Verifying message-passing programs with dependent behavioural types

Alceste Scalas
Imperial College London
and Aston University, 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
Concurrent and distributed programming is notoriously hard. Modern languages and toolkits address the challenge by offering message-passing abstractions, such as actors (e.g., Erlang, Akka, Orleans) or processes (e.g., Go): they allow for simpler reasoning w.r.t. shared-memory concurrency, but do not ensure that a program implements a given specification.

To address this challenge, it would be desirable to specify and verify the intended behaviour of message-passing applications using types, and ensure that, if a program type-checks and compiles, then it will run and communicate as desired.

We develop this idea in theory and practice. We formalise a concurrent functional language \(\pi\)-calculi, with a new blend of behavioural types (from \(\pi\)-calculus theory), and dependent function types (from the Dotty programming language, a.k.a. the future Scala 3). Our theory yields four main payoffs: (1) it verifies safety and liveness properties of programs via type-level model checking; (2) unlike previous work, it accurately verifies channel-passing (covering a typical pattern of actor programs) and higher-order interaction (i.e., sending/receiving mobile code); (3) it is directly embedded in Dotty, as a toolkit called \(\Eff\), offering a simplified actor-based API; (4) it enables an efficient runtime system for \(\Eff\), for highly concurrent programs with millions of processes/actors.

CCS Concepts • Theory of computation → Process calculi; 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:

1 Introduction
Consider this specification for a payment service with auditing (from a use case for the Akka Typed toolkit [39, 47]):

1. the service waits for Pay messages, carrying an amount;
2. the service can decide to either:
   a. reject the payment, by sending Rejected to the payer;
   b. accept the payment. Then, it must report it to an auditing service, and send Accepted to the payer;
3. then, the service loops to 1, to handle new Payments.

This can be implemented using various languages and tools for concurrent and distributed programming. E.g., using Scala and Akka Typed [47], a developer can write a solution similar to Fig. 1: payment is an actor, receiving messages of type Pay (line 1); aud is the actor reference of the auditor, used to send messages of type Audit; whenever a pay message is received (line 3), payment checks the amount (line 4), and uses the pay.replyTo field to answer either Accepted or Rejected — notifying the auditor in the first case.

The typed actor references in Fig. 1 guarantee type safety: e.g., writing `send(aud, "Hi")` causes a compilation error. However, the payment service specification is not enforced: e.g., if the developer forgets to write line 7, the code still compiles, but accepted payments are not audited. This is a typical concurrency bug: a missing or out-of-order communication can cause protocol violations, deadlocks, or livelocks. Such bugs are often spotted late, during software testing or maintenance — when they are more difficult to find and fix, and harmful: e.g., what if unaudited payments violate fiscal rules?

These issues were considered during the design of Akka Typed, with the idea of using types for specifying protocols...
Our proposal is a new take on specifying and statically verifying the behaviour of concurrent programs, in two steps.

Step 1: enforcing protocols at compile-time We develop Effpi, a toolkit for message-passing programming in Dotty (a.k.a. Scala 3), that allows to verify the code in Fig. 1 against its specification, at compile time. This is achieved by replacing the rightmost “_” (line 1) with a behavioural type:

\[
\text{Forever[(In[Pay], (p: Pay) ⇒ // Depend...t[p.type] )) ] } ]
\]

With this type annotation, the code in Fig. 1 still type-checks and compiles; but if, e.g., line 7 is forgotten, or changed in a way that does not audit properly (e.g., writing null instead of aud), then a compilation error ensues. The type above formalises the payment service specification by capturing the desired behaviour of its implementation, and tracking which ActorReferences are used for interacting, and when. Type “In” (provided by Effpi) requires to wait for a message \( p \) of type Pay, and then either (\( | \) means “or”) send \( \text{Rejected} \) on \( p \).replyTo, or send an audit, and then \( (\Rightarrow) \) send \( \text{Accepted} \). Notably, \( p \) is bound by a dependent function type [15].

Effpi is built upon a concurrent functional calculus for channel-based interaction, called \( L_\pi \); its novelty is a blend of behavioural types (inspired by \( \pi \)-calculus literature) with dependent function types (inspired by Dotty’s foundation \( D_\pi \) [2]), achieving unique specification and verification capabilities. Effpi implements \( L_\pi \) as an internal DSL in Dotty – plus syntactic sugar for an actor-based API (cf. Fig. 1).

Step 2: verification of safety / liveness properties In Step 1, we establish the correspondence between protocols and programs, via syntax-driven typing rules. But this is not enough: programs may be expected to have safety properties (“unwanted events never happen”) or liveness properties (“desired events will happen”) [40]. E.g., in our example, we want each accepted payment to be audited; but in principle, an auditor’s implementation might be based on a type like:

\[
\text{In[Audit[]], (a: Audit[]) ⇒ End ]}
\]

(i.e., receive one Audit message \( a \), and terminate). This implementation, in isolation, may be deemed correct by mere type checking; however, if such an auditor is composed with the payment service above (receiving messages sent on aud), the resulting application would not satisfy the desired property: only one accepted payment is audited. With complex protocols, similar problems become more difficult to spot.

The issue is that types in \( L_\pi \) and Effpi can specify rich protocols — but when such protocols (and their implementations) are composed, they might yield undesired behaviours. Hence, we develop a method to: (1) compose types/protocols, and decide whether they enjoy safety / liveness properties; (2) transfer behavioural properties of types to programs.

**Contribution** We present a new method to develop message-passing programs with verified safety/liveness properties, via type-level model checking. The key insight is: we use variables in types, to track inputs/outputs in programs, through a novel blend of behavioural+dependent function types. Unlike previous work, our theory can track channels across transmissions, and verify mobile code, covering important features of modern message-passing programs.

Outline. §2 formalises the \( L_\pi \) calculus, at the basis of Effpi. §3 presents type system of \( L_\pi \). §4 shows the correspondence between type / process transitions (Thm. 4.4, 4.5), and how to transfer temporal logic judgements on types (that are decidable, by Lemma 4.7) to processes. This yields Thm. 4.10: our new method to verify safety / liveness properties of programs. §5 explains how the design of \( L_\pi \) naturally leads Effpi’s implementation (i.e., the paper’s companion artifact), and evaluates: (1) its run-time performance and memory use (compared with Akka Typed); (2) the speed of type-level model checking. §6 discusses related work. The technical report [61] contains proofs and more material.

2 The \( L_\pi \)-Calculus

The theoretical basis of our work is a \( \lambda \)-calculus extended with channels, input/output, and parallel composition, called \( L_\pi \). The “\( \pi \)” denotes both: (1) its use of dependent function types, that, together with subtyping \( \ll\), are cornerstones of its typing system (§3); and (2) its connection with the \( \pi \)-calculus [51, 52, 60]. Indeed, \( L_\pi \) is a monadic-style encoding of the higher-order \( \pi \)-calculus: continuations are \( \lambda \)-terms, and this will be helpful for typing (§3) and implementation (§5).

**Definition 2.1.** The syntax of \( L_\pi \) is in Fig. 2. Elements of \( C \) are run-time syntax. Free/bound variables \( fv(t) / bv(t) \) are defined as usual. We adopt the Barendregt convention: bound
variables are syntactically distinct from each other, and from free variables. We write λx... for λx.x... when x ∈ fv(t).

