Benefits of Session Types for Software Development

A. Laura Voinea
School of Computing Science
University of Glasgow, UK
a.voinea.1@research.gla.ac.uk

Simon J. Gay
School of Computing Science
University of Glasgow, UK
Simon.Gay@glasgow.ac.uk

Abstract

Session types are a formalism used to specify and check the correctness of communication based systems. Within their scope, they can guarantee the absence of communication errors such as deadlock, sending an unexpected message or failing to handle an incoming message. Introduced over two decades ago, they have developed into a significant theme in programming languages. In this paper we examine the beliefs that drive research into this area and make it popular. We look at the claims and motivation behind session types throughout the literature. We identify the hypotheses upon which session types have been designed and implemented, and attempt to clarify and formulate them in a more suitable manner for testing.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.2 [*Software Engineering*]: Design Tools and Techniques; D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Session types, hypotheses, empirical studies

1. Introduction

Session types are specifications of communication protocols, that describe the allowed sequences of messages together with the types of individual messages between participating entities. They guarantee properties such as privacy, communication safety and session fidelity. A well-typed program cannot send or receive a message of the wrong type.

In this paper we aim to identify beliefs held by the session type community about how session types affect software development. We then formulate some explicit hypotheses. This is done as a first step to test session type designs and implementations through empirical studies. One of the aims of the ABCD project [1], which this work is part of, is to empirically evaluate session type designs and implementations. But first, let us take a closer look at what session types are.

Session types model structured communication between two (binary) or more (multiparty) entities. They are defined as a sequence of input and output operations, specifying the types of messages to be exchanged. Multiparty session types extend the original binary framework to model the interactions of more than two participants. The interaction between all entities is captured as a global type. This global type can then be projected into local types, that describe the interaction from the point of view of each participant. The local type can be used to check that the implementation conforms, either statically taking advantage of the compiler or dynamically using runtime monitoring.

The now classic multiparty example is the Two Buyer protocol [5], in which Buyer1 and Buyer2 want to purchase a book from a book seller (Seller), negotiating the cost between them. Buyer1 selects an item and asks the Seller for the price, which sends the price back. Buyer1 then quotes a contribution to Buyer2. If Buyer2 agrees on the purchase, the other two parties are informed. Otherwise, the protocol terminates. This example illustrates the basic constructs of session types, such as input, output, internal choice (*select* from one of the possible options on hand) and external choice (*branch* - the offering of a set of alternatives).

```
1  global protocol TwoBuyer(role Buyer1, role Buyer2,
       role Seller) {
2    price_request(String) from Buyer1 to Seller;
3    choice at Seller {
4      price_response(int) from Seller to Buyer1;
5      quote(int) from Buyer1 to Buyer2;
6      choice at Buyer2 {
7       agree(String) from Buyer2 to Buyer1, Seller;
8      } or {
9       quit(String) from Buyer2 to Buyer1, Seller;
10     }
11   } or {
12     outOfStock() from Seller to Buyer1, Buyer2;
13   }
14 }
```

Example 1. Two Buyer protocol

This example is written in Scribble, a protocol description language that describes how two or more participating entities should interact with each other [19].

One popular approach to applying session types to practice is extending existing languages to support static ses-

PLATEAU '16, November 01 2016, Amsterdam, Netherlands
© 2016 ACM. ISBN 978-1-4503-4638-2/16/11$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/3001878.3001883

sion typing. Some examples would be Session J [9], or Mungo [11], both extending Java. New languages have also been developed based on session types, such as SILL [18]. A majority of research has been done on static session typing, with the contribution often being to develop a type system that types more programs, or to develop a type system that guarantees stronger properties of programs. Examples of extending these to enable typing of more programs include introducing subtyping [4], and recent work on context-free session types [22]. Examples of guaranteeing stronger properties include type systems that guarantee progress or deadlock-freedom; multiparty session types, capturing dependencies between sessions. There is an underlying assumption that this is a good thing to do, based on the belief that static typechecking for data types is a good thing. However, this has at most anecdotal evidence, based on the idea that a good static type system enforced by the compiler will catch most bugs before the code can even be executed.

Another approach is dynamic monitoring of communication [15], based on applying session types more directly to existing languages. While more flexible, dynamic verification loses the benefits of static type checking such as compile-time error detection and IDE support.

The newest approach, hybrid session verification [8], combines features of the previous two. Behaviour, i.e. sending messages in the right order, is checked statically using the native type system, while linearity, i.e. using the channels correctly, is checked at runtime. Static typing for session types relies on an unrealistic assumption that the whole world is statically typed. So, it is natural to statically check a component, and add dynamic monitoring to ensure that it does not receive any incorrect inputs.

Since their appearance [6, 20, 7], session types have become a hot topic in programming languages, subject to a wide body of research into theoretical foundations, tools and techniques. They have proved popular, finding their way into various programming languages and tools. As of late there has been an emphasis on implementations, be that in existing languages: C [17], Erlang [3], Go [16], Java [11, 9], to name just a few, or in new languages: Scribble, Links [13], SILL. The session type community certainly believes it to be worthwhile to add session types to as many languages as possible.

