of terminology for objects of this kind. Ravara and Vasconcelos [67] refer to them as *non-uniform*. We have previously used the term *dynamic interface* [74] to indicate that the interface, i.e. the set of available operations, changes with time. The term *typestate* [69] is also well established.

Figure 1 defines the class File, which we imagine to be part of an API for using a file system. The definition does not include method bodies, as these would typically be implemented natively by the file system. What it does contain is method signatures and, crucially, a session type definition which specifies the availability of methods. We will refer to a skeleton class definition of this kind as an *interface*, using the term informally to mean

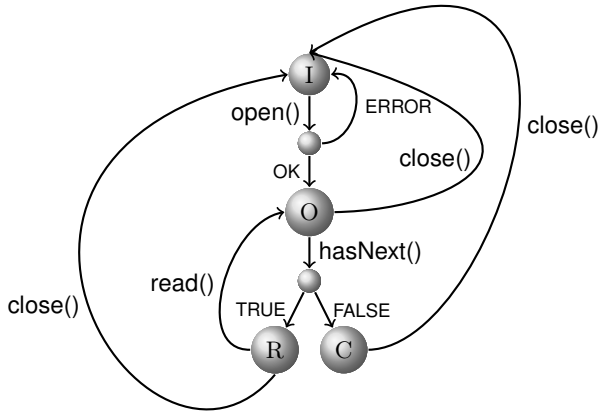

Figure 2: Diagrammatic representation of the session type of class File in Figure 1

that method definitions are omitted. The figure shows the method signatures in the session type and not as part of the method definitions as is normal in many programming languages. This style is closer to the formal language that we define in Section 3.

Line 3 declares the initial session type Init for the class. This and other session types are defined on lines 4–7. We will explain them in detail; they are types of objects, indicating which methods are available at a given point and which is the type after calling a method. In a session type, the constructor {...}, which we call *branch*, indicates that certain methods are available. In this example, Init declares the availability of one method (open), states Open and Read allow for two methods each, and state Close for a single method (close). For technical convenience, the presence of data is tested by calling the method hasNext, in the style of a Java iterator, rather than by calling an endOfFile method. If desired, method hasNext could also be included in state Read.

The constructor $\langle$ ... $\rangle$, which we call *variant*, indicates that a method returns a value from an enumeration, and that the subsequent type depends on the result. For example, from state Init the only available method is open, and it returns a value from an enumeration comprising the constants (or labels) OK and ERROR. If the result is OK then the next state is Open; if the result is ERROR then the state remains Init . It is also possible for a session type to be the empty set of methods, meaning that no methods are available; this feature is not used in the present example, but would indicate the end of an object's useful life.

The session type can be regarded as a finite state automaton whose transitions correspond to method calls and results. This is illustrated in Figure 2. Notice the two types of nodes ({...} and $\langle$ ... $\rangle$) and the two types of labels in arcs (method names issuing from {...} nodes and enumeration constants issuing from $\langle$ ... $\rangle$ nodes).

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 doing initialisation that might be included in a constructor. Notice that open has the filename as a parameter. Unlike a typical file system API, creating an object of class File does not associate it with a particular file; instead this happens when open is called.

The reader might expect a declaration **void** close() rather than **Null** close(); for simplicity, we do not address procedures in this paper, instead working with the type **Null** inhabited by a single value, **null**. Methods open and hasNext return a constant from an enumeration:

```
class FileReader {                                                    1
  session Init                                                        2
  where Init = {Null init (): {Null read (String): Final}}            3
        Final = {String toString(): Final}                            4
                                                                      5
  file; text;                                                         6
                                                                      7
  init() {                                                            8
    file = new File();                                                9
    text = ""; // Evaluates to null                                  10
  }                                                                  11
  read(filename) {                                                   12
    switch (file.open(filename)) {                                   13
      case ERROR:                                                    14
        null;                                                        15
      case OK:                                                       16
        while (file.hasNext())                                       17
          text = text + file.read();                                 18
        file.close(); // Returns null                                19
    }                                                                20
  }                                                                  21
  toString() { text; }                                               22
}                                                                    23
```

Figure 3: A client that reads from a File

OK or ERROR for method open, and TRUE or FALSE for method hasNext. Enumerations are simply sets of labels, and do not need to be declared with names.

Figure 3 defines the class FileReader, which uses an object of class File. FileReader has a session type of its own, defined on lines 2–3. It specifies that methods must be called in the sequence init, read, toString, toString, ... Line 5 defines the fields of FileReader. The formal language does not require a type declaration for fields, since fields always start with type **Null**, and are initialised to value **null**. Fields are always *private* to a class, even if we do not use a corresponding keyword. Lines 7–10 define the method init, which has initialisation behaviour typical of a constructor. Lines 12–19 illustrate the **switch** construct. In this particular case the **switch** is on the result of a method call. One of the distinctive features of our language is that it is possible, instead, to store the result of the method call in a field and later **switch** on the field; we will explain this in detail later. This contrasts with, for example, *Sing#* [31], in which the call/**switch** idiom is the only possibility. The **while** loop (lines 16–17) is similar in the sense that the result of file.hasNext must be tested in order to find out whether the loop can continue, calling file.read, or must terminate. Line 21 defines the method toString which simply accesses a field.

Typechecking the class FileReader according to our type system detects many common mistakes. Each of the following code fragments contains a type error.

- ```
  file.open(filename);
  file.read() ...;
  ```

  The open method returns either OK or ERROR, and the type of file is the variant type ⟨OK: Open, ERROR: Init⟩. The tag for this variant type is the result of open. Because the type of file is a variant, a method cannot be called on it; first we must
  ```

```
class FileReadToEnd {
  session Init
  where Init  = {{OK,ERROR} open(String): ⟨OK: Open, ERROR: {Init}⟩}
        Open  = {{TRUE,FALSE} hasNext(): ⟨TRUE: Read, FALSE: Close⟩}
        Read  = {String read(): Open}
        Close = {Null close(): {Init}}
}
```

Figure 4: Interface for class FileReadToEnd

use a **switch** statement to analyse the result of open and discover which part of the variant we are in.

- ```
  switch(file.open(filename))
    case OK: text = file.read();
  ```

  Here, a **switch** is correctly used to find out whether or not the file was successfully opened. However, if file is in state Open, the read method cannot be called immediately. First, hasNext must be called, with a corresponding **switch**.

- ```
  result = file.open(filename);
  ...
  switch(result)
    case ERROR: file.close();
  ```

  In state ERROR, method close is not available (because the file was not opened successfully). The only available method is open.

- ```
  file.close();
  if (file.hasNext()) ...
  ```

  After calling close, the file is in state Init, so the method hasNext is not available. Only open is available.

Clearly, correctness of the code in Figure 3 requires that the sequence of method calls on field file within class FileReader matches the available methods in the session type of class File, and that the appropriate **switch** or **while** loops are performed when prescribed by session types of the form ⟨ ... ⟩ in class File. Our static type system, defined in Section 3, enables this consistency to be checked at compile-time. A distinctive feature of our type system is that methods are checked in a precise order: that prescribed by the session type (init, read, toString in class FileReader, Figure 3). As such the type of the private reference file always has the right type (and no further annotations—pre/post conditions—are required when in presence of non-recursive methods). Also, in order to check statically that an object with a dynamic interface such as file is used correctly, our type system treats the reference linearly so that aliases to it cannot be created. This restriction is not a problem for a simple example such as this one, but there is a considerable literature devoted to more flexible approaches to unique ownership. We discuss this issue further in Sections 4.6, 9 and 10.

In order to support *separate compilation* we require only the interface of a class, including the class name and the session type (which in turn includes the signature of each method). For example, in order to typecheck classes that are clients of FileReader, we only need its interface. Similarly, to typecheck class FileReader, which is a client of File, it suffices to use the interface for class File, thus effectively supporting typing clients of classes containing *native methods*.
  ```

Figure 4 defines the interface for a class FileReadToEnd. This class has the same method definitions as File, but the close method is not available until all of the data has been read. According to the subtyping relation defined in Section 3.4, type Init of File is a subtype of type Init of FileReadToEnd, which we express as File . Init <: FileReadToEnd.Init. Subtyping guarantees safe substitution: an object of type File . Init can be used whenever an object of type FileReadToEnd.Init is expected, by forgetting that close is available in more states. As it happens, FileReader reads all of the data from its File object and could use a FileReadToEnd instead.

3. A Core Sequential Language

We now present the formal syntax, operational semantics, and type system of a core sequential language. As usual, the formal language makes a number of simplifications with respect to the more practical syntax used in the examples in Section 2. We summarise below the main differences with what was discussed in the previous section; in Section 4, we will discuss in more detail how some usual programming idioms, which would be expected in a full programming language, can be encoded into this formal core.

- Every method has exactly one parameter. This does not affect expressivity, as multiple parameters can be passed within an object, and a dummy parameter can be added if necessary: we consider a method call of the form $f.m()$ as an abbreviation for $f.m(\mathsf{null})$.
- Field access and assignment are defined in terms of a swap operation $f \leftrightarrow e$ which puts the value of e into the field f and evaluates to the former content of f. This operation is formally convenient because our type system forbids aliasing. In Java, the expression $f = e$ computes the result of e and then both puts it into f and evaluates to it, allowing expressions such as $g = (f = e)$ which create aliases; reading a field without removing its content also allows creation of aliases. The swap operation is a combined read-write which does not permit aliasing.

  The normal assignment operation $f = e$ is an abbreviation for $f \leftrightarrow e; \mathsf{null}$ (where the sequence operator explicitly discards the former content of f) and field read as the standalone expression f is an abbreviation for $f \leftrightarrow \mathsf{null}$. They differ from usual semantics by the fact that field read is destructive and that the assigment expression evaluates to null.
- In the examples, all method signatures appearing in a branch session type indicate both a return type and a subsequent session type. In general, those types are two separate things. However, when the subsequent behaviour of the object depends on the returned value, like the case of Open in type Init on line 3 of Fig. 1, the return type is an enumerated set of labels and the subsequent session type is a variant which must provide cases for exactly these labels. To simplify definitions, we avoid this redundant specification in the formal language, and when the subsequent session type is a variant, the return type of the method is always the special type variant-tag, which indicates that the method will return a label from the variant.

3.1. **Syntax.** We separate the syntax into the top-level language (Figure 5) and the extensions required by the type system and operational semantics (Figure 6). Identifiers C, m, f and l are taken from disjoint countable sets representing names of classes, methods, fields and labels respectively. The vector arrow indicates a sequence of zero or more elements of

Class dec	D	$::=$ $\mathsf{class}\ C\ \{S; \vec{f}; \vec{M}\}$
Class session types	S	$::=$ $\{T'_i\ m_i(T_i) : S_i\}_{i \in I}$ | $\langle l : S_l \rangle_{l \in E}$ | X | $\mu X.S$
Method dec	M	$::=$ $m(x)\ \{e\}$
Types	T	$::=$ Null | S | E | $\mathsf{variant\text{-}tag}$
Label sets	E	$::=$ $\{l, \ldots, l\}$
Values	v	$::=$ null | l
Expressions	e	$::=$ v | $f.m(e)$ | x | $e; e$ | $\mathsf{switch}\ (e)\ \{l : e_l\}_{l \in E}$ | $f \leftrightarrow e$ | $\mathsf{new}\ C()$

Figure 5: Top level syntax

Types	T	$::=$ $\ldots$ | $\mathsf{link}\ f$ | $C[F]$
Field types	F	$::=$ $\{T_i\ f_i\}_{i \in I}$ | $\langle l : F_l \rangle_{l \in E}$
Values	v	$::=$ $\ldots$ | o
Paths	r	$::=$ o | $r.f$
Expressions	e	$::=$ $\ldots$ | $\mathsf{return}\ e$
Object records	R	$::=$ $C[\{f_i = v_i\}_{i \in I}]$
Heaps	h	$::=$ ε | $h, \{o = R\}$
States	s	$::=$ $(h * r\ ,\ e)$
Contexts	$\mathcal{E}$	$::=$ $[_]$ | $\mathcal{E}; e$ | $\mathsf{switch}\ (\mathcal{E})\ \{l : e_l\}_{l \in E}$ | $\mathsf{return}\ \mathcal{E}$ | $f \leftrightarrow \mathcal{E}$ | $f.m(\mathcal{E})$

The productions for types, values and expressions extend those in Figure 5. Session types may never contain types of the form $\mathsf{link}\ f$, even in the extended syntax.

Figure 6: Extended syntax, used only in the type system and semantics

the syntactic class it is above. Similarly, constructs indexed by a set denote a finite sequence. We use E to specifically denote finite sets of labels l, whereas I is any finite indexing set.

Field names always refer to fields of the current object; there is no qualified field specification $o.f$. In other words, all fields are private. Method call is only available on a field, not an arbitrary expression. This is because calling a method changes the session type of the object on which the method is called, and in order for the type system to record this change, the object must be in a specified location (field).

Conversely, there is no unqualified method call in the core language. Calling a method on the current object this, which we call a *self-call*, behaves differently from external calls with respect to typing and will be discussed as an extension in Section 4.5,

A program consists of a sequence of class declarations D. In the core language, types in a top-level program only occur in the *session* part of a class declaration: no type is declared for fields because they can vary at run-time and are always initially null, and method declarations are also typeless, as explained earlier.

A session type S corresponds to a view of an object from outside. It shows which methods can be called, and their signatures, but the fields are not visible. We refer to $\{T'_i\ m_i(T_i) : S_i\}_{i \in I}$ as a *branch* type and to $\langle l : S_l \rangle_{l \in E}$ as a *variant* type. Session type end abbreviates the empty branch type $\{\}$. 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 require contractivity so

that every session type defines some behaviour. The μ operator is a binder, giving rise, in the standard way, to notions of bound and free variables and alpha-equivalence. A type is *closed* if it includes no free variables. We denote by $T\{^U/_X\}$ the capture-avoiding substitution of U for X in T.

Value types which can occur either as parameter or return type for a method are: Null which has the single value null, a session type S which is the type of an object, or an enumerated type E which is an arbitrary finite set of labels l. Additionally, the specific return type variant-tag is used for method occurrences after which the resulting session type is a variant, and means that the method result will be the tag of the variant. The set of possible labels appears in the variant construct of the session type, so it is not necessary to specify it in the return type of the method. However, in the example code, the set of labels is written instead of variant-tag, so that the method signature shows the return type in the usual way.

The type system, which we will describe later, enforces the following restrictions on session types: the immediate components of a variant type are always branch types, and the session type in a class declaration is always a branch. This is because a variant type is used only to represent the effect of method calls and the dependency between a method result and the subsequent session type, so it only makes sense immediately within a branch type.

Figure 6 defines additional syntax that is needed for the formal system but is not used in top-level programs. This includes:

- some extra forms of types, which are used internally to type some subexpressions but cannot be the argument type or return type of a method, and thus are never written in a program;
- intermediate expressions that cannot appear in a program but arise from the operational semantics;
- syntax for the heap.

Internal types. The first internal type we add is the type link f, where f is the name of a field. This type is related to variant session types and the variant-tag type, in the way illustrated by the following example: suppose that, in some context, field f of the current object contains an object whose type is $\{\mathsf{variant\text{-}tag}\ m(\mathsf{Null}) : \langle l : S_l\rangle_{l\in E}\}$. This means that the expression $f.m(\mathsf{null})$ is allowed in this context and will both: change the abstract state of the object in f to one of the S_l, and return the label l corresponding to that particular state. Thus, there is a link between the value of the expression and the type of field f after evaluating the expression, and the type system needs to keep track of this link; to this end, the expression $f.m(\mathsf{null})$ is given, internally, the type link f, rather than variant-tag which does not contain enough information. The use of link f is also illustrated in Figure 13.

The second internal type is an alternative form of object type, $C[F]$, which has a field typing instead of a session type. Recall that in our language, all object fields are private; therefore, normally the type of an object is a session type which only refers to methods. However, an object has access to its own fields, so for typechecking a method definition, the type environment needs to provide types for the fields of the current object (this). Since the types of the fields change throughout the life of the object, the class definition is not enough to know their types at a particular point. We thus use a field typing F, which is usually a record type associating one type to each field of the object. For example, $C[\{\mathsf{Null}\ f_1; S\ f_2\}]$ represents an object of class C with exactly two fields, f_1, which currently contains a value

of type Null, and f_2 which currently contains an object in state S. Note that in $C[F]$, F must provide types for exactly all the fields of class C.

The other form of field typing is a variant field typing, which regroups several possible sets of field types indexed by labels. For example, $C[\langle l_1 : \{\mathsf{Null}\, f_1; S\, f_2\}; l_2 : \{E\, f_1; S'\, f_2\}\rangle]$ represents an object of class C, whose two fields are f_1 and f_2, and where either f_1 has type Null and f_2 has type S, or f_1 has type E and f_2 has type S', depending on the value of the label.

These field typings cannot be the type of expressions (which cannot evaluate to this); they only represent the type of the current object in type environments. The relation between field typings and session types will be discussed in Section 3.5.2.

Internal expressions, heap, and states. These other additions are used to define the operational semantics. A heap h maps object identifiers o, taken from yet another countable set of names, to object records R. We write $dom(h)$ for the set of object identifiers in h. The identifiers are values, which may occur in expressions. The operation $h, \{o = R\}$ represents adding a record for identifier o to the heap h and we consider it to be associative and commutative, that is, h is essentially an unordered set of bindings. It is only defined if $o \notin dom(h)$. *Paths* r represent locations in the heap. A path consists of a top-level object identifier followed by an arbitrary number of field specifications. We use the following notation to interpret paths relative to a given heap.

Definition 3.1 (Heap locations)**.**

- If $R = C[\{f_i = v_i\}_{i\in I}]$, we define $R.f_i = v_i$ (for all i) and $R.\mathsf{class} = C$. For any value v and any $j \in I$, we also define $R\{f_j \mapsto v\} = C[\{f_i = v'_i\}_{i\in I}]$ where $v'_i = v_i$ for $i \neq j$ and $v'_j = v$.
- If $h = (h', \{o = R\})$, we define $h(o) = R$, and for any field f of R, $h\{o.f \mapsto v\} = (h', \{o = R\{f \mapsto v\}\})$.
- If $r = r'.f$ and $h(r').f = o$, then we also define $h(r) = h(o)$ and $h\{r.f' \mapsto v\} = h\{o.f' \mapsto v\}$.
- In any other case, these operations are not defined. Note in particular that $h(r)$ is not defined if r is a path that exists in h but does not point to an object identifier.

There is a new form of expression, return e, which is used to represent an ongoing method call.

Finally, a state consists of a heap and an expression, and the operational semantics will be defined as a reduction relation on states; $\mathcal{E}$ are evaluation contexts in the style of Wright and Felleisen [76], used in the definition of reduction.

The semantic and typing rules we will present next are implicitly parameterized by the set of declarations D which constitute the program. It is assumed that the whole set is available at any point and that any class is declared only once. We do not require the sets of method or field names to be disjoint from one class to another. We will use the following notation: if class C $\{S; \vec{f}; \vec{M}\}$ is one of the declarations, $C.\mathsf{session}$ means S and $C.\mathsf{fields}$ means $\vec{f}$, and if $m(x)$ $\{e\} \in \vec{M}$ then $C.m$ is e.

3.2. **Operational Semantics.** Figure 7 defines an operational semantics on states $(h{*}r\ ,\ e)$ consisting of a heap h, a path r in the heap indicating the *current object*, and an expression e. In general, e is an expression obtained by a series of reduction steps from a method

$$(\text{R-Call})\ \frac{m(x)\ \{e\} \in h(r.f).\mathsf{class}}{(h * r\ ,\ f.m(v)) \longrightarrow (h * r.f\ ,\ \mathsf{return}\ e\{^{v}/_{x}\})} \quad (\text{R-Return})\ \ (h * r.f\ ,\ \mathsf{return}\ v) \longrightarrow (h * r\ ,\ v)$$

$$(\text{R-New})\ \frac{o \notin dom(h) \qquad C.\mathsf{fields} = \vec{f}}{(h * r\ ,\ \mathsf{new}\ C()) \longrightarrow (h, \{o = C[\vec{f} = \overrightarrow{\mathsf{null}}]\} * r\ ,\ o)}$$

$$(\text{R-Swap})\ \frac{h(r).f = v}{(h * r\ ,\ f \leftrightarrow v') \longrightarrow (h\{r.f \mapsto v'\} * r\ ,\ v)}$$

$$(\text{R-Switch})\ \frac{l_0 \in E}{(h * r\ ,\ \mathsf{switch}\ (l_0)\ \{l : e_l\}_{l \in E}) \longrightarrow (h * r\ ,\ e_{l_0})} \quad (\text{R-Seq})\ \ (h * r\ ,\ v; e) \longrightarrow (h * r\ ,\ e)$$

$$(\text{R-Context})\ \frac{(h * r\ ,\ e) \longrightarrow (h' * r'\ ,\ e')}{(h * r\ ,\ \mathcal{E}[e]) \longrightarrow (h' * r'\ ,\ \mathcal{E}[e'])}$$

Figure 7: Reduction rules for states

body, where the method was called on the object identified by the path r. All rules have the implicit premise that the expressions appearing in them must be defined. For example, $f \leftrightarrow v$ only reduces if $h(r)$ is an object record containing a field named f. An example of reduction, together with typing, is presented in Figure 13 and discussed at the end of the present section.

The current object path r is used to resolve field references appearing in the expression e. It behaves like a call stack: as shown in R-Call, when a method call on a field f (relative to the current object located at r) is entered, the object in $r.f$ becomes the current object; this is indicated by changing the path to $r.f$. Additionally, the method body, with the actual parameter substituted for the formal parameter, is wrapped in a return expression and replaces the method call. When the body has reduced to a value, this value is unwrapped by R-Return which also pops the field specification f from the path, recovering the previous current object r. This is illustrated in Figure 13, which also shows the typing of expressions in a series of reductions. R-New creates a new object in the heap, with null fields. R-Swap updates the value of a field and reduces to its former value.

R-Switch is standard. R-Seq discards the result of the first part of a sequential composition. R-Context is the usual rule for reduction in contexts.

To complete the definition of the semantics we need to define the initial state. The idea is to designate a particular method m of a particular class C as the *main* method, which is called in order to begin execution. The most convenient way to express this is to have an initial heap that contains an object of class C, which is also chosen as the current object, and an initial expression e which is the body of m. The initial state is therefore

$$(\mathsf{top} = C[C.\mathsf{fields} = \vec{\mathsf{null}}] * \mathsf{top}\ ,\ e)$$

where top is the identifier of the top-level object of class C, which is the only object in the heap, and the current object path is also top. Strictly speaking, method m must have a parameter x; we take x to be of type Null and assume that it does not occur in e.

3.3. **Example of reduction.** Assume that the top-level class is C, containing fields f and g. Assume also that there is another class C' which defines the set of methods $\{m_i \mid i \in I\}$. Finally, assume that the body of the main method of C is

$$f = \mathsf{new}\ C'(); g = f.m_j(); \mathsf{switch}\ (g)\ \{l\colon e_l\}_{l \in E}$$

$$(\mathsf{top} = C[f = \mathsf{null}, g = \mathsf{null}] * \mathsf{top}\ ,\ f = \mathsf{new}\ C'(); g = f.m_j(); \mathsf{switch}\ (g)\ \{l\colon e_l\}_{l \in E})$$
$$\downarrow (\text{Expand})$$
$$(\mathsf{top} = C[f = \mathsf{null}, g = \mathsf{null}] * \mathsf{top}\ ,\ f \leftrightarrow \mathsf{new}\ C'(); \mathsf{null}; g = f.m_j(); \mathsf{switch}\ (g)\ \{l\colon e_l\}_{l \in E})$$
$$\downarrow (\text{R-New})$$
$$(\mathsf{top} = C[f = \mathsf{null}, g = \mathsf{null}], o = C'[] * \mathsf{top}\ ,\ f \leftrightarrow o; \mathsf{null}; g = f.m_j(); \mathsf{switch}\ (g)\ \{l\colon e_l\}_{l \in E})$$
$$\downarrow (\text{R-Swap})$$
$$(\mathsf{top} = C[f = o, g = \mathsf{null}], o = C'[] * \mathsf{top}\ ,\ \mathsf{null}; \mathsf{null}; g = f.m_j(); \mathsf{switch}\ (g)\ \{l\colon e_l\}_{l \in E})$$
$$\downarrow (\text{R-Seq}, \text{R-Seq})$$
$$(\mathsf{top} = C[f = o, g = \mathsf{null}], o = C'[] * \mathsf{top}\ ,\ g = f.m_j(); \mathsf{switch}\ (g)\ \{l\colon e_l\}_{l \in E})$$
$$\downarrow (\text{Expand})$$
$$(\mathsf{top} = C[f = o, g = \mathsf{null}], o = C'[] * \mathsf{top}\ ,\ g \leftrightarrow f.m_j(); \mathsf{null}; \mathsf{switch}\ (g \leftrightarrow \mathsf{null})\ \{l\colon e_l\}_{l \in E})$$
$$\downarrow (\text{Simplify})$$
$$(\mathsf{top} = C[f = o, g = \mathsf{null}], o = C'[] * \mathsf{top}\ ,\ g \leftrightarrow f.m_j(); \mathsf{switch}\ (g \leftrightarrow \mathsf{null})\ \{l\colon e_l\}_{l \in E})$$
$$\downarrow (\text{R-Call})$$
$$(\mathsf{top} = C[f = o, g = \mathsf{null}], o = C'[] * \mathsf{top}.f\ ,\ g \leftrightarrow \mathsf{return}\ e; \mathsf{switch}\ (g \leftrightarrow \mathsf{null})\ \{l\colon e_l\}_{l \in E})$$
$$\downarrow (\text{Evaluate } e)$$
$$(\mathsf{top} = C[f = o, g = \mathsf{null}], o = C'[] * \mathsf{top}.f\ ,\ g \leftrightarrow \mathsf{return}\ l_0; \mathsf{switch}\ (g \leftrightarrow \mathsf{null})\ \{l\colon e_l\}_{l \in E})$$
$$\downarrow (\text{R-Return})$$
$$(\mathsf{top} = C[f = o, g = \mathsf{null}], o = C'[] * \mathsf{top}\ ,\ g \leftrightarrow l_0; \mathsf{switch}\ (g \leftrightarrow \mathsf{null})\ \{l\colon e_l\}_{l \in E})$$
$$\downarrow (\text{R-Swap}, \text{R-Seq})$$
$$(\mathsf{top} = C[f = o, g = l_0], o = C'[] * \mathsf{top}\ ,\ \mathsf{switch}\ (g \leftrightarrow \mathsf{null})\ \{l\colon e_l\}_{l \in E})$$
$$\downarrow (\text{R-Swap})$$
$$(\mathsf{top} = C[f = o, g = \mathsf{null}], o = C'[] * \mathsf{top}\ ,\ \mathsf{switch}\ (l_0)\ \{l\colon e_l\}_{l \in E})$$
$$\downarrow (\text{R-Switch})$$
$$(\mathsf{top} = C[f = o, g = \mathsf{null}], o = C'[] * \mathsf{top}\ ,\ e_{l_0})$$

Figure 8: A series of reduction steps.

where j is some element of I.

The initial state is

$$(\mathsf{top} = C[f = \mathsf{null}, g = \mathsf{null}] * \mathsf{top}\ ,\ f = \mathsf{new}\ C'(); g = f.m_j(); \mathsf{switch}\ (g)\ \{l\colon e_l\}_{l \in E})$$

where for simplicity we have ignored the parameter of m_j. Figure 8 shows the sequence of reduction steps until one of the cases of the switch is reached.

The first step is expansion of the syntactic sugar for assignment, translating it into a swap followed by null. Another similar translation step occurs later. The first real reduction step is R-New, creating an object o, followed by R-Swap to complete the assignment into field f and then R-Seq to tidy up. Next, the step labelled "simplify" informally removes null in order to avoid carrying it through to an uninteresting R-Seq reduction later.

Now assume that the body of method m_j is e. Reduction by R-Call changes the current object path to $\mathsf{top}.f$ because the current object is now o. Several reduction steps convert e to a particular element of the enumerated type E, which we call l_0. After that, R-Return changes the current object path back to top, and then some R-Swap and R-Seq steps bring l_0 into the guard of the switch, finally allowing the appropriate case e_{l_0} to be selected.

We will return to this example in Section 3.6, to show how each state is typed.

$$\text{(S-Record)}\ \frac{\forall i \in I, T_i <: T'_i}{\{T_i\ f_i\}_{i\in I} <: \{T'_i\ f_i\}_{i\in I}} \qquad \text{(S-Variant)}\ \frac{E \subseteq E' \qquad \forall l \in E, F_l <: F'_l}{\langle l : F_l\rangle_{l\in E} <: \langle l : F'_l\rangle_{l\in E'}}$$

$$\text{(S-Field)}\ \frac{F <: F'}{C[F] <: C[F']}$$

Figure 9: Subtyping rules for fields

3.4. **Subtyping.** Subtyping is an essential ingredient of the theory of session types. Originally proposed by Gay and Hole [37], it has been widely used in other session-based systems, with subject-reduction and type-safety holding. The guiding principle is the "safe substitutability principle" of Liskov and Wing [51], which states that, if S is a subtype of T, then objects of type T in a program may be safely replaced with objects of type S.

Two kinds of types in the top-level core language are subject to subtyping: enumerated types and session types. The internal language also has field typings; subtyping on them is derived from subtyping on top-level types by the rules in Figure 9.

Subtyping for enumerated types is defined as simple set inclusion: $E <: E'$ if and only if $E \subseteq E'$. We refer to subtyping for session types as the *sub-session* relation. Because session types can be recursive, the sub-session relation is defined coinductively, by defining necessary conditions it must satisfy and taking the largest relation satisfying them. The definition involves checking compatibility between different method signatures, which itself is dependent on the whole subtyping relation. We proceed as follows: given a candidate sub-session relation $\mathcal{R}$, we define an $\mathcal{R}$-compatibility relation between types and between method signatures which uses $\mathcal{R}$ as a sub-session relation. We then use $\mathcal{R}$-compatibility in the structural conditions that $\mathcal{R}$ must satisfy in order to effectively be a sub-session relation.

Let $\mathcal{S}$ denote the set of contractive, closed, class session types. We deal with recursive types using the following unfold operator:

Definition 3.2 (Unfolding)**.** The operator unfold is defined inductively on $\mathcal{S}$ by $\mathsf{unfold}(\mu X.S) = \mathsf{unfold}(S\{^{(\mu X.S)}/_X\})$ and $\mathsf{unfold}(S) = S$ if S is not of the form $\mu X.S$. Since the types in $\mathcal{S}$ are contractive, this definition is well-founded.

We now define the two compatibility relations we need.

Definition 3.3 ($\mathcal{R}$-Compatibility (Types))**.** Let $\mathcal{R}$ be a binary relation on $\mathcal{S}$. We say that type T is $\mathcal{R}$-compatible with type T' if one of the following conditions is true.