The set of values V includes booleans B, channel instances C, function abstraction, the unit (), and error. The terms (in T) can be variables (from X), values (from V), various standard constructs (negation ¬, if/then/else, let binding, function application), and also channel creation chan(), and process terms (from P). The primitive chan() evaluates by returning a fresh channel instance from C — whose elements are part of the run-time syntax, and cannot be written by programmers. Process terms include the terminated process end, the output primitive send(t, t′) (meaning: send t′ through t, and continue as t′), the input primitive recv(t, t′) (meaning: receive a value from t, and continue as t′), and the parallel composition t∥t′ (meaning: t and t′ run concurrently, and can interact). λT can be routinely extended with, e.g., integers, strings, records, variants: we use them in examples.

Example 2.2. A ping-pong system in λπ is written as:

```
let pinger = λself. Aponge. (let ponger = λself. (send(ponge, self, λ_. (recv(self, λreply. (send(replyTo, "Hi!", λ_. (end )))))) )
let sys = λy. λz. (pinger y z ‖ ponger z )
let main = λ_. let y = chan() in let z = chan() in sys y z
```

- pinger is an abstract process that takes two channels: self (its own input channel), and ponge. It uses ponge to send self, then uses self to receive a response, and ends;
- ponger takes a channel self, uses it to receive replyTo, then uses replyTo to send "Hi!", and ends;
- sys takes channels y′, z′, and uses them to instantiate pinger and ponger in parallel;
- invoking main () instantiates sys with y and z (containing channel instances): this lets pinger and ponger interact.

Note that in pinger and ponger, the last argument of send/recv is always an abstract process term: this is expected by the semantics (Def. 2.4), and enforced via typing (§3).

Remark 2.3. In Ex. 2.2, pinger/ponger use channel passing to realise a typical pattern of actor programs: they have their own "mailbox" (self), and interact by exchanging their own "reference" (again, self). We will leverage this intuition in §5.

Definition 2.4 (Semantics of λπ). Evaluation contexts E and reduction → are illustrated in Fig. 3, where congruence ≡ is defined as: t1 ≡ t2 ≡ t1 and end ≡ end ≡ end, plus α-conversion. We write → for the reflexive and transitive closure of →. We say "t has an error" iff t ∈ E[err] (for some E). We say "t is safe" iff ∀t′ : t → t′ implies t′ has no error.

Def. 2.4 is a standard call-by-value semantics, with two rules for concurrency. [R-chan] says that chan() returns a fresh channel instance; [R-Com] says that the parallel composition send(a, u, v1)recv(a, v2), where both sides operate on a same channel instance a, transfers the value u on the receiver side, yielding v1 () | v2 u: hence, if v1 and v2 are function values, the process keeps running by applying v1 () and v2 u — i.e. the sent value is substituted inside v2. The error rules say how terms can "go wrong": they include usual type mismatches (e.g., it is an error to apply a non-function value u to any v), and three rules for concurrency: it is an error to receive/send data using a value u that is not a channel, and it is an error to put a value in a parallel composition (i.e., only processes from P in Fig. 2 are safely composed by |).

3 Type System

We now introduce the type system of λπ. Its design is reminiscent of the simply-typed λ-calculus, except that (1) we include union types and equi-recursive types, (2) we add types for channels and processes, and (3) we allow types to contain variables from the term syntax (inspired by Dc, the calculus behind Dotty [2]). The syntax of types is in Def. 3.1.

Notably, points (1) and (3) establish a similarity between λ≤ and Fc. (System F with subtyping [8]) equipped with equi-recursive types [29]. Indeed, point (3) means that a type T is only valid if its variables exist in the typing environment — which, in turn, must contain valid types. Similarly, in Fc, polymorphic types can depend on type variables in the environment; hence, we use mutually-defined judgements, akin to those of Fc, to assess the validity of environments, types, subtyping, and typed terms (Def. 3.2).

Definition 3.1 (Syntax of types). Types, ranged over by S, T, U, ..., are inductively defined by the productions:

```
| bool | () | T | U | T ' ∨' U |
| cT | µx. T | x |
| proc | nil | o[S, T, U] | i[S, T] | p[T, U] |
```

Free/bound variables are defined as usual. We write U(S/T) for the type obtained from U by replacing its free occurrences of x with S. If T = µx. (x'/U)' ‖ U, then T S stands for U(S/x).

We write Π(T) for Π(x/S)T if x /∈ fv(T), and distinguish recursion variables as t, t′, ..., i.e., we write µt.T. We write T for an n-tuple T1, ..., Tn, and T ‖ U if T occurs in U.

The relation ≡ is the smallest congruence such that:

```
```

The first row of productions in Def. 3.1 includes booleans, the unit type (.), top/bottom types ⊤,⊥, the union type T ∨ U, the dependent function type Π(x/S)T and the recursive type µx.T (they both bind x with scope T), and variables x (from the set X in Def. 2.1): the underlining is a visual clue to better distinguish x used in a type, from x used in a λT term.

The second row of Def. 3.1 formalises channel types: cT denotes a channel allowing to input or output values of type T; instead, cT only allows for input, and cT for output. The third row of Def. 3.1 formalises process types. The generic process type proc denotes any process term; nil denotes a
terminated process; the output type \( t_0 \in \text{Out}[S,T,U] \) denotes a process that sends a \( T \)-typed value on an \( S \)-typed channel, and continues as \( U \); the input type \( \text{In}[S,T,U] \) denotes a process that receives a value from an \( S \)-typed channel and continues as \( T \); the parallel type \( [T, U] \) denotes the parallel composition of two processes (of types \( T \) and \( U \)).

**Definition 3.2.** These judgement are formalised in Fig. 4:

\[
\begin{align*}
& \Gamma \vdash \text{env} \quad \Gamma \text{ is a valid typing environment} \\
& \Gamma \vdash T \quad \text{is a valid type in } \Gamma \\
& \Gamma \vdash T \quad \text{holds iff } \forall U \in \Gamma : \Gamma = \Pi U \text{ type} \\
& \Gamma \vdash T \quad \text{is a valid process type in } \Gamma \\
& \Gamma \vdash T \quad \text{holds iff } \forall U \in \Gamma : \Gamma = \Pi T \text{ type} \\
& \Gamma \vdash t : T \\
& \Gamma \vdash t : T \\
& \vdash ?E \quad \Gamma \text{ is a valid evaluation context} \\
& \vdash \text{notin} \quad \Gamma \text{ is a valid type} \\
& \vdash \text{notin} \quad \Gamma \text{ is a valid process type} \\
& \vdash \text{notin} \quad \Gamma \text{ is a valid process} \\
& \vdash \text{notin} \quad \Gamma \text{ is a valid error type} \\
& \vdash \text{notin} \quad \Gamma \text{ is a valid error process} \\
\end{align*}
\]

A typing environment \( \Gamma \) maps variables (from \( X \) in Def. 2.1) to types; the order of the entries of \( \Gamma \) is immaterial. All judgements in Fig. 4 are inductive, except subtyping, that is coinductive (hence the double inference lines). Crucially, in Fig. 4 we have two valid type judgements, for two kinds of types: \( \Gamma \vdash T \) and \( \Gamma \vdash T \pi \text{-type} \). The former is standard (except for rule \( \lceil \rightarrow \rceil \), for valid channel types) while the latter distinguishes process types. Note that subtyping only relates types of the same kind. Importantly, a typing environment \( \Gamma \) can map a variable to a type (rule \( \lceil \rightarrow \rceil \)), but not to a \( \pi \)-type; this also means that function arguments cannot be \( \pi \)-typed.

