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 $\lambda^{\pi}_{\leqslant}$, with a new blend of *behavioural types* (from π-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:
Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Verifying Message-Passing Programs with Dependent Behavioural Types . In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3322484

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

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 `Payment`s.

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), `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* [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*.

```
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. Implemention of the payment service specification (§1). Although similar to Akka Typed [50], it is written in Dotty and Effpi, described in §5; ">>" (l.7) means *"and then."*

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 [64], 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*:

```
Forever[ In[Pay, (p: Pay) =>    // Dependent function type [16]
                 Out[p.replyTo.type, Rejected]
                 | ( Out[aud.type, Audit[p.type]] >>:
                     Out[p.replyTo.type, Accepted] ) ] ]
```

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 `ActorReference`s are used for interacting, and when. Type "`In`" (provided by `Effpi`) requires to wait for a message `p` of type `Pay`, and then either (`|` means *"or"*) send `Rejected` on `p.replyTo`, or send an audit, and then (`>>:`) send `Accepted`. Notably, `p` is bound by a *dependent function type* [16].

Effpi is built upon a concurrent functional calculus for channel-based interaction, called $\lambda^{\pi}_{\leqslant}$; its novelty is a blend of *behavioural types* (inspired by π-calculus literature) with *dependent function types* (inspired by Dotty's foundation $D_{<:}$ [2]), achieving unique specification and verification capabilities. Effpi implements $\lambda^{\pi}_{\leqslant}$ 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") [43]. 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:

```
In[ Audit[_], (a: Audit[_]) => End ]
```

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

$$
\begin{array}{rcl}
\mathbb{B} \;=\; \{\mathbf{tt}, \mathbf{ff}\} & \mathbb{C} \;=\; \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \ldots\} & \mathbb{X} \;=\; \{x, y, z, \ldots\}
\end{array}
$$

$$
\begin{array}{rcl}
\text{terms } \mathbb{T} \ni t, t', \ldots & ::= & \mathbb{X} \mid \mathbb{V} \mid \neg t \mid \mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \\
 & & \mathbf{let}\ x = t\ \mathbf{in}\ t' \mid t\,t' \mid \mathbf{chan}() \mid \mathbb{P} \\
\text{values } \mathbb{V} \ni u, v, \ldots & ::= & \mathbb{B} \mid \mathbb{C} \mid \lambda x.t \mid () \mid \mathbf{err} \\
\text{processes } \mathbb{P} \ni p, q, \ldots & ::= & \mathbf{end} \mid \mathbf{send}(t, t', t'') \mid \mathbf{recv}(t, t') \mid t \parallel t'
\end{array}
$$

Figure 2. Syntax of $\lambda^{\pi}_{\leqslant}$ terms. The set $\mathbb{C}$ (highlighted) contains channel instances, that are part of the run-time syntax.

The issue is that types in $\lambda^{\pi}_{\leqslant}$ 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 $\lambda^{\pi}_{\leqslant}$ calculus, at the basis of Effpi. **§3** presents type system of $\lambda^{\pi}_{\leqslant}$. **§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 $\lambda^{\pi}_{\leqslant}$ 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 $\lambda^{\pi}_{\leqslant}$-Calculus

The theoretical basis of our work is a λ-calculus extended with channels, input/output, and parallel composition, called $\lambda^{\pi}_{\leqslant}$. The "$\pi$" denotes both: *(1)* its use of *dependent function types*, that, together with subtyping $\leqslant$, are cornerstones of its typing system (§3); and *(2)* its connection with the π-calculus [54, 55, 63]. Indeed, $\lambda^{\pi}_{\leqslant}$ is a monadic-style encoding of the higher-order π-calculus: continuations are λ-terms, and this will be helpful for typing (§3) and implementation (§5).

Definition 2.1. The syntax of $\lambda^{\pi}_{\leqslant}$ is in Fig. 2. Elements of $\mathbb{C}$ are run-time syntax. Free/bound variables $\mathrm{fv}(t)/\mathrm{bv}(t)$ 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 $\lambda x.t$, when $x \notin \mathrm{fv}(t)$.

The set of values $\mathbb{V}$ includes booleans $\mathbb{B}$, *channel instances* $\mathbb{C}$, function abstraction, the unit $()$, and **err**or. The terms (in $\mathbb{T}$) can be variables (from $\mathbb{X}$), values (from $\mathbb{V}$), various standard constructs (negation $\neg t$, **if**/**then**/**else**, **let** binding, function application), and also *channel creation* **chan**(), and *process terms* (from $\mathbb{P}$). The primitive **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* **end**, the *output primitive* $\mathbf{send}(t, t', t'')$ (meaning: send t' through t, and continue as t''), the *input primitive* $\mathbf{recv}(t, t')$ (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). $\lambda^{\pi}_{\leqslant}$ can be routinely extended with, e.g., integers, strings, records, variants: we use them in examples.

Example 2.2. A ping-pong system in $\lambda^{\pi}_{\leqslant}$ is written as:

```
let pinger  =  λself.λpongc.(        let ponger  =  λself.(
   send(pongc, self, λ_.(               recv(self, λreplyTo.(
      recv(self, λreply.(                  send(replyTo, "Hi!", λ_.(
         end  )))))                           end  )))))
let sys  =  λy′.λz′.( pinger y′ z′  ∥  ponger z′ )
let main  =  λ_.let y = chan() in  let z = chan() in  sys y z
```

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

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

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

Definition 2.4 (Semantics of $\lambda^{\pi}_{\leqslant}$)**.** *Evaluation contexts* $\mathcal{E}$ and *reduction* $\longrightarrow$ are illustrated in Fig. 3, where *congruence* $\equiv$ is defined as: $t_1 \,\|\, t_2 \equiv t_2 \,\|\, t_1$ and $\mathbf{end} \,\|\, \mathbf{end} \equiv \mathbf{end}$, plus α-conversion. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. We say "*t has an error*" iff $t = \mathcal{E}[\mathbf{err}]$ (for some $\mathcal{E}$). We say "*t is safe*" iff $\forall t' : t \longrightarrow^* t'$ implies t' has no error.

Def. 2.4 is a standard call-by-value semantics, with two rules for concurrency. [R-**chan**()] says that **chan**() returns a fresh channel instance; [R-Comm] says that the parallel composition $\mathbf{send}(\mathrm{a}, u, v_1) \,\|\, \mathbf{recv}(\mathrm{a}, v_2)$, where both sides operate on a 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 $\lambda^{\pi}_{\leqslant}$. Its design is reminiscent of the simply-typed λ-calculus, except that *(1)* we include union types and equi-recursive types, *(2)* we add types for channels and processes, and *(3)* we allow types to contain variables from the term syntax (inspired by $D_{<:}$, the calculus behind Dotty [2]). The syntax of types is in Def. 3.1.

Notably, points *(1)* and *(3)* establish a similarity between $\lambda^{\pi}_{\leqslant}$ and $F_{<:}$ (System 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 $F_{<:}$, polymorphic types can depend on type variables in the environment; hence, we use mutually-defined judgements, akin to those of $F_{<:}$, 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:

$$\mathrm{bool} \;\big|\; () \;\big|\; \top \;\big|\; \bot \;\big|\; T \vee U \;\big|\; \Pi(\underline{x}{:}U)T \;\big|\; \mu\underline{x}.T \;\big|\; \underline{x}$$
$$\mathrm{c}^{\mathrm{io}}[T] \;\big|\; \mathrm{c}^{\mathrm{i}}[T] \;\big|\; \mathrm{c}^{\mathrm{o}}[T]$$
$$\mathbf{proc} \;\big|\; \mathbf{nil} \;\big|\; \mathbf{o}[S, T, U] \;\big|\; \mathbf{i}[S, T] \;\big|\; \mathbf{p}[T, U]$$

Free/bound variables are defined as usual. We write $U\{S/\underline{x}\}$ for the type obtained from U by replacing its free occurrences of $\underline{x}$ with S. If $T = \Pi(\underline{x}{:}U')U$, then $T\,S$ stands for $U\{S/\underline{x}\}$.

We write $\Pi()T$ for $\Pi(\underline{x}{:}())T$ if $\underline{x} \notin \mathrm{fv}(T)$, and distinguish recursion variables as $\mathbf{t}, \mathbf{t}', \ldots$ (i.e., we write $\mu\mathbf{t}.T$). We write $\widetilde{T}$ 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:

$$T \vee U \equiv U \vee T \quad S \vee (T \vee U) \equiv (S \vee T) \vee U \quad \mu\mathbf{t}.T \equiv T\{\mu\mathbf{t}.T/\mathbf{t}\}$$
$$\mathbf{p}[T, U] \equiv \mathbf{p}[U, T] \quad \mathbf{p}[S, \mathbf{p}[T, U]] \equiv \mathbf{p}[\mathbf{p}[S, T], U] \quad \mathbf{p}[T, \mathbf{nil}] \equiv T$$

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

The second row of Def. 3.1 formalises *channel types*: $\mathrm{c}^{\mathrm{io}}[T]$ denotes a channel allowing to input or output values of type T; instead, $\mathrm{c}^{\mathrm{i}}[T]$ only allows for input, and $\mathrm{c}^{\mathrm{o}}[T]$ for output.

$$\begin{array}{rl}\mathcal{E} ::= & [\,] \mid \neg\mathcal{E} \mid \textbf{if } \mathcal{E} \textbf{ then } t_1 \textbf{ else } t_2 \mid \textbf{let } x = \mathcal{E} \textbf{ in } t \mid \textbf{let } x = w \textbf{ in } \mathcal{E} \mid \mathcal{E}\,t \mid w\,\mathcal{E} \\ & \textbf{send}(\mathcal{E}, t, t') \mid \textbf{send}(w, \mathcal{E}, t') \mid \textbf{send}(w, w', \mathcal{E}) \mid \textbf{recv}(\mathcal{E}, t) \mid \textbf{recv}(w, \mathcal{E}) \mid \mathcal{E} \parallel t \quad (\text{where } w, w' \in \mathbb{V} \cup \mathbb{X})\end{array}$$

$$\frac{t_1' \equiv t_1 \quad t_1 \to t_2 \quad t_2 \equiv t_2'}{t_1' \to t_2'}\,[\text{R-}\equiv] \qquad \frac{t \to t'}{\mathcal{E}[t] \to \mathcal{E}[t']}\,[\text{R-}\mathcal{E}] \qquad \begin{array}{l}\neg\textbf{tt} \to \textbf{ff}\ [\text{R-}\neg\textbf{tt}] \\ \neg\textbf{ff} \to \textbf{tt}\ [\text{R-}\neg\textbf{ff}]\end{array} \qquad (\lambda x.t)\,v \to t\{v/x\}\ [\text{R-}\lambda] \qquad \begin{array}{l}\textbf{if tt then } t_1 \textbf{ else } t_2 \to t_1\ [\text{R-if-tt}] \\ \textbf{if ff then } t_1 \textbf{ else } t_2 \to t_2\ [\text{R-if-ff}]\end{array}$$

$$\frac{w \in \mathbb{V} \cup \mathbb{X}}{\textbf{let } x = w \textbf{ in } \mathcal{E}[x] \to \textbf{let } x = w \textbf{ in } \mathcal{E}[w]}\,[\text{R-let}] \qquad \frac{x \notin \text{fv}(t)}{\textbf{let } x = w \textbf{ in } t \to t}\,[\text{R-let}_{\text{GC}}] \qquad \frac{\text{a fresh}}{\textbf{chan}() \to \text{a}}\,[\text{R-chan()}] \qquad \frac{}{\textbf{send}(\text{a}, u, v_1) \parallel \textbf{recv}(\text{a}, v_2) \to v_1\,() \parallel v_2\,u}\,[\text{R-Comm}]$$

$$\frac{v \notin \mathbb{B}}{\neg v \to \textbf{err}} \qquad \frac{u \notin \{\lambda x.t \mid x \in \mathbb{X},\ t \in \mathbb{T}\}}{u\,v \to \textbf{err}} \qquad \frac{v \notin \mathbb{B}}{\textbf{if } v \textbf{ then } t' \textbf{ else } t'' \to \textbf{err}} \qquad \frac{u \notin \mathbb{C}}{\textbf{recv}(u, v) \to \textbf{err}} \qquad \frac{u \notin \mathbb{C}}{\textbf{send}(u, v_1, v_2) \to \textbf{err}} \qquad \frac{t \in \mathbb{V}}{t \parallel t' \to \textbf{err}}$$

Figure 3. Semantics of $\lambda^{\pi}_{\leqslant}$: evaluation contexts $\mathcal{E}$ (top), reduction rules (middle), and error rules (last row).