2. Claims

Are session type justified at all? Are session type tools and languages useful for developers? If so, which implementations are the most useful? Is the overhead added by session types to languages worth it?

Looking through the literature, most claims regarding session types fall into one of the following themes, those of software safety, software efficiency or usability (as in programmer efficiency, learnability or errors).

In terms of software safety, we find out that session types can eliminate certain communication errors, such as communication mismatch or deadlock. There is an underlying assumption that session types are good because they support typechecking of communication, which should in turn improve software safety. Using session types to model a protocol "enables verification to ensure that the protocol can be implemented without resulting in unintended consequences, such as deadlocks" [19]. Equivalent statements about verification can be found in most papers under one form or another.

In terms of usability, there are various claims regarding the efficiency of the development process, the improvement in the number and type of errors the programmers will or will not make, and ease of understanding and working with the new construct. It is often claimed through the literature that session types, whatever the flavour, help one reason about the type and order of communication. On one hand, this can be interpreted as proving program correctness, on the other, as helping one to understand the protocol easier. Most likely it is a combination of both.

Other papers make claims about the effect session types will have on the code programmers will produce. Session types "allow programmers to organise programs as a combination of multiple flows of (possibly unbounded) reciprocal interactions in a simple and elegant way, subsuming the preceding communication primitives such as method invocation and rendez-vous" [7]. Session types allow for "a concise description of the continuous interactions among partners in a concurrent computation" [24].

On its website, our project—ABCD, an EPSRC programme grant—boldly summarises the views regarding the influence of session types. It claims that "Session types will play a crucial role in all aspects of software" [1]. It draws a parallel with the data type, claiming that session types will be for communication that the data type is for algorithms. It continues to explain how "architects, programmers, and software tools will all be aided by session types to reduce the cost of producing concurrent and distributed software, while increasing its reliability and efficiency".

3. Hypotheses

With these claims in mind, some hypotheses regarding session types can now be formulated.

Hypothesis 1. *Programming with session types can be understood and used efficiently by real world programmers.* All papers extending various programming languages [17, 3, 11, 9, 13, 16] assume that this is the case. However, session types add another layer of complexity to programming. The user has to put in the effort of learning the new feature, and deal with any overheads associated with its use. For instance, linearity, a key concept in session types, is generally seen as difficult to grasp. Such overheads may have an effect on how

fast and how well a programmer can start using session type languages and tools with their systems.

Hypothesis 2. *Programming with session types improves software reliability. Protocols specified with session types can be formally verified. This will result in software with less bugs and less likely to unexpectedly fail.*

This is an implicit hypothesis, found throughout the literature that forms the very basis of session type research. It relies on the idea that well-typed programs never fail. In the case of static type systems, errors are caught at compile time, before the software is even run. In contrast, in dynamic systems components are monitored to ensure that they do not receive any incorrect inputs. In either case, communication errors should be caught before the system is deployed.

While this hypothesis may seem obvious, just as programming with static type systems seems safer than with dynamic systems, there is little evidence to support it.

Hypothesis 3. *Session types encourage programmers to structure their code better, thus making the code more readable and easier to understand.*

Found in various papers starting with earlier works [7, 24], this hypothesis plays a big part in Scribble [19] and the Mungo/StMungo [11] tool chain. Furthermore, types can be viewed as a form of documentation, in this case giving the programmer additional information about the communication being modelled. A developer looking at a new piece of software for the first time, can look at the session type definition and quickly establish, who the participants are, and what sort of data is being exchanged.

The next three hypotheses are closely related; however, since the concepts are different enough and the way one would go about testing these is not necessarily the same, we opted for expressing them separately.

Hypothesis 4. *Session types promote modular programming, through the structure given to the implementation.*

Subprotocols and polymorphism may encourage developers to organise code into modules following the structure of the session types. In the case of multiparty session types, projection corresponds to modular decomposition, having independent roles separated which may encourage programmers to have a more modular implementation. Tools such as Scribble [19], Mungo/StMungo [11] encourage the user to structure their code in according to protocol structure and according to the various entities at play in the protocol. This may affect the overall modularity of the system.

Hypothesis 5. *Session types can help increase software maintainability.*

The structure imposed on code promotes separation of concerns, having a clear distinction between the session types implementation and the rest of the program. This should make any changes easier to accommodate [17]. Moreover, through subtyping, components can safely be replaced. Subprotocols and multiparty session type projection could also be influencing factors in software maintainability.

This ties in closely with the previous hypotheses, each of them being influencing factors in software maintainability.

Hypothesis 6. *Session typed languages provide useful additional diagnostic information that can make debugging faster and more precise.*

The structure of the session type can be exploited both by programmers, and by tools to identify communication errors. In static systems such errors can be caught and reported by the compiler, while in dynamic systems the errors would be reported by the monitors at runtime. Error messages can be related to the type definition, helping the programmer quickly identify where the correction needs to be made.