Still, in a function type \( \Pi \langle x : T \rangle U \), the return type \( U \) can be a \( \pi \)-type (rule \( \lceil \rightarrow \rceil \)); hence, it is possible to define abstract process types (cf. Ex. 3.3 and 3.4 later on). Rules \( \lceil ?T \rangle \) and \( \lceil \pi \rangle \) are based on [29, §2], and require recursive types to be contractive: e.g., \( \mu_1, \mu_2, \ldots, \mu_n \) (\( \mu_1 \cup V \)) is not a type; clause \( \langle x \notin \text{fv}^{-1}(T) \rangle \) means that variable \( x \) is not bound in negative position in \( T \), as in \( F^e \). (Details: [61]). Recursion is supported by \( \lceil \rightarrow \rceil \); in \( \vdash x = t \text{ in } T \), \( t \) can refer to \( x \). Rule \( \lceil \nu \rceil \), based on [9], ensures decidability of subtyping [29, §1]: it is often needed in practice, and we use it in Def. 4.2, Lemma 4.7. The rest of Fig. 4 is standard; we now discuss the main judgements.

**Variables, types, subtyping, and dependencies** The environment \( \Gamma : x : T \) assigns type \( T \) to variable \( x \). Hence, by rule \( \lceil \rightarrow \rceil \), the type \( x \) is valid in \( \Gamma \); and indeed, by rule \( \lceil \rightarrow \rceil \), we can infer \( \Gamma \vdash x : T \), i.e., the term \( x \) has type \( T \). Intuitively, this means that \( x \) is the "most precise" type for term \( x \); this is formally supported by the subtyping rule \( \lceil \vdash \rceil \), that says: as \( \Gamma \) maps term \( x \) to \( T \), type \( x \) is smaller than \( T \). To retrieve from \( \Gamma \) the information that term \( x \) has (also) type \( T \), we use subtyping and subsumption (rule \( \lceil \rightarrow \rceil \)), as shown here. Since \( x \) is the smallest type for term \( x \), the judgement \( \Gamma \vdash t : x \) convays that \( t \) should be "something that evaluates to \( x \)" e.g., \( t = x \) or \( t = \text{if } tt \text{ then } x \text{ else } x \); similarly, the dependent function type \( \Pi \langle x : \text{bool} \rangle X \) is inhabited by terms like \( \lambda x. x \) or \( \lambda x. (\lambda y. y) x \). Thus, we can roughly say: if \( x \) occurs in \( T \), then \( T \)-typed terms correspondingly use \( x \). This insight will be crucial for our results.

**Channels, processes, and their types** By \( \lceil \text{chan} \rangle \), a (type-annotated) term \( \text{chan} \langle T \rangle \) has type \( \text{chan}[T] \). Rule \( \lceil \rightarrow \rceil \) is similar, for channel instances. By \( \lceil \text{end} \rangle \), process end has type \( \text{nil} \).

By \( \lceil \nu \rceil \), both sub-types of \( t_0 \leq t_2 \) are \( \pi \)-typed.

By \( \lceil \text{send} \rangle \), \( \text{send}(t_0, t_2, t_3) \) has type \( \text{Out}[S,T,U] \), under the validity constraints of rule \( \lceil \pi \rangle \). Hence, \( T_0 \) has a channel type for sending values of type \( T_0 \) and \( t_2 \) (the term being sent) must have type \( T_0 \); also, \( t_3 \)’s type must be \( U = \Pi \langle \text{U} \rangle \) (for a \( \pi \)-type \( T_0 \)): i.e., \( t_3 \) is a process thunk, run by applying \( t_3 \).

By \( \lceil \text{recv} \rangle \), \( \text{recv}(t_0, t_2) \) has type \( \text{In}[S,T] \), which is well-formed under rule \( \lceil \pi \rangle \). Hence, the sub-term \( t_1 \) must have a channel type with input \( U \), while \( t_2 \) must be an abstract process of type \( T = \Pi \langle x : \text{U} \rangle T' \), with \( T' \)-type. Crucially, by rule \( \lceil \pi \rangle \), we have \( \Gamma \vdash U \leq U' \); hence, it is safe to receive a value \( u \) from \( t_1 \), and apply \( t_2 \) to get a continuation process that uses \( u \).

We explain subtyping in Fig. 4 later, after a few examples.

**Example 3.3.** In Ex. 2.2, we have the type assignments:

\[
\begin{align*}
& \text{pinger} : T_{\text{ping}} = \Pi \langle \text{self} : \text{chan}[\text{str}] \rangle \Pi \langle \text{pong} : \text{chan}[\text{str}] \rangle \Pi \langle \text{pong} : \text{chan}[\text{str}] \rangle \\
& \text{pinger} : T_{\text{pong}} = \Pi \langle \text{self} : \text{chan}[\text{str}] \rangle \\
& \text{sys} : T_{\text{pp}} = \Pi \langle \text{chan}[\text{str}] \rangle \Pi \langle \text{chan}[\text{str}] \rangle \Pi \langle \text{chan}[\text{str}] \rangle \\
\end{align*}
\]

Notice how \( T_{\text{pp}} \) captures the ping/pong composition of sys, preserving its channel topology: the type-level applications
Verifying message-passing programs with dependent behavioural types

PLDI ’19, June 22–26, 2019, Phoenix, AZ, USA

Figure 4. Judgements of the \( \lambda^{\pi^T} \) type system (Def. 3.2). The main concurrency-related rules are highlighted.

By inspecting \( T_m \), we infer that, e.g., \( T_m \)-typed terms cannot be forkbombs; also, \( x \lor y \) does not allow to send on \( o \) a value not coming from \( i_1 \) or \( i_2 \) (we will formalise these intuitions in Ex. 4.11). The terms below implement \( T_m \): \( m_1 \) always sends \( o \) received from \( i_1 \), then recursively calls itself, swapping \( i_1 \) with \( i_2 \); \( m_2 \) sends the maximum between \( x \) and \( y \).

\[
\begin{align*}
&\text{let } m_1 = \lambda i_1. \lambda i_2. \lambda o. \\
&\quad \text{recv}(i_1, \lambda x. \lambda x. \text{recv}(i_2, \lambda y. \text{send}(o, x. x. \lambda x. o, i_1 i_2 o))) \\
&\text{let } m_2 = \lambda i_1. \lambda i_2. \lambda o. \\
&\quad \text{recv}(i_1, \lambda x. \lambda x. \text{recv}(i_2, \lambda y. \text{send}(o, (x > y \text{ then } x \text{ else } y), x. \lambda m_2 i_1 i_2 o))
\end{align*}
\]

Below, \( \text{srv} \) is a data processing server. It takes two channels: \( \text{cm} \) and \( \text{out} \); it creates two private channels \( i_1 \) and \( i_2 \), uses \( \text{cm} \) to receive an abstract process \( p \), and runs it, in parallel with two \( \text{producers} \) (omitted) that send values on \( i_1 \) or \( i_2 \):

\[
\begin{align*}
&\text{let } m_1 = \lambda i_1. \lambda i_2. \lambda o. \\
&\quad \text{recv}(i_1, \lambda x. \lambda x. \text{recv}(i_2, \lambda y. \text{send}(o, (x > y \text{ then } x \text{ else } y), x. \lambda m_2 i_1 i_2 o))
\end{align*}
\]
let srv = \text{\textsf{lambda}}.m.\text{\textsf{out}}.
    let z1 = \textsf{chan}() in let z2 = \textsf{chan}() in
    \text{\textsf{recv}}(m, \lambda p. (p z1 z2 out \parallel \text{\textsf{prod}} z1 \parallel \text{\textsf{prod}} z2))