The third row of Def. 3.1 formalises *process types.* The *generic process type* **proc** denotes any process term; **nil** denotes a terminated process; the *output type* $\mathbf{o}[S, T, U]$ denotes a process that sends a T-typed value on an S-typed channel, and continues as U; the *input type* $\mathbf{i}[S, T]$ denotes a process that receives a value from an S-typed channel and continues as T; the *parallel type* $\mathbf{p}[T, U]$ denotes the parallel composition of two processes (of types T and U).

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

$\vdash \Gamma$ env	Γ is a valid typing environment
$\Gamma \vdash T$ type	T is a valid type in Γ
$\Gamma \vdash \widetilde{T}$ type	holds iff $\forall U \in \widetilde{T} : \Gamma \vdash U$ type
$\Gamma \vdash T$ π-type	T is a valid process type in Γ
$\Gamma \vdash \widetilde{T}$ π-type	holds iff $\forall U \in \widetilde{T} : \Gamma \vdash U$ π-type
$\Gamma \vdash \widetilde{T}$ *-type	holds if $\Gamma \vdash \widetilde{T}$ type or $\Gamma \vdash \widetilde{T}$ π-type
$\Gamma \vdash T \leqslant U$	T is subtype of U in Γ, if $\Gamma \vdash T, U$ *-type
$\Gamma \vdash t : T$	t has type T in Γ

A typing environment Γ maps variables (from $\mathbb{X}$ in Def. 2.1) to types; the order of the entries of Γ is immaterial. All judgements in Fig. 4 are inductive, *except* subtyping, that is *co*inductive (hence the double inference lines). Crucially, in Fig. 4 we have *two* valid type judgements, for *two* kinds of types: $\Gamma \vdash T$ type and $\Gamma \vdash T$ π-type. The former is standard (except for rule [T-c], for valid channel types); the latter distinguishes *process types.* Note that subtyping only relates types of the same kind. Importantly, a typing environment Γ can map a variable to a type (rule [Γ-x]), but *not* to a π-type; this also means that function arguments cannot be π-typed. Still, in a function type $\Pi(\underline{x}{:}T)U$, the return type U can be a π-type (rule [$T\pi$-Π]): i.e., it is possible to define *abstract process types* (cf. Ex. 3.3 and 3.4 later). Rules [T-μ] and [π-μ] are based on [32, §2], and require recursive types to be *contractive*: e.g., $\mu\mathbf{t}_1.\mu\mathbf{t}_2.\dots\mu\mathbf{t}_n.(\mathbf{t}_1 \vee U)$ is not a type; clause "$\underline{x} \notin \text{fv}^{-}(T)$" means that variable $\underline{x}$ is not bound in negative position in T, as in $F_{<:}$ *(Details: [70]).* Recursion is handled by [t-**let**]: in $\textbf{let } x = t \textbf{ in } t'$, term t can refer to x. Rule [$\leqslant$-Π], based on [9], ensures decidability of subtyping [32, §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 [T-$\underline{x}$], the type $\underline{x}$ is valid in Γ; and indeed, by rule [t-x], we can infer $\Gamma \vdash x : \underline{x}$, i.e., the term x has type $\underline{x}$. Intuitively, this means that $\underline{x}$ is the "most precise" type for term x; this is formally supported by the subtyping rule [t-x], that says: as Γ maps term x to T, type $\underline{x}$ is smaller than T. To retrieve from Γ the information that term x has (also) type T, we use subtyping and subsumption (rule [t-$\leqslant$]), as shown here. Since $\underline{x}$ is the smallest type for term x, the judgement $\Gamma \vdash t : \underline{x}$ conveys that t should be "something that evaluates to x," e.g., $t = x$ or $t = \textbf{if tt then } x \textbf{ else } x$; similarly, the dependent function type $\Pi(\underline{x}{:}\text{bool})\underline{x}$ is inhabited by terms like $\lambda x.x$ or $\lambda x.(\lambda y.y)\,x$. Thus, we can roughly say: if $\underline{x}$ occurs in T, then T-typed terms correspondingly use x. This insight will be crucial for our results.

$$\frac{\dfrac{\vdash \Gamma \text{ env}}{\Gamma \vdash x : \underline{x}}\,[t\text{-}x] \qquad \dfrac{\dfrac{\Gamma(x) \equiv T}{\Gamma \vdash \Gamma(x) \leqslant T}\,[\leqslant\text{-refl}]}{\Gamma \vdash \underline{x} \leqslant T}\,[\leqslant\text{-}\underline{x}]}{\Gamma \vdash x : T}\,[t\text{-}\leqslant]$$

Channels, processes, and their types By [t-**chan**], a (type-annotated) term $\textbf{chan}()^{T}$ has type $\text{c}^{\text{io}}[T]$. Rule [t-$\mathbb{C}$] is similar, for channel instances. By [t-**end**], process **end** has type **nil**.

By [t-$\parallel$], both sub-terms of $t_1 \parallel t_2$ are π-typed.

By [t-**send**], $\textbf{send}(t_1, t_2, t_3)$ has type $\mathbf{o}[S, T, U]$, under the validity constraints of rule [π-**o**]. Hence, t_1 has a channel type for sending values of type T, and t_2 (the term being sent) must have type T; also, t_3's type must be $U = \Pi()U'$ (for a π-type U'): i.e., t_3 is a process thunk, run by applying $t_3\,()$.

By [t-**recv**], $\textbf{recv}(t_1, t_2)$ has type $\mathbf{i}[S, T]$, which is well-formed under rule [π-**i**]. Hence, the sub-term t_1 must have a channel type with input U, while t_2 must be an abstract process of type $T = \Pi(\underline{x}{:}U')T'$, with T' π-type. Crucially, by rule [π-**i**], we have $\Gamma \vdash U \leqslant U'$: hence, it is safe to receive a value v from t_1, and apply $t_2\,v$ to get a continuation process that uses v.

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

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

$$\begin{array}{rl} \mathit{pinger} : T_{\mathit{ping}} & = \Pi(\underline{\mathit{self}}{:}\text{c}^{\text{io}}[\text{str}])\,\Pi(\underline{\mathit{pongc}}{:}\text{c}^{\text{o}}[\text{c}^{\text{o}}[\text{str}]]) \\ & \qquad \mathbf{o}\Big[\underline{\mathit{pongc}}, \underline{\mathit{self}}, \mathbf{i}\Big[\underline{\mathit{self}}, \Pi(\underline{\mathit{reply}}{:}\text{str})\mathbf{nil}\Big]\Big] \\ \mathit{ponger} : T_{\mathit{pong}} & = \Pi(\underline{\mathit{self}}{:}\text{c}^{\text{io}}[\text{c}^{\text{o}}[\text{str}]]) \\ & \qquad \mathbf{i}\Big[\underline{\mathit{self}},\ \Pi(\underline{\mathit{replyTo}}{:}\text{c}^{\text{o}}[\text{str}])\,\mathbf{o}\Big[\underline{\mathit{replyTo}}, \text{str}, \Pi()\mathbf{nil}\Big]\Big] \\ \mathit{sys} : T_{\mathit{pp}} & = \Pi(\underline{y}{:}\text{c}^{\text{io}}[\text{str}])\,\Pi(\underline{z}{:}\text{c}^{\text{io}}[\text{c}^{\text{o}}[\text{str}]])\,\mathbf{p}\Big[T_{\mathit{ping}}\,\underline{y}\,\underline{z}\,,\, T_{\mathit{pong}}\,\underline{z}\Big] \end{array}$$

Notice how T_{pp} captures the ping/pong composition of *sys*, preserving its channel topology: the type-level applications

$\vdash \Gamma\ \text{env}$	$\dfrac{}{\vdash \emptyset\ \text{env}}$ [Γ-∅] $\dfrac{\Gamma \vdash T\ \text{type} \quad x \notin \text{dom}(\Gamma)}{\vdash \Gamma, x{:}T\ \text{env}}$ [Γ-x]
$\Gamma \vdash T\ \text{type}$	$\dfrac{\vdash \Gamma\ \text{env} \quad T \in \{\text{bool}, (), \top, \bot\}}{\Gamma \vdash T\ \text{type}}$ [T-BASE] $\dfrac{\vdash \Gamma\ \text{env} \quad \underline{x} \in \text{dom}(\Gamma)}{\Gamma \vdash \underline{x}\ \text{type}}$ [T-$\underline{x}$] $\dfrac{\Gamma, x{:}T \vdash U\ \text{type}}{\Gamma \vdash \Pi(\underline{x}{:}T)U\ \text{type}}$ [T-Π] $\dfrac{\begin{array}{c}\Gamma, x{:}\top \vdash T\ \text{type} \quad \underline{x} \notin \text{fv}^{-}(T) \\ T \notin \{U \mid \exists U', \underline{z} \in \mathbb{X} : U \equiv U' \vee \underline{z}\}\end{array}}{\Gamma \vdash \mu\underline{x}.T\ \text{type}}$ [T-μ] $\dfrac{\Gamma \vdash T\ \text{type} \quad \Gamma \vdash U\ \text{type}}{\Gamma \vdash T \vee U\ \text{type}}$ [T-∨] $\dfrac{\Gamma \vdash T\ \text{type}}{\Gamma \vdash \text{c}^{\text{io}}[T]\ \text{type} \quad \Gamma \vdash \text{c}^{\text{i}}[T]\ \text{type} \quad \Gamma \vdash \text{c}^{\text{o}}[T]\ \text{type}}$ [T-c]
