Behavioral Types in Programming Languages

Davide Ancona
Viviana Bono
Mario Bravetti
Joana Campos
Giuseppe Castagna
Pierre-Malo Deniélov
Simon J. Gay
Nils Gesbert
Elena Giachino
Raymond Hu
Einar Broch Johnsen
Francisco Martins
Viviana Mascardi
Fabrizio Montesi
Rumyana Neykova
Nicholas Ng
Luca Padovani
Vasco T. Vasconcelos
Nobuko Yoshida
Editorial Scope

Topics

Foundations and Trends® in Programming Languages publishes survey and tutorial articles in the following topics:

- Abstract interpretation
- Compilation and interpretation techniques
- Domain specific languages
- Formal semantics, including lambda calculi, process calculi, and process algebra
- Language paradigms
- Mechanical proof checking
- Memory management
- Partial evaluation
- Program logic
- Programming language implementation
- Programming language security
- Programming languages for concurrency
- Programming languages for parallelism
- Program synthesis
- Program transformations and optimizations
- Program verification
- Runtime techniques for programming languages
- Software model checking
- Static and dynamic program analysis
- Type theory and type systems

Information for Librarians

Foundations and Trends® in Programming Languages, 2016, Volume 3, 4 issues. ISSN paper version 2325-1107. ISSN online version 2325-1131. Also available as a combined paper and online subscription.
Behavioral Types in Programming Languages

Davide Ancona, DIBRIS, Università di Genova, Italy
Viviana Bono, Dipartimento di Informatica, Università di Torino, Italy
Mario Bravetti, Università di Bologna, Italy / INRIA, France
Joana Campos, LaSIGE, Faculdade de Ciências, Univ de Lisboa, Portugal
Giuseppe Castagna, CNRS, IRIF, Univ Paris Diderot, Sorbonne Paris Cité, France
Pierre-Malo Deniélou, Royal Holloway, University of London, UK
Simon J. Gay, School of Computing Science, University of Glasgow, UK
Nils Gesbert, Université Grenoble Alpes, France
Elena Giachino, Università di Bologna, Italy / INRIA, France
Raymond Hu, Department of Computing, Imperial College London, UK
Einar Broch Johnsen, Institutt for informatikk, Universitetet i Oslo, Norway
Francisco Martins, LaSIGE, Faculdade de Ciências, Univ de Lisboa, Portugal
Viviana Mascardi, DIBRIS, Università di Genova, Italy
Fabrizio Montesi, University of Southern Denmark
Rumyana Neykova, Department of Computing, Imperial College London, UK
Nicholas Ng, Department of Computing, Imperial College London, UK
Luca Padovani, Dipartimento di Informatica, Università di Torino, Italy
Vasco T. Vasconcelos, LaSIGE, Faculdade de Ciências, Univ de Lisboa, Portugal
Nobuko Yoshida, Department of Computing, Imperial College London, UK
Abstract

A recent trend in programming language research is to use behavioral type theory to ensure various correctness properties of large-scale, communication-intensive systems. Behavioral types encompass concepts such as interfaces, communication protocols, contracts, and choreography. The successful application of behavioral types requires a solid understanding of several practical aspects, from their representation in a concrete programming language, to their integration with other programming constructs such as methods and functions, to design and monitoring methodologies that take behaviors into account. This survey provides an overview of the state of the art of these aspects, which we summarize as the pragmatics of behavioral types.
Modern society is increasingly dependent on large-scale software systems that are distributed, collaborative and communication-centered. Correctness and reliability of such systems depend on compatibility between components and services that are newly developed or may already exist. The consequences of failure are severe, including security breaches and unavailability of essential services. Current software development technology is not well suited to producing these large-scale systems, because of the lack of high-level structuring abstractions for complex communication behavior.