(1) $T = T'$
(2) T and T' are enumerated types and $T \subseteq T'$
(3) $T, T' \in \mathcal{S}$ and $(T, T') \in \mathcal{R}$.

Definition 3.4 ($\mathcal{R}$-Compatibility (Signatures))**.** Let $\mathcal{R}$ be a binary relation on $\mathcal{S}$. Let $\sigma = U\ m(T) : S$ and $\sigma' = U'\ m(T') : S'$ be components of branch types, both for the same method name m, i.e. method signatures with subsequent session types. We say that σ is $\mathcal{R}$-compatible with σ' if T' is $\mathcal{R}$-compatible with T and either:

(1) U is $\mathcal{R}$-compatible with U' and $(S, S') \in \mathcal{R}$, or
(2) U is an enumerated type E, $U' = \mathsf{variant\text{-}tag}$ and $(\langle l : S\rangle_{l\in E}, S') \in \mathcal{R}$.

The compatibility relation on method signatures is, as expected, covariant in the return type and the subsequent session type and contravariant in the parameter type, but with one addition: if a method has an enumerated return type E and subsequent session type S, then it can always be used as if it had a return type of $\mathsf{variant\text{-}tag}$ and were followed by the

uniform variant session type $\langle l : S\rangle_{l\in E}$. Indeed, both signatures mean that the method can return any label in E and will always leave the object in state S.

We can now state the necessary conditions for a sub-session relation.

Definition 3.5 (Sub-session)**.** Let $\mathcal{R}$ be a binary relation on $\mathcal{S}$. We say that $\mathcal{R}$ is a sub-session relation if $(S, S') \in \mathcal{R}$ implies:

(1) If $\mathsf{unfold}(S) = \{U_i\ m_i(T_i) : S_i\}_{i\in I}$ then $\mathsf{unfold}(S')$ is of the form $\{U'_j\ m_j(T'_j) : S'_j\}_{j\in J}$ with $J \subseteq I$, and for all $j \in J$, $U_j\ m_j(T_j) : S_j$ is $\mathcal{R}$-compatible with $U'_j\ m_j(T'_j) : S'_j$.
(2) If $\mathsf{unfold}(S) = \langle l : S_l\rangle_{l\in E}$ then $\mathsf{unfold}(S')$ is of the form $\langle l : S'_l\rangle_{l\in E'}$ with $E \subseteq E'$ and for all $l \in E$, $(S_l, S'_l) \in \mathcal{R}$.

For the sake of simplicity we will now, when we refer to this definition later on, make the unfolding step implicit by assuming, without loss of generality, that neither S nor S' is of the form $\mu X.S''$.

Lemma 3.6. The union of several sub-session relations is a sub-session relation.

Proof. Let $\mathcal{R} = \bigcup_{i\in I} \mathcal{R}_i$, where the $\mathcal{R}_i$ are sub-session relations. Let $(S, S') \in \mathcal{R}$. Then there is j in I such that $(S, S') \in \mathcal{R}_j$. This implies that (S, S') satisfies the conditions in Definition 3.5 with respect to $\mathcal{R}_j$. Just notice that, because $\mathcal{R}_j \subseteq \mathcal{R}$, the conditions are satisfied with respect to $\mathcal{R}$ as well — in particular, $\mathcal{R}_j$-compatibility implies $\mathcal{R}$-compatibility. Indeed, the conditions for $\mathcal{R}$ only differ from those for $\mathcal{R}_j$ by requiring particular pairs of session types to be in $\mathcal{R}$ rather than in $\mathcal{R}_j$, so they are looser. □

We now define the subtyping relation $<:$ on session types to be the largest sub-session relation, i.e. the union of all sub-session relations. The subtyping relation on general top-level types is just $<:$-compatibility.

Subtyping on session types means that either both are branches or both are variants. In the former case, the supertype must allow fewer methods and their signatures must be compatible; in the latter case, the supertype must allow more labels and the common cases must be in the subtyping relation. Like the definition of subtyping for channel session types [37], the type that allows a choice to be made (the branch type here, the $\oplus$ type for channels) has contravariant subtyping in the set of choices.

The following lemma shows that the necessary conditions of Definition 3.5 are also sufficient in the case of $<:$.

Lemma 3.7. (1) Let $S = \{U_i\ m_i(T_i) : S_i\}_{i\in I}$ and $S' = \{U'_j\ m_j(T'_j) : S'_j\}_{j\in J}$ with $J \subseteq I$. If for all $j \in J$, $U_j\ m_j(T_j) : S_j$ is $<:$-compatible with $U'_j\ m_j(T'_j) : S'_j$, then $S <: S'$.
(2) Let $S = \langle l : S_l\rangle_{l\in E}$ and $S' = \langle l : S'_l\rangle_{l\in E'}$ with $E \subseteq E'$. If for all l in E we have $S_l <: S'_l$, then $S <: S'$.

Proof. The relation $<: \cup \{(S, S')\}$ is a sub-session relation. □

Finally, we prove that this subtyping relation provides a preorder on types.

Proposition 3.8. The subtyping relation is reflexive and transitive.

Proof. First note that session types can only be related by subtyping to other session types; the same applies to enumerated types and, in the internal system, field typings. Since the relation for enumerated types is just set inclusion, we already know the result for it. We now prove the properties for session types; the fact that they hold for field typings is then a straightforward consequence.

For reflexivity, just notice that the diagonal relation $\{(S,S) \mid S \in \mathcal{S}\}$ is a sub-session relation, hence included in $<:$.

For transitivity, what we need to prove is that the relation $\mathcal{R} = \{(S,S') \mid \exists S'', S <: S'' \wedge S'' <: S'\}$ is a sub-session relation. Let $(S,S') \in \mathcal{R}$ and let S'' be as given by the definition of $\mathcal{R}$.

In case (1) where we have $S = \{U_i\ m_i(T_i) : S_i\}_{i\in I}$, we know that:

- $S'' = \{U''_j\ m_j(T''_j) : S''_j\}_{j\in J}$ with $J \subseteq I$, and for all $j \in J$, $\sigma_j = U_i\ m_i(T_i) : S_j$ is $<:$-compatible with $\sigma''_j = U''_j\ m_j(T''_j) : S''_j$.
- Therefore, S' is of the form $\{U'_k\ m_k(T'_k) : S'_k\}_{k\in K}$ with $K \subseteq J$, and for all $k \in K$, σ''_k is $<:$-compatible with $\sigma'_k = U'_k\ m_k(T'_k) : S'_k$.

Straightforwardly $K \subseteq I$. For every k in K, we have to prove that σ_k is $\mathcal{R}$-compatible with σ'_k. We deduce it from the two $<:$-compatibilities we know by looking into the definition of compatibility point by point:

- We have $T''_k <: T_k$ and $T'_k <: T''_k$. Either these types are all session types, and then $(T'_k, T_k) \in \mathcal{R}$ by definition of $\mathcal{R}$, or none of them is and we have $T'_k <: T_k$ by transitivity of subtyping on base types. In both cases, T'_k is $\mathcal{R}$-compatible with T_k.
- We also have either:
  - $U_k <: U''_k$, $U''_k <: U'_k$, $S_k <: S''_k$ and $S''_k <: S'_k$. In this case, the former two conditions imply, similarly to the above, that U_k is $\mathcal{R}$-compatible with U'_k. The latter two imply $(S_k, S'_k) \in \mathcal{R}$.
  - Or U_k is an enumerated type E, $U''_k = U'_k = \mathsf{variant\text{-}tag}$, $\langle l : S_k\rangle_{l\in E} <: S''_k$ and $S''_k <: S'_k$. Then we have $(\langle l : S_k\rangle_{l\in E}, S'_k) \in \mathcal{R}$, which is all we need.
  - Or, finally, U_k is an enumerated type E, U''_k is an enumerated type E'' such that $E \subseteq E''$, $S_k <: S''_k$ and $\langle l : S''_k\rangle_{l\in E''} <: S'_k$. Then from $S_k <: S''_k$ and $E \subseteq E''$ we deduce, using case (2) of Lemma 3.7, $\langle l : S_k\rangle_{l\in E} <: \langle l : S''_k\rangle_{l\in E''}$. We thus have, again, $(\langle l : S_k\rangle_{l\in E}, S'_k) \in \mathcal{R}$ which is the required condition.

In case (2) where we have $S = \langle l : S_l\rangle_{l\in E}$, we obtain $S'' = \langle l : S''_l\rangle_{l\in E''}$ and $S' = \langle l : S'_l\rangle_{l\in E'}$, with $E \subseteq E'' \subseteq E'$ and for any l in E, $S_l <: S''_l$ and $S''_l <: S'_l$, which imply by definition of $\mathcal{R}$ that (S_l, S'_l) is in $\mathcal{R}$. □

Definition 3.9 (Type equivalence)**.** We define *equivalence* of session types S and S' as $S <: S'$ and $S' <: S$. This corresponds precisely to S and S' having the same infinite unfoldings (up to the ordering of cases in branches and variants). Henceforth types are understood up to type equivalence, so that, for example, in any mathematical context, types $\mu X.S$ and $S\{^{(\mu X.S)}/_X\}$ can be used interchangeably, effectively adopting the equi-recursive approach [63, Chapter 21].

3.5. **Type System.** We introduce a static type system whose purpose is to ensure that typable programs satisfy a number of safety properties. As usual, we make use of a type preservation theorem, which states that reduction of a typable expression produces another typable expression. Therefore the type system is formulated not only for top-level expressions but for the states (i.e. (heap, expression) pairs) on which the reduction relation is defined.

An important feature of the type system is that the method definitions within a particular class are not checked independently, but are analyzed in the order specified by the session type of the class. This is expressed by rule T-Class, the last rule in Figure 10, which uses a consistency relation $\overrightarrow{\mathsf{Null}}\ \vec{f} \vdash C : S$ between field typings and session types, defined in

$$\text{(T-Null)}\ \ \Gamma * r \rhd \mathsf{null} : \mathsf{Null} \lhd \Gamma * r \qquad \text{(T-Label)}\ \ \Gamma * r \rhd l : \{l\} \lhd \Gamma * r$$

$$\text{(T-New)}\ \ \Gamma * r \rhd \mathsf{new}\ C() : C.\mathsf{session} \lhd \Gamma * r \qquad \text{(T-LinVar)}\ \ \Gamma, x : S * r \rhd x : S \lhd \Gamma * r$$

$$\text{(T-Var)}\ \frac{T \text{ is not an object type}}{\Gamma, x : T * r \rhd x : T \lhd \Gamma, x : T * r}$$

$$\text{(T-Swap)}\ \frac{\Gamma * r \rhd e : T \lhd \Gamma' * r' \qquad \Gamma'(r'.f) = T' \qquad T \neq \mathsf{variant\text{-}tag} \quad T' \text{ is not a variant}}{\Gamma * r \rhd f \leftrightarrow e : T' \lhd \Gamma'\{r'.f \mapsto T\} * r'}$$

$$\text{(T-Call)}\ \frac{\begin{array}{c}\Gamma * r \rhd e : T'_j \lhd \Gamma' * r' \qquad \Gamma'(r'.f) = \{T_i\ m_i(T'_i) : S_i\}_{i \in I} \\ j \in I \qquad T = \mathsf{link}\ f \text{ if } T_j = \mathsf{variant\text{-}tag},\ T = T_j \text{ otherwise}\end{array}}{\Gamma * r \rhd f.m_j(e) : T \lhd \Gamma'\{r'.f \mapsto S_j\} * r'}$$

$$\text{(T-Seq)}\ \frac{\Gamma * r \rhd e : T \lhd \Gamma' * r' \qquad \Gamma' * r' \rhd e' : T' \lhd \Gamma'' * r' \qquad T \neq \mathsf{link}\ _ \text{ or } \mathsf{variant\text{-}tag}}{\Gamma * r \rhd e; e' : T' \lhd \Gamma'' * r'}$$

$$\text{(T-Switch)}\ \frac{\Gamma * r \rhd e : E' \lhd \Gamma' * r' \qquad E' \subseteq E \qquad \forall l \in E', \Gamma' * r' \rhd e_l : T \lhd \Gamma'' * r'}{\Gamma * r \rhd \mathsf{switch}\ (e)\ \{l : e_l\}_{l \in E} : T \lhd \Gamma'' * r'}$$

$$\text{(T-SwitchLink)}\ \frac{\begin{array}{c}\Gamma * r \rhd e : \mathsf{link}\ f \lhd \Gamma' * r' \qquad \Gamma'(r'.f) = \langle l : S_l \rangle_{l \in E'} \\ E' \subseteq E \qquad \forall l \in E', \Gamma'\{r'.f \mapsto S_l\} * r' \rhd e_l : T \lhd \Gamma'' * r'\end{array}}{\Gamma * r \rhd \mathsf{switch}\ (e)\ \{l : e_l\}_{l \in E} : T \lhd \Gamma'' * r'}$$

$$\text{(T-VarF)}\ \frac{\Gamma * r \rhd e : E \lhd \Gamma' * r' \qquad \Gamma'(r') = C[F'] \qquad F' \text{ is a record}}{\Gamma * r \rhd e : \mathsf{variant\text{-}tag} \lhd \Gamma'\{r' \mapsto C[\langle l : F' \rangle_{l \in E}]\} * r'}$$

$$\text{(T-Sub)}\ \frac{\Gamma * r \rhd e : T \lhd \Gamma' * r' \qquad T <: T'}{\Gamma * r \rhd e : T' \lhd \Gamma' * r'} \qquad \text{(T-SubEnv)}\ \frac{\Gamma * r \rhd e : T \lhd \Gamma' * r' \qquad \Gamma' <: \Gamma''}{\Gamma * r \rhd e : T \lhd \Gamma'' * r'}$$

$$\text{(T-Class)}\ \frac{\overrightarrow{\mathsf{Null}}\ \vec{f} \vdash C : S}{\vdash \mathsf{class}\ C\ \{S; \vec{f}; \vec{M}\}}$$

Figure 10: Typing rules for the top level language

Section 3.5.2. Checking this relation requires checking the definitions of the methods occurring in S, in order. Checking method definitions uses the typing judgement for expressions, which is defined by the other rules in Figure 10.

In the following sections we describe the type system in several stages.

3.5.1. *Typing expressions.*

Definition 3.10 (Type environments)**.** We use type environments of the form $\Gamma = \alpha_1 : T_1, \ldots, \alpha_n : T_n$ where each α is either a method parameter x or an object identifier o.

The typing judgement for expressions is $\Gamma * r \rhd e : T \lhd \Gamma' * r'$. In such a judgement, Γ and Γ' are type environments and r and r' are paths. The paths parameters are in fact only necessary for typing runtime expressions; they are needed for our type preservation theorem, but not for type-checking a program, where they always have the value this. They will be discussed in Section 3.5.4.

The expression e and its type T appear in the central part of the judgement. The Γ' on the right hand side shows the change, if any, that e causes in the type environment. There are several reasons for Γ' to differ from Γ; the most important is that if e contains a method call on an object, then the session type of that object is different in Γ' than it was in Γ. Another one is linearity: if a linear parameter x is used in e, then x does not appear in Γ' because it has been consumed.

When type-checking a program (as opposed to typing a runtime expression, which needs not be implemented), the judgements for expressions always have the particular form $\mathsf{this} : C[F], V * \mathsf{this} \rhd e : T \lhd \mathsf{this} : C[F'], V' * \mathsf{this}$, with only one object identifier in the environment, this, representing the object to which fields referred to in e belong. The rest of the environment, V, is either empty or has the form $x : U$ where U is the type of the parameter x of the method currently being type-checked, and is thus a top-level type. The initial type of this is the internal type $C[F]$, where F is a field typing; the final type is $C[F']$, as e may change the types of the fields (for example, by calling methods on them). The final parameter typing V' is either the same as V or empty, depending whether the parameter was consumed by e.

We extend subtyping to a relation on type environments, as follows.

Definition 3.11 (Environment subtyping)**.** $\Gamma <: \Gamma'$ if for every $\alpha \in dom(\Gamma')$, where α is either a parameter or an object identifier, we have $\alpha \in dom(\Gamma)$ and $\Gamma(\alpha) <: \Gamma'(\alpha)$.

Essentially $\Gamma <: \Gamma'$ if Γ is more precise (contains more information) than Γ': it contains types for everything in Γ' (and possibly more) and those types are more specific.

3.5.2. *Consistency between field typings and session types.* There are two possible forms for the type of an object. One is a session type S, which describes the view of the object from outside, i.e. from the perspective of code in other classes. The session type specifies which methods may be called, but does not reveal information about the fields. The other form, $C[F]$, contains a field typing F, and describes the internal view of the object, i.e. from the perspective of code in its own methods. Consider a sequence of method calls in a particular class. There are two senses in which it may be considered correct or incorrect. (1) In the sense that it is allowed, or not allowed, by the session type of the class. (2) In the sense that each call in the sequence leaves the fields of the object in a state which ensures the next call does not produce a type error. For example, if we consider the class FileReader of Figure 3, we see that the session type allows calling read() just after init (), making the sequence init (); read() correct in sense (1). It is correct in sense (2) if and only if the body of read() typechecks under the precondition that the fields file and text have the types produced by the evaluation of init ().

In order to type a class definition, these two senses of correctness must be consistent according to the following coinductive definition.

Definition 3.12. Let C be a class and let $\mathcal{R}$ be a relation between field typings F and session types S. We say that $\mathcal{R}$ is a C-consistency relation if $(F, S) \in \mathcal{R}$ implies:

(1) If $S = \{T_i\ m_i(T'_i) : S_i\}_{i \in I}$, then F is not a variant and for all i in I, there is a definition $m_i(x_i)\ \{e_i\}$ in the declaration of class C, and a field typing F_i, such that

$$\mathsf{this} : C[F], x_i : T'_i * \mathsf{this} \rhd e_i : T_i \lhd \mathsf{this} : C[F_i] * \mathsf{this}$$

and $(F_i, S_i) \in \mathcal{R}$.

(2) If $S = \langle l : S_l \rangle_{l \in E}$, then $F = \langle l : F_l \rangle_{l \in E'}$ with $E' \subseteq E$ and for all l in E' we have $(F_l, S_l) \in \mathcal{R}$.

In clause (1), S is the session type (external view) before calling one of the methods m_i, and F is the field typing (internal view). If a particular m_i is called then the subsequent session type is S_i, and the subsequent field typing, arising from the typing judgement for the method body, is F_i. These types must be related. Clause (2) requires variant session

types and field typings, arising from a method call that returns an enumeration label, to match. The inclusion $E' \subseteq E$ allows the method to return labels from a smaller set than the one defined by the session type.

Lemma 3.13. The union of several C-consistency relations is a C-consistency relation.

Proof. Similar to (but simpler than) the proof of Lemma 3.6. □

For any class C, we define the relation $F \vdash C : S$ between field typings F and session types S to be the largest C-consistency relation, i.e. the union of all C-consistency relations.

The relation $F \vdash C : S$ represents the fact that an object of class C with internal (private) field typing F can be safely viewed from outside as having type S. Clause (2) in Definition 3.12 accounts for correspondence between variant types. The main clause is clause (1): if the object's fields have type F and its session type allows a certain method to be called, then it means that the method body is typable with an initial field typing of F and the declared type for the parameter. Furthermore, the type of the expression must match the declared return type and the final type of the fields must be compatible with the subsequent session type. The parameter may or may not be consumed by the method, but T-SubEnv at the end of Figure 10 (see next section) allows discarding it silently in any case, hence its absence from the final environment.

The definition implies that a method with return type variant-tag must be followed by a variant session type, for the following reason. Suppose that in clause (1), some T_i is variant-tag. The only way for e_i to have type variant-tag is by using rule T-VarF (discussed in the next section), which implies that F_i must be a variant field typing. The condition $(F_i, S_i) \in \mathcal{R}$ implies, by clause (1), that S_i is a variant.

The rule T-Class, last rule in Figure 10, checks that the initial session type of a class is consistent with the initial Null field typing. It refers to the above definition of consistency, which itself refers to typing judgements built using the other rules in the figure.

3.5.3. *Typing rules for top-level expressions.* The typing rules for top-level expressions (the syntax in Figure 5) are in Figure 10. They use the following notation for interpreting paths relative to type environments, analogously to Definition 3.1 for heaps.

Definition 3.14 (Locations in environments)**.**

- If $\Gamma = \Gamma', o : T$ then we define $\Gamma(o) = T$ and $\Gamma\{o \mapsto T'\} = \Gamma', o : T'$
- Inductively, if $r = r'.f$, and if $\Gamma(r') = C[F]$ where F is a record field typing containing f, then $\Gamma(r)$ is defined as $F(f)$ and $\Gamma\{r \mapsto T'\}$ as $\Gamma\{r' \mapsto C[F\{f \mapsto T'\}]\}$.
- In any other case, these operations are not defined. In particular, if $\Gamma(r')$ is defined but is a session type, then $\Gamma(r'.f)$ is not defined for any f.

A pair $\Gamma * r$ only makes sense if $\Gamma(r)$ is defined and is of the form $C[F]$.

We now comment on these rules: T-Null and T-Label type constants. A label is given a singleton enumerated type, which is the smallest type it can have, but subsumption can be used to increase its type. T-New types a new object, giving it the initial session type from the class declaration. T-LinVar and T-Var are used to access a method's parameter, removing it from the environment if it has an object type (which is linear). For simplicity, this is the only way to use a parameter. In particular, we do not allow calling methods directly on parameters: to call a method on a parameter, it must first be assigned to a field. This is just a simplification for this formal presentation and does not limit expressivity; this will be discussed in Section 4.

T-Swap types the combined read-write field access operation, exchanging the types of the field and expression. There are two restrictions on its use. T is not allowed to be variant-tag, because this particular type only makes sense as the return type of a method. This condition effectively forbids the use of rule T-VarF in the typing derivation for e, because e is not what the method returns. It also has the consequence that $\Gamma'(r')$ is not a variant type, because the only rule that could produce one is T-VarF; hence $\Gamma'(r'.f)$ is defined. The other condition is that T' is not allowed to be a variant type, because it is not allowed in our system to extract from a field a variantly-typed value without having switched on the associated tag first. Indeed, link types refer to fields by name, so moving variantly-typed values around would lose the connection between the value and its tag.

T-Call checks that field f has a session type that allows method m_j to be called. The type of the parameter is checked as usual, and the final type environment is updated to contain the new session type of the object in f. If the return type of the method is variant-tag (method open() in Figure 2 is one such example), it means that the value returned is a label describing the state of this object; since the object is in f, it is changed into link f. Because the return type appears in the session type and is therefore expressed in the top-level syntax, it cannot already be of the form link f. Observe that although types of the form link f are not written by the programmer, they can nevertheless occur in a typechecking derivation as the types of top-level expressions (as in the example in Figure 13).

T-Seq accounts for the effects of the first expression on the environment and checks that a label is not discarded, which would leave the associated variant unusable.

T-Switch types a switch whose expression e does not have a link type. All relevant branches are required to have the same type and final environment, and the whole switch expression inherits them. A typical example is if the branches just contain different labels: in that case they are given singleton types by T-Label and then T-Sub is used to give all of them the same enumerated type. If the type E' of the parameter expression is strictly smaller than the set E of case labels in the switch expression, branches corresponding to the extra cases are ignored.

T-SwitchLink is the only rule for deconstructing variants. It types a switch, similarly to the previous one, but the type of e must be a link to a field f with a variant session type. The relevant branches are then typed with initial environments containing the different case types for f according to the value of the label. As before, they must all have the same type and final environment, and if the switch expression defines extra branches for labels which do not appear in the variant type of f, they are ignored.

T-VarF constructs a variant field typing for the current object. Here E is typically, but not necessarily, a singleton type, and e is typically a literal label. The field typing before applying the rule must be a record as nested variants are not permitted, and the rule transforms it into a variant with identical cases for all labels in E. It can then be extended to a variant with arbitrary other cases using rule T-SubEnv. This rule is used for methods leading to variant session types, which, as Definition 3.12 implies, must finish with a variant field typing. As a simple example, consider the following expression, which could end a method body in some class D:

$$\mathsf{switch}\ (e)\ \{\mathsf{TRUE} : f \leftrightarrow \mathsf{new}\ C(); \mathsf{OK} \quad | \quad \mathsf{FALSE} : f \leftrightarrow \mathsf{null}; \mathsf{ERROR}\}$$

If S is the declared session type of class C, we have, using rules T-New, T-Swap, T-Label and T-Seq, the following judgements (T is just the initial type of f):

$$\mathsf{this} : D[T\ f] * \mathsf{this} \rhd f \leftrightarrow \mathsf{new}\ C(); \mathsf{OK} : \{\mathsf{OK}\} \lhd \mathsf{this} : D[S\ f] * \mathsf{this}$$

$$(\text{T-Ref})\ \frac{r \neq o.\vec{f}}{\Gamma, o : T * r \rhd o : T \lhd \Gamma * r}$$

$$(\text{T-VarS})\ \frac{\Gamma * r \rhd e : E \lhd \Gamma' * r' \qquad \Gamma'(r'.f) = S \qquad S \text{ is a branch}}{\Gamma * r \rhd e : \mathsf{link}\ f \lhd \Gamma'\{r'.f \mapsto \langle l : S\rangle_{l \in E}\} * r'}$$

$$(\text{T-Return})\ \frac{\begin{array}{c}\Gamma * r \rhd e : T \lhd \Gamma' * r'.f \qquad \Gamma'(r'.f) = C[F] \qquad F \vdash C : S \\ T \neq \mathsf{link}\ _ \qquad T' = \mathsf{link}\ f \text{ if } T = \mathsf{variant\text{-}tag},\ T' = T \text{ otherwise}\end{array}}{\Gamma * r \rhd \mathsf{return}\ e : T' \lhd \Gamma'\{r'.f \mapsto S\} * r'}$$

Figure 11: Typing rules for expressions in the internal language

$$(\text{T-Hempty})\ \vdash \varepsilon : \emptyset$$

$$(\text{T-Hadd})\ \frac{\vdash h : \Gamma \qquad \Gamma, o : C[\{\mathsf{Null}\ f_i\}_{1 \leqslant i \leqslant n}] * o \rhd f_1 \leftrightarrow v_1; \ldots; f_n \leftrightarrow v_n : \mathsf{Null} \lhd \Gamma' * o}{\vdash h, \{o = C[\{f_i = v_i\}_{1 \leqslant i \leqslant n}]\} : \Gamma'}$$

$$(\text{T-Hide})\ \frac{\vdash h : \Gamma, o : C[F] \qquad F \vdash C : S}{\vdash h : \Gamma, o : S} \qquad (\text{T-State})\ \frac{\vdash h : \Gamma \qquad \Gamma * r \rhd e : T \lhd \Gamma' * r'}{\Gamma \rhd (h * r\ ,\ e) : T \lhd \Gamma' * r'}$$

Figure 12: Typing rules for states

and

$$\mathsf{this} : D[T\ f] * \mathsf{this} \rhd f \leftrightarrow \mathsf{null}; \mathsf{ERROR} : \{\mathsf{ERROR}\} \lhd \mathsf{this} : D[\mathsf{Null}\ f] * \mathsf{this}.$$

Then T-VarF can be applied to both these judgements, giving both expressions the same type variant-tag, and giving at the same time to this, in the final environment, the variant types $D[\langle \mathsf{OK} : \{S\ f\}\rangle]$ and $D[\langle \mathsf{ERROR} : \{\mathsf{Null}\ f\}\rangle]$ respectively. These two types are both subtypes of the combined variant $D[\langle \mathsf{OK} : \{S\ f\}, \mathsf{ERROR} : \{\mathsf{Null}\ f\}\rangle]$ and T-SubEnv can thus be applied to both judgements to increase the final type of this to this common supertype. It is then possible to use T-Switch to type the whole expression. Note that the final type of the expression is always variant-tag: as T-VarF is the only rule for constructing variants, this is the only possible return type for a method leading to a variant.

T-Sub is a standard subsumption rule, and T-SubEnv allows subsumption in the final environment. The main use of the latter rule, as illustrated above, is to enable the branches of a switch to be given the same final environments.

3.5.4. *Typing rules for internal expressions, heaps and states.* The type system described so far is all we need to type check class declarations and hence programs, which are sequences of class declarations. In order to describe the runtime consequences of well-typedness, we now introduce an extended set of typing rules for expressions that occur only at runtime (Figure 11) and for program states including heaps (Figure 12). When typing an expression e as part of a runtime state, the path r, which was always this when typing programs, varies and indicates the currently active object (the one the method at the top of the call stack belongs to). Any difference between r and r' means that e contains return; in that case, r and r' represent the call stack during and after a method call. Recall that return expressions are what a method call reduces to and are introduced by R-Call and suppressed by R-Return. Therefore, these expressions can be nested and can appear, at runtime, in any part of the expression in which reduction can happen. For example they can appear in the argument of a switch or of a function call, but not in the second term of a sequence (which does not reduce until the sequence itself has reduced). In the rules of Figure 10, the difference between r and r' in some, but not all, of the premises, accounts for that fact.

Runtime expressions may contain object identifiers, typed by T-Ref. In this rule, the current object path r must not be within o, meaning that the current object or any object containing it cannot be used within an expression. This is part of the linear control of objects: somewhere there must be a reference to the object at r, in order for a method to have been called on that object, which is what gives rise to the evaluation of an expression whose current object path is r. So obtaining another reference to the object at r, within the active expression, would violate linearity.

Another new rule is T-VarS, which constructs a variant session type for a field of the current object. At the top level, the only expression capable of constructing a variant session type is a method call, but once the method call has reduced into something else this rule is necessary for type preservation.

The last additional rule for expressions is T-Return, which types a return expression representing an ongoing method call. The subexpression e represents an intermediate state in a method of object $r'.f$. If e itself does not contain return, we have $r = r'.f$; otherwise they are different. For example, if we have an expression of the form $f.m()$, the body of m is $f'.m'()$, and the body of m' is e' (omitting parameters for simplicity), then $(h * r' \,,\, f.m())$ reduces to $(h * r'.f.f' \,,\, \mathsf{return}\ (\mathsf{return}\ e'))$ in two steps of R-Call. The typing derivation for this last expression would look like:

$$\dfrac{\dfrac{\Gamma * r'.f.f' \rhd e' : T \lhd \Gamma' * r'.f.f' \qquad \begin{array}{c} F' \vdash C' : S' \\ \Gamma'(r'.f.f') = C'[F'] \end{array}}{\Gamma * r'.f.f' \rhd \mathsf{return}\ e' : T \lhd \Gamma'\{r'.f.f' \mapsto S'\} * r'.f} \qquad \begin{array}{c} F \vdash C : S \\ (\Gamma'\{r'.f.f' \mapsto S'\})(r'.f) = C[F] \end{array}}{\Gamma * r'.f.f' \rhd \mathsf{return}\ (\mathsf{return}\ e') : T \lhd \Gamma'\{r'.f \mapsto S\} * r'}$$

As e is an intermediate state in a method of $r'.f$, it is typed with final current object $r'.f$ and a final environment where the type of $r'.f$ is of the form $C[F]$, representing an inside view of the object, where the fields are visible. This T-Return rule then steps out of the object, hides its fields and changes its type into the outside view of a session type, which must be consistent with the internal type ($F \vdash C : S$). The particular case where T is variant-tag is the same as in T-Call. T is not allowed to already be of the form link f' since it would break encapsulation (f' would refer to a field of $r'.f$ which is not known outside of the object).

