

Theory and Applications of Behavioural Types

Edited by

Simon Gay¹, Vasco T. Vasconcelos², Philip Wadler³, and
Nobuko Yoshida⁴

1 University of Glasgow, GB, simon.gay@glasgow.ac.uk

2 University of Lisbon, PT, vmvasconcelos@ciencias.ulisboa.pt

3 University of Edinburgh, GB, wadler@inf.ed.ac.uk

4 Imperial College London, GB, yoshida@doc.ic.ac.uk

Abstract

This report documents the programme and the outcomes of Dagstuhl Seminar 17051 “Theory and Applications of Behavioural Types”. Behavioural types describe the dynamic aspects of programs, in contrast to data types, which describe the fixed structure of data. Perhaps the most well-known form of behavioural types is session types, which are type-theoretic specifications of communication protocols. More generally, behavioural types include typestate systems, which specify state-dependent availability of operations; choreographies, which specify collective communication behaviour; and behavioural contracts.

In recent years, research activity in behavioural types has increased dramatically, in both theoretical and practical directions. Theoretical work has explored new relationships between established behavioural type systems and areas such as linear logic, automata theory, process calculus testing theory, dependent type theory, and model-checking. On the practical side, there are several implementations of programming languages, programming language extensions, software development tools, and runtime monitoring systems, which are becoming mature enough to apply to real-world case studies.

The seminar brought together researchers from the established, largely European, research community in behavioural types, and other participants from outside Europe and from related research topics such as effect systems and actor-based languages. The questions that we intended to explore included:

- How can we understand the relationships between the foundations of session types in terms of linear logic, automata, denotational models, and other type theories?
- How can the scope and applicability of behavioural types be increased by incorporating ideas and approaches from gradual typing and dependent type theory?
- What is the relationship, in terms of expressivity and tractability, between behavioural types and other verification techniques such as model-checking?
- What are the theoretical and practical obstacles to delivering behavioural types to software developers in a range of mainstream programming languages?
- What are the advantages and disadvantages of incorporating behavioural types into standard programming languages or designing new languages directly based on the foundations of session types?
- How can we evaluate the effectiveness of behavioural types in programming languages and software development?

Seminar January 29–3, 2017 – <http://www.dagstuhl.de/17051>

1998 ACM Subject Classification D.1.1 Applicative (Functional) Programming, D.1.3 Concurrent Programming, D.2.4 Software/Program Verification, D.3.1 Formal Definitions and Theory, D.3.3 Language Constructs and Features, F.3.2 Semantics of Programming Languages



Except where otherwise noted, content of this report is licensed under a Creative Commons BY 3.0 Unported license

Theory and Applications of Behavioural Types, *Dagstuhl Reports*, Vol. 7, Issue 1, pp. 158–189

Editors: Simon Gay, Vasco T. Vasconcelos, Philip Wadler, and Nobuko Yoshida



DAGSTUHL
REPORTS

Dagstuhl Reports
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Keywords and phrases Behavioural Types, Programming Languages, Runtime Verification, Type Systems

Digital Object Identifier 10.4230/DagRep.7.1.158

Edited in cooperation with Tzu-Chun Chen

1 Executive Summary

Simon Gay

Vasco T. Vasconcelos

Philip Wadler

Nobuko Yoshida

License © Creative Commons BY 3.0 Unported license
© Simon Gay, Vasco T. Vasconcelos, Philip Wadler, and Nobuko Yoshida

Behavioural types describe dynamic aspects of a program, in contrast to data types, which describe the fixed structure of data. Behavioural types include session types, tpestate, choreographies, and behavioural contracts. Recent years have seen a substantial increase in research activity, including theoretical foundations, design and implementation of programming languages and tools, studies of the relationships between different forms of behavioural types, and studies of the relationships between behavioural types and more general type-theoretic ideas such as gradual typing and dependent typing. The aim of this seminar was to bring together researchers on behavioural types and related topics, in order to understand and advance the state of the art.

Many of the participants have been active in COST Action IC1201: Behavioural Types for Reliable Large-Scale Software Systems (BETTY), a European research network on behavioural types. Other participants were invited from related research areas and from outside Europe, in order to broaden the scope of the seminar and to make connections between communities.

The programme for the first half of the week was planned in advance, with priority given to two kinds of presentation: (1) demonstrations of programming language implementations and tools, and (2) presentations by participants from outside the BETTY community. The programme for the second half of the week evolved during the seminar, with more emphasis on group discussion sessions.

The seminar was judged to be a success by all the participants. At least one conference submission resulted from collaboration started during the week, other existing collaborations made substantial progress, and several participants planned a submission to the EU RISE funding scheme. We intend to propose a follow-on seminar on a similar topic in the future.

This report contains the abstracts of the talks and software demonstrations, and summaries of the group discussion sessions.

2 Table of Contents

Executive Summary

Simon Gay, Vasco T. Vasconcelos, Philip Wadler, and Nobuko Yoshida 159

Overview of Talks

Towards Inferring Session Types

Gul Agha 163

Effects as Capabilities

Nada Amin 163

Observed Communication Semantics for Classical Processes

Robert Atkey 165

Stateful Programming in Idris

Edwin Brady 165

Behavioral Types, Type Theory, and Logic

Luis Caires 166

Session Types for Fault-tolerant Distributed Systems

Patrick Thomas Eugster 166

Statically Detecting (Dead)locks in the Linear Pi-calculus

Adrian Francalanza 166

Gradual Typing

Ronald Garcia 167

Practical Affine Types and Typestate-Oriented Programming

Philipp Haller 167

DCR Tools

Thomas Hildebrandt 168

Using Session Types for Reasoning About Boundedness in the Pi-Calculus

Hans Hüttel 168

Session-ocaml: A Session-based Library with Polarities and Lenses

Keigo Imai, Nobuko Yoshida, and Shoji Yuen 169

Lightweight Functional Session Types

J. Garrett Morris 169

Composable Actor Behaviour

Roland Kuhn 169

Adaptive Interaction-Oriented Choreographies in Jolie

Ivan Lanese 170

Failure-Aware Protocol Programming

Hugo-Andrés López 170

Chaperone Contracts for Higher-Order Sessions

Hernán Melgratti 170

Static Deadlock Detection for Go

Nicholas Ng and Nobuko Yoshida 171


Session Types with Linearity in Haskell <i>Dominic Orchard and Nobuko Yoshida</i>	172
A Simple Library Implementation of Binary Sessions <i>Luca Padovani</i>	172
Concurrent TypeState-Oriented Programming <i>Luca Padovani</i>	172
Precise Subtyping <i>Jovanka Pantovic</i>	173
Concurrent C0 <i>Frank Pfenning</i>	173
Manifest Sharing with Session Types <i>Frank Pfenning</i>	173
Detecting Concurrency Errors of Erlang Programs via Systematic Testing <i>Konstantinos Sagonas</i>	174
Lightweight Session Programming in Scala <i>Alceste Scalas</i>	174
Programming Protocols with Scribble and Java <i>Alceste Scalas</i>	175
Partial Type Equivalences for Verified Dependent Interoperability <i>Nicolas Tabareau</i>	175
Gradual Session Types <i>Peter Thiemann</i>	176
Choreographies, Modularly: Components for Communication Centred Programming <i>Hugo Torres Vieira</i>	176
From Communicating Machines to Graphical Choreographies <i>Emilio Tuosto</i>	177
Fencing off Go <i>Nobuko Yoshida</i>	177
Undecidability of Asynchronous Session Subtyping <i>Nobuko Yoshida</i>	177
Working groups	
Group Discussion: Integrating Static and Dynamic Typing <i>Laura Bocchi</i>	178
Group Discussion: Behavioural Types in Non-Communication Domains <i>Simon Gay</i>	179
Group discussion: Dependent Session Types <i>Simon Gay</i>	181
Group Discussion: Future Activities and Funding Possibilities <i>Simon Gay</i>	182
Group Discussion: Session Sharing and Races <i>Simon Gay</i>	183

Group Discussion: Standardisation of a Programming Language with Session Types <i>Simon Gay</i>	184
Group Discussion: Behavioural Types for Mainstream Software Development <i>Philipp Haller</i>	185
Group Discussion: Educational Resources for Behavioural Types <i>Hugo Torres Vieira</i>	186
Open problems	
A Meta Theory for Testing Equivalences <i>Giovanni Tito Bernardi</i>	188
Participants	189

3 Overview of Talks

3.1 Towards Inferring Session Types


Gul Agha (University of Illinois – Urbana-Champaign, US)

License  Creative Commons BY 3.0 Unported license
© Gul Agha

In sequential systems, programmers are responsible for specifying a total order of events in a system. This results in overly constraining when events may occur. In contrast, concurrent systems allow nondeterministic interleaving of actions at autonomous actors. Without additional constraints on the order of events at participating actors, an interleaving may lead to incorrect operations – for example, one that results in a deadlock. Moreover, the correct order of events at an actor is dependent on what interaction it is participating in. For example, an actor may be in the role of a client in one interaction protocol and the role of a backup server in another. To facilitate such flexibility, synchronization should be specified separately from the functional behavior of an actor – in terms of its interface rather than its representation. I will argue for the use of synchronization constraints as a user friendly language whose semantics is given by multiparty session types. Moreover, I propose that it is possible to infer session types with a degree of confidence by analyzing ordering patterns in traces of program execution: if an ordering pattern is repeatedly observed in such traces, we can impose the ordering to avoid Heisenbugs that may occur from rarer schedules that violate the observed order.