The system works correctly if the received code \( p \) is \( m_1 \) or \( m_2 \) above — or any instance of \( T_m \). In order to ensure that \( \text{\textsf{srv}} \) can only receive a \( T_m \)-typed term on \( cm \), we check its type:

\[
0 \vdash \text{\textsf{srv}} : T_{cm} = \Pi((\text{\textsf{cm}}c(T_m)) \Pi(\text{\textsf{out}}c(\text{\textsf{int}})) \text{\textsf{proc}})
\]

and this guarantees that, e.g., the parallel composition

\[
\text{\textsf{send}}(x, \lambda \_\_. \_\_\_) \parallel \text{\textsf{srv}} x o (\text{client sends } t \text{ to server, via } x)
\]

is typable in \( \Gamma \) only if \( \Gamma \vdash x : c^2(T_m) \), implying \( \Gamma \vdash t : T_m \). We can replace \text{\textsf{proc}} with a more precise type. If \( U_1 / U_2 \) are types of \( \text{\textsf{prod}}_1/\text{\textsf{prod}}_2 \), the \text{\textsf{recv}}(...) sub-term of \text{\textsf{srv}} has type:

\[
T_{\text{\textsf{srv}}} = \Pi [c, \Pi (p; T_m)] \Pi [p \in T_m z_1 z_2 out \in U_1, z_1 \in U_2]
\]

where the server uses \( c \) to receive a \( T_m \)-typed abstract process \( p \), and then behaves as \( T_m \) (applied to \( z_1, z_2, out \)) composed in parallel with \( U_1 / U_2 \) (applied to \( z_1 / z_2 \)).

4 Type-Level Model Checking

Our typing guarantees conformance between processes and types (Fig. 4), and absence of run-time errors (Thm. 3.6). However, as seen in §1, our types can describe a wide range of behaviours, from desirable ones (e.g., formalising a specification), to undesirable ones (e.g., deadlocks); moreover, complex (and potentially unwanted) behaviours can arise when \( \lambda^T \) terms are allowed to interact.

To avoid this issue, we might want to check whether a process \( t \) (possibly consisting of multiple parallel sub-processes) satisfies a property \( \phi \) in some temporal logic [64]: \( \phi \) could be, e.g., a safety property \( \Box(\neg \phi') \) (“\( \phi' \) is never true while \( t \) runs”) or a liveness property \( \Diamond \phi' \)”(\( t \) will eventually satisfy \( \phi' \)). However, this problem is undecidable (unless \( \phi \) is trivial), since \( \lambda^T \) is Turing-powerful even in its productive fragment (due to recursion and channel creation [7]).

Luckily, our theory allows to: (1) mimic the parallel composition of terms by composing their types (as shown in Ex. 3.3), and (2) mimic the behaviour of processes by giving a semantics to types (as we show in this section). This means that we can ensure that a (composition of) typed processes \( t \) has a desired safety/liveness property, by model-checking its type \( T \) (that is not Turing-powerful). Moreover, we do not need to know how \( t \) is implemented: we only need to know that it has type \( T \). We now illustrate the approach, and its preconditions (roughly: for the verification of liveness properties, we need productivity, and use of open variables).

Outline First, we need to surround a typical obstacle for behavioural type systems. Ex. 3.5 shows that accurate types require open terms in their typing environment — but Def. 2.4 works on closed terms; so, observing how \( T_1 \) in Ex. 3.5 uses \( x \), we sense that \( t_1 \) should interact via \( x \) — but by Def. 2.4, \( t_1 \) is stuck. To trigger communication, we may bind \( x \) in \( t_1 \) with a channel instance, e.g., \( t'_1 = \text{let } x = \text{\textsf{chan}}() \text{ in } t_1 \) — but \( t'_1 \)'s type cannot mention \( x \), hence cannot convey which channel(s) \( t'_1 \) uses. Thus, we develop a type-based analysis in four steps: (1) we define an over-approximating LTS semantics for typed \( \lambda^T \) terms with free variables (Def. 4.1); (2) we define an LTS semantics for types (Def. 4.2); (3) we prove subject transition and type fidelity (Thm. 4.4, 4.5); (4) using them, we show how temporal logic judgements on type terms transfer to processes.
In general, we want a novel use of type variables, and dependent function types, sequence of internal moves transitions through contexts, unless labels refer to bound

Example 4.3. Take sys from Ex. 2.2, \( T_{pp} \) from Ex. 3.3. Let:

\[
\Gamma = \lambda x : \text{str}, \; z : \text{str} \vdash \text{send} \; \text{str}
\]

\[
T = T_{pp} \; \text{send} \; \text{str} = \mathcal{P} \{ [o \vdash \mathcal{I} \Gamma \Pi \Pi \; \text{str} \; \text{send} \; \text{str}, \; \Pi \Pi] \}
\]

By Def. 4.1, we have \( \Gamma + t \vdash T \). By Def. 4.1, we have:

\[
\Gamma + t \vdash \lambda x, \Pi \Pi : c \quad \text{recv}(y, \ldots) \quad \text{send}(x, \Pi \Pi, \Pi \Pi)
\]

By Def. 4.2, applying rule \( \text{r-[i-o]} \) twice, we get:

\[
\Gamma + T \vdash \lambda x, \Pi \Pi : c \quad \text{recv}(y, \ldots) \quad \text{send}(x, \Pi \Pi, \Pi \Pi)
\]

Observe that \( T \) closely mimicks the transitions of \( t \): the type-level substitution of \( y \) in place of \( \text{recv} \) allows to track the usage of \( y \) after its transmission, capturing pinger’s reply to ping. This realises our insight: tracking inputs/outputs of programs, by using variables in their types. Technically, it is achieved via the dependent function type inside \([\ldots, \ldots] \).

Subject transition and type fidelity With the semantics of Def. 4.1, we prove a result yielding Thm. 3.6 as a corollary.

Theorem 4.4 (Subject transition). Assume \( \Gamma + t : T \). If \( \Gamma + t \vdash t' \) then \( \Gamma + t \vdash t' \). Otherwise, when \( \Gamma + t \vdash t' \), we have:

1. \( \Gamma + t \vdash t' \) with \( \text{r-\alpha} \) (Fig. 5) implies \( \Gamma + t \vdash T \);
2. \( \Gamma + t \vdash t' \) and \( \alpha \in \text{[\text{x}(w), \text{x}(w), \text{r}[x], \text{r}[\text{Comm}]]} \) implies one:
   a. \( \Gamma + t' \vdash T \) and \( \text{proc} \in T \);
   b. \( \alpha \equiv \text{x}(w) \) and \( \exists S, U, T' : \Gamma + x : S, w : U, t' : T' \) and
   c. \( \Gamma + t \vdash \text{r}[x], \text{r}[\text{Comm}], T \);
   d. \( \alpha = \text{x}(w) \) and \( \exists S, U, T' : \Gamma + x : S, w : U, t' : T' \) and
   e. \( \alpha = \text{r}[\text{Comm}] \) and \( \exists S, S', T' : \Gamma + x : S, w : U, t' : T' \) and
   and \( \Gamma + t \vdash \text{r}[x], \text{r}[\text{Comm}], T' \);

Assume \( \Gamma + t : T \), with \( t \) reducing to \( t' \). Thm 4.4 says that when the reduction is caused by the functional fragment of \( \lambda^2 \) (hypothesis \( \Gamma + t \) type, or case 1), then \( t' \) has the same type \( T \). Instead, if the reduction is caused by input, output
Theorem 4.5 (Type fidelity). Within productive $\lambda^\pi_{\leq}$, assume $\Gamma \vdash t : T$ and $\Gamma \vdash T \pi$-type. Then:

1. $\Gamma \vdash T \equiv (U \times \pi) \rightarrow T'$ implies $\exists w, t' : \Gamma \vdash w : U, t' : T'$ and $\Gamma \vdash t \overset{\pi}{\rightarrow} s^\pi t$; $t'$;
2. $\Gamma \vdash T \equiv (U \times \pi) \rightarrow T'$ implies $\forall w : \Gamma \vdash w : U, \exists \nu' : \Gamma \vdash t' : T'$ and $\Gamma \vdash t \overset{\pi}{\rightarrow} s^\pi (\overline{x}^{\nu'})$;
3. $\Gamma \vdash T \equiv (U \times \pi) \rightarrow T'$ implies $\exists \nu' : \Gamma \vdash t' : T'$ and $\Gamma \vdash t \overset{\pi}{\rightarrow} s^\pi \overline{x} = \nu'$; $t'$;
4. $\Gamma \vdash T \overset{\pi}{\rightarrow} t'$ implies either: (a) $\exists T' : \Gamma \vdash T \overset{\pi}{\rightarrow} T'$ and $\Gamma \vdash t' : T'$ or (b) $\exists t' : \Gamma \vdash t \overset{\pi}{\rightarrow} t'$ with $\tau'(\alpha) \downarrow (\text{Fig. } 5)$ and $\Gamma \vdash t' : T'$.

