Verifying Message-Passing Programs with Dependent Behavioural Types

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
Concurrent and distributed programming is notoriously hard. Modern languages and toolkits ease this difficulty 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 $\lambda\pi$, 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 Effpi, offering a simplified actor-based API; (4) it enables an efficient runtime system for Effpi, 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 [42, 50]):

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 [50], 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), paymett 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 [46], and produce compilation errors when a program violates a desired protocol. However, the resulting experiments [41] had no rigorous grounding: although inspired by the session types theory [3, 26], the approach was informal, and the kind of assurances that it could provide are unclear. Still, the idea has intriguing potential: if realised, it would allow to check the payment specification above at compile-time.
(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 $\lambda^\pi_\leq$ 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 $\lambda^\pi_\leq$ calculus, at the basis of Effpi. §3 presents type system of $\lambda^\pi_\leq$. §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 $\lambda^\pi_\leq$ 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 [70] contains proofs and more material.

2 The $\lambda^\pi_\leq$-Calculus

The theoretical basis of our work is a $\lambda$-calculus extended with channels, input/output, and parallel composition, called $\lambda^\pi_\leq$. The “$\pi$” denotes both: (1) its use of dependent function types, that, together with subtyping $\leq$, are cornerstones of its typing system (§3); and (2) its connection with the $\pi$-calculus [54, 55, 63]. Indeed, $\lambda^\pi_\leq$ 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 $\lambda^\Pi_\Sigma$ is in Fig. 2. Elements of $\Sigma$ 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 $\lambda_.t$ for $\lambda x.x$.t, when $x \not\in fv(t)$.

The set of values $V$ includes booleans $\mathbb{B}$, channel instances $\mathbb{C}$, function abstraction, the unit $()$, and error. The terms (in $T$) can be variables (from $\Sigma$), values (from $V$), various standard constructs (negation $\neg t$, if/then/else, let binding, function application), and also channel creation $\text{chan}(\cdot)$, and process terms (from $\mathbb{P}$). The primitive $\text{chan}(\cdot)$ evaluates by returning a fresh channel instance from $\mathbb{C}$ — 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 $\text{send}(t, t', t'')$ (meaning: send $t'$ through $t$, and continue as $t''$), the input primitive $\text{recv}(t, t')$ (meaning: receive a value from $t$, and continue as $t'$), and the parallel composition $\parallel(t,t')$ (meaning: $t$ and $t'$ run concurrently, and can interact). $\lambda^\Pi_\Sigma$ can be routinely extended with, e.g., integers, strings, records, variants: we use them in examples.

**Example 2.2.** A ping-pong system in $\lambda^\Pi_\Sigma$ is written as:

let $\text{pinger} = \lambda$ self. $\text{pong}$.($\text{send}$)($\text{self}$, $\lambda_.\text{self}$. ($\text{recv}$)(self, $\lambda$. $\text{replyTo}$.($\text{send}$)(replyTo, "Hi!", $\lambda_.\text{self}.\text{recv}$).($\text{end}$))).($\text{end}$))

let $\text{sys} = \lambda y'. \lambda z'. (\text{pinger} y' z' \parallel \text{ponger} z')$

let $\text{main} = \lambda_. \text{let } y = \text{chan} \text{ in } \text{let } z = \text{chan} \text{ in } \text{sys } y z$

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

Note that in $\text{pinger}$ and $\text{ponger}$, the last argument of $\text{send}/\text{recv}$ 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, $\text{pinger} / \text{ponger}$ 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 $\lambda^\Pi_\Sigma$). Evaluation contexts $E$ and reduction $\rightarrow$ are illustrated in Fig. 3, where congruence $\equiv$ is defined as: $t_1 \parallel t_2 \equiv t_1 \parallel t_2 \parallel t_1 \parallel t_2 \parallel \text{end} \equiv \text{end} \parallel \text{end} \equiv \text{end}$, plus $\alpha$-conversion. We write $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$. We say "$t$ has an error" iff $t = E[\epsilon]\text{err}$ (for some $E$). We say "$t$ is safe" iff $\forall t' : t \rightarrow t'$ implies $t'$ has no error.

Def. 2.4 is a standard call-by-value semantics, with two rules for concurrency. [R-chanc] says that $\text{chan}(\cdot)$ returns a fresh channel instance; [R-Comm] says that the parallel composition $\text{send}(a, u, v_1) \parallel \text{recv}(a, v_2)$, where both sides operate on a same channel instance $a$, transfers the value $u$ on the receiver side, yielding $v_1 \parallel v_2 u$; hence, if $v_1$ and $v_2$ are function values, the process keeps running by applying $v_1 ()$ and $v_2 u$ — i.e. the sent value is substituted inside $v_2$. 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 $\mathbb{P}$ in Fig. 2 are safely composed by $\parallel$).

### 3 Type System

We now introduce the type system of $\lambda^\Pi_\Sigma$. Its design is reminiscent of the simply-typed $\lambda$-calculus, except that (1) we include union types and equi-recursive types, (2) we add types for channels and processes, and (3) allow types to contain variables from the term syntax (inspired by $\mathbb{D}_\mu$, the calculus behind Dotty [2]). The syntax of types is in Def. 3.1.

Notably, points (1) and (3) establish a similarity between $\lambda^\Pi_\Sigma$ and $\mathbb{F}_\Sigma$: (System $\mathbb{F}$ with subtyping [8]) equipped with equi-recursive types [32]. 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 $\mathbb{F}_\Sigma$, polymorphic types can depend on type variables in the environment; hence, we use mutually-defined judgements, akin to those of $\mathbb{F}_\Sigma$, to assess the validity of environments, types, subtyping, and typed terms (Def. 3.2).