3.2 Effects as Capabilities

Nada Amin (EPFL – Lausanne, CH)

License  Creative Commons BY 3.0 Unported license
© Nada Amin
Joint work of Fengyun Liu, Nicolas Stucki, Sandro Stucki, Martin Odersky

It seems quite natural that one should track effects by means of a static typing discipline, similarly to what is done for arguments and results of functions. After all, to understand a function’s contract and how it can be composed, knowing its effects is just as important as knowing the types of its arguments and result. Yet after decades of research [3, 4, 6, 5, 7, 8, 11, 13], why are effect systems not as mainstream as type systems?

The static effect discipline with the most widespread use is no doubt Java’s system of checked exceptions. Ominously, they are now widely regarded as a mistake [2]. One frequent criticism is about the notational burden they impose. Throws clauses have to be laboriously threaded through all call chains. All too often, programmers make the burden go away by catching and ignoring all exceptions that they think cannot occur in practice. In effect, this disables both static and dynamic checking, so the end result is less safe than if one started with unchecked exceptions only. Another common problem of Java’s exceptions is lack of polymorphism: Often we would like to express that a function throws the same exceptions as the (statically unknown) functions it invokes. Effect polymorphism can be expressed in Java only at the cost of very heavy notation, so it is usually avoided. Java’s system of checked exceptions may be an extreme example, but it illustrates the general pitfalls of checking effects by shifting the burden of tracking effects to the programmer.

We are investigating a new approach to effect checking, that flips the requirements around. The central idea is that instead of talking about effects we talk about capabilities. For instance, instead of saying a function “throws an `IOException`” we say that the function “needs the capability to throw an `IOException`”. Capabilities are modeled as values of some capability type. For instance, the aforementioned capability could be modeled as a value of type `CanThrow[IOException]`. A function that might throw an `IOException` needs to have access to an instance of this type. Typically it takes an argument of the type as a parameter.

It turns out that that the treatment of effects as capabilities gives a simple and natural way to express “effect polymorphism” – the ability to write a function once, and to have it interact with arguments that can have arbitrary effects. Since capabilities are just function parameters, existing language support for polymorphism, such as type abstraction and subtyping, is readily applicable to them. But there are two areas where work is needed to make capabilities as effects sound and practical.

First, when implemented naively, capabilities as parameters are even more verbose than effect declarations such as throws clauses. Not only do they have to be declared, but they also have to be propagated as additional arguments at each call site. We propose to make use of the concept of implicit parameters [9, 10, 14] to cut down on the boilerplate. Implicit parameters make call-site annotations unnecessary, but they still have to be declared just like normal parameters. To avoid repetition, we propose to investigate a way of abstracting implicit parameters into implicit function types. With implicits, the approach provides the common case of propagation for free, and an easy migration path from impure to pure.

Second, there is one fundamental difference between the usual notions of capabilities and effects: capabilities can be captured in closures. This means that a capability present at closure construction time can be preserved and accessed when the closure is applied. Effects on the other hand, are temporal: it generally does make a difference whether an effect occurs when a closure is constructed or when it is used. We propose to address this discrepancy by introducing a “pure function” type, instances of which are not allowed to close over effect capabilities.

In this talk, we report on work in progress, exploring the idea of effects as capabilities in detail. We have worked on minimal formalizations for implicit parameters and pure functions and studied encodings of higher-level language constructs into these calculi. Based on the theoretical modelization we are developing a specification for adding effects to Scala.


References

- 1 Lewis, Jeffrey R and Launchbury, John and Meijer, Erik and Shields, Mark B. *Implicit parameters: Dynamic scoping with static types*. Proceedings of POPL, 2000.
- 2 Thomas Whitmore. *Checked exceptions, Java’s biggest mistake*. Literal Java Blog, 2015.
- 3 Gifford, David K and Lucassen, John M. *Integrating functional and imperative programming*. Proceedings of POPL, 1986.
- 4 Lucassen, John M and Gifford, David K. *Polymorphic effect systems*. Proceedings of POPL, 1988.
- 5 Talpin, Jean-Pierre and Jouvelot, Pierre. *The type and effect discipline*. Information and computation, 1994.
- 6 Talpin, Jean-Pierre and Jouvelot, Pierre. *Polymorphic type, region and effect inference*. Journal of Functional Programming, 1992.
- 7 Wadler, Philip and Thiemann, Peter. *The marriage of effects and monads*. ACM Transactions on Computational Logic, 2003.
- 8 Filinski, Andrzej. *Monads in Action*. Proceedings of POPL, 2010.

- 9 Odersky, Martin. *Poor Man's Typeclasses*. Presentation to IFIP WG 2.8, 2006. <http://lampwww.epfl.ch/~odersky/talks/wg2.8-boston06.pdf>
- 10 Oliveira, Bruno CdS and Moors, Adriaan and Odersky, Martin. *Type classes as objects and implicits*. Proceedings of OOPSLA, 2010.
- 11 Rytz, Lukas and Odersky, Martin and Haller, Philipp. *Lightweight polymorphic effects*. Proceedings of ECOOP, 2012.
- 12 Ben Lippmeier. *Type Inference and Optimisation for an Impure World*. PhD Thesis, Australian National University, 2010.
- 13 Andrej Bauer and Matija Pretnar. *Programming with algebraic effects and handlers*. J. Log. Algebr. Meth. Program. 2015.
- 14 Oliveira, Bruno C.d.S. and Schrijvers, Tom and Choi, Wontae and Lee, Wonchan and Yi, Kwangkeun. *The Implicit Calculus: A New Foundation for Generic Programming*. Proceedings of PLDI, 2012.

3.3 Observed Communication Semantics for Classical Processes

Robert Atkey (University of Strathclyde – Glasgow, GB)

License  Creative Commons BY 3.0 Unported license
 © Robert Atkey
 URL <http://materials.dagstuhl.de/files/17/17051/17051.RobertAtkey.Slides.pdf>

Classical Linear Logic (CLL) has long inspired readings of its proofs as communicating processes. Wadler's CP calculus is one of these readings. Wadler gave CP an operational semantics by selecting a subset of the cut-elimination rules of CLL to use as reduction rules. This semantics has an appealing close connection to the logic, but does not resolve the status of the other cut-elimination rules, and does not admit an obvious notion of observational equivalence. We propose a new operational semantics for CP based on the idea of observing communication, and use this semantics to define an intuitively reasonable notion of observational equivalence. To reason about observational equivalence, we use the standard relational denotational semantics of CLL. We show that this denotational semantics is adequate for our operational semantics. This allows us to deduce that, for instance, all the cut-elimination rules of CLL are observational equivalences.

3.4 Stateful Programming in Idris

Edwin Brady (University of St. Andrews, GB)

License  Creative Commons BY 3.0 Unported license
 © Edwin Brady

I present a library for giving precise types to interactive, stateful programs in Idris, a dependently typed pure functional programming language. I show how to describe state transition systems in types, capturing pre- and post-conditions of operations, and dealing with errors and feedback from the environment. I demonstrate with socket programming, and an asynchronous server for a simple network protocol.

3.5 Behavioral Types, Type Theory, and Logic

Luis Caires (New University of Lisbon, PT)

License  Creative Commons BY 3.0 Unported license
© Luis Caires

Joint work of Luis Caires, Frank Pfenning, Bernardo Toninho, Jorge Perez, Joao Seco

We review a collection of recent work providing a logical Curry-Howard foundation to the notion of behavioural type, useful to describe intensional usage protocols for state-full objects such as e.g., sessions. In particular we show how the basic linear logic interpretation discovered by Caires and Pfenning can be naturally extended to incorporate dependent types, allowing us to express higher order processes, value dependent behaviour, assertions, and proof carrying code; polymorphic types, allowing us to express behavioural genericity, and sums, allowing us to express non-determinism, and other typing constructs, relevant for typing shared state concurrency. We conclude by arguing that such linear logic interpretations provide a way of rooting the notion of behavioural type, and the notion of session type in particular, in the common house of Type Theory, from which the most fundamental programming language typing concepts have also emerged.

3.6 Session Types for Fault-tolerant Distributed Systems

Patrick Thomas Eugster (TU Darmstadt, DE)

License  Creative Commons BY 3.0 Unported license
© Patrick Thomas Eugster

Distributed systems are hard to get right, due to the possibility of partial failures where certain components or participants fail while others continue to operate. Session types are an appealing approach to aid programmers in reasoning about complex interaction in the presence of partial failures, yet have so far focused more on high-level programming models such as Web Services, where many failures are abstracted. Our contributions to address the problem in this talk are twofold. First we propose a set of abstractions allowing programmers to describe the handling of failures of different kinds. Together with information about the underlying system model we infer how and where to notify participants of failures in order to achieve a consistent failure handling as described by programmers. Second, we discuss the integration of failure handling mechanisms with failure masking approaches. In the latter context, we focus on supporting different broadcast models in order to support redundancy.

