University of Glasgow | School of Computing Science

Session types in the wild

A. Laura Voinea

First Supervisor: Dr. Simon Gay
Second Supervisor: Dr. Wim Wanderbaude

2nd Year PhD Report

June 5, 2017

Contents

1 Introduction

Nowadays, distributed systems are ubiquitous and communication is an important feature and reason for its success. Communication-centred programming has proven to be one of the most successful attempts to replace shared memory for building concurrent, distributed systems. Communication is easier to reason about and scales well as opposed to shared memory, making it a more suitable approach for systems where scalability is a must, as in the case of multi-core programming, service-oriented applications or cloud computing[1].

Communication is usually standardised via protocols that specify the possible interactions between the communicating parties in a specific order. Mainstream programming languages fail to adequately support the development of communication-centred software. Thus, implementations of communication behaviours are often based on informal protocol specification, and so informal verification. As a result they are prone to errors such as communication mismatch, when the message sent by one party is not expected by the other party, or deadlock, when parties are waiting for a message from each other causing the system to block[1].

To allow formal protocol specification within the programming language session types have been devised. Session types describe communication by specifying the type and direction of messages exchanged between parties[6]. Programmers can express a protocol specification as a session type, which can guarantee, within the scope where the session type applies, that communications will always match and the system will never deadlock. The main goal of the ABCD project[1] is to improve the practice of software development for concurrent and distributed systems through the use of session types. This is to be accomplished through built-in language support for protocol codification in existing languages such as Java or Python, in new languages such as Links[2], inter-language interoperability via session types, and through adapting interactive development environments and modelling techniques to support session types. Logical and automata foundations of session types will be further developed to express a wider class of behaviour and, as need arises, to support the former. Empirical studies to assess methodologies and tools are to be carried out, with results being used to improve language and tool design and implementation.

1.1 Research Questions

The aim of this PhD is to explore the extend to which session types are useful in practice. To that effect some questions will be considered:

- What are the theoretical and practical challenges to expressing ‘real’ life protocols with session types?
- How does the theory need to be extended to help overcome these?
- How do existing tools need to be improved?
- Is there a need for additional tools?

1.2 Thesis Statement

Hypothesis: Session types are a useful addition to the syntax and semantics of modern languages. Programming with session types and the constraints that this comes with i.e. linearity can be understood and used by real world programmers. Moreover session types can help programmers understand the problem they are tying to solve with more ease and structure their code better. Session typed languages provide useful additional safeguards and diagnostic information that lead to a system with the expected behaviour with less effort.

1.3 Outline

A short literature review of the most relevant work can be found in 2. I have done implementation work on two session type tools developed at the University of Glasgow as part of the ABCD project[1], namely Mungo, and StMungo[21], as well as looked at some possible usecases. This will be discussed in more detail in section 3. Alongside that, I have looked into defining a suitable methodology and plan to carry out a user study to explore how "real-world" programmers will interact with session types through Mungo, StMungo and Scribble. More on this can be found in 3.3.2.

Further work along with a plan for the time remaining is discussed in section 4.

2 Literature review

2.1 Session types, a short history

Session types have been introduced by Honda[18], and further extended by Takeuchi et al.[30], Honda et al.[19], and others.

Session types are a way of modelling communication between different entities and enforcing the order of communication, as well as the types of the messages that may be sent.

Session types have and continue to be the subject of a wide body of work aimed at extending their theoretical foundations as well as developing new tools and techniques. For example, session types have been augmented with subtyping polymorphism to enable protocols to describe richer behaviours[14]. They have been extended to ensure the progress property and deadlock freedom [11]. Session types have also been extended to multiparty session types to support communication instances with more than two participants while still guaranteeing the absence of deadlock[17]. In other work [8], global session types have been used to detect choreographies that can be realized in the context of web services.