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

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| $\text{bool}$ | $()$ | $\top$ | $\perp$ | $T \lor U$ | $\Pi (x : U) T$ | $\mu x . T$ | $x$
| $\epsilon^n(T)$ | $\epsilon^0(T)$ |
| $\text{proc}$ | $\text{nil}$ | $\text{nil}$ | $\text{nil}$ | $\text{nil}$ | $\text{nil}$ | $\text{nil}$ | $\text{nil}$ |
```

Free/bound variables are defined as usual. We write $U(\{x\}$ for the type obtained from $U$ by replacing its free occurrences of $x$ with $S$. If $T = \Pi (x : U) T$, then $TS$ stands for $U(\{x\}$.

We write $\Pi(T)$ for $\Pi (x : (\cdot)) T$ if $x \not\in fv(T)$, and distinguish recursion variables as $t, t', \ldots$ (i.e., we write $\mu t . T$). We write $\overline{T}$ for an $n$-tuple $T_1, \ldots, T_n$, and $T \in U$ if $T$ occurs in $U$.

The relation $\equiv$ is the smallest congruence such that:

- $T \lor U \equiv U \lor T$
- $T \lor (T \lor V) \equiv (T \lor V) \lor U$
- $\mu T \equiv T[\mu T/x]$
- $\Pi(T, U) \equiv \Pi(U, T)\Pi(U, T)$
- $\Pi(U, T)$
- $\Pi(U, T)$
- $\Pi(U, T)$

The first row of productions in Def. 3.1 includes booleans, the unit type $(),$ top/bottom types $\top/\perp$, the union type $T \lor U$, the dependent function type $\Pi(x : U) T$ and the recursive type $\mu x . T$ (they both bind $x$ with scope $T$), and variables $x$ (from the set $\mathcal{X}$ in Def. 2.1): the underlining is a visual clue to better distinguish $x$ used in a type, from $x$ used in a $\lambda^\Pi_\Sigma$ term.

The second row of Def. 3.1 formalises channel types: $\epsilon^n[T]$ denotes a channel allowing to input or output values of type $T$; instead, $\epsilon^0[T]$ only allows for input, and $\epsilon^0[T]$ for output.
The third row of Def. 3.1 formalises process types. The generic process type \( \text{proc} \) denotes any process term; \( \text{nil} \) denotes a terminated process; the output type \( o[S,T,U] \) denotes a process that sends a \( T \)-typed value on an \( S \)-typed channel, and continues as \( U \); the input type \( i[S,T,U] \) denotes a process that receives a value from an \( S \)-typed channel and continues as \( T \); the parallel type \( p[T,U] \) denotes the parallel composition of two processes (of types \( T \) and \( U \)).

### Definition 3.2.
These judgements are formalised in Fig. 4:

- \( \Gamma \vdash \text{env} \) \( \Gamma \) is a valid typing environment
- \( \Gamma \vdash T \text{ type} \) \( T \) is a valid type in \( \Gamma \)
- \( \Gamma \vdash \overline{T} \text{ type} \) holds iff \( \forall U \in \overline{T} : \Gamma \vdash U \text{ type} \)
- \( \Gamma \vdash \pi \text{ type} \) \( \pi \) is a valid process type in \( \Gamma \)
- \( \Gamma \vdash \overline{\pi} \text{ type} \) holds iff \( \forall U \in \overline{\pi} : \Gamma \vdash U \pi \text{-type} \)
- \( \Gamma \vdash \overline{\pi} \text{ type} \) holds if \( \Gamma \vdash T \pi \text{ type} \) or \( \Gamma \vdash \overline{T} \pi \text{ type} \)
- \( \Gamma \vdash \pi U \leq U \) \( T \) is a valid channel type in \( \Gamma \), if \( \Gamma \vdash T,U \pi \text{-type} \)
- \( \Gamma \vdash t : T \) \( t \) has type \( T \) in \( \Gamma \)

A typing environment \( \Gamma \) maps variables (from \( \overline{X} \) in Def. 2.1) to types; the order of the entries of \( \Gamma \) is immaterial. All judgements in Fig. 4 are inductive, except \( \text{except} \), that is coinductive (hence the double inference lines). Crucially, in Fig. 4 we have two valid type judgements, for two kinds of types: \( \Gamma \vdash T \text{ type} \) and \( \Gamma \vdash \overline{T} \pi \text{-type} \). The former is standard (except for rule \( \overline{\tau} \text{-c} \), for valid channel types); 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 \( \overline{T} \text{-c} \)), but not to a \( \pi \)-type; this also means that function arguments cannot be \( \pi \)-typed. Still, in a function type \( \Pi(x:U)U \), the return type \( U \) can be a \( \pi \)-type (rule \( \overline{T} \pi \text{-c} \)); i.e., it is possible to define abstract process types (cf. Ex. 3.3 and 3.4 later). Rules \( \overline{T} \pi \text{-c} \) and \( \overline{\pi} \text{-c} \) are based on \( [32, \S 2] \), and require recursive types to be contractive: e.g., \( \mu t_1.t_2.\ldots.t_n \), \( \forall t \in \overline{U} \) is not a type; clause \( "\overline{x} \notin \text{fv}(T)" \) means that \( \overline{x} \) is not bound in negative position in \( T \), as in \( F_c \). (Details: [70]). Recursion is handled by \( \overline{\text{let}} \)-in \( \text{let } x = t \text{ in } t' \), term \( t \) can refer to \( x \). Rule \( \overline{\text{let}} \), based on [9], ensures decidability of subtyping [32, \S 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 discuss the main judgements.

### Variables, types, subtyping, and dependencies
The environment \( \Gamma = x : T \) assigns type \( T \) to variable \( x \). Hence, by rule \( \overline{T} \pi \text{-c} \), the type \( \overline{x} \) is valid in \( \Gamma \); and indeed, by rule \( \overline{T} \pi \text{-c} \), we can infer \( \Gamma \vdash x : T \), i.e., the term \( x \) has type \( x \). Intuitively, this means that \( x \) is the "most precise" type for term \( x \); this is formally supported by the subtyping rule \( \overline{\tau} \text{-c} \), 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 subordination (rule \( \overline{\text{sub}} \)), as shown here. Since \( \overline{x} \) is the smallest type for term \( x \), the judgement \( \Gamma \vdash t : T \) conveys that \( t \) should be "something" that evaluates to \( x \), e.g., \( t = x \) or \( t = \text{if} \overline{tt} \text{ then } \overline{tt} \text{ else } \overline{tt} \); similarly, the dependent function type \( \Pi(x: \text{bool})x \) 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 \( \overline{\text{chan}} \), a (type-annotated) term \( \text{chan}(T) \) has type \( c^o[T] \). Rule \( \overline{\text{c}} \) is similar, for channel instances. By \( \overline{\text{end}} \), process \( \text{end} \) has type \( \text{nil} \).

