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 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 \( \lambda\Pi \), 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 \( \text{Eff}_p \), offering a simplified actor-based API; (4) it enables an efficient runtime system for \( \text{Eff}_p \), for highly concurrent programs with millions of processes/actors.

CCS Concepts  
- Theory of computation \( \rightarrow \) Process calculi; Type structures; Verification by model checking  
- Software and its engineering \( \rightarrow \) Concurrent programming languages.

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

ACM Reference Format:

1 def payment(aud: ActorRef[Audit[_]]): Actor[Pay, _] =
2 forever {
3   read (pay: Pay =>
4     if (pay.amount > 42000) {
5       send(pay.replyTo, Rejected("Too high!"))
6     } else {
7       send(aud, Audit(pay)) >>
8       send(pay.replyTo, Accepted)
9   })

Figure 1. Implementation of the payment service specification (§1). Although similar to Akka Typed [47], it is written in Dotty and \( \text{Eff}_p \), described in §5; ">>" (l.7) means "and then."

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 [43], and produce compilation errors when a program violates a desired protocol. However, the resulting experiments [38] had no rigorous grounding: although inspired by the session types theory [3, 25], 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.

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:

\[
\text{Forever} \left[ \text{In}[\text{Pay}, \ (p: \text{Pay}) \Rightarrow \text{Out}(p.\text{replyTo}.\text{type}, \text{Rejected}) \ | \ (\text{Out}[\text{aud}.\text{type}, \text{Audit}[p.\text{type}]] \Rightarrow \text{Out}(p.\text{replyTo}.\text{type}, \text{Accepted}) ) \right]
\]

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 of type Pay, and then either (‘| means “or”) send `Rejected` on `p.replyTo`, or send an audit, and then (‘|’) send `Accepted`. Notably, `p` is bound by a dependent function type [15].

Effpi is built upon a concurrent functional calculus for channel-based interaction, called \(\lambda P\), its novelty is a blend of behavioural types (inspired by \(\pi\)-calculus literature) with dependent function types (inspired by Dotty’s foundation \(\lambda C\) [2]), achieving unique specification and verification capabilities. Effpi implements \(\lambda P\) 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:

\[
\text{In[\text{Audit[...], (a: Audit[...]] \Rightarrow End ]}
\]

(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 \(\lambda P\) 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 \(\lambda P\) calculus, at the basis of Effpi. §3 presents type system of \(\lambda P\). §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 \(\lambda P\) 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 \(\lambda P\)-Calculus

The theoretical basis of our work is a \(\lambda\)-calculus extended with channels, input/output, and parallel composition, called \(\lambda P\). 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 [51, 52, 60]. Indeed, \(\lambda P\) 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 \(\lambda P\) is in Fig. 2. Elements of \(\mathbb{C}\) are run-time syntax. Free/bound variables \(fv(t)/bv(t)\) are defined as usual. We adopt the Barendregt convention: bound

\[
B = \{tt, ff\} C = \{a, b, c, \ldots\} X = \{x, y, z, \ldots\}
\]

\[
\begin{align*}
\text{terms } & \vdash t, t', \ldots : : \mathbb{X} \ | \ \mathbb{V} \ | \ \neg t \quad \text{if then } t_1 \ 	ext{else } t_2 \\
& \text{let } x = t \ \text{in } t' | t' t' | \text{chan()} | \text{P} \\
\text{values } & \vdash u, u, \ldots : : \mathbb{B} | \mathbb{C} | \lambda x.t \ | \ (t) | \text{err} \\
\text{processes } & \vdash p, q, \ldots : : \text{end} | \text{send}(t, t') | \text{recv}(t, t') | t | t'
\end{align*}
\]

Figure 2. Syntax of \(\lambda P\) terms. The set \(\mathbb{C}\) (highlighted) contains channel instances, that are part of the run-time syntax.
variables are syntactically distinct from each other, and from free variables. We write $\lambda_x.t$ for $\lambda x.t$, when $x \not\in \text{fv}(t)$.

The set of values $\mathbb{V}$ includes booleans $\mathbb{B}$, channel instances $\mathbb{C}$, function abstraction, the unit ($\bot$), and error. The terms (in $\mathbb{T}$) can be variables (from $\mathbb{X}$), values (from $\mathbb{V}$), various standard constructs (negation $\neg$, if/then/else, let binding, function application), and also channel creation $\text{chan}(\_)$, and process terms (from $\mathbb{P}$). The primitive $\text{chan}(\_)$ 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 $\text{end}$, 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 $t | t'$ (meaning: $t$ and $t'$ run concurrently, and can interact). $\lambda^\mathbb{T}$ can be routinely extended with, e.g., integers, strings, records, variants: we use them in examples.