Session types have been successfully applied in functional programming languages [32], object-oriented languages like Java [20, 15], low-level programming languages like C in [26], dynamically-typed languages like Python[25] or Erlang [24], or in the operating systems context with the Sing# language [12].

2.2 Typestate

Typestate is a refinement of the concept of a type in programming. A typestate defines the valid sequence of operations that can be performed on an instance of a certain type by associating state information with variables of that type. This state information can then be used at compile-time to determine the operations that can be invoked with valid results on an instance of a type. A typical example is that of a file, which can be read or written only after being open, and never after being closed. In most cases such constraints on the sequences of method calls are at best informally documented through method descriptions or comments. However in this form, the information cannot be used by the compiler to detect any violations.

Since typestate has been introduced in[28], there have been many efforts to add it to practical programming languages. In [10], the concept of typestate, originally introduced for imperative programs, was extended for the object-oriented paradigm, resulting in the Fugue system. Sing# [12] is an extension of C# which incorporates typestate-like contracts, to specify protocols. Sing# has been successfully used to implement Singularity, a message-passing based operating system used for research at Microsoft. A core calculus for *Sing#* has been introduced in Bono et al. [7] proved type safe.

Plural[6] is another noteworthy example. Based on Java, it has been used to study access control systems[5] and transactional memory[4], and to evaluate the effectiveness of typestate in Java APIs[6]. However, Plural's use of annotations to define typestates is not ideal, while annotations are convenient for attaching extra information that can be consumed by tools, it makes expressing more complex concepts difficult and can be a burden on the programmer. Plural provides static analysis, and does not alter the runtime representation of classes in any way. Consequently, dynamic checking might still be needed if the system will interact with components that are not trusted. Stemmed from the research on Plural, is Plaid, a general purpose programming language that goes one step further and incorporates typestates as native feature of the programming language, and encourages developers to design objects around their protocol [3, 29]. Instead of class definitions, a program consists of state definitions containing methods that cause transitions to other states. Transitions are specified in a similar way to Plural's pre- and post-conditions. Similarly to classes, states can be structured into an inheritance hierarchy.

Typestates are are well-suited to represent behavioral type refinements, such as session types. In [15] Gay et al. proposed the integration of session types and object-oriented programming by using typestate. A session type is considered as being a special case of typestate in [15], constraining the use of send and receive methods, and introduce a session-inspired notation for specifying method sequences. A channel with a session type can be viewed as an object with non-uniform method availability: send and receive are only available when the session type says that they should be available. Mungo [21] follows this approach of [15] to formalise a typestate inference system for a core object-oriented language, Java, and prove its correctness. Mungo is implemented as a front-end typechecking tool for a subset of Java. Inference removes the need for typestate annotations on parameters and return types as in the system by Gay et al [15]. The possible sequences of method calls are explicitly defined, rather than being consequences of pre- and post-conditions as in the Plural system. A typestate in Mungo can depend on the return value of a method call. The basic difference between Mungo and other typestate approaches is the Java-like syntax for describing state machine protocols, instead of associating pre- and post-conditions with methods. The motivation for developing such a typestate syntax is the ability to globally describe object behaviour and subsequently the general integration of session behaviour in objects.

One of the challenges in typestate analysis is the problem of aliasing. Aliasing occurs when an object has more than one reference or pointer that points to it. For a correct analysis, a state change to a given object must be reflected in all references that point to that object. Otherwise method calls via aliases can cause inconsistent state changes. However tracking all such references is a difficult problem. Fugue follows an "adoption and focus" approach, while Plural and Plaid follow permission-based approaches. To avoid the possibility of conflicting state changes through different aliases, Mungo goes for linearity and does not allow aliasing of objects that have a typestate protocol.