Figure 5. Over-approximating labelled semantics of $\lambda^\pi_{\leq}$ terms. We will sometimes use label $\tau$ to denote any $\tau[-]$-label above.

Figure 6. Semantics of $\lambda^\pi_{\leq}$ types. We will sometimes use label $\tau$ to denote either $\tau[\upsilon]$ or $\tau[S,S']$ (for some $S,S'$).

or interaction events, then we observe a corresponding labelled transition in the type, possibly after some $\tau[\upsilon]$ moves (cases 2b–2e); the exception is case 2a: if $t'$ keeps type $T$, then that $T$ syntactically contains proc, types which a reducing sub-term of $t$ before and after its reduction (via rule $[\tau \leq]$).

We can also prove the opposite direction of Thm. 4.4: if type $T$ interacts, then a typed term $t$ interacts accordingly. This intuition holds under two conditions, leading to Thm. 4.5:

(c1) we only use productive $\lambda^\pi_{\leq}$ terms, i.e., all functions must be total (always return a value or process when applied). This means that, e.g., if $\Gamma \vdash t : \text{of}[x,\text{int},T']$, then $t$ will output on $x$; this excludes cases like $t = \text{if } \omega \text{ then send}(x,42,t') \text{ else send}(x,43,t'')$ (with $\omega = (\lambda y.y)z$). Productivity is obtained with many methods from literature (e.g., [20, 63]);
(c2) the subjects of input/output/interaction transitions of $T$ must be type variables: this allows to precisely relate them to occurrences of (open) variables in $t$.

Theorem 4.5 (Type fidelity). Within productive $\lambda^\pi_{\leq}$, assume $\Gamma \vdash t : T$ and $\Gamma \vdash T \pi$-type. Then:

1. $\Gamma \vdash T \equiv (U \times \pi) \rightarrow T'$ implies $\exists w, t' : \Gamma \vdash w : U, t' : T'$ and $\Gamma \vdash t \overset{\pi}{\rightarrow} s^\pi t$; $t'$;
2. $\Gamma \vdash T \equiv (U \times \pi) \rightarrow T'$ implies $\forall w : \Gamma \vdash w : U, \exists \nu' : \Gamma \vdash t' : T'$ and $\Gamma \vdash t \overset{\pi}{\rightarrow} s^\pi (\overline{x}^{\nu'})$;
3. $\Gamma \vdash T \equiv (U \times \pi) \rightarrow T'$ implies $\exists \nu' : \Gamma \vdash t' : T'$ and $\Gamma \vdash t \overset{\pi}{\rightarrow} s^\pi \overline{x} = \nu'$; $t'$;
4. $\Gamma \vdash T \overset{\pi}{\rightarrow} t'$ implies either: (a) $\exists T' : \Gamma \vdash T \overset{\pi}{\rightarrow} T'$ and $\Gamma \vdash t' : T'$ or (b) $\exists t' : \Gamma \vdash t \overset{\pi}{\rightarrow} t'$ with $\tau'(\alpha) \downarrow (\text{Fig. } 5)$ and $\Gamma \vdash t' : T'$.

Items 1–3 of Thm. 4.5 say that if $T$ can input/output/interact, then $t$ can do the same, possibly after a sequence of $\tau$-steps (without communication, cf. Def. 4.1); the $\tau$-sequence is finite, since $t$ is productive by hypothesis. By item 4, if $T$ can make a choice ($\upsilon$), then $t$ could have already chosen one option (case a), or could choose later (cases b or c).

Process verification via type verification By exploiting the correspondence between process / type reductions in Thm. 4.4 and 4.5, we can transfer (decidable) verification results from types to processes. To this purpose, we analyse the labelled transition systems (LTSs) of types and processes using the linear-time $\mu$-calculus [19, §3]. We chose it for two reasons: (1) the open term / type semantics (Def. 4.1 / 4.2) are over-approximating, and a linear-time logic is a natural tool to ensure that all possible executions ("real" or approximated) satisfy a formula; and (2) linear-time $\mu$-calculus is decidable for our types, with minimal restrictions (Lemma 4.7).

Definition 4.6 (Linear-time $\mu$-calculus). Given a set of actions $\mathbf{Act}$ ranged over by $\alpha$, the linear-time $\mu$-calculus formulas are defined as follows (where $\mathbf{A}$ is a subset of $\mathbf{Act}$):

Basic formulas: $\phi ::= Z \mid \neg \phi \mid \phi_1 \land \phi_2 \mid (\alpha) \phi \mid vZ.\phi$

Derived formulas: $(\mathbf{A})\phi \mid (\neg \mathbf{A})\phi \mid \phi_1 \lor \phi_2 \mid \phi_1 \Rightarrow \phi_2 \mid \mu \mathbf{Z}.\phi \mid \Box \phi \mid \Diamond \phi$
In Def. 4.6, $\phi$ describes accepted sequences of actions; $\phi$ can be a variable $Z$, negation, conjunction, prefixing ($\alpha\phi$) ("accept a sequence if it starts with $\alpha$, and then $\phi$ holds"), or greatest fixed point $\nu^Z\phi$. Basic formulas are enough [6, 64] to derive true/false (accept any/no sequence of actions), disjunction, implication, least fixed points $\mu Z\phi$. $(\alpha\phi)$ accepts sequences that start with any $\alpha \in A$, then satisfy $\phi$; dually, $(\neg\alpha\phi)$ requires $\alpha \in \text{Act}\setminus\alpha$. We also derive usual temporal formulas $\phi_1 U \phi_2$ ("$\phi_1$ holds, until $\phi_2$ eventually holds"), $\Box\phi$ ("$\phi$ is always true"), and $\Diamond\phi$ ("$\phi$ is eventually true"). Given a process $p$ with LTS of labels $\text{Act}$, a run of $p$ is a finite or infinite sequence of labels fired along a complete execution of $p$; we write $p \models \phi$ if $\phi$ accepts all runs of $p$. (Details: [61])

We can decide $\phi$ on a guarded type $T$, as shown in Lemma 4.7. Here, we instantiate $\text{Act}$ (Def. 4.6) as $\text{Act}_T(T)$, which is the set of labels fired along $T$’s transitions in $\Gamma$, (Def. 4.2); notably, $\text{Act}_T(T)$ is finite and syntactically determined. (Details: [61])

**Lemma 4.7.** Given $\Gamma$, we say that $T$ is guarded iff, for all $\pi$-type subterms $\mu T. U$ of $T$, $t$ can occur in $U$ only as subterm of $t[\ldots]$ or $t[\ldots]$; then, if $T$ is guarded, $T \vdash \phi$ is decidable.