**Example 2.2.** A ping-pong system in $\lambda^\mathbb{T}$ is written as:

```ml
let pinger = $\lambda$self.( $\text{send}($pongc, self, $\_$. ) $\text{recv}(self, $\_$. $\text{replyTo}, $\_$. )

let ponger = $\lambda$self.( $\text{recv}(self, $\_$. $\text{reply}, $\_$. ) $\text{send}($replyTo, $\_$. $\text{ReplyTo}, $\_$. $\text{end}$. $\_$. ))

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

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

• pinger is an abstract process that takes two channels: self (its own input channel), and pongc. It uses pongc to send self, then uses self to receive a response, and ends;

• ponger takes a channel self, uses it to receive $\text{replyTo}$, then uses $\text{replyTo}$ to send "$\_$. $\text{ReplyTo}"$, and ends;

• sys takes channels $y'$, $z'$, and uses them to instantiate $\text{pinger}$ and $\text{ponger}$ 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 $\lambda^\mathbb{T}$). Evaluation contexts $\mathcal{E}$ and reduction $\rightarrow$ are illustrated in Fig. 3, where congruence $\equiv$ is defined as: $t_1 \equiv t_2 \equiv t_3 \equiv t_4 \equiv t_5 \equiv t_6$, and $\text{end} \equiv \text{end}$, plus $\alpha$-conversion. We write $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$. We say "$t$ has an error" iff $t \in \text{E} (\text{error})$ (for some $\mathcal{E}$). We say "$t$ is safe" iff $\forall t': t \not\rightarrow t'$ implies $t'$ has no error.

Def. 2.4 is a standard call-by-value semantics, with two rules for concurrency. $\text{R-CHAN}$ says that chan() returns a fresh channel instance; $\text{R-COMM}$ says that the parallel composition $\text{send}(a, u, v_1) | \text{recv}(a, v_2)$, where both sides operate on the same channel instance $a$, transfers the value $u$ on the receiver side, yielding $v_1() | v_2: u$: 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 $|$).

### 3 Type System

We now introduce the type system of $\lambda^\mathbb{T}$. 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) we allow types to contain variables from the term syntax (inspired by $\lambda_{\text{c}}$, the calculus behind Dotty [2]). The syntax of types is in Def. 3.1.

Notably, points (1) and (3) establish a similarity between $\lambda^\mathbb{T}$ and $\lambda_{\text{c}}$: (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 $\lambda_{\text{c}}$, polymorphic types can depend on type variables in the environment; hence, we use mutually-defined judgements, akin to those of $\lambda_{\text{c}}$, to assess the validity of environments, types, subtyping, and typed terms (Def. 3.2).

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

- $\text{bool} | () | \top | \bot | T \lor U | \Pi(x:T)U | \mu x.T | x$
- $\mathbb{C} | \mathbb{B} | \mathbb{P}$

- $\text{proc} | \text{nil} | \text{nil}[S, T, U] | [S, T, U] | \text{nil}[T, U]$

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

We write $\Pi(T)$ for $\Pi(x:T)T$ if $\not\in \text{fv}(T)$, and distinguish recursion variables as $T, T', \ldots$, i.e., we write $\nu T$. We write $\bar{T}$ 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:

- $T \equiv U \lor T \lor V \lor U \lor V$ such that $U \equiv T \lor (\Pi x:T)U$.
- $\mathbb{C} | \mathbb{B} | \mathbb{P}$ such that $\Pi(S, T, U) \equiv [S, T, U] \equiv [T, \text{nil}] \equiv T$

The first row of productions in Def. 3.1 includes booleans, the unit type $(),$ top/bottom types $\top, \bot$, 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 $\mathbb{X}$ in Def. 2.1): the underlining is a visual clue to better distinguish $x$ used in a type, from $x$ used in a $\lambda^\mathbb{T}$ term.