An important point is that the only expression that changes the current object is return. Several rules besides T-Return can inherit in the conclusion a change of current object from a subexpression in a premise, but they do not add further changes. Thus the final current object path is always a prefix of the initial one, and the number of field specifications removed is equal to the number of returns contained in the expression. Also note that the second part of a sequence and the branches of a switch are not reduction contexts; therefore, they should not contain return and are not allowed by the rules to change the current object.

As we saw in Section 3.2, a runtime state consists of a heap, a current object path, and a runtime expression. Figure 12 describes how these parts are related by typing: by rule T-State, a typing judgement for the expression gives one for the state provided the current object is the same and the initial environment reflects the content of the heap; this last constraint is represented by the judgement $\vdash h : \Gamma$. Such a judgement is constructed starting from the axiom T-HEmpty which types an empty heap and adding objects into the heap one by one with rule T-HAdd, converting their types into sessions using T-Hide as needed. As T-HAdd is the only rule that adds to Γ, we have the property that $\vdash h : \Gamma$ implies that every identifier in Γ also appears in h.

$$
\begin{array}{rcl}
\mathsf{top} : C[\{m_i : S_i\}_{i\in I}\ f, \mathsf{Null}\ g] * \mathsf{top} & \triangleright & g \leftrightarrow f.m_j(); \mathsf{switch}\ (g \leftrightarrow \mathsf{null})\ \{l\colon e_l\}_{l\in E} \\
 & \downarrow & (\textsc{R-Call}) \\
\mathsf{top} : C[C'[F]\ f, \mathsf{Null}\ g] * \mathsf{top}.f & \triangleright & g \leftrightarrow \mathsf{return}\ e; \mathsf{switch}\ (g \leftrightarrow \mathsf{null})\ \{l\colon e_l\}_{l\in E} \\
 & \downarrow * & \\
\mathsf{top} : C[C'[F_{l_0}]\ f, \mathsf{Null}\ g] * \mathsf{top}.f & \triangleright & g \leftrightarrow \mathsf{return}\ l_0; \mathsf{switch}\ (g \leftrightarrow \mathsf{null})\ \{l\colon e_l\}_{l\in E} \\
 & \downarrow & (\textsc{R-Return}) \\
\mathsf{top} : C[S_{l_0}\ f, \mathsf{Null}\ g] * \mathsf{top} & \triangleright & g \leftrightarrow l_0; \mathsf{switch}\ (g \leftrightarrow \mathsf{null})\ \{l\colon e_l\}_{l\in E} \\
 & \downarrow & (\textsc{R-Swap}, \textsc{R-Seq}) \\
\mathsf{top} : C[\langle l : S_l\rangle_{l\in E}\ f, (\mathsf{link}\ f)\ g] * \mathsf{top} & \triangleright & \mathsf{switch}\ (g \leftrightarrow \mathsf{null})\ \{l\colon e_l\}_{l\in E} \\
 & \downarrow & (\textsc{R-Swap}) \\
\mathsf{top} : C[S_{l_0}\ f, \mathsf{Null}\ g] * \mathsf{top} & \triangleright & \mathsf{switch}\ (l_0)\ \{l\colon e_l\}_{l\in E} \\
 & \downarrow & (\textsc{R-Switch}) \\
\mathsf{top} : C[S_{l_0}\ f, \mathsf{Null}\ g] * \mathsf{top} & \triangleright & e_{l_0}
\end{array}
$$

Figure 13: Example of the interplay between method call, switch and link types. The heap and the rightmost typing environment are omitted.

T-HAdd essentially says that adding a new object with given field values to the heap affects the environment in the same way as an expression that starts from an empty object and puts the values into the fields one by one. The most important feature of this rule is that whenever a v_i is an object identifier, the typing derivation for the expression has to use T-Ref, which implies both that the initial environment contains v_i and that the final one, which represents the type of the extended heap, does not. This means that a type environment corresponding to a heap never contains entries for object identifiers that appear in fields of other objects, and it also implies that a heap with multiple references to the same object is not typable. The numbering of the fields in the rightmost premise is arbitrary, meaning it must not be interpreted as requiring the sequence of swaps to be done in any particular order; all possible orders are valid instances of the premise. This is important if the type of the object being added is to contain links and variants: suppose that field f contains an object o and field g a label l; it must be possible to attribute a variant type to f and the type $\mathsf{link}\ f$ to g, but this can only be done as a result of typing the sequence of swaps if $f \leftrightarrow o$ occurs before $g \leftrightarrow l$.

3.6. **Example of reduction and typing.** We now return to the example from Section 3.3, to illustrate the way in which the environment used to type an expression changes as the expression reduces (see Theorem 3.19 on page 26). To shorten the series of steps in which the current object path does not change, Figure 13 starts from the point at which the initial expression has reduced to

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

Recall that this expression is an abbreviation for

$$g \leftrightarrow f.m_j(); \mathsf{null}; \mathsf{switch}\ (g \leftrightarrow \mathsf{null})\ \{l\colon e_l\}_{l\in E}$$

which we simplify to

$$g \leftrightarrow f.m_j(); \mathsf{switch}\ (g \leftrightarrow \mathsf{null})\ \{l\colon e_l\}_{l\in E}$$

The initial typing environment is

$$\mathsf{top} : C[\{m_i : S_i\}_{i\in I}\ f, \mathsf{Null}\ g]$$

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

$$\mathsf{this} : C'[F] * \mathsf{this} \rhd e : \mathsf{variant\text{-}tag} \lhd \mathsf{this} : C'[\langle l : F_l\rangle_{l\in E}] * \mathsf{this}$$

and we assume that m_j returns $l_0 \in E$. According to Definition 3.12 and the typing of the declaration of class C' we have $F_{l_0} \vdash C' : S_{l_0}$ and $F \vdash C' : \{m_i : S_i\}_{i\in I}$.

The figure shows the environment in which each expression is typed; the environment changes as reduction proceeds, for several reasons explained below. The typing of an expression is $\Gamma \rhd e : T \lhd \Gamma'$ but we only show Γ because Γ' does not change and T is not the interesting part of this example. We also omit the heap, showing the typing of expressions instead of states. However, an important point to keep in mind is that Γ corresponds to the typing environment obtained *after* typing the heap: $\vdash \Gamma : h$ is obtained after a number of T-HADD steps and corresponds to the final typing environment for the heap.

Calling $f.m_j()$ changes the type of field f to $C'[F]$ because we are now inside the object; the current object path changes from top to $\mathsf{top}.f$. As e reduces to l_0 the type of f may change, finally becoming $C'[F_{l_0}]$ so that it has the component of the variant field typing $C'[\langle l : F_l\rangle_{l\in E}]$ corresponding to l_0. The reduction by R-RETURN changes the type of f to S_{l_0} because we are now outside the object again, but the type is still the component of a variant typing corresponding to l_0. At this point f is popped from the current object path. Here the type of l_0 is $\mathsf{link}\ f$ (which was the expected result type for $f.m_j()$), and this type is obtained by applying T-VARS, so in the intermediate typing environment *after* typing l_0, f has the variant type $\langle l : S_l\rangle_{l\in E}$.

The next step, swap, moves l_0 from the expression to the heap. Therefore the application of T-VARS needed to type it now occurs in the derivation for typing the heap, of which Γ is the result. This is why in Γ the type of f is now $\langle l : S_l\rangle_{l\in E}$, which is S_j, the type we were expecting after the method call. At this point the information about which component of the variant typing we have is stored in $\mathsf{top}.g$, the field the label was swapped into: the type of the expression $f.m_j()$ is $\mathsf{link}\ f$, which appears as the type of $\mathsf{top}.g$ after the swap is executed. When extracting the value of g in order to switch on it, the type $\mathsf{link}\ f$ disappears from the environment and becomes the type of the subexpression $g \leftrightarrow \mathsf{null}$, at the same time resolving the variant type of f according to the particular enumerated value l_0.

3.7. **Typing the initial state.** Recall the discussion of the initial state for execution of a program, from the end of Section 3.2. The initial state is $(\mathsf{top} = C[C.\mathsf{fields} = \vec{\mathsf{null}}] * \mathsf{top}\ ,\ e)$ where class C has a designated *main* method m with body e. In order to type this initial state, we require that m is immediately available in $C.\mathsf{session}$, and assume that the program is typable, i.e. that rule T-CLASS is applicable to every class definition. If $C.\mathsf{fields} = \vec{f}$ then the hypothesis of T-CLASS is $\overrightarrow{\mathsf{Null}}\ \vec{f} \vdash C : C.\mathsf{session}$; this is what the type checking algorithm defined in Section 8 checks. The definition of $F \vdash C : S$ gives $\mathsf{this} : C[\overrightarrow{\mathsf{Null}}\ \vec{f}], x : \mathsf{Null} * \mathsf{this} \rhd e : T \lhd \mathsf{this} : C[F] * \mathsf{this}$ for some field typing F. The type T is irrelevant. Lemma 7.10 (Substitution), to be proved later, gives $\mathsf{top} : C[\overrightarrow{\mathsf{Null}}\ \vec{f}] * \mathsf{top} \rhd e : T \lhd \mathsf{top} : C[F] * \mathsf{top}$, as we assumed that $e\{^{\mathsf{null}}/_x\} = e$. Straightforward use of T-HEMPTY and T-HADD gives $\vdash \mathsf{top} = C[C.\mathsf{fields} = \vec{\mathsf{null}}] : \mathsf{top} : C[\overrightarrow{\mathsf{Null}}\ \vec{f}]$ and then T-STATE gives a typing for the initial state.

3.8. **Properties of the type system.** The main results in this sequential setting are standard: type preservation under reduction (also known as Subject Reduction) and absence of stuck states for well-typed programs. Furthermore, the system also enjoys of a conformance property: all executions of well-typed programs follow what is specified by the classes' session types.

3.8.1. *Soundness of subtyping.* In this section we prove that the subtyping relation is sound with respect to the type system, in the sense that it preserves not only typing judgements but also consistency between field typings and session typings, reflecting the safe substitution property.

Lemma 3.15. If $\Gamma * r \rhd e : T \lhd \Gamma' * r'$ and $\Gamma'' <: \Gamma$, then $\Gamma'' * r \rhd e : T \lhd \Gamma' * r'$.

Proof. Straightforward induction on the derivation. □

Proposition 3.16 (Soundness of subtyping for fields)**.** If $F \vdash C : S$ and $F' <: F$ then $F' \vdash C : S$.

Proof. Straightforward using Lemma 3.15. □

Before proving the case where subtyping is on the right, we first remark that, similarly to sub-session, the necessary conditions in the definition of C-consistency (Definition 3.12) become sufficient once we consider the largest relation:

Lemma 3.17. Let C be a class and F a field typing for that class.

(1) Suppose F is not a variant, and suppose there is a set of method definitions $\{m_i(x_i)\ \{e_i\}\}_{i\in I}$ in the declaration of class C such that, for all i, we have:

$$\mathsf{this} : C[F], x_i : T'_i * \mathsf{this} \rhd e_i : T_i \lhd \mathsf{this} : C[F_i] * \mathsf{this}$$

with $F_i \vdash C : S_i$. Then $F \vdash C : \{T_i\ m_i(T'_i) : S_i\}_{i\in I}$ holds.

(2) Suppose $F = \langle l : F_l\rangle_{l\in E'}$ and let $(S_l)_{l\in E}$ be a family of session types such that $E' \subseteq E$ and $F_l \vdash C : S_l$ for all $l \in E'$. Then $F \vdash C : \langle l : S_l\rangle_{l\in E}$ holds.

Proof. Let S be either $\{T_i\ m_i(T'_i) : S_i\}_{i\in I}$ or $\langle l : S_l\rangle_{l\in E}$ depending on the case. Just notice that $(\bullet \vdash C : \bullet) \cup \{(F, S)\}$ is a C-consistency relation. □

Proposition 3.18 (soundness of subtyping for sessions)**.** If $F \vdash C : S$ and $S <: S'$ then $F \vdash C : S'$.

Proof. For any class C, we define the following relation:

$$\mathcal{R}_C = \{(F, S') \mid \exists S,\ F \vdash C : S \text{ and } S <: S'\}$$

and prove that it is a C-consistency relation (Definition 3.12). Let $(F, S') \in \mathcal{R}_C$, and let S be as given by the definition of the relation. We have two cases depending on the form of S' (branch or variant).

The first one is $S' = \{U'_j\ m_j(T'_j) : S'_j\}_{j\in J}$. Then $S <: S'$ means (Definition 3.5) that we have: $S = \{U_i\ m_i(T_i) : S_i\}_{i\in I}$ with $J \subseteq I$ and for all $j \in J$, $U_j\ m_j(T_j) : S_j$ is $<:$-compatible with $U'_j\ m_j(T'_j) : S'_j$. Let $j \in J$, we know from $F \vdash C : S$ that C contains a method declaration $m_j(x)\ \{e\}$ such that the following judgement:

$$x : T_j, \mathsf{this} : C[F] * \mathsf{this} \rhd e : U_j \lhd \mathsf{this} : C[F_j] * \mathsf{this}$$

holds, with $F_j \vdash C : S_j$. $<:$-compatibility between the two signatures of m_j (Definition 3.4) gives us, first, $T'_j <: T_j$, which allows us to apply Lemma 3.15 to this judgement and replace T_j by T'_j in it, and second, either:

(1) $U_j <: U'_j$ and $S_j <: S'_j$. The former allows us to use T-Sub to replace U_j by U'_j in the typing judgement for e, fulfilling the first condition in the definition of C-consistency. The latter, together with $F_j \vdash C : S_j$, implies $(F_j, S'_j) \in \mathcal{R}_C$, fulfilling the second one.

(2) U_j is an enumerated type E, $U'_j = \mathsf{variant\text{-}tag}$ and $\langle l : S_j \rangle_{l \in E} <: S'_j$. In this case we first apply T-VarF to the judgement, yielding:

$$x : T'_j, \mathsf{this} : C[F] * \mathsf{this} \rhd e : \mathsf{variant\text{-}tag} \lhd \mathsf{this} : C[\langle l : F_j \rangle_{l \in E}] * \mathsf{this}.$$

From $F_j \vdash C : S_j$ we deduce $\langle l : F_j \rangle_{l \in E} \vdash C : \langle l : S_j \rangle_{l \in E}$ using Lemma 3.17, and conclude $(\langle l : F_j \rangle_{l \in E}, S'_j) \in \mathcal{R}_C$.

In the second case, where $S' = \langle l : S'_l \rangle_{l \in E'}$, then $S = \langle l : S_l \rangle_{l \in E}$ with $E \subseteq E'$ and $\forall l \in E, S_l <: S'_l$. From $F \vdash C : S$ we know that $F = \langle l : F_l \rangle_{l \in E''}$ with $E'' \subseteq E$ and $F_l \vdash C : S_l$ for any l in E''. Just notice that $E'' \subseteq E'$ (by transitivity of $\subseteq$) and $(F_l, S'_l) \in \mathcal{R}_C$ for any l in E''. □

3.8.2. *Type preservation.*

Theorem 3.19 (Subject Reduction)**.** Let $\mathcal{D}$ be a set of well-typed declarations, that is, such that for every class declaration D in $\mathcal{D}$ we have $\vdash D$.

If, in a context parameterised by $\mathcal{D}$, we have $\Gamma * r \rhd (h * r \,,\, e) : T \lhd \Gamma' * r'$, and if $(h * r \,,\, e) \longrightarrow (h' * r'' \,,\, e')$, then there exists Γ'' such that $\Gamma'' * r'' \rhd (h' * r'' \,,\, e') : T \lhd \Gamma' * r'$.

Proof. This theorem is a particular case of Theorem 7.16 which will be proved in Section 7.

3.8.3. *Type safety.*

Theorem 3.20 (No Stuck Expressions)**.** Let $\mathcal{D}$ be a set of well-typed declarations, that is, such that for every class declaration D in $\mathcal{D}$ we have $\vdash D$.

If, in a context parameterised by $\mathcal{D}$, we have $\Gamma * r \rhd (h * r \,,\, e) : T \lhd \Gamma' * r'$, then either e is a value or there exists $(h' * r'' \,,\, e')$ such that $(h * r \,,\, e) \longrightarrow (h' * r'' \,,\, e')$.

Proof. This theorem is also a consequence of Theorem 7.16, so we postpone its proof until Section 7.

3.8.4. *Conformance.* We show that, in well-typed programs, *a sequence* of method calls (interleaved with their respective return labels) of a given class *is a path* of its session type. In order to state this property precisely, we introduce a few definitions.

Definition 3.21 (Call trace)**.** A *call trace* is a sequence $m_1 l_1 m_2 l_2 \ldots m_n l_n$ in which each m_i is a method name and each l_i may be absent or, if present, is a label.

Definition 3.22 (LTS on session types)**.** Define a labelled transition relation on class session types by the following rules. α stands for m or l.

$$\frac{j \in I}{\{T'_i\ m_i(T_i) : S_i\}_{i\in I} \xrightarrow{m_j} S_j} \qquad \frac{l_0 \in E}{\langle l : S_l\rangle_{l\in E} \xrightarrow{l_0} S_{l_0}}$$

$$\frac{S \text{ is not a variant}}{S \xrightarrow{l} S} \qquad \frac{S\{^{\mu X.S}/_X\} \xrightarrow{\alpha} S'}{\mu X.S \xrightarrow{\alpha} S'}$$

Definition 3.23 (Call trace mapping)**.** A *call trace mapping* for a heap h is a function tr from $dom(h)$ to call traces.

Definition 3.24 (Validity of mappings)**.** A call trace mapping tr for a heap h is *valid* if for every entry $o = C[\ldots]$ in h, we have $C.\mathsf{session} \xrightarrow{tr(o)}{}^*$. An element in a call trace which does not allow the corresponding session type to reduce is a *type error*. (Thus a call trace is valid if and only if it does not contain type errors).

Definition 3.25. If tr is a call trace mapping for a heap h then we define $tr(h, r)$ for references r such that $h(r)$ is defined, as follows:

$$tr(h, o) = tr(o)$$
$$tr(h, r.f) = tr(h(r).f)$$

Definition 3.26 (Original reduction rule)**.** If $(h * r\ ,\ e) \longrightarrow (h' * r'\ ,\ e')$ then the derivation of this reduction consists of a number of applications of R-Context, preceded by another rule which forms a unique leaf node in the derivation. We say that the rule at the leaf node is the *original reduction rule* for the reduction, or that the reduction *originates from* this rule.

Definition 3.27 (Extension of call traces)**.** Suppose tr is a call trace mapping for h and $(h * r\ ,\ e) \longrightarrow (h' * r'\ ,\ e')$. Define a call trace mapping tr' for h' as follows:

- If the reduction originates from R-Call with method m and field f then $tr' = tr\{h(r.f) \mapsto tr(h(r.f))m\}$.
- If the reduction originates from R-Return with value v, and v is a label l, then $tr' = tr\{h(r) \mapsto tr(h(r))l\}$.
- If the reduction originates from R-New and the fresh object is o then $tr' = tr\{o \mapsto \varepsilon\}$.
- Otherwise, $tr' = tr$.

The conformance property is the following: in a sequence of reductions starting from the initial state of a well-typed program, the call traces built using the extension mechanism defined above are valid throughout the sequence. We need a couple of lemmas to properly relate call traces and typings in the case of variant types.

Lemma 3.28. If $\vdash h : \Gamma$, then Γ does not contain type variant-tag or any variant field typing.

Proof. It suffices to show that rule T-VarF cannot be used in the derivation of $\vdash h : \Gamma$, since it is the only rule that introduces variant-tag or variant field typings.

This rule can only be used on an expression of enumerated type, and the only place where such an expression can occur in the derivation of $\vdash h : \Gamma$ is as the right member of a

swap in the second premise of T-Hadd (the swap expression itself has type Null because of the initial environment). It corresponds to the first premise of T-Swap. However, the third premise of T-Swap forbids that the type of the expression be variant-tag, hence T-VarF cannot be used there. □

Lemma 3.29 (Variant consistency)**.** If $\vdash h : \Gamma$ and $\Gamma(r) = \langle l : S_l \rangle_{l \in E}$, then:

(1) r is of the form $r'.f$
(2) there exists f' such that $\Gamma(r'.f') = \mathsf{link}\ f$ and $h(r'.f') \in E$.

Proof. Consider how a variant session type can be introduced in the derivation of $\vdash h : \Gamma$. Because of Lemma 3.28, it cannot be a consequence of T-Hide: indeed, $F \vdash C : S$ where S is a variant can only hold if F is a variant as well. Thus the only possibility is T-VarS, and it can only occur when typing one of the values in the right premise of T-Hadd. (1) follows from the fact that T-VarS acts on a field of the current object. Then T-SubEnv can be applied but the original label T-VarS was applied to is still in the final E. (2) follows from the structure of the derivation: the label T-VarS is applied to is then swapped into a field of the same object. □

Definition 3.30 (Actual session type)**.** Let Γ and h be such that $\vdash h : \Gamma$. For any r in Γ such that $\Gamma(r)$ is a session type S, we define S', the *actual session type* of r in h according to Γ, as follows:

- If S is a branch then $S' = S$.
- If S is a variant $\langle l : S_l \rangle_{l \in E}$, then $S' = S_{h(r'.f')}$, where r' and f' are as given by Lemma 3.29.

Definition 3.31 (Consistency of call traces)**.** Let tr be a call trace mapping for a heap h and let Γ be a type environment such that $\vdash h : \Gamma$. We say that tr is *consistent with* Γ if for every r in Γ with actual session type S we have $\mathsf{class}(h(r)).\mathsf{session} \xrightarrow{tr(h,r)}{}^{*} S$.

Theorem 3.32 (Conformance)**.** Suppose we are in a context parameterised by a set of well-typed declarations.

Let $(h_1 * r_1\ ,\ e_1)$ be a program state together with a valid call trace mapping tr_1, and suppose that $(h_1 * r_1\ ,\ e_1) \longrightarrow \cdots \longrightarrow (h_n * r_n\ ,\ e_n)$ is a reduction sequence such that r_1 is a prefix of all r_i. Definition 3.27 gives a corresponding sequence of call traces tr_i.

If there exists Γ such that tr_1 is consistent with Γ and $\Gamma * r_1 \rhd (h_1 * r_1\ ,\ e_1) : T \lhd \Gamma' * r'$ then for all i, tr_i is valid.

Proof. Postponed, again, to Section 7 as it makes use of the proof of Theorem 7.16 which will be proved there.

Corollary 3.33. Given a well-typed program, starting from the initial state described at the end of Section 3.2 with the initial call trace mapping $\{\mathsf{top} \mapsto m\}$, and given a reduction sequence from there, the call trace mappings obtained by Definition 3.27 following the reductions are valid throughout the sequence.

Proof. We just have to see that:

(1) the initial call trace mapping is valid, as the main method m is required to appear in the initial session type of the main class;
(2) it is also consistent with the initial typing given in Section 3.7, as the initial Γ contains no session type;

Top-level syntax (add to Figure 5):

$$e \ ::= \ \ldots \ | \ \mathsf{while}\ (e)\ \{e\}$$

Reduction rule (add to Figure 7):

$$\text{(R-While)} \quad (h * r\ ,\ \mathsf{while}\ (e)\ \{e'\}) \longrightarrow (h * r\ ,\ \mathsf{switch}\ (e)\ \{\mathsf{True} : e'; \mathsf{while}\ (e)\ \{e'\}, \mathsf{False} : \mathsf{null}\})$$

Top-level typing rules (add to Figure 10):

$$\text{(T-While)}\ \frac{\Gamma * r \rhd e : \{\mathsf{True}, \mathsf{False}\} \lhd \Gamma' * r' \qquad \Gamma' * r' \rhd e' : \mathsf{Null} \lhd \Gamma * r}{\Gamma * r \rhd \mathsf{while}\ (e)\ \{e'\} : \mathsf{Null} \lhd \Gamma' * r'}$$

$$\text{(T-WhileLink)}\ \frac{\Gamma * r \rhd e : \mathsf{link}\ f \lhd \Gamma' * r' \qquad \Gamma'(r'.f) = \langle \mathsf{True} : S_{\mathsf{True}}, \mathsf{False} : S_{\mathsf{False}} \rangle \qquad \Gamma'\{r'.f \mapsto S_{\mathsf{True}}\} * r' \rhd e' : \mathsf{Null} \lhd \Gamma * r}{\Gamma * r \rhd \mathsf{while}\ (e)\ \{e'\} : \mathsf{Null} \lhd \Gamma'\{r'.f \mapsto S_{\mathsf{False}}\} * r'}$$

Figure 14: Rules for **while**

(3) the initial current object path is reduced to an object identifier and, therefore, stays a prefix of the current object path throughout any reduction sequence. □

4. Towards a Full Programming Language

In this section, we show how the core calculus presented in the previous section can be extended towards a full programming language. The extensions include constructs which can be considered abbreviations and may be translated into the core calculus without changing it, and actual extensions to the formal system.

4.1. **Assignment and Field Access.** As explained in the introduction of Section 3, we add to expressions:

- the field access expression f (not followed by a dot or by $\leftrightarrow$), which translates into the core expression $f \leftrightarrow \mathsf{null}$. This expression evaluates to the content of f and has the side effect of setting f to null;
- the assignment expression $f = e$, which translates into the core expression $f \leftrightarrow e; \mathsf{null}$. This expression stores the value of e in field f and evaluates to null.

4.2. **Multiple Parameters.** It is straightforward to generalise the reduction and typing rules so that methods have multiple parameters. In rule T-Call, the environments would be threaded through a series of parameter expressions, in the same way as in rule T-Seq.

4.3. **Local Variables.** Local variables can be simulated by introducing extra parameters.

4.4. **While Loops.** The language can easily be extended to include while loops, by adding the rules in Figure 14. The reduction rule defines while recursively in terms of switch. There are two typing rules, derived from T-Switch and T-SwitchLink. The first deals with a straightforward while loop that has no interaction with session types, and the second deals with the more interesting case in which the condition of the loop is linked to the session type of an object.

Top-level syntax (add to Figure 5) :

$$M ::= \ldots \mid \mathsf{req}\ F\ \mathsf{ens}\ F\ \mathsf{for}\ T\ m(T\ x)\ \{e\}$$
$$e ::= \ldots \mid m(e)$$

Reduction rule (add to Figure 7):

$$(\text{R-SelfCall})\ \frac{\mathsf{req}\ _\ \mathsf{ens}\ _\ \mathsf{for}\ _\ m(_\ x)\ \{e\} \in h(r).\mathsf{class}}{(h * r\ ,\ m(v)) \longrightarrow (h * r\ ,\ e\{^{v}/_{x}\})}$$

Top-level typing rules (add or replace in Figure 10):

$$(\text{T-SelfCall})\ \frac{\Gamma * r \rhd_C e : T \lhd \Gamma' * r' \qquad \Gamma'(r') = C[F] \qquad \mathsf{req}\ F\ \mathsf{ens}\ F'\ \mathsf{for}\ T'\ m(T\ x)\ \{e'\} \in C}{\Gamma * r \rhd_C m(e) : T' \lhd \Gamma'\{r' \mapsto C[F']\} * r'}$$

$$(\text{T-AnnotMeth})\ \frac{\mathsf{this} : C[F], x : T' * \mathsf{this} \rhd_C e : T \lhd \mathsf{this} : C[F'], x : T'' * \mathsf{this} \qquad F' \neq \langle _ \rangle}{\vdash_C \mathsf{req}\ F\ \mathsf{ens}\ F'\ \mathsf{for}\ T\ m(T'\ x)\ \{e\}}$$

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

Figure 15: Rules for recursive methods and other self-calls

4.5. **Self-Calls and Recursive Methods.** The rules in Figure 15 extend the language to include self-calls (method calls on this). This extension also supports recursive calls, which are necessarily self-calls. Self-calls do not check or advance the session type, and a method that is only self-called does not appear in the session type. A method that is self-called and called from outside appears in the session type, and calls from outside do check and advance the session type. The reason why it is safe to not check the session type for self-calls is that the effect of the self-call on the field typing is included in the effect of the method that calls it. All of the necessary checking of session types is done because of the original outside call that eventually leads to the self-call.

Because they are not in the session type, self-called methods must be explicitly annotated with their initial (req) and final (ens) field typings. The annotations are used to type self-calls (T-SelfCall) and method definitions (T-AnnotMeth). The result type and parameter type are also specified as part of the method definition, again because the method is not in the session type.

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

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

4.6. **Shared Types and Base Types.** The formal language described in this paper has a very strict linear type system. It is straightforward to add non-linear classes as an orthogonal extension: they would not have session types and their instances would be shared objects, treated in a completely standard way. Including them in the formalisation, however, would only complicate the typing rules.

More interesting, and more challenging, is the possibility of introducing a more refined approach to aliasing and ownership, for example along the lines of the systems discussed in Section 9. We intend to investigate this in the future.

```
FileReadCh = &{OPEN: ?String.⊕{OK: OpenCh, ERROR: FileReadCh}, QUIT: End}
OpenCh = &{HASNEXT: ⊕{TRUE: CanReadCh, FALSE: MustCloseCh}, CLOSE: FileReadCh}
MustCloseCh = &{CLOSE: FileReadCh}
CanReadCh = &{READ: !String.OpenCh, CLOSE: FileReadCh }
```

Figure 16: Remote file server version 1: channel session type (server side)

Base types such as **int** are also straightforward to add, and would be treated non-linearly.

4.7. **Inheritance.** The formal language uses a structural type system in which class names are only used in order to obtain their session types; method availability is determined solely by the session type, and method signatures are also in the session type. In particular, the subtyping relation is purely structural and makes no reference to class names. It is straightforward to adapt the language to include features associated with nominal subtyping, such as an explicitly declared inheritance hierarchy for classes with inheritance and overriding of method definitions. In this case, if class C is declared to inherit from class D, and both define session types (alternatively, C might inherit its session type from D), then the condition $C.\mathsf{session} <: D.\mathsf{session}$ would be required in order for the definition of C to be accepted.

5. A Distributed Example

We now present an example of a distributed system, illustrating the way in which our language unifies session-typed channels and more general typestate. Recall that our programming model is based on communication over TCP/IP-style socket connections, which we refer to as channels. The scenario is a file server, which clients can communicate with via a channel. The file server uses a local file, represented by a File object as defined in Section 2, and responds to requests such as OPEN and HASNEXT on the channel. On the client side, the remote file is represented by an object of class RemoteFile, whose interface is similar to File. In this "stub" object, methods such as open are implemented by communicating with the file server.