By \( \overline{\text{let}} \), both sub-terms of \( \overline{t_1, t_2} \) are \( \pi \)-typed. By \( \overline{\text{send}} \), \( \overline{\text{recv}}(t_1, t_2, t_3) \) has type \( o[S,T,U] \), under the validity constraints of rule \( \overline{\pi} \text{-c} \). Hence, \( \overline{t_1} \) has a channel type for sending values of type \( t_2 \), and \( t_2 \) (the term being sent) must have type \( T \); also, \( t_2 \)'s type must be \( U = \Pi(U') \) (for a \( \pi \)-type \( U' \)): i.e., \( t_3 \) is a process thunk, run by applying \( t_3 \). By \( \overline{\text{recv}} \), \( \overline{\text{recv}}(t_1, t_2) \) has type \( i[S,T] \), which is well-formed under rule \( \overline{\pi} \text{-c} \). Hence, the sub-term \( t_1 \) must have a channel type with input \( U \), while \( t_2 \) must be an abstract process of type \( T = \Pi(x : U')U' \), with \( T' \pi \text{-type} \). Crucially, by rule \( \overline{\pi} \text{-c} \), we have \( \overline{\Gamma \vdash T \leq U} \); hence, it is safe to receive a value \( u \) from \( t_1 \), and apply \( t_2, u \) 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:

- \( \overline{pinger} : T_{\overline{\text{pinger}}} = \Pi(x : c^o[\text{str}]) \Pi(\overline{\text{pongc}}c^o[\text{str}])\overline{x} \overline{\text{pong}}.\overline{x}.\overline{\text{pong}}(x)\overline{x}\overline{\text{pong}}(x)\overline{x} \)
- \( \overline{ponger} : T_{\overline{\text{ponger}}} = \Pi(x : c^o[\text{str}])\overline{x} \overline{\text{pong}}.\overline{x}.\overline{\text{pong}}(x)\overline{x} \)
- \( \overline{sys} : T_{\overline{\text{sys}}} = \Pi(x : c^o[\text{str}])\Pi(\overline{\text{sys}}c^o[\text{str}])\Pi(\overline{\text{sys}}c^o[\text{str}])\Pi(\overline{\text{sys}}c^o[\text{str}])\overline{\text{sys}}(x)\overline{\text{sys}}(x)\overline{\text{sys}}(x)\overline{\text{sys}}(x)\overline{\text{sys}}(x) \)

Notice how \( T_{\overline{\text{pp}}} \) captures the ping/pong composition of \( \overline{sys} \), preserving its channel topology: the type-level applications
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash \text{send}(t_1, t_2, t_3) : \text{os}(T)$</td>
<td>$\Gamma = \forall x : \text{cap}(T), \exists U : \text{cap}(T'). \forall t : \text{cap}(T) \cdot \Gamma$; $t_1, t_2, t_3 : \text{cap}(T)$; $x : \text{cap}(T')$</td>
</tr>
</tbody>
</table>
let \( \text{srv} = \lambda c.m. \text{m.out} \).

let \( z_1 = \text{chan()} \) in
\[ \text{recv(cm, } \lambda p. ( p \parallel z_1 \text{ out} \parallel \text{prod}_1 z_1 \parallel \text{prod}_2 z_2 ) \) \]

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

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

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

\[ \text{send}(x, t, \lambda \_ \_ \_ \text{end}) \parallel \text{srv} \times \text{out} \quad (\text{client sends } t \text{ to server, via } x) \]

is typable in \( \Gamma \) only if \( \Gamma \vdash x : c^{\Pi}[T_m] \), implying \( \Gamma \vdash t : T_m \).

We can replace \text{proc} with a more precise type. If \( U_1/U_2 \) are types of \( \Pi \times \Pi \), the \text{recv}(...) sub-term of \text{srv} has type:

\[ T'_{\text{srv}} = \Pi(cm, \Pi(p : T_m) p \downarrow T_m z_1 z_2 \downarrow \text{out}, U_1 z_1, U_2 z_2) \]

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

**Subtyping, subsumption, and private channels**

The subtyping rules in Fig. 4 are standard (based on \( F_c \) [8, 32]) except the highlighted ones. By rule \( \ll \), subtyping for channel types is covariant for inputs, and contravariant for outputs, as expected [61]: intuitively, channels with smaller types can be used more liberally. Rule \( \ll \times \text{proc} \) says that \text{proc} is the top type for \( \Pi \times \Pi \)-types. Rules \( \ll p/q \ll \ll p/p \) say that types for input/output/parallel processes are covariant in all parameters.

As usual, supertyping / subsumption (rule \( \ll \) caters for Liskov & Wing’s substitution principle [51]: a smaller object can replace a larger one. Crucially, in our theory, supertyping also allows to **drop information when typing private channels**.

This is shown in Ex. 3.5: via supertyping, we do not precisely track how private (i.e., bound) channels are used. This information loss is key to type Turing-powerful \( \lambda \Pi \leq \text{terms with a non-Turing-complete type language, for the results in §4.}

**Example 3.5** (Subtyping, binding, and precision loss). Let:

\[ t_1 = \text{send}(x, 42, \lambda \_ \_ \text{end}) \parallel \text{recv}(x, \lambda \_ \_ \text{end}) \]
\[ t_2 = (\lambda z = \text{chan()} \in \text{send}(z, 42, \lambda \_ \_ \text{end}) \parallel \text{recv}(x, \lambda \_ \_ \text{end}) \]

\[ T_1 = p \left[ \begin{array}{l} \dot{\text{in}}, \Pi(\text{nil}) \end{array} \right], \left[ \begin{array}{l} \dot{\text{in}}, \Pi(\text{nil}) \end{array} \right] \]
\[ T_2 = p \left[ \begin{array}{l} \dot{\text{in}}, \Pi(\text{nil}) \end{array} \right], \left[ \begin{array}{l} \dot{\text{in}}, \Pi(\text{nil}) \end{array} \right] \]

