Effpi: Verified Message-Passing Programs in Dotty

Alceste Scalas
Imperial College London
and Aston University, Birmingham
UK
a.scalas@aston.ac.uk

Nobuko Yoshida
Imperial College London
UK
n.yoshida@imperial.ac.uk

Elias Benussi
Imperial College London
and Faculty Science Ltd.
UK
elias@faculty.ai

Abstract

We present `Effpi`: an experimental toolkit for *strongly-typed* concurrent and distributed programming in Dotty, with verification capabilities based on *type-level model checking*.

`Effpi` addresses a main challenge in creating and maintaining concurrent programs: errors like protocol violations, deadlocks, and livelocks are often spotted *late*, at run-time, when applications are tested or (worse) deployed. `Effpi` aims at finding them *early*, when code is written and compiled.

`Effpi` provides: *(1)* a set of Dotty classes for describing *communication protocols as types*; *(2)* an embedded DSL for concurrent programming, with process-based and actor-based abstractions; *(3)* a Dotty compiler plugin to verify whether protocols and programs enjoy desirable properties, such as deadlock-freedom; and *(4)* an efficient run-time system for executing `Effpi`'s DSL-based programs. The combination of *(1)* and *(2)* allows the Dotty compiler to check whether an `Effpi` program implements a desired protocol/type; and this, together with *(3)*, means that many typical concurrent programming errors are found and ruled out *at compile-time*. Further, *(4)* allows to run highly concurrent `Effpi` programs with millions of interacting processes/actors, by scheduling them on a limited number of CPU cores.

In this paper, we give an overview of `Effpi`, illustrate its design and main features, and discuss its future.

CCS Concepts • **Theory of computation** → *Type structures*; *Verification by model checking*; • **Software and its engineering** → *Concurrent programming languages.*

Keywords behavioural types, dependent types, processes, actors, Dotty, Scala, temporal logic, model checking

ACM Reference Format:
Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Effpi: Verified Message-Passing Programs in Dotty. In *Tenth ACM SIGPLAN Scala Symposium (Scala '19), July 17, 2019, London, United Kingdom.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3337932.3338812

1 Introduction

Concurrent and distributed programming is hard. Modern programming languages and toolkits provide high-level concurrency abstractions (such as processes and actors) to simplify reasoning, and make software developers' life easier: e.g., Erlang [9], Go [11], Orleans [23], Akka [20]. Recent developments leverage types to rule out (some) concurrency errors early, at compile-time. E.g., the Akka Typed toolkit [21] introduces *typed mailboxes and actor references* (reminiscent of [13]): an actor reference `r` of type `ActorRef[Int]` points to an actor handling messages of type `Int`, and the Scala compiler raises an error if a program tries to use `r` to send, e.g., a `String`. Typed actor references allow to approximate *protocols* [17], i.e., sequences of message exchanges; this prompted experiments on checking *sessions* at compile-time [15], with informal inspiration from *session types* [1, 14].

`Effpi` is our contribution to this line of work: an experimental, formally-grounded toolkit allowing to define *protocols as types*, with verification capabilities based on a combination of type checking plus *type-level model checking*. The theory behind `Effpi` is illustrated in [32]. Its website is:

https://alcestes.github.io/effpi

It includes `Effpi`'s source code, instructions, and a ready-to-use virtual machine. In this paper, we provide an example-driven overview, and discuss future research directions.

2 Fundamentals

Unlike the toolkits cited in §1, `Effpi` is designed on a formal foundation: $\lambda^{\pi}_{\leqslant}$, a functional concurrent message-passing calculus blending *behavioural types* (from π-calculus literature) [1, 26] and *dependent function types* (from Dotty) [4]. This theory, its related work, and some implementation details are presented in [32]; here we give an informal summary.

Behavioural Types In π-calculus literature, types are dubbed *behavioural* if they describe the interactions of a program, i.e., its *protocol*: a type like "$?int; !string$" means *"receive an integer; then, send a string."* Behavioural type systems ensure that, if program P type-checks vs. type/protocol T, then running P will yield the interactions in T; if P does not abide by

T, type-checking fails. To model programs interacting via multiple *channels*, one can use finer behavioural types, e.g.:

$$c_1?\mathit{int};\ c_2!\mathit{string} \tag{1}$$

which means *"receive an integer from channel c_1; then, send a string over channel c_2."* Many works try to bridge the gap from π-calculus theory to practice, by creating new programming languages, or seeking ways to represent types like (1) in general-purpose languages. This is not easy: some properties (e.g., static linearity checks) are tricky, and often lost in the translation to existing languages. For a survey, see [10]; works on Scala [27–30] resort to run-time linearity checks.