Lemma 4.7 holds since guarded $\pi$-types are encodable in CCS without restriction [50], then in Petri nets [21, §4.1], for which linear-time $\mu$-calculable is decidable [19]. Notably, Lemma 4.7 covers infinite-state types (with $[\ldots]$ under $\mu T. [\ldots]$) that type $\Omega$ terms with unbounded parallel subterms.

Now, assuming $\Gamma \vdash t : T$, we can ensure that $\phi$ holds for $t$, by deciding a related formula $\phi' \omega$ on $T$. We need to take into account that type semantics approximate process semantics:

(i) if we do not want $t$ to perform an action on channel $x$, we check that $T$ never potentially uses type variable $x$;

(ii) if we want $t$ to eventually perform an action on channel $x$, we need $t$ productive, and check that $T$ eventually uses $x$ — without doing “imprecise” actions before.

We formalise such intuitions in various cases, in Thm. 4.10; but first, we need the tools of Def. 4.8 and 4.9.

**Definition 4.8.** The input / output uses of $x$ by $T$ in $\Gamma$ are:

input uses: $\exists \Gamma, U, x. T \vdash \text{in}(x)[U] \rightarrow (\exists S'. \Omega) \rightarrow S' \leq S'$

output uses: $\exists \Gamma, U, x. T \vdash \text{out}(x)[U] \rightarrow (\exists S'. \Omega) \rightarrow S' \leq S'$

**Definition 4.9.** Given a set of type (resp. term) variables $\Sigma$, the $\Sigma$-limited transitions of $T$ (resp. $t$) in $\Gamma$ are:

$\Gamma \vdash T \Rightarrow \Sigma \rightarrow T' \Rightarrow \Sigma$ implies $S \leq \Sigma$

$T \vdash \Sigma \Rightarrow \Sigma \Rightarrow T' \vdash \Sigma$

$\Gamma \vdash t \Rightarrow \Sigma \rightarrow t' \Rightarrow \Sigma \rightarrow w \vdash \Sigma \rightarrow w' : a \in (w[w(w), \tilde{w}(w)])$ implies $w \leq \Sigma$

$\Gamma \vdash t \Rightarrow \Sigma \Rightarrow \Sigma \rightarrow t' \vdash \Sigma$

**Theorem 4.10.** Within productive $\lambda^T_\Sigma$, assume $\Gamma \vdash t : T$, with $\Gamma \vdash \lambda^T_\Sigma$, proc $\notin T$. Also assume, for all $i[S, \Pi(x)[U]]$ occurring in $T$, that there is $y$ such that $\Gamma \vdash y : U$ holds.¹

For $\mu$-calculus judgements on $T$, let $\text{Act} = \text{Act}_\Sigma(T)$, and $\text{Act}_\Sigma = \{\tau[S, S'] \in \text{Act}_\Sigma(T) \mid \{S, S'\} \not\subseteq \text{dom}(T)\}$. Then, the implications in Fig. 7 hold.

Assume $\Gamma \vdash t : T$. The sets $\exists \Gamma, T(x)[U] / \exists \Gamma, t(x)[U]$ in Def. 4.8 contain all transition labels that might be fired by $T$, when $x$ is used for input/output by $t$. The operator $\uparrow \Gamma, \{x_i \in \ldots\}$ (Def. 4.9) limits the observable inputs/outputs of $T/t$ to those occurring on channel $x_i$ — while other (open) channels can only reduce by communicating, via $\tau$-actions; i.e., $x_1, \ldots, x_n$ are interfaces to other types/processes, and are "probed" for verification (this is common in model checking tools).

In Thm. 4.10, item (1) can be seen as a case of intuition (1) above: if $T$ never fires a label (\(\exists \Gamma (\neg \omega)\)) that is a potential output use of $x_i$ ($i \in 1..n$), then $t$ never uses $x_i$ for output.

The "potential output use", by Def. 4.8, is any label $\exists \Gamma (\neg \omega)$ fired by $T$ where $S'$ is a supertype of $x$; this accounts for "imprecise typing", discussed in Ex. 3.5. Item (3) of Thm. 4.10 is a case of intuition (2): to ensure that $t$ eventually outputs on $x_i$ ($i \in 1..n$), we check that $T$ eventually fires a label $\exists \Gamma (\neg \omega)$; moreover, we check $T$ does not fire any label in $\text{Act}_\Sigma$, until $(U)$ the output $\exists \Gamma (\neg \omega)$ occurs. The set $\text{Act}_\Sigma$ contains all "imprecise" synchronisation labels $\tau[S, S']$ where either $S$ or $S'$ is not a type variable: we exclude them because, if $T$ fires one, then we cannot use Thm. 4.5(3) to ensure that $t$ reduces accordingly; i.e., if we do not exclude $\text{Act}_\Sigma$, then $t$ might deadlock and never perform $\exists \Gamma (\neg \omega)$ (for any $w$). Finally, item (4) combines the intuitions of both previous cases: we want to ensure that whenever $t$ receives $z$ on channel $x$, then it eventually forwards $z$ through channel $y$, without doing other inputs on $x$ before; to this purpose, we check that whenever $T$ inputs $z$ on a channel $S$ (representing a potential use of $\text{in}(x)[U]$), then $T$ eventually fires $\exists \Gamma (\neg \omega)$ — without doing potential inputs on $x$, nor firing any label in $\text{Act}_\Sigma$, before.

**Example 4.11.** Take $\Gamma, t, T$ in Ex. 4.3. To ensure that $t$ eventually uses $y$ to output a message, we check $T \vdash \{y\} \models \phi$, with $\phi$ in Fig. 7(3) (right).

Take ponger (Ex. 2.2), $T_{pong}$ (Ex. 3.3), and $\Gamma = z : \alpha[a]\{\text{c}[\text{str}]\}$. To ensure that the term ponger $z$ is responsive on $z$, we check $(T_{pong} z) \vdash \{z\} \models \phi$, with $\phi$ in Fig. 7(6) (right).

Take $T'_{srv}$ (Ex. 3.4). With an easy adaptation of properties (5) and (4) in Fig. 7 (right), we can verify that: in all implementations $sr' \in T'_{srv}$ whenever $sr' \text{ receives } p' = \text{type } T_m \text{ from channel } cm$, $sr' \text{ becomes reactive on } z_1, z_2$, picking one input and forwarding it on out.¹

5 Implementation and Evaluation

We designed $\lambda^T_\Sigma$ to leverage subtyping and dependent function types, with a formulation close to (a fragment of) Dotty compromising between simplicity and generality, that is sufficient to verify our examples. Besides this, the existence of $y$ such that $\Gamma \vdash y : U$ can be assumed w.l.o.g.: if $\Gamma \vdash t : T$ but $\exists \Gamma (\neg \omega)$ such that $\Gamma \vdash \phi$, we can pick $y' \notin \text{dom}(T)$, extend $\Gamma$ as $\Gamma' = \Gamma, y'/U$, and get $\Gamma' \vdash y' : U$ and $\Gamma' \vdash t : T$.¹
Figure 7. Process verification (Thm. 4.10): the judgement on the left is implied by the companion judgement on the right. Here, \( w \) ranges over \( \bigvee \bigcup \) and we write \( \mathfrak{T}(w) \) as shorthand for the (infinite) set of labels \( \{\mathfrak{T}(w)\mid w \in \bigvee \bigcup\} \) (and similarly for \( x(w) \)). For brevity, in (4) and (6) we write \((\alpha)\Rightarrow \phi\) instead of \((\alpha)\Rightarrow (\alpha)\phi\) (i.e., if we observe \( \alpha \), then \( \phi \) holds afterwards).