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

**Definition 3.2.** These judgements are formalised in Fig. 4.

<table>
<thead>
<tr>
<th>Type of Term</th>
<th>Judgement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash T$</td>
<td>$\Gamma$ is a valid type environment</td>
</tr>
<tr>
<td>$\Gamma \vdash \pi : T$</td>
<td>$T$ is a valid type in $\Gamma$</td>
</tr>
<tr>
<td>$\Gamma \vdash \pi$</td>
<td>$\forall \bar{U} \in \bar{T}: \Gamma \vdash \text{U}$ type</td>
</tr>
<tr>
<td>$\Gamma \vdash \pi : \forall \bar{U} \in \bar{T}$</td>
<td>$\Gamma$ is a valid type process in $\Gamma$</td>
</tr>
<tr>
<td>$\Gamma \vdash \pi : \forall \bar{U} \in \bar{T}$</td>
<td>$\Gamma \vdash \pi : \forall \bar{U} \in \bar{T}$ type</td>
</tr>
<tr>
<td>$\Gamma \vdash \pi : \forall \bar{U} \in \bar{T}$</td>
<td>$\Gamma \vdash \pi : \forall \bar{U} \in \bar{T}$ type</td>
</tr>
<tr>
<td>$\Gamma \vdash \pi : \forall \bar{U} \in \bar{T}$</td>
<td>$\Gamma \vdash \pi : \forall \bar{U} \in \bar{T}$ type</td>
</tr>
<tr>
<td>$\Gamma \vdash \pi : \forall \bar{U} \in \bar{T}$</td>
<td>$\Gamma \vdash \pi : \forall \bar{U} \in \bar{T}$ type</td>
</tr>
</tbody>
</table>

A typing environment $\Gamma$ maps variables (from $X$ in Def. 2.1) to types; the order of the entries of $\Gamma$ is immaterial. In 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: $\Gamma \vdash \pi : T$ and $\Gamma \vdash \pi : \forall \bar{U} \in \bar{T}$ type. The former is standard (except for rule $\{\pi\}$, 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 $\{\pi\}$), but not to a $\pi$-type; this also means that function arguments cannot be $\pi$-typed. Still, if in a function type $\Pi(x:T)U$, the return type $U$ can be a $\pi$-type (rule $\{\pi\}$): hence, it is possible to define abstract process types (cf. Ex. 3.3 and 3.4 later on). Rules $\{\mu\}$ and $\{\pi\}$ are based on [29, $\S$2], and require recursive types to be contractive: e.g., $\mu_1, \mu_2, ..., \mu_n (T_1 \cup U)$ is not a type: clause $\forall \bar{U} \in \bar{T}$ means that variable $V$ is not bound in negative position in $T$, as in $\text{f}$. (Details: [61]). Recursion is supported by $\{\pi\}$: in $T = t \in T'$, term $t$ can refer to $x$. Rule $\{\bar{U}\}$, based on [9], ensures decidability of subtyping [29, $\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 now discuss the main judgements.

**Variables, types, subtyping, and dependencies** The environment $\Gamma = x:T$ assigns type $T$ to variable $x$. Hence, by rule $\{\pi\}$, the type $x$ is valid in $\Gamma$; and indeed, by rule $\{\pi\}$, we can infer $\Gamma \vdash x : x$, 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 $\{\pi\}$, that says: $\Gamma$ 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 $\{\pi\}$), as shown here. Since $\Gamma \vdash x : T$, the judgement $\Gamma \vdash t : x$ holds if $t$ is a subtype of $x$ and $t$ is not $\pi$-typed. Crucially, by rule $\{\pi\}$, that $\Gamma \vdash t : x$ and $\Gamma \vdash t : T'$ (denotes a process that sends a $T'$-typed value on an $S$-typed channel, and continues as $U$; the input type $\text{i}[S, T]$ denotes a process that receives a value from an $S$-typed channel and continues as $T$; the parallel type $\text{p}[T, U]$ denotes the parallel composition of two processes ($T$ and $U$).

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