3.7 Statically Detecting (Dead)locks in the Linear Pi-calculus

Adrian Francalanza (University of Malta – Msida, MT)

License  Creative Commons BY 3.0 Unported license
© Adrian Francalanza

Joint work of Adrian Francalanza, Marco Giunti, Antonio Ravara

We propose an alternative approach to the study of type-based (dead)lock analysis in the context of the linear pi-calculus. Instead of targeting the class of (dead)lock-free processes, we study type-based techniques for statically approximating the class of (dead)locked processes. We develop type-based analyses that return lists of problematic channels on which (dead)locks

occur once the analysed program is executed. Such information is arguably more useful in the case of erroneous programs, because it directs the programmer to the source of the error. Another distinguishing aspect of our work is that the semantic guarantees of our type-based analysis ensure verdict precision (i.e. the absence of false negatives), but allow for occasionally classifying erroneous programs as bug-free. This differs from more mainstream static analysis approaches that tend to favour soundness, but is more useful for automated error resolution procedures where, ideally, the analysed programs are not be modified unnecessarily.

3.8 Gradual Typing


Ronald Garcia (University of British Columbia – Vancouver, CA)

License  Creative Commons BY 3.0 Unported license
© Ronald Garcia

Programming language design has recently exhibited a recurring trend: languages perceived as “statically typed” are beginning to exhibit “dynamic typing” features, while “dynamically typed” languages are exhibiting the converse. The theory of Gradual Typing has been developed to help provide a foundation for languages that wish to exhibit similar combinations while ensuring sound reasoning principles. This talk gives a high-level introduction to the concepts underlying gradual typing, with some historical context, some recent work on developing a general framework for developing gradually typed languages, and a list of open challenges that pertain to the behavioural types community.

3.9 Practical Affine Types and Typestate-Oriented Programming

Philipp Haller (KTH Royal Institute of Technology – Stockholm, SE)

License  Creative Commons BY 3.0 Unported license
© Philipp Haller
URL <http://materials.dagstuhl.de/files/17/17051/17051.PhilippHaller.Slides1.pdf>

Aliasing is a known source of challenges in the context of imperative object-oriented languages, which have led to important advances in type systems for aliasing control. However, their large-scale adoption has turned out to be a surprisingly difficult challenge. While new language designs show promise, they do not address the need of aliasing control in existing languages.

This talk presents a new approach to isolation and uniqueness in an existing, widely-used language, Scala. The approach is unique in the way it addresses some of the most important obstacles to the adoption of type system extensions for aliasing control. First, adaptation of existing code requires only a minimal set of annotations. Only a single bit of information is required per class. Surprisingly, the talk shows that this information can be provided by the object-capability discipline, widely-used in program security. The type system is implemented for the full Scala language, providing, for the first time, a sound integration with Scala’s local type inference. Finally, we present an ongoing effort to generalize the type system to typestates.

3.10 DCR Tools

Thomas Hildebrandt (IT University of Copenhagen, DK)

License  Creative Commons BY 3.0 Unported license
© Thomas Hildebrandt

Joint work of Thomas Hildebrandt, Søren Debois, Tijs Slaats, Morten Marquard

Main reference S. Debois, T. T. Hildebrandt, M. Marquard, T. Slaats, “The DCR Graphs Process Portal”, in Proc. of the BPM Demo Track 2016, pp. 7–11, 2016.

URL <http://ceur-ws.org/Vol-1789/bpm-demo-2016-paper2.pdf>

The presentation give a quick tour of the tools for modelling and simulating Dynamic Condition Response (DCR) graphs. DCR graphs is a declarative process notation for the modelling of flexible adaptable choreographies developed through a number of research projects jointly with industry with the aim to support the design, analysis and execution of flexible and adaptable workflow and business processes. Formally, DCR graphs generalise labelled event structures to allow (1) finite descriptions of infinite behaviour, (2) represent mandatory (pending) events that must eventually happen or become in conflict with events that happened, (3) allow dynamic, asymmetric conflict. Regarding expressiveness, the core model can express exactly the languages that are a union of regular and omega-regular languages (if one ignore true concurrency) – but the model maps to true concurrency models such as event structures. The presentation focus on the tools that have been developed at the IT University of Copenhagen (<http://dcr.tools>) and the industry partner Exformatics (<http://dcrgraphs.net>). The development of the two tools also demonstrate a model for transferring research to industry, where the academic tool serves as a means to demonstrate new developments that later are transferred to the industrial tool.

3.11 Using Session Types for Reasoning About Boundedness in the Pi-Calculus

Hans Hüttel (Aalborg University, DK)

License  Creative Commons BY 3.0 Unported license
© Hans Hüttel

Depth-bounded and name-bounded processes are pi-calculus processes for which some of the decision problems that are undecidable for the full calculus become decidable. P is depth-bounded at level k if every reduction sequence for P contains successor processes with at most k active nested restrictions. P is name-bounded at level k if every reduction sequence for P contains successor processes with at most k active bound names. We use binary session types to formulate two type systems that give sound characterizations of these properties: If a process is well-typed, it is depth-bounded, respectively name-bounded.

3.12 Session-ocaml: A Session-based Library with Polarities and Lenses

Keigo Imai (Gifu University, JP), Nobuko Yoshida (Imperial College London, GB), and Shoji Yuen (Nagoya University, JP)

License © Creative Commons BY 3.0 Unported license
 © Keigo Imai, Nobuko Yoshida, and Shoji Yuen
Main reference K. Imai, N. Yoshida, S. Yuen, “Session-ocaml: a session-based library with polarities and lenses”, Manuscript, 2017.
URL <http://www.ct.info.gifu-u.ac.jp/~keigo/session-ocaml/>

We propose session-ocaml, a novel library for session-typed concurrent/distributed programming in OCaml. Our technique is based only on the parametric polymorphism, hence common to various statically-typed programming languages. The key ideas are follows: (1) The polarised session types gives an alternative formulation of duality enabling OCaml to infer the appropriate session type in a session with a reasonable notational overhead. (2) A parameterized monad with lenses enables full session type implementation including delegation. We show an application of session-ocaml including an SMTP client and a database server.

3.13 Lightweight Functional Session Types

J. Garrett Morris (University of Edinburgh, GB)

License © Creative Commons BY 3.0 Unported license
 © J. Garrett Morris
Joint work of Sam Lindley, J. Garrett Morris
Main reference S. Lindley, J. G. Morris, “Lightweight Functional Session Types”, in “Behavioural Types: from Theory to Tools”, River Publishers, 2017; pre-print available from author’s webpage.
URL <http://homepages.inf.ed.ac.uk/slindley/papers/fst.pdf>

Row types provide an account of extensibility that combines well with parametric polymorphism and type inference. We discuss the integration of row types and session types in a concurrent functional programming language, and how row types can be used to describe extensibility in session-typed communication.

3.14 Composable Actor Behaviour

Roland Kuhn (Actyx AG – München, DE)


License © Creative Commons BY 3.0 Unported license
 © Roland Kuhn
URL <http://materials.dagstuhl.de/files/17/17051/17051.RolandKuhn.Slides.pdf>

This presentation focuses on the composition of the behavior of distributed components—modeled using Actors—from reusable pieces. Allowing abstraction and type-safety to be applied within these components for operations that are fundamentally non-local is seen as a prerequisite for offering safe construction of distributed systems in a widely and practically applicable programming tool.

Please see the linked article for an introduction to the tool (based on Scala and Akka); pointers to the source code and how to try it out are given towards the end.

3.15 Adaptive Interaction-Oriented Choreographies in Jolie


Ivan Lanese (University of Bologna, IT)

License  Creative Commons BY 3.0 Unported license
© Ivan Lanese

We will give a demo of AIOCJ, Adaptive Interaction-Oriented Choreographies in Jolie. The tool is composed by an Eclipse plugin and a running environment to program distributed applications, and to adapt them at runtime by replacing pre-selected pieces of code with new code coming from outside the application. Notably, a single program describes the whole distributed application, and a single adaptation may involve many components. The application is free from communication races and deadlocks by construction, both before and after the adaptation.

3.16 Failure-Aware Protocol Programming


Hugo-Andrés López (Technical University of Denmark – Lyngby, DK)

License  Creative Commons BY 3.0 Unported license
© Hugo-Andrés López
URL <http://materials.dagstuhl.de/files/17/17051/17051.Hugo-Andr%C3%A9sL%C3%B3pez.Slides.pdf>

Motivated by challenging scenarios in Cyber-Physical Systems (CPS), we study how choreographic programming can cater for dynamic infrastructures where not all endpoints are always available. We introduce the Global Quality Calculus (GCq), a variant of choreographic programming for the description of communication systems where some of the components involved in a communication might fail. GCq features novel operators for multiparty, partial and collective communications. In this talk I will study the nature of failure-aware communication: First, we introduce GCq syntax, semantics and examples of its use. The interplay between failures and collective communications in a choreography can lead to choreographies that cannot progress due to absence of resources. In our second contribution, we provide a type system that ensures that choreographies can be realized despite changing availability conditions. A specification in GCq guides the implementation of distributed endpoints when paired with global (session) types. Our third contribution provides an endpoint-projection based methodology for the generation of failure-aware distributed processes. We show the correctness of the projection, and that well-typed choreographies with availability considerations enjoy progress.