Letting \( \Gamma = x : c^{\Pi}[\text{int}] \), we have \( \Gamma \vdash x \leq c^{\Pi}[\text{int}] \) and \( \Gamma \vdash T_1 \leq T_2 \). For \( t_1 \), we have both \( \Gamma \vdash t_1 : T_1 \) and \( \Gamma \vdash t_1 : T_2 \) (by \( \ll \) ): in the first judgement, \( T_1 \) precisely captures that \( x \) is used to send/receive an integer; instead, in the second judgement, \( T_2 \) is less accurate, and says that some term with type \( c^{\Pi}[\text{int}] \) is used to send, while \( x \) is used to receive.

We also have \( \Gamma \vdash t_2 : T_2 \); and notably, since \( z \) is bound in the "let..." subterm of \( t_2 \), it cannot appear in the type: i.e., we cannot write a more accurate type for \( t_2 \). This is due to rule \( \ll \) (Fig. 4): since \( z \) is bound by \( \text{let...} \), its occurrence in \text{send}(...) is typed by a supertype of \( z \) that is suitable for both \( x \) and \( \text{chan()} \) — in this case, \( c^{\Pi}[\text{int}] \). Specifically:

\[ \Gamma \vdash c^{\Pi}[\text{int}] \leq c^{\Pi}[\text{int}] \]
\[ \Gamma, \triangleright c^{\Pi}[\text{int}] \vdash \text{chan()} : c^{\Pi}[\text{int}] \]
\[ \Gamma, \triangleright c^{\Pi}[\text{int}] \vdash \text{send}(z, 42, \lambda \_ \_ \text{end}) : \Pi z, \Pi(\text{nil}) \]
\[ \Gamma \vdash \text{send}(z, 42, \lambda \_ \_ \text{end}) : \Pi z, \Pi(\text{nil}) \]

Typing guarantees that well-typed terms never go wrong.

**Theorem 3.6** (Type safety). If \( \Gamma \vdash t : T \), then \( t \) is safe.

Thm. 3.6 follows by: \( \Gamma \vdash t : T \) and \( t \to t' \) implies \( \exists T' \) such that \( \Gamma \vdash t' : T' \) — i.e., typed terms only reduce to typed terms, without (untypable) \text{err} subterms. In §4, we study how \( T \) and \( T' \) are related, and how they constrain \( t \)'s behaviour.

### 4 Type-Level Model Checking

Our typing discipline 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 \[73]\): \( \phi \) could be, e.g., a safety property \( \square (\neg \phi') \) ("\( \phi' \) 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 process(es) \( t \) 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 surmount 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 \( T_j \) in Ex. 3.5 uses \( \chi \), 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., \( t'_1 = \text{let } x = \text{chan()} \text{ in } t_1 \) — but \( t'_1 \)'s type cannot mention \( \chi \), 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 types transfer to processes.
**Definition 4.1** (Labelled semantics of open typed terms). When \( \Gamma \vdash t : T \) (for any \( \Gamma, t, T \)), the judgements \( \Gamma \vdash t \Downarrow, t' \) and \( \Gamma \vdash t \Downarrow^\ast, t' \) are inductively defined in Fig. 5.

Unlike Def. 2.4, Def. 4.1 lets an open term like \( \sim x \) reduce, by non-deterministically instantiating \( x \) to \( tt \) or \( ff \); the assumption \( \Gamma \vdash \sim x : T \) ensures that \( x \) is a boolean. Rule \([\text{SR} \rightarrow t]\) inherits “concrete” reductions from Def. 2.4: if \( t \rightarrow t' \) is induced by base rule \([s]\), the transition label is \( \tau\{s\} \). Rules \([\text{SR-send}] / [\text{SR-recv}] \) send/receive a value/value \( w' \) using a (channel-typed) value/value \( w \). Note that in \([\text{SR-recv}] \), \( w' \) is any value/value of type \( T_i \), which is the input type of \( x \) (in \( \pi \)-calculus jargon, it is an *early* semantics [63]). Rule \([\text{SR-Comm}] \) lets processes exchange a payload \( w' \) via a channel/value \( w \), recording \( w \) in the transition label. Rule \([\text{SR-x}] \) “applies” \( x \) by instantiating it with any suitably-typed \( \lambda y.v \) (i.e., \( \lambda y.v \) must be a function that, when applied to \( w \), yields a term \( v(w/y) \) of type \( T_j \); it also records \( x \) in the transition label. Rule \([\text{SR-λ}] \) applies a function to a variable \( x \), with the expected substitution. Rule \([\text{SR-Σ}] \) propagates transitions through contexts, unless labels refer to bound variables. Finally, \( \Gamma \vdash t \Downarrow^\ast, t' \) holds when \( t \) reaches \( t' \) via a finite sequence of internal moves excluding *interaction*: i.e., labels \( w(w'), \overline{w}(w'), \tau[w], \) and \( \tau[R-Comm] \) are forbidden.

*Type semantics*  
We now equip our types with labelled transition semantics (Def. 4.2): this is not unusual for *behavioural* type systems in \( \pi \)-calculus literature [3, 30] — but our novel use of type variables, and dependent function types, yields new capabilities, and requires some sophistication.

The type processes should mimic the semantics of typed processes. Hence, take \( T_1 \) and \( t_1 \) from Ex. 3.5: we want \( T_1 \) to reduce, simulating the term reduction \( \Gamma \vdash t_1 \Downarrow^\ast, \text{end} \end{flalign} \). This suggests that a type like \( p[\omega, x, \ldots, i[x, \ldots]] \) should reduce with a communication on \( x \). But consider \( T_2 \) in Ex. 3.5: \( T_2 \) also types \( t_1 \), hence it should also simulate \( t_1 \)'s reduction — i.e., a type like \( p[\omega, c[x^\omega], i[x, \ldots]] \) should reduce, too. In general, we want \( p[\omega, S, i[T, \ldots]] \) to reduce if \( S \) and \( T \) “might interact”, i.e., they could type a same channel/value: we formalise this idea as \( \Gamma \vdash S \bowtie T \) in Def. 4.2.

**Definition 4.2** (Type semantics). Let \( S \bowtie T \) be the greatest subtype of \( S \) and \( T \) in \( \Gamma \), up-to \( \equiv \) (Def. 3.1). The judgement \( \Gamma \vdash S \bowtie T \) (read “\( S \) and \( T \) might interact in \( \Gamma \)”) is:

\[
\Gamma \not\vdash S \bowtie T \quad \text{if} \quad \Gamma \vdash S \bowtie T
\]

A type reduction context \( E \) is inductively defined as:

\[