The channel between the client and the server has a session type in the standard sense [70], which defines a communication protocol. In our language, each endpoint of the channel is represented by an object of class Chan, with a class session type derived from the channel session type. This class session type also expresses the definition of the communication protocol, by specifying when the methods **send** and **receive** are available.

For the purpose of this example, we imagine that the communication protocol (channel session type) is defined by the provider of the file server, while the class session type of RemoteFile is defined by the implementor of a file system API. We therefore present two versions of the example: one in which the channel session type, and the class session type of RemoteFile, have the same structure; and one in which they have different structures.

5.1. **Distributed Example Version 1.** Figure 16 defines a channel session type for interaction between a file server and a client. The type of the server's endpoint is shown, and the type FileReadCh is the starting point of the protocol. The type constructor & means that the server offers a choice, in this case between OPEN and QUIT; the client makes a choice by sending one of these labels. If OPEN is selected, the server receives (constructor ?) a String

```
FileRead_cl = {Null send({OPEN}): {Null send(String): {{OK, ERROR} receive():
                                                        ⟨OK: Open_cl,
                                                         ERROR: FileRead_cl⟩}},
              Null send({QUIT}): {}}
where
  Open_cl = {Null send({HASNEXT}): {{TRUE, FALSE} receive():
                                    ⟨TRUE: CanRead_cl, FALSE: MustClose_cl⟩},
            Null send({CLOSE}): FileRead_cl}
  MustClose_cl = {Null send({CLOSE}): FileRead_cl}
  CanRead_cl = {Null send({READ}): {String receive(): Open_cl},
                Null send({CLOSE}): FileRead_cl}

FileRead_s = {{OPEN, QUIT} receive(): ⟨OPEN: {String receive():
                                              {Null send({OK}): Open_s,
                                               Null send({ERROR}): FileRead_s},
                                       QUIT: {}⟩}
where
  Open_s = {{HASNEXT, CLOSE} receive():
              ⟨HASNEXT: {Null send({TRUE}): CanRead_s,
                         Null send({FALSE}): MustClose_s},
               CLOSE: FileRead_s⟩}
  MustClose_s = {{CLOSE} receive(): ⟨CLOSE: FileRead_s⟩}
  CanRead_s = {{READ, CLOSE} receive(): ⟨READ: {Null send(String): Open_s},
                                         CLOSE: FileRead_s⟩}
```

Figure 17: Remote file server version 1: client and server class session types generated from channel session type FileReadCh

and then (the . constructor means sequencing) the constructor $\oplus$ indicates that the server can choose either OK or ERROR by sending the appropriate label. The remaining definitions are read in the same way; End means termination of the protocol. The type of the client's endpoint is dual, meaning that receive (?) and send (!) are exchanged, as are offer (&) and select ($\oplus$). When the server offers a choice, the client must make a choice, and *vice versa*.

The structure of the channel session type is similar to that of the class session type of File from Section 2, in the sense that HASNEXT is used to discover whether or not data can be read.

We regard each endpoint of a channel as an object with **send** and **receive** methods. For every channel session type there is a corresponding class session type that specifies the availability and signatures of **send** and **receive**. The general translation is defined in Section 6, Figure 26. For the particular case of FileReadCh, the client and server class session types are as defined in Figure 17: FileRead_cl for the client and FileRead_s for the server.

The requirement to make a choice ($\oplus$) in the channel session type corresponds to availability of **send** with a range of signatures, each with a parameter type representing one of the possible labels; here we are taking advantage of overloading, disambiguated by parameter type. The requirement to offer a choice (&) in the channel session type corresponds to availability of **receive**, with the subsequent session depending on the label that is received. Sending (!) and receiving (?) data in the channel session type correspond straightforwardly to **send** and **receive** with appropriate signatures.

Figure 18 defines the class RemoteFile, which acts as a local proxy for a remote file server. Its interface is similar to that of the class File from Section 2; the only difference

```
class RemoteFile {                                                          1
 session {connect: Init}                                                    2
  where Init = {open: ⟨OK: Open, ERROR: Init⟩}                              3
        Open = {hasNext: ⟨TRUE: Read, FALSE: Close⟩, close: Init }          4
        Read = {read: Open, close: Init }                                   5
        Close = {close: Init }                                              6
                                                                            7
  channel;                                                                  8
                                                                            9
  Null connect(⟨FileReadCh⟩ server) {                                      10
    channel = server.request();                                            11
  }                                                                        12
  {OK,ERROR} open(String name) {                                           13
    channel.send(OPEN);                                                    14
    channel.send(name);                                                    15
    switch (channel.receive()) {                                           16
      OK: OK;                                                              17
      ERROR: ERROR;                                                        18
   }                                                                       19
  }                                                                        20
  {TRUE,FALSE} hasNext () {                                                21
    channel.send(HAS_NEXT);                                                22
    switch (channel.receive()) {                                           23
      TRUE: TRUE;                                                          24
      FALSE: FALSE;                                                        25
    }                                                                      26
  }                                                                        27
  String read() {                                                          28
    channel.send(READ);                                                    29
    channel.receive();                                                     30
  }                                                                        31
  Null close() {                                                           32
    channel.send(CLOSE);                                                   33
  }                                                                        34
}                                                                          35
```

Figure 18: Remote file server version 1: client side stub

is that RemoteFile has an additional method connect, which must be called in order to establish a connection to the file server. The types RemoteFile.Init and File . Init are equivalent (Definition 3.9): each is a subtype of the other, and they can be used interchangeably.

The methods of RemoteFile are implemented by communicating over a channel to a file server. The connect method has a parameter of type ⟨FileReadCh⟩. A value of this type represents an access point, analogous to a URL, on which a connection can be requested by calling the **request** method (line 11); the resulting channel endpoint has type FileRead_cl.

The remaining methods communicate on the channel, and thus advance the type of the field channel. The similarity of structure between the channel session type FileReadCh and the class session type Init is reflected in the simple definitions of the methods, which just copy information between their parameters and results and the channel. There is one point of interest in relation to the close method. It occurs three times in the class session type, and according to our type system, its body is type checked once for each occurrence. Each time, the initial type environment in which the body is checked has a different type for the

```
class FileServer {
  session { Null main(⟨FileReadCh⟩ port): {} }

  channel; file;

  Null main(⟨FileRead_c⟩ port) {
    file = new File();
    channel = port.accept();
    fileRead();
  }
  req FileRead_s channel, Init file
  ens {} channel, Init file
  Null fileRead() {
    switch (channel.receive()) {
      OPEN:
        switch (file.open(channel.receive())) {
          OK: open();
          ERROR: fileRead();
        }
      QUIT: null;
    }
  }
  req Open_s channel, Open file
  ens {} channel, Init file
  Null open() {
    switch (channel.receive()) {
      HASNEXT:
        switch (file.hasNext()) {
          TRUE: channel.send(TRUE); canRead();
          FALSE: channel.send(FALSE); mustClose();
        }
      CLOSE: file.close(); fileRead();
  }
  req MustClose_s channel, Close file
  ens {} channel, Init file
  Null mustClose() {
    switch (channel.receive()) {
      CLOSE: file.close(); fileRead();
    }
  }
  req CanRead_s channel, Read file
  ens {} channel, Init file
  Null canRead() {
    switch (channel.receive()) {
      READ: channel.send(file.read()); open();
      CLOSE: file.close(); fileRead();
    }
  }
}
```

Figure 19: Remote file server version 1: server code

```
FileChannel = &{OPEN: ?String.⊕{OK: CanRead, ERROR: FileChannel}, QUIT: End}
CanRead = &{READ: ⊕{EOF: FileChannel, DATA: !String.CanRead}, CLOSE: FileChannel}
```

Figure 20: Remote file server version 2: channel session type (server side)

```
ClientCh = {Null send({OPEN}): {Null send(String): {{OK, ERROR} receive():
                                                    ⟨OK: CanRead_cl,
                                                     ERROR: ClientCh⟩}},
           Null send({QUIT}): {}}
where CanRead_cl = {Null send({READ}): {{EOF, DATA} receive():
                                         ⟨EOF: ClientCh,
                                          DATA:{String receive(): CanRead_cl}⟩},
                   Null send({CLOSE}): ClientCh}

ServerCh = {{OPEN, QUIT} receive(): ⟨OPEN: {String receive():
                                            {Null send({OK}): CanRead_cl,
                                             Null send({ERROR}): ServerCh}},
                                       QUIT: {}⟩}
where CanRead_cl = {{READ, CLOSE} receive():
                    ⟨READ: {Null send({EOF}): ServerCh,
                            Null send({DATA)}:{Null send(String): CanRead_cl}},
                     CLOSE: ServerCh⟩}
```

Figure 21: Remote file server version 2: client and server class session types generated from channel session type FileChannel

channel field: Open_cl, MustClose_cl or CanRead_cl. Type checking is successful because all of these types allow **send**({CLOSE}).

Figure 19 defines the class FileServer, which accesses a local file system and uses the server endpoint of a channel of type FileReadCh. The session type of this class contains the single method main, with a parameter of type ⟨FileReadCh⟩. We imagine this main method to be the top-level entry point of a stand-alone application, with the parameter value (the access point or URL for the server) being provided when the application is launched. The server uses **accept** to listen for connection requests, and when a connection is made, it obtains a channel endpoint of type FileRead_s.

The remaining methods of FileServer are mutually recursive in a pattern that matches the structure of FileRead_s. The methods are self-called, and do not appear in the class session type; instead, they are annotated with pre- and post-conditions on the types of the fields channel and file. The direct correspondence between the structure of the channel session type and the class session type of File is again reflected in the code, for example on lines 29 and 30 where the result of calling hasNext on file directly answers the HASNEXT query on channel.

Most systems of session types support delegation, which is the ability to send a channel as a message on another channel. It is indicated by the occurrence of a session type as the type of the message in a send (!) or receive (?) constructor. In our language, delegation is realised by sending an object representing a channel endpoint; it corresponds to a **send** method with a parameter of type, for example, Chan[FileRead_cl]. Transfer of channel endpoints from one process to another is supported by the operational semantics in Section 6.

```
class RemoteFile {
  session {connect: Init}
  where Init  = {open: ⟨OK: Open, ERROR: Init⟩}
        Open  = {hasNext: ⟨TRUE: Read, FALSE: Close⟩, close: Init}
        Read  = {read: Open, close: Init}
        Close = {close: Init}

  channel; state;

  Null connect(⟨FileChannel⟩ c) {
    channel = c.request();
  }
  {OK,ERROR} open(String name) {
    channel.send(OPEN);
    channel.send(name);
    switch (channel.receive()) {
      OK: state = READ; OK;
      ERROR: ERROR;
    }
  }
  {TRUE,FALSE} hasNext() {
    channel.send(READ);
    switch (channel.receive()) {
      EOF: state = EOF; FALSE;
      DATA: state = DATA; TRUE;
    }
  }
  String read() {
    state = READ;
    channel.receive();
  }
  Null close() {
    switch (state) {
      EOF: null;
      READ: channel.send(CLOSE);
      DATA: channel.receive(); channel.send(CLOSE);
    }
  }
}
```

Figure 22: Remote file server version 2: client side stub

5.2. **Distributed Example Version 2.** This version has a different channel session type, FileChannel, defined in Figure 20, which does not match the class session type FileRead. The difference is that there is no HASNEXT option; instead, the READ option is always available. If there is no more data then EOF is returned in response to READ; alternatively, DATA is returned, followed by the desired data. The corresponding class session types for the client and server endpoints are defined in Figure 21.

The implementation of RemoteFile must now mediate between the different structures of the class session type FileRead and the channel session type FileChannel. The new definition is in Figure 22. The main point is that the definition of the close method must depend on the state of the channel. For example, if close is called immediately after a call of hasNext that returns TRUE, then the channel session type requires data to be read before CLOSE can

```
class FileServer {
  session {main: {}}

  channel; file;

  Null main(⟨FileChannel⟩ c) {
    file = new File();
    channel = c.accept();
    serverCh();
  }
  req ServerCh channel, Init file
  ens {} channel, Init file
  Null serverCh() {
    switch (channel.receive()) {
      OPEN:
        switch (file.open(channel.receive())) {
          OK: canRead();
          ERROR: serverCh();
        }
      QUIT: null;
    }
  }
  req CanRead_s channel, Open file
  ens {} channel, Init file
  Null canRead() {
    switch (channel.receive()) {
      READ:
        switch (file.hasNext()) {
          TRUE: channel.send(DATA); channel.send(file.read());
                canRead();
          FALSE: channel.send(EOF); file.close(); serverCh();
        }
      CLOSE: file.close(); serverCh();
  }
}
```

Figure 23: Remote file server version 2: server code

be sent. We therefore introduce the field state, which stores a value of the enumerated type {EOF, READ, DATA}. This field represents the state of the channel (equivalently, the session type of the channel field): EOF corresponds to ClientCh, READ corresponds to CanRead, and DATA corresponds to the point after the DATA label in CanRead. The definition of close contains a **switch** on state, with appropriate behaviour for each possible value. It is also possible for state to be **null**, but this only occurs before open has been called, and at this point close is not available.

In order to type check this example we take advantage of the fact that the body of the close method is repeatedly checked, according to its occurrence in the class session type. The value of state always corresponds to the state of channel. This correspondence is not represented in the type system — that would require some form of dependent type — but whenever the body of close is type checked, the *type* of channel is compatible with the *value* of state, and so typechecking succeeds. More precisely, each possible value of state corresponds to a different singleton type for state (typing rule T-Label), and rule

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

Figure 24: Additional syntax for channels and states

T-Switch only checks the branches that correspond to possible values in the enumerated type of the condition. So each time the body of close is type checked, only *one* branch (because the type of state is a singleton) of the **switch** is checked, corresponding to the value of state for that occurrence of close.

6. A Core Distributed Language

We now define the core of the distributed language illustrated in Section 5. For simplicity, communication is synchronous. Formalising asynchronous communication is well-understood (for example, Gay and Vasconcelos [39] define a functional language with similar communication primitives but adds the complication of message buffers to the operational semantics).

Our integration of channel session types and the typestate system of this paper is based on binary session types [70] (actually, we adopt the now standard constructs of session types). It should be straightforward to adapt the technique to multi-party session types [42], because that system also depends on specifying the sequence of send and receive operations on each channel endpoint.

The only additions to the top-level language are access points and their types $\langle T\rangle$, channel session types and their translation to class session types, and the spawn primitive. However, there are significant changes to the internal language, in order to introduce a layer of concurrently executing components that communicate on channels.

6.1. **Syntax.** Figure 24 defines the new syntax. The types of access points are top-level declarations. Of the new values, access points n can appear in top-level programs, but channel endpoints, c^+ and c^-, are part of the internal language. If an access point n is declared with access $\langle\Sigma\rangle$ n then we define n.protocol to mean Σ. The spawn primitive was not used in the example in Section 5, but its behaviour is to start a new thread executing the specified method on a new instance of the specified class (just like it happens in Java; other works, e.g. [27], use similar approaches). Although a parameter is required as in any method call, for simplicity the type system restricts the parameter's type to be Null in this case, so that there is only one form of inter-thread communication. The syntax of channel session types Σ is included so that the types of access points can be declared. Channels are created by the interaction of methods request and accept in different threads, one thread keeps the c^- endpoint whereas the other keeps c^+. The two threads then communicate on channel c by reading and writing on their channel ends. The syntax of states is extended to include parallel composition and a channel binder νc, which binds both endpoints c^+

Structural congruence:

$$s_1 \parallel s_2 \;\equiv\; s_2 \parallel s_1 \qquad s_1 \parallel (s_2 \parallel s_3) \;\equiv\; (s_1 \parallel s_2) \parallel s_3 \qquad \text{(E-Comm,E-Assoc)}$$

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

Reduction rules:

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

$$\text{(R-ComBase)}\ \frac{h(r).f = c^p \qquad h'(r').f' = c^{\overline{p}} \qquad v \notin \mathcal{O}}{(h * r\ ,\ \mathcal{E}[f.\mathsf{send}(v)]) \parallel (h' * r'\ ,\ \mathcal{E}'[f'.\mathsf{receive}()]) \longrightarrow (h * r\ ,\ \mathcal{E}[\mathsf{null}]) \parallel (h' * r'\ ,\ \mathcal{E}'[v])}$$

$$\text{(R-ComObj)}\ \frac{h(r).f = c^p \qquad h'(r').f' = c^{\overline{p}} \qquad \varphi \in \mathsf{Inj}(dom(h \downarrow o), \mathcal{O} \setminus dom(h'))}{\begin{array}{l}(h * r\ ,\ \mathcal{E}[f.\mathsf{send}(o)]) \parallel (h' * r'\ ,\ \mathcal{E}'[f'.\mathsf{receive}()]) \longrightarrow \\ \qquad (h \uparrow o * r\ ,\ \mathcal{E}[\mathsf{null}]) \parallel (h' + \varphi(h \downarrow o) * r'\ ,\ \mathcal{E}'[\varphi(o)])\end{array}}$$

$$\text{(R-Spawn)}\ \frac{o \text{ fresh} \qquad C.\mathsf{fields} = \vec{f} \qquad m(x)\ \{e\} \in C}{(h * r\ ,\ \mathcal{E}[\mathsf{spawn}\ C.m(v)]) \longrightarrow (h * r\ ,\ \mathcal{E}[\mathsf{null}]) \parallel (o = C[\vec{f} = \vec{\mathsf{null}}] * o\ ,\ e\{^{\mathsf{null}}/_x\})}$$

$$\text{(R-Par)}\ \frac{s \longrightarrow s'}{s \parallel s'' \longrightarrow s' \parallel s''} \quad \text{(R-Str)}\ \frac{s \equiv s' \quad s' \longrightarrow s'' \quad s'' \equiv s'''}{s \longrightarrow s'''} \quad \text{(R-NewChan)}\ \frac{s \longrightarrow s'}{(\nu c)\, s \longrightarrow (\nu c)\, s'}$$

Top-level typing rules:

$$\text{(T-Spawn)}\ \frac{\Gamma * r \rhd e : \mathsf{Null} \lhd \Gamma' * r' \qquad C.\mathsf{session} = \{_\ m(\mathsf{Null}) : _; \ldots\}}{\Gamma * r \rhd \mathsf{spawn}\ C.m(e) : \mathsf{Null} \lhd \Gamma' * r'}$$

$$\text{(T-Name)}\ \frac{\mathsf{access}\ \langle\Sigma\rangle\ n}{\Gamma * r \rhd n : [\![\langle\Sigma\rangle]\!] \lhd \Gamma * r}$$

Figure 25: Reduction and top-level typing rules for concurrency and channels

and c^- in the style of Gay and Vasconcelos [39]. In a parallel composition, the states are exactly states from the semantics of the sequential language; in particular, each one has its own heap. This means that spawn generates a new heap as well as a new executing method body. Communication between parallel expressions is only via channels.

The syntax extensions do not include request, accept, send and receive, as they are treated as method names.

6.2. **Semantics.** Figure 25 defines the reduction rules for the distributed language, as well as the top-level typing rules. The reduction rules make use of a pi-calculus style structural congruence relation, again following Gay and Vasconcelos [39]. It is the smallest congruence (with respect to parallel and binding) that is also closed under the given rules.

Rule R-Init defines interaction between accept and request, which creates a fresh channel c and substitutes one endpoint into each expression.

There are two rules for communication, involving interaction between send and receive. Rule R-ComBase is for communication of non-objects and rule R-ComObj is for communication of objects. Let $\mathcal{O}$ be the set of all object identifiers. R-ComBase expresses a straightforward transfer of a value, while R-ComObj also transfers part of the heap corresponding to the contents of a transferred object. In R-ComObj, φ is an arbitrary renaming function which associates to every identifier in $dom(h)$ an identifier not in $dom(h')$. This

rule can easily be made deterministic in practice by using a total ordering on identifiers and a mechanism to generate fresh ones.

R-Spawn creates a new parallel state whose heap contains a single instance of the specified class. As discussed above, communication between threads is only through channels in order to keep the formal system a reasonable size; therefore, no data is transmitted to the new thread and the body of the method being spawned always has its parameter replaced by the literal null. The type system will ensure that $v = \mathsf{null}$, so that this semantics makes sense. The remaining rules are standard.

Returning to R-ComObj, there is some additional notation associated with identifying the part of the heap that must be transferred; we now define it.

Definition 6.1. Let h be a heap. For any entry $o = C[\{f_i = v_i\}_{i \in I}]$ in h, we define the *children* of o in h to be the set of all v_i which are object identifiers: $\mathsf{children}_h(o) = \{v_i \mid i \in I\} \cap \mathcal{O}$.

We say that an object identifier o in $dom(h)$ is a *root* in h if there is no o' in $dom(h)$ such that $o \in \mathsf{children}_h(o')$. We note $\mathsf{roots}(h)$ the set of roots in h.

We say that h is *complete* if for any o in $dom(h)$ we have $\mathsf{children}_h(o) \subseteq dom(h)$.

If h is complete, we define the descendants of o in h to be the smallest set containing o and the children of any object it contains. Formally, let $\mathsf{children}^0_h(o) = \{o\}$ and for $i \geqslant 1$, $\mathsf{children}^i_h(o) = \bigcup_{\omega \in \mathsf{children}^{i-1}_h(o)} \mathsf{children}_h(\omega)$. Then $\mathsf{desc}_h(o) = \bigcup_{i \in \mathbb{N}} \mathsf{children}^i_h(o)$.

Definition 6.2 (Heap separation)**.** Let h be a complete heap and o a root of h. We define $h \downarrow o$ to be the sub-heap obtained by restricting h to the descendants of o, and $h \uparrow o$ to be the sub-heap obtained by removing from h the descendants of o. Note that $h \downarrow o$ is complete and has the property that o is its only root.

Definition 6.3 (Additional notation)**.**

- Let h be a heap and let φ be a function from $\mathcal{O}$ to $\mathcal{O}$. We denote by $\varphi(h)$ the result of applying φ to all object identifiers in h, including inside object records.
- We denote by $\mathsf{Inj}(A, B)$ the set of injective functions from A to B.
- We denote by $+$ the disjoint union of heaps or environments, i.e. the operation $h + h'$ is defined by merging h and h' if their domains are disjoint and undefined otherwise.

6.3. **Type System.** The type system treats send, $\mathsf{receive}$, $\mathsf{request}$ and accept as method calls on objects whose session types are defined by the translations in Figure 26. A channel endpoint with (channel) session type Σ is treated as an object with (class) session type $[\![\Sigma]\!]$. The type constructor & (offer) is translated into a $\mathsf{receive}$ method with return type $\mathsf{variant\text{-}tag}$ in order to capture the relationship between the received label and the subsequent type. The type constructor $\oplus$ (select) is translated into a collection of send methods with different parameter types, each being a singleton type for the corresponding label.

In a similar but much simpler way, an access type $\langle\Sigma\rangle$ is translated into a (class) session type that allows both $\mathsf{request}$ and accept to be called repeatedly and at any time. These two methods need to return dual channel endpoints, which requires the following definition.

Definition 6.4 (Dual channel type)**.** The dual type $\overline{\Sigma}$ of a channel session type Σ is defined by

$$\overline{\Sigma} = dual(\Sigma, \iota)$$

Given a channel session type Σ, define a class session type $[\![\Sigma]\!]$ as follows.

$$
\begin{aligned}
[\![\mathsf{end}]\!] &= \{\} \\
[\![X]\!] &= X \\
[\![\mu X.\Sigma]\!] &= \mu X.[\![\Sigma]\!] \\
[\![?\,[T]\,.\,\Sigma]\!] &= \{T\ \mathsf{receive}(\mathsf{Null}) : [\![\Sigma]\!]\} \\
[\![!\,[T]\,.\,\Sigma]\!] &= \{\mathsf{Null}\ \mathsf{send}(T) : [\![\Sigma]\!]\} \\
[\![\&\,\{l : \Sigma_l\}_{l\in E}]\!] &= \{\mathsf{variant\text{-}tag}\ \mathsf{receive}(\mathsf{Null}) : \langle l : [\![\Sigma_l]\!]\rangle_{l\in E}\} \\
[\![\oplus\,\{l : \Sigma_l\}_{l\in E}]\!] &= \{\mathsf{Null}\ \mathsf{send}(\{l\}) : [\![\Sigma_l]\!]\}_{l\in E}
\end{aligned}
$$

In the type system, a channel endpoint with session type Σ is treated as an object with type $[\![\Sigma]\!]$. Calls of send and receive are typed as standard method calls.

Given an access type $\langle\Sigma\rangle$, define a class session type $[\![\langle\Sigma\rangle]\!]$ by

$$[\![\langle\Sigma\rangle]\!] = \mu X.\{[\![\overline{\Sigma}]\!]\ \mathsf{request}(\mathsf{Null}) : X, [\![\Sigma]\!]\ \mathsf{accept}(\mathsf{Null}) : X\}.$$

In the type system, an access point with type $\langle\Sigma\rangle$ is treated as an object with type $[\![\langle\Sigma\rangle]\!]$. Calls of request and accept are typed as standard method calls.

Figure 26: Object types for channels and access points

These rules add to or replace the rules in Figure 12.

$$\text{(T-Chan)}\quad \Gamma, c^p : T * r \rhd c^p : T \lhd \Gamma * r \qquad\qquad \text{(T-Hempty)}\quad \Theta \vdash \varepsilon : [\![\Theta]\!]$$

$$\text{(T-Hadd)}\ \frac{\Theta \vdash h : \Gamma \qquad \Gamma, o : C[\{\mathsf{Null}\ f_i\}_{1\leqslant i\leqslant n}] * o \rhd f_1 \leftrightarrow v_1; \ldots; f_n \leftrightarrow v_n : \mathsf{Null} \lhd \Gamma' * o}{\Theta \vdash h, \{o = C[\{f_i = v_i\}_{1\leqslant i\leqslant n}]\} : \Gamma'}$$

$$\text{(T-Hide)}\ \frac{\Theta \vdash h : \Gamma, o : C[F] \qquad F \vdash C : S}{\Theta \vdash h : \Gamma, o : C[S]} \qquad \text{(T-State)}\ \frac{\Theta \vdash h : \Gamma \qquad \Gamma * r \rhd e : T \lhd \Gamma' * r'}{\Theta; \Gamma \rhd (h * r\ ,\ e) : T \lhd \Gamma' * r'}$$

$$\text{(T-Thread)}\ \frac{\Theta; \Gamma \rhd (h * r\ ,\ e) : T \lhd \Gamma' * r'}{\Theta \vdash (h * r\ ,\ e)} \qquad \text{(T-Par)}\ \frac{\Theta \vdash s \qquad \Theta' \vdash s'}{\Theta + \Theta' \vdash s \,||\, s'}$$

$$\text{(T-NewChan)}\ \frac{\Theta, c^+ : \Sigma, c^- : \overline{\Sigma} \vdash s}{\Theta \vdash (\nu c)\, s}$$

Figure 27: Internal typing rules for the distributed language

where ι is the identity substitution, $\sigma\{^{\Sigma'}/_X\}$ denotes the extension of substitution σ by the mapping $X \mapsto \Sigma'$, and $\Sigma\sigma$ denotes the application of the substitution σ to the session type Σ; the auxiliary function $dual(\Sigma, \sigma)$ is defined on session types Σ and substitutions σ by:

$$
\begin{aligned}
dual(\mathsf{end}, \sigma) &= \mathsf{end} \\
dual(X, \sigma) &= X \\
dual(\mu X.\Sigma, \sigma) &= \mu X.dual(\Sigma, \sigma\{^{\Sigma\sigma}/_X\}) \\
dual(!\,[T]\,.\,\Sigma, \sigma) &= ?\,[T\sigma]\,.\,dual(\Sigma, \sigma) \\
dual(?\,[T]\,.\,\Sigma, \sigma) &= !\,[T\sigma]\,.\,dual(\Sigma, \sigma) \\
dual(\oplus\,\{l : \Sigma_l\}_{l\in E}, \sigma) &= \&\,\{l : dual(\Sigma_l, \sigma)\}_{l\in E} \\
dual(\&\,\{l : \Sigma_l\}_{l\in E}, \sigma) &= \oplus\,\{l : dual(\Sigma_l, \sigma)\}_{l\in E}
\end{aligned}
$$

With this definition, duality commutes with unfolding [6]; this property is essential in order to use the equi-recursive convention (Definition 3.9).

By convention, request returns a channel endpoint of type $[\![\overline{\Sigma}]\!]$ and accept returns an endpoint of type $[\![\Sigma]\!]$.

Because access points n are global constants, they can be used repeatedly even though their session types are linear; there is no restriction to a single occurrence of a given name.

The only new typing rules for the top-level language are in Figure 25. T-Spawn allows a method to be used in a spawn expression if it is available in the initial session type of the specified class. T-Name obtains the type of an access point from its declaration, and assigns an object type according to the translation described above.

Figure 27 contains typing rules for the internal language with the concurrency extensions. Rules with the same names as rules in Figure 12 are replacements.

Rule T-Chan takes the type of a channel endpoint from the typing environment. The remaining rules involve a new typing environment Θ, which maps channel endpoints to channel session types Σ; these are indeed *channel* session types, not their translations into class session types. T-HAdd, T-Hide and T-State are just the corresponding rules from Figure 12 with Θ added. In T-HEmpty the notation $[\![\Theta]\!]$ means that the translation from channel session types to class session types is applied to the type of each channel endpoint. In combination with T-State, this means that the typing of expressions uses class session types for channel endpoints; the T in T-Chan is a class session type.

T-Thread lifts a typed state to a typed concurrent component, preserving only the channel typing Θ, which is used in T-Par and T-NewChan. In T-Par, $\Theta + \Theta'$ means union, with the assumption that Θ and Θ' have disjoint domains. T-NewChan requires the complementary endpoints of each channel to have dual session types.

6.4. **Subtyping.** We have two subtyping relations between channel session types: $\Sigma <: \Sigma'$ as defined by Gay and Hole [37], and $[\![\Sigma]\!] <: [\![\Sigma']\!]$ as defined in this paper. To avoid a detour into the definition of $\Sigma <: \Sigma'$, we state the following result without proof.

Proposition 6.5. $\Sigma <: \Sigma' \Rightarrow [\![\Sigma]\!] <: [\![\Sigma']\!]$.

Interestingly, the converse is not true, as subtyping between translations of channel session types is a larger relation. For example:

- for all Σ, $[\![\Sigma]\!] <: [\![\mathsf{end}]\!]$
- if E is an enumeration then $[\![?\,[E]\,.\,\Sigma]\!] <: [\![\&\,\{l : \Sigma\}_{l \in E}]\!]$
- $[\![\oplus\,\{l : \Sigma\}]\!] = [\![!\,[\{l\}]\,.\,\Sigma]\!]$ and therefore by transitivity any translated $\oplus$ type is a subtype of all the corresponding individual translated send types.