3.17 Chaperone Contracts for Higher-Order Sessions

Hernán Melgratti (University of Buenos Aires, AR)

License  Creative Commons BY 3.0 Unported license
© Hernán Melgratti
URL <http://materials.dagstuhl.de/files/17/17051/17051.Hern%C3%A1nMelgratti.Slides.pdf>

Sessions in concurrent programs play the same role of functions and objects in sequential ones. This calls for a way to describe properties and relationships of messages exchanged in sessions using behavioral contracts, in the spirit of the design-by-contract approach to software development. Unlike functions and objects, however, the kind, direction, and properties of

messages exchanged in a session may vary over time, as the session progresses. This feature of sessions enriches the “behavioral” qualification of session contracts, which must evolve along with the session they describe.

In this work, we extend to sessions the notion of chaperone contract (roughly, a contract that applies to a mutable object) and investigate the ramifications of contract monitoring in a higher-order calculus equipped with a session type system. We give a characterization of correct module, one that honors the contracts of the sessions it uses, and prove a blame theorem. Guided by the calculus, we describe a lightweight and portable implementation of monitored sessions as an OCaml module with which programmers can benefit from static session type checking and dynamic contract monitoring using an off-the-shelf version of OCaml.

3.18 Static Deadlock Detection for Go

Nicholas Ng (Imperial College London, GB) and Nobuko Yoshida (Imperial College London, GB)

License © Creative Commons BY 3.0 Unported license
© Nicholas Ng and Nobuko Yoshida

Joint work of Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida

Main reference J. Lange, N. Ng, B. Toninho, N. Yoshida, “Fencing off go: liveness and safety for channel-based programming”, in Proc. of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017), pp. 748–761, ACM, 2017.

URL <http://dx.doi.org/10.1145/3009837.3009847>

Go is a production-level statically typed programming language whose design features explicit message-passing primitives and lightweight threads, enabling (and encouraging) programmers to develop concurrent systems where components interact through communication more so than by lock-based shared memory concurrency. Go can only detect global deadlocks at runtime, but provides no compile-time protection against all too common communication mismatches or partial deadlocks.

In this talk we present a static verification framework for liveness and safety in Go programs, able to detect communication errors and partial deadlocks in a general class of realistic concurrent programs, including those with dynamic channel creation, unbounded thread creation and recursion. Our approach infers from a Go program a faithful representation of its communication patterns as a behavioural type. By checking a syntactic restriction on channel usage, dubbed fencing, we ensure that programs are made up of finitely many different communication patterns that may be repeated infinitely many times. This restriction allows us to implement a decision procedure for liveness and safety in types which in turn statically ensures liveness and safety in Go programs.

Details of our verification tool-chain are available on <http://mrg.doc.ic.ac.uk/tools/gong/>.

3.19 Session Types with Linearity in Haskell

Dominic Orchard (University of Kent – Canterbury, GB) and Nobuko Yoshida (Imperial College London, GB)

License © Creative Commons BY 3.0 Unported license
© Dominic Orchard and Nobuko Yoshida

Main reference D. Orchard, N. Yoshida, “Sessions types with linearity in Haskell”, in Behavioural Types: from Theory to Tools, River Publishers, 2017.

Type systems with parametric polymorphism can encode communication patterns over channels, providing part of the power of session types. However, statically enforcing linearity properties of session types is more challenging. Haskell provides various features that can overcome this challenge. However, current approaches lead to a programming style which is either non-idiomatic for Haskell, or types which are too hard to write and read. I’ll demo an early version of a Haskell library for session types that does it all: session-typed, linear, idiomatic Haskell with easy-to-read-and-write types.

3.20 A Simple Library Implementation of Binary Sessions

Luca Padovani (University of Turin, IT)

License © Creative Commons BY 3.0 Unported license
© Luca Padovani

Main reference L. Padovani, “A simple library implementation of binary sessions”, Journal of Functional Programming, Vol. 27:e4, 2017.

URL <http://dx.doi.org/10.1017/S0956796816000289>

This demo is about FuSe, a simple OCaml implementation of binary sessions that supports delegation, equi-recursive, polymorphic, context-free session types, session subtyping, and allows the OCaml compiler to perform session type checking and inference.

3.21 Concurrent TypeState-Oriented Programming

Luca Padovani (University of Turin, IT)

License © Creative Commons BY 3.0 Unported license
© Luca Padovani

Joint work of Silvia Crafa, Luca Padovani

Main reference S. Crafa, L. Padovani, “The chemical approach to typestate-oriented programming”, in Proc. of the 2015 ACM SIGPLAN Int’l Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015), pp. 917–934, ACM, 2015.

URL <http://dx.doi.org/10.1145/2814270.2814287>

This demo is about CobaltBlue, a tool for the static behavioural analysis of Objective Join Calculus scripts. The tool checks that concurrent objects and actors (modelled as terms in the Objective Join Calculus) are consistent with – and are used according to – their protocol.

3.22 Precise Subtyping

Jovanka Pantovic (University of Novi Sad, RS)

License © Creative Commons BY 3.0 Unported license
© Jovanka Pantovic

Joint work of Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, Nobuko Yoshida

Main reference M. Dezani-Ciancaglini, S. Ghilezan, S. Jaksic, J. Pantovic, N. Yoshida, “Precise subtyping for synchronous multiparty sessions”, in Proc. of PLACES 2015, pp. 29–43, 2015.

URL <http://dx.doi.org/10.4204/EPTCS.203.3>

A subtyping relation is operationally precise if both the soundness and the completeness with respect to type safety are satisfied. Soundness provides safe replacement of a term of a smaller type when a term of a bigger type is expected. If such a relation is the greatest one, we get the completeness. We discuss the notion of operational preciseness, methodology for proving the completeness and show how it works on the example of multiparty session subtyping.

3.23 Concurrent C0

Frank Pfenning (Carnegie Mellon University – Pittsburgh, US)

License © Creative Commons BY 3.0 Unported license
© Frank Pfenning

Joint work of Max Willsey, Rokhini Prabhu, Frank Pfenning

Main reference M. Willsey, R. Prabhu, F. Pfenning, “Design and Implementation of Concurrent C0”, in Proc. LINEARITY 2016, EPTCS 238, pp. 73–82, 2017.

URL <http://dx.doi.org/10.4204/EPTCS.238.8>

We give a demo of Concurrent C0, an imperative language extended with session-typed message-passing concurrency. C0 is a type-safe and memory-safe subset of C, extended with a layer of contracts, and has been used in teaching introductory programming at Carnegie Mellon University since 2010. The extension follows the Curry-Howard interpretation of intuitionistic linear sequent calculus, adapted to the linear setting. Considerable attention has been paid to programmer-friendly features such as good error messages from the lexer, parser, and (linear) type-checker. Access to Concurrent C0 can be obtained from the author. The live-coded demo of a concurrent append function is available in the additional materials.

3.24 Manifest Sharing with Session Types

Frank Pfenning (Carnegie Mellon University – Pittsburgh, US)

License © Creative Commons BY 3.0 Unported license
© Frank Pfenning

Joint work of Stephanie Balzer, Frank Pfenning

Main reference S. Balzer, F. Pfenning, “Manifest Sharing with Session Types”, Technical Report CMU-CS-17-106, Carnegie Mellon University, 2017.

URL <http://materials.dagstuhl.de/files/17/17051/17051.FrankPfenning.Slides.pdf>

We report on work in progress to reconcile sharing of resources in logically based session typed languages. The key idea is to decompose the exponential modality of linear logic into two adjoint modalities and then give a nonstandard operational interpretation of the shared layer. As a side effect, it seems we can faithfully interpret the (untyped) asynchronous pi-calculus, answering a question by Wadler.

3.25 Detecting Concurrency Errors of Erlang Programs via Systematic Testing

Konstantinos Sagonas (Uppsala University, SE)

License © Creative Commons BY 3.0 Unported license
© Konstantinos Sagonas

Main reference P. Abdulla, S. Aronis, B. Jonsson, K. Sagonas, “Optimal Dynamic Partial Order Reduction”, in Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 373–384, ACM, 2014. Extended and significantly revised version to appear in the Journal of the ACM.

URL <https://dx.doi.org/10.1145/2535838.2535845>

Testing and verification of concurrent programs is an important but also challenging problem. Effective techniques need to faithfully model the semantics of the language primitives and have a way to combat the combinatorial explosion of the possible different ways that threads may interleave (scheduling non-determinism). In this talk we will focus on a particular verification technique known as stateless model checking (a.k.a. systematic concurrency testing) and we will present Concuerror, a state-of-the-art tool for finding and reproducing errors in concurrent Erlang programs. Time permitting, we will briefly review the algorithms that Concuerror employs in order to examine only an optimal (but sound) subset of all interleavings.

More information about the tool can be found at <http://www.concuerror.com>.

3.26 Lightweight Session Programming in Scala

Alceste Scalas (Imperial College London, GB)

License © Creative Commons BY 3.0 Unported license
© Alceste Scalas

Joint work of Alceste Scalas, Nobuko Yoshida

Main reference A. Scalas, N. Yoshida, “Lightweight Session Programming in Scala”, in Proc. of the 30th European Conf. on Object-Oriented Programming (ECOOP 2016), LIPICs, Vol. 56, pp. 21:1–21:28, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016.