$\Gamma \vdash T\ \pi\text{-type}$	$\dfrac{\vdash \Gamma\ \text{env} \quad T \in \{\mathbf{nil}, \mathbf{proc}\}}{\Gamma \vdash T\ \pi\text{-type}}$ [π-BASE] $\dfrac{\Gamma \vdash S \leqslant \text{c}^{\text{o}}[T_o] \quad \Gamma \vdash T \leqslant T_o \quad \Gamma \vdash U\ \pi\text{-type}}{\Gamma \vdash \mathbf{o}[S, T, \Pi()U]\ \pi\text{-type}}$ [π-o] $\dfrac{\begin{array}{c}\Gamma \vdash S \leqslant \text{c}^{\text{i}}[T_i] \quad \Gamma \vdash T_i \leqslant T \\ \Gamma, x{:}T \vdash U\ \pi\text{-type}\end{array}}{\Gamma \vdash \mathbf{i}\big[S, \Pi(\underline{x}{:}T)U\big]\ \pi\text{-type}}$ [π-i] $\dfrac{\Gamma \vdash T\ \pi\text{-type} \quad \Gamma \vdash U\ \pi\text{-type}}{\Gamma \vdash \mathbf{p}[T, U]\ \pi\text{-type}}$ [π-p] $\dfrac{\Gamma, x{:}T \vdash U\ \pi\text{-type}}{\Gamma \vdash \Pi(\underline{x}{:}T)U\ \text{type}}$ [$T\pi$-Π] $\dfrac{\begin{array}{c}\Gamma, x{:}\top \vdash T\ \pi\text{-type} \quad \underline{x} \notin \text{fv}^{-}(T) \\ T \notin \{U \mid \exists U', \underline{z} \in \mathbb{X} : U \equiv U' \vee \underline{z}\}\end{array}}{\Gamma \vdash \mu\underline{x}.T\ \pi\text{-type}}$ [π-μ] $\dfrac{\begin{array}{c}\Gamma \vdash T\ \pi\text{-type} \\ \Gamma \vdash U\ \pi\text{-type}\end{array}}{\Gamma \vdash T \vee U\ \pi\text{-type}}$ [π-∨]
$\Gamma \vdash T \leqslant U$	$\dfrac{}{\Gamma \vdash T \leqslant \top}$ [⩽-⊤] $\dfrac{}{\Gamma \vdash \bot \leqslant T}$ [⩽-⊥] $\dfrac{T \equiv T'}{\Gamma \vdash T \leqslant T'}$ [⩽-REFL] $\dfrac{\Gamma \vdash T \leqslant S \quad \Gamma \vdash U \leqslant S}{\Gamma \vdash T \vee U \leqslant S}$ [⩽-∨L] $\dfrac{\Gamma \vdash S \leqslant T}{\Gamma \vdash S \leqslant T \vee U}$ [⩽-∨R] $\dfrac{\Gamma \vdash \Gamma(x) \leqslant T}{\Gamma \vdash \underline{x} \leqslant T}$ [⩽-$\underline{x}$] $\dfrac{\Gamma, x{:}T \vdash U \leqslant U'}{\Gamma \vdash \Pi(\underline{x}{:}T)U \leqslant \Pi(\underline{x}{:}T)U'}$ [⩽-Π] $\dfrac{\Gamma \vdash T \leqslant T'}{\Gamma \vdash \text{c}^{\text{io}}[T] \leqslant \text{c}^{\text{i}}[T'] \quad \Gamma \vdash \text{c}^{\text{i}}[T] \leqslant \text{c}^{\text{i}}[T'] \quad \Gamma \vdash \text{c}^{\text{io}}[T'] \leqslant \text{c}^{\text{o}}[T] \quad \Gamma \vdash \text{c}^{\text{o}}[T'] \leqslant \text{c}^{\text{o}}[T]}$ [⩽-c] $\dfrac{}{\Gamma \vdash T \leqslant \mathbf{proc}}$ [⩽-proc] $\dfrac{\Gamma \vdash S \leqslant S' \quad \Gamma \vdash T \leqslant T' \quad \Gamma \vdash U \leqslant U'}{\Gamma \vdash \mathbf{o}[S, T, U] \leqslant \mathbf{o}[S', T', U']}$ [⩽-o] $\dfrac{\Gamma \vdash T \leqslant T' \quad \Gamma \vdash U \leqslant U'}{\Gamma \vdash \mathbf{i}[T, U] \leqslant \mathbf{i}[T', U']}$ [⩽-i] $\dfrac{\Gamma \vdash T \leqslant T' \quad \Gamma \vdash U \leqslant U'}{\Gamma \vdash \mathbf{p}[T, U] \leqslant \mathbf{p}[T', U']}$ [⩽-p]
$\Gamma \vdash t : T$	$\dfrac{\vdash \Gamma, x{:}T\ \text{env}}{\Gamma, x{:}T \vdash x : \underline{x}}$ [t-x] $\dfrac{\vdash \Gamma\ \text{env} \quad v \in \mathbb{B}}{\Gamma \vdash v : \text{bool}}$ [t-$\mathbb{B}$] $\dfrac{\vdash \Gamma\ \text{env}}{\Gamma \vdash () : ()}$ [t-()] $\dfrac{\Gamma \vdash t : \text{bool}}{\Gamma \vdash \neg t : \text{bool}}$ [t-¬] $\dfrac{\Gamma, x{:}U \vdash t : T}{\Gamma \vdash \lambda x^{U}.t : \Pi(\underline{x}{:}U)T}$ [t-λ] $\dfrac{\Gamma \vdash t : T \quad \Gamma \vdash T \leqslant U}{\Gamma \vdash t : U}$ [t-⩽] $\dfrac{\Gamma \vdash T \vee U\ {}^{*}\text{-type} \quad \Gamma \vdash t : \text{bool} \quad \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : U}{\Gamma \vdash \mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 : T \vee U}$ [t-if] $\dfrac{\Gamma \vdash t_1 : \Pi(\underline{x}{:}U)T \quad \Gamma \vdash t_2 : U' \quad \Gamma \vdash U' \leqslant U}{\Gamma \vdash t_1\, t_2 : T\{^{U'}/_{\underline{x}}\}}$ [t-APP] $\dfrac{\Gamma, x{:}U \vdash t : U' \quad \Gamma, x{:}U \vdash t' : T \quad \Gamma \vdash U' \leqslant U}{\Gamma \vdash \mathbf{let}\ x^{U} = t\ \mathbf{in}\ t' : T\{^{U'}/_{\underline{x}}\}}$ [t-let] $\dfrac{\Gamma \vdash \text{c}^{\text{io}}[T]\ \text{type}}{\Gamma \vdash \text{a}^{T} : \text{c}^{\text{io}}[T]}$ [t-ℂ] $\dfrac{\Gamma \vdash \text{c}^{\text{io}}[T]\ \text{type}}{\Gamma \vdash \mathbf{chan}()^{T} : \text{c}^{\text{io}}[T]}$ [t-chan] $\dfrac{\vdash \Gamma\ \text{env}}{\Gamma \vdash \mathbf{end} : \mathbf{nil}}$ [t-end] $\dfrac{\Gamma \vdash \mathbf{p}[T, U]\ \pi\text{-type} \quad \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : U}{\Gamma \vdash t_1 \parallel t_2 : \mathbf{p}[T, U]}$ [t-‖] $\dfrac{\Gamma \vdash \mathbf{o}[S, T, U]\ \pi\text{-type} \quad \Gamma \vdash t_1 : S \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : U}{\Gamma \vdash \mathbf{send}(t_1, t_2, t_3) : \mathbf{o}[S, T, U]}$ [t-send] $\dfrac{\Gamma \vdash \mathbf{i}[S, T]\ \pi\text{-type} \quad \Gamma \vdash t_1 : S \quad \Gamma \vdash t_2 : T}{\Gamma \vdash \mathbf{recv}(t_1, t_2) : \mathbf{i}[S, T]}$ [t-recv]

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

$T_{ping}\,\underline{y}\,\underline{z}$ and $T_{pong}\,\underline{z}$ (yielded by rule [t-APP], Fig. 4) substitute $\underline{y}$ and $\underline{z}$ in T_{ping} and T_{pong}'s bodies (by Def. 3.1). This is obtained by leveraging dependent function types, and is key for combining types/protocols and verifying them (§4).

Example 3.4 (Mobile code). Modern languages and toolkits for message-passing programs support sending/receiving *mobile code* (e.g., [18, 49, 52]). Consider this scenario: a data analysis server lets its clients send custom code, for on-the-fly data filtering. In $\lambda^{\pi}_{\leqslant}$, 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 $\underline{i_1}$ / $\underline{i_2}$, and an output channel $\underline{o}$; it must use $\underline{i_1}$ / $\underline{i_2}$ to input integers $\underline{x}$ / $\underline{y}$, and then it must send one of them along $\underline{o}$, recursively.

$$\begin{array}{l}T_m = \Pi(\underline{i_1}{:}\text{c}^{\text{i}}[\text{int}])\,\Pi(\underline{i_2}{:}\text{c}^{\text{i}}[\text{int}])\,\Pi(\underline{o}{:}\text{c}^{\text{o}}[\text{int}]) \\ \qquad\qquad \mu\mathbf{t}.\mathbf{i}\Big[\underline{i_1}, \Pi(\underline{x}{:}\text{int})\mathbf{i}\Big[\underline{i_2}, \Pi(\underline{y}{:}\text{int})\mathbf{o}\Big[\underline{o}, (\underline{x} \vee \underline{y}), \Pi()\mathbf{t}\Big]\Big]\Big]\end{array}$$

By inspecting T_m, we infer that, e.g., T_m-typed terms cannot be forkbombs; also, "$\underline{x} \vee \underline{y}$" does not allow to send on *out* a value not coming from $\underline{i_1}$ / $\underline{i_2}$ (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.

$$\begin{array}{l}\mathbf{let}\ m_1 = \lambda i_1.\lambda i_2.\lambda o. \\ \quad \mathbf{recv}(i_1, \lambda x.\mathbf{recv}(i_2, \lambda\_.\mathbf{send}(o, x, \lambda\_.\,m_1\ i_2\ i_1\ o))) \\ \mathbf{let}\ m_2 = \lambda i_1.\lambda i_2.\lambda o. \\ \quad \mathbf{recv}(i_1, \lambda x.\mathbf{recv}(i_2, \lambda y.) \\ \qquad \mathbf{send}(o, (\mathbf{if}\ x > y\ \mathbf{then}\ x\ \mathbf{else}\ y), \lambda\_.\,m_2\ i_1\ i_2\ o))\end{array}$$

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:

$$\begin{array}{l}\mathbf{let}\ srv = \lambda cm.\lambda out.\\ \qquad \mathbf{let}\ z_1 = \mathbf{chan}()\ \mathbf{in}\ \ \mathbf{let}\ z_2 = \mathbf{chan}()\ \mathbf{in}\\ \qquad\qquad \mathbf{recv}(cm, \lambda p.\big(p\, z_1\, z_2\, out \parallel prod_1\, z_1 \parallel prod_2\, z_2\big))\end{array}$$

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 srv can only receive a T_m-typed term on cm, we check its type:

$$\emptyset \vdash srv : T_{srv} \quad = \quad \Pi(\underline{cm}{:}\mathrm{c}^{\mathrm{i}}[T_m])\,\Pi(\underline{out}{:}\mathrm{c}^{\mathrm{o}}[\mathrm{int}])\,\mathbf{proc}$$

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

$$\mathbf{send}(x, t, \lambda\_.\mathbf{end}) \parallel srv\ x\ out \qquad \textit{(client sends } t \textit{ to server, via } x\textit{)}$$

is typable in Γ only if $\Gamma \vdash x : \mathrm{c}^{\mathrm{io}}[T_m]$, implying $\Gamma \vdash t : T_m$. We can replace **proc** with a more precise type. If U_1/U_2 are types of $prod_1/prod_2$, the $\mathbf{recv}(\ldots)$ sub-term of srv has type:

$$T'_{srv} \ = \ \mathbf{i}\Big[\underline{cm}, \Pi(\underline{p}{:}T_m)\mathbf{p}\Big[\,\mathbf{p}\Big[T_m\,\underline{z_1}\,\underline{z_2}\,\underline{out}\,,\ U_1\,\underline{z_1}\Big], U_2\,\underline{z_2}\Big]\Big]$$

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, 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, 32]) except the highlighted ones. By rule [$\leqslant$-c], 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 [$\leqslant$-**proc**] says that **proc** is the top type for π-types. Rules [$\leqslant$-**o**]/[$\leqslant$-**i**]/[$\leqslant$-**p**] say that types for input/output/parallel processes are covariant in all parameters.