Behavioural Types in Dotty `Effpi` provides types for describing the desired behaviour of concurrent programs:

- `Chan[A]` is the type of a channel that can be used to send/receive values of type `A`;
- `Out[A, B]` is the type of a program that uses a channel of type `A` to send a value of type `B`;
- `In[A, B, C]` is the type of a program that uses a channel of type `A` to receive a value of type `B`, and pass it to a continuation of type `C` (which is a function type taking `B`);
- `A >>: B` is the type of a program that performs the communications of `A`, followed by those of `B`;
- `Par[A, B]` is the type of a program that executes two subprograms of type `A` and `B` in parallel, letting them interact;
- `Rec[X, A]` is the type of a program that executes a subprogram of type `A`, possibly looping;
- `Loop[X]` is the type of a program that loops;[1]
- `Proc` is the abstract supertype of all types above (except `Chan`): it represents a program that may interact (or not).

These types become quite powerful when combined with one of Dotty's distinguishing features: *dependent function types* [4]. In fact, `Effpi` builds upon a novel, fundamental insight: dependent function types can be used to *track channel usage in programs*. E.g., the type of a function taking two channels `c1` and `c2`, and using them according to (1), is:

```scala
type T = (c1: Chan[Int], c2: Chan[String]) =>
    In[ c1.type, Int, (x: Int) => Out[c2.type, String] ]
```

To produce programs with such a type, `Effpi` provides a DSL that looks like the following code snippet:

```scala
1  val f: T = (c1: Chan[Int], c2: Chan[String]) => {
2    receive(c1) { x =>         // Use c1 to receive x
3      println(s"Received: ${x}")
4      if (x > 42) send(c2, "OK") // Send "OK" via c2
5      else send(c2, "KO")        // Send "KO" via c2
6  } }
```

The key intuition is that `Effpi`'s DSL provides methods (such as `receive()` / `send()` above) to construct objects that *describe* a program performing structured sequences of inputs/outputs. E.g., `receive()` takes two arguments: a channel used to receive a value `x`, and a function that takes `x` and performs the continuation of the input; the object returned by `receive()` has type `In`. Similarly, `send()` returns an object of type `Out`. Such objects are interpreted and executed by `Effpi`'s runtime system (discussed in §4), which performs the actual input/output operations.

The `Effpi` DSL allows to write programs performing arbitrary communications; to restrict them, a programmer can add type annotations, to *statically enforce desired protocols*. E.g., the type annotation "`f: T`" (line 1 above) restricts the possible implementations of `f`, ensuring that `f` realises the protocol described by `T`: hence, `f` uses a channel of type "`c1.type`" (that is only inhabited by `f`'s argument `c1`) to receive an `Integer`, and then uses a channel of type "`c2.type`" (only inhabited by `f`'s argument `c2`) to send a `String`. Consequently, any violation of the type/protocol `T` is found at compile-time: if, e.g., the "`else`" branch on line 5 is forgotten, or `f` uses channels `c1` and `c2` in other ways, or in a different order, or tries to interact via some channel `c3` defined elsewhere, the Dotty compiler raises a type mismatch error.

Notably, several Dotty features play a crucial role in the design of `Effpi`. E.g., the union type "`|`" [7] allows to model choices in a protocol: `Out[C1, Int] | Out[C2, String]` is the type of a process that can either send an `Integer` on channel `C1`, or a `String` on `C2`. In the next sections, we show how `Effpi` takes advantage of other characteristics of Dotty.

3 A Whirlwind Tour of `Effpi`

We now give an overview of `Effpi`'s main features, proceeding by examples. First, we focus on its core (channel-based) communication model, by showing how to implement (§3.1) and verify (§3.2) a well-known concurrency problem. Then, we illustrate `Effpi`'s higher-level, actor-like API (§3.3).