\begin{align*}
&[] \quad | \quad o(E, T, U) \quad | \quad o(S, E, U) \quad | \quad o(S, T, E) \quad | \quad i(E, T) \quad | \quad i(S, E) \quad | \quad p[E, T] \\
\end{align*}
\]

Judgements \( \Gamma \vdash T \Downarrow, T' \) and \( \Gamma \vdash T \Downarrow^\ast, T' \) are in Fig. 6.

By Def. 4.2, \( \Gamma \vdash S \bowtie T' \) holds when \( S \) and \( T' \) have a common subtype besides \( \perp \), i.e., they might type a same term in \( \Gamma \), via rule \( \{\bowtie\} \). The judgement \( \Gamma \vdash T \Downarrow^\ast, T' \) says that \( T \cup U \) can reduce to \( T \) or \( U \), firing label \( \tau[V] \). Rule \([\text{R-α}] \) reduces an output type, recording the used channel type \( S \) and payload \( T \) in the transition label. Rule \([\text{R-β}] \) is similar for input types, recording the payload \( T' \). We have two communication rules:

- \([\text{R-in}] \) fires when, in \( p[U, U'] \), there might be an interaction with a type variable \( x \) as payload. Note that, by \([\text{R-β}] \), \( x \) is bound in \( U' \), hence it can appear in its future transitions. The rule yields a transition label \( \tau[S, S'] \), recording which channel types were used;
- \([\text{R-in}] \) is similar, but fires if the payload \( T \) is not a variable.

Finally, \( \Gamma \vdash T \Downarrow^\ast, T' \) holds if \( T \) reaches \( T' \) via a finite sequence of internal choices \( \tau[V] \).

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

\[
\begin{align*}
\Gamma &= \tau[x]\text{[str]} \vdash \tau[\omega]\text{[str]} \quad t = \text{ysyz} \\
T &= \text{ppyz} = \{ p[y, \Pi(\text{replyTo}[\text{str}])], \{ z, \Pi(\text{replyTo}[\text{str}]) \} \} \\
\end{align*}
\]

By Def. 3.2, we have \( \Gamma \vdash t : T \). By Def. 4.1, we have:

\[
\begin{align*}
\Gamma \vdash t \Downarrow^\ast &\Downarrow^\ast, \text{end} \end{flalign} \)
\]

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

\[
\begin{align*}
\Gamma \vdash T \Downarrow^\ast &\Downarrow^\ast \Downarrow^\ast, \text{end} \end{flalign} \)
\]

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

**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 \( \Gamma \vdash t : T \). If \( \Gamma \vdash T \) type, then \( \Gamma \vdash t \Downarrow^\ast, t' \) implies \( \Gamma \vdash t' : T \). Otherwise, when \( \Gamma \vdash T \) \( π \)-type, we have:

1. \( \Gamma \vdash t \Downarrow^\ast, t' \) with \( \tau^\ast(\alpha) \) (Fig. 5) implies \( \Gamma \vdash t' : T \);
2. \( \Gamma \vdash t \Downarrow^\ast, t' \) and \( \alpha \in \{ \text{∃}(w), \text{x}(w), \tau[x], \tau[R-Comm] \} \) implies one of:
   a. \( \Gamma \vdash t' : T \) and \( \text{proc} \in T \);
   b. \( \alpha = \text{x}(w) \) and \( \exists S, U, T' : \Gamma \vdash x : S, w : U, t' : T' \) and \( \Gamma \vdash T \Downarrow^\ast, \text{end} \)
   c. \( \alpha = \text{x}(w) \) and \( \exists S, U, T' : \Gamma \vdash x : S, w : U, t' : T' \) and \( \Gamma \vdash T \Downarrow^\ast, \text{end} \)
   d. \( \alpha = \tau[x] \) and \( \exists S, S', T' : \Gamma \vdash x : S, x : S', t' : T' \) and \( \Gamma \vdash T \Downarrow^\ast, \text{end} \)
   e. \( \alpha = \tau[R-Comm] \) and \( \exists S, S', T' : \{ S, S' \} \not\subseteq \text{x}, \Gamma \vdash t' : T' \) and \( \Gamma \vdash T \Downarrow^\ast, \text{end} \)

Assume \( \Gamma \vdash t : T \), with \( t \) reducing to \( t' \): Thm 4.4 says that when the reduction is caused by the functional fragment of \( \lambda^2_S \) (hypothesis \( \Gamma \vdash T \) type, or case 1), then \( t' \) has the same
Theorem 4.5 (Type fidelity). Within productive $\lambda^T$-types, assume $\Gamma \vdash t : T$ and $\Gamma \vdash \Pi$-type $\Pi$-type. Then:

1. $\Gamma \vdash T \frac{S(U)}{\Pi \to T}$ implies $\exists w, t' : \Gamma \vdash w : U, \Pi' : T$ and $\Gamma \vdash t \frac{\tau^*_{\Pi}}{S(w)} \Pi \to T$;
2. $\Gamma \vdash T \frac{\bar{s}(U)}{\Pi' \to T}$ implies $\forall w : \Pi \vdash w : U$, then $\exists \Pi' : T$ is productive by hypothesis. By item 4, if $T$ can make a choice ($\forall T$), 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 [20, §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 $\textbf{Act}$ ranged over by $\alpha$, the linear-time $\mu$-calculus formulas are defined as follows (where $\mathbb{A}$ is a subset of $\textbf{Act}$):

- **Basic formulas**: $\phi : \top \vdash \bot \vdash \phi \land \phi \vdash (\alpha)\phi \vdash vZ.\phi$
- **Derived formulas**: $\phi \vdash (A)\phi \vdash (\neg A)\phi \vdash \phi \cup \phi \vdash \phi \mu Z.\phi$

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 α, and then φ holds"), or greatest fixed point \(vZ, \phi\). Basic formulas are enough [6, 73] to derive true/false (accept any/no sequence of actions), disjunction, implication, least fixed points \(\mu Z, \phi; (A)\phi\) accepts sequences that start with any \(a \in A\), then satisfy \(\phi\); dually, \((-\lambda)\phi\) requires \(a \in \text{Act}\setminus A\). We also derive usual temporal formulas \(\phi_1 U \phi_2 (\text{"} \phi_1 \text{"} holds, until \(\phi_2 \text{"} eventually holds\)), \(\Box \phi\) (\(\phi\) 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: [70])

We can decide \(\phi\) on a guarded type \(T\), as shown in Lemma 4.7. Here, we instantiate \(\text{Act}\) (Def. 4.6) as \(A_T(T)\), which is the set of labels fired along \(T\)’s transitions in \(\Gamma\), (Def. 4.2); notably, \(A_T(T)\) is finite and syntactically determined. (Details: [70])

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

Lemma 4.7 holds since guarded \(\pi\)-types are encodable in CCS without restriction [53], then in Petri nets [22, §4.1], for which linear-time \(\mu\)-calculus is decidable [20]. Notably, Lemma 4.7 covers infinite-state types (with \(p[...]|[...