2.3 Language usability and evaluation

Programming languages should help programmers develop software effectively. With advances in technology, they should be concerned more with programmer efficiency rather than programming language efficiency. The most successful languages are not necessarily the most efficient ones, but the ones that are the easiest to use to satisfactorily achieve one's goals. Languages such as Java or Python are widely used, despite known flaws, while languages like Agda, or Idris are rarely even heard of outside academic circles. Language designers seem to overlook the people that will use their language. New language features are advertised as being an improvement over existing work, and, so, desirable, however, little evidence is usually brought forth to support such claims. In recent years, this has become a more visible issue, and there are now more efforts to improve it.

One such effort is the Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU) at SPLASH[1]. The aim of the workshop is to discuss methods, metrics and techniques for evaluating the usability of languages and language tools. The workshop highlights some areas of interest as: empirical studies of programming languages, methodologies and philosophies behind language and tool evaluation, software design metrics and their relations to the underlying language, user studies of language features and software engineering tools, visual techniques for understanding programming languages, critical comparisons of programming paradigms, tools to support evaluating programming languages, psychology of programming, domain specific language usability and evaluation. As such it seemed to be the perfect place to start when trying to design an empirical study for a new construct.

3 Work Undertaken

3.1 Mungo

Mungo[21] is a Java front-end tool, developed at the University of Glasgow, used to statically typecheck Java programs augmented with typestate Mungo implements two main components. First, a Java-like syntax to define typestate specifications for classes, and second, a typechecker that checks whether objects that have typestate specifications are used correctly.

Typestates are specified in separate files and are associated with Java classes by means of a Java annotation. This allows programs that have been checked by Mungo to be compiled and run using standard Java tools. If a class has a typestate specification, the Mungo typechecker analyses each object of that class in the program and extracts the method call behaviour (sequences of method

[1]http://2016.splashcon.org/track/plateau2016

calls) through the object's life. Finally, it checks the extracted information against the sequences of method calls allowed by the typestate specification.

Mungo supports typechecking for a subset of Java. The programmer can define both classes that follow a typestate specification and classes that do not. The typechecking procedure follows objects (instances of the former classes) through argument passing and return values. Moreover, the typechecking procedure for the fields of a class follows the typestate specification of the class to infer a typestate usage for the fields. For this reason fields that follow a typestate specification are only allowed to be defined in a class that also follows a typestate specification.

Mungo uses the JastAdd framework[2][16]. The JastAdd framework provides a Java parser which was used for the implementation of the Mungo typechecker. The JastAdd suite was also used to implement a parser for the Java-like typestate specification language.

The following work has been undertaken on the Mungo tool:

- The tool has been moved from the Java 1.4 compiler to the Java 1.8 compiler and adapted to work with the new framework.(joint work with Dr. Dimitrios Kouzapas)
- Moving the tool to a new compiler framework was a good opportunity for some Refactoring(such as getting rid of dead code or method extraction).
- the tool has been extended to support Java enumerations.(joint work with Dr. Dimitrios Kouzapas)
- Syntactic support for annotations
- Syntactic support for generic types
- Special annotation for typestate

Areas of ongoing work:

- Update repositories.
- Typecheck exceptions. Exceptions are supported syntactically but are type-checked under the (unsound) assumption that no exceptions are thrown; a `try {...} catch(Exception e) {...}` statement is typechecked by typechecking the try body and if an exception is thrown a typestate violation may result.
- Typecheck generic types. Generics are currently supported syntactically, but not typechecked.
- Extended support for more straightforward Java features such as synchronised statements, inner and anonymous classes, or static initialisers.

Areas of future work:

- typecheck collections
- context-free typestates
- aliasing

[2]http://jastadd.org/web/extendj/

3.2 StMungo

StMungo (Scribble to Mungo) [21] is a Java-based tool, developed at the University of Glasgow, that translates a Scribble[27, 34] local protocol into a Mungo specification and skeleton socket-based implementation code. The resulting code is typechecked using Mungo. Scribble is a protocol description language that can describe how two or more participating entities interact should interact with each other.