As usual, supertyping / subsumption (rule [t-$\leqslant$]) 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 $\lambda^{\pi}_{\leqslant}$ terms with a non-Turing-complete type language, for the results in §4.

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

$$\begin{array}{l}t_1 \ = \ \mathbf{send}(x, 42, \lambda\_.\mathbf{end}) \parallel \mathbf{recv}(x, \lambda\_.\mathbf{end})\\ t_2 \ = \ \big(\mathbf{let}\ z = \mathbf{chan}()\ \mathbf{in}\ \mathbf{send}(z, 42, \lambda\_.\mathbf{end})\big) \parallel \mathbf{recv}(x, \lambda\_.\mathbf{end})\\ T_1 \ = \ \mathbf{p}\Big[\,\mathbf{o}\big[\underline{x}, \mathrm{int}, \Pi()\mathbf{nil}\big]\,,\ \mathbf{i}\Big[\underline{x}, \Pi(\underline{y}{:}\mathrm{int})\mathbf{nil}\Big]\Big]\\ T_2 \ = \ \mathbf{p}\Big[\,\mathbf{o}\big[\mathrm{c}^{\mathrm{io}}[\mathrm{int}], \mathrm{int}, \Pi()\mathbf{nil}\big]\,,\ \mathbf{i}\Big[\underline{x}, \Pi(\underline{y}{:}\mathrm{int})\mathbf{nil}\Big]\,\Big]\end{array}$$

Letting $\Gamma = x{:}\mathrm{c}^{\mathrm{io}}[\mathrm{int}]$, we have $\Gamma \vdash \underline{x} \leqslant \mathrm{c}^{\mathrm{io}}[\mathrm{int}]$ and $\Gamma \vdash T_1 \leqslant T_2$. For t_1, we have both $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_1 : T_2$ (by [t-$\leqslant$]): 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 $\mathrm{c}^{\mathrm{io}}[\mathrm{int}]$ 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 [t-**let**] (Fig. 4): since z is bound by **let**..., its occurrence in **send**(...) is typed by a supertype of $\underline{z}$ that is suitable for both z and **chan**() — in this case, $\mathrm{c}^{\mathrm{io}}[\mathrm{int}]$. Specifically:

$$\frac{\Gamma \vdash \mathrm{c}^{\mathrm{io}}[\mathrm{int}] \leqslant \mathrm{c}^{\mathrm{io}}[\mathrm{int}] \qquad \Gamma, z{:}\mathrm{c}^{\mathrm{io}}[\mathrm{int}] \vdash \mathbf{chan}() : \mathrm{c}^{\mathrm{io}}[\mathrm{int}] \qquad \Gamma, z{:}\mathrm{c}^{\mathrm{io}}[\mathrm{int}] \vdash \mathbf{send}(z, 42, \lambda\_.\mathbf{end}) : \mathbf{o}\big[\underline{z}, \mathrm{int}, \Pi()\mathbf{nil}\big]}{\Gamma \vdash \mathbf{let}\ z = \mathbf{chan}()\ \mathbf{in}\ \mathbf{send}(z, 42, \lambda\_.\mathbf{end}) : \mathbf{o}\big[\underline{z}, \mathrm{int}, \Pi()\mathbf{nil}\big]\big\{{}^{\mathrm{c}^{\mathrm{io}}[\mathrm{int}]}/_{\underline{z}}\big\}}\ [t\text{-}\mathbf{let}]$$

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) **err** subterms. In §4, we study how T and T' are related, and how they constraint 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 $\lambda^{\pi}_{\leqslant}$ 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 ϕ in some temporal logic [73]: ϕ could be, e.g., a *safety property* $\Box(\neg\phi')$ ("ϕ' is never true while t runs") or a *liveness property* $\Diamond\phi'$ ("t will eventually satisfy ϕ'"). However, this problem is undecidable (unless ϕ is trivial), since $\lambda^{\pi}_{\leqslant}$ 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_1 in Ex. 3.5 uses $\underline{x}$, we sense that t_1 should interact via x — but by Def. 2.4, t_1 is stuck. To trigger communication, we may bind x in t_1 with a channel instance, e.g., $t'_1 = \mathbf{let}\ x = \mathbf{chan}()\ \mathbf{in}\ t_1$ — but t'_1's type cannot mention $\underline{x}$, hence cannot convey which channel(s) t'_1 uses. Thus, we develop a type-based analysis in four steps: *(1)* we define an over-approximating LTS semantics for typed $\lambda^{\pi}_{\leqslant}$ 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 Γ, t, T), the judgements $\Gamma \vdash t \xrightarrow{\alpha} t'$ and $\Gamma \vdash t \xrightarrow{\tau^\bullet}{}^{*} t'$ 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-→] inherits "concrete" reductions from Def. 2.4: if $t \longrightarrow t'$ is induced by base rule [R], the transition label is τ[R]. 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 π-calculus jargon, it is an *early* semantics [63]). Rule [SR-Comm] lets processes exchange a payload w' via a channel/variable 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); it also records x in the transition label. Rule [SR-λ()] applies a function to a variable x, with the expected substitution. Rule [SR-$\mathcal{E}$] propagates transitions through contexts, unless labels refer to bound variables. Finally, $\Gamma \vdash t \xrightarrow{\tau^\bullet}{}^{*} t'$ holds when t reaches t' via a *finite* sequence of internal moves *excluding interaction*: i.e., labels $w(w')$, $\overline{w}\langle w' \rangle$, $\tau[w]$, and τ[R-Comm] are forbidden.

Using Def. 4.1 on t_1 from Ex. 3.5, we get the transition $\Gamma \vdash t_1 \xrightarrow{\tau[x]} \mathbf{end} \parallel \mathbf{end}$, 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 π-calculus literature [3, 30] — 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 $\Gamma \vdash t_1 \xrightarrow{\tau[x]} \mathbf{end} \parallel \mathbf{end}$. This suggests that a type like $\mathbf{p}\big[\mathbf{o}\big[\underline{x}, \ldots\big], \mathbf{i}\big[\underline{x}, \ldots\big]\big]$ should reduce with a communication on $\underline{x}$. 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 $\mathbf{p}\big[\mathbf{o}\big[\mathrm{c}^{\mathrm{io}}[\mathrm{int}], \ldots\big], \mathbf{i}\big[\underline{x}, \ldots\big]\big]$ should reduce, too. In general, we want $\mathbf{p}[\mathbf{o}[S, \ldots], \mathbf{i}[T, \ldots]]$ to reduce if S and T "might interact", i.e., they could type a same channel/variable: we formalise this idea as $\Gamma \vdash S \bowtie T$ in Def. 4.2.

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

$$\frac{\Gamma \nvdash S \sqcap_\Gamma T \leqslant \bot}{\Gamma \vdash S \bowtie T}\;[\bowtie\text{-c}]$$

A *type reduction context* $\mathcal{E}$ is inductively defined as:

$[\,] \mid \mathbf{o}[\mathcal{E}, T, U] \mid \mathbf{o}[S, \mathcal{E}, U] \mid \mathbf{o}[S, T, \mathcal{E}] \mid \mathbf{i}[\mathcal{E}, T] \mid \mathbf{i}[S, \mathcal{E}] \mid \mathbf{p}[\mathcal{E}, T]$

Judgements $\Gamma \vdash T \xrightarrow{\alpha} T'$ and $\Gamma \vdash T \xrightarrow{\tau[\vee]}{}^{*} T'$ are in Fig. 6.

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

- [T→**io**x] fires when, in $\mathbf{p}[U, U']$, there might be an interaction with a type variable $\underline{x}$ as payload. Note that, by [T→**i**], the $\underline{x}$ 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;
- [T→**io**] is similar, but fires if the payload T is *not* a variable.

Finally, $\Gamma \vdash T \xrightarrow{\tau[\vee]}{}^{*} T'$ holds if T reaches T' via a finite sequence of internal choices $\tau[\vee]$.

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

$$\begin{aligned}
\Gamma &= y{:}\mathrm{c}^{\mathrm{io}}[\mathrm{str}]\,,\; z{:}\mathrm{c}^{\mathrm{io}}[\mathrm{c}^{\mathrm{o}}[\mathrm{str}]]\\
t &= \mathit{sys}\; y\; z\\
T &= T_{pp}\; \underline{y}\; \underline{z} \;=\; \mathbf{p}\left[\begin{array}{l}\mathbf{o}\Big[\underline{z}, \underline{y}, \mathbf{i}\Big[\underline{y}, \Pi(\underline{\mathit{reply}}{:}\mathrm{str})\mathbf{nil}\Big]\Big],\\ \mathbf{i}\Big[\underline{z}, \Pi(\underline{\mathit{replyTo}}{:}\mathrm{c}^{\mathrm{o}}[\mathrm{str}])\,\mathbf{o}\Big[\underline{\mathit{replyTo}}, \mathrm{str}, \Pi()\mathbf{nil}\Big]\Big]\end{array}\right]
\end{aligned}$$

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

$$\Gamma \vdash t \xrightarrow{\tau[z]} \xrightarrow{\tau^\bullet}{}^{*} \begin{pmatrix}\mathbf{recv}(y, \ldots) \;\parallel\\ \mathbf{send}(y, \texttt{"Hi!"}, \ldots)\end{pmatrix} \xrightarrow{\tau[y]} \xrightarrow{\tau^\bullet}{}^{*} \begin{pmatrix}\mathbf{end} \parallel\\ \mathbf{end}\end{pmatrix}$$

By Def. 4.2, applying rule [T→**io**x] twice, we get:

$$\Gamma \vdash T \xrightarrow{\tau[\underline{z}, \underline{z}]} \mathbf{p}\left[\begin{array}{l}\mathbf{i}\Big[\underline{y}, \Pi(\underline{\mathit{reply}}{:}\mathrm{str})\mathbf{nil}\Big],\\ \mathbf{o}\Big[\underline{\mathit{replyTo}}, \mathrm{str}, \Pi()\mathbf{nil}\Big]\Big\{\underline{y}/\underline{\mathit{replyTo}}\Big\}\end{array}\right] \xrightarrow{\tau[\underline{y}, \underline{y}]} \mathbf{p}\begin{bmatrix}\mathbf{nil},\\ \mathbf{nil}\end{bmatrix}$$

Observe that T closely mimicks the transitions of t: the type-level substitution of $\underline{y}$ in place of $\underline{\mathit{replyTo}}$ 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 $\mathbf{i}[\ldots, \ldots]$.