```
ping : T_ping = \self\{\text{self} : \text{c}[]\} \Pi \text{p}[:c[]][\text{self} . \Pi \text{reply} \text{str} \text{nil}]
ponger : T_ponger = \self\{\text{self} : \text{c}[]\} \Pi \text{p}[:c[]][\text{self} . \Pi \text{reply} \text{str} \text{nil}]
sys : T_pp = \Pi \text{x} : \text{c}[][\text{x} : \text{c}[]][\text{sys} \Pi \text{c}[][\text{sys}]]
```

Notice how $T_pp$ captures the ping/pong composition of sys, preserving its channel topology: the type-level applications
Figure 4. Judgements of the $\lambda^\Pi_{\leq}$ type system (Def. 3.2). The main concurrency-related rules are highlighted.

Example 3.4 (Mobile code). Modern languages and toolkits for message-passing programs support sending/receiving mobile code (e.g., [17, 46, 49]). Consider this scenario: a data analysis server lets its clients send custom code, for on-the-fly data filtering. In $\lambda^\Pi_{\leq}$, the intended behaviour of custom code can be formalised by a type like $T_m$ below: it describes an abstract process, taking two input channels $i_1$/$i_2$ and an output channel $o$; it must use $i_1$/$i_2$ to input integers $x/y$, and then it must send one of them along $o$, recursively.

$T_m = \Pi(i_1 : c^\Pi([\mathbb{I}]]) \Pi(i_2 : c^\Pi([\mathbb{I}])) \Pi(o : c^\Pi([\mathbb{I}])))$

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 $out$ a value not coming from $i_1$/$i_2$ (we will formalise these intuitions in Ex. 4.11). The terms below implement $T_m$: $m_1$ always sends $x$ received from $i_1$, then recursively calls itself, swapping $i_1$/$i_2$; $m_2$ sends the maximum between $x$ and $y$.

let $m_1 = \lambda i_1. i_2. o. (x \lor y)$
let $m_2 = \lambda i_1. i_2. o. (\lambda x. x \lor y)$

Below, $srv$ is a data processing server. It takes two channels: $cm$ and $out$; it creates two private channels $z_1$ and $z_2$, uses $cm$ to receive an abstract process $p$, and runs it, in parallel with two producers (omitted) that send values on $z_1$/$z_2$.
We can replace a larger one. Crucially, in our theory, supertyping Liskov & Wing's substitution principle [48]: a smaller object \( \tau \) has type \( \sigma \) only if \( \Gamma \vdash \sigma \to \tau \), implying \( \Gamma \vdash \tau \). We can replace \( \text{proc} \) with a more precise type. If \( U_1/U_2 \) are types of \( \prod \sigma_1 \sigma_2 \), the \( \text{recv}(\ldots) \) sub-term of \( \text{srv} \) has type:

\[
T_{\text{srv}} = \Pi (p : T_m) \Pi (c : \text{cmt} \{ p \}) \text{recv}(\ldots) \mid \text{srv} \mid \text{x} = \text{cmt} \{ p \} \text{recv}(\ldots)
\]

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 \( F_{<, [8, 29]} \)) except the highlighted ones. By \( \text{proc} \), subtyping for channel types is covariant for inputs, and contravariant for outputs, as expected [58]: intuitively, channels with smaller types can be used more liberally. Rule \( \text{proc} \) says that \( \text{proc} \) is the top type for \( \pi \)-types. Rules \( \text{proc} \) say that types for input/output/parallel processes are covariant in all parameters.

As usual, supertyping / subsumption (rule \( \text{let} \)) caters for Liskov & Wing's substitution principle [48]: a smaller object can replace a larger one. Crucially, in our theory, supertyping also allows to \textit{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 \( \lambda^\Sigma \) terms with a non-Turing-complete type language, for the results in \( \S 4 \).

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

\[
\begin{align*}
t_1 & = \text{send}(x, z_2, z_3) \mid \text{recv}(x, z_2, z_3) \\
t_2 & = (\text{let } z = \text{chan}() \mid \text{send}(z, 42, z_2, z_3)) \mid \text{recv}(x, z_2, z_3) \\
t_1 & = p \mid o \mid x \mid \text{int} \mid \text{nil} \\
t_2 & = p \mid o \mid c^\text{int} \mid \text{int} \mid \text{nil} \\
\end{align*}
\]

Letting \( \Gamma = x : c^\text{int} \), we have \( \Gamma \vdash x \leq c^\text{int} \) 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 \( \text{proc} \)): 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 \( c^\text{int} \) 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 \textit{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 \( \text{let} \) (Fig. 4): since \( z \) is bound by \textit{let}..., its occurrence in \text{send}(...) is typed by a supertype of \( z \) that is suitable for both \( z \) and \text{chan}() in this case, \( c^\text{int} \). Specifically:

\[
\Gamma \vdash c^\text{int} \leq c^\text{int} \\
\Gamma, z : c^\text{int} \vdash \text{chan}() : c^\text{int} \\
\Gamma, z : c^\text{int} \vdash \text{send}(z, 42, \lambda \text{end} \to \text{send}(z, 42, \lambda \text{end} \to \text{recv}(\ldots)) \mid o \mid \text{int} \mid \text{nil}) \\
\Gamma \vdash \text{let } z = \text{chan}() \mid \text{send}(z, 42, \lambda \text{end} \to \text{recv}(\ldots)) \mid o \mid \text{int} \mid \text{nil} \leq c^\text{int} \]