URL <http://dx.doi.org/10.4230/LIPICs.ECOOP.2016.21>

Designing, developing and maintaining concurrent applications is an error-prone and time-consuming task; most difficulties arise because compilers are usually unable to check whether the inputs/outputs performed by a program at runtime will adhere to a given protocol specification. To address this problem, we propose lightweight session programming in Scala: we leverage the native features of the Scala type system and standard library, to introduce (1) a representation of session types as Scala types, and (2) a library, called lchannels, with a convenient API for session-based programming, supporting local and distributed communication. We generalise the idea of Continuation-Passing Style Protocols (CPSPs), studying their formal relationship with session types. We illustrate how session programming can be carried over in Scala: how to formalise a communication protocol, and represent it using Scala classes and lchannels, letting the compiler help spotting protocol violations. We attest the practicality of our approach with a complex use case, and evaluate the performance of lchannels with a series of benchmarks.

3.27 Programming Protocols with Scribble and Java

Alceste Scalas (Imperial College London, GB)

License © Creative Commons BY 3.0 Unported license
© Alceste Scalas

Joint work of Raymond Hu, Alceste Scalas, Nobuko Yoshida

Main reference R. Hu, N. Yoshida, “Hybrid Session Verification Through Endpoint API Generation”, in Proc. of the Int’l Conf. on Fundamental Approaches to Software Engineering (FASE 2016), LNCS, Vol. 9633, pp. 401–418, Springer, 2016.

URL http://dx.doi.org/10.1007/978-3-662-49665-7_24

I will provide a brief introduction to Scribble – <http://www.scribble.org/>.

Scribble is both a language for defining global protocols involving multiple participants, and a tool that can verify the properties of such protocols (in particular: absence of deadlocks and orphan messages). Scribble can also automatically generate APIs that simplify the implementation of protocol-abiding programs. Its approach is based on the Multiparty Session Types framework.

During the talk I will illustrate the Scribble description of the SMTP protocol, and an SMTP client based on Scribble-generated APIs for Java.

3.28 Partial Type Equivalences for Verified Dependent Interoperability

Nicolas Tabareau (Ecole des Mines de Nantes, FR)

License © Creative Commons BY 3.0 Unported license
© Nicolas Tabareau

Joint work of Pierre-Évariste Dagand, Nicolas Tabareau, Éric Tanter

Main reference P.-E. Dagand, N. Tabareau, E. Tanter, “Partial type equivalences for verified dependent interoperability”, in Proc. of the 21st ACM SIGPLAN Int’l Conf. on Functional Programming (ICFP 2016), pp. 298–310, ACM, 2016.

URL <http://dx.doi.org/10.1145/2951913.2951933>

Full-spectrum dependent types promise to enable the development of correct-by-construction software. However, even certified software needs to interact with simply-typed or untyped programs, be it to perform system calls, or to use legacy libraries. Trading static guarantees for runtime checks, the dependent interoperability framework provides a mechanism by which simply-typed values can safely be coerced to dependent types and, conversely, dependently-typed programs can defensively be exported to a simply-typed application. In this paper, we give a semantic account of dependent interoperability. Our presentation relies on and is guided by a pervading notion of type equivalence, whose importance has been emphasized in recent works on homotopy type theory. Specifically, we develop the notion of partial type equivalences as a key foundation for dependent interoperability. Our framework is developed in Coq; it is thus constructive and verified in the strictest sense of the terms. Using our library, users can specify domain-specific partial equivalences between data structures. Our library then takes care of the (sometimes, heavy) lifting that leads to interoperable programs. It thus becomes possible, as we shall illustrate, to internalize and hand-tune the extraction of dependently-typed programs to interoperable OCaml programs within Coq itself.

3.29 Gradual Session Types

Peter Thiemann (Universität Freiburg, DE)

License © Creative Commons BY 3.0 Unported license
© Peter Thiemann

URL <http://materials.dagstuhl.de/files/17/17051/17051.PeterThiemann.Slides.pdf>

Session types describe structured communication on heterogeneously typed channels at a high level. They lift many of the safety claims that come with sound type systems to operations on communication channels.

The use of session types requires a fairly rich type discipline including linear types in the host language. However, web-based applications and micro services are often written on purpose in a mix of languages, with very different type disciplines in the spectrum between static and dynamic typing.

Effective use of session typing in this setting requires a mix of static and dynamic type checking. Gradual session types address this mixed setting by providing a framework which grants seamless transition between statically typed handling of sessions and any required degree of dynamic typing.

We propose GradualGV as an extension of the functional session type system GV with dynamic types and casts. We use AGT as a guideline to obtain a consistent static semantics which conservatively extends GV. We demonstrate type and communication safety as well as blame safety, thus extending previous results to functional languages with session-based communication. Our system differs from previous gradually typed systems in two respects: the interplay of linearity and dynamic types as well as the necessity to deal with changing type state requires a novel approach to specifying the dynamics of the language.

3.30 Choreographies, Modularly: Components for Communication Centred Programming

Hugo Torres Vieira (IMT – Lucca, IT)

License © Creative Commons BY 3.0 Unported license
© Hugo Torres Vieira

Joint work of Marco Carbone, Fabrizio Montesi, Hugo Torres Vieira

As communicating systems are becoming evermore complex it is crucial to conceive programming abstractions that support modularity in the development of communicating systems. In this talk we present a new model for the modular development of component-based software, following the reactive style, i.e., computations in a component are triggered by the availability of new data. The key novelty is the mechanism for composing components, which is based on multiparty protocols given as choreographies. We show how our model can be compiled to a fully-distributed implementation by translating our terms into a process calculus, and present a type system for ensuring communication safety, deadlock-freedom, and liveness.

3.31 From Communicating Machines to Graphical Choreographies

Emilio Tuosto (University of Leicester, GB)

License © Creative Commons BY 3.0 Unported license
© Emilio Tuosto

I will showcase ChorGram (https://bitbucket.org/emlio_tuosto/chorgram/wiki/Home), a tool for the reconstruction of choreographies from systems consisting of a class of communicating automata. After a very brief and lightweight introduction to the underlying theory, I will demonstrate how ChorGram can help in designing and analyse communication-centric applications.

3.32 Fencing off Go

Nobuko Yoshida (Imperial College London, GB)

License © Creative Commons BY 3.0 Unported license
© Nobuko Yoshida
URL <http://materials.dagstuhl.de/files/17/17051/17051.NobukoYoshida.Preprint.pdf>

Go is a production-level statically typed programming language whose design features explicit message-passing primitives and lightweight threads, enabling (and encouraging) programmers to develop concurrent systems where components interact through communication more so than by lock-based shared memory concurrency. Go can only detect global deadlocks at runtime, but provides no compile-time protection against all too common communication mis-matches or partial deadlocks. This work develops a static verification framework for liveness and safety in Go programs, able to detect communication errors and partial deadlocks in a general class of realistic concurrent programs, including those with dynamic channel creation, unbounded thread creation and recursion. Our approach infers from a Go program a faithful representation of its communication patterns as a behavioural type. By checking a syntactic restriction on channel usage, dubbed fencing, we ensure that programs are made up of finitely many different communication patterns that may be repeated infinitely many times. This restriction allows us to implement a decision procedure for liveness and safety in types which in turn statically ensures liveness and safety in Go programs. We have implemented a type inference and decision procedures in a tool-chain and tested it against publicly available Go programs.

3.33 Undecidability of Asynchronous Session Subtyping

Nobuko Yoshida (Imperial College London, GB)

License © Creative Commons BY 3.0 Unported license
© Nobuko Yoshida
URL <http://materials.dagstuhl.de/files/17/17051/17051.NobukoYoshida1.Preprint.pdf>

Asynchronous session subtyping has been studied extensively and applied in the literature. An open question was whether this subtyping relation is decidable. This paper settles the question in the negative. To prove this result, we first introduce a new sub-class of two-party communicating finite-state machines (CFSMs), called asynchronous duplex (ADs), which we show to be Turing complete. Secondly, we give a compatibility relation over CFSMs, which

is sound and complete wrt. safety for ADs, and is equivalent to the asynchronous subtyping. Then we show that checking whether two CFSMs are in the relation reduces to the halting problem for Turing machines. In addition, we show the compatibility relation to be decidable for three sub-classes of ADs.

4 Working groups

4.1 Group Discussion: Integrating Static and Dynamic Typing

Laura Bocchi (University of Kent – Canterbury, GB)

License  Creative Commons BY 3.0 Unported license
© Laura Bocchi

Joint work of Approximately half of the seminar participants.

The starting point for the discussion was that most of us have worked or are working on static typing, some of us on dynamic typing and monitoring, or even on the combination of static and dynamic verification in a network (but not in the same node), and only a few have direct experience of gradual and hybrid typing. Gradual/hybrid behavioural typing is quite a new thread.

The motivation is that run-time monitoring is critical in several contexts (e.g. when addressing security issues, in untrusted networks, in real-time scenarios where it is harder to make precise predictions). Run-time mechanisms provide programmers with better access to the current state of objects, which is often unclear at compile-time. Usability is also a motivation.