A recent trend in current research is to use behavioral type theory as the basis for new foundations, programming languages, and software development methods for communication-intensive distributed systems. Behavioral type theory encompasses concepts such as interfaces, communication protocols, contracts, and choreography. Roughly speaking, a behavioral type describes a software entity, such as an object, a communication channel, or a Web Service, in terms of the sequences of operations that allow for a correct interaction among the involved entities. The precise notions of “operations” and of “correct interaction” are very much context-dependent. Typical examples of op-
operations are invoking a method on an object, connecting a client with a Web Service in a distributed system, sending a message between cores in a parallel program. The notion of correct interaction may encompass both safety properties (such as the communication of valid method arguments, the absence of communication errors, the absence of deadlocks) as well as liveness properties (such as the eventual receipt of a message or the eventual termination of an interaction).

To illustrate some paradigmatic aspects of behavioral type theory more concretely, consider the diagram in Figure 1.1 depicting the interaction between two entities, named Customer and Agency. In this diagram, the horizontal lines represent interaction events between the two entities and the vertical lines represent their temporal ordering. The $s$-labeled line at the top of the diagram denotes the establishment of a connection between the two entities and the definition of an interaction scope that is often called *session*. The identifier $s$ distinguishes this particular session from others (not depicted) in which Customer and Agency may be involved. We can think of $s$ as the name of a communication channel that is known only to Customer and Agency. The proper interaction consists of two phases: the first one, marked as “repeat” in the figure, is made of an unbound number of queries issued by a Customer who is planning a trip through a travel Agency. Each query

**Figure 1.1:** Graphical representation of the Customer-Agency protocol.
includes the journey details, abstracted as a message of type *String*, to which the Agency answers with the price of the journey, represented as a message of type *Double*. In the second phase, marked as “choice”, Customer decides whether to book one of the journeys, which it signals by sending either an *ACCEPT* message followed by the *Address* to which the physical documents related to the journey should be delivered at some *Date* estimated by Agency, or a *REJECT* message that simply terminates the interaction. Arrows in the diagram denote the direction of messages. The discontinuity in the vertical development of the protocol suggests that the sub-protocols beginning with the *ACCEPT* and *REJECT* messages are mutually exclusive, the decision being taken by Customer. Eventually, the interaction between Customer and Agency terminates and the session that connects them is closed. This is denoted by the *s*-labeled line at the bottom of the diagram. In summary, the diagram describes a communication protocol between Customer and Agency as a set of valid sequences of interactions. Making sure that some piece of code modeling either Customer or Agency adheres to this protocol is among the purposes of behavioral type systems, and the technical instrument through which this is accomplished is behavioral types.

In the setting of typed programming languages, the challenge posed by describing a channel like *s* with a type is that the *same entity* *s* is used for exchanging messages of *different types* (labels such as *ACCEPT* and *REJECT*, integers, strings, floating-point numbers, dates, etc.) at *different times* and traveling in *different directions* (both Customer and Agency send and receive messages on *s*). Therefore, it is not obvious which *unique* type should be given to a channel like *s* or, equivalently, to the functions/methods for using it. The solution adopted in conventional (*i.e.*, non-behavioral) type systems, and that is found in virtually all mainstream programming languages used today, is to declare that communication channels like *s* can be used for exchanging “raw” messages in the form of *byte arrays* or *strings*. It is up to the programmer to appropriately *marshal* data into such raw messages before transmission and correspondingly *unmarshal* raw messages into data when they reach their destination. In Java, for instance, the *InputStream* and *OutputStream* interfaces and related ones provide *read* and *write*
methods that respectively read data from a stream to a byte array and write data from a byte array to a stream. The main shortcoming of this approach is that it jeopardizes all the benefits and guarantees provided by the type system: such lax typing of channels and of the operations for using them provides no guarantee that the (un)marshalled data has the expected type, nor does it guarantee that messages flow along a channel in the direction intended by the protocol. Essentially, the approach corresponds to using *untyped* channels and establishes a border beyond which the type system of the programming language is no longer in effect. The resulting code is declared well typed by the compiler, but it may suffer from type-related errors (or other issues, such as deadlocks) at runtime.