Definition 4.1 (Labelled semantics of open typed terms). When $\Gamma \vdash t : T$ (for any $\Gamma$, $t$, $T$), the judgements $\Gamma \vdash t \triangleleft x$, $t'$ and $\Gamma \vdash t \xrightarrow{x}^* t'$ are inductively defined in Fig. 5.

Unlike Def. 2.4, Def. 4.1 lets an open term like $\neg x$ reduce, by non-deterministically instantiating $x$ to $tt$ or $ff$; the assumption $\Gamma \vdash \neg x : T$ ensures that $x$ is a boolean. Rule $[SR-\neg]$ inherits "concrete" reductions from Def. 2.4: if $t \rightarrow t'$ is induced by base rule $[s]$, the transition label is $\tau[\alpha]$; Rules $[SR-send]/[SR-recv]$ send/receive a value/variable $w'$ using a (channel-typed) value/variable $w$. Note that in $[SR-recv]$, $w'$ is any value/variable of type $T_i$, which is the input type of $x$ (in $\pi$-calculus jargon, it is an early semantics [60]). Rule $[SR-Comm]$ lets processes exchange a payload $w'$ via a channel/value $w$, recording $w$ in the transition label. Rule $[SR-x]$ "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 $[SR-\triangleright]$ applies a function to a variable $x$, with the expected substitution. Rule $[SR-\triangleright]$ propagates transitions through contexts, unless labels refer to bound variables. Finally, $\Gamma \vdash t \xrightarrow{x}^* t'$ holds when $t$ reaches $t'$ via a finite sequence of internal moves excluding interaction: i.e., labels $w'(w')$, $\bar{w}(w')$, $\tau[w]$, and $\tau[R-Comm]$ are forbidden.

Using Def. 4.1 on $t_1$ from Ex. 3.5, we get the transition

$$\Gamma \vdash t_1 \xrightarrow{\tau[\alpha]} \text{ end } \text{ end},$$

and we observe the use of $x$, as desired.

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, 27] — but our novel use of type variables, and dependent function types, yields new capabilities, and requires some sophistication.

The type transitions 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 $\Gamma \vdash t_1 \xrightarrow{\tau[\alpha]} \text{ end } \text{ end}$. This suggests a type like $p\{ o[x, \ldots], i[x, \ldots] \}$ 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 $p\{ o[c^0[\text{int}], \ldots], i[x, \ldots] \}$ should reduce, too. In general, we want $p\{ o[S, \ldots], i[T, \ldots] \}$ to reduce if $S$ and $T$ "might interact", i.e., they could type a same channel/variable: we formalise this idea as $\Gamma \vdash S \equiv T$ in Def. 4.2.

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

$$\Gamma \vdash S \not\equiv T \land \Gamma \vdash S \equiv T \quad \iff \quad \Gamma \vdash S \equiv T$$

A type reduction context $E$ is inductively defined as:

$$[\_] \mid o[E, T, U] \mid o[S, E, U] \mid o[S, T, E] \mid i[E, T] \mid i[S, E] \mid p[E, T]$$

Judgements $\Gamma \vdash t \xrightarrow{\alpha} t'$ and $\Gamma \vdash t \xrightarrow{[S]} t'$ are in Fig. 6.

By Def. 4.2, $\Gamma \vdash S \equiv T$ holds when $S$ and $T$ have a common subtype besides $\bot$, i.e., they might type a same term in $\Gamma$, via rule $[\cdot \triangleleft \cdot]$. The judgement $\Gamma \vdash T \xrightarrow{\alpha} T'$ says that $T \equiv U$ can reduce to $T$ or $U$, firing label $\tau[V]$. Rule $[\cdot \rightarrow [\cdot]]$ reduces an output type, recording the used channel type $S$ and payload $T$ in the transition label. Rule $[\cdot \rightarrow [\cdot]]$ is similar for input types, recording the payload $T'$. We have two communication rules:

- $[\cdot \rightarrow [\cdot]]$ fires when, in $p[U, U']$, there might be an interaction with a type variable $x$ as payload. Note that, by $[\cdot \rightarrow [\cdot]]$, the $x$ sent by $U$ is substituted 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;
- $[\cdot \rightarrow [\cdot]]$ is similar, but fires if the payload $T$ is not a variable.

Finally, $\Gamma \vdash T \xrightarrow{[S]} T'$ 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:

$$\Gamma = \psi x \bowtie [\text{str}], \psi z \bowtie [\text{str}]$$

$$t = \text{sys y z}$$

$$T = T_{pp} y z = \{ p\{ \alpha_{\psi x, \psi z}, \psi z, \Pi[\text{replyTo}].\text{str} \}, o\{ \psi y, \Pi[\text{replyTo}].\text{str} \} \}$$

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

$$\Gamma \vdash t \xrightarrow{\psi z} \xrightarrow{\psi y} \xrightarrow{o[\psi z]} \xrightarrow{p[\psi z]}$$

By Def. 4.2, applying rule $[\cdot \rightarrow [\cdot]]$ twice, we get:

$$\Gamma \vdash T \xrightarrow{[\psi z]} p\{ \alpha_{\psi x, \psi z}, \psi z, \Pi[\text{replyTo}].\text{str} \} \equiv p\{ \psi y, \Pi[\text{replyTo}].\text{str} \} \xrightarrow{[\tau]} p[\psi z]$$

Observe that $T'$ closely mimics 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 function type inside $\bar{i}[…]$. 

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 $\Gamma \vdash t : T$ type, then $\Gamma \vdash t \xrightarrow{\alpha} t' \iff \Gamma \vdash t' : T$. Otherwise, when $\Gamma \vdash T$ $\pi$-type, we have:

1. $\Gamma \vdash t \xrightarrow{\alpha} t'$ with $\tau[\alpha]$ (Fig. 5) implies $\Gamma \vdash t' : T$;
2. $\Gamma \vdash t \xrightarrow{\alpha} t'$ and $\alpha \in \{ \exists x (w), x(w), \tau[x], \tau[R-Comm] \}$ implies one:
   a. $\alpha = \tau[t]'$ and $\text{proc} \in T$;
   b. $\alpha = \exists x (w)$ and $\exists S, U : T' \equiv \Gamma \vdash x : S, w : U, t' : T'$ and
   $$\Gamma \vdash t \xrightarrow{\tau[x]} \exists S, U : T;$$
   c. $\alpha = x(w)$ and $\exists S, U : T' \equiv \Gamma \vdash x : S, w : U, t' : T'$ and
   $$\Gamma \vdash t \xrightarrow{\tau[x]} \exists S, U : T;$$
   d. $\alpha = \tau[x] \text{ and } \exists S, S', T' \equiv \Gamma \vdash x : S, x : S', t' : T'$ and
   $$\Gamma \vdash t \xrightarrow{\tau[x]} \exists S, S' : T;$$
   e. $\alpha = \tau[R-Comm] \text{ and } \exists S, S', T' : \{ S, S' \} \not\subseteq \exists \Gamma \vdash t' : T$ and
   $$\Gamma \vdash t \xrightarrow{\tau[R-Comm]} \exists S, S' : T.$$
Theorem 4.5 (Type fidelity). Within productive $\lambda^\pi\subseteq\tau$ terms, assume $\Gamma \vdash t : T$ and $\Gamma \vdash T \pi$-type. Then:

1. $\Gamma \vdash T \quad\text{and}\quad \Gamma \vdash T \pi$-type.
2. $\Gamma \vdash T \pi$-type.
3. $\Gamma \vdash T \pi$-type.

4. $\Gamma \vdash T \pi$-type.

Proof: 1. $\Gamma \vdash T$.

2. $\Gamma \vdash T$.

3. $\Gamma \vdash T$.

4. $\Gamma \vdash T$.

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

Figure 6. Semantics of $\lambda^\pi$ terms. We will sometimes use label $\tau$ to denote either $\tau\cdot\cdot\cdot$ or $\tau\cdot\cdot\cdot$ (for some $S, S'$).

(type $T$. Instead, if the reduction is caused by input, output or interaction events, then we observe a corresponding labelled transition in the type, possibly after some $\tau\cdot\cdot\cdot$ moves (cases 2b–2e); the exception is case 2a: if $t' \vdash \tau\cdot\cdot\cdot$ then $t$ syntactically contains proc, which types a reducing sub-term of $t$ before and after its reduction (via rule $\tau\cdot\cdot\cdot$). 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 $\lambda^\pi$ terms, i.e., all functions must be total (always return a value or process when applied). This means that, e.g., if $\Gamma \vdash t : \text{of} X\cdot\cdot\cdot$ then $t$ will output on $x$: this excludes cases like $t = \text{if } w \text{ then } \text{send}(x, 42, t') \text{ else } \text{send}(x, 43, t''')$ (with $w = (\lambda y. y) y (\lambda z. z z)$). 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 $\lambda^\pi\subseteq\tau$, assume $\Gamma \vdash t : T$ and $\Gamma \vdash T \pi$-type. Then:

1. $\Gamma \vdash T \quad\text{and}\quad \Gamma \vdash T \pi$-type.
2. $\Gamma \vdash T \pi$-type.
3. $\Gamma \vdash T \pi$-type.

4. $\Gamma \vdash T \pi$-type.

Proof: 1. $\Gamma \vdash T$.

2. $\Gamma \vdash T$.

3. $\Gamma \vdash T$.

4. $\Gamma \vdash T$.

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 $\text{Act}$ ranged over by $\alpha$, the linear-time $\mu$-calculus formulas are defined as follows (where $A$ is a subset of $\text{Act}$):

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

Derived formulas: 

In Def. 4.6, $\phi$ describes accepted sequences of actions; $\phi$ can be a variable $Z$, negation, conjunction, prefixing $(\alpha)\phi$.
Verifying message-passing programs with dependent behavioural types
PLDI ’19, June 22–26, 2019, Phoenix, AZ, USA

("accept a sequence if it starts with α, and then φ holds"), or
greatest fixed point υ Z. φ. Basic formulas are enough [6, 64]
to derive true/false (accept any/no sequence of actions), dis-
junction, implication, least fixed points μZ. φ; (A)φ accepts
sequences that start with any α ∈ A, then satisfy φ; dually,
(¬A)φ requires α ∈ Act \ A. We also derive usual temporal
formulas φ1; U φ2 ("φ1 holds until φ2 eventually holds"), □φ
("φ is always true"), and ◻φ ("φ is eventually true"). Given
a process p with LTS of labels Act, a run of p is a finite or
infinite sequence of labels fired along a complete execution
of p; we write p =⇒ φ if φ accepts all runs of p. (Details: [61])

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

Lemma 4.7. Given Γ, we say that T is guarded iff, for all
π-type subterms µT U of T, t can occur in U only as subterm
of [t[...]] or α[...]; then, if T is guarded, T =⇒ φ is decidable.

Lemma 4.7 holds since guarded π-types are encodable in
CCS without restriction [50], then in Petri nets [21, §4.1],
for which linear-time μ-calculus is decidable [19]. Notably,
Lemma 4.7 covers finite-state types (with p[....,....] under
µt[....]), that type λσ terms with unbounded parallel subterms.

Now, assuming Γ + t : T, we can ensure that φ holds for
t, by deciding a related formula φ′ 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 χ;
(ii) if we want t to eventually perform an action on channel