3.1 Defining, Composing & Implementing Protocols

`Effpi` allows to define and compose protocols using Dotty's type aliases and parameters. E.g., consider the well-known Dijkstra's dining philosopher problem: two processes (philosophers) share two resources (forks), and want to acquire both (to eat), then release them. Philosophers eat after acquiring two forks, and drop the first only after picking the second. The goal is: let both philosophers eat, without deadlocks. A behavioural type for the desired fork behaviour is:

```scala
type Fork[ Acq <: Chan[Unit], Rel <: Chan[Unit] ] =
  Rec[ RecX, Out[Acq, Unit] >>:
             In[ Rel, Unit, (_x: Unit) => Loop[RecX] ] ]
```

i.e.: given two channel types `Acq` and `Rel`, use a channel of type `Acq` to send a message of type `Unit` (signalling that the fork is available for Acquisition), and then (`>>:`) use a channel of type `Rel` to receive a message (signalling that the fork is Released); repeat infinitely (`Rec[RecX, ...Loop[RecX]]`).

Here is an implementation of the `Fork` protocol:

[1]It requires `X` "bound" by `Rec[X, A]`, and `Loop[X]` occurring in `A`: a workaround to type recursive programs, as Dotty lacks recursive type aliases.

```
def fork(id: Int, acq: Chan[Unit],
         rel: Chan[Unit]): Fork[acq.type, rel.type] = {
  rec(RecX) {
    println(s"Fork ${id}: available")
    send(acquire, ()) >> {
      println(s"Fork ${id}: picked")
      receive(release) { _ =>
        loop(RecX) } } } }
```

The type annotation `fork(...): Fork[acq.type, rel.type]` ensures that `fork()` uses exactly its arguments `acq` and `rel`; if the `fork`'s code tries, e.g., to use `acq`/`rel` in the wrong order, then it will not compile. Similarly, we can write the behavioural type of a philosopher, whose parameters are channel types to signal when forks are `Picked`/`Dropped`:

```
type Philo[Pick1 <: Chan[Unit], Drop1 <: Chan[Unit],
           Pick2 <: Chan[Unit], Drop2 <: Chan[Unit]] =
  Rec[RecX,
    In[Pick1, Unit, (_f1: Unit) =>
      In[Pick2, Unit, (_f2: Unit) =>
        (Out[Drop1, Unit] >>: Out[Drop2, Unit]) >>: Loop[RecX] ] ] ]
```

Then, we can write a philosopher implementation, and type-annotate it, to ensure it picks/drops the forks as desired:

```
def philo(name: String, pick1: Chan[Unit], drop1: Chan[Unit],
          pick2: Chan[Unit],
          drop2: Chan[Unit]): Philo[pick1.type, drop1.type,
                                    pick2.type, drop2.type] = {
  rec(RecX) {
    println(s"${name}: picking first fork...")
    receive(pick1) { _ =>
      println(s"${name}: picking second fork...")
      receive(pick2) { _ =>
        println(s"${name}: eating, then dropping forks...")
        send(drop1, ()) >> send(drop2, ()) >> {
          println(s"${name}: Thinking...")
          loop(RecX) } } } } }
```

We can also write and implement a type describing a desired composition of philosophers and forks (below): the type annotation enforces the desired interconnection of channels.

```
type Dining[C1pick <: Chan[Unit], C1drop <: Chan[Unit],
            C2pick <: Chan[Unit], C2drop <: Chan[Unit]] =
  Par4[ Philo[C2pick, C2drop, C1pick, C1drop], Fork[C1pick, C1drop],
        Philo[C1pick, C1drop, C2pick, C2drop], Fork[C2pick, C2drop] ]

def dining(p1: Chan[Unit], d1: Chan[Unit],
           p2: Chan[Unit], d2: Chan[Unit]): Dining[p1.type, d1.type,
                                                   p2.type, d2.type] = {
  par( philo("Socrates", p2, d2, p1, d1), fork(1, p1, d1),
       philo("Aristotle", p1, d1, p2, d2), fork(2, p2, d2) ) }
```