After the Scribble protocol is translated to a Mungo specification, Mungo **??** is used to generate a Java implementation for the protocol. This tool allows an easy transition from a Scribble global protocol definition to working Java implementation. We start by specifying distributed multiparty protocol in Scribble. We can then use the Scribble toolchain to validate and project the global protocol into a local one describing the interactions from the point of view of a specific participant. For every Scribble local protocol, StMungo will produce .mungo files containing: a typestate specification describing the local protocol as a sequence of method calls, an API for the participant implementing the typestate methods and a main class skeleton calling the methods in the typestate.

To improve this tool various extensions have been implemented:

- extended the tool to translate messages with no payload i.e.
  ```
  message_operator ()
  ```
- extended the tool to translate messages with multiple payload i.e.
  ```
  message_operator ( payload_type1 , ... , payload_typen )
  ```
- extended the tool to translate messages without a message signature i.e.
  ```
  ( payload_type1 , ... , payload_typen )
  ```
- extended the tool to translate messages with annotated payloads i.e.
  ```
  message_operator ( annotation:payload_type)
  ```
- various small improvements to allow most translations to run without having to be edited by a human
- various improvements allowing the tool to crash gracefully
- adapted the tool to work with multiple versions of scribble specification
- improved the tool by implementing support for special cases of recursions nested in choice structures A simple example of a problematic scribble specification is:
  ```
  global protocol Example(role S, role C) {
  choice at C{
          rcpt(String) from C to S;
      } or {
          msg(String) from C to S;
          rec loop {
                  subject(String) from C to S;
                  continue loop;
              }
          }
      }
  }
  ```

- improved the tool by implementing support for special cases of nested choice inside a recursion

  A simple example of a problematic scribble specification is:

  ```
  global protocol ProtocolName(role S, role C) {
    command(String) from C to S;
    rec overall {
    choice at S
        {
        ok(String) from S to C;
                        choice at S {end(String) from S to C;}
                        or {
                        sum(String) from S to C;
                        }
                        message(String) from S to C;
        } or {    error(String) from S to C;      }
        continue overall;
    }
    }
  ```

- refactoring(such as method extraction or getting rid of dead code) to keep everything simple
- regression testing to find any new bugs introduced
- collaborated with fellow PhD student Florian Weber on a plug-in for mapping between concrete and abstract messages from Scribble to the 'real-world' representation. Author's contributions being: debugging, refactoring, integrating with StMungo, and partly writing a paper(rejected).

Areas of further work:

- extending the tool to translate to support inlined protocols and sub-protocols(ongoing)
- extension to translate more complex constructs such as interruptible or parallel
- keeping it up to date with changes in Scribble and Mungo (ongoing)

3.3 Session type evaluation

3.3.1 Usecases

To better understand the expressive power of current session type technology together with any limitations that may need to be addressed, the current use case repository[3] was surveyed as a first step. As a second step, new real-world examples were sought. From the various protocols looked after, representations were attempted for two, Paxos and the File Transfer Protocol(FTP).

One protocol chosen was the File Transfer Protocol described by Request for Comments(RFC): 959[13]. FTP is a standard network protocol for transferring files between a client and server on a network. FTP is an unusual protocol in that it utilizes two ports, a data port and a command(control) port. FTP may run in active or passive mode, which determines how the data connection is established. In both cases, the client creates a TCP control connection from a random, usually an unprivileged, port number to the FTP server command port 21.

[3]https://github.com/epsrc-abcd/session-types-use-cases

Some representations of the algorithm have been attempted using Scribble, combined with StMungo and Mungo to give a working Java code. However in trying to represent it some shortcomings of StMungo became apparent. Hence, work on an FTP representation has been paused to improve StMungo and Mungo, to allow a better representation. Work on this usecase is planned to be restarted in the autumn.

