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 one of the main challenges in developing and maintaining concurrent programs: many concurrency errors (like protocol violations, deadlocks, livelocks) are often spotted late, at run-time, when applications are tested or (worse) deployed in production. Effpi aims at finding such problems early, when programs are 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 provide an overview of Effpi; then, we illustrate its design and main features, and discuss its future developments.

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

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: see, e.g., Erlang [9], Go [11], Orleans [23], and Akka [20]. Recent developments foster the use of types to rule out (some) concurrency errors early, at compile-time. E.g., the Akka Typed toolkit [21] replaces the traditional, untyped Akka

actors with typed mailboxes and actor references (reminiscent of [13]): an actor reference r of type ActorRef[Int] points to an actor that handles 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 can be used to approximate protocols [17], i.e., predetermined sequences of message exchanges; this idea prompted experiments on checking sessions at compile-time [15], with informal inspiration from the theory of 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, and type-level model checking. The theoretical underpinning of Effpi is illustrated in [32]. The (temporary, soon on GitHub) home page of the toolkit is:

https://www.doc.ic.ac.uk/~ascalas/tmp/pldi19

It includes the source code, some instructions, and a ready-to-use virtual machine. In this paper, we provide an example-driven overview of the toolkit, and discuss future research directions.

2 Fundamentals
Unlike other toolkits cited in §1, Effpi is designed on a formal foundation: a functional, concurrent message-passing calculus (called λΣ) with a blend of behavioural types (from π-calculus literature) [1, 26] and dependent function types (from Dotty) [4]. This theory, its related work, and some details about its implementation (as an embedded DSL in Dotty) are presented in [32]; here we give an informal summary.

Behavioural Types In π-calculus literature, the term behavioural type covers various kinds of types describing the communication behaviour of a program — i.e., its protocol. E.g., a type like !!:\text{int}; \text{string}!\text{send an integer; then, send a string}!!. Behavioural type systems ensure that, if a program P type-checks vs. a type/protocol T, then running P will yield the interactions specified by T; if P has interactions disallowed by T, type-checking fails. To model programs that interact with others via multiple communication channels, one can use more accurate behavioural types, e.g.:

\[
c_1?\text{int}; c_2!\text{string}
\]

which means “receive an integer from channel c_1; then, send a string over channel c_2.” Many works try to bridge the gap

PL ’18, January 01–03, 2019, New York, NY, USA
2019.

9
from π-calculus theory to practice, by creating new program-112 ming languages, or seeking ways to represent types like (1) in general-purpose languages. This is non-trivial, as some properties (e.g., static linearity checks) are tricky, and often lost in the translation to existing languages. For a survey, see [10]; some works targeting Scala are [27–30].

**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;
- Proc is the abstract supertype of all types above (except Chan): it represents a program that may interact (or not).

The types above become quite powerful when combined with one of Dotty’s distinguishing features: **dependent function types** [4]. In fact, Effpi builds upon a fundamental insight: dependent function types can be used in a novel way to track channel usage in programs. E.g., the type of a function that takes two channels c1 and c2, and uses them according to the behavioural type (1) above, is rendered as:

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

To produce programs with the types above, Effpi provides a DSL, that looks like the following code snippet (for now, ignore the optional type annotation “...: T” on line 1):

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

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 input/output operations. 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 above. 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" (which 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 "|" [6] 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 (§3.3) 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 protocols, and compose them, by leveraging Dotty’s type aliases and parameters. E.g., consider the well-known Dijkstra’s dining philosopher problem: two processes (the philosophers) share two resources (the forks), and want to acquire both (so they can eat), and then release them. A philosopher can only eat after acquiring both forks, and will not drop the first fork before picking the second. The goal is to let both philosophers eat, without deadlocks.

A type describing the intended behaviour of a fork 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, \(\ldots\)Loop[RecX]]).