5 Implementation and Evaluation

We designed \(\lambda^\pi\) to leverage subtyping and dependent function types, with a formulation close to (a fragment of) Dotty (a.k.a. the future Scala 3 programming language), and its

("accept a sequence if it starts with α, and then φ holds"), or greatest fixed point \(vZ, \phi\). Basic formulas are enough [6, 73] to derive true/false (accept any/no sequence of actions), disjunction, implication, least fixed points \(\mu Z, \phi; (A)\phi\) accepts sequences that start with any \(a \in A\), then satisfy \(\phi\); dually, \((-\lambda)\phi\) requires \(a \in \text{Act}\setminus A\). We also derive usual temporal formulas \(\phi_1 U \phi_2 (\text{"} \phi_1 \text{"} holds, until \(\phi_2 \text{"} eventually holds\)), \(\Box \phi\) (\(\phi\) 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: [70])

We can decide \(\phi\) on a guarded type \(T\), as shown in Lemma 4.7. Here, we instantiate \(\text{Act}\) (Def. 4.6) as \(A_T(T)\), which is the set of labels fired along \(T\)’s transitions in \(\Gamma\), (Def. 4.2); notably, \(A_T(T)\) is finite and syntactically determined. (Details: [70])

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

Lemma 4.7 holds since guarded \(\pi\)-types are encodable in CCS without restriction [53], then in Petri nets [22, §4.1], for which linear-time \(\mu\)-calculus is decidable [20]. Notably, Lemma 4.7 covers infinite-state types (with \(p[...]|[...

5 Implementation and Evaluation

We designed \(\lambda^\pi\) to leverage subtyping and dependent function types, with a formulation close to (a fragment of) Dotty (a.k.a. the future Scala 3 programming language), and its

("accept a sequence if it starts with α, and then φ holds"), or greatest fixed point \(vZ, \phi\). Basic formulas are enough [6, 73] to derive true/false (accept any/no sequence of actions), disjunction, implication, least fixed points \(\mu Z, \phi; (A)\phi\) accepts sequences that start with any \(a \in A\), then satisfy \(\phi\); dually, \((-\lambda)\phi\) requires \(a \in \text{Act}\setminus A\). We also derive usual temporal formulas \(\phi_1 U \phi_2 (\text{"} \phi_1 \text{"} holds, until \(\phi_2 \text{"} eventually holds\)), \(\Box \phi\) (\(\phi\) 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: [70])

We can decide \(\phi\) on a guarded type \(T\), as shown in Lemma 4.7. Here, we instantiate \(\text{Act}\) (Def. 4.6) as \(A_T(T)\), which is the set of labels fired along \(T\)’s transitions in \(\Gamma\), (Def. 4.2); notably, \(A_T(T)\) is finite and syntactically determined. (Details: [70])

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

Lemma 4.7 holds since guarded \(\pi\)-types are encodable in CCS without restriction [53], then in Petri nets [22, §4.1], for which linear-time \(\mu\)-calculus is decidable [20]. Notably, Lemma 4.7 covers infinite-state types (with \(p[...]|[...

5 Implementation and Evaluation

We designed \(\lambda^\pi\) to leverage subtyping and dependent function types, with a formulation close to (a fragment of) Dotty (a.k.a. the future Scala 3 programming language), and its
1. Non-use of \( x_1, \ldots, x_n \): none of \( x_1, \ldots, x_n \) is used for output while \( t \) runs. (Simple variation: never use \( x_1, \ldots, x_n \) for input)
\[
\text{Def. 1.1: } \begin{align*}
T \vdash \varphi [x_1, \ldots, x_n] & \Rightarrow \Box \neg \left( \forall x_1, \ldots, x_n \left( \text{Chan}(x_i) \right) \right) \\
T \vdash \varphi [x_1, \ldots, x_n] & \Rightarrow \Box \neg \left( \forall x_1, \ldots, x_n \left( \text{Chan}(x_i) \right) \right)
\end{align*}
\]

2. Deadlock-freedom modulo \( x_1, \ldots, x_n \): \( t \) might only use channels \( x_1, \ldots, x_n \) to interact with other processes, and never gets stuck.
\[
T \vdash \varphi [x_1, \ldots, x_n] & \Rightarrow \Box \neg \left( \forall x_1, \ldots, x_n \left( \text{Chan}(x_i) \right) \right)
\]

3. Eventual usage of \( x_1, \ldots, x_n \): some \( x_i \) \((i \in \{1, \ldots, n\})\) is used for output, while \( t \) runs. (Simple variations: use some \( x_i \) for input or communication)
\[
T \vdash \varphi [x_1, \ldots, x_n] & \Rightarrow \Box \neg \left( \forall x_1, \ldots, x_n \left( \text{Chan}(x_i) \right) \right)
\]

4. Forwards from \( x \) to \( y \): whenever some \( z \) is received from \( x \), it is eventually forwarded via \( y \), before \( x \) is used for input again.
\[
T \vdash \varphi [x, y] & \Rightarrow \Box \neg (x(z)T) \Rightarrow (y \neg \varphi [x, y] (z)T)
\]

5. Reactiveness on \( x \): \( t \) runs forever, and is always eventually able to receive inputs from \( x \) (possibly after a finite number of \( \tau \) steps).
\[
T \vdash \varphi [x] & \Rightarrow \Box \neg (\varphi [x] (x)T) \Rightarrow (\varphi [x] \neg \varphi [x] (x)T)
\]

6. Responsiveness on \( x \): whenever some \( z \) is received from \( x \), it is eventually sent as a response, before \( x \) is used for input again.
\[
T \vdash \varphi [x] & \Rightarrow \Box \neg (\varphi [x] (x)T) \Rightarrow (\varphi [x] \neg \varphi [x] (x)T)
\]

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 \( \forall \cup X \), and we write \( \overline{\exists}(w) \) as shorthand for the (finite) set of labels \( \{\exists(w) \mid w \in \forall \cup X\} \) (and similarly for \( x(w) \)). For brevity, in (4) and (6) we write \( (a) \Rightarrow \phi \) instead of \( (a) \Rightarrow \phi \) (i.e., if we observe \( a \), then \( \phi \) holds afterwards).

5.1 Implementation