(a.k.a. the future Scala 3 programming language), and its foundation \( D < \). [2]. This naturally leads to an implementation, in three phases, shown in §5.1: (1) internal embedding of \( \lambda T \); (2) actor-based APIs, via syntactic sugar; and (3) compiler plugin for type-level model checking. To assess our approach, we performed the measurements detailed in §5.2.

5.1 Implementation

A payoff of the \( \lambda T \) design is that we can implement it as an internal embedded domain-specific language (EDSL) in Dotty: i.e., we can reuse the existing syntax and type system of Dotty, to define: (1) typed communication channels, (2) dedicated methods to render the \( \lambda T \) concurrency primitives (send, recv, \( \llbracket \), \( \rrbracket \), end), and (3) dedicated classes to render their types (of\[\ldots\], if\[\ldots\], \( \{\ldots\}, \) nil), including the well-foundedness and subtyping constraints illustrated in Fig. 4. The result is a software library called Effpi. As usual for internal language embeddings, the Effpi API does not directly cause side-effects: e.g., calling \( \text{receive}(c)\{x \Rightarrow P\} \) does not cause an input from channel \( c \). Instead, the receive method returns an object of type \( \text{In}\[\ldots\] \) (corresponding to \( \text{i}[\ldots] \) in Def. 3.1), which describes the act of using \( c \) to receive a value \( v \), and continues as \( P[v/X] \). Such objects are executed by the Effpi interpreter, according to the \( \lambda T \) semantics (Def. 2.4).

Effpi programs look like the code on the right (which is \( \text{ponger} \) from Ex. 2.2): they follow the \( \lambda T \) syntax. Also, types are rendered isomorphically: the type \( \text{"x"} \) in \( \lambda T \) is rendered as \( \text{"x".type} \) in Dotty, and dependent function types become:

\[
\Pi(x:T)\text{\texttt{o}}[y\cdot x\cdot T'] \Rightarrow (x:T) \Rightarrow \text{Out}[y\cdot \text{type}, x\cdot \text{type}, T']
\]

Thus, the Scala compiler can check the program syntax (§2) and perform type checking (§3), ensuring type safety (Thm. 3.6). Dotty also supports (local) type inference.

For better usability, Effpi also provides some extensions over \( \lambda T \), like buffered channels, and a sequencing operator "\( >\)" (see above, and in Fig. 1). Moreover, Effpi simplifies the definition and composition of types-as-protocols by leveraging Dotty’s type aliases. E.g., the type of two parallel processes sending an Integer on a same channel can be defined as \( U > \) - notice how \( T \) is reused, passing \( U \)'s parameter. Also notice how the type of \( f \)'s argument \((x). type \) is passed to \( U \), and then to \( T \): consequently, the type of \( f \) expands into \( \text{Par}[\text{Out}[x\cdot.type, \text{Int}], \text{Out}[x\cdot.type, \text{Int}]] \).

To guide Effpi’s design, we implemented the full “payment with audit” use case from the experimental “session” extension for Akka Typed [38] (cf. §1, code snippet in Fig. 1).

An efficient Effpi interpreter For performance and scalability reasons, many distributed programming toolkits (such as Go, Erlang, and Akka) schedule a (potentially very high) number of logical processes on a limited number of executor threads (e.g., one per CPU core). We follow a similar approach for the Effpi interpreter, leveraging the fact that, in Effpi programs as in \( \lambda T \), input/output actions and their continuations are represented by \( \lambda \)-terms (closures), that can be easily stored away (e.g., when waiting for an input from a channel), and executed later (e.g., when the desired input becomes available). Thus, we implemented a non-preemptive scheduling system partly inspired by Akka dispatchers [44], with a notable difference: in Effpi, processes yield control (and can be suspended) both when waiting for inputs (as in Akka), and also when sending outputs; this feature requires some sophistication in the scheduling system.

**Actor-based API** On top of the \( \lambda T \) EDSL, Effpi provides a simplified actor-based API [24], in a flavour similar to Akka Typed [46, 47] (i.e., actors have typed mailboxes and
ActorReferences): see Fig. 1. This API models an actor $A$ with mailbox of type $T$, with the intuition in Remark 2.3:

- $A$ is a process with a unique, implicit input channel $m$, of type $c_i[T]$ (Def. 3.1). Hence, $A$ can only use $m$ to receive messages of type $T$ — i.e., $m$ is $A$’s mailbox;
- $A$ receives $T$-typed messages by calling read — which is syntactic sugar for recv($m$, ...) (see Fig. 1, and notice that the input channel $m$ is left implicit);
- other processes/actors can send messages to $A$ through its ActorReference $r$ — which is just the output endpoint of its channel/mailbox $m$. The type of $r$ is $c_i[T]$ (Def. 3.1): it only allows to send messages of type $T$.

To this purpose, Effpi uses Dotty’s implicit function types [54]: i.e., type ActorReference [...] in Fig. 1 hides an input channel.

Type-level model checking The implementation details discussed thus far cover the $\lambda^{\tau}$, syntax, semantics, and typing — i.e., §2 and §3. The type-level analysis presented in §4 goes beyond the capabilities of the Dotty compiler; hence, we implement it as a Dotty compiler plugin (i.e., a compiler phase [56]) accessing the typed program AST. The plugin looks for methods annotated with "@effpi.verifier.verify":

```plaintext
@effpi.verifier.verify(\phi)

def f(x:\cdots, y:\cdots): T = ...
```

Such annotations ask to check if a program of type $T$ satisfies $\phi$, which is a conjunction/disjunctions of the properties from Fig. 7 (left). Note that $T$ can refer to the parameters $x,y$... of $f$, and it can be either written by programmers, or inferred by Dotty. Then, the plugin:

1. tries to convert $T$ into a $\lambda^{\tau}$ type $T'$, as per Def. 3.1;
2. checks if $T \models \phi'$ holds — where $\phi'$ is the companion formula of $\phi$ in Fig. 7 (right). This step uses the mCRL2 model checker [22]: we encode $T$ into an mCRL2 process, and check if $\phi'$ holds;
3. returns an error (located at the code annotation) if steps 1 or 2 fail. Otherwise, the compilation proceeds.

When compilation succeeds, any program of return type $T$ (including $f$ above) enjoys the property $\phi$ at run-time, by Thm. 4.10. This works both when $f$ is implemented, and when it is an unimplemented stub (i.e., when $f$ is defined as "???" in Dotty). This allows to compose the types/protocols of multiple services, and verify their interactions, even without their full implementation. E.g., consider Ex. 2.2, 3.3, and 4.11: a programmer implementing ponger (code above) in Effpi can (a) annotate the method ponger to verify that it is responsive (Fig. 7(6)), and/or (b) annotate an unimplemented stub def $f'(\cdots): T' = ???$ with type $T'$ matching $T_{pp}$ (Ex. 3.3), to verify that if ponger interacts with any implementation of type $T_{ping}$, then ponger’s self channel is used for output (Fig. 7(3)). Also, a programmer can annotate

3To obtain an mCRL2 encoding of $T$ with semantics adhering to Def. 4.2, we use the encoding into CCS (without restriction) mentioned after Lemma 4.7.

Known limitations The implementation of our verification approach, outlined above, has three main limitations.