Hypothesis 7. *Programming with session types speeds up the development process.*

The session types discipline can help developers design and implement their systems more efficiently. The formalism can help in modelling protocols and distributed scenarios. Session typed languages provide useful additional safeguards and diagnostic information that lead to a system with the expected behaviour with less effort.

Session types may also be useful for testing. They could be used as a guide in manually writing unit tests, or by a tool to automatically generate them.

Hypothesis 8. *Programming with session types can lead to improved software efficiency.*

Session types can be exploited to optimise concurrent and distributed software, or fine tune communication in the case of parallel systems [23, 14].

4. Next Steps and Conclusion

Starting from the above hypotheses the next steps are to conduct empirical studies of the effect and usefulness of session types on software development. Some, such as Hypothesis 8, would need more research and development into that particular area to generate artefacts to evaluate. For others, things are more straightforward. While for the most part there is no clear-cut way to do it, studies of the different language designs and implementations can be carried out. It is expected that various evaluation methodologies would need to be used, such as observations, interviews, or standardised questionnaires. Some relevant studies have been presented in [2], to compare the programming efficiency of a proposed language versus a mainstream one, in [12] to measure the effect of a new tool on programming tasks, or in [10] to gain understanding of the usage and adoption of a tool by deploying it among professional programmers for several weeks.

There are different approaches to adding session types to existing languages. In the case of Java, we have Session J [9], Mungo, and Hu and Yoshida's API generation [8]. However, there has not been a comparison of the benefits of each approach for practical programming. Most papers on adding session types to programming languages focus on the language, not on the associated development environment or methodology. Hu's API generation approach allows a stan-

dard Java IDE such as Eclipse to inform the programmer about protocol errors. There are additional ways in which an IDE could help programmers, for example by showing a transition diagram or the complete session type. These possibilities have not yet been fully investigated or compared.

While session types have been developed for some time now with industry input, a closer look at what exactly their effect is on software development is necessary. Otherwise there is the risk of developing something that may not be quite right or suitable for its purpose. 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 [21]. By identifying which designs and implementations help or hinder programmers, we can improve them to help developers use session type effectively. Our hope is that this will lead to session type designs, tools and implementations that are better suited for "real-world", industrial software development.

Acknowledgments

This research was supported by UK EPSRC grant "From Data Types to Session Types: A Basis for Concurrency and Distribution" (EP/K034413/1).

References

[1] A Basis for Concurrency and Distribution. `http://groups.inf.ed.ac.uk/abcd/`.

[2] F. Cuenca, J. V. d. Bergh, K. Luyten, and K. Coninx. A user study for comparing the programming efficiency of modifying executable multimodal interaction descriptions: A domain-specific language versus equivalent event-callback code. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU 2015, pages 31–38. ACM, 2015.

[3] S. Fowler. An Erlang implementation of multiparty session actors. In Proceedings 9th *Interaction and Concurrency Experience*, volume 223 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–23. Open Publishing Association, 2016.

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

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

[6] K. Honda. Types for dyadic interaction. In *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR)*, volume 715 of *Springer LNCS*, pages 509–523, 1993.

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

[8] R. Hu and N. Yoshida. Hybrid session verification through endpoint API generation. In *FASE '16*, volume 9633 of *Springer LNCS*, pages 401–418, 2016.

[9] R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in Java. In *European Conference on Object-Oriented Programming*, pages 516–541. Springer, 2008.

[10] M. R. Jakobsen and K. Hornbæk. Fisheyes in the field: using method triangulation to study the adoption and use of a source code visualization. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1579–1588. ACM, 2009.

[11] D. Kouzapas, O. Dardha, R. Perera, and S. J. Gay. Typechecking protocols with Mungo and StMungo. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, PPDP '16, pages 146–159. ACM, 2016.

[12] P. O. Kristensson and C. L. Lam. Aiding programmers using lightweight integrated code visualization. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 17–24. ACM, 2015.

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

[14] D. Mostrous and N. Yoshida. Two session typing systems for higher-order mobile processes. In *International Conference on Typed Lambda Calculi and Applications*, pages 321–335. Springer, 2007.

[15] R. Neykova. Session types go dynamic or how to verify your Python conversations. *arXiv preprint arXiv:1312.2704*, 2013.

[16] N. Ng and N. Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 174–184. ACM, 2016.

[17] N. Ng, N. Yoshida, X. Y. Niu, and K. H. Tsoi. Session types: towards safe and fast reconfigurable programming. *ACM SIGARCH Computer Architecture News*, 40(5):22–27, 2012.

[18] F. Pfenning and D. Griffith. Polarized substructural session types. In *International Conference on Foundations of Software Science and Computation Structures*, pages 3–22. Springer, 2015.

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

[20] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. *PARLE, Springer LNCS*, 817:398–413, 1994.

[21] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen. Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 456–468. ACM, 2016.

[22] P. Thiemann and V. T. Vasconcelos. Context-free session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 462–475. ACM, 2016.

[23] Exploiting Parallelism through Type Transformations for Hybrid Manycore Systems. `http://tytra.org.uk`.

[24] V. T. Vasconcelos. Fundamentals of session types. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 158–186. Springer, 2009.