This suggests the possibility, in the context of the work of Gay and Hole [37], of generalizing the subtyping relation between channel session types by considering branch/select labels as values in an enumerated type. We do not explore this idea further in the present paper.

7. Results

The key results concerning the distributed language supporting self-calls are, again, Subject-Reduction, Type Safety, and Conformance. Notice that we can no longer guarantee the absence of stuck states for all well-typed programs, as one endpoint of a channel may try to send when the other endpoint is not available to receive.

7.1. **Properties of typing derivations.** This subsection is mostly a collection of lemmas which will be used to prove the main theorems in the following subsections. They draw various useful consequences from the fact that a program state is well-typed. Their proofs can be found in Appendix 10.

We define $\mathsf{chans}(h)$ as the set of channel endpoints appearing in object records in h. We define $\mathsf{chans}(\Gamma)$ and $\mathsf{objs}(\Gamma)$ as the sets of, respectively, channel endpoints and object identifiers in $dom(\Gamma)$. We have $dom(\Gamma) = \mathsf{chans}(\Gamma) \cup \mathsf{objs}(\Gamma)$.

Lemma 7.1. Suppose $\Theta \vdash h : \Gamma$. Then *(a)* h is complete, *(b)* $\mathsf{chans}(\Gamma) \subseteq dom(\Theta) \setminus \mathsf{chans}(h)$ and *(c)* $\mathsf{objs}(\Gamma) \subseteq \mathsf{roots}(h)$.

Lemma 7.2 (Rearrangement of typing derivations for expressions)**.** Suppose we have $\Gamma * r \rhd e : T \lhd \Gamma' * r'$. Then there exists a typing derivation for this judgement in which:

(1) T-Sub only occurs at the very end, just before T-Switch or T-SwitchLink as the last rule in the derivation for each of the branches, or just before T-Call as the last rule in the derivation for the parameter;
(2) T-SubEnv only occurs immediately before T-Sub in the first three cases and does not occur at all in the fourth, i.e. T-Call.

Lemma 7.3 (Rearrangement of typing derivations for heaps)**.** Suppose $\Theta \vdash h : \Gamma$ holds. Let o be an arbitrary root of h. Then there exists a typing derivation for it such that:

(1) T-Sub is never used;
(2) T-SubEnv is used at most once, as the last rule leading to the right premise of the last occurrence of T-Hadd;
(3) every occurrence of T-Hide follows immediately the occurrence of T-Hadd concerning the same object identifier;
(4) the occurrence of T-Hadd concerning an identifier o' is always immediately preceded (on the left premise) by the occurrences of T-Hadd/T-Hide concerning the descendants of o';
(5) the first root added is o.

Lemma 7.4 (Splitting of the heap)**.** Suppose $\Theta \vdash h : \Gamma, o : T$. Let $\Theta_1 = \Theta \setminus \mathsf{chans}(h \downarrow o)$ and let Θ_2 be Θ restricted to $\mathsf{chans}(h \downarrow o)$. Then we have: $\Theta_1 \vdash (h \uparrow o) : \Gamma$ and $\Theta_2 \vdash (h \downarrow o) : o : T$.

Lemma 7.5 (Merging of heaps)**.** Suppose $\Theta \vdash h : \Gamma$ and $\Theta' \vdash h' : \Gamma'$ with $dom(h) \cap dom(h') = \emptyset$ and $dom(\Theta) \cap dom(\Theta') = \emptyset$. Then we have $\Theta + \Theta' \vdash h + h' : \Gamma + \Gamma'$.

These two lemmas show, if we apply them repeatedly, that a typing derivation for a heap can be considered as a set of separate typing derivations leading to each root of the heap. This will allow us in particular to show results for particular cases where a heap has only one root and generalize them.

Lemma 7.6. Suppose $\Theta \vdash h : o : S$. Let φ be an injective function from $dom(h)$ to $\mathcal{O}$. Then we have $\Theta \vdash \varphi(h) : \varphi(o) : S$.

Lemma 7.7 (Opening)**.** If $\Theta \vdash h : \Gamma$, if $\Gamma(r)$ is a branch session type S and if $h(r)$ is an object identifier o, then we know from Lemma 7.1 that h contains an entry for o. Let C be the class of this entry, then there exists a field typing F for C such that $\Theta \vdash h : \Gamma\{r \mapsto C[F]\}$ and $F \vdash C : S$.

Lemma 7.8 (Closing)**.** If $\Theta \vdash h : \Gamma$ and $\Gamma(r) = C[F]$ and $F \vdash C : S$, then $\Theta \vdash h : \Gamma\{r \mapsto S\}$.

Lemma 7.9 (modification of the heap)**.** Suppose that we have $\Theta \vdash h : \Gamma$ and $\Gamma * r \rhd v' : T' \lhd \Gamma' * r$, and that $\Gamma'(r.f) = T$ where T is not a variant. Let $v = h(r).f$. The modified heap $h\{r.f \mapsto v'\}$ can be typed as follows:

(1) if v is an object identifier or a channel endpoint, then:

$$\Theta \vdash h\{r.f \mapsto v'\} : \Gamma'\{r.f \mapsto T'\}, v : T$$

(2) if v is not an object or channel and T is not a link type, then:

$$\Theta \vdash h\{r.f \mapsto v'\} : \Gamma'\{r.f \mapsto T'\}$$

(3) if $v = l_0$ and $T = \mathsf{link}\ f'$, then:
- $\Gamma'(r.f') = \langle l : S_l \rangle_{l \in E}$ for some E such that $l_0 \in E$ and some set of branch session types S_l. Note that this implies $f \neq f'$.
- $\Theta \vdash h\{r.f \mapsto v'\} : \Gamma'\{r.f \mapsto T'\}\{r.f' \mapsto S_{l_0}\}$

Lemma 7.10 (Substitution)**.** If $\mathsf{this} : C[F], x : T' * \mathsf{this} \rhd e : T \lhd \mathsf{this} : C[F'] * \mathsf{this}$, and if $\Gamma(r) = C[F]$, then:

(1) if T' is a base type (i.e. neither an object type nor a link) and v is a literal value of that type, or if v is an access point name declared with type $\langle \Sigma \rangle$ and $[\![\langle \Sigma \rangle]\!] <: T'$, we have:

$$\Gamma * r \rhd e\{^v/_x\} : T \lhd \Gamma\{r \mapsto C[F']\} * r.$$

(2) if T' is an object type and v is an object identifier or a channel endpoint, we have:

$$\Gamma, v : T' * r \rhd e\{^v/_x\} : T \lhd \Gamma\{r \mapsto C[F']\} * r.$$

Lemma 7.11 (Typability of Subterms)**.** If $\mathcal{D}$ is a derivation of $\Gamma * r \rhd \mathcal{E}(e) : T \lhd \Gamma' * r'$ then there exist Γ_1, r_1 and U such that $\mathcal{D}$ has a subderivation $\mathcal{D}'$ concluding $\Gamma * r \rhd e : U \lhd \Gamma_1 * r_1$ and the position of $\mathcal{D}'$ in $\mathcal{D}$ corresponds to the position of the hole in $\mathcal{E}$.

Lemma 7.12 (Replacement)**.** If

(1) $\mathcal{D}$ is a derivation of $\Gamma * r \rhd \mathcal{E}(e) : T \lhd \Gamma' * r'$
(2) $\mathcal{D}'$ is a subderivation of $\mathcal{D}$ concluding $\Gamma * r \rhd e : U \lhd \Gamma_1 * r_1$
(3) the position of $\mathcal{D}'$ in $\mathcal{D}$ corresponds to the position of the hole in $\mathcal{E}$
(4) $\Gamma'' * r'' \rhd e' : U \lhd \Gamma_1 * r_1$

then $\Gamma'' * r'' \rhd \mathcal{E}(e') : T \lhd \Gamma' * r'$.

7.2. **Type preservation.** We use $\vdash s$ as an abbreviation for $\emptyset \vdash s$; this represents well-typedness of a closed configuration. We have the following result:

Theorem 7.13 (Subject Reduction)**.** If, in a context parameterised by a set of well-typed declarations, we have $\vdash s$ and $s \longrightarrow s'$, then $\vdash s'$.

This global result is a consequence of a subject reduction theorem for a single thread, which is similar but not identical to what we stated as Theorem 3.19 (which will be a particular case). The reason it is not identical is that we need to prove that the type of an expression is preserved not only when this expression reduces on its own but also when it communicates with another thread. In order to state precisely this thread-wise type preservation theorem, we introduce a labelled transition system for threads. Transition labels can be: τ indicating internal reduction, $c^p\,!\,[v]$ or $c^p\,?\,[v]$ indicating that the non-object value v is sent or received on channel c^p, $c^p\,!\,[h]$ or $c^p\,?\,[h]$, where h is a heap with a

single root o, indicating that the object o (together with its content) is sent or received on channel c^p, $n[c^p]$ indicating that the channel endpoint c^p is received from access point n, or, finally, $C.m()$ indicating that the thread spawns another one using method m of class C.

Definition 7.14 (Labelled transition system)**.** We define a labelled transition system for threads by the following rules:

$$(\textsc{Tr-Red})\ \frac{(h * r\ ,\ e) \longrightarrow (h' * r'\ ,\ e')}{(h * r\ ,\ e) \xrightarrow{\tau} (h' * r'\ ,\ e')}$$

$$(\textsc{Tr-Send})\ \frac{h(r).f = c^p \qquad v \notin \mathcal{O}}{(h * r\ ,\ \mathcal{E}[f.\mathsf{send}(v)]) \xrightarrow{c^p\ ![v]} (h * r\ ,\ \mathcal{E}[\mathsf{null}])}$$

$$(\textsc{Tr-SendObj})\ \frac{h(r).f = c^p}{(h * r\ ,\ \mathcal{E}[f.\mathsf{send}(o)]) \xrightarrow{c^p\ ![h \downarrow o]} (h \uparrow o * r\ ,\ \mathcal{E}[\mathsf{null}])}$$

$$(\textsc{Tr-Receive})\ \frac{h(r).f = c^p \qquad v \notin \mathcal{O}}{(h * r\ ,\ \mathcal{E}[f.\mathsf{receive}()]) \xrightarrow{c^p\ ?[v]} (h * r\ ,\ \mathcal{E}[v])}$$

$$(\textsc{Tr-RcvObj})\ \frac{h(r).f = c^p \qquad \mathsf{roots}(h') = \{o\} \qquad dom(h) \cap dom(h') = \emptyset}{(h * r\ ,\ \mathcal{E}[f.\mathsf{receive}()]) \xrightarrow{c^p\ ?[h']} (h + h' * r\ ,\ \mathcal{E}[o])}$$

$$(\textsc{Tr-Accept})\ \frac{h(r).f = n}{(h * r\ ,\ \mathcal{E}[f.\mathsf{accept}()]) \xrightarrow{n[c^+]} (h * r\ ,\ \mathcal{E}[c^+])}$$

$$(\textsc{Tr-Request})\ \frac{h(r).f = n}{(h * r\ ,\ \mathcal{E}[f.\mathsf{request}()]) \xrightarrow{n[c^-]} (h * r\ ,\ \mathcal{E}[c^-])}$$

$$(\textsc{Tr-Spawn})\ \ (h * r\ ,\ \mathcal{E}[\mathsf{spawn}\ C.m(v)]) \xrightarrow{C.m()} (h * r\ ,\ \mathcal{E}[\mathsf{null}])$$

Note that *both* τ and $C.m()$ correspond to the thread being able to reduce on its own. An important feature of this transition relation is that, for all rules, the right-hand state is fully determined by the left-hand one and the transition label. Moreover, the only case where several different transitions are possible from a given state is when applying the rule receive, as the right-hand side depends on the value received.

Definition 7.15. A similar transition relation, with the same set of labels, is defined on channel environments Θ as follows:

$$\Theta \xrightarrow{\tau} \Theta \qquad \Theta \xrightarrow{C.m()} \Theta \qquad \frac{n.\mathsf{protocol} = \langle\Sigma\rangle \qquad \forall p, c^p \notin dom(\Theta)}{\Theta \xrightarrow{n[c^+]} \Theta, c^+ : \Sigma}$$

$$\frac{n.\mathsf{protocol} = \langle\Sigma\rangle \qquad \forall p, c^p \notin dom(\Theta)}{\Theta \xrightarrow{n[c^-]} \Theta, c^- : \overline{\Sigma}} \qquad \frac{[\![\Sigma']\!] <: T}{\Theta, c^p : ![T].\Sigma, c'^{p'} : \Sigma' \xrightarrow{c^p\,![c'^{p'}]} \Theta, c^p : \Sigma}$$

$$\frac{[\![\Sigma']\!] <: T}{\Theta, c^p : ?[T].\Sigma \xrightarrow{c^p\,?[c'^{p'}]} \Theta, c^p : \Sigma, c'^{p'} : \Sigma'}$$

$$\frac{\emptyset \rhd v : T \lhd \emptyset}{\Theta, c^p : ![T].\Sigma \xrightarrow{c^p\,![v]} \Theta, c^p : \Sigma} \qquad \frac{\emptyset \rhd v : T \lhd \emptyset}{\Theta, c^p : ?[T].\Sigma \xrightarrow{c^p\,?[v]} \Theta, c^p : \Sigma}$$

$$\frac{l_0 \in E}{\Theta, c^p : \oplus\{l : \Sigma_l\}_{l\in E} \xrightarrow{c^p\,![l_0]} \Theta, c^p : \Sigma_{l_0}} \qquad \frac{l_0 \in E}{\Theta, c^p : \&\{l : \Sigma_l\}_{l\in E} \xrightarrow{c^p\,?[l_0]} \Theta, c^p : \Sigma_{l_0}}$$

$$\frac{\Theta_1 \vdash h : o : S \qquad dom(\Theta_1) \subseteq \mathsf{chans}(h)}{\Theta_1 + \Theta_2, c^p : ![S].\Sigma \xrightarrow{c^p\,![h]} \Theta_2, c^p : \Sigma}$$

$$\frac{\Theta' \vdash h : o : S \qquad dom(\Theta') \subseteq \mathsf{chans}(h) \qquad dom(\Theta) \cap dom(\Theta') = \emptyset}{\Theta, c^p : ?[S].\Sigma \xrightarrow{c^p\,?[h]} \Theta + \Theta', c^p : \Sigma}$$

Where we use $\emptyset \rhd v : T \lhd \emptyset$ as an abbreviation for $\mathsf{dummy} : C[] * \mathsf{dummy} \rhd v : T \lhd \mathsf{dummy} : C[] * \mathsf{dummy}$ — meaning that v is a literal value (or access point name) of type T.

We can now state our thread-wise type preservation theorem.

Theorem 7.16 (Thread-wise progress and type preservation)**.** Let $\mathcal{D}$ be a set of well-typed declarations, that is, such that for every class declaration D in $\mathcal{D}$ we have $\vdash D$. In a context parameterised by $\mathcal{D}$, suppose we have $\Theta; \Gamma \rhd (h * r\ ,\ e) : T \lhd \Gamma' * r'$.

Then either e is a value or there exists a transition label λ such that we have $(h{*}r\ ,\ e) \xrightarrow{\lambda} (h' * r''\ ,\ e')$ for some h', r'' and e'.

Furthermore, if λ is such that $\Theta \xrightarrow{\lambda} \Theta'$ for some Θ', then there exists Γ'' such that $\Theta'; \Gamma'' \rhd (h' * r''\ ,\ e') : T \lhd \Gamma' * r'$ holds.

Theorem 3.19 is the particular case where $\lambda = \tau$.

Corollary 7.17 (Theorem 3.20)**.** If $\mathcal{D}$ contains no name declaration and Θ is empty, then there exists s' such that $(h * r\ ,\ e) \longrightarrow s'$.

(Corollary). In that particular case, $\Theta \vdash h : \Gamma$ implies that the heap cannot contain any n or c^p, hence λ can only be τ or of the form $C.m()$. □

(Theorem). We always use typing derivations where subsumption steps only occur at the positions described in Lemma 7.2. Furthermore, it is sufficient to consider only cases where subsumption does not occur at the end: indeed, if it does occur, then we can add a similar subsumption step to the new judgement. The hypothesis in the theorem that $\Theta; \Gamma \rhd (h * r\ ,\ e) : T \lhd \Gamma' * r'$ holds is necessarily a result of T-State and therefore is

equivalent to the two hypotheses $\Theta \vdash h : \Gamma$ and $\Gamma * r \rhd e : T \lhd \Gamma' * r'$, which we will sometimes refer to directly.

We prove the theorem by induction on the structure of e with respect to contexts, and present the inductive case first:

If e is of the form $\mathcal{E}[e_1]$ where e_1 is not a value and $\mathcal{E}$ is not just $[_]$, then Lemma 7.11 tells us that $\Gamma * r \rhd e_1 : U \lhd \Gamma_1 * r_1$ appears in the typing derivation of $\Gamma * r \rhd e : T \lhd \Gamma' * r'$ for some U, r_1 and Γ_1. From there we can apply T-State and derive $\Theta; \Gamma \rhd (h * r \;,\; e_1) : U \lhd \Gamma_1 * r_1$. This allows us to use the induction hypothesis and get λ, e_2, r'' and h' such that $(h * r \;,\; e_1) \xrightarrow{\lambda} (h' * r'' \;,\; e_2)$. Then we straightforwardly have $e \xrightarrow{\lambda} \mathcal{E}[e_2]$, either by applying R-Context if λ is τ or by replacing the context in the transition rule if it is something else. Now if λ is such that $\Theta \xrightarrow{\lambda} \Theta'$, then the induction hypothesis[1] also gives us Γ'' such that $\Theta'; \Gamma'' \rhd (h' * r'' \;,\; e_2) : U \lhd \Gamma_1 * r_1$ holds. From this we get, by reading T-State upwards, $\Theta' \vdash h' : \Gamma''$ and $\Gamma'' * r'' \rhd e_2 : U \lhd \Gamma_1 * r_1$. We use Lemma 7.12 with the latter in order to obtain $\Gamma'' * r'' \rhd \mathcal{E}[e_2] : T \lhd \Gamma' * r'$ and conclude with T-State.

The base cases are if e is of the form $\mathcal{E}[v]$ with $\mathcal{E}$ elementary (i.e. not of the form $\mathcal{E}'[\mathcal{E}'']$ with $\mathcal{E}'' \neq [_]$) and if it is not of the form $\mathcal{E}[e_1]$ at all. We list them below.

- If e is a value, there is nothing to prove.
- e cannot be a variable. Indeed, $\Theta \vdash h : \Gamma$ implies that $dom(\Gamma)$ contains only object identifiers and channel endpoints. Therefore, $\Gamma * r \rhd e : T \lhd \Gamma' * r'$ cannot be a conclusion of T-Var or T-LinVar, thus e is not a variable.
- $e = v; e'$. Then the expression reduces by R-Seq and the initial derivation is as follows:

$$\dfrac{\Theta \vdash h : \Gamma \;\textbf{(a)} \qquad \dfrac{(1)\;\dfrac{\cdots}{\Gamma * r \rhd v : T' \lhd \Gamma_1 * r} \qquad \Gamma_1 * r \rhd e' : T \lhd \Gamma' * r' \;\textbf{(b)}}{\Gamma * r \rhd v; e' : T \lhd \Gamma' * r'}\,(\text{T-Seq})}{\Theta; \Gamma \rhd (h * r \;,\; v; e') : T \lhd \Gamma' * r'}\,(\text{T-State})$$

  Furthermore, T' is not a link type. Therefore, (1) cannot be T-VarF or T-VarS and it is either T-Ref, T-Chan, T-Name, T-Label or T-Null, since these are the only rules for typing values. If it is T-Null, T-Label or T-Name, then $\Gamma = \Gamma_1$; if it is T-Ref or T-Chan, then $\Gamma <: \Gamma_1$ and we can use Lemma 3.15 to get $\Gamma * r \rhd e' : T \lhd \Gamma' * r'$ from (b) in both cases. We conclude from this using (a) and T-State.
- $e = \mathsf{new}\ C()$. Then the expression reduces by R-New and the initial reduction is as follows:

$$(\text{T-State})\;\dfrac{\Theta \vdash h : \Gamma \;\textbf{(a)} \qquad \dfrac{}{\Gamma * r \rhd \mathsf{new}\ C() : C.\mathsf{session} \lhd \Gamma * r}\,(\text{T-New})}{\Theta; \Gamma \rhd (h * r \;,\; \mathsf{new}\ C()) : C.\mathsf{session} \lhd \Gamma * r}$$

  Let $S = C.\mathsf{session}$. From the hypothesis that $\mathcal{D}$ is well-typed, we have $\vdash \mathsf{class}\, C\ \{S; \vec{f}; \vec{M}\}$. This must come from T-Class, therefore we have $\overrightarrow{\mathsf{Null}}\ \vec{f} \vdash C : S$ **(b)**.

  We build the following derivation:

$$(\text{T-Hide})\;\dfrac{(\text{T-Hadd})\;\dfrac{(\text{a}) \qquad \dfrac{\text{T-Null, T-Swap, T-Seq}}{\Gamma, o : C[\overrightarrow{\mathsf{Null}}\ \vec{f}] * o \rhd \vec{f} \leftrightarrow \overrightarrow{\mathsf{Null}} : \mathsf{Null} \lhd \Gamma, o : C[\overrightarrow{\mathsf{Null}}\ \vec{f}] * o}}{\Theta \vdash h, \{o = C[\vec{f} = \overrightarrow{\mathsf{null}}]\} : \Gamma, o : C[\overrightarrow{\mathsf{Null}}\ \vec{f}]} \qquad (\text{b})}{\Theta \vdash h, \{o = C[\vec{f} = \overrightarrow{\mathsf{null}}]\} : \Gamma, o : S}$$

[1] clearly there is no λ such that we would have $\mathcal{E}[e_1] \xrightarrow{\lambda}$ but not $e_1 \xrightarrow{\lambda}$, hence it is legitimate to use the induction hypothesis here.

then conclude $\Theta; \Gamma, o : S \rhd (h, \{o = C[\vec{f} = \overrightarrow{\mathsf{null}}]\} * r \ , \ o) : S \lhd \Gamma * r$ using T-Ref (it is not possible that r starts with o since o is fresh) and T-State.

- $e = \mathsf{switch}\ (v)\ \{l : e_l\}_{l \in E}$. Then we have two cases. The slightly more complex one is if the initial derivation is as follows:

$$\dfrac{\dfrac{\dfrac{v \text{ is a label}}{\Gamma * r \rhd v : \{v\} \lhd \Gamma * r} \qquad \Gamma(r.f) = S}{\Gamma * r \rhd v : \mathsf{link}\ f \lhd \Gamma\{r.f \mapsto \langle v : S\rangle\} * r} \qquad v \in E\ \textbf{(b)} \qquad \Gamma * r \rhd e_v : T \lhd \Gamma' * r\ \textbf{(c)}}{\Gamma * r \rhd \mathsf{switch}\ (v)\ \{l : e_l\}_{l \in E} : T \lhd \Gamma' * r}$$

(using T-Label, T-VarS, T-SwitchLink top to bottom). As usual we also have $\Theta \vdash h : \Gamma$ **(a)** as the other premise of T-State (omitted for lack of space). The reason why the initial environment of judgement (c) is Γ is because it is obtained from the version of Γ with the type of $r.f$ modified by modifying this type again, putting back S instead of the variant. (b) implies that the expression reduces by R-Switch. As regards type preservation, we can conclude $\Theta; \Gamma \rhd (h * r \ , \ e_v) : T \lhd \Gamma' * r$ directly from (a), (c), and T-State. The other case is when the T-VarS step is absent and the following rule is T-Switch instead of T-SwitchLink; the argument is the same.

- $e = f \leftrightarrow v$. Then the initial derivation is as follows:

$$\dfrac{\dfrac{\cdots}{\Gamma * r \rhd v' : T' \lhd \Gamma_1 * r\ \textbf{(b)}} \qquad \Gamma_1(r.f) = T\ \textbf{(c)} \qquad T \text{ is not a variant } \textbf{(d)}}{\Gamma * r \rhd f \leftrightarrow v' : T \lhd \Gamma_1\{r.f \mapsto T'\} * r}\ \text{(T-Swap)}$$

and we also have, as usual, $\Theta \vdash h : \Gamma$ **(a)**. The fact that $\Gamma_1(r.f)$ is defined implies that $\Gamma(r.f)$ is also defined, indeed the effect of typing v can only remove from the environment or create a variant type, so it can only decrease the set of valid field references. Thus $h(r).f$ is defined as well, and the expression reduces by R-Swap. Let $v = h(r).f$. From (a), (b), (c) and (d), we use Lemma 7.9 to get Γ'' such that $\Theta \vdash h\{r.f \mapsto v'\} : \Gamma''$. We then notice that in each of the three cases of the lemma we have $\Gamma'' * r \rhd v : T \lhd \Gamma_1\{r.f \mapsto T'\} * r$:

  (1) If v is an object identifier or channel endpoint, then $\Gamma'' = \Gamma_1\{r.f \mapsto T'\}, v : T$. We use T-Ref or T-Chan.
  (2) If v is not an object or channel and T is not a link type, then v is either null, an access point name or a label. We use T-Null, T-Name or T-Label.
  (3) If $T = \mathsf{link}\ f'$, then $\Gamma_1(r.f') = \langle l : S_l\rangle_{l \in E}$ with $v \in E$ and we have $\Gamma'' = \Gamma_1\{r.f \mapsto T'\}\{r.f' \mapsto S_v\}$. We use T-Label, T-VarS and T-SubEnv.

  Finally we conclude with T-State.

- $e = \mathsf{return}\ v$. Then the expression reduces by R-Return. The initial derivation is as follows:

$$\dfrac{(1)\ \dfrac{\cdots}{\Gamma * r.f \rhd v : T_1 \lhd \Gamma_1 * r.f} \qquad \Gamma_1(r.f) = C[F] \qquad F \vdash C : S\ \textbf{(b)}}{\Gamma * r.f \rhd \mathsf{return}\ v : T \lhd \Gamma_1\{r.f \mapsto S\} * r}\ \text{(T-Return)}$$

with also $\Theta \vdash h : \Gamma$ **(a)**. We distinguish cases depending on what rule (1) is:

  – If (1) is T-Null, T-Name or T-Label, then $\Gamma = \Gamma_1$, and if it is T-Ref or T-Chan, then $\Gamma = \Gamma_1, v : T_1$. In both cases we have $\Gamma(r.f) = \Gamma_1(r.f)$ and $T_1 = T$. From (a) and (b) we deduce $\Theta \vdash h : \Gamma\{r.f \mapsto S\}$ using the closing lemma (Lemma 7.8). We then use T-Null, T-Name, T-Label, T-Chan or T-Ref, as appropriate, to get $\Gamma\{r.f \mapsto S\} * r \rhd v : T_1 \lhd \Gamma_1\{r.f \mapsto S\} * r$, and we conclude with T-State.
  – (1) cannot be T-VarS because T-Return forbids that T_1 be of the form $\mathsf{link}\ f'$.

– If (1) is T-VarF, then $T_1 = \mathsf{variant\text{-}tag}$ and $T = \mathsf{link}\ f$. Furthermore, v is a label, $F = \langle v : F' \rangle$ with F' not a variant, and $\Gamma = \Gamma_1\{r.f \mapsto C[F']\}$. (b) then implies that S is of the form $\langle l : S_l \rangle_{l \in E}$ with $v \in E$ and $F' \vdash C : S_v$. Note that because F' is not a variant, S_v must be a branch. Now, from that judgement and (a), we use the closing lemma to get $\Theta \vdash h : \Gamma\{r.f \mapsto S_v\}$. Let $\Gamma'' = \Gamma\{r.f \mapsto S_v\}$. Since Γ only differs from Γ_1 by the type of $r.f$, it is also the case of Γ'', and as $\langle v : S_v \rangle$ is a subtype of S, we have $\Gamma''\{r.f \mapsto \langle v : S_v \rangle\} <: \Gamma_1\{r.f \mapsto S\}$. From all this, we build the following derivation:

$$\text{(T-State)}\ \dfrac{\Theta \vdash h : \Gamma\{r.f \mapsto S_v\} \qquad \text{(T-SubEnv)}\ \dfrac{\text{(T-VarS)}\ \dfrac{\text{(T-Label)}\ \dfrac{v \text{ is a label}}{\Gamma'' * r \rhd v : \{v\} \lhd \Gamma'' * r} \qquad S_v \text{ branch}}{\Gamma'' * r \rhd v : T \lhd \Gamma''\{r.f \mapsto \langle v : S_v \rangle\} * r}}{\Gamma'' * r \rhd v : T \lhd \Gamma' * r}}{\Theta; \Gamma'' \rhd (h * r\ ,\ v) : T \lhd \Gamma' * r}$$

- $e = \mathsf{spawn}\ C.m(v)$. The initial derivation involves T-Spawn, and v is null. The premise that the method exists implies that the state can reduce by R-Spawn, which corresponds to a $C.m()$ transition. The new derivation is obtained replacing T-Spawn with T-Null.
- $e = f.m(v)$. The initial derivation is as follows, with $m = m_j$ and $j \in I$:

$$\text{(T-Call)}\ \dfrac{\text{(T-Sub)}\ \dfrac{\text{(1)}\ \dfrac{\cdots}{\Gamma * r \rhd v : T' \lhd \Gamma_1 * r}}{\Gamma * r \rhd v : T'_j \lhd \Gamma_1 * r} \qquad \Gamma_1(r.f) = \{T_i\ m_i(T'_i) : S_i\}_{i \in I}\ \textbf{(b)}}{\Gamma * r \rhd f.m_j(v) : T \lhd \Gamma_1\{r.f \mapsto S_j\} * r}$$

and we also have $\Theta \vdash h : \Gamma$ **(a)**. T is obtained from T_j as specified in T-Call, i.e. replacing $\mathsf{variant\text{-}tag}$ with $\mathsf{link}\ f$ if necessary. Let $S = \{T_i\ m_i(T'_i) : S_i\}_{i \in I}$. First note that T'_j is a part of a method signature and that only a restricted set of types is allowed there: it cannot be of the form $\mathsf{link}\ f'$. Furthermore, (1) cannot be T-VarF because of (b), thus T' is not $\mathsf{variant\text{-}tag}$ either. Indeed, if $\Gamma_1(r)$ were a variant, $\Gamma_1(r.f)$ would not be defined. Therefore (1) is either T-Null, T-Label, T-Chan, T-Name or T-Ref and in all cases we have $\Gamma(r.f) = \Gamma_1(r.f)$. As it is a session type, it implies because of (a) that $h(r).f$ exists and is either an object identifier, an access point name or a channel endpoint. We distinguish these three cases:

– $h(r).f$ is an object identifier o. We use (a) and the opening lemma (Lemma 7.7) to get a field typing $C[F]$ such that $\Theta \vdash h : \Gamma\{r.f \mapsto C[F]\}$ and $F \vdash C : S$. This last judgement implies, by definition, that F is not a variant; that, among others, method m_j appears in the declaration of class C; and that, if e_j is its body and x its parameter, we have $x : T'_j, \mathsf{this} : C[F] * \mathsf{this} \rhd e_j : T_j \lhd \mathsf{this} : C[F_j] * \mathsf{this}$ and $F_j \vdash C : S_j$. The fact that the method is declared implies $(h * r\ ,\ e) \longrightarrow (h * r.f\ ,\ \mathsf{return}\ e_j\{^v/_x\})$; we now have to type this resulting state. For this, we apply the substitution lemma (Lemma 7.10) to the typing judgement for e_j, using $\Gamma_1\{r.f \mapsto C[F]\}$ as the Γ of the lemma and $r.f$ as the r of the lemma. The first case of the lemma corresponds to (1) being T-Null, T-Label or T-Name; the second one corresponds to (1) being T-Ref or T-Chan. In both cases, the resulting judgement is:

$$\Gamma\{r.f \mapsto C[F]\} * r.f \rhd e_j\{^v/_x\} : T_j \lhd \Gamma_1\{r.f \mapsto C[F_j]\} * r.f$$

Indeed, the difference between Γ and Γ_1 depends on (1) in the same way as the lemma's result. From this and $F_j \vdash C : S_j$ we can now apply T-Return and

get:

$$\Gamma\{r.f \mapsto C[F]\} * r.f \rhd \mathsf{return}\ e_j\{^v/_x\} : T \lhd \Gamma_1\{r.f \mapsto S_j\} * r$$

where T is the same as in the initial derivation. We then conclude, using the heap typing that was provided by the opening lemma, with T-State.