A payoff of \( \lambda \mathcal{X} \) is that we can implement it as an internal embedded domain-specific language (EDSL) in Dotty: i.e., we can reuse Dotty’s syntax and type system, to define: (1) typed communication channels, (2) dedicated methods to render the \( \lambda \mathcal{X} \) concurrency primitives \( \text{send}, \text{recv}, \llbracket, \llbracket \rangle \), and (3) dedicated classes to render their types \( o[...], i[...], p[...], nil \), including the well-formedness and subtyping constraints illustrated in Fig. 4. As usual for internal language embeddings, the Eppi DSL does not directly cause side-effects: e.g., calling receive(c) \((x \Rightarrow P)\) does not cause an input from channel \( c \). Instead, the receive method returns an object of type \( \text{InT}[... \] (corresponding to \( i[... \)) in Def. 5.1), which describes the act of using \( c \) to receive a value \( v \), and continue as \( P(v/s) \). Such objects are executed by the Eppi interpreter, according to the \( \lambda \mathcal{X} \) semantics (Def. 2.4).

Eppi programs look like the code on the right (which is pongo from Ex. 2.2): they follow the \( \lambda \mathcal{X} \) syntax. Also, types are rendered isomorphically: the type “\( x \)” in \( \lambda \mathcal{X} \) is rendered as “\( x \)” in Dotty, and dependent function types become:

\[
\Pi(x: T) o \left[ y, x, T' \right] \sim \sim (x: T) : \Rightarrow \text{Out}[y, x, \text{type}, x, \text{type}, T']
\]

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, Eppi also provides some extensions over \( \lambda \mathcal{X} \), like buffered channels, and a sequencing operator “>>” (see above, and in Fig. 1). Moreover, Eppi 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 \( U \) (right): notice how \( T \) is reused, passing \( U \)'s parameter.

Next, also notice how the type of \( f \)'s argument \((x . \text{type})\) is passed to \( U \), and then to \( T \): consequently, the type of \( f \) expands into \( \text{Par}[\text{Out}[x . \text{type}, Int], \text{Out}[x . \text{type}, Int]] \). To guide Eppi’s design, we implemented the full “payment with audit” use case from the experimental “session” extension for Akka Typed [41] (cf. §1, code snippet in Fig. 1).

An efficient Eppi 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 Eppi interpreter, leveraging the fact that, in Eppi programs as in \( \lambda \mathcal{X} \), 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 [47], with a notable difference: in Eppi, 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 \( \lambda \mathcal{X} \) EDSL, Eppi provides a simplified actor-based API [25], in a flavour similar to Akka Typed [49, 50] (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:
Verifying Message-Passing Programs with Dependent Behavioural Types  

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

• A is a process with a unique, \textit{implicit} input channel \(m\), of type \(c[T]\) (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 \texttt{read} — which is syntactic sugar for 
  \begin{equation*}
  \texttt{recv}(m, \ldots)
  \end{equation*}
  (see Fig. 1, and notice that the input channel \(m\) is left implicit);
• other processes/actors can send messages to A through its \texttt{ActorReference} \(r\) — which is just the output endpoint of its channel/mailbox \(m\). The type of \(r\) is \(c[T]\) (Def. 3.1): it only allows to send messages of type \(T\).

To this purpose, \textsc{Effpi} uses Dotty’s implicit function types \cite{57}: i.e., type \texttt{Actor[\ldots]} in Fig. 1 hides an input channel.

\textbf{Type-level model checking}  
The implementation details discussed thus far cover the \(\lambda_\Pi\) syntax, semantics, and typing — i.e., \S 2 and \S 3. The type-level analysis presented in \S 4 goes beyond the capabilities of the Dotty compiler; hence, we implement it as a Dotty compiler plugin (i.e., a compiler phase \cite{59}) accessing the typed program AST. The plugin looks for methods annotated with 
\begin{equation*}
\texttt{@effpi.verifier.verify}:
\end{equation*}
\begin{equation*}
\texttt{def f(x:\ldots, y:\ldots): T = \ldots}
\end{equation*}
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 \(x,y,\ldots\) of \(f\), and it can be either written by programmers, or inferred by Dotty. Then, the plugin:

1. tries to convert \(T\) into a \(\lambda_\Pi\) type \(\lambda_\Pi T\), as per Def. 3.1;
2. checks if \(\lambda_\Pi T \models \phi'\) holds — where \(\phi'\) is the companion formula of \(\phi\) in Fig. 7 (right). This step uses the mCRL2 model checker \cite{23}: we encode \(\lambda_\Pi T\) into an mCRL2 process,\footnote{To 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.} 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 \texttt{ponger} (code above) in \textsc{Effpi} can (a) annotate the method \texttt{ponger} to verify that it is responsive (Fig. 7(6)), and/or (b) annotate an unimplemented stub 
\begin{equation*}
\texttt{def f'(\ldots): T' = ???}
\end{equation*}
with type \(T'\) matching \(T_{pp}\) (Ex. 3.3), to verify that if \texttt{ponger} interacts with any implementation of type \(T_{ping}\), then \texttt{ponger}’s self channel is used for output (Fig. 7(3)). Also, a programmer can annotate payment (Fig. 1) to verify that it is reactive and responsive on its (implicit) mailbox, and \texttt{Accepts} payments after notifying on \texttt{aud} (with a variation of properties (5), (4) in Fig. 7, right).

\textbf{Known limitations}  
The implementation of our verification approach, outlined above, has three main limitations.