Here is an implementation of the Fork protocol:
The type annotation `fork(...)`: Fork[acq.type, rel.type] ensures that the channels used by `fork()` are exactly its arguments `acq` and `rel`; and if the fork’s code tries, e.g., to use `acq/rel` in the wrong order, then it will not compile.

With the same approach, we can write the desired behaviour of a philosopher as a type, whose parameters are channel types to signal when forks are Picked and Dropped:

```scala
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 the channels used by `fork()` are exactly its arguments `acq` and `rel`; and if the fork’s code tries, e.g., to use `acq/rel` in the wrong order, then it will not compile.

With the same approach, we can write the desired behaviour of a philosopher as a type, whose parameters are channel types to signal when forks are Picked and Dropped:

```scala
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)
        }
      }
    }
  }
}
```

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

```scala
def philo2(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 a type describing a desired composition of philosophers and forks, and implement it:

```scala
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", p1, d1, p2, d2),
    philo("Aristotle", p1, d1, p2, d2))
}
```

Notice that the type annotation enforces the desired interconnection of channels among philosophers and forks.

### 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 Dining type itself is “wrong,” as it does not guarantee a desired property: deadlock freedom.

In general, when types/protocols are composed, and their components interact, they may exhibit unwanted behaviours. E.g., if we add the following annotation to `dining()` above...

```scala
@verify(property = "deadlock_free")
```

...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, checking whether the property selected in the `@verify(...)` annotation holds. If the verification succeeds, then the implementation enjoys the property.

In the example above, the verification fails: deadlock freedom does *not* hold for `dining()`’s type, hence `dining()` itself 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). It suffices to swap the arguments of the first Philo type, i.e.:

```scala
def dining2(p1: Chan[Unit], d1: Chan[Unit],
p2: Chan[Unit], d2: Chan[Unit]): Dining[p1.type, d1.type,
p2.type, d2.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)
        }
      }
    }
  }
}
```

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

```scala
@verify(property = "deadlock_free")
```

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, that follows its theoretical foundations (i.e., $\lambda\pi\pi \leq \pi\pi$ [32]). On top of it, Effpi includes higher-level abstractions and extensions, aiming at a more developer-friendly API.

One such extensions leverages Dotty’s implicit function types [7, 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 can either accept or reject them — but must report accepted payments to an auditor (the scenario is distilled from a use case for the Akka Typed toolkit [16, 21]):

```scala
@verify(property = "reactive(mb_)(aud) &
responsive(mb_)(aud) &
output_ev_followed(aud)(Accepted(mb_))")
def payment(aud: ActorRef[Audit[_]]): Actor[Pay, ...] =
forever {
  read { pay: Pay =>
    if (pay.amount > 42000) {
      send(pay.replyTo, Rejected("Too high!"))
    } else {
      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 mentioned in §2, the process/channel-based API of Effpi is an internal embedding of the $\lambda\pi\pi$ calculus [32] in Dotty, with minimal adaptations: this allows to leverage Dotty’s type system features (dependent function types, union types, ...), and allows for easy interoperability with other libraries and toolkits running on the Java Virtual Machine. E.g., Effpi processes (and actors) can easily interoperate with Akka Typed, via “bridges” that forward messages between Effpi channels and Akka ActorRefs; this trick can also be used to let 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 language embedding naturally yields a DSL where the continuations of input/output actions are functions (closures). We took advantage of this fact, to implement a runtime system with a (non-preemptive) scheduler that decouples Effpi processes/actors from system threads, similarly to Akka Dispatchers [18]: i.e., it interleaves the execution of active processes/actors, unschedules them when they are waiting for input, and resumes them when an input becomes available. Effpi’s runtime supports highly concurrent programs: for some benchmarks, and an encouraging comparisons with Akka’s performance, 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.

The broader goal behind Effpi is providing lightweight software verification capabilities that (1) can be used by programmers that 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. In particular, we believe that match types [5] 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 case. This would allow to represent and verify more protocols, possibly covering the whole range of multiparty session types [2, 31]. Effpi supports programs with 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].
References