– $h(r).f$ is an access point name n. Then $\Gamma(r.f)$ must come, in the derivation of $\Theta \vdash h : \Gamma$, from T-Name, which implies that n is declared, that m_j is either accept or request, and that $T_j :> [\![\Sigma]\!]$ where Σ is either the declared type or its dual depending on which one m_j is. All this implies that the state does a $n[c^p]$ transition where c is fresh and p depends, again, on m_j, and that $\Theta \xrightarrow{n[c^p]} \Theta, c^p : \Sigma$. The resulting state is typed using $\Gamma'' = \Gamma, c^p : [\![\Sigma]\!]$ and T-Chan.
– $h(r).f$ is a channel endpoint c^p. Then $\Theta \vdash h : \Gamma$ implies that $c^p \in dom(\Theta)$ and $S :> [\![\Theta(c^p)]\!]$. Hence m_j is either send or receive. We distinguish the two cases. In the first case, the fact that S contains send implies that $\Theta(c^p)$ is either of the form $!\left[T''_j\right].\Sigma$ with $T'_j <: T''_j$ or $\oplus\{l : \Sigma_l\}_{l\in E}$ and then $T'_j = \{v\}$ and $v \in E$. If v is not an object identifier, then the state does a $c^p\,![v]$ transition. We can see that in both cases (send and select), Θ is able to follow that transition and evolves in such a way that $\Theta' \vdash h : \Gamma'$ holds: the session type of c^p is advanced and if v was a channel it is removed from the environment, which corresponds to the difference between Γ and Γ', thus it suffices to change the instance of T-Hempty at the root of the derivation leading to (a) to get this new typing. Then the new state is typed using T-Null and T-State. If v is an object identifier, then (1) is T-Ref and thus $v \in dom(\Gamma)$, which implies (using (a)) that v is a root of h, so the state does a $c^p\,![h \downarrow v]$ transition. We use the splitting lemma (Lemma 7.4) to see that Θ is able to follow this transition and yields a Θ' such that we have $\Theta' \vdash h \uparrow v : \Gamma'$. We can then again conclude using T-Null and T-State.
In the case where m_j is receive, the state can straightforwardly do a transition, which will be a receive on channel c^p, however the transition label is not completely determined by the original state as we do not know what will be received. So we have to prove type preservation in all cases where the transition label λ is such that $\Theta \xrightarrow{\lambda} \Theta'$ for some Θ'. If λ is of the form $c^p\,?\,[v']$, then this hypothesis tells us that $\Theta(c^p)$ is either of the form $?\,[T_0]\,.\Sigma$, and then v' must be a literal value of type T_0 or a channel endpoint which gets added to the environment with a type smaller that T_0, or of the form $\&\,\{l : \Sigma_l\}_{l\in E}$, and then $v' \in E$. In the first case we must have $T_0 <: T_j$, thus the resulting expression, which is v', can be typed using the appropriate literal value rule, or T-Chan, and subsumption. In the second one, $T_j =$ variant-tag so that $T =$ link f; the resulting expression can be typed using T-Label and T-VarS. As for the new initial environment, it is obtained, as in the case of send, by replacing the instance of T-Hempty at the top of the derivation for (a) with one using Θ' instead of Θ, so that v' gets added to the initial environment if it is a channel and that the session type of $r.f$ is correctly advanced, meaning, in the case of a branch, that it is advanced to the particular session corresponding to v', the variant type being reconstituted in the final environment by T-VarS. Finally, if λ is of

the form $c^p\,?\,[h']$, then we have $\Theta' = \Theta + \Theta''$ with $\Theta'' \vdash h' : o : T_j$, where o is the only root of h'. The merging lemma (Lemma 7.5) gives us a typing for the new heap and, as in the other cases, advancing the session type of c^p yields a session type change in $r.f$, corresponding to the difference between Γ and Γ'. We conclude using T-Ref and T-State.

□

The following two lemmas will allow us to deduce from this theorem the proof of subject reduction for configurations.

Lemma 7.18. If $\Theta \vdash s$ and $s \equiv s'$, then $\Theta \vdash s'$.

Proof. By induction on the derivation of $s \equiv s'$. □

Lemma 7.19. If $s \longrightarrow s'$, then either:

(1) $s \equiv (\nu\vec{c})\,((h * r\ ,\ e) \mid\mid s'')$,
$s' \equiv (\nu\vec{c})\,((h' * r'\ ,\ e') \mid\mid s'')$
and $(h * r\ ,\ e) \xrightarrow{\tau} (h' * r'\ ,\ e')$, or

(2) $s \equiv (\nu\vec{c})\,((h_1 * r_1\ ,\ e_1) \mid\mid (h_2 * r_2\ ,\ e_2) \mid\mid s'')$,
$s' \equiv (\nu\vec{c})(\nu d)\,((h_1 * r_1\ ,\ e_1') \mid\mid (h_2 * r_2\ ,\ e_2') \mid\mid s'')$,
$(h_1 * r_1\ ,\ e_1) \xrightarrow{n[d^+]} (h_1 * r_1\ ,\ e_1')$ and $(h_2 * r_2\ ,\ e_2) \xrightarrow{n[d^-]} (h_2 * r_2\ ,\ e_2')$, or

(3) $s \equiv (\nu\vec{c})\,((h_1 * r_1\ ,\ e_1) \mid\mid (h_2 * r_2\ ,\ e_2) \mid\mid s'')$,
$s' \equiv (\nu\vec{c})\,((h_1 * r_1\ ,\ e_1') \mid\mid (h_2 * r_2\ ,\ e_2') \mid\mid s'')$,
$(h_1 * r_1\ ,\ e_1) \xrightarrow{c^p\,![v]} (h_1 * r_1\ ,\ e_1')$ and $(h_2 * r_2\ ,\ e_2) \xrightarrow{c^p\,?[v]} (h_2 * r_2\ ,\ e_2')$, or

(4) $s \equiv (\nu\vec{c})\,((h_1 * r_1\ ,\ e_1) \mid\mid (h_2 * r_2\ ,\ e_2) \mid\mid s'')$,
$s' \equiv (\nu\vec{c})\,((h_1' * r_1\ ,\ e_1') \mid\mid (h_2' * r_2\ ,\ e_2') \mid\mid s'')$,
$(h_1 * r_1\ ,\ e_1) \xrightarrow{c^p\,![h']} (h_1' * r_1\ ,\ e_1')$ and $(h_2 * r_2\ ,\ e_2) \xrightarrow{c^p\,?[\varphi(h')]} (h_2' * r_2\ ,\ e_2')$
with $h' = h_1 \downarrow o$, $h_1' = h_1 \uparrow o$, and $h_2' = h_2 + \varphi(h')$, or

(5) $s \equiv (\nu\vec{c})\,((h * r\ ,\ e) \mid\mid s'')$,
$s' \equiv (\nu\vec{c})\,((h * r\ ,\ e') \mid\mid (o = C[\vec{f} = \overrightarrow{\mathsf{null}}] * o\ ,\ e''\{^{\mathsf{null}}/_x\}) \mid\mid s'')$
and $(h * r\ ,\ e) \xrightarrow{C.m()} (h * r\ ,\ e')$, where $C.\mathsf{fields} = \vec{f}$, o is fresh and $m(x)\ \{e''\} \in C$.

Proof. This is nothing more than a reformulation of the reduction rules in terms of labelled transitions: the derivation for $s \longrightarrow s'$ can contain any number of instances of R-Par, R-Str or R-NewChan but must have one of the other rules at the top. It is straightforward to see that depending on that top rule we are in one of the five cases listed: (1) for any of the single-thread rules in Figure 7, (2) for R-Init, (3) for R-ComBase, (4) for R-ComObj, and (5) for R-Spawn. □

We can now prove Theorem 7.13.

(Theorem 7.13). Because of Lemma 7.18 we only need to look at the different cases described in Lemma 7.19.

In cases (1) and (5), the initial derivation is as follows:

$$\text{(T-NewChan)}\ \dfrac{\text{(T-Par)}\ \dfrac{\text{(T-Thread)}\ \dfrac{\Theta_1;\Gamma \rhd (h * r\ ,\ e) : T \lhd \Gamma' * r''}{\Theta_1 \vdash (h * r\ ,\ e)} \qquad \Theta_2 \vdash s''}{\Theta_1 + \Theta_2 \vdash (h * r\ ,\ e) \mid\mid s''}}{\vdash s}$$

In case (1), Theorem 7.13 gives us $\Theta_1; \Gamma'' \rhd (h' * r' \ , \ e') : T \lhd \Gamma' * r''$; from there the final derivation is the same.

In case (5), the theorem gives us the same result, but the final derivation is more complicated as there is one more parallel component. The $C.m()$ transition tells us that e must be of the form $\mathcal{E}[\mathsf{spawn}\ C.m(v)]$. From Lemma 7.11, this implies that the subexpression $\mathsf{spawn}\ C.m(v)$ is typable, which must be a consequence of T-Spawn, implying that m appears in the initial session type S of C with a Null argument type. As, by hypothesis, the declaration of class C is well-typed, this implies (from T-Class) $x : \mathsf{Null}, \mathsf{this} : C[\overrightarrow{\mathsf{Null}}\ \vec{f}] * \mathsf{this} \rhd e'' : T \lhd \mathsf{this} : C[F] * \mathsf{this}$. We apply the substitution lemma (7.10) to this judgement to replace this with o and x with null, and we build the heap typing $\emptyset \vdash o = C[\vec{f} = \overrightarrow{\mathsf{null}}] : o : C[\overrightarrow{\mathsf{Null}}\ \vec{f}]$ from T-Hempty and T-Hadd. This gives a typing for the new thread, with an empty Θ, using T-State and T-Thread and we can conclude with T-Par.

In cases (2), (3), and (4), the initial derivation is:

$$\text{(T-NewChan)}\ \dfrac{\text{(T-Par)}\ \dfrac{\text{(T-Par)}\ \dfrac{\dfrac{\Theta_1; \Gamma_1 \rhd (h_1 * r_1 \ , \ e_1) : T_1 \lhd \Gamma'_1 * r'_1}{\Theta_1 \vdash (h_1 * r_1 \ , \ e_1)} \quad \dfrac{\Theta_2; \Gamma_2 \rhd (h_2 * r_2 \ , \ e_2) : T_2 \lhd \Gamma'_2 * r'_2}{\Theta_2 \vdash (h_2 * r_2 \ , \ e_2)}}{\Theta_1 + \Theta_2 \vdash (h_1 * r_1 \ , \ e_1) \mid\mid (h_2 * r_2 \ , \ e_2)} \quad \Theta \vdash s''}{\Theta_1 + \Theta_2 + \Theta \vdash (h_1 * r_1 \ , \ e_1) \mid\mid (h_2 * r_2 \ , \ e_2) \mid\mid s''}}{\vdash s}$$

Furthermore, we can deduce from the transition labels that the expressions in the two topmost premises are of the form $\mathcal{E}_1[f_1.m_1(v_1)]$ and $\mathcal{E}_2[f_2.m_2(v_2)]$ with $h_1(r_1).f_1$ and $h_2(r_2).f_2$ being, in case (2), n, and in cases (3) and (4), respectively c^p and $c^{\overline{p}}$. These two topmost premises must come from T-State, which implies $\Theta_1 \vdash h_1 : \Gamma_1$ and $\Theta_2 \vdash h_2 : \Gamma_2$, from which we deduce, in case (2), that n is a declared access point name and in cases (3) and (4) that $[\![\Theta_1(c^p)]\!] <: \Gamma_1(r_1.f_1)$ and $[\![\Theta_2(c^{\overline{p}})]\!] <: \Gamma_2(r_2.f_2)$. We use Theorem 7.13 on these two topmost premises and distinguish cases.

In case (2), Θ_1 and Θ_2 make transitions which introduce two dual types for d^+ and d^-, which are fresh so that the disjoint unions are still possible, and we just need to add an additional step of T-NewChan before the last one.

In cases (3) and (4), we first remark that because T-NewChan in the derivation leads to an empty environment, c must be one of the channels in $(\nu\vec{c})$ and we must have $\Theta_1(c^p) = \Sigma$ and $\Theta_2(c^{\overline{p}}) = \overline{\Sigma}$ for some Σ. Then we use Lemma 7.11 to get a typing judgement for the method call subexpression on the sending side (thread 1). This judgement has Γ_1 as an initial typing environment and comes from T-Call; as we have $\Sigma <: \Gamma_1(r_1.f_1)$, this implies that Σ is either of the form $!\,[T]\,.\Sigma'$ with v (in case (3)) or o (in case (4)) of type T, or (only in case (3)) of the form $\oplus\,\{l : \Sigma_l\}_{l \in E}$ with $v \in E$. The simplest case is (3): then this typing information, together with the duality of the two endpoint types, shows that Θ_2 follows the transition with the new type of $c^{\overline{p}}$ still dual to the new type of c^p. In the case where v is a channel endpoint, its typing goes from Θ_1 to Θ_2 but stays the same, so that it is unchanged in the sum environment yielded by T-Par. Thus we can still apply T-NewChan.

Case (4) is similar but, additionally, a renaming function is applied to the transmitted heap. We use Lemma 7.6 to see that the type of its only root, which is all we need, stays the same, so that again Θ_2 can follow the transition. We also have that a whole part of the channel environment can go from Θ_1 to Θ_2 but the effect is the same as with just one channel: it does not affect the sum environment resulting from T-Par. So again we can still apply T-NewChan. $\square$

7.3. **Type safety.** We now have the following safety result, ensuring not only race-freedom (no two sends or receives in parallel on the same endpoint of a channel) but also that the communication is successful.

Theorem 7.20 (No Communication Errors)**.** Let

$$s \equiv (\nu\vec{c})(s' \mid\mid (h * r \ , \ \mathcal{E}[r.f.m(v)]) \mid\mid (h' * r' \ , \ \mathcal{E}[r'.f'.m'(v')]))$$

and suppose that $\vdash s$ holds. If $h(r).f = c^p$ and $h'(r').f' = c^q$ then:

(1) $q = \overline{p}$,
(2) channel c does not occur in s', and
(3) there exists s'' such that $s \longrightarrow s''$.

As the statement is true in particular when s' is empty, it implies that communication between the two threads is possible.

Proof. This is an essentially straightforward consequence of $\vdash s$. The typing derivation is similar to the one shown for cases 2/3/4 in Theorem 7.13 above; the two top premises must be consequences of T-State and the heap typing necessary to apply this rule implies, respectively, $\Gamma_1(r.f) :> [\![\Theta_1(c^p)]\!]$ and $\Gamma_2(r'.f') :> [\![\Theta_2(c^q)]\!]$. Because of the disjoint unions in T-Par, $c^p \in dom(\Theta_1)$ and $c^q \in dom(\Theta_2)$ immediately imply (1) and (2); (3) is then a consequence of the duality constraint imposed by T-NewChan: looking at the translations of dual channel types, and because the method call subexpressions must be typed by T-Call, if m is send then m' must be receive and vice-versa. □

This theorem, together with the progress aspect of Theorem 7.16, restricts the set of blocked configurations to the following: if $\vdash s$ and $s \not\longrightarrow$, then all parallel components in s are either terminated (reduced to values), unmatched accepts or requests, or method calls on *pairwise distinct* channels — this last case corresponding to a deadlock.

7.4. **Conformance.** We now have the technical material necessary to prove Theorem 3.32 (conformance). Note that we do not formally extend this result to the distributed setting, as stating a similar property in that case would require more complex definitions describing, among other things, how call traces are moved around between threads; however we can see informally that, because objects keep their content and session type when transmitted, all necessary information is kept such that we still have a conformance property.

Proof. We first prove, by strong induction on n, a slightly different result, namely the following: for each i there is Γ_i such that $\Gamma_i \rhd (h_i * r_i \ , \ e_i) : T \lhd \Gamma' * r'$ and tr_i is consistent with Γ_i.

We suppose that this property is true for any reduction sequence of length n or less whose initial state satisfies the hypotheses and prove that it is true also for length $n+1$. The base case $n = 1$ is trivial.

If the nth reduction step $(h_n * r_n \ , \ e_n) \longrightarrow (h_{n+1} * r_{n+1} \ , \ e_{n+1})$ does not originate from R-Return, we use the induction hypothesis on the beginning of the sequence; we refer to the cases in the proof of Theorem 7.16 to show that the Γ_{n+1} it allows to construct from Γ_n indeed is consistent with tr_{n+1}. Because we are only interested in Γ_{n+1} and not Γ', in most cases we can use Lemmas 7.11 and 7.12 to ignore any context $\mathcal{E}$ and proceed as if the reduction is exactly an instance of its original rule.

If the rule is R-Seq, R-Switch or R-Swap then $tr_{n+1} = tr_n$.

If the rule is R-Seq or R-Switch then the proof of Theorem 7.16 shows that we can choose $\Gamma_{n+1} = \Gamma_n$, so there is nothing more to prove.

If the rule is R-Swap then the proof of Theorem 7.16 indicates that Γ_{n+1} (called Γ'' in subject reduction) can be defined using Lemma 7.9 from the Γ''' (called Γ_1 in subject reduction) obtained after typing v', the value that gets swapped into the field. First of all note that most objects, notably all those which are not v' and not in a field of r, have the same type and position in the heap in Γ_{n+1} as they have in Γ. For all them the result is straightforward: we only concentrate on those objects that move or change type. Depending on the nature of T and T' (object, link, or base type), there may be one or two of them. Recall that neither type can be variant-tag as else the expression would not be typable. We distinguish cases separately for T and T', knowing that any combination is possible (except both linking to the same field). Cases for T':

- If T' is an object type (thus v' is an object name o'), then $\Gamma_{n+1}(r.f) = \Gamma_n(o')$ (the rule used for v' is T-Ref). We also have $tr_{n+1}(h_{n+1}(r.f)) = tr_{n+1}(o') = tr_n(o')$, so tr_{n+1} is indeed consistent with Γ_{n+1} with respect to reference $r.f$.
- If T' is link f', the rule used for v' is T-VarS, and $\Gamma_{n+1}(r.f') = \langle v' : S_{v'} \rangle$. We have $\Gamma_{n+1}(r.f) = \mathsf{link}\ f'$ and $h_{n+1}(r.f) = v'$, hence the actual session type of $r.f'$ in h_{n+1} according to Γ_{n+1} is $S_{v'}$. Thus consistency is preserved for $r.f'$.

Cases for T (corresponding respectively to cases 1 and 3 of Lemma 7.9):

- If T is an object type (thus $h_n(r.f)$ is an object name o), then Γ_{n+1} contains a new entry for o, with type $\Gamma_n(r.f)$. Consistency for this new entry comes from consistency for $r.f$ at the previous step.
- If T is link f'', then $\Gamma_n(r.f') = \langle l : S_l \rangle_{l \in E}$ and $h_n(r.f) = l_0$ is in E. Thus the actual session type of $r.f''$ in h_n according to Γ_n is S_{l_0}. Lemma 7.9 also gives us $\Gamma_{n+1}(r.f') = S_{l_0}$, hence the actual session type of $r.f''$ has not changed, and consistency is preserved.

If the rule is R-New then the proof of Theorem 7.16 shows that a suitable Γ_{n+1} is of the form $\Gamma_n, o : C.\mathsf{session}$ where o is the fresh object name introduced by the reduction. Definition 3.27 states that tr_{n+1} extends tr_n by assigning an empty call trace to o; clearly tr_{n+1} is consistent with Γ_{n+1}.

If the rule is R-Call then the proof of Theorem 7.16 shows that a suitable Γ_{n+1} is Γ_n with the type of $r.f$ replaced by a type which is not a session. So there is no consistency requirement in Γ_{n+1} for $r.f$, and every other reference is given the same call trace by tr_{n+1} as by tr_n. Therefore tr_{n+1} is consistent with Γ_{n+1}.

Now if the nth step originates from R-Return, we reason slightly differently. We know by hypothesis that r_1 is a prefix of r_{n+1}. Furthermore, since the nth step is R-Return, r_n is of the form $r_{n+1}.f$. Reduction rules can only alter the current object by removing or adding one single field reference at once, therefore there must be a previous reduction step in the sequence, say the ith, that last went from r_{n+1} to $r_{n+1}.f$. That is, we chose i such that $r_{n+1}.f$ is a prefix of all r_j for j between $i+1$ and n and that $r_i = r_{n+1}$. That step must originate from R-Call as it is the only rule which adds a field specification to the current object. Thus, it is of the form $(h_i * r_{n+1}\ ,\ \mathcal{E}(f.m(v'))) \longrightarrow (h_{i+1} * r_{n+1}.f\ ,\ \mathcal{E}(\mathsf{return}\ e))$, where e is the method body of m with the parameter substituted. Then it is straightforward to see that the whole reduction sequence from $i+1$ to n consists of reductions of e inside the context $\mathcal{E}(\mathsf{return}\ [_])$.

We first use the induction hypothesis on the first part of the reduction (1 to i) so as to get judgments up to $\Gamma_i \rhd (h_i * r_{n+1}\ ,\ \mathcal{E}(f.m(v'))) : T \lhd \Gamma' * r'$. We then use Lemma 7.11 to get $\Gamma_i \rhd (h_i * r_{n+1}\ ,\ f.m(v')) : T' \lhd \Gamma'' * r''$ and note that this judgment must come from T-Call, which implies that $r'' = r_{n+1}$, that $\Gamma_i(r_{n+1}.f)$ is of the form $\{T'\ m(\ldots) : S, \ldots\}$ and that $\Gamma''(r_{n+1}.f) = S$. Furthermore, T' is either a base type if S is a branch or $\mathsf{link}\ f$ if it is a variant. We know that tr_i is consistent with Γ_i, therefore we have $\mathsf{class}(o).\mathsf{session} \xrightarrow{tr_i(o)}{}^{*} \{T'\ m(\ldots) : S, \ldots\} \xrightarrow{m} S$.

We now use the induction hypothesis again on the reduction sequence from i to n for this particular call subexpression, recalling that i has been defined such that the hypothesis on the current object is indeed satisfied by this sequence. We can also use Lemma 7.12 at each step in order to lift the judgements thus obtained to the whole expression. To summarise, this means that for any j between $i+1$ and n we have: $e_j = \mathcal{E}(\mathsf{return}\ e'_j)$ for some e'_j, $\Gamma_j \rhd (h_j * r_j\ ,\ \mathsf{return}\ e'_j) : T' \lhd \Gamma'' * r_{n+1}$ and $\Gamma_j \rhd (h_j * r_j\ ,\ e_j) : T \lhd \Gamma' * r'$, and that tr_j is consistent with Γ_j.

For the last reduction step, R-Return, the proof of Theorem 7.16 tells us that we can choose a Γ_{n+1} which is identical to Γ_n except for the type of $r_{n+1}.f$, and as the call trace for other references is not modified, consistency is preserved for them. For $r_{n+1}.f$ we have to look back at the initial subexpression on step i. First note that R-Swap can only act on a field of the current object, therefore since $r_{n+1}.f$ is a prefix of the current object during the whole subsequence, its content cannot change and is the same object o throughout. Similarly, there is no other R-Call or R-Return acting on that particular object, hence $tr_n(o) = tr_{i+1}(o) = tr_i(o)m$. We saw above that this call trace leads the initial session of o to S. Then the judgement for the final subexpression, at step $n+1$, is of the form $\Gamma_{n+1} \rhd (h_{n+1} * r_{n+1}\ ,\ v) : T' \lhd \Gamma'' * r_{n+1}$. There are two cases, as in the proof of Theorem 7.16. If T' is a base type then S is a branch and it is possible to decide that $\Gamma_{n+1}(r_{n+1}.f)$ is equal to S. In that case the call trace either does not change or has a label appended, but as S is a branch it can do a transition to itself with any label, therefore $tr_{n+1}(o)$ is consistent with $\Gamma_{n+1}(r_{n+1}.f)$ in both cases. If T' is $\mathsf{link}\ f$, then v is a label, S is a variant $\langle l : S_l \rangle_{l \in E}$ and $\Gamma_{n+1}(r_{n+1}.f)$ can be chosen equal to S_v. We have $tr_{n+1}(o) = tr_n(o)v$ and $S \xrightarrow{v} S_v$, so consistency is preserved.

This completes the inductive proof that for every step i in the reduction sequence there is Γ_i such that $\Gamma_i \rhd (h_i * r_i\ ,\ e_i) : T \lhd \Gamma' * r'$ and tr_i is consistent with Γ_i. This fact obviously implies that tr_i is valid for all the objects which have a session type in Γ_i; we now argue that it is also the case for the other objects, namely those which either are not at all in Γ_i or do not have a session type. We know by hypothesis that it is the case for tr_1 and show by a very simple induction that it cannot change from i to $i+1$. The ith step can only change the call trace for an object o if it originates from R-Call or R-Return concerning that object. R-Call can only occur if the reducible part of the expression is indeed a method call on a field which contains o, and that is only typable if Γ_i contains a session type for that field which is a branch containing the method, and thus allows the appropriate transition: therefore validity of the call trace for o is preserved in that case. R-Return on the other hand can only occur if the reducible part of the expression is a return and if the current object is (the address of) o, and we saw that in that case the Γ_{i+1} constructed in our proof contains a session type for o, so this case is covered by the consistency result. $\square$

$\mathcal{W}(C) = \mathcal{A}_C(C.\mathsf{session},\ C.\mathsf{fields},\ \emptyset)$
if for every $\mathsf{req}\ F\ \mathsf{ens}\ F'\ \mathsf{for}\ T'\ m(T\ x)\ \{e\} \in C$
$F' \neq \langle _ \rangle$ and $\mathcal{B}_C(e,\ F,\ x : T) = (T',\ F',\ _)$
$\mathcal{A}_C(S,\ F,\ \Delta) = \Delta$ if $(F, S) \in \Delta$
$\mathcal{A}_C(\mu X.S,\ F,\ \Delta) = \mathcal{A}_C(S\{^{\mu X.S}/_X\},\ F,\ \Delta \cup \{(F, \mu X.S)\})$
$\mathcal{A}_C(\{T_i\ m_i(U_i) : S_i\}_{1 \leqslant i \leqslant n},\ F,\ \Delta_0) = \Delta_n$
where for $i = 1$ to n
let $(T'_i,\ F'_i,\ _) = \mathcal{B}_C(e_i,\ F,\ x_i : U_i)$ where $m_i(x_i)\ \{e_i\} \in C$
if $T'_i <: T_i$ then let $\Delta_i = \mathcal{A}_C(S_i,\ F'_i,\ \Delta_{i-1})$
else if T'_i is an enumeration E and $T_i = \mathsf{variant\text{-}tag}$
then let $\Delta_i = \mathcal{A}_C(S_i,\ \langle l : F'_i \rangle_{l \in E},\ \Delta_{i-1})$
else if $T'_i = \mathsf{variant\text{-}tag}$ and T_i is an enumeration E and $F'_i = \langle l : F_l \rangle_{l \in E'}$ and $E' \subseteq E$
then let $\Delta_i = \mathcal{A}_C(S_i,\ \bigvee_{l \in E'} F_l,\ \Delta_{i-1})$
$\mathcal{A}_C(\langle l : S_l \rangle_{l \in E},\ \langle l : F_l \rangle_{l \in E'},\ \Delta_0) = \Delta_n$
where $E' = \{l_1 \ldots l_n\} \subseteq E$ and for $i = 1$ to n, $\Delta_i = \mathcal{A}_C(S_{l_i},\ F_{l_i},\ \Delta_{i-1})$

Combining variants

$\{T_i\ f_i\}_{i \in I} \vee \{T'_i\ f_i\}_{i \in I} = \{(T_i \vee T'_i)\ f_i\}_{i \in I}$
$\langle l : F_l \rangle_{l \in I} \vee \langle l : F'_l \rangle_{l \in J} = \langle l : F''_l \rangle_{l \in I \cup J}$
where $F''_l = F_l \vee F'_l$ if $l \in I \cap J$, F_l if $l \notin J$, F'_l if $l \notin I$

Figure 28: Typechecking: algorithms $\mathcal{W}$ and $\mathcal{A}$.

8. Type Checking Algorithm

This section introduces a type checking algorithm, sound and complete with respect to the type system in Section 6, and describes a prototype implementation of a programming language based on the ideas of the paper.

8.1. **The Algorithm.** Figures 28 and 29 define a type checking algorithm for the distributed language, including the sequential extensions from Section 4. The algorithm is applied to each component of a distributed system, and in order to ensure type safety of the complete system there must be some separate mechanism to check that each access point n is given the same type everywhere. A program is type checked by calling algorithm $\mathcal{W}$ on each class definition and checking that no call generates an error. The definition of algorithm $\mathcal{W}$ follows the typing rule T-Class in Figure 15. It calls algorithm $\mathcal{A}$ to check the relation $F \vdash C : S$ and algorithm $\mathcal{B}$ to type check the bodies of the methods that have req/ens annotations. Algorithm $\mathcal{A}$ also calls algorithm $\mathcal{B}$ to typecheck the bodies of the methods that appear in the session type.