Another protocol chosen as a usecase was Paxos. Paxos is a protocol for solving consensus in a network of unreliable processes. It ensures that a single value among the proposed values can be chosen. It assumes an asynchronous, non-Byzantine model.[22]

Two major advantages of Paxos are that it is provably correct in asynchronous networks that eventually become synchronous and it does not block if a majority of participants are available. Furthermore it has provably minimal message delays in the best case. Despite it's reputation of being difficult to understand & implement it is widely, a couple of examples would be Google in Chubby[9], Yahoo use something based on it in ZooKeeper. The protocol comes with three roles and a two-phase approach. A proposer responsible for initiating the protocol, that handles client requests and proposes values to be chosen. An acceptor that responds to messages from proposers by either rejecting them or agreeing in principle and making a promise about the proposals it will accept in the future. An a listener or learner, who wants to know which value was chosen. Each Paxos server can act as any or all 3 roles.[23] Some representations of the algorithm have been attempted using Scribble, combined with StMungo and Mungo to give a working Java code. However in trying to represent it some shortcomings of the toolset became apparent:

- representing broadcasting
- representing quorum/a majority
- representing express the dynamic aspects such as processes failing, restarting
- express multiple instances of the protocol

A successful implementation of Paxos has been achieved using the session type system for unreliable broadcast communication devised by Gutkovas, Kouzapas and Gay.

3.3.2 User study

New programming language constructs are more often than not introduced without first exploring how well suited their are for their purpose or how they would be used in the real world. While proving they solve the problem is a good thing, checking how well they solve it would be nice.

Session types have been developed for some time now with industry input, and a closer look at what exactly their effect is on software development is in order. Otherwise, we run the risk of developing something that may not be quite suitable. An example of this can be seen in gradual typing, another very active area of research, which is now having its practicality called into question [31]. By identifying which designs and implementations help or hinder programmers, we can improve them to help developers use session type effectively.

As a first step in testing session type designs and implementations through empirical studies, I surveyed existing literature to identify beliefs held by the session type community about how session types affect software development. Some explicit hypotheses were formulated as a result.[33]

Areas of further work:

- develop study methodology(ongoing)
- develop necessary tools for carrying out studies(ongoing)
- carry out studies for Scribble and Mungo

3.4 Other Activities

As part of the first year of my PhD, various additional activities have been undertaken, such as training courses , ABCD group meetings of various sizes, seminars and talks(e.g. The Scottish programming language seminar series, FATA seminars) which improved my knowledge of the field and gave me some insight of the exciting research ongoinge. Some notable events attended so far were the BETTY (Behavioural Types for Reliable Large-Scale Software Systems)[4], Wadlerfest[5], BETTY summer school in 2017 in Limassol, Cyprus, PLATEAU 2016[6] or POPL 2017[7].

4 Further Work

Some of further work planned has been highlighted throughout this report. The activities planed for the rest of this PHD are split into tasks and represented together with their estimated duration in figure 4.

[4]http://www.behavioural-types.eu/meetings/wg-mc-meetings-17th-18th-march-2016-in-malta
[5]http://events.inf.ed.ac.uk/wf2016/
[6]http://2016.splashcon.org/track/plateau2016
[7]http://conf.researchr.org/home/POPL-2017

Work plan

Task	Status	2017							2018												
		June	July	Aug	Sep	Oct	Nov	Dec	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sept	Oct	Nov	Dec
Literature survey																				
Literature survey	In progress	■		■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
Mungo Work																				
Generics	In progress	■		■	■	■	■	■	■	■										
Collections	Not started					■	■	■	■	■	■	■	■	■	■	■				
Exceptions	In progress						■	■	■	■	■	■	■	■	■					
Aliasing	Not started							■	■	■	■	■	■	■	■	■	■	■	■	■
Context-free typestates	Not started	■		■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
StMungo Work																				
Keep up to date with Scribble and Mungo	In progress	■		■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
Support more Scribble constructs	Not started	■		■	■	■	■	■												
Integrate with Mungo	Not started								■	■										
Language usability and evaluation																				
Develop necessary methodology	In progress	■		■	■	■	■	■	■	■	■	■	■	■	■	■	■	■		
Develop necessary tools	In progress			■	■	■	■	■												
Evaluate Scribble	Not started								■	■	■	■	■							
Evaluate Mungo	Not started																■	■	■	■
Evaluate through usecases	In progress			■	■	■	■	■	■											
Dissertation write-up																				
Literature survey Chapter	Not started																			■
Mungo Work Chapter	Not started																			
StMungo Work Chapter	Not started																			
Usability and evaluation Chapter	Not started																			
Conclusion Chaper	Not started																			
Additional chapters(optional)	Not started																			