3.2 Verifying Protocols, and Their Implementations

The `dining()` program above type-checks and compiles. But if we run it, we may get the execution below: the application deadlocks. This is a typical case of a concurrency error spotted late, at run-time, during testing (or in production). Can we find the error at compile-time? The problem here is that the type `Dining` itself is "wrong," as it does not guarantee a desired property: deadlock freedom. In general, when types/protocols are composed, and interact, they may exhibit unwanted behaviours. To avoid this issue, `Effpi` provides a compiler plugin to verify if a set of desired run-time properties hold. E.g., if we add the following annotation to `dining()` above...

```
Fork 1: available
Fork 2: available
Socrates: picking first fork...
Fork 1: picked
Aristotle: picking first fork...
Fork 2: picked
Socrates: picking second fork...
Aristotle: picking second fork...
```

```
@verify(property = "deadlock_free()") // The compile-time check fails
def dining(p1:..., d1:..., p2:..., d2:...): Dining[p1.type, d1.type,
                                                   p2.type, d2.type] = ...
```

...then, `Effpi`'s compiler plugin verifies deadlock freedom, via *type-level model checking*: it takes the type of the annotated function definition, translates it to a format supported by the mCRL2 model checker [3, 8, 12], and analyses its potential behaviours against the property in the `@verify(...)` annotation. If verification succeeds, then the implementation enjoys the property. In the example above, verification fails: deadlock freedom does *not* hold for `dining()`'s type, hence `dining()` might deadlock (and indeed, it does: see the execution above). We can fix `Dining` by letting one philosopher pick the forks in the opposite order w.r.t. the other(s), by just swapping the arguments of the first `Philo` type, i.e.:

```
type Dining2[C1pick <: Chan[Unit], C1drop <: Chan[Unit],
             C2pick <: Chan[Unit], C2drop <: Chan[Unit]] =
  Par4[ Philo[C1pick, C1drop, C2pick, C2drop], Fork[C1pick, C1drop],
        Philo[C1pick, C1drop, C2pick, C2drop], Fork[C2pick, C2drop] ]
```

And to verify whether the solution is correct, we can try:

```
@verify(property = "deadlock_free()") // The verification succeeds
def dining2(p1:..., d1:..., p2:..., d2:...): Dining2[p1.type, d1.type,
                                                     p2.type, d2.type] = ???
```

Since the verification succeeds, we know that if we replace "???" with *any* implementation that type-checks, then `dining2()` will never deadlock. One such implementations is obtained from `dining()` above, by swapping the arguments of the first `philo()`: their correct order is enforced by the type annotation `dining2(...):Dining2[...]`. Moreover, the verification result means that we can implement and deploy the program components (forks and philosophers) separately, and they will not deadlock — provided that they have types `Fork`/`Philo`, and are interconnected as per `Dining2`.

`Effpi` allows to verify more properties: some are discussed in §3.3 below; for an (incomplete) list, see [32, Fig. 7]; for an evaluation of the verification performance, see [32, Fig. 9].

3.3 Actor-Like DSL

The overview above covers the "low-level" channel-based API of `Effpi`, based on the $\lambda^{\pi}_{\leqslant}$ calculus [32]). On top of it, `Effpi` provides higher-level abstractions and extensions, aiming at a more developer-friendly API. One such extensions uses Dotty's implicit function types [5, 25] to hide a "default" input channel, yielding an actor-like DSL reminiscent of Akka Typed [21]. E.g., from [32, §1], this is an `Effpi` actor that receives payments requests, and either accepts or rejects them — reporting accepted payments to an auditor (scenario distilled from a use case for Akka Typed [16, 21]):

```
 1 @verify(property = "reactive(mb_)(aud) &&
 2                     responsive(mb_)(aud) &&
 3                     output_ev_followed(aud)(Accepted)(mb_)")
 4 def payment(aud: ActorRef[Audit[_]]): Actor[Pay, ...] =
 5   forever {
 6     read { pay: Pay =>
 7       if (pay.amount > 42000) {
 8         send(pay.replyTo, Rejected("Too high!"))
 9       } else {
10         send(aud, Audit(pay)) >>
11         send(pay.replyTo, Accepted) } } }
```

On line 4, the type annotation `Actor[Pay, ...]` says that `payment()` returns an actor accepting messages of type `Pay`, and behaving according to the (omitted) protocol specification "..." (see [32, §1] for its details). On line 6, `read` is just a disguised `receive()` (cf. §2) that awaits inputs from an implicit channel of type `Chan[Pay]`. In this case, each received message `pay` has a `replyTo` field: it is an actor reference allowing to send a response (lines 8, 11). As in Akka Typed, actor references are type-constrained: e.g., in line 1, the type of `aud` ensures that `aud` can only be used to send messages of type `Audit`. Under the hood, `aud` is just a channel of type `Chan[Audit[_]]`. This actor-like DSL is a thin layer on top of the DSL illustrated in the previous sections, and is executed by the same interpreter and runtime system.