The key idea of a behavioral type theory is to enrich the expressiveness of types so that it becomes possible to formally describe the sequences of messages (informally depicted in Figure 1.1) that are expected to be exchanged along the communication channel $s$ that connects Customer and Agency. This type can then be used by a type checker to verify that the programs implementing Customer and Agency interact in accordance with the intended communication protocol. In fact, we can imagine two different types associated with channel $s$, depending on viewpoint we take, that of the Customer or that of the Agency. If we take the first viewpoint, we can describe $s$ with a type $T$ defined as

$$T = \bigoplus \left\{ \begin{array}{ll}
\text{QUERY} & : !\text{String}.?\text{Double}.T \\
\text{ACCEPT} & : !\text{String}.?\text{Date}.\text{end} \\
\text{REJECT} & : \text{end}
\end{array} \right\}$$

where:

- The symbol $\bigoplus$ denotes a choice of possible behaviors that Customer can attain to, each choice being represented by a symbolic label. In this case, the possible behaviors for Customer are querying the Agency (label QUERY), accepting an offer from the Agency (label ACCEPT), or quitting the interaction (label REJECT).

- The punctuation marks ! and ? respectively prefix the type of messages sent (String) and received (Double and Date) by Cus-
customer. With these annotations, we can specify the intended direction of messages.

- The punctuation marks : and . represent the sequentiality of actions described by the type. In this case, a Customer that queries an Agency must first send a message of type String and then wait for a message of type Double. With these annotations, we can specify how the capabilities of the channel change as the channel is used for input/output operations.

- The occurrence of $T$ on the right hand side of the equation indicates that $T$ is a recursive type, therefore allowing for an unbounded number of queries from Customer to Agency. This makes it possible to specify recursive protocols.

- end marks the points in which the interaction between Customer and Agency terminates and no more messages are supposed to be exchanged.

If we take the Agency viewpoint, it is reasonable to expect that the type of $s$ should express complementary behaviors: the Agency offers choices when Customer selects one, the Agency receives a message when the Customer sends one, and vice versa. Customer and Agency should also agree on the moments in which the interaction terminates. This relation between the behaviors of Customer and Agency can be formalized as a notion of duality between the two types of $s$. In particular, the dual of $T$ is the type $S$ defined as

$$S = \sum \left\{ \begin{array}{l}
\text{QUERY} : ?\text{String}.!\text{Double}.S \\
\text{ACCEPT} : ?\text{String}.!\text{Date}.\text{end} \\
\text{REJECT} : \text{end}
\end{array} \right\}$$

obtained from $T$ by swapping choices $\oplus$ with offers $\sum$, inputs $?$ with outputs $!$, and leaving end unchanged. Now, checking that Customer and Agency use the respective ends of $s$ according to $T$ and $S$ makes sure that choices and offers match and messages of the right type are exchanged at the right time. In summary, that Customer and Agency interact correctly.
The successful application of behavioral types to the development of reliable, large-scale software requires both the study of formal type theories but also understanding and addressing more practical aspects, including the representation of behavioral types such as \( T \) and \( S \) in a concrete programming language, the integration of behavioral type checking with other programming constructs like methods and functions, and also design methodologies that take behaviors into account. The aim of this survey is to provide a first comprehensive overview of the state of the art of these aspects, which we may summarize as the *pragmatics* of behavioral types. The survey is structured as a series of chapters, each covering a particular programming paradigm or methodology. Below is an account of the content of each chapter:

- Chapter 2 is devoted to the integration of behavioral types into Object-Oriented languages. Object-oriented languages are relevant for their widespread adoption in the current development of software, for the wealth and popularity of tools that are available, and because objects nicely fit a distribution model to which behavioral types can be applied naturally. The integration can be achieved in different ways: either by *enriching* the languages with constructs (in particular, sessions) that call for a corresponding extension at the type level, or by *amalgamating* sessions and objects to the point that the objects themselves become the entities for which a behavioral description is required, for example to specify the order in which methods must/can be invoked. We also survey a parallel, but related line of research concerning *typestate*. This concept, originally introduced for discriminating the state of imperative variables (uninitialized, initialized, finalized), finds a natural application to describing object protocols and has been recently converging to behavioral typing.

- Chapter 3 explores the integration of behavioral types within functional languages. Functional languages are relevant for their qualities of being easily endowed with high-level type-theoretic and concurrent extensions, for their natural support to parallelism, and since they permit rapid prototyping. We survey three different approaches, one akin to an effect system, one based on
explicit continuation passing, and one based on monads. Besides providing an out-of-the-box application of behavioral types to a concrete programming language, the continuation-based and monadic approaches can take advantage of the type inference engine of the language so that the programmer is not required to explicitly write (or annotate programs with) behavioral types, which can be automatically reconstructed from the source code of the program.