References

[1] A basis for concurrency and distribution. http://groups.inf.ed.ac.uk/abcd/.

[2] Links: Linking theory to practice for the web. http://groups.inf.ed.ac.uk/links/.

[3] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *OOPSLA '09*, pages 1015–1022. ACM Press, 2009.

[4] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In *OOPSLA '08*, pages 227–244. ACM Press, 2008.

[5] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA '07*, pages 301–320. ACM Press, 2007.

[6] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions. In *ECOOP '09*, volume 5653 of *Springer LNCS*, pages 195–219, 2009.

[7] Viviana Bono, Chiara Messa, and Luca Padovani. Typing copyless message passing. In *ESOP '11*, volume 6602 of *Springer LNCS*, pages 57–76, 2011.

[8] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*

[9] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[10] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP '04*, volume 3086 of *Springer LNCS*, pages 465–490, 2004.

[11] Mariangiola Dezani-Ciancaglini, Ugo de' Liguoro, and Nobuko Yoshida. On Progress for Structured Communications. In Gilles Barthe and Cédric Fournet, editors, *TGC'07*, volume 4912 of *LNCS*, pages 257–275. Springer, 2008.

[12] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*, pages 177–190. ACM Press, 2006.

[13] File transfer protocol, RFC 959. `https://tools.ietf.org/html/rfc959`.

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

[15] Simon J. Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL '10*, pages 299–312. ACM Press, 2010.

[16] Görel Hedin. An introductory tutorial on JastAdd attribute grammars. In *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Springer LNCS*, pages 166–200, 2011.

[17] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL '08*, pages 273–284. ACM Press, 2008.

[18] Kohei Honda. Types for dyadic interaction. In *CONCUR '93*, volume 715 of *Springer LNCS*, pages 509–523, 1993.

[19] Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP '98*, volume 1381 of *Springer LNCS*, pages 122–138, 1998.

[20] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *ECOOP '08*, volume 5142 of *Springer LNCS*, pages 516–541, 2008.

[21] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo. In *PPDP '16*, pages 146–159. ACM Press, 2016.

[22] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[23] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[24] Dimitris Mostrous and VascoT. Vasconcelos. Session typing for a featherweight erlang. In *Coordination Models and Languages*, volume 6721 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011.

[25] Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. SPY: local verification of global protocols. In *RV '13*, volume 8174 of *Springer LNCS*, pages 358–363, 2013.

[26] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty session C: safe parallel programming with message optimisation. In *TOOLS '12*, pages 202–218, 2012.

[27] Scribble project homepage. `www.scribble.org`.

[28] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[29] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in Plaid. In *OOPSLA '11*, pages 713–732. ACM Press, 2011.

[30] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. *PARLE, Springer LNCS*, 817:398–413, 1994.

[31] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16. ACM, 2016.

[32] Vasco T. Vasconcelos, António Ravara, and Simon J. Gay. Session types for functional multithreading. In *CONCUR '04*, volume 3170 of *Springer LNCS*, pages 497–511. Springer, 2004.

[33] A. Laura Voinea and Simon J. Gay. Benefits of session types for software development. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU 2016. ACM.

[34] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble protocol language. In *TGC '13*, volume 8358 of *Springer LNCS*, pages 22–41, 2013.