1. It does not check purity of annotated code: such checks are unsupported in Dotty, and in most programming languages. Hence, programmers must ensure that all functions invoked from their $\text{Effpi}$ code eventually return a value — otherwise, liveness properties might not hold at run-time (cf. condition (c1) in §4).
2. It does not verify processes with unbounded parallel components (i.e., with parallel composition under recursion); hence, it rejects types having $p[\cdots]$ under $\mu T$. This does not impact the examples in this paper.
3. It uses iso-recursive types [57, Ch. 21] because, unlike $\lambda^{\tau}$ (Def. 3.2), Dotty does not have equi-recursive types.

Limitations 1 and 3 might be avoided by implementing $\lambda^{\tau}$ as a new programming language. However, our Dotty embedding is simpler, and lets Effpi programs access methods and data from any library on the JVM: e.g., Effpi actors/processes can communicate over a network (via Akka Remoting [45]), and with Akka Typed actors.

5.2 Evaluation From §5.1, two factors can hamper Effpi: (1) the run-time impact of its interpreter (speed and memory usage); (2) the verification time of the properties in Fig. 7. We evaluate both.

Run-time benchmarks We adopted a set of benchmarks from the Savina suite [28], with diverse interaction patterns:

- chameneos: n actors (“chameneos”) connect to a central broker, who picks pairs and sends them their respective ActorReferences, so they can interact peer-to-peer [31];
- counting: actor $A$ sends $n$ numbers to $B$, who adds them;
- fork-join — creation (FJ-C): creation of $n$ new actors, who signal their readiness to interact;
- fork-join — throughput (FJ-T): creation of $n$ new actors, and transmission of a sequence of messages to each.

- ping-pong: $n$ pairs of actors exchange requests-responses;
- ring: $n$ actors, connected in a ring, pass each other a token;
- streaming ring: similar to ring, but passing $m$ tokens consecutively (i.e., at most $m$ actors can be active at once).

For all benchmarks, we performed two measurements:

- performance vs. size: how long it takes for the benchmark to complete, depending on the size (i.e., the number of actors, or the number of messages being sent/received);

1This is because mCRL2 checks formulas of the branching-time $\mu$-calculus, on finite-state systems. We are not aware of model checkers focused on the linear-time $\mu$-calculus, and supporting infinite-state systems.
- memory vs. size: how many times the JVM garbage collector runs, depending on the size of the benchmark — and also the maximum memory used before collection.

The results are in Fig. 8: we compare two instances of the Effp1 runtime (with two scheduling policies: “default” and “channel FSM”) against Akka, with default setup. Our approach appears viable: Effp1 is a research prototype, and still, its performance is not too far from Akka. The negative exception is “chameneos” (Effp1 is ~2× slower); the positive exceptions are fork-join throughput (Effp1 is ~2× faster), and the ring variants (Akka has exponential slowdown).

Model checking benchmarks We evaluated the “extreme cases”: the time needed to verify formulas in Fig. 7 on protocols with a large number of states — obtained, e.g., by enlarging the examples in the paper (e.g., composing many parallel ping-pong pairs), aiming at state space explosion. The results are in Fig. 9. Our model checking approach appears viable: it can provide (quasi)real-time verification results, suitable for interactive error reporting on an IDE. Still, model checking performance depends on the size of the model, and on the formula being verified. As expected, our measurements show that verification becomes slower when models are expanded by adding more parallel components, and thus enlarging the state space; they also highlight that some properties (e.g., our mCRL2 translations of “forwarding” and “responsive”) are particularly sensitive to the model size.

6 Conclusion and Related Work

We presented a new approach to developing message-passing programs, and verifying their run-time properties. Its cornerstone is a new blend of behavioural-dependent function types, enabling program verification via type-level model checking.

Behavioural types with LTS semantics have been studied in many works [3]: the idea dates back to [53] (for Concurrent ML); type-based verification of temporal logic properties was addressed in [26, 27] (for the π-calculus); recent applications include, e.g., the verification of Go programs [41, 42]. Our key insight is to infuse dependent function types, in order to (1) connect a type variable \( x \) to a process variable \( x \), and (2) gain a form of type-level substitution (Def. 3.1). Item (2), in particular, is not present in previous work; we take advantage of it to compose protocols (Ex. 3.3) and precisely track channel passing and use (Ex. 4.3). Thus, we can verify safety and liveness properties (Fig. 7) while supporting: (1) channel passing, thus covering a core pattern of actor-based programming (Ex. 2.2, Remark 2.3, Ex. 4.11, Fig. 1), and (2) higher-order processes that send/receive mobile code, thus covering an important feature of modern programming toolkits (Ex. 3.4, 4.11). Further, our theory is designed for language embedding: we implemented it in Dotty, and our evaluation supports the viability of the approach (§5).

A form of type/channel dependency related to ours is in [23, 69, 70]: their types depend on process channels, and they...
### Verifying message-passing programs with dependent behavioural types

<table>
<thead>
<tr>
<th>states</th>
<th>deadlock-free</th>
<th>ev-usage</th>
<th>forwarding</th>
<th>non-usage</th>
<th>reactive</th>
<th>responsive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pay &amp; audit + 8 clients</td>
<td>3328</td>
<td>true (0.05 ± 0.38%)</td>
<td>false (6.26 ± 4.16%)</td>
<td>false (0.02 ± 0.66%)</td>
<td>true (1.01 ± 3.95%)</td>
<td>true (15.40 ± 6.57%)</td>
</tr>
<tr>
<td>Pay &amp; audit + 10 clients</td>
<td>13312</td>
<td>true (0.06 ± 1.65%)</td>
<td>false (0.02 ± 1.41%)</td>
<td>false (0.02 ± 2.02%)</td>
<td>false (0.02 ± 1.91%)</td>
<td>false (0.02 ± 3.35%)</td>
</tr>
<tr>
<td>Pay &amp; audit + 12 clients</td>
<td>35248</td>
<td>true (0.07 ± 1.17%)</td>
<td>false (0.02 ± 1.35%)</td>
<td>false (0.02 ± 2.78%)</td>
<td>true (0.99 ± 2.89%)</td>
<td>true (345.22 ± 8.72%)</td>
</tr>
</tbody>
</table>

**Figure 9.** Behavioural property verification: outcome (true/false) and average time (seconds ± std. dev.). The number of states is approximated > 2×10^6 when the LTS is too big to fit in memory. (Intel i7 @ 3.60GHz, 16 GB RAM, mcrL2 201808.0, 30 runs)

### Future work

We will study \( \lambda^T \) embeddings in other programming languages — although only Dotty provides both subtyping and dependent function types. We will extend the supported properties in Fig. 7, and study how to improve their verification, along three directions: 1. increase speed, trying more mcrL2 options, and tools like LTSMv [32]; 2. support infinite-state systems, trying tools like Brc [30] (that does not cover the linear-time \( \mu \)-calculus in Def. 4.6, but is used e.g. in [14] to verify safety properties of actor programs); 3. introduce assume-guarantee reasoning for type-level model checking, inspired by [59]. The Effpi runtime system can be optimised: we will attempt its integration with Akka Dispatchers [44], and explore other (non-preemptive) scheduling strategies, e.g., work stealing [1, 5].

### Acknowledgements

Thanks to the anonymous reviewers for their remarks, to Sung-Shik Jongmans for his comments, and to Raymond Hu for testing the artifact. Work partially...
References


[34] Naoki Kobayashi. 2006. A New Type System for Deadlock-Free Processes. In CONCUR. https://doi.org/10.1109/18117949_16


Verifying message-passing programs with dependent behavioural types

Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2017. Fencing off go: liveness and safety for channel-based programming. In POPL. https://doi.org/10.1145/3093333.3099847

Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A static verification framework for message passing in Go using behavioural types. In ICFSE. https://doi.org/10.1145/3180153.3180157