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 $\Gamma \vdash t \xrightarrow{\alpha} t'$ implies $\Gamma \vdash t' : T$. Otherwise, when $\Gamma \vdash T$ π-type, we have:*

1. $\Gamma \vdash t \xrightarrow{\alpha} t'$ *with* $\tau^\bullet(\alpha)$ *(Fig. 5) implies* $\Gamma \vdash t' : T$;
2. $\Gamma \vdash t \xrightarrow{\alpha} t'$ *and* $\alpha \in \{\overline{x}\langle w \rangle, x(w), \tau[x], \tau_{\text{[R-Comm]}}\}$ *implies one:*
   a. $\Gamma \vdash t' : T$ *and* $\mathbf{proc} \in T$;
   b. $\alpha = \overline{x}\langle w \rangle$ *and* $\exists S, U, T' : \Gamma \vdash x : S, w : U, t' : T'$ *and* $\Gamma \vdash T \xrightarrow{\tau[\vee]}{}^{*} \xrightarrow{\overline{S}\langle U \rangle} T'$;
   c. $\alpha = x(w)$ *and* $\exists S, U, T' : \Gamma \vdash x : S, w : U, t' : T'$ *and* $\Gamma \vdash T \xrightarrow{\tau[\vee]}{}^{*} \xrightarrow{S(U)} T'$;
   d. $\alpha = \tau[x]$ *and* $\exists S, S', T' : \Gamma \vdash x : S, x : S', t' : T'$ *and* $\Gamma \vdash T \xrightarrow{\tau[\vee]}{}^{*} \xrightarrow{\tau[S, S']} T'$;
   e. $\alpha = \tau_{\text{[R-Comm]}}$ *and* $\exists S, S', T' : \{S, S'\} \nsubseteq \mathbb{X}$, $\Gamma \vdash t' : T'$ *and* $\Gamma \vdash T \xrightarrow{\tau[\vee]}{}^{*} \xrightarrow{\tau[S, S']} T'$.

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 $\lambda^{\pi}_{\leqslant}$ (hypothesis $\Gamma \vdash T$ type, or case 1), then t' has the same

$$\dfrac{t \longrightarrow t' \ \text{ by base rule [R]}}{\Gamma \vdash t \xrightarrow{\tau[\mathrm{R}]} t'}\ [\text{SR-}\!\longrightarrow] \qquad \begin{array}{l}\Gamma \vdash \neg x \xrightarrow{\tau[\neg x]} \mathbf{tt} \\ \Gamma \vdash \neg x \xrightarrow{\tau[\neg x]} \mathbf{ff}\end{array} \qquad \begin{array}{l}\Gamma \vdash \mathbf{if}\ x\ \mathbf{then}\ t\ \mathbf{else}\ t' \xrightarrow{\tau[\mathbf{if}\ x]} t \\ \Gamma \vdash \mathbf{if}\ x\ \mathbf{then}\ t\ \mathbf{else}\ t' \xrightarrow{\tau[\mathbf{if}\ x]} t'\end{array} \qquad \dfrac{w, w', w'' \in \mathbb{X} \cup \mathbb{V}}{\Gamma \vdash \mathbf{send}(w, w', w'') \xrightarrow{\overline{w}\langle w'\rangle} w''\,()}\ [\text{SR-}\mathbf{send}]$$

$$\dfrac{w, w', w'' \in \mathbb{X} \cup \mathbb{V} \quad \Gamma \vdash w : \mathrm{c}^{\mathrm{i}}[T] \quad \Gamma \vdash w' : T}{\Gamma \vdash \mathbf{recv}(w, w'') \xrightarrow{w(w')} w''\, w'}\ [\text{SR-}\mathbf{recv}] \qquad \dfrac{\Gamma \vdash t \xrightarrow{\overline{w}\langle w'\rangle} t' \quad \Gamma \vdash t'' \xrightarrow{w(w')} t'''}{\Gamma \vdash t \parallel t'' \xrightarrow{\tau[w]} t' \parallel t'''}\ [\text{SR-Comm}]$$

$$\dfrac{\Gamma \vdash x\,w : T \quad w \in \mathbb{X} \cup \mathbb{V} \quad \Gamma \vdash v\{w/y\} : T}{\Gamma \vdash x\,w \xrightarrow{\tau[x()]} v\{w/y\}}\ [\text{SR-}x()] \qquad \dfrac{}{\Gamma \vdash (\lambda y.t)\,x \xrightarrow{\tau[\lambda()]} t\{x/y\}}\ [\text{SR-}\lambda()] \qquad \dfrac{\Gamma \vdash t' \xrightarrow{\alpha} t'' \quad \mathrm{fv}(\alpha) \cap \mathrm{bv}(\mathcal{E}) = \emptyset}{\Gamma \vdash \mathcal{E}[t] \xrightarrow{\alpha} \mathcal{E}[t']}\ [\text{SR-}\mathcal{E}]$$

$$\dfrac{}{\Gamma \vdash t \xrightarrow{\tau^\bullet}{}^{*}\, t} \qquad \dfrac{\Gamma \vdash t \xrightarrow{\tau^\bullet}{}^{*}\, t' \quad \Gamma \vdash t' \xrightarrow{\alpha} t'' \quad \tau^\bullet(\alpha)}{\Gamma \vdash t \xrightarrow{\tau^\bullet}{}^{*}\, t''} \qquad \text{where } \tau^\bullet(\alpha) \text{ holds iff } \alpha \in \{\tau[\neg x], \tau[\mathbf{if}\ x], \tau[x()], \tau[\lambda()], \tau[\mathrm{R}] \mid x \in \mathbb{X},\ [\mathrm{R}] \neq [\text{R-Comm}]\}$$

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

$$\dfrac{}{\Gamma \vdash T \vee U \xrightarrow{\tau[\vee]} T} \qquad \dfrac{\Gamma \vdash T \xrightarrow{\alpha} T'}{\Gamma \vdash \mathcal{E}[T] \xrightarrow{\alpha} \mathcal{E}[T']} \qquad \dfrac{T' \equiv T \quad \Gamma \vdash T \xrightarrow{\alpha} U \quad U \equiv U'}{\Gamma \vdash T' \xrightarrow{\alpha} U'} \qquad \dfrac{}{\Gamma \vdash \mathbf{o}[S, T, \Pi()U] \xrightarrow{\overline{S}\langle T\rangle} U}\ [T\!\to\!\mathbf{o}]$$

$$\dfrac{\Gamma \vdash T' \leqslant T \qquad T' = T \ \text{ or } \ T' \in \mathbb{X}}{\Gamma \vdash \mathbf{i}\big[S, \Pi(\underline{x}{:}T)U\big] \xrightarrow{S(T')} U\{T'/\underline{x}\}}\ [T\!\to\!\mathbf{i}] \qquad \dfrac{\Gamma \vdash U \xrightarrow{\overline{S}\langle \underline{x}\rangle} U' \quad \Gamma \vdash U'' \xrightarrow{S'(\underline{x})} U''' \quad \Gamma \vdash S \bowtie S'}{\Gamma \vdash \mathbf{p}[U, U''] \xrightarrow{\tau[S, S']} \mathbf{p}[U', U''']}\ [T\!\to\!\mathbf{io}x]$$

$$\dfrac{\Gamma \vdash U \xrightarrow{\overline{S}\langle T\rangle} U' \quad \Gamma \vdash U'' \xrightarrow{S'(T')} U''' \quad \Gamma \vdash S \bowtie S' \quad \Gamma \vdash T \leqslant T' \quad T \notin \mathbb{X}}{\Gamma \vdash \mathbf{p}[U, U''] \xrightarrow{\tau[S, S']} \mathbf{p}[U', U''']}\ [T\!\to\!\mathbf{io}] \qquad \dfrac{}{\Gamma \vdash T \xrightarrow{\tau[\vee]}{}^{*}\, T} \qquad \dfrac{\Gamma \vdash T \xrightarrow{\tau[\vee]}{}^{*}\, T' \qquad \Gamma \vdash T' \xrightarrow{\tau[\vee]} T''}{\Gamma \vdash T \xrightarrow{\tau[\vee]}{}^{*}\, T''}$$

Figure 6. Semantics of $\lambda^{\pi}_{\leqslant}$ types. We will sometimes use label τ to denote either $\tau[\vee]$ or $\tau[S, S']$ (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 $\tau[\vee]$ moves (cases 2b–2e); the exception is case 2a: if t' keeps type T, then that T syntactically contains **proc**, which types a reducing sub-term of t before and after its reduction (via rule [t-$\leqslant$]).

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* $\lambda^{\pi}_{\leqslant}$ terms, i.e., all functions must be total (always return a value or process when applied). This means that, e.g., if $\Gamma \vdash t : \mathbf{o}\big[\underline{x}, \mathrm{int}, T'\big]$, then t *will* output on x; this excludes cases like $t = \mathbf{if}\ \omega\ \mathbf{then}\ \mathbf{send}(x, 42, t')\ \mathbf{else}\ \mathbf{send}(x, 43, t'')$ (with $\omega = (\lambda y.y\,y)\,(\lambda z.z\,z)$). Productivity is obtained with many methods from literature (e.g., [21, 72]);
- (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 $\lambda^{\pi}_{\leqslant}$, assume $\Gamma \vdash t : T$ and $\Gamma \vdash T$ π-type. Then:*

1. *$\Gamma \vdash T \xrightarrow{\overline{x}\langle U\rangle} T'$ implies $\exists w, t' : \Gamma \vdash w : U, t' : T'$ and $\Gamma \vdash t \xrightarrow{\tau^\bullet}{}^{*} \xrightarrow{\overline{x}\langle w\rangle} t'$;*
2. *$\Gamma \vdash T \xrightarrow{x(U)} T'$ implies $\forall w$: if $\Gamma \vdash w : U$, then $\exists t'$: $\Gamma \vdash t' : T'$ and $\Gamma \vdash t \xrightarrow{\tau^\bullet}{}^{*} \xrightarrow{x(w)} t'$;*
3. *$\Gamma \vdash T \xrightarrow{\tau[x,x]} T'$ implies $\exists t'$ such that $\Gamma \vdash t' : T'$ and $\Gamma \vdash t \xrightarrow{\tau^\bullet}{}^{*} \xrightarrow{\tau[x]} t'$;*
4. *$\Gamma \vdash T \xrightarrow{\tau[\vee]}$ implies either: (a) $\exists T' : \Gamma \vdash T \xrightarrow{\tau[\vee]} T'$ and $\Gamma \vdash t : T'$; or, (b) $\exists t' : \Gamma \vdash t \xrightarrow{\alpha} t'$ with $\tau^\bullet(\alpha)$ (Fig. 5) and $\Gamma \vdash t' : T$; or, (c) $\exists T' : \Gamma \vdash T \xrightarrow{\alpha} T'$ with $\alpha \neq \tau[\vee]$.*

Items 1–3 of Thm. 4.5 say that if T can input/output/interact, then t can do the same, possibly after a sequence of τ-steps (without communication, cf. Def. 4.1); the τ-sequence is finite, since t is productive by hypothesis. By item 4, if T can make a choice ($\vee$), 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 μ-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 μ-calculus is decidable for our types, with minimal restrictions (Lemma 4.7).