The `Effpi` compiler plugin can verify such actor-like programs. The annotation on lines 1–3 verifies that `payment()` is always eventually ready to receive messages from its mailbox (`mb_`), will always send back a response, and will send `Accepted` whenever it outputs something on `aud`.

4 Design and Implementation

Core DSL As noted in §2, the process/channel-based API of `Effpi` is an internal embedding of the $\lambda^{\pi}_{\leqslant}$ calculus [32] in Dotty (with minimal adaptations): this leverages Dotty's type system features (e.g., dependent function types, union types), and allows for easy interoperability with libraries and toolkits running on the Java Virtual Machine. E.g., `Effpi` processes/actors can easily interoperate with Akka Typed, via "bridges" that forward messages between `Effpi` channels and Akka `ActorRef`s; the same trick lets `Effpi` processes/actors interact across a network, via Akka Remoting [19].

Actor-Like DSL The actor-like DSL discussed in §3.3 is inspired by Akka Typed [21]; in particular, we used the "payment with audit" use case [16, 21] as a reference for DSL design, trying to make the use case implementation simple and developer-friendly. Its full implementation in `Effpi` is provided as an example with `Effpi`'s source code, and uses various features and extensions not shown here (e.g., an "ask pattern" [22], or sub-actors yielding values to their creator). Such features are covered by the compile-time check of program/protocol conformance (§2, §3.1, §3.3), but are not yet supported by the verification plugin (§3.2).

Runtime System The $\lambda^{\pi}_{\leqslant}$ embedding yields a DSL where the continuations of I/O actions are functions (closures). We took advantage of this fact to implement a runtime system with a (non-preemptive) scheduler, decoupling `Effpi` processes/actors from system threads, similarly to Akka Dispatchers [18]: i.e., processes/actors are run in an interleaved fashion, unscheduled when waiting for input, and resumed when an input is available. A difference is that Akka can only interrupt actors waiting for input, whereas `Effpi`'s also interrupts output operations. From our benchmarks, `Effpi`'s performance is not too far from Akka, and supports highly concurrent programs: for measurements, see [32, Fig. 8].

5 Conclusion, Vision, and Future Work

We gave an overview of `Effpi`, a toolkit for strongly-typed message-passing programs in Dotty. `Effpi` allows to spot concurrency errors (e.g., protocol violations, deadlocks) at compile-time, with a recipe that mixes behavioural types, Dotty's dependent function types, and model checking.

Works [27–30] implement session types in Scala. `Effpi`'s types and session types are related, but have different design and capabilities (cf. [32, §6]); moreover, [27–30] resort to *run-time* linearity checks, whereas `Effpi` does not need them.

The broader goal behind `Effpi` is providing *lightweight* software verification capabilities that *(1)* can be used by programmers who are not expert in, e.g., theorem proving or model checking; and *(2)* do not require the adoption of entirely new programming languages and toolchains. We found that Dotty can help achieving this goal, thanks to its features, and to its interoperability with the JVM ecosystem.

Much future work lies ahead: some is discussed in [32, §6]. We are particularly interested in finding more ways to leverage Dotty features for behavioural verification. We believe that match types [6] can be used to represent (a limited form of) data-dependent choices: e.g., a channel allows to receive `A` or `B`, and the protocol continues as `T` in the first case, or `T'` in the second. This would allow to represent and verify more protocols, possibly covering the whole range of multiparty session types [2, 31]. `Effpi` supports mobile code (i.e., sending/receiving program thunks) [32, Example 3.4]: we will investigate distributed implementations of the feature, that may benefit from the work on Spores [24].

Acknowledgments

Work partially supported by EPSRC projects EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1.

References

[1] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, 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, and Nobuko Yoshida. 2017. Behavioral Types in Programming Languages. *Foundations and Trends*

in Programming Languages 3(2-3) (2017). https://doi.org/10.1561/2500000031

[2] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2015. A Gentle Introduction to Multiparty Asynchronous Session Types. In *Formal Methods for Multicore Programming*. https://doi.org/10.1007/978-3-319-18941-3_4

[3] Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink, and Tim A. C. Willemse. 2013. An Overview of the mCRL2 Toolset and Its Recent Advances. In *Tools and Algorithms for the Construction and Analysis of Systems*. https://doi.org/10.1007/978-3-642-36742-7_15

[4] Dotty developers. 2019. Dotty documentation: dependent function types. https://dotty.epfl.ch/docs/reference/new-types/dependent-function-types.html.