1. It does not check productivity of annotated code: such checks are unsupported in Dotty, and in most programming languages. Hence, programmers must ensure that all functions invoked from their \textsc{Effpi} code eventually return a value — otherwise, liveness properties might not hold at run-time (cf. condition (c1) in \S 4).
2. It does not verify processes with unbounded parallel components (i.e., with parallel composition under recursion);\footnote{This 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.} hence, it rejects types having \(p[\ldots\ldots]\) under \(\mu\). This does not impact the examples in this paper.
3. It uses iso-recursive types \cite{60, 21} because, unlike \(\lambda_\Pi\) (Def. 3.2), Dotty does not have equi-recursive types.

Limitations 1 and 3 might be avoided by implementing \(\lambda_\Pi\) as a new programming language. However, our Dotty embedding is simpler, and lets \textsc{Effpi} programs access methods and data from any library on the JVM: e.g., \textsc{Effpi} actors/processes can communicate over a network (via Akka Remoting \cite{48}), and with Akka Typed actors.

\subsection{5.2 Evaluation}  

From \S 5.1, two factors can hamper \textsc{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.

\textbf{Run-time benchmarks}  

We adopted a set of benchmarks from the Savina suite \cite{31}, with diverse interaction patterns:

• \texttt{chameneos}: \(n\) actors (“chameneos”) connect to a central broker, who picks pairs and sends them their respective \texttt{ActorReferences}, so they can interact peer-to-peer \cite{34};
• \texttt{counting}: actor \(A\) sends \(n\) numbers to \(B\), who adds them;
• \texttt{fork-join — creation (FJ-C)}: creation of \(n\) new actors, who signal their readiness to interact;
• \texttt{fork-join — throughput (FJ-T)}: creation of \(n\) new actors, and transmission of a sequence of messages to each.
• \texttt{ping-pong}: \(n\) pairs of actors exchange requests-responses;
• \texttt{ring}: \(n\) actors, connected in a ring, pass each other a token;
• \texttt{streaming ring}: similar to \texttt{ring}, but passing \(m\) tokens consecutively (i.e., at most \(m\) actors can be active at once).

For all benchmarks, we performed two measurements:

• \texttt{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);
• \texttt{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 Effpi runtime (with two scheduling policies: "default" and "channel FSM") against Akka, with default setup. Our approach appears viable: Effpi is a research prototype, and still, its performance is not too far from Akka. The negative exception is "chameneos" (Effpi is ~2x slower); the positive exceptions are fork-join throughput (Effpi is ~2x 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 highlighting 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 [56] (for Concurrent ML); type-based verification of temporal logic properties was addressed in [29, 30] (for the π-calculus); recent applications include, e.g., the verification of Go programs [44, 45].

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–passing; (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 [24, 78, 79]: their types depend on process channels, and they check if a process might use a channel $x$ — but cannot say if, when or how $x$ is used, nor verify behavioural properties.
Verifying Message-Passing Programs with Dependent Behavioural Types

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

or deadlock-freedom analysis, e.g., [36–39, 58]. [13] type-checks
ear logic-based session types with shared channels: it adds
multiple channels (more liberally than session types [12]), and
can verify liveness properties across interleaved use of mul-
match types

On the one hand, we do not have an explicit external
Our theory has a different design, yielding different fea-
and indexed types [ 10, 14, 75–77]; session types have in-
[13], with decidable verification (by Lemma. 4.7).

Various π-calculus type systems specialise on accurate
actors with unordered mailboxes, carrying messages of dif-
derent types; it ensures deadlock-freedom, and (assuming
actors with unordered mailboxes, carrying messages of dif-

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. (\texttt{4cIntel i7 @ 3.60GHz, 16 GB RAM, mCRL2 20180800, 30 runs})

works above develop new languages, or build upon a powerful
dependently-typed host language (Coq) with interactive
proofs, to support rich representations of protocol state. We,
instead, aim at Dotty embedding (with limited type depend-
encies) and automated verification of process properties (via
type-level model checking); hence, our protocols and logic
are action-based, to ensure decidability (Lemma 4.7). Our
approach covers many stateful protocols (e.g., locking/uxet,
mentioned above); but beyond this, a finer type-level rep-
resentation of state may make model checking undecidable
[19], thus requiring decidability conditions, or novel heurist-
ic/interactive proof techniques. This topic can foster exciting
future work, and a cross-pollination of results between the
realms of protocol-aware verification, and process calculi.

Future work We will study λπ embeddings in other pro-
gramming 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 LTSmin [35];
2. support infinite-state systems, trying tools like Bmc [33]
(that does not cover the linear-time μ-calculus in Def. 4.6,
but is used e.g. in [15] to verify safety properties of actor
programs); 3. introduce assume-guarantee reasoning for type-
level model checking, inspired by [62]. The Effπ1 runtime
system can be optimised: we will attempt its integration with
Akka Dispatchers [47], 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
supported by EPSRC projects EP/K034413/1, EP/K011715/1,
EP/L00058X/1, EP/N027833/1, EP/N028201/1.