Definition 4.6 (Linear-time μ-calculus). Given a *set of actions* $\mathbb{A}\mathrm{ct}$ ranged over by α, the *linear-time μ-calculus formulas* are defined as follows (where $\mathbb{A}$ is a subset of $\mathbb{A}\mathrm{ct}$):

$$\begin{array}{ll} \textit{Basic formulas:} & \phi \ ::= \ \mathsf{Z} \ \big|\ \neg\phi \ \big|\ \phi_1 \wedge \phi_2 \ \big|\ (\alpha)\phi \ \big|\ \nu\mathsf{Z}.\phi \\ \begin{array}{l}\textit{Derived} \\ \textit{formulas}\end{array} & \left\{ \begin{array}{l} \top \ \big|\ \bot \ \big|\ \phi_1 \vee \phi_2 \ \big|\ \phi_1 \Rightarrow \phi_2 \ \big|\ \mu\mathsf{Z}.\phi \\ (\mathbb{A})\phi \ \big|\ (-\mathbb{A})\phi \ \big|\ \phi_1 \,\mathsf{U}\, \phi_2 \ \big|\ \Box\phi \ \big|\ \Diamond\phi \end{array} \right. \end{array}$$

In Def. 4.6, ϕ describes accepted sequences of actions; ϕ 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 $\nu Z.\phi$. Basic formulas are enough [6, 73] to derive true/false (accept any/no sequence of actions), disjunction, implication, least fixed points $\mu Z.\phi$; $(\mathbb{A})\phi$ accepts sequences that start with any $\alpha \in \mathbb{A}$, then satisfy ϕ; dually, $(-\mathbb{A})\phi$ requires $\alpha \in \mathbb{A}\mathrm{ct} \setminus \mathbb{A}$. We also derive usual temporal formulas $\phi_1 \mathbin{\mathsf{U}} \phi_2$ (“ϕ_1 holds, until ϕ_2 eventually holds”), $\Box\phi$ (“ϕ is always true”), and $\Diamond\phi$ (“ϕ is eventually true”). Given a process p with LTS of labels $\mathbb{A}\mathrm{ct}$, 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 ϕ accepts all runs of p. *(Details: [70])*

We can decide ϕ on a *guarded type* T, as shown in Lemma 4.7. Here, we instantiate $\mathbb{A}\mathrm{ct}$ (Def. 4.6) as $\mathbb{A}_\Gamma(T)$, which is the set of labels fired along T’s transitions in Γ, (Def. 4.2); notably, $\mathbb{A}_\Gamma(T)$ is *finite* and syntactically determined. *(Details: [70])*

Lemma 4.7. *Given Γ, we say that T* is guarded *iff, for all π-type subterms $\mu\mathbf{t}.U$ of T, $\mathbf{t}$ can occur in U only as subterm of $\mathbf{i}[\ldots]$ or $\mathbf{o}[\ldots]$; then, if T is guarded, $T \models \phi$ is decidable.*

Lemma 4.7 holds since guarded π-types are encodable in CCS without restriction [53], then in Petri nets [22, §4.1], for which linear-time μ-calculus is decidable [20]. Notably, Lemma 4.7 covers *infinite-state* types (with $\mathbf{p}[\ldots,\ldots]$ under $\mu\mathbf{t}.\ldots$), that type $\lambda^{\pi}_{\leqslant}$ terms with unbounded parallel subterms.

Now, assuming $\Gamma \vdash t : T$, 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:

(i1) if we do *not* want t to perform an action on channel x, we check that T never *potentially uses* type variable $\underline{x}$;

(i2) if we *want* t to eventually perform an action on channel x, we need t productive, and check that T eventually uses $\underline{x}$ — without doing “imprecise” actions before.

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

Definition 4.8. The *input / output uses of $\underline{x}$ by T in Γ* are:

input uses: $\mathbb{U}^{\mathbf{i}}_{\Gamma,T}(\underline{x}) = \{S'(U') \in \mathbb{A}_\Gamma(T) \mid \Gamma \vdash \underline{x} \leqslant S'\}$

output uses: $\mathbb{U}^{\mathbf{o}}_{\Gamma,T}(\underline{x}) = \{\overline{S'}\langle U'\rangle \in \mathbb{A}_\Gamma(T) \mid \Gamma \vdash \underline{x} \leqslant S'\}$

Definition 4.9. Given a set of type (resp. term) variables $\mathbb{Y}$, the $\mathbb{Y}$*-limited transitions of T (resp. t) in Γ* are:

$$\frac{\Gamma \vdash T \xrightarrow{\alpha} T' \quad \forall S, U : \alpha \in \{S(U), \overline{S}\langle U\rangle\} \text{ implies } S \in \mathbb{Y}}{T \uparrow_\Gamma \mathbb{Y} \xrightarrow{\alpha} T' \uparrow_\Gamma \mathbb{Y}}$$

$$\frac{\Gamma \vdash t \xrightarrow{\alpha} t' \quad \forall w, w' : \alpha \in \{w(w'), \overline{w}\langle w'\rangle\} \text{ implies } w \in \mathbb{Y}}{t \uparrow_\Gamma \mathbb{Y} \xrightarrow{\alpha} t' \uparrow_\Gamma \mathbb{Y}}$$

Theorem 4.10. *Within productive $\lambda^{\pi}_{\leqslant}$, assume $\Gamma \vdash t : T$, with $\Gamma \vdash T$ π-type, $\mathbf{proc} \notin T$. Also assume, for all $\mathbf{i}[S, \Pi(\underline{x}{:}U)U']$ occurring in T, that there is y such that $\Gamma \vdash y : U$ holds.*[1] *For μ-calculus judgements on T, let $\mathbb{A}\mathrm{ct} = \mathbb{A}_\Gamma(T)$, and $\mathbb{A}_\tau = \left\{\tau[S, S'] \in \mathbb{A}_\Gamma(T) \,\middle|\, \{S, S'\} \not\subseteq \mathrm{dom}(\Gamma)\right\}$. Then, the implications in Fig. 7 hold.*

Assume $\Gamma \vdash t : T$. The sets $\mathbb{U}^{\mathbf{i}}_{\Gamma,T}(\underline{x}) / \mathbb{U}^{\mathbf{o}}_{\Gamma,T}(\underline{x})$ in Def. 4.8 contain all transition labels that *might* be fired by T, when x is used for input/output by t. The operator $\uparrow_\Gamma \{x_i\}_{i\in 1..n}$ (Def. 4.9) limits the observable inputs/outputs of T/t to those occurring on channel x_i — while other (open) channels can only reduce by communicating, via τ-actions; i.e., $x_1, \ldots, x_n$ are interfaces to other types/processes, and are “probed” for verification (this is common in model checking tools).

In Thm. 4.10, item (1) can be seen as a case of intuition (i1) above: if T never fires a label $(\Box(\neg\ldots))$ that is a *potential output use* of $\underline{x_i}$ ($i \in 1..n$), then t never uses x_i for output. The “potential output use”, by Def. 4.8, is any label $\overline{S'}\langle U'\rangle$ fired by T where S' is a supertype of $\underline{x}$: this accounts for “imprecise typing”, discussed in Ex. 3.5. Item (3) of Thm. 4.10 is a case of intuition (i2): to ensure that t eventually outputs on x_i ($i \in 1..n$), we check that T eventually fires a label $\overline{\underline{x}}\langle U\rangle$; moreover, we check T does *not* fire any label in $\mathbb{A}_\tau$, until (U) the output $\overline{\underline{x}}\langle U\rangle$ occurs. The set $\mathbb{A}_\tau$ contains all “imprecise” synchronisation labels $\tau[S, S']$ where either S or S' is *not* a type variable: we exclude them because, if T fires one, then we cannot use Thm. 4.5(3) to ensure that t reduces accordingly; i.e., if we do *not* exclude $\mathbb{A}_\tau$, then t might deadlock and never perform $\overline{x_i}\langle w\rangle$ (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 $\underline{z}$ on a channel S (representing a *potential* use of $\underline{x}$), then T eventually fires $\overline{\underline{y}}\langle \underline{z}\rangle$ — without doing *potential* inputs on $\underline{x}$, nor firing any label in $\mathbb{A}_\tau$, before.

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

Take *ponger* (Ex. 2.2), T_{pong} (Ex. 3.3), and $\Gamma = z{:}\mathrm{c}^{\mathrm{io}}[\mathrm{c}^{\mathrm{o}}[\mathrm{str}]]$. To ensure that the term *ponger* z is responsive on z, we check $(T_{pong}\ \underline{z}) \uparrow_\Gamma \{\underline{z}\} \models \phi$, with ϕ in Fig. 7(6) (right).

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

5 Implementation and Evaluation

We designed $\lambda^{\pi}_{\leqslant}$ 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] This implicitly requires $\Gamma \vdash U$ type, hence $\mathrm{fv}(U) \cap \mathrm{bv}(T) = \emptyset$: 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 $\Gamma \vdash y : U$ can be assumed w.l.o.g.: if $\Gamma \vdash t : T$ but $\nexists y$ such that $\Gamma \vdash y : U$, we can pick $y' \notin \mathrm{dom}(\Gamma)$, extend Γ as $\Gamma' = \Gamma, y'{:}U$, and get $\Gamma' \vdash y' : U$ and $\Gamma' \vdash t : T$.

(1) Non-usage 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)*

$t \uparrow_\Gamma \{x_i\}_{i \in 1..n} \models \Box(\neg(\bigvee_{i \in 1..n} (\overline{x_i}\langle w \rangle)\top))$ | $T \uparrow_\Gamma \{\underline{x_i}\}_{i \in 1..n} \models \Box\Big(\neg\Big(\bigvee_{i \in 1..n} (\mathbb{U}^{\mathbf{o}}_{\Gamma,T}(\underline{x_i}))\top\Big)\Big)$

(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.

$t \uparrow_\Gamma \{x_i\}_{i \in 1..n} \models \Box((\tau)\top \vee \bigvee_{i \in 1..n} (x_i(w) \cup \overline{x_i}\langle w \rangle)\top)$ | $T \uparrow_\Gamma \{\underline{x_i}\}_{i \in 1..n} \models \Box(-\mathbb{A}_\tau)\top \wedge \Box((\tau)\top \vee \bigvee_{i \in 1..n} (\{\underline{x_i}(U'), \overline{\underline{x_i}}\langle U' \rangle \mid \text{any } U'\})\top)$

(3) Eventual usage of $x_1, \ldots, x_n$**:** some x_i ($i \in 1..n$) is used for output, while t runs. *(Simple variations: use some* x_i *for input or communication)*

$t \uparrow_\Gamma \{x_i\}_{i \in 1..n} \models \Diamond(\bigvee_{i \in 1..n} (\overline{x_i}\langle w \rangle)\top)$ | $T \uparrow_\Gamma \{\underline{x_i}\}_{i \in 1..n} \models (-\mathbb{A}_\tau)\top \mathbin{\mathsf{U}} (\bigvee_{i \in 1..n} (\{\overline{\underline{x_i}}\langle U' \rangle \mid \text{any } U'\})\top)$

(4) Forwarding from x **to** y**:** whenever some z is received from x, it is eventually forwarded via y, before x is used for input again.