χ, we need t productive, and check that T eventually
uses χ without doing “impossible” 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 Γ are:
input uses: U[Γ] ={S(U′) ∈ Act(T) | Γ + x ≤ S} ;
output uses: U[Γ] = {S(U′) ∈ Act(T) | Γ + x ≤ S}

Definition 4.9. Given a set of type (resp. term) variables Υ,
the ′-limited transitions of T (resp. t) in Γ are:

Γ + t =⇒ T′ ∀w, w′ : α ∈ (S(w), S(w′)) implies w =⇒ Y

Theorem 4.10. Within productive λσ[4], assume Γ + t : T, with
Γ + T π-type, proc /∈ T. Also assume, for all i[S, II(x;U′)]
occuring in T, that there is y such that Γ + y : U holds.1

1This implicitly requires T + U type; hence f v(U) ∩ bv(T) = ∅: this
assumption could be relaxed (with a more complicated clause), but offers a
compromise between simplicity and generality, that is sufficient to verify
our examples. Besides this, the existence of y such that Γ + y : U can

For μ-calculus judgements on T, let Act = Act(T), and Aσ =
{r[S, S′] ∈ Act(T) | {S, S′} ⊆ dom(Γ)}. Then, the implications in
Fig. 7 hold.

Assume Γ + t : T. The sets U[Γ] = U[Γ] = U[Γ] of Def. 4.8
contain all transition labels that might be fired by T, when
x is used for input/output by t. The operator ↑[Γ] = {x}i∈1...n
(Def. 4.9) limits the observable inputs/outputs of t/Γ to those
occurring on channel x — while other (open) channels can
only reduce by communicating, via τ-actions; i.e., x1, ..., xn
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 (i)
above: if T never fires a label (□¬(¬ ...)), that is a potential
output use of x (i ∈ 1..n), then t never uses x for output.
The "potential output use", by Def. 4.8, is any label S(U′)
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 (ii): to ensure that t eventually outputs
on x (i ∈ 1..n), we check that T eventually fires a label
S(U); moreover, we check T does not fire any label in Aσ,
until (U) the output S(U) occurs. The set Aσ contains all
“imprecise” synchronisation labels r[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 Aσ, then t
might deadlock and never perform S(U)(ω) (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 at a channel S (representing a potential
use of z), then T eventually fires S(U) — without doing
dependent inputs on x, nor firing any label in Aσ, before.

Example 4.11. Take Γ, t, T in Ex. 4.3. To ensure that t
eventually uses y to output a message, we check Γ =⇒ T′[Γ]
with φ in Fig. 7(3) (right).

Take ponger (Ex. 2.2), Tpong (Ex. 3.3), and Γ = z:τ[0][σ][str].
To ensure that the term ponger z is responsive on z, we
check (Tpong z) =⇒ T[Γ][z[Γ]] + φ, with φ in Fig. 7(6) (right).

Take T σ′ (Ex. 3.4). With an easy adaptation of properties
(4) and (5) in Fig. 7 (right), we can verify that: in all imple-
mentations sσ′ of T σ′ whenever sσ receives any mobile
code p (of type Tm) from channel cm, sσ′ becomes reactive
on z1 and z2, picking one input and forwarding it on out.

5 Implementation and Evaluation

We designed λσ[4] 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

be assumed w.l.o.g.; if Γ + t : T, then y : U, we can pick
y′ ∈ dom(Γ), extend Γ as Γ = Γ + y′ : U, and get Γ + y′ : U and Γ + t : T.
foundation $D_{\lambda}$. [2]. This naturally leads to an implementation, in three phases, shown in §5.1: (1) internal embedding of $\lambda_{\leq}$; (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 $\lambda_{\leq}$ 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 $\lambda_{\leq}$ concurrency primitives (send, recv, [], end), and (3) dedicated classes to render their types (o[], i[], p[], nil), including the well-formedness 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 receive(c)(x => P) does not cause an input from channel c. Instead, the receive method returns an object of type $\text{In}[]$ (corresponding to $\text{if}[]$ in Def. 3.1), which describes the act of using c to receive a value v, and continue as P[v/x]. Such objects are executed by the Effpi interpreter, according to the $\lambda_{\leq}$ syntax (Def. 2.4).

\begin{verbatim}
Effpi programs look like def ponger(self: T): T = { receive(self) { replyTo => send(replyTo, "Hi!") } end }
\end{verbatim}

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 $\lambda_{\leq}$, 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 $U$ (right): notice how $T$ is reused, passing U’s parameter.

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 $\lambda_{\leq}$, 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 $\lambda_{\leq}$ 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 $c[T]$ (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 $c[T]$ (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 $\text{Actor} [...] \text{ in Fig. 1 hides an input channel.}$

**Type-level model checking** The implementation details discussed thus far cover the $\lambda^\Sigma$ 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":