[5] Dotty developers. 2019. Dotty documentation: implicit function types. https://dotty.epfl.ch/docs/reference/new-types/implicit-function-types-spec.html.

[6] Dotty developers. 2019. Dotty documentation: match types. http://dotty.epfl.ch/docs/reference/new-types/match-types.html.

[7] Dotty developers. 2019. Dotty documentation: union types. https://dotty.epfl.ch/docs/reference/new-types/union-types.html.

[8] Technische Universiteit Eindhoven. 2019. mCRL2 website. https://mcrl2.org/.

[9] Ericsson. 2019. The Erlang/OTP Programming Language and Toolkit. http://erlang.org/.

[10] Simon Gay and António Ravara. 2017. *Behavioural Types: From Theory to Tools*. River Publishers, Series in Automation, Control and Robotics. https://doi.org/10.13052/rp-9788793519817

[11] Google. 2019. The Go Programming Language. https://golang.org/.

[12] Jan Friso Groote and Mohammad Reza Mousavi. 2014. *Modeling and Analysis of Communicating Systems*. The MIT Press.

[13] Jiansen He, Philip Wadler, and Philip W. Trinder. 2014. Typecasting actors: from Akka to TAkka. In *SCALA@ECOOP*. https://doi.org/10.1145/2637647.2637651

[14] Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR*. https://doi.org/10.1007/3-540-57208-2_35

[15] Roland Kuhn. 2017. Akka Typed Session. https://github.com/rkuhn/akka-typed-session.

[16] Roland Kuhn. 2017. Akka Typed Session: audit example. https://github.com/rkuhn/akka-typed-session/blob/master/src/test/scala/com/rolandkuhn/akka_typed_session/auditdemo/ProcessBased.scala.

[17] Lightbend, Inc. 2017. Akka Typed: Protocols. https://akka.io/blog/2017/05/12/typed-protocols.

[18] Lightbend, Inc. 2019. Akka Dispatchers documentation. https://doc.akka.io/docs/akka/2.5/dispatchers.html.

[19] Lightbend, Inc. 2019. Akka remoting documentation. https://doc.akka.io/docs/akka/2.5/remoting.html.

[20] Lightbend, Inc. 2019. The Akka toolkit and runtime. http://akka.io/.

[21] Lightbend, Inc. 2019. Akka Typed documentation. https://doc.akka.io/docs/akka/2.5/typed/index.html.

[22] Lightbend, Inc. 2019. Commonly used patterns with Akka. https://doc.akka.io/api/akka/2.5/akka/pattern/index.html.

[23] Microsoft. 2019. The Orleans Framework. https://dotnet.github.io/orleans/.

[24] Heather Miller, Philipp Haller, and Martin Odersky. 2014. Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution. In *ECOOP*. https://doi.org/10.1007/978-3-662-44202-9_13

[25] Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2017. Simplicitly: Foundations and Applications of Implicit Function Types. *Proc. ACM Program. Lang.* 2, POPL, Article 42 (2017). https://doi.org/10.1145/3158130

[26] Davide Sangiorgi and David Walker. 2001. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press.

[27] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2017.24

[28] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming (Artifact). *Dagstuhl Artifacts Series* 3, 1 (2017). https://doi.org/10.4230/DARTS.3.2.3

[29] Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2016.21

[30] Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala (Artifact). *Dagstuhl Artifacts Series* 2, 1 (2016). https://doi.org/10.4230/DARTS.2.1.11

[31] Alceste Scalas and Nobuko Yoshida. 2019. Less is More: Multiparty Session Types Revisited. *Proc. ACM Program. Lang.* 3, POPL, Article 30 (Jan. 2019). https://doi.org/10.1145/3290343

[32] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Verifying Message-Passing Programs with Dependent Behavioural Types. In *PLDI*. https://doi.org/10.1145/3314221.3322484 To appear. Preprint: http://mrg.doc.ic.ac.uk/publications/verifying-message-passing-programs-with-dependent-behavioural-types/pldi19-preprint.pdf.