$t \uparrow_\Gamma \{x, y\} \models \Box((x(z))\top \Rightarrow ((-x(w))\top \mathbin{\mathsf{U}} (\overline{y}\langle z \rangle)\top))$ | $T \uparrow_\Gamma \{\underline{x}, \underline{y}\} \models \Box\Big((\{S(\underline{z}) \mid S(\underline{z}) \in \mathbb{U}^{\mathbf{i}}_{\Gamma,T}(\underline{x})\})\top \Rightarrow \Big((-(\mathbb{A}_\tau \cup \mathbb{U}^{\mathbf{i}}_{\Gamma,T}(\underline{x})))\top \mathbin{\mathsf{U}} (\overline{\underline{y}}\langle \underline{z} \rangle)\top\Big)\Big)$

(5) Reactiveness on x**:** t runs forever, and is always eventually able to receive inputs from x (possibly after a finite number of τ-steps).

$t \uparrow_\Gamma \{x\} \models \Box((\tau)\top \mathbin{\mathsf{U}} (x(w))\top)$ | $T \uparrow_\Gamma \{\underline{x_i}\} \models \Box(-\mathbb{A}_\tau)\top \wedge \Box((\tau)\top \vee (\{\underline{x}(U') \mid \text{any } U'\})\top)$

(6) Responsiveness on x**:** whenever some z is received from x, it is eventually used to send a response, before x is used for input again.

$t \uparrow_\Gamma \{x\} \models \Box((x(z))\top \Rightarrow ((-x(w))\top \mathbin{\mathsf{U}} (\overline{z}\langle w \rangle)\top))$ | $T \uparrow_\Gamma \{\underline{x}\} \models \Box\Big((\{S(\underline{z}) \mid S(\underline{z}) \in \mathbb{U}^{\mathbf{i}}_{\Gamma,T}(\underline{x})\})\top \Rightarrow \Big((-(\mathbb{A}_\tau \cup \mathbb{U}^{\mathbf{i}}_{\Gamma,T}(\underline{x})))\top \mathbin{\mathsf{U}} (\{\overline{\underline{z}}\langle U' \rangle \mid \text{any } U'\})\top\Big)\Big)$

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 $\mathbb{V} \cup \mathbb{X}$, and we write $\overline{x}\langle w \rangle$ as shorthand for the (infinite) set of labels $\{\overline{x}\langle w \rangle \mid w \in \mathbb{V} \cup \mathbb{X}\}$ (and similarly for $x(w)$). For brevity, in (4) and (6) we write $(\alpha)\top \Rightarrow \phi$ instead of $(\alpha)\top \Rightarrow (\alpha)\phi$ (i.e., if we observe α, then ϕ holds afterwards).

foundation $D_{<:}$ [2]. This naturally leads to a three-step implementation strategy: *(1)* internal embedding of $\lambda^{\pi}_{\leqslant}$; *(2)* actor-based APIs, via syntactic sugar; and *(3)* compiler plugin for type-level model checking. The result is a software toolkit called `Effpi`, available at: **https://alcestes.github.io/effpi**

5.1 Implementation

A payoff of the $\lambda^{\pi}_{\leqslant}$ design 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 $\lambda^{\pi}_{\leqslant}$ 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. As usual for internal language embeddings, the `Effpi` DSL 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 `In[...]` (corresponding to **i**[...] 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 $\lambda^{\pi}_{\leqslant}$ semantics (Def. 2.4).

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

```
def ponger(self: T): T1 = {
  receive(self) { replyTo =>
    send(replyTo, "Hi!") >>
    end } }
```

$$\Pi(\underline{x}{:}T)\mathbf{o}\Big[\underline{y}, \underline{x}, T'\Big] \rightsquigarrow \texttt{(x:T)=>Out[y.type, x.type, T']}$$

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

For better usability, `Effpi` also provides some extensions over $\lambda^{\pi}_{\leqslant}$, 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. Also notice how the type of `f`'s argument (`x.type`) is passed to `U`, and then to `T`: consequently, the type of `f` expands into `Par[Out[x.type, Int],Out[x.type, Int]]`.

```
type T[X <: Chan[Int]] = Out[X, Int]
type U[Y <: Chan[Int]] = Par[T[Y],T[Y]]
def f(x: OChan[Int]): U[x.type] = ...
```

To guide `Effpi`'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 `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 $\lambda^{\pi}_{\leqslant}$, input/output actions and their continuations are represented by λ-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 `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 $\lambda^{\pi}_{\leqslant}$ EDSL, `Effpi` 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:

- A is a process with a unique, *implicit* input channel m, of type $c^{i}[T]$ (Def. 3.1). Hence, A can only use m to receive messages of type T — i.e., m is A's mailbox;
- A receives T-typed messages by calling `read` — which is syntactic sugar for $\mathbf{recv}(m, \ldots)$ (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 $c^{o}[T]$ (Def. 3.1): it only allows to send messages of type T.

To this purpose, `Effpi` uses Dotty's implicit function types [57]: i.e., type `Actor[...]` in Fig. 1 hides an input channel.

Type-level model checking The implementation details discussed thus far cover the $\lambda^{\pi}_{\leqslant}$ 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 [59]) accessing the typed program AST. The plugin looks for methods annotated with "`@effpi.verifier.verify`":

```
@effpi.verifier.verify(ϕ)
def f(x:…, y:…): T = …
```

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

1. tries to convert `T` into a $\lambda^{\pi}_{\leqslant}$ type T, as per Def. 3.1;
2. checks if $T \models \phi'$ holds — where ϕ' is the companion formula of ϕ in Fig. 7 (right). This step uses the mCRL2 model checker [23]: we encode T into an mCRL2 process,[2] and check if ϕ' 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 ϕ 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_{ping}, 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 `Accept`s 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, liveness properties might not hold at run-time (cf. condition (c1) in §4).
2. It does not verify processes with unbounded parallel components (i.e., with parallel composition under recursion);[3] hence, it rejects types having $\mathbf{p}[\ldots, \ldots]$ under $\mu\mathbf{t}.\ldots$. This does not impact the examples in this paper.
3. It uses iso-recursive types [60, Ch. 21] because, unlike $\lambda^{\pi}_{\leqslant}$ (Def. 3.2), Dotty does not have equi-recursive types.

Limitations 1 and 3 might be avoided by implementing $\lambda^{\pi}_{\leqslant}$ 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 [48]), 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 [31], with diverse interaction patterns:

- *chameneos: n* actors ("chameneos") connect to a central broker, who picks pairs and sends them their respective `ActorReference`s, so they can interact peer-to-peer [34];
- *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.

---

[2] 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.

[3] This is because mCRL2 checks formulas of the branching-time μ-calculus, on finite-state systems. We are not aware of model checkers focused on the linear-time μ-calculus, and supporting infinite-state systems.

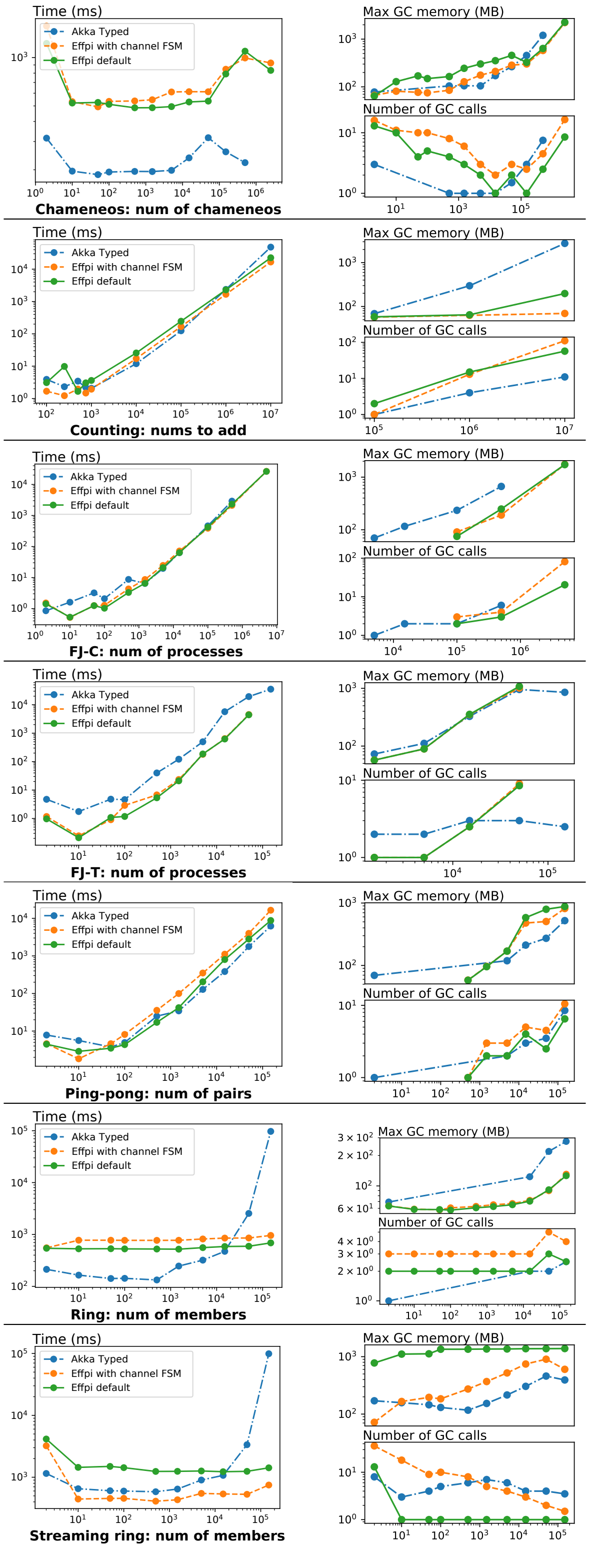

Figure 8. `Effpi`: mean execution time vs. size (left column, 10 runs, low is better), and memory vs. size (right). Some plots end early (e.g., chameneos+Akka) due to out-of-memory crashes; memory use is plotted when GC runs at least once. *(4×Intel i7@3.6GHz, Dotty 0.9.0-RC1, Scala 2.12.7, Akka 2.5.17, 4GB max heap)*

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 ~2× slower); the positive exceptions are fork-join throughput (`Effpi` is ~2× faster), and the ring variants (Akka has exponential slowdown).