In both $\mathcal{A}$ and $\mathcal{B}$ there are several "if" and "where" clauses; they should be interpreted as conditions which, if not satisfied, cause termination with a typing error.

Because of the coinductive definition of $F \vdash C : S$, algorithm $\mathcal{A}$ uses a set Δ of assumed relationships between field typings F and session types S. If there is no error then the algorithm returns Δ, but at the top level we are only interested in success or failure, not in the returned value.

Algorithm $\mathcal{B}$ checks the typing judgement for expressions, defined in Figure 10, specialized to the top-level form $\mathsf{this} : C[F], V * \mathsf{this} \rhd e : T \lhd \mathsf{this} : C[F'], V' * \mathsf{this}$ as explained in Section 3.5.1. The definition of $\mathcal{B}$ follows the typing rules (Figure 10) except for one point: T-VarF means that the rules are not syntax-directed, as any expression with type E can also be given type variant-tag. For this reason, clause l of $\mathcal{B}$ produces type variant-tag and a

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

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

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

$\mathcal{B}_C(f \leftrightarrow e,\, F,\, V) = (U,\, F'',\, V')$
where $(T,\, F',\, V') = \mathcal{B}_C(e,\, F,\, V)$ and $F'(f)$ is not a variant and
if $T = \mathsf{variant\text{-}tag}$ then $F' = \langle l : F_l\rangle_{l\in E}$ and $U = (\bigvee_{l\in E} F_l)(f)$ and $F'' = (\bigvee_{l\in E} F_l)\{f \mapsto E\}$
else $U = F'(f)$ and $F'' = F'\{f \mapsto T\}$

$\mathcal{B}_C(l,\, F,\, V) = (\mathsf{variant\text{-}tag},\, \langle l : F\rangle,\, V)$

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

$\mathcal{B}_C(f.m_j(e),\, F,\, V) = (T,\, F'',\, V')$
where $(T',\, F',\, V') = \mathcal{B}_C(e,\, F,\, V)$ and
if $T' = \mathsf{variant\text{-}tag}$ then $F' = \langle l : F_l\rangle_{l\in E}$ and $(\bigvee_{l\in E} F_l)(f) = \{T_i\ m_i(T'_i) : S_i\}_{i\in I}$ and $j \in I$
and T'_j is an enumeration E' and $E \subseteq E'$ and $F'' = (\bigvee_{l\in E} F_l)\{f \mapsto S'_j\}$ and
$T = \mathsf{link}\ f$ if $T_j = \mathsf{variant\text{-}tag}$, $T = T_j$ otherwise
else $F'(f) = \{T_i\ m_i(T'_i) : S_i\}_{i\in I}$ and $j \in I$ and $T' <: T'_j$ and
$F'' = F'\{f \mapsto S'_j\}$ and $T = \mathsf{link}\ f$ if $T_j = \mathsf{variant\text{-}tag}$, $T = T_j$ otherwise

$\mathcal{B}_C(m(e),\, F,\, V) = (T',\, F''',\, V')$
where $(T,\, F',\, V') = \mathcal{B}_C(e,\, F,\, V)$ and $\mathsf{req}\ F''\ \mathsf{ens}\ F''\ \mathsf{for}\ T'\ m(T''\ x)\ \{e\} \in C$ and
if $T = \mathsf{variant\text{-}tag}$ then $F' = \langle l : F_l\rangle_{l\in E}$ and T'' is an enumeration E' and
$E \subseteq E'$ and $\bigvee_{l\in E} F_l <: F''$
else $T <: T''$ and $F' <: F''$

$\mathcal{B}_C(\mathsf{switch}\ (e)\ \{l : e_l\}_{l\in E},\, F,\, V) = (T,\, \bigvee_{l\in E} F''_l,\, V'')$
where $(U,\, F',\, V') = \mathcal{B}_C(e,\, F,\, V)$ and
if $U = E'$ then $E' \subseteq E$ and $\forall l \in E'.(T,\, F''_l,\, V'') = \mathcal{B}_C(e_l,\, F',\, V')$
else if $U = \mathsf{variant\text{-}tag}$ then $F' = \langle m : G_m\rangle_{m\in E'}$ and $E' \subseteq E$ and
$\forall l \in E'.(T,\, F''_l,\, V'') = \mathcal{B}_C(e_l,\, \bigvee_{m\in E'} G_m,\, V')$
else if $U = \mathsf{link}\ f$ then
$F'(f) = \langle l : S_l\rangle_{l\in E'}$ and $E' \subseteq E$ and $\forall l \in E'.(T,\, F''_l,\, V'') = \mathcal{B}_C(e_l,\, F'\{f \mapsto S_l\},\, V')$

$\mathcal{B}_C(\mathsf{while}\ (e)\ \{e'\},\, F,\, V) = (\mathsf{Null},\, F'',\, V')$
where $(U,\, F',\, V') = \mathcal{B}_C(e,\, F,\, V)$ and
if $U = E'$ then $E' \subseteq \{\mathsf{True}, \mathsf{False}\}$ and $\mathcal{B}_C(e',\, F',\, V') = (\mathsf{Null},\, F,\, V)$ and $F'' = F'$
else if $U = \mathsf{variant\text{-}tag}$ then $F' = \langle l : F_l\rangle_{l\in E}$ and $E \subseteq \{\mathsf{True}, \mathsf{False}\}$ and
$\mathcal{B}_C(e',\, \bigvee_{l\in E} F_l,\, V') = (\mathsf{Null},\, F,\, V)$ and $F'' = \bigvee_{l\in E} F_l$
else if $U = \mathsf{link}\ f$ then $F'(f) = \langle \mathsf{True} : S_{\mathsf{True}}, \mathsf{False} : S_{\mathsf{False}}\rangle$ and
$\mathcal{B}_C(e',\, F'\{f \mapsto S_{\mathsf{True}}\},\, V') = (\mathsf{Null},\, F,\, V)$ and $F'' = F'\{f \mapsto S_{\mathsf{False}}\}$

$\mathcal{B}_C(e; e',\, F,\, V) = \mathcal{B}_C(e',\, F'',\, V')$
where $(T,\, F',\, V') = \mathcal{B}_C(e,\, F,\, V)$ and $T \neq \mathsf{link}\ _$ and
if $T = \mathsf{variant\text{-}tag}$ then $F' = \langle l : F_l\rangle_{l\in E}$ and $F'' = \bigvee_{l\in E} F_l$
else $F'' = F'$

$\mathcal{B}_C(\mathsf{spawn}\ C'.m(e),\, F,\, V) = (\mathsf{Null},\, F',\, V')$
where $(\mathsf{Null}, F', V') = \mathcal{B}_C(e,\, F,\, V)$ and $\mathsf{Null}\ m(\mathsf{Null}) \in C'.\mathsf{session}$

Figure 29: Typechecking: algorithm $\mathcal{B}$.

variant field typing with the single label l. More general variant field typings are produced when typing switch expressions, as the $\vee$ operator is used to combine the field typings arising from the branches. This is the typical situation when typing the body of a method whose return type is variant-tag: the body contains a switch whose branches return different labels with different associated field typings.

It is possible, however, that giving type variant-tag to l is incorrect. It might turn out that the expression needs to have an enumerated type E, for example in order to be passed

```
class C {
  session { linkthis m(int): ⟨ FALSE: SCf, TRUE: SCt ⟩ }
  where SCf = ..., SCt = ...
  ...
}

class D {
  session { linkthis a(int): ⟨ FALSE: SDf, TRUE: SDt ⟩,
            linkthis b(int): ⟨ FALSE: SDf, TRUE: SDt ⟩,
            { FALSE, TRUE } c(int): SD1,
            { FALSE, TRUE, UNKNOWN } d(int): SD2 }
  where SDf = ..., SDt = ..., SD1 = ..., SD2 = ...

  f;

  a(x) { // Not allowed, because return type is link f
    f ↔ new C();
    f.m(x); }

  aa(x) { // Allowed, because body type is linkthis
    f ↔ new C();
    switch (f.m(x)) {
      case FALSE: FALSE;
      case TRUE: TRUE; } }

  b(x) { // Allowed, by creating a uniform variant
    even(x); }

  bb(x) { // Allowed, because body type is linkthis
    switch (even(x)) {
      case FALSE: FALSE;
      case TRUE: TRUE; } }

  c(x) { // Allowed, by taking a join of field typings
    f ↔ new C();
    switch (f.m(x)) {
      case FALSE: FALSE;
      case TRUE: TRUE; } }

  cc(x) { // Allowed, by taking a join of equal field typings
    switch (even(x)) {
      case FALSE: FALSE;
      case TRUE: TRUE; } }

  d(x) { // Allowed, because of subtyping between enumerations
    even(x); }
}
```

Figure 30: Example for type checking.

as a method parameter or returned as a method result of type E. An expression that has been inappropiately typed with variant-tag can, in general, be associated with any variant field typing, for example if it contains a switch whose branches yield different field typings. In this case, the algorithm uses $\vee$ to combine the branches of the variant field typing into a

single field typing; the join is always over all of the labels in the variant. This happens in several places in algorithm $\mathcal{B}$, indicated by conditions of the form "if $T =$ variant-tag", and in the final "else" branch of the third clause of algorithm $\mathcal{A}$.

The algorithm for checking subtyping is not described here but is similar to the one defined for channel session types by Gay and Hole [37]. We write $S \vee S'$ for the least upper bound of S and S' with respect to subtyping. It is defined by taking the intersection of sets of methods and the least upper bound of their continuations. Details of a similar definition (greatest lower bound of channel session types) can be found in the work of Mezzina [52].

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

We have not yet investigated type inference, but there are two ways in which it might be beneficial. One would be to infer the req/ens annotations. The other would be to support some form of polymorphism over field typings, along the lines that if method m does not use field f then it should be callable independently of the type of f. This might reduce the need to type check the definition of m every time it occurs in the session type.

8.2. **Examples of Type Checking.** Figure 30 defines classes C and D. In class C, only the outer layer of the session type is of interest; the example uses an object of class C but does not need the definition of method m. Class D, as well as the outer layer of the session type, contains a field f and one or two candidate definitions for each of the methods a, b, c and d. The definitions of a and aa are alternatives for the method a specified in the session type, and so on.

The definition of a is not typable because the type of the returned expression is link f. Allowing this would let the caller of a have access to field f. Instead, the result of f.m(x) must be analyzed with a **switch**, as in the definition of aa, which is typable. The **linkthis** type required by the signature of a is introduced by the enumeration labels FALSE and TRUE in the branches of the **switch**. A compiler could insert **switch**es of this kind automatically, allowing the definition of a as syntactic sugar.

The remaining method definitions are all typable and illustrate different features of the type system and the algorithm. In the definition of b, the method even is supposed to be the obvious function for testing parity of an integer, returning TRUE or FALSE. This definition is typable even though the body of b does not introduce a **linkthis** type, because algorithm $\mathcal{A}$ constructs a variant field typing over {TRUE,FALSE} in which both options are the same. This is seen in the first *else* clause of $\mathcal{A}$. The definition of bb achieves the same effect by using the labels FALSE and TRUE to introduce the type **linkthis**. Each label corresponds to a partial variant field typing, and checking the **switch** combines them by means of the $\vee$ operator. Because the field f is not involved in the method body, the field typing is the same in both options of the variant.

Method c has the same definition as a, but this time the signature in the session type specifies a simple enumeration as the return type. This is allowed, by using the $\vee$ operator to construct the join of the field typings, in the second *else* clause of $\mathcal{A}$. This means that when the algorithm proceeds to type check method definitions in the session type SD1, the type of f is taken to be the join of SCf and SCt. Whether or not this loss of information causes a problem will depend on the particular definitions of those types, which we have not shown. Method cc is handled in the same way, but this time there is no loss of information because the types being joined are identical; this in turn is because f is not involved in the method body.

Finally, method d illustrates straightforward subtyping between enumerations, defined as set inclusion.

8.3. **Correctness of the Algorithm.** The following sequence of results outlines the proof of soundness and completeness of the algorithm. The detailed proofs are routine and are omitted.

Theorem 8.1. Algorithm $\mathcal{A}$ always terminates, either with an error (and then the function $\mathcal{A}$ is undefined) or with a result.

Proof. Similar to proofs about algorithms for coinductively-defined subtyping relations [63, Chapter 16]. □

Lemma 8.2. If $\mathsf{this} : C[F], V * \mathsf{this} \rhd e : \mathsf{variant\text{-}tag} \lhd \mathsf{this} : C[F'], V' * \mathsf{this}$ then for some E and $\{F_l\}_{l \in E}$, $F' = \langle l : F_l \rangle_{l \in E}$ and $\mathsf{this} : C[F], V * \mathsf{this} \rhd e : E \lhd \mathsf{this} : C[\bigvee_{l \in E} F_l], V' * \mathsf{this}$.

Proof. By induction on the typing derivation. □

Lemma 8.3. If $\mathcal{B}_C(e, F, V) = (T, F', V')$ then $\mathsf{this} : C[F], V * \mathsf{this} \rhd e : T \lhd \mathsf{this} : C[F'], V' * \mathsf{this}$.

Proof. By induction on the structure of e. □

Lemma 8.4. If $\mathsf{this} : C[F], V * \mathsf{this} \rhd e : T \lhd \mathsf{this} : C[F'], V' * \mathsf{this}$ and $\mathcal{B}_C(e, F, V) = (T', F'', V'')$ then $V'' <: V'$ and either

(1) $T' <: T$ and $F'' <: F'$, or
(2) $T = \mathsf{variant\text{-}tag}$, T' is an enumeration E, $F' = \langle l : F_l \rangle_{l \in E'}$, $E \subseteq E'$ and $\forall l \in E.\ F'' <: F_l$, or
(3) T is an enumeration E, $T' = \mathsf{variant\text{-}tag}$, $F'' = \langle l : F_l \rangle_{l \in E'}$, $E' \subseteq E$ and $\forall l \in E'.\ F_l <: F'$.

Proof. By induction on the typing derivation. □

Theorem 8.5. If $F \vdash C : S$ then $\mathcal{A}_C(S, F, \emptyset)$ is defined.

Proof. Consider the execution of $\mathcal{A}_C(S, F, \emptyset)$. It terminates and has various calls of the form $\mathcal{A}_C(S', F', \Delta)$, including the top-level call. We prove the following statement, by induction on the number of recursive calls in the execution of $\mathcal{A}_C(S', F', \Delta)$: if $\Delta \subseteq \bullet \vdash C : \bullet$ and $F' \vdash C : S'$ then $\mathcal{A}_C(S', F', \Delta)$ is defined and $\mathcal{A}_C(S', F', \Delta) \subseteq \bullet \vdash C : \bullet$. □

Lemma 8.6. If $\mathcal{A}_C(S, F, \Delta)$ is defined then for any Δ', $\mathcal{A}_C(S, F, \Delta \cup \Delta') = \mathcal{A}_C(S, F, \Delta) \cup \Delta'$.

Proof. Similar to the proof of Theorem 8.5, by induction on the recursive calls within a given top-level call. □

Lemma 8.7. Suppose $\mathcal{A}_C(\mu X.S_0, F_0, \emptyset)$ is defined and $(F_0, \mu X.S_0) \notin \Delta$. Then for all S and F, if $\mathcal{A}_C(S, F, \Delta \cup \{(F_0, \mu X.S_0)\})$ is defined then $\mathcal{A}_C(S, F, \Delta)$ is defined.

Proof. Similar to the proof of Theorem 8.5, by induction on the recursive calls within a given top-level call. □

Lemma 8.8. If $\mathcal{A}_C(\mu X.S, F, \emptyset)$ is defined then $\mathcal{A}_C(S\{^{\mu X.S}/_X\}, F, \emptyset)$ is defined.

Proof. By the definition of $\mathcal{A}$, $\mathcal{A}_C(\mu X.S, F, \emptyset) = \mathcal{A}_C(S\{^{\mu X.S}/_X\}, F, \{(F, \mu X.S)\})$, which is therefore defined. By Lemma 8.7, $\mathcal{A}_C(S\{^{\mu X.S}/_X\}, F, \emptyset)$ is defined. □

Corollary 8.9. If $\mathcal{A}_C(S,\, F,\, \emptyset)$ is defined then $\mathcal{A}_C(\mathsf{unfold}(S),\, F,\, \emptyset)$ is defined.

Theorem 8.10. If $\mathcal{A}_C(S,\, F,\, \emptyset)$ is defined then $F \vdash C : S$.

Proof. By Corollary 8.9 and the fact that $F \vdash C : S$ is defined in terms of the unfolded structure of session types, it is sufficient to consider the case in which S is guarded.

Similarly to the proof of Theorem 8.5, consider the recursive calls in the execution of $\mathcal{A}_C(S_0,\, F_0,\, \emptyset)$. We show that the following relation is a C-consistency relation:

$$\mathcal{R} = \{(F, S) \mid \mathcal{A}_C(S,\, F',\, \Delta) \text{ is called for some } \Delta \text{ and } F' \text{ with } F <: F'\}.$$

This is easily checked, using the three cases of Lemma 8.4 to correspond to the three cases in the third clause of the definition of $\mathcal{A}$. □

8.4. **Implementation.** The ideas introduced in this paper can be used to extend a conventional Java compiler, by including @session annotations in classes and in method parameters, as well as @req and @ens annotations for recursive methods (cf. Section 4.5). The extension only concerns type checking; there is no need to touch the back-end of the compiler.

To keep in line with the expectations of Java programmers, annotations follow the first style in Figure 1, page 4. Also, the type system is nominal (cf. Section 4.7); label sets (cf. Figure 5) are explicitly introduced via **enum** declarations. The concepts contained in our core language can then be extended towards the whole of Java. In particular:

- The **while** loop technique described in Section 4.4 can be extended to handle **for** and **do**−**while** loops.
- The same idea can be used to type the various goto instructions present in Java: exceptions, **break**, **continue** and **return**, labelled versions included.
- All control flow instructions (including **if**−**then**, not discussed in the paper) can be used with conventional or with session-related **boolean**/**enum** values.
- Classes not featuring a @session annotation are considered shared rather than linear. Their objects can be treated very much like the **null** value (cf. Section 4.6). We do not allow a shared class to contain a linear field, even though it is perfectly acceptable for a method of a shared class to have a linear parameter.
- The same technique used for "top-level" classes can be used for inner, nested, local (defined within methods) and anonymous classes.
- In order to mention overloaded methods in @session annotations, alias names for these methods can be introduced via extra annotations.
- Static fields are always shared.
- Class inheritance (cf. Section 4.7) can be supported.

We have used the Polyglot [61] system for an initial prototype extension of Java, but a more thorough design and implementation are left for future work.

9. Related Work

There is a large amount of related work, originating from several different approaches. Our discussion of related work is organised according to these approaches.

Previous work on session types for object-oriented languages. Dezani-Ciancaglini, Yoshida et al. [17, 27, 28, 30] have taken an approach in which a class define sessions *instead of*

methods. Invoking a session on an object creates a channel which is used for communication between two blocks of code: the body of the session, and a *co-body* defined by the invoker of the session. A session is therefore a generalization of a method, in which there can be an extended dialogue between caller and callee instead of a single exchange of parameters and result. The structure of this dialogue is defined by a session type. This approach proposes a new paradigm for concurrent object-oriented programming, and as far as we know it has not yet been implemented. In contrast, our approach maintains the standard execution model of method calls.

The SJ (Session Java) language, developed by Hu [45], is a less radical extension of the object-oriented paradigm. Channels, described by session types, are essentially the same as those in the original work based on process calculus. Program code is located in methods, as usual, and can create channels, communicate on them, and pass them as messages. SJ has a well-developed implementation and has been applied to a range of situations. However, SJ has one notable restriction: a channel cannot be stored in a field of an object. This means that a channel, once created, must be either completely used, or else delegated (sent along another channel), within the same method. It is possible for a channel to be passed as a parameter to another method, but it is not possible for a session to be split into methods that can be called separately, each implementing part of the session type of a channel that is stored in a field. A distinctive feature of our work is that we can store a channel in a field of an object and allow several methods to use it. This is illustrated in Figures 18 and 22.

Hu et al. [44, 59] have also extended SJ to support event-driven programming, with a session type discipline to ensure safe event handling and progress. We have not considered event-driven programming in our setting.

Campos and Vasconcelos [15, 16] developed MOOL, a simple class-based object-oriented language, to study object usage and access. The novelties are that class usage types are attached to class definitions, and the communication mechanism is based on method call instead of being channel-based. The latter feature is the main difference with respect to our work.

Non-uniform concurrent objects/active objects. Another related line of research, started by Nierstrasz [60], aimed at describing the behaviour of non-uniform *active* objects in concurrent systems, whose behaviour (including the set of available methods) may change dynamically. He defined subtyping for active objects, but did not formally define a language semantics or a type system. The topic has been continued, in the context of process calculi, by several authors [13, 14, 21, 22, 56, 57, 65, 66, 67]. The work by Caires [13] is the most relevant work; it uses an approach based on spatial types to give very fine-grained control of resources, and Militão [53] has implemented a Java prototype based on this idea. Damiani et al. [24] define a concurrent Java-like language incorporating inheritance and subtyping and equipped with a type-and-effect system, in which method availability is made dependent on the state of objects.

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

Specifically related to the notion of subtyping between session types, the work of Rossie [68] is worth mentioning. He proposes a type-based approach to ensure that both component objects and their clients have compatible protocols. The typing discipline specifies not only how to use the component's methods, but also the notifications it sends to its

clients. Rossie calls this enhanced specification a Logical Observable Entity (LOE), which is a finite-state machine equipped with a subtyping notion. An LOE is a high-level description of an object, specifying which transitions (method executions) change its state, providing for each state both the available methods and notifications to be sent to the clients. LOEs support behavioural subtyping, in its afferent aspects (how clients may affect the LOE) — a subtype must allow at least the traces of its supertype, and in its efferent aspects (how a LOE processing a method request has effects on clients) — the subtype must not send more notifications than the supertype. This behavioural subtyping notion on finite-state machines, which is in its spirit very similar to the one of session types — "more offers, less requests", is defined as a simulation relation. Rossie shows that this relation ensures safe substitutability.

Typestate. Based on the fact that method availability depends on an object's internal state (the situation identified by Nierstrasz, as mentioned above), Strom and Yemini [69] proposed *typestate*. The concept consists of identifying the possible states of an object and defining pre- and post-conditions that specify in which state an object should be so that a given method would be available, and in which state the method execution would leave the object.

Vault [25, 32] follows the typestate approach. It uses linear types to control aliasing, and uses the *adoption and focus* mechanism [32] to re-introduce aliasing in limited situations. *Fugue* [26, 33] extends similar ideas to an object-oriented language, and uses explicit pre- and post-conditions.

Bierhoff and Aldrich [7] also work on a typestate approach in an object-oriented language, defining a sound modular automated static protocol-checking setting. They define a state and method refinement relation achieving a behavioural subtyping relation. The work is extended with access permissions, that combine typestate with aliasing information about objects [8], and with concurrency, via the atomic block synchronization primitive used in transactional memory systems [5]. Like us, they allow the typestate to depend on the result of a method call. *Plural* is a prototype language implementation that embodies this approach, providing automated static analysis in a concurrent object-oriented language [9]. To evaluate their approach they annotated and verified several standard Java APIs [10].

Militão et al. [54] develop a new aliasing control mechanism, finer and more expressive than previous proposals, based on defining object views according to specific access constraints. The discipline is implemented in a type system combining views and a typestate approach, checking user defined aliasing patterns.

Sing# [31] is an extension of C# which has been used to implement Singularity, an operating system based on message-passing. It incorporates session types to specify protocols for communication channels, and introduces typestate-like *contracts*. Bono et al. [12] have formalised a core calculus based on *Sing#* and proved type safety. 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.

Aldrich et al. [1] have proposed *typestate-oriented programming*. The aim is to integrate typestate into language design from the beginning, instead of adding typestate constraints to an existing language. Their prototype language is called Plaid. Instead of class definitions, a program consists of state definitions; each state has methods which cause transitions to other states when they are called. Like classes, states are organised into an inheritance

hierarchy. The specifications of state transitions caused by methods are similar to the pre- and post-conditions of Plural. Aliasing is managed by a system of access permissions [8]. More recent work [35, 75] combines gradual typing and typestate, to integrate static and dynamic typestate checking.

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

Affine types. Tov and Pucella [71] have developed Alms, a language in the style of OCaml with an affine type system as a generalisation of linear typing. Alms is a general-purpose programming language, in which the affine type system provides an infrastructure suitable for defining a variety of type-based resource control patterns including alias control, session types and typestate. It has been implemented, and type safety has been proved for a formal calculus. Representing a particular approach to typestate, such as our specifications of allowed sequences of method calls, would require an encoding; in contrast, our language aims to provide a convenient high-level programming style.

Static verification of protocols. *Cyclone* [40] and *CQual* [34] are systems based on the C programming language that allow protocols to be statically enforced by a compiler. *Cyclone* adds many benefits to C, but its support for protocols is limited to enforcing locking of resources. Between acquiring and releasing a lock, there are no restrictions on how a thread may use a resource. In contrast, our system uses types both to enforce locking of objects (via linearity) and to enforce the correct sequence of method calls. *CQual* expects users to annotate programs with type qualifiers; its type system, simpler and less expressive than the above, provides for type inference.

Unique ownership of objects. In order to demonstrate the key idea of modularizing session implementations by integrating session-typed channels and non-uniform objects, we have taken the simplest possible approach to ownership control: strict linearity of non-uniform objects. This idea goes back at least to the work of Baker [4] 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* [41], Almeida's *balloon types* [2], Clarke et al.'s *ownership types* [20], Fähndrich and DeLine's *adoption and focus* [32], Östlund et al.'s Joe_3 [62] 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 [48] 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. Igarashi and Kobayashi have developed a generic framework [47] including a verification tool [49] in which to define type systems for analyzing various behavioural properties including sequences of resource uses [50]. In some of this work, types are themselves abstract processes, and therefore in some situations resemble our session types. Chaki at al. [19] use CCS to describe properties of pi-calculus programs, and verify the validity of temporal formulae via a combination of type-checking and model-checking techniques, thereby going beyond static analysis.

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

10. Conclusion

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

We have formalized the syntax, operational semantics and static type system of a core distributed class-based object-oriented language incorporating these ideas. Soundness of the type system is expressed by type preservation, conformance and correct communication theorems. The type system includes a form of typestate and uses simple linear type theory to guarantee unique ownership of non-uniform objects. It allows the typestate of an object after a method call to depend on the result of the call, if this is of an enumerated type, and in this situation, the necessary case-analysis of the method result does not need to be done immediately after the call.

We have illustrated our ideas with an example based on a remote file server, and described a prototype implementation. By incorporating further standard ideas from the

related literature, it should be straightforward to extend the implementation to a larger and more practical language.

In the future we intend to work on the following topics. (1) More flexible control of aliasing. The mechanism for controlling aliasing should be orthogonal to the theory of how operations affect uniquely-referenced objects. We intend to adapt existing work to relax our strictly linear control and obtain a more flexible language. (2) In Section 4.7 we outlined an adaptation of our structural type system to a nominal type system as found in languages such as Java. We would also like to account for Java's distinction and relationship between classes and interfaces. (3) Specifications involving several objects. Multi-party session types [11, 42] and conversation types [14] specify protocols with more than two participants. It would be interesting to adapt those theories into type systems for more complex patterns of object usage.

Acknowledgements

We thank Jonathan Aldrich and Luís Caires for helpful discussions. Gay was partially supported by the UK EPSRC (EP/E065708/1 "Engineering Foundations of Web Services", EP/F037368/1 "Behavioural Types for Object-Oriented Languages", EP/K034413/1 "From Data Types to Session Types: A Basis for Concurrency and Distribution" and EP/L00058X/1 "Exploiting Parallelism through Type Transformations for Hybrid Many-core Systems"). He thanks the University of Glasgow for the sabbatical leave during which part of this research was done. Gay and Ravara were partially supported by the Security and Quantum Information Group at Instituto de Telecomunicações, Portugal. Vasconcelos was partially supported by the Large-Scale Informatics Systems Laboratory, Portugal. Ravara was partially supported the Portuguese Fundação para a Ciência e a Tecnologia FCT (SFRH/BSAB/757/2007), and by the UK EPSRC (EP/F037368/1). Gesbert was supported by the UK EPSRC (EP/E065708/1) and by the French ANR (project ANR-08-EMER-004 "CODEX"). All of the authors have received support from COST Action IC1201 "Behavioural Types for Reliable Large-Scale Software Systems".

References

[1] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *Proceedings of Onward!*, 2009.

[2] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer, 1997.

[3] D. Ancona, V. Bono, M. Bravetti, J. Campos, P.-M. Deniélou, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. Martins, V. Mascardi, F. Montesi, N. Ng, R. Neykova, L. Padovani, and V. T. Vasconcelos. BETTY WG3 – Languages: State of the Art Report. Report of the EU COST Action IC1201 – Behavioural Types for Reliable Large-Scale Software Systems. `www.behavioural-types.eu/publications/WG3-State-of-the-Art.pdf`, 2014.

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

[5] N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 227–244. ACM, 2008.

[6] G. Bernardi, 2014. Private communication.

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

[8] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 301–320. ACM, 2007.

[9] K. Bierhoff and J. Aldrich. PLURAL: checking protocol compliance under aliasing. In *Companion of the 30th International Conference on Software Engineering (ICSE)*, pages 971–972. ACM, 2008.

[10] K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*, volume 5653 of *Lecture Notes in Computer Science*, pages 195–219. Springer, 2009.

[11] E. Bonelli and A. Compagnoni. Multipoint Session Types for a Distributed Calculus. In *Proceedings of the 3rd International Symposium on Trustworthy Global Computing (TGC)*, volume 4912 of *Lecture Notes in Computer Science*, pages 240–256. Springer, 2007.

[12] V. Bono, C. Messa, and L. Padovani. Typing copyless message passing. In *Proceedings of ESOP*, number 6602 in LNCS, pages 57–76, 2011.

[13] L. Caires. Spatial-behavioral types for concurrency and resource control in distributed systems. *Theoretical Computer Science*, 402(2–3):120–141, 2008.

[14] L. Caires and H. T. Vieira. Conversation types. *Theoretical Computer Science*, 411(51–52):4399–4440, 2010.

[15] J. Campos. Linear and shared objects in concurrent programming. Master's thesis, University of Lisbon, 2010.

[16] J. Campos and V. T. Vasconcelos. Channels as objects in concurrent object-oriented programming. In *Proceedings of the 3rd International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES)*, Electronic Proceedings in Theoretical Computer Science (EPTCS), 2010. To appear.

[17] 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*, 410(2–3):142–167, 2009.

[18] M. Carbone, K. Honda, and N. Yoshida. Structured global programming for communication behaviour. In *Proceedings of the 16th European Symposium on Programming Languages and Systems (ESOP)*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2007.

[19] S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 45–57. ACM, 2002.

[20] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 48–64. ACM, 1998.