What does it mean to monitor? The notion of monitors is strictly related to the notion of contract. Monitors are contracts, which are used to check interactions. There are two main aspects of monitoring: verification (check behaviour and determine blame) and enforcement (e.g. suppress bad messages).

What does it mean to “go right and wrong”? There is a need for systematic construction of, and reasoning about, monitors.

Gradual/hybrid typing require a more complex notion of correctness than the usual type safety given by static typing. Critical to this aim is the role of blame. There are several views of blame, including at least the following, and any points in between. Blame the the less precisely-typed code (e.g., when hybrid typing) [1]. Blame anybody who has violated the contract [2] Blame who originated the first contract violation (implicitly assumed in [3]).

Blame in the case of sharable resources is not obvious. Shared resources may be linked to “something” linear which makes it not obvious to assign blame. This is a problem that comes with linearity (affinity would be ok).

There is a general interest in a mathematical model of blame.

Blame is also useful as it introduces a “social process” in the sense that it makes programmers want to work hard to satisfy their contracts (assuming that contract violations throws blame on others). This may promote the use of contracts in practice.

There are some limitations of dynamic types. One critical problem is to define the boundaries between static and dynamic typing. Both static and dynamic typing have advantages and disadvantages. We focused, in our discussion, on the limitations of dynamic typing: worse performance (due to overhead), no progress guarantees, in some cases more expressive but in some other cases less expressive (e.g. when using parametricity or talking

about multiple runs, to check branching-time properties, limitations like monitorability [3] when having assertions on message content, loss of transparency in timed scenarios)

References

- 1 Philip Wadler, Robby Findler. *Well-typed programs can't be blamed*. Proceedings of ESOP, 2009.
- 2 Massimo Bartoletti, Alceste Scalas, Emilio Tuosto, Roberto Zunino. *Honesty by Typing*. Logical Methods in Computer Science 12(4), 2016.
- 3 Laura Bocchi, Tzu-chun Chen, Romain Demangeon, Kohei Honda, Nobuko Yoshida. *Monitoring Networks through Multiparty Session Types*. Proceedings of FORTE, 2013.
- 4 Limin Jia, Hannah Gommerstadt, Frank Pfenning. *Monitors and Blame Assignment for Higher-Order Session Types*. Proceedings of POPL, 2016.
- 5 Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, Matthias Felleisen. *Correct blame for contracts: no more scapegoating*. Proceedings of POPL, 2011.
- 6 Cameron Swords, Amr Sabry, Sam Tobin-Hochstadt. *Expressing Contract Monitors as Patterns of Communication*. Proceedings of ICFP, 2015.
- 7 Tim Disney, Cormac Flanagan, Jay McCarthy. *Temporal Higher-Order Contracts*. Proceedings of ICFP, 2011.

4.2 Group Discussion: Behavioural Types in Non-Communication Domains

Simon Gay (University of Glasgow, GB)

License  Creative Commons BY 3.0 Unported license
© Simon Gay

Joint work of Approximately half of the seminar participants

The discussion focused on three main topics.

What is a behavioural type?

Hans Hüttel quoted the definition “... notions of typing that are also able to describe properties associated with the behaviour of programs and in this way also describe how a computation proceeds. This often includes accounting for the notions of causality and choice.” [1] Examples include session types, type state, effect types, coeffect types, information flow, intersection types, differential types. In many systems, behavioural types evolve with the reduction of terms, whereas standard types remain the same. However, consider the functional programming style based on the GV calculus: the types don't change with reduction, but instead due to rebinding – use linearity to encode changing types. The simply-typed lambda calculus is not an example of a behavioural type system. Whilst this can be translated into communication [2, 3], simple-types within the lambda calculus are not themselves behavioural. Computations in one language can be translated into communication in another language, capturing intensional aspects of a program [2, 4, 5].

Another view is that non-behavioural types characterise the final value of the computation, whereas behavioural types describe how the computation proceeds. Simple types control termination, but are not seen as inherently behavioural. We could say that behavioural types include everything that's not a simple type. Logical relations give meaning to types, and can have computational content, e.g. due to effects (trace properties).

In what areas, other than communication, do type-state like constraints occur?

It is interesting to infer type-state specifications for a given API to avoid some undesired behaviour, e.g. to avoid dereferencing a null, or to infer sequencing constraints. Jonathan Aldrich’s group have done empirical work on the occurrence of type-state in the wild [6]. It relates strongly to notions of identity and state but a linear discipline allows it to be decomposed in a copying semantics (e.g. in a pure functional setting). Another empirical study of programming protocols is in Joshua Sunshine’s thesis, Chapter 3: “Quantitative study of API protocol usability”.

Several people gave examples.

Hugo López: cyberphysical systems have control events where the connection between events are unknown (cf. shared memory concurrency) and timing plays a part. In message-passing concurrency the links between events are much more clear. This is related to [7].

Keigo Imai: Type-state example: in a smartphone there is a lot of context switching, involving serialising state, on a low-memory device. Applications are in various active/inactive states, and this changes the user’s capabilities to interact with each.

Francisco Martins: Related example: different hardware components get turned on and off or have different capabilities for the purposes of battery saving. Programmers could be forced to follow the protocols such that a resource’s handles are closed and battery is saved.

Garrett Morris: L4 microkernel off-loads responsibility for memory to user programs. Programs have to ask kernel to subdivide their heap allocation for subthreads, which then they give up some capability. Can’t DDOS the kernel by asking for lots of thread control blocks, because now these are within the purview of the programs. Have to manage the capabilities yourself. The state of the capabilities is a key part of the kernel/program interaction; the server can refuse requests that violate previous capability assignments. Microkernel design could benefit from type state definition.

Dimitrios Kouzapas: Data processing, and private data, e.g. camera photographs and recognises number plate, if car is speeding the data is kept, if not the data is dropped. This is a protocol on the data and relates to provenance.

Thomas Hildebrandt: There are legal frameworks for the behaviour of how our data is processed. We want to mediate between the contracts.

Giovanni Bernardi: The idea of effects generalises the idea of communication, and mathematically this works out as a model of some kind of modality.

We considered why behavioural and linear types became linked in the first place. There are behavioural specifications which don’t need linearity. Linearity gives us a way to encode the state changes. But is this too much? Behavioural types could help people who write concurrent data structures. Does this provide a way to explain where locking is and isn’t needed?

Areas for future research

- Consider behavioural types for concurrent data structures and algorithms.
- Perhaps we could weaken certain assumptions about shared channels so that protocols of interaction are more easily distributed and non-binary interactions are expressible, on shared channels, which may or may not imply locking/mutual exclusion.
- Look at the work of Beckman et al. [6] to give us a source of case studies for type-state systems and see what features can be captured by our current tools.

- Re-examine the assumption that linearity is necessary, or at least re-examine its realisation, in the light of work such as Frank Pfenning’s talk during the seminar.
- Review work on formal theories about the interaction of different behaviours, e.g., communication and hardware schedules in FPGA systems [8], probabilities and exceptions [9], differential types and state.

References

- 1 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniérou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira and Gianluigi Zavattaro. *Foundations of Session Types and Behavioural Contracts*. ACM Computing Surveys 49(1) 3:1–3:36, 2016.
- 2 Robin Milner. *Functions as processes*. Proceedings of ICALP, 1990.
- 3 Bernardo Toninho, Luís Caires and Frank Pfenning. *Functions as session-typed processes*. Proceedings of FOSSACS, 2012.
- 4 Dominic Orchard and Nobuko Yoshida. *Effects as sessions, sessions as effects*. Proceedings of POPL, 2016.
- 5 Cameron Swords, Amr Sabry and Sam Tobin-Hochstadt. *Expressing Contract Monitors as Patterns of Communication*. Proceedings of ICFP, 2015.
- 6 Nels Beckman, Duri Kim and Jonathan Aldrich. *An Empirical Study of Object Protocols in the Wild*. Proceedings of ECOOP, 2011.
- 7 Tim Disney, Cormac Flanagan, Jay McCarthy. *Temporal Higher-Order Contracts*. Proceedings of ICFP, 2011.
- 8 Xinyu Niu, Nicholas Ng, Tomofumi Yuki, Shaojun Wang, Nobuko Yoshida and Wayne Luk. *EURECA compilation: Automatic optimisation of cycle-reconfigurable circuits*. Proceedings of FPL, 2016.
- 9 Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvert and Tarmo Uustalu. *Combining effects and coeffects via grading*. Proceedings of ICFP, 2016.

4.3 Group discussion: Dependent Session Types

Simon Gay (University of Glasgow, GB)

License © Creative Commons BY 3.0 Unported license
© Simon Gay

Joint work of Approximately half of the seminar participants.

Several researchers have studied dependent session types, in which the types of messages may depend on the values of previous messages. Three lines of work have been produced by different participants in the seminar.

Frank Pfenning, Luis Caires and Bernardo Toninho have included dependent types in the linear logic / Curry-Howard approach to session types, and have shown how they can encode features such as proof-irrelevance.

Conor McBride has developed a general setting for combining dependent types and linear types, by distinguishing between “consumption” and “contemplation”, that is, value-level and type-level uses of data. He has used dependent session types as an example of a specific type theory that can be developed in this setting.

Edwin Brady has embedded session types and related concepts of typestate in his general-purpose dependently-typed programming language, Idris.

The early part of the discussion focused on understanding the relationships between these three approaches. Conor McBride emphasised the need to be clear about what it is that a

session type can depend on: in his view it is the traffic on a channel, and this is consistent with the other approaches; note that dependence on the identity of a channel would be a different concept.

There was some discussion about the possibility of dependence on the behaviour of a process. Dependence on traffic is one aspect of this idea, but there could be others. It leads to the need to define equivalence between processes.

Towards the end of the discussion, areas for further research were identified.

- More detailed comparisons between the different approaches to dependent session types.
- Further study of process equivalence.
- The relationship between intuitionistic and classical formulations of the logical foundation of session types.
- Understanding the possibility of dependence on channel identity.

4.4 Group Discussion: Future Activities and Funding Possibilities

Simon Gay (University of Glasgow, GB)

License  Creative Commons BY 3.0 Unported license
© Simon Gay

Joint work of Approximately half of the seminar participants.

The background to this discussion is that most of the participants in the seminar were involved in COST Action IC1201 (BETTY: Behavioural Types for Reliable Large-Scale Software Systems), which ran for four years from October 2012 to October 2016. The seminar included participants from outside the BETTY group, in order to bring in relevant ideas from broader research topics. The end of the COST Action naturally prompted discussion about future activities for the community, and future funding for research on behavioural types. These two points are closely linked.

First we discussed future activities, independently of funding. We agreed that another Dagstuhl seminar would be worthwhile, with an expanded or different combination of people from related topics. Concurrent Separation Logic was mentioned as a relevant topic. There were several suggestions for different ways of organising a Dagstuhl seminar, especially in relation to the choice of discussion topics and the possibility of specifying discussion topics before the beginning of the seminar. We also noted the possibility of a more focussed meeting on a topic such as programming language design. Dagstuhl is not the only possible location for a similar seminar: we could consider the Shonan centre in Japan, or the Banff centre in Canada. However, Dagstuhl is the most convenient and we agreed to propose another seminar. The organisers of the present seminar said that they would be willing to organise another one.

Discussion moved on to the question of funding. Hans Huttel spoke about Horizon 2020 calls, noting that our community had applied unsuccessfully in recent calls. This led to a discussion about whether we had chosen the right calls, and then the higher-level question of how the topics of the calls are defined. Antonio Ravara argued that we have been too passive, and that we should try to get leaders of our community onto the committees that define the funding calls. Failing that, we should try to influence people who are on the committees. This requires long-term strategy and it's too late for Horizon 2020, but we need to start thinking about the next cycle of research funding. An immediate action, which we agreed on, is to redevelop the BETTY website in order to raise the profile of the research area and community.

There was some discussion about specific areas, which have the possibility of funding, in which to try to apply behavioural types. Some members of our community had applied to Internet of Things calls, without success. Ivan Lanese noted that if we want to introduce behavioural types into IoT applications, then we need to work with people who have more practical experience with IoT; one way to start would be to invite such people to the next Dagstuhl seminar. Giovanni Bernardi mentioned a specific high-profile systems researcher at his institution, who might be a useful contact.

The final topic of discussion focussed on national funding schemes. We should all apply for national projects, and perhaps it would be possible to submit coordinated applications in more than one country to support a collaborative project. Connecting with the earlier discussion about adding session types to mainstream languages, people could apply for national projects to do that. We could also try to systematically take advantage of schemes for visiting professors and researchers, in order to arrange visits within the community. Another possibility is to explore industrial funding, such as Google's faculty grants; it was also mentioned that Mozilla are funding PhD students at Northeastern University in the USA. Finally, we in the community could support each other by sharing successful funding proposals and by providing support for individual fellowship applications from early-career researchers.

4.5 Group Discussion: Session Sharing and Races

Simon Gay (University of Glasgow, GB)

License © Creative Commons BY 3.0 Unported license
© Simon Gay

Joint work of Approximately half of the seminar participants.


There was a discussion about the problems posed by allowing sharing and races within session type systems, and various approaches to controlling these generalisations of the classical session type systems. The group produced a list of relevant references.

References

- 1 Damiano Mazza. *The true concurrency of differential interaction nets*. Mathematical Structures in Computer Science, 2016.
- 2 Stephen Brookes, Peter O'Hearn. *Concurrent Separation Logic*. ACM SIGLOG News, 2016.
- 3 Ilya Sergey. *Concurrent Separation Logic family tree*. <http://ilyasergey.net/other/CSL-Family-Tree.pdf>
- 4 Ralf Jung et al. *Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning*. Proceedings of POPL, 2015.
- 5 Tzu-chun Chen, Kohei Honda. *Specifying stateful asynchronous properties for distributed programs*. Proceedings of CONCUR, 2012.
- 6 Lindsey Kuper, Ryan Newton. *LVars: lattice-based data structures for deterministic parallelism*. Proceedings of FHPC, 2013.
- 7 Filipe Militão, Jonathan Aldrich, Luís Caires. *Composing Interfering Abstract Protocols*. Proceedings of ECOOP, 2016
- 8 Filipe Militão, Jonathan Aldrich, Luís Caires. *Rely-Guarantee Protocols*. Proceedings of ECOOP, 2014.
- 9 Luís Caires, João Costa Seco. *The type discipline of behavioural separation*. Proceedings of POPL, 2013.

4.6 Group Discussion: Standardisation of a Programming Language with Session Types

Simon Gay (University of Glasgow, GB)

License  Creative Commons BY 3.0 Unported license
© Simon Gay

Joint work of Most of the seminar participants.

At dinner the previous evening, there was a discussion between the seminar organisers (Phil Wadler, Nobuko Yoshida, Simon Gay, Vasco Vasconcelos) together with Mariangiola Dezani and Frank Pfenning. We agreed to suggest a project to integrate session types into Haskell, OCaml, Rust and Scala. This would include working with the language developers to add support for linear types in order to avoid the need for all the coding tricks we have seen in Haskell, OCaml and Scala.

The general discussion involved 30 people, who between them represented most of the topics and approaches that had been presented during the seminar. Phil Wadler opened the discussion by summarising the situation that led to the development of Haskell as a standard lazy functional language: several languages were being developed by different research groups, and the community decided that it would be productive to adopt a single common language as a platform for exploring and promoting lazy functional programming.

The group agreed that there are two distinct possibilities for developing a standard language that includes session types: (1) we develop a new language, similarly to the development of Haskell; (2) we pick an existing language (or several languages) and work with its developers to put session types into it. There was also discussion about whether standardising a programming language is the right level to work at. An alternative would be to follow the Imperial College group in using Scribble as a standard language-independent formalism for describing protocols.

Each approach has advantages and disadvantages, which were discussed thoroughly. The main advantage of working with an existing language (or languages) is that they already have programmers and communities who would be able to take advantage of session types in a familiar setting. The main disadvantage is that it might not be straightforward to combine session types, especially the necessary linear typing, with a full range of existing language features such as polymorphism. This problem could be reduced by working with Rust, which already has affine types. Frank Pfenning explained that he has already made contact with Mozilla about integrating session types into Rust, although he doesn't yet have a concrete proposal for a language extension. It was also noted that extending an existing language and getting the extension into the main release would require deep involvement in that language's community.

The main advantage of developing a new language is that the design is not constrained by existing features. Frank Pfenning has developed two small languages based on session types: SILL, and Concurrent C0. The latter was demonstrated during the seminar. The main disadvantage of developing a new language is that a great deal of engineering work is required to produce a usable full-spectrum language. This can be reduced to some extent by building on the runtime system of an existing language, as Scala did with Java. Perhaps Erlang would be an interesting base. Conor McBride took the view, based on his experience with dependently-typed programming and the relationship between Haskell and languages such as Agda, that the aim should not be to achieve widespread adoption of a new language; instead, success consists of features being stolen by mainstream languages.

There is a question of whether session types are a sufficiently foundational feature to justify a new language design. Concurrency is a cross-cutting concern, orthogonal to the

main language paradigm, so it seems that a new language design would have to commit to one of the existing main paradigms and this decision would provoke disagreement before any work is done on session typing features. Countering this point could be the argument that session-typed concurrent programming, based on pi calculus, could be a paradigm for controlling massively parallel architectures.

It was noted that a language design from our community should have a well-specified formal semantics. Derek Dreyer (not present at the seminar) has an ERC grant to formally study the semantics and type system of Rust; we agreed that it would be useful to involve him in future meetings.

There was some discussion about the advantages and disadvantages of full type inference, with significant support for the idea that we should not aim for it. In relation to a session-typed methodology for developing distributed systems, it makes more sense to start with explicitly declared types as part of the system design. Interactive programming guided by session types has some attractions.

In the end, there was enthusiasm from around half of the people present, for the idea of developing a new language based on session types. More detailed discussion will follow in the future, and other interested people will be able to join in. It was noted that Dagstuhl has the possibility of small focussed meetings, as well as full seminars, and this could be a way of proceeding with a language design effort.

4.7 Group Discussion: Behavioural Types for Mainstream Software Development

Philipp Haller (KTH Royal Institute of Technology – Stockholm, SE)

License © Creative Commons BY 3.0 Unported license
© Philipp Haller

Joint work of Approximately half of the seminar participants.

The starting point of the discussion was the question: how to support software development using behavioral types? Practical software development requires the use of widely-used programming languages, such as Scala, OCaml, or Haskell. Several approaches to encoding session types in the type systems of these languages were presented during the seminar. Thus, a natural question to ask was whether these existing implementations are “enough”, or whether they have fundamental limitations that should be addressed in future work by the community. It was noted that current implementations already improve upon programming models used in industry; it would thus be worthwhile to create industrial-strength systems building upon the approaches of existing session type implementations.

Two principal approaches were identified to implementing session types for existing languages: the first approach encodes session types in the type system of an existing language; this approach requires the host language to have a sufficiently powerful type system. The second approach directly extends an existing language. It was mentioned that widely-used languages have developed to include features which support more direct encodings than are possible today in Haskell, OCaml, or Scala. For example, Rust has support for affine types, and there is at least one implementation of session types in Rust.

An important limitation of existing implementations of session types was identified, namely, useful and informative type error messages. Luca Padovani pointed out that in his OCaml implementation, if a developer does not implement an end point correctly, error messages are informative, and error locations are precise. However, when connecting two

end points and a type error exists, the error may occur “far away” from where the actual problem is. It was also noted that type inference may be a source of difficulties. Making types explicit typically improves type error messages, for example, in the existing Scala library implementations. Finally, Gul Agha pointed out that most informative would be error traces corresponding to session types.

A group of discussion participants identified development tools as important for the adoption of session types by practitioners. It was noted that session types appear related to UML sequence diagrams, widely used in practice. Simon Gay pointed out that the generation of code templates can guide developers during the implementation of communication protocols. Code generation could also be integrated with modern IDEs.

On a less technical level, it was noted that establishing good feedback channels between software developers and designers of languages and libraries for programming with session types is a challenge. In this context, interaction with open-source maintainers may be an effective way to receive valuable feedback. In addition, integrating session types into open-source projects could demonstrate their value to software developers.

Education and training were also identified as important for the adoption of session types by the broader software development community. It was suggested that curricula at colleges and universities helped adoption of functional programming languages like Haskell. As a possible route discussion participants suggested the collection of patterns, inspired by the Gang-of-Four book, showing how session types or linearity help address common issues in software development. These patterns could then be taught at universities.

Modularity and reuse of session types was identified as a topic requiring further research. While FSMs are often natural for expressing the communication patterns of individual processes, their complexity can easily explode in the context of several participants. Gul Agha had explained this challenge in his presentation earlier during the seminar. Both Gul Agha and Luca Padovani pointed out that in different contexts, the same concurrent object might be used with different protocols/session types. Thus, modular specifications of session types are required which separate protocols from concurrent objects.

The discussion participants identified the following future directions. First, the development of compelling use cases that mirror modern software development; these use cases must be “complex enough” to showcase the power of session types. Second, the development of design patterns in the style of the Gang-of-Four book. Third, work on suitable abstractions, beyond state machines, that are provided to developers. Fourth, the development of a systematic approach to evaluate session types in the context of professional software development (taking open source software into account). Fifth, exchanges between industry and academia, and collaboration with industrial research labs. Finally, development of suitable concepts and curricula for teaching and education.

4.8 Group Discussion: Educational Resources for Behavioural Types

Hugo Torres Vieira (IMT – Lucca, IT)

License  Creative Commons BY 3.0 Unported license
© Hugo Torres Vieira

Joint work of Approximately half of the seminar participants.

The discussion started with a short description of past experiences of teaching courses related to behavioural types.

At Imperial College London a course on concurrent programming uses LTSA (<https://www.doc.ic.ac.uk/ltsa/>) for the verification of systems modelled as a set of interacting finite state machines, and Java for the actual implementation. A gap between the high-level modelling and the Java implementations was identified as an issue for the students' learning experience, both conceptually and at the level of tool support.

At the University of Leicester more than one course on concurrent programming was mentioned, an optional module based on Java (offered to 3rd year BSc and MSc students) and notably an MSc course (core to several MSc degrees) that uses choreographies for system specification. CFSMs are used to model the global interaction scenario which are then used in a top-down style to obtain the local implementations. Students reacted positively to the inclusion of some encompassing theory in the latest edition, and tool support with ChorGram is a goal for the next one.

At CMU some courses related to behavioural types were mentioned, most of which using C0 <http://c0.typesafety.net>. In particular, the introductory course on imperative programming uses behavioural type like specifications to provide contracts written in the language itself. Some other courses that address data structures in a concurrency setting were also mentioned.

At IMT Lucca a PhD level module on type-based verification was mentioned, where behavioural types were the topic of the last few lectures.

At the University of British Columbia, at the EPFL, and at the University of Strathclyde the reported experiences with courses related to type-based verification mostly concerned sequential languages, and the relationship with behavioural types is a topic of interest for further developments.

After the report on past and ongoing experiences, some desirable future goals for our community were discussed.

Obtaining language and tool support for reducing the gap between specifications and implementations.

Creating a repository for the exchange of existing solutions to example scenarios in the existing approaches (the repository created by the ABCD project at Glasgow University, Edinburgh University and Imperial College London was mentioned as an existing resource).

Creating a repository with related documentation of courses taught at various places to serve as a reference for complementary courses. For instance, courses on designing/modelling based on behavioural types may refer to courses/material on programming based on behavioural types


Establishing a common agreement on the principles to be taught in courses related to behavioural types so that training can lead to the desired effect of creating a community of programmers with the specialized know-how (a mention to Benjamin Pierce's Software Foundations course was made, as a reference for future courses).

In the discussion it was noted that tool support in a module requires to spend time on the usability of tools. However, this time can be compensated if some tools offer some automatic marking features.

5 Open problems

5.1 A Meta Theory for Testing Equivalences

Giovanni Tito Bernardi (University Paris-Diderot, FR)

License  Creative Commons BY 3.0 Unported license
© Giovanni Tito Bernardi

Main reference Giovanni Bernardi, Matthew Hennessy, “Mutually Testing Processes”, Logical Methods in Computer Science, 11(2:1), 2015.

URL [http://dx.doi.org/10.2168/LMCS-11\(2:1\)2015](http://dx.doi.org/10.2168/LMCS-11(2:1)2015)

Testing equivalences are an alternative to bisimulation equivalence that provide in a natural way semantic models for session types. In this talk we will recall the chief ideas behind testing equivalences, along with part of the state of the art. We will also present an open problem, hopefully spurring discussion.

Participants

- Gul Agha
University of Illinois –
Urbana-Champaign, US
- Nada Amin
EPFL – Lausanne, CH
- Robert Atkey
University of Strathclyde –
Glasgow, GB
- Giovanni Tito Bernardi
University Paris-Diderot, FR
- Laura Bocchi
University of Kent –
Canterbury, GB
- Edwin Brady
University of St. Andrews, GB
- Luis Caires
New University of Lisbon, PT
- Marco Carbone
IT University of
Copenhagen, DK
- Ilaria Castellani
INRIA Sophia Antipolis, FR
- Tzu-chun Chen
TU Darmstadt, DE
- Mariangiola Dezani
University of Turin, IT
- Patrick Thomas Eugster
TU Darmstadt, DE
- Adrian Francalanza
University of Malta – Msida, MT
- Ronald Garcia
University of British Columbia –
Vancouver, CA
- Simon Gay
University of Glasgow, GB
- Philipp Haller
KTH Royal Institute of
Technology – Stockholm, SE
- Thomas Hildebrandt
IT University of
Copenhagen, DK
- Hans Hüttel
Aalborg University, DK
- Keigo Imai
Gifu University, JP
- Dimitrios Kouzapas
University of Glasgow, GB
- Roland Kuhn
Actyx AG – München, DE
- Ivan Lanese
University of Bologna, IT
- Hugo-Andrés López
Technical University of Denmark
– Lyngby, DK
- Francisco Martins
University of Lisbon, PT
- Conor McBride
University of Strathclyde –
Glasgow, GB
- Hernán Melgratti
University of Buenos Aires, AR
- Fabrizio Montesi
University of Southern Denmark –
Odense, DK
- J. Garrett Morris
University of Edinburgh, GB
- Nicholas Ng
Imperial College London, GB
- Dominic Orchard
University of Kent –
Canterbury, GB
- Luca Padovani
University of Turin, IT
- Jovanka Pantovic
University of Novi Sad, RS
- Frank Pfenning
Carnegie Mellon University –
Pittsburgh, US
- Antonio Ravara
Universidade Nova de Lisboa, PT
- Konstantinos Sagonas
Uppsala University, SE
- Alceste Scalas
Imperial College London, GB
- Nicolas Tabareau
Ecole des Mines de Nantes, FR
- Peter Thiemann
Universität Freiburg, DE
- Hugo Torres Vieira
IMT – Lucca, IT
- Emilio Tuosto
University of Leicester, GB
- Vasco T. Vasconcelos
University of Lisbon, PT
- Philip Wadler
University of Edinburgh, GB
- Nobuko Yoshida
Imperial College London, GB
- Shoji Yuen
Nagoya University, JP