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

	states	deadlock-free	ev-usage	forwarding	non-usage	reactive	responsive
Pay & audit + 8 clients	3328	true (0.05 ± 1.38%)	true (0.11 ± 0.92%)	false (6.26 ± 4.16%)	false (0.02 ± 2.66%)	true (1.01 ± 3.95%)	true (15.40 ± 6.57%)
Pay & audit + 10 clients	13312	true (0.06 ± 1.65%)	true (0.19 ± 1.07%)	false (21.90 ± 11.19%)	false (0.02 ± 5.55%)	true (0.96 ± 13.22%)	true (73.37 ± 8.28%)
Pay & audit + 12 clients	53248	true (0.07 ± 1.17%)	true (0.23 ± 1.05%)	false (98.72 ± 12.28%)	false (0.02 ± 2.78%)	true (0.99 ± 2.89%)	true (345.22 ± 8.72%)
Dining philos. (4, deadlock)	4096	false (0.16 ± 1.41%)	true (0.02 ± 2.02%)	false (1.04 ± 9.84%)	false (0.02 ± 3.55%)	false (2.01 ± 4.79%)	false (1.06 ± 19.65%)
Dining philos. (4, no deadlock)	4096	true (0.16 ± 0.70%)	true (0.02 ± 2.33%)	false (1.19 ± 28.13%)	false (0.02 ± 1.47%)	false (1.91 ± 14.08%)	false (1.07 ± 19.19%)
Dining philos. (5, deadlock)	32768	false (0.54 ± 0.80%)	true (0.03 ± 2.46%)	false (4.58 ± 10.54%)	false (0.02 ± 3.55%)	false (5.10 ± 5.78%)	false (3.05 ± 5.11%)
Dining philos. (5, no deadlock)	32768	true (0.55 ± 1.85%)	true (0.03 ± 1.58%)	false (3.05 ± 4.85%)	false (0.02 ± 3.04%)	false (4.21 ± 8.29%)	false (3.01 ± 1.19%)
Dining philos. (6, deadlock)	262144	false (2.35 ± 0.51%)	true (0.03 ± 0.87%)	false (13.61 ± 14.39%)	false (0.03 ± 4.22%)	false (16.58 ± 8.22%)	false (10.72 ± 3.88%)
Dining philos. (6, no deadlock)	262144	true (2.37 ± 0.61%)	true (0.03 ± 2.93%)	false (9.20 ± 5.63%)	false (0.03 ± 3.76%)	false (17.28 ± 6.11%)	false (6.36 ± 6.25%)
Ping-pong (6 pairs)	4096	true (0.05 ± 1.68%)	true (0.01 ± 3.92%)	false (0.95 ± 14.43%)	false (0.01 ± 16.42%)	false (0.98 ± 6.02%)	false (0.98 ± 5.34%)
Ping-pong (6 pairs, responsive)	46656	true (0.26 ± 2.65%)	true (0.02 ± 1.70%)	false (1.05 ± 13.51%)	false (0.02 ± 1.39%)	false (1.00 ± 5.47%)	true (1.98 ± 5.09%)
Ping-pong (8 pairs)	65536	true (0.23 ± 0.82%)	true (0.01 ± 3.07%)	false (2.00 ± 1.25%)	false (0.01 ± 3.27%)	false (2.01 ± 2.48%)	false (1.53 ± 30.27%)
Ping-pong (8 pairs, responsive)	1679616	true (1.60 ± 1.90%)	true (0.03 ± 2.43%)	false (6.89 ± 3.14%)	false (0.03 ± 5.62%)	false (4.58 ± 9.96%)	true (9.39 ± 6.48%)
Ping-pong (10 pairs)	1048576	true (2.40 ± 1.63%)	true (0.02 ± 2.35%)	false (8.63 ± 13.49%)	false (0.01 ± 1.69%)	false (9.53 ± 10.27%)	false (1.99 ± 2.69%)
Ping-pong (10 pairs, responsive)	$>2\times10^6$	true (8.74 ± 10.83%)	true (0.04 ± 2.66%)	false (17.00 ± 1.62%)	false (0.03 ± 1.39%)	false (23.49 ± 4.76%)	true (50.97 ± 5.80%)
Ring (10 elements)	2048	true (0.01 ± 3.58%)	true (0.01 ± 3.82%)	true (11.34 ± 1.48%)	false (0.01 ± 2.44%)	true (7.81 ± 0.35%)	false (1.00 ± 1.10%)
Ring (15 elements)	65536	true (0.02 ± 1.57%)	true (0.02 ± 1.56%)	true (562.48 ± 4.72%)	false (0.01 ± 1.79%)	true (407.47 ± 7.13%)	false (108.61 ± 3.10%)
Ring (10 elements, 3 tokens)	4096	true (0.06 ± 3.14%)	true (0.01 ± 1.72%)	true (23.79 ± 9.10%)	false (0.01 ± 4.07%)	true (15.53 ± 0.38%)	false (1.99 ± 8.18%)
Ring (15 elements, 3 tokens)	131072	true (0.39 ± 0.60%)	true (0.01 ± 1.44%)	true (1146.57 ± 2.11%)	false (0.01 ± 2.19%)	true (827.58 ± 1.00%)	false (2.01 ± 7.92%)

Figure 9. Behavioural property verification: outcome (true/false) and average time (seconds ± std. dev.). The number of states is approximated "$>2\times10^6$" when the LTS is too big to fit in memory. *(4×Intel i7 @ 3.60GHz, 16 GB RAM, mCRL2 201808.0, 30 runs)*

Various π-calculus type systems specialise on accurate (dead)lock-freedom analysis, e.g., [36–39, 58]. [13] 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 [13], with decidable verification (by Lemma. 4.7).

Our protocols-as-types are related to session types [11, 26, 27, 69], and their combination with value-dependent and indexed types [10, 14, 75–77]; session types have inspired various implementations [3], also in Scala [65–68]. 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* [17], 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 [12]), and we are not limited to linear/confluent protocols: e.g., $T = \mathbf{p}\Big[\mathbf{p}\Big[\mathbf{o}\big[\underline{x}, \underline{y}, T\big], \mathbf{o}\big[\underline{x}, \underline{z}, T'\big]\Big], \mathbf{i}\big[\underline{x}, \Pi(\underline{z'}{:}\mathrm{c}^{\mathrm{io}}[\mathrm{int}])U\big]\Big]$ types parallel processes with a race on channel $\underline{x}$; we can verify such processes, capturing that either $\underline{y}$ or $\underline{z}$ may replace $\underline{z'}$ in the U-typed continuation. This covers locking/mutex protocols, allowing, e.g., to implement and verify Dijkstra's dining philosopher 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., [40, 71, 74]. 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 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/mutex, mentioned above); but beyond this, a finer type-level representation of state may make model checking undecidable [19], thus requiring 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 $\lambda^{\pi}_{\leqslant}$ 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 [35]; 2. support infinite-state systems, trying tools like Bfc [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 `Effpi` 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.

References

[1] Umut A. Acar, Arthur Chargueraud, and Mike Rainey. 2013. Scheduling Parallel Programs by Work Stealing with Private Deques. In *PPoPP*. https://doi.org/10.1145/2442516.2442538

[2] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. https://doi.org/10.1007/978-3-319-30936-1_14

[3] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2017. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages* 3(2-3) (2017). https://doi.org/10.1561/2500000031

[4] Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 37 (2017). https://doi.org/10.1145/3110281

[5] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (1999), 29. https://doi.org/10.1145/324133.324234

[6] Julian Bradfield and Colin Stirling. 2007. Modal μ-calculi. In *Handbook of Modal Logic*. Elsevier. https://doi.org/10.1016/S1570-2464(07)80015-2

[7] Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. 2009. On the expressive power of recursion, replication and iteration in process calculi. *Mathematical Structures in Computer Science* 19, 6 (2009), 1191–1222. https://doi.org/10.1017/S096012950999017X

[8] L. Cardelli, S. Martini, J.C. Mitchell, and A. Scedrov. 1994. An Extension of System F with Subtyping. *Information and Computation* 109, 1 (1994). https://doi.org/10.1006/inco.1994.1013

[9] Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *Comput. Surveys* 17, 4 (1985), 53. https://doi.org/10.1145/6041.6042

[10] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed Programming Using Role-parametric Session Types in Go: Statically-typed Endpoint APIs for Dynamically-instantiated Communication Structures. *Proc. ACM Program. Lang.* 3, POPL, Article 29 (2019). https://doi.org/10.1145/3290342

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

[12] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2015. Global Progress for Dynamically Interleaved Multiparty Sessions. *MSCS* 760 (2015). https://doi.org/10.1017/S0960129514000188

[13] Ugo de'Liguoro and Luca Padovani. 2018. Mailbox Types for Unordered Interactions. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2018.15

[14] Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. 2012. Parameterised Multiparty Session Types. *Logical Methods in Computer Science* 8, 4 (2012). https://doi.org/10.2168/LMCS-8(4:6)2012

[15] Emanuele D'Osualdo, Jonathan Kochems, and C. H. Luke Ong. 2013. Automatic Verification of Erlang-Style Concurrency. In *Static Analysis*. Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-38856-9_24

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

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

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

[19] Javier Esparza. 1994. On the decidability of model checking for several μ-calculi and Petri nets. In *Trees in Algebra and Programming — CAAP*. https://doi.org/10.1007/BFb0017477

[20] Javier Esparza. 1997. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica* 34, 2 (1997). https://doi.org/10.1007/s002360050074

[21] Jürgen Giesl, Matthias Raffelsieper, Peter Schneider-Kamp, Stephan Swiderski, and René Thiemann. 2011. Automated Termination Proofs for Haskell by Term Rewriting. *TOPLAS* 33, 2, Article 7 (2011). https://doi.org/10.1145/1890028.1890030

[22] Ursula Goltz. 1990. CCS and Petri nets. In *Semantics of Systems of Concurrent Processes*. https://doi.org/10.1007/3-540-53479-2_14

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

[24] Matthew Hennessy, Julian Rathke, and Nobuko Yoshida. 2005. safeDpi: a language for controlling mobile code. *Acta Informatica* 42, 4-5 (2005). https://doi.org/10.1007/s00236-005-0178-y

[25] Carl Hewitt, Peter Bishop, Irene Greif, Brian Smith, Todd Matson, and Richard Steiger. 1973. Actor Induction and Meta-evaluation. In *POPL*. https://doi.org/10.1145/512927.512942

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

[27] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *POPL*. https://doi.org/10.1145/1328438.1328472 Journal version in [28].

[28] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1, Article 9 (2016). https://doi.org/10.1145/2827695

[29] Atsushi Igarashi and Naoki Kobayashi. 2001. A Generic Type System for the Pi-calculus. In *POPL*. https://doi.org/10.1145/360204.360215

[30] Atsushi Igarashi and Naoki Kobayashi. 2004. A generic type system for the π-calculus. *TCS* 311, 1 (2004). https://doi.org/10.1016/S0304-3975(03)00325-6

[31] Shams M. Imam and Vivek Sarkar. 2014. Savina — An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries *(AGERE!)*. https://doi.org/10.1145/2687357.2687368

[32] Alan Jeffrey. 2001. A Symbolic Labelled Transition System for Coinductive Subtyping of $F_{\mu<}$ Types. In *LICS*. https://doi.org/10.1109/LICS.2001.932508

[33] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. 2015. Bfc - A Widening Approach to Multi-Threaded Program Verification. http://www.cprover.org/bfc/.

[34] Claude Kaiser and Jean-Francois Pradat-Peyre. 2003. Chameneos, a concurrency game for Java, Ada and others. In *ACS/IEEE Int. Conf. on Computer Systems and Applications. Book of abstracts*. https://doi.org/10.1109/AICCSA.2003.1227495

[35] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. 2015. LTSmin: High-Performance Language-Independent Model Checking. In *TACAS*. https://doi.org/10.1007/978-3-662-46681-0_61

[36] Naoki Kobayashi. 1998. A Partially Deadlock-Free Typed Process Calculus. *TOPLAS* 20, 2 (1998). https://doi.org/10.1145/276393.278524

[37] Naoki Kobayashi. 2006. A New Type System for Deadlock-Free Processes. In *CONCUR*. https://doi.org/10.1007/11817949_16

[38] Naoki Kobayashi and