```effpi
@effpi.verifier.verify(\phi)
def f(x: ..., y: ...): 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 $\mathbb{T}$ 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 $\lambda^\Sigma$ type $T'$, as per Def. 3.1;
2. checks if $T \models T'$ holds — where $T'$ 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 $T'$ 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 $f'(...)$: $T' = ???$ with type $T'$ matching $T_{pp}$ (Ex. 3.3), to verify that if ponger interacts with any implementation of type $T_{pp}$, then 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 Accepts payments after notifying on aud (with a variation of properties (5), (4) in Fig. 7, right).

**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 Effpi code eventually return a value — otherwise, liveliness properties might not hold at run-time (cf. condition (e1) 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[...[... under $\mu$)... This does not impact the examples in this paper.
3. It uses iso-recursive types [57, Ch. 21] because, unlike $\lambda^\Sigma$ (Def. 3.2), Dotty does not have equi-recursive types.

Limitations 1 and 3 might be avoided by implementing $\lambda^\Sigma$ 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:

- **chameleons**: $n$ actors ("chameleons") 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);
- **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.

---

1To 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.
The results are in Fig. 8: we compare two instances of the Eff\(\pi\) runtime (with two scheduling policies: “default” and “channel FSM”) against Akka, with default setup. Our approach appears viable: Eff\(\pi\) is a research prototype, and still, its performance is not too far from Akka. The negative exception is “chameneos” (Eff\(\pi\) is \(\sim 2\times\) slower); the positive exceptions are fork-join throughput (Eff\(\pi\) is \(\sim 2\times\) 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 [53] (for Concurrent ML); type-based verification of temporal logic properties was addressed in [26, 27] (for the \(\pi\)-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 check if a process might use a channel \(x\) — but cannot say if, when or how \(x\) is used, nor verify behavioural properties.
We share similar goals, although we adopt a different theory and design, leading to different tradeoffs: crucially, the works above develop new languages, or build upon a powerful (deadlock)-freedom analysis, e.g., [33–36, 55]. [12] type-checks ρ-terminating continuations. This covers locking/mutex protocols, e.g., used e.g. in [14] to verify safety properties of actor programs; 2. support infinite-state systems, trying tools like Bmc [30] (that does not cover the linear-time μ-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 Effπ 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 supported by EPSRC projects EP/K034413/1, EP/K011715/1, EP/L000588/X, EP/N027833/1, EP/N028201/1.

Figure 9. Behavioural property verification: outcome (true/false) and average time (seconds ± std. dev.). The number of states is approximated > 2×10⁶ when the LTS is too big to fit in memory. (4×Intel i7 @ 3.60GHz, 16 GB RAM, mCRL2 2018080.0, 30 runs)

Various π-calculus type systems specialise on accurate (deadlock)-freedom analysis, e.g., [33–36, 55]. [12] type-checks actors with unordered mailboxes, carrying messages of different types; it ensures deadlock-freedom, and (assuming termination) message consumption. Unlike ours, the works above do not support an extensible set of μ-calculus properties (Fig. 7), nor address higher-order processes. Although our actors are similar to Akka Typed (with single-type mailboxes), we conjecture that our types also support actors like [12], with decidable verification (by Lemma 4.7).

Our protocols-as-types are related to session types [3, 25], and their combination with value-dependent and indexed types [10, 13, 66–68]. Our theory has a different design, yielding different features. On the one hand, we do not have an explicit external choice construct (we plan to integrate it via match types [16], but leave it as future work); on the other hand, we can verify liveness properties across interleaved use of multiple channels (more liberally than session types [11]), and we are not limited to linear/confluent protocols: e.g., \( T = p \parallel \tau \parallel o \parallel x \parallel y \parallel T \), \( o \parallel x \parallel z \parallel T \) \( . \mid i \mid \tau \parallel \Pi(T) \parallel \Pi(\text{int}) \parallel \Pi(\text{int}) \) \parallel T \) \parallel \text{types}

parallel processes with a race on channel \( x \); we can verify such processes, capturing that either \( y \) or \( z \) may replace \( z' \) in the \( U \)-typed continuation. This covers locking/mutex protocols, allowing, e.g., to implement and verify Dijkstra’s dining philosophers problem (mentioned in Fig. 9). [4] extends linear logic-based session types with shared channels: it adds non-determinism, weakening deadlock-freedom guarantees.

Outside the realm of process calculi, various works tackle the problem of protocol-aware verification, e.g., [37, 62, 65]. We share similar goals, although we adopt a different theory and design, leading to different tradeoffs: crucially, the 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 dependencies) and automated verification of protocol 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/mutex, mentioned above); but beyond this, a finer type-level representation of state may make model checking undecidable [18], thus requiring the study of decidability conditions, or novel heuristic/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 \( \lambda^\Pi \) 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 LTSmin [32]; 2. support infinite-state systems, trying tools like Bmc [30] (that does not cover the linear-time μ-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 Effπ 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].
References


[42] Julian Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A static verification framework for message passing in Go using
behavioural types. In ICSE. https://doi.org/10.1145/3180155.3180157


DRAFT