[21] J.-L. Colaço, M. Pantel, F. Dagnat, and P. Sallé. Safety analysis for non-uniform service availability in Actors. In *Proceedings of the IFIP TC6/WG6.1 3rd International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 139 of *IFIP Conference Proceedings*. Kluwer, 1999.

[22] J.-L. Colaço, M. Pantel, and P. Sallé. A set-constraint-based analysis of Actors. In *Proceedings of the IFIP TC6/WG6.1 International Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 107–122. Chapman & Hall, 1997.

[23] L. Cruz-Filipe, I. Lanese, F. Martins, A. Ravara, and V. T. Vasconcelos. The stream-based service-centered calculus: a foundation for service-oriented programming. *Formal Aspects of Computing*, 26:865–918, 2014.

[24] 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.

[25] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 59–69. ACM, 2001.

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

[27] M. Dezani-Ciancaglini, S. Drossopoulou, E. Giachino, and N. Yoshida. Bounded session types for object-oriented languages. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects (FMCO 2006)*, volume 4709 of *Lecture Notes in Computer Science*, pages 207–245. Springer, 2007.

[28] M. Dezani-Ciancaglini, S. Drossopoulou, D. Mostrous, and N. Yoshida. Objects and session types. *Information and Computation*, 207(5):595–641, 2009.

[29] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopolou. Session types for object-oriented languages. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.

[30] M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopolou. A distributed object-oriented language with session types. In *Proceedings of the International Symposium on Trustworthy Global Computing (TGC)*, volume 3705 of *Lecture Notes in Computer Science*, pages 299–318. Springer, 2005.

[31] 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*, pages 177–190. ACM, 2006.

[32] M. Fähndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 13–24. ACM, 2002.

[33] M. Fähndrich and R. DeLine. Typestates for objects. In *Proceedings of the 13th European Symposium on Programming Languages and Systems (ESOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004.

[34] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM, 2002.

[35] R. Garcia, E. Tanter, R. Wolff, and J. Aldrich. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems*, 2014.

[36] S. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-oriented programming. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 299–312. ACM, 2010.

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

[38] 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, 2003.

[39] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010.

[40] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293. ACM, 2002.

[41] J. Hogg. Islands: aliasing protection in object-oriented languages. In *Proceedings of the 6th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 271–285. ACM, 1991.

[42] 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)*, pages 273–284. ACM, 2008.

[43] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of the 7th European Symposium on Programming Languages and Systems (ESOP)*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[44] R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, and K. Honda. Type-safe eventful sessions in Java. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP)*, volume 6183 of *Lecture Notes in Computer Science*, pages 329–353. Springer, 2010.

[45] R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in Java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP)*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541. Springer, 2008.

[46] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P.-M. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro. Foundations of behavioural types. Report of the EU COST Action IC1201 – Behavioural Types for Reliable Large-Scale Software Systems. `www.behavioural-types.eu/publications/WG1-State-of-the-Art.pdf`, 2014.

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

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

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

[50] 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.

[51] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

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

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

[54] F. Militão, J. Aldrich, and L. Caires. Aliasing control with view-based typestate. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs (FTfJP)*. ACM, 2010.

[55] D. Mostrous and V. T. Vasconcelos. Session typing for a featherweight erlang. In *Proceedings of the 13th International Conference on Coordination Models and Languages (COORDINATION'11)*, volume 6721 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2011.

[56] E. Najm, A. Nimour, and J.-B. Stefani. Guaranteeing liveness in an object calculus through behavioral typing. In *Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*, pages 203–221. Kluwer, 1999.

[57] E. Najm, A. Nimour, and J.-B. Stefani. Infinite types for distributed object interfaces. In *Proceedings of the IFIP TC6 WG6.1 3rd International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 139 of *IFIP Conference Proceedings*. Kluwer, 1999.

[58] M. Neubauer and P. Thiemann. An implementation of session types. In *Proceedings of the 6th International Symposium on Practical Aspects of Declarative Languages (PADL)*, volume 3057 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2004.

[59] N. Ng, N. Yoshida, O. Pernet, R. Hu, and Y. Kryftis. Safe parallel programming with session java. In *Proceedings of the 13th International Conference on Coordination Models and Languages (COORDINATION'11)*, volume 6721 of *Lecture Notes in Computer Science*, pages 110–126. Springer, 2011.

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

[61] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction (CC)*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2003.

[62] J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, uniqueness and immutability. In *Objects, Components, Models and Patterns: 46th International Conference, TOOLS EUROPE*, volume 11 of *Lecture Notes in Business Information Processing*, pages 178–197. Springer, 2008.

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

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

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

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

[67] A. Ravara and V. T. Vasconcelos. Typing non-uniform concurrent objects. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR)*, volume 1877 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2000.

[68] J. G. Rossie, Jr. Logical observable entities. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 154–165. ACM, 1998.

[69] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.
[70] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *Proceedings of the 6th International Conference on Parallel Architectures and Langauges Europe (PARLE)*, volume 817 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 1994.
[71] J. A. Tov and R. Pucella. Practical affine types. In *Proceedings of POPL*, 2011.
[72] A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of software components using session types. *Fundamenta Informaticæ*, 73(4):583–598, 2006.
[73] V. T. Vasconcelos, S. J. Gay, and A. Ravara. Type checking a multithreaded functional language with session types. *Theoretical Computer Science*, 368(1–2):64–87, 2006.
[74] V. T. Vasconcelos, S. J. Gay, A. Ravara, N. Gesbert, and A. Z. Caldeira. Dynamic interfaces. Presented at the International Workshop on Foundations of Object-Oriented Languages (FOOL), 2009.
[75] R. Wolff, R. Garcia, E. Tanter, and J. Aldrich. Gradual typestate. In *Proceedings of ECOOP*, 2011.
[76] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

Appendix: Proofs of lemmas from Section 7.1

Lemma 7.1. Suppose $\Theta \vdash h : \Gamma$. Then *(a)* h is complete, *(b)* $\mathsf{chans}(\Gamma) \subseteq dom(\Theta) \setminus \mathsf{chans}(h)$ and *(c)* $\mathsf{objs}(\Gamma) \subseteq \mathsf{roots}(h)$.

Proof. By induction on the derivation of $\Theta \vdash h : \Gamma$. The only axiom is T-Hempty for which the properties are true. Then T-Hide does not change either h or $dom(\Gamma)$ so it preserves all three properties. The other case is T-Hadd. Let h' be the heap in the conclusion. Then $\mathsf{children}_{h'}(o)$ is the set of v_i which are object identifiers. Let K be the set of v_i which are channel endpoints. The typing derivation for the sequence of swaps in the right premise must include an occurrence of T-Ref for each object identifier, and of T-Chan for each channel endpoint, each followed by T-Swap and a number of occurrences of T-Seq. Looking at these rules, we can see that this implies:

(1) $\mathsf{children}_{h'}(o) \cup K \subseteq dom(\Gamma)$ and
(2) $dom(\Gamma') \subseteq (dom(\Gamma) \setminus (\mathsf{children}_{h'}(o) \cup K)) \cup \{o\}$. (Note that o cannot be one of the v_i because it is the current object in the judgement: the premise of T-Ref forbids it.)

From (1) and induction hypothesis (c) we get $\mathsf{children}_{h'}(o) \subseteq \mathsf{roots}(h)$. We have $\mathsf{roots}(h) \subseteq dom(h) \subseteq dom(h')$ and o is the only new object in h', so h' is complete.

If we project (2) onto just channel endpoints, we get $\mathsf{chans}(\Gamma') \subseteq \mathsf{chans}(\Gamma) \setminus K$. From the definition of h', $\mathsf{chans}(h')$ is equal to $\mathsf{chans}(h) \cup K$. Hence induction hypothesis (b) yields (b) again for h'.

If we project (2) onto just object identifiers, we get $\mathsf{objs}(\Gamma') \subseteq (\mathsf{objs}(\Gamma) \setminus \mathsf{children}_{h'}(o)) \cup \{o\}$. From induction hypothesis (a) and the fact that $o \notin dom(h)$ we get that o is a root in h'. Furthermore, all roots of h which are not children of o are also roots of h'. Thus induction hypothesis (c) allows us to conclude $\mathsf{objs}(\Gamma') \subseteq \mathsf{roots}(h')$. □

Lemma 7.2 (Rearrangement of typing derivations for expressions)**.** Suppose we have $\Gamma * r \rhd e : T \lhd \Gamma' * r'$. Then there exists a typing derivation for this judgement in which:

(1) T-Sub only occurs at the very end, just before T-Switch or T-SwitchLink as the last rule in the derivation for each of the branches, or just before T-Call as the last rule in the derivation for the parameter;
(2) T-SubEnv only occurs immediately before T-Sub in the first three cases and does not occur at all in the fourth, i.e. T-Call.

Proof. First note that T-Sub and T-SubEnv commute and that any consecutive sequence of occurrences of one of these rules can collapse into a single occurrence using transitivity. What remains to be shown is that these rules can be pushed down in all cases but those mentioned in the statement. We enumerate the cases below.

- T-Swap. T-Sub before the premise can be replaced with T-SubEnv after the conclusion as T has been transferred to the environment. If T-SubEnv was used before the premise, it means the initial derivation looks like:

$$\dfrac{\dfrac{\Gamma * r \rhd e : T \lhd \Gamma' * r'}{\Gamma * r \rhd e : T \lhd \Gamma'' * r'}\,(\text{T-SubEnv}) \qquad \Gamma''(r'.f) = T' \qquad \dots}{\Gamma * r \rhd f \leftrightarrow e : T' \lhd \Gamma''\{r'.f \mapsto T\} * r'}\,(\text{T-Swap})$$

  with $\Gamma' <: \Gamma''$. Let $T_0 = \Gamma'(r'.f)$; we have $T_0 <: T'$ since $T' = \Gamma''(r'.f)$. Because the type of $r'.f$ moves from the environment to the expression, when we push the subsumption step down, we have to use both T-SubEnv and T-Sub; we can transform the derivation into:

$$\dfrac{\dfrac{\dfrac{\Gamma * r \rhd e : T \lhd \Gamma' * r' \qquad \Gamma'(r'.f) = T_0 \qquad \dots}{\Gamma * r \rhd f \leftrightarrow e : T_0 \lhd \Gamma'\{r'.f \mapsto T\} * r'}\,(\text{T-Swap})}{\Gamma * r \rhd f \leftrightarrow e : T_0 \lhd \Gamma''\{r'.f \mapsto T\} * r'}\,(\text{T-SubEnv})}{\Gamma * r \rhd f \leftrightarrow e : T' \lhd \Gamma''\{r'.f \mapsto T\} * r'}\,(\text{T-Sub})$$

- T-Call. If T-SubEnv is used on the premise to increase the type of something else than $r'.f$ it can be moved to the conclusion. If the type of $r'.f$ is changed, first note that the only relevant part is the signature of m_j. Suppose the subsumption step changes it from $U_j\ m_j(U'_j) : S'_j$ to $T_j\ m_j(T'_j) : S_j$. For the parameter type we have $T'_j <: U'_j$ so we can use T-Sub on the premise to increase the type of e from T'_j to U'_j instead. For the session and result types, we have two cases:
  - if $U_j <: T_j$ and $S'_j <: S_j$ it can just be moved to a T-SubEnv step on the conclusion.
  - if $U_j = E$, $T_j = \mathsf{variant\text{-}tag}$ and $\langle l : S'_j \rangle_{l \in E} <: S_j$, then the original conclusion of the rule (with T-SubEnv on the premise) was:

$$\Gamma * r \rhd f.m_j(e) : \mathsf{link}\ f \lhd \Gamma'\{r'.f \mapsto S_j\} * r'$$

    and the new one with the subsumption step removed is:

$$\Gamma * r \rhd f.m_j(e) : E \lhd \Gamma'\{r'.f \mapsto S'_j\} * r'.$$

    So in that case the original judgement can be obtained back from this new conclusion using T-VarS followed by T-SubEnv.
- T-Seq. T-Sub on the first premise is irrelevant and T-SubEnv on the same premise can be removed using Lemma 3.15. Subsumption on the second premise straightforwardly commutes to the conclusion.
- T-Switch. Lemma 3.15 allows us to remove T-SubEnv on the first premise. Straightforwardly T-Sub can be removed as well as it just makes E' smaller.
- T-SwitchLink. T-Sub is irrelevant; removing T-SubEnv can only make E' and the initial typing environments for the branches smaller and we can use Lemma 3.15.
- T-VarF and T-VarS. T-Sub can increase E which becomes the indexing set of the variant in the conclusion. By definition of subtyping for variants it is possible to increase it afterwards using T-SubEnv. T-SubEnv straightforwardly commutes.

- T-Return. T-Sub straightforwardly commutes, as well as the part of T-SubEnv not concerning $r'.f$. Subsumption on $\Gamma'(r'.f)$ can be removed using Proposition 3.16.

□

Lemma 7.3 (Rearrangement of typing derivations for heaps)**.** Suppose $\Theta \vdash h : \Gamma$ holds. Let o be an arbitrary root of h. Then there exists a typing derivation for it such that:

(1) T-Sub is never used;
(2) T-SubEnv is used at most once, as the last rule leading to the right premise of the last occurrence of T-Hadd;
(3) every occurrence of T-Hide follows immediately the occurrence of T-Hadd concerning the same object identifier;
(4) the occurrence of T-Hadd concerning an identifier o' is always immediately preceded (on the left premise) by the occurrences of T-Hadd/T-Hide concerning the descendants of o';
(5) the first root added is o.

Proof. The first two points are a consequence of Lemma 7.2: the only expressions which appear in the typing derivation are sequences of swaps, not containing any switch or method call; furthermore their type is always Null, making T-Sub at the end irrelevant. What remains to be checked is then just that T-SubEnv at the end of the derivation for one sequence of swaps can be pushed down to the next occurrence of T-Hadd whenever there is one. This is just a matter of using Proposition 3.16 in the case of T-Hide and Lemma 3.15 in the case of T-Hadd.

Note that these points imply in particular that in all applications of T-Hadd but the last one, any element in $dom(\Gamma)$ which is not one of the v_i also occurs in Γ' with exactly the same type.

For the third point, first notice that the premise of T-Hide implies o is a root of h because of Lemma 7.1. This implies that the rule immediately above T-Hide either is a T-Hadd introducing o or does not concern o at all (in particular, o cannot be a v_i, otherwise it would not be a root in the conclusion). In the second case, T-Hide can be pushed upwards.

The fourth and fifth points are a consequence of the remark we made about the first two: if o' is not a descendant of o nor vice-versa, then the occurrences of T-Hadd and T-Hide concerning o and its descendants commute with those concerning o' and its descendants as they affect completely disjoint parts of the environment. In the case of the last occurrence of T-Hadd there may be a subsumption step but it is still possible to commute with it by pushing this subsumption step down again. □

Lemma 7.4 (Splitting of the heap)**.** Suppose $\Theta \vdash h : \Gamma, o : T$. Let $\Theta_1 = \Theta \setminus \mathsf{chans}(h \downarrow o)$ and let Θ_2 be Θ restricted to $\mathsf{chans}(h \downarrow o)$. Then we have: $\Theta_1 \vdash (h \uparrow o) : \Gamma$ and $\Theta_2 \vdash (h \downarrow o) : o : T$.

Proof. We know from Lemma 7.1 that o is a root in h. We consider the particular derivation given by Lemma 7.3 where o is the first root added to the heap. Now if we look at the conclusion of the last rule concerning o (T-Hadd or T-Hide depending whether T is a field or session type), we know that at this point the heap is $h \downarrow o$, and therefore the only object identifier in the environment is its only root: o. Furthermore, this part of the derivation is still true if we replace the initial Θ with Θ_2, with the only difference that then the final Γ contains no channels, and thus is of the form $o : T'$. We also know that the type of o is

not changed in the rest of the derivation except possibly by the subsumption step at the end; therefore T' is a subtype of T. If they are session types, using Proposition 3.18 we can change the last occurrence of T-Hide to use T instead of T' and get $\Theta_2 \vdash (h \downarrow o) : o : T$. Otherwise, we can add a subsumption step to the derivation for the sequence of swaps on the right of T-Hadd to get the same result.

For the rest of the derivation, we know that o is not used, therefore it can be removed from the initial environment without affecting the derivation except by the fact that it will not be in the final environment either. Furthermore, we know from Lemma 7.1 that the initial environment minus its only object identifier o is included in $\Theta \setminus \mathsf{chans}(h \downarrow o) = \Theta_1$. More precisely, the lemma gives us inclusion of domains, but because subsumption is not used in the first part of the derivation we also know that the types are the same. Thus we can replace the first part of the derivation by an instance of T-Hempty using Θ_1 and the second part is still valid (with all the descendants of o removed from the heap), yielding $\Theta_1 \vdash (h \uparrow o) : \Gamma$ at the bottom. □

Lemma 7.5 (Merging of heaps)**.** Suppose $\Theta \vdash h : \Gamma$ and $\Theta' \vdash h' : \Gamma'$ with $dom(h) \cap dom(h') = \emptyset$ and $dom(\Theta) \cap dom(\Theta') = \emptyset$. Then we have $\Theta + \Theta' \vdash h + h' : \Gamma + \Gamma'$.

Proof. Since Θ and Θ' are disjoint, the channels in Θ' cannot appear anywhere in the typing derivation for h. Thus, it is possible to add Θ' to every typing environment occurring in the derivation for h without altering its validity, yielding $\Theta + \Theta' \vdash h : \Gamma + \Theta'$. Looking now at the derivation for h', since the domains of the heaps are disjoint and $\mathsf{objs}(\Gamma) \subseteq \mathsf{roots}(h)$, none of the identifiers in Γ can appear anywhere in it. Thus we can add Γ to every typing environment and h to every heap occurring in the derivation for h', replacing the T-Hempty at the top with the conclusion of the other derivation, which yields the result we want. □

Lemma 7.6. Suppose $\Theta \vdash h : o : S$. Let φ be an injective function from $dom(h)$ to $\mathcal{O}$. Then we have $\Theta \vdash \varphi(h) : \varphi(o) : S$.

Proof. Straightforward. Changing the names does not affect the typing derivation in any way. □

Lemma 7.7 (Opening)**.** If $\Theta \vdash h : \Gamma$, if $\Gamma(r)$ is a branch session type S and if $h(r)$ is an object identifier o, then we know from Lemma 7.1 that h contains an entry for o. Let C be the class of this entry, then there exists a field typing F for C such that $\Theta \vdash h : \Gamma\{r \mapsto C[F]\}$ and $F \vdash C : S$.

Proof. We prove this by induction on the depth of r. The base case is $r = o$. Using Lemmas 7.4 and 7.5, we can restrict ourselves to the case where o is the only root of h. In that case we know that the last rule used in the typing derivation for $\Theta \vdash h : o : S$ must be T-Hide. The result we want is constituted precisely by the premises of that rule.

For the inductive case, r is of the form $o'.f.\vec{f}$. We consider the case where o' is the only root. The typing derivation then ends with T-Hadd and f gets populated in the sequence of swaps by some object identifier[2] o''. Let $r' = o''.\vec{f}$, and consider what $\Gamma(r')$ can be, knowing that in the conclusion r has a branch session type: the only way the type can be modified in the sequence of swaps is by subsumption. Indeed, T-VarS, the other possibility, introduces a variant type. Therefore $\Gamma(r') = S'$ with $S' <: S$. We can thus use the induction hypothesis to replace Γ with $\Gamma\{r \mapsto C[F]\}$ on the left premise, with $F \vdash C : S'$. Then just

[2]We know o'' is an object identifier and not a channel endpoint because, according to the hypotheses, either it is o itself or $\vec{f}$ is nonempty, implying o'' has fields.

use Proposition 3.18 to see that we also have $F \vdash C : S$ and see that the type yielded in the conclusion by this new premise is what we want. □

Lemma 7.8 (Closing)**.** If $\Theta \vdash h : \Gamma$ and $\Gamma(r) = C[F]$ and $F \vdash C : S$, then $\Theta \vdash h : \Gamma\{r \mapsto S\}$.

Proof. Again we prove this by induction on the depth of r and the base case is $r = o$. In that case the lemma is nothing more than T-Hide. The inductive case is very similar to the above: we look at the type of r' (defined as above) in the Γ on the left premise of the last T-Hadd, noticing that the type of r in the conclusion can only differ from it by subsumption, this time because we know from Lemma 3.28 that T-VarF is never used. Hence the original type is $C[F']$ with $F' <: F$. Proposition 3.16 gives us $F' \vdash C : S$ and thus we can use the induction hypothesis to change the type of r' in this premise, which propagates to the type of r in the conclusion. □

Lemma 7.9 (modification of the heap)**.** Suppose that we have $\Theta \vdash h : \Gamma$ and $\Gamma * r \rhd v' : T' \lhd \Gamma' * r$, and that $\Gamma'(r.f) = T$ where T is not a variant. Let $v = h(r).f$. The modified heap $h\{r.f \mapsto v'\}$ can be typed as follows:

(1) if v is an object identifier or a channel endpoint, then:

$$\Theta \vdash h\{r.f \mapsto v'\} : \Gamma'\{r.f \mapsto T'\}, v : T$$

(2) if v is not an object or channel and T is not a link type, then:

$$\Theta \vdash h\{r.f \mapsto v'\} : \Gamma'\{r.f \mapsto T'\}$$

(3) if $v = l_0$ and $T = \mathsf{link}\ f'$, then:
- $\Gamma'(r.f') = \langle l : S_l \rangle_{l \in E}$ for some E such that $l_0 \in E$ and some set of branch session types S_l. Note that this implies $f \neq f'$.
- $\Theta \vdash h\{r.f \mapsto v'\} : \Gamma'\{r.f \mapsto T'\}\{r.f' \mapsto S_{l_0}\}$

Proof. First of all, note that the hypothesis that $\Gamma'(r.f)$ is defined implies that $\Gamma'(r)$ is not a variant, hence T' is not $\mathsf{variant\text{-}tag}$. In other words, the judgement cannot be derived from T-VarF. Furthermore, the fact that T is not a variant either means that the judgement is not derived from an instance of T-VarS referring to field f. As this rule is the only possibility (beside subsumption) for a judgement typing a value to depend on, and modify, the type of a field, this implies that $\Gamma(r.f)$ is a subtype of T and also that the judgement would still hold with another type for f. In particular we have $\Gamma\{r.f \mapsto \mathsf{Null}\} * r \rhd v' : T' \lhd \Gamma'\{r.f \mapsto \mathsf{Null}\} * r$ **(a)**. We will in the following use this judgement (a) rather than the one in the hypothesis.

We prove the lemma by induction on the depth of r, but the inductive case is straightforward (just apply the induction hypothesis to the left premise of T-Hadd). In the base case, r is an object identifier o. We use Lemma 7.4 to consider a typing derivation for the sub-heap $h \downarrow o$. Let $\Gamma_o = o : T_o$ and Θ_o be the environments corresponding to that part of the heap. We look at the application of T-Hadd which ends the derivation for $\Theta_o \vdash h_o : \Gamma_o$. As T is not a variant, it is possible to consider that f is the last field to get populated in the swap sequence. We thus have something of the form:

$$\dfrac{\Theta_o \vdash (h \downarrow o) \setminus o : \Gamma_1 \qquad \dfrac{\dfrac{\cdots \qquad \dfrac{(1)\ \dfrac{\cdots}{\Gamma_2 * o \rhd v : T_v \lhd \Gamma_3 * o}}{\Gamma_2 * o \rhd f \leftrightarrow v : \mathsf{Null} \lhd \Gamma_3\{o.f \mapsto T_v\} * o}\ \text{(T-Swap)}}{\Gamma_1 * o \rhd \ldots; f \leftrightarrow v : \mathsf{Null} \lhd \Gamma_3\{o.f \mapsto T_v\} * o}\ \text{(T-Seq)}}{\Gamma_1 * o \rhd \ldots; f \leftrightarrow v : \mathsf{Null} \lhd \Gamma_o * o}\ \text{(T-SubEnv)}}{\Theta_o \vdash h \downarrow o : \Gamma_o}\ \text{(T-Hadd)}$$

with $T_v <: T$ and $\Gamma_3\{o.f \mapsto T_v\} <: \Gamma_o$. If we change the type of $o.f$, this last relation becomes $\Gamma_3 <: \Gamma_o\{o.f \mapsto \mathsf{Null}\}$ **(b)**.

What we want to do is to replace the judgement on the top right, which is an application of some rule (1), by a judgement typing v'. For this, we need the rest of the environment. We consider the judgement for the rest of the heap, $\Theta \setminus \Theta_o \vdash h \uparrow o : \Gamma \setminus o$. Since the domains are disjoint, we can apply Lemma 7.5 to this and the leftmost premise of T-Hadd, yielding $\Theta \vdash h \setminus o : \Gamma \setminus o + \Gamma_1$. If we replace our left premise with this, the initial environment we get on the top right is now $\Gamma_4 = \Gamma \setminus o + \Gamma_2$, as the additional part is unaffected by the sequence of swaps. This environment is almost $\Gamma\{o.f \mapsto \mathsf{Null}\}$, but not quite. We now have three cases depending on what rule (1) is, which correspond to the three cases of the lemma.

(1) If (1) is T-Ref or T-Chan, meaning v is an object identifier or a channel endpoint, then $\Gamma_2 = \Gamma_3, v : T_v$. Using (b), this yields $\Gamma_2 <: \Gamma_o\{o.f \mapsto \mathsf{Null}\}, v : T_v$. Adding $\Gamma \setminus o$ to both sides, we get $\Gamma_4 <: \Gamma\{o.f \mapsto \mathsf{Null}\}, v : T_v$. If we replace the initial environment in (a) with this one, we get the $v : T_v$ back in the final environment. We then use Lemma 3.15 to replace this initial environment with Γ_4, and T-SubEnv to change T_v into T in the final one: $\Gamma_4 * o \rhd v' : T' \lhd \Gamma'\{o.f \mapsto \mathsf{Null}\}, v : T * o$. Just see that it yields what we want at the bottom of the derivation.

(2) If (1) is T-Label or T-Null or T-Name, i.e. if v is a literal value of non-link, non-linear type, then Γ_2 is identical to Γ_3 and we have, using (b) and adding $\Gamma \setminus o$ to both sides, $\Gamma_4 <: \Gamma\{o.f \mapsto \mathsf{Null}\}$, so we can directly (with Lemma 3.15) use judgement (a).

(3) If (1) is T-VarS, the last possibility, then v is a label l_0 and T_v is $\mathsf{link}\ f'$ for some f'. As it has no strict supertype, we have $T = \mathsf{link}\ f'$ as well. We also have $\Gamma_3(o.f') = \langle l_0 : S\rangle$. From (b) we have that $\Gamma_o(o.f') = \Gamma(o.f')$ is a supertype of this variant type, thus also a variant. This implies that (a) cannot come from a T-VarS concerning f'; therefore, $\Gamma'(o.f')$ is a supertype of $\Gamma(o.f')$ and hence, by transitivity, of $\langle l_0 : S\rangle$, which gives us the first item of the conclusion, with $S <: S_{l_0}$. We now just have to notice that $\Gamma_2 = \Gamma_3\{o.f' \mapsto S\}$ and that (a) is independent of the type of f' just like it is of the type of f, and we can conclude similarly to the two previous cases.

□

Lemma 7.10 (Substitution)**.** If $\mathsf{this} : C[F], x : T' * \mathsf{this} \rhd e : T \lhd \mathsf{this} : C[F'] * \mathsf{this}$, and if $\Gamma(r) = C[F]$, then:

(1) if T' is a base type (i.e. neither an object type nor a link) and v is a literal value of that type, or if v is an access point name declared with type $\langle\Sigma\rangle$ and $[\![\langle\Sigma\rangle]\!] <: T'$, we have:

$$\Gamma * r \rhd e\{^v/_x\} : T \lhd \Gamma\{r \mapsto C[F']\} * r.$$

(2) if T' is an object type and v is an object identifier or a channel endpoint, we have:

$$\Gamma, v : T' * r \rhd e\{^v/_x\} : T \lhd \Gamma\{r \mapsto C[F']\} * r.$$

Proof. In order to do an induction, we add the following case where x is still present in the final environment : if we have $\mathsf{this} : C[F], x : T' * \mathsf{this} \rhd e : T \lhd \mathsf{this} : C[F'], x : T'' * \mathsf{this}$, and if T' is an object type and v is an object identifier, then we have $\Gamma, v : T' * r \rhd e\{^v/_x\} : T \lhd \Gamma\{r \mapsto C[F']\}, v : T'' * r$.

We prove this by induction on the derivation of $\mathsf{this} : C[F], x : T' * \mathsf{this} \rhd e : T \lhd \mathsf{this} : C[F'], V * \mathsf{this}$ (where V is either empty or $v : T''$ depending on the case). For most

toplevel rules, the result is immediate. The only ones for which it is not are T-Var and T-LinVar. For T-Var the result is obtained using either T-Null if T' is Null or T-Label and T-Sub if it is an enumerated type. In the case of an extension adding new base types, we assume there is a similar rule to type the corresponding literal values. For T-LinVar, if v is an access point name the result is obtained using T-Name and T-Sub. Otherwise, v is an object identifier and the result is obtained using T-Ref, noticing that because $\Gamma(r)$ is defined and v is not in Γ, the path r does not start with v and the premise is satisfied. □

Lemma 7.11 (Typability of Subterms)**.** If $\mathcal{D}$ is a derivation of $\Gamma * r \rhd \mathcal{E}(e) : T \lhd \Gamma' * r'$ then there exist Γ_1, r_1 and U such that $\mathcal{D}$ has a subderivation $\mathcal{D}'$ concluding $\Gamma * r \rhd e : U \lhd \Gamma_1 * r_1$ and the position of $\mathcal{D}'$ in $\mathcal{D}$ corresponds to the position of the hole in $\mathcal{E}$.

Proof. A straightforward induction on the structure of $\mathcal{E}$; the expression e is always at the extreme left of the typing derivation for $\mathcal{E}(e)$. □

Lemma 7.12 (Replacement)**.** If

(1) $\mathcal{D}$ is a derivation of $\Gamma * r \rhd \mathcal{E}(e) : T \lhd \Gamma' * r'$
(2) $\mathcal{D}'$ is a subderivation of $\mathcal{D}$ concluding $\Gamma * r \rhd e : U \lhd \Gamma_1 * r_1$
(3) the position of $\mathcal{D}'$ in $\mathcal{D}$ corresponds to the position of the hole in $\mathcal{E}$
(4) $\Gamma'' * r'' \rhd e' : U \lhd \Gamma_1 * r_1$

then $\Gamma'' * r'' \rhd \mathcal{E}(e') : T \lhd \Gamma' * r'$.

Proof. Replace $\mathcal{D}'$ in $\mathcal{D}$ by the derivation of $\Gamma'' * r'' \rhd e' : U \lhd \Gamma_1 * r_1$. □