• High-performance computing often relies on parallel processes that synchronize by means of message passing. Chapter 4 describes the use of behavioral types in conjunction with Message Passing Interface (MPI) which is the de facto standard API for high-performance computing. Also in this case, behavioral types provide an effective means for making sure that communications occur without errors. We survey three alternative approaches making use of behavioral types in this context: one based on higher-level structuring abstractions, one based on source code verification, and one based on source code generation.

• Chapter 5 describes an application of behavioral types to multi-agent systems. The latter have been proved to be an industrial-strength technology for integrating and coordinating autonomous and heterogeneous entities in open systems. In this setting, the possibility of formally describing interaction protocols in the form of behavioral types enables forms of runtime monitoring for multi-agent systems.

• Chapter 6 provides an overview of the use of behavioral types in Singularity OS, a prototype Operating System developed by Microsoft that adopts communication as the fundamental and only synchronization mechanism between processes. Sing#, the programming language used for the implementation of Singularity OS, is an extension of C# that comprises both object-oriented and functional constructs and provides a native notion of channel contract, a form of behavioral type. The formal investigation of behavioral types in this setting has led to the discovery of un-
forseen system configurations that yield memory leaks and to the
development of refined behavioral type theories preventing them.

- The WSDL and UDDI standards are technologies currently en-
  abling the description of Web Service interfaces and the creation
  of Web Service repositories. Chapter 7 explores the potential of
  behavioral types, intended as abstract descriptions of Web Ser-
  vice behaviors, as natural generalizations of WSDL interfaces to
  realize sophisticated forms discovering, composition, and orches-
  tration of Web Services.

- Chapter 8 illustrates the design-by-contract methodology for the
  development of possibly distributed, communicating systems. Ac-
  cording to this methodology, behavioral types are used for de-
  scribing, from a vantage point of view, the topology of the com-
  munication network, the communications that are supposed to
  occur, and in which order. Such global specifications serve mul-
  tiple purposes: they are a valuable form of abstract specification
  of the overall behavior of a distributed system; they can be projected
  for describing the local behavior of the network participants to
  allow the modular type checking of complex systems; they enable
  the generation of monitors to verify, at runtime, that the partici-
  pants of a heterogeneous distributed system behave as expected,
  even if only some or none of them have been type checked against
  their supposed or claimed behavior.

Overall, the survey provides substantial evidence that behavioral
types have sprinkled a remarkable interest in the research community
concerned with programming languages. The adoption of behavioral
types beside the academic context proceeds more slowly, but nonethe-
less there are encouraging signals. As a matter of fact, it is known
that programming languages tend to evolve slowly, especially when it
comes to the integration of sophisticated typing disciplines. In this re-
spect, approaches that rely on the encoding of behavioral types using
conventional type constructors (§3.3), that allow for the verification of
existing code (§4.2), or the type-driven generation of runtime monitors
(Chapters 5 and 8), enable developers to fill the gap between theory
Introduction

and practice of behavioral typing with little or no changes to their programming environment and development workflow. The survey also contains pointers to industrial projects in which behavioral types already play a key role: the Ocean Observatories Initiative, which aims at the realization of a planetary-scale network for the transmission of environmental data (§8.2), and the programming language Sing#, developed by Microsoft, which offers behavioral types as a native and key feature (Chapter 6). These early examples of industrial applications of behavioral types indirectly hint at their effectiveness in supporting the development of complex, large-scale systems for which correctness and reliability guarantees are of paramount importance.
References


References


References


References


References


References


References


References


OASIS. Web Services Business Process Execution Language. [http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html](http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html), 2007.


Luca Padovani. A Simple Library Implementation of Binary Sessions. Technical Report hal-01216310, Dipartimento di Informatica, Università di Torino, Italy, 2015. Available at [https://hal.archives-ouvertes.fr/hal-01216310](https://hal.archives-ouvertes.fr/hal-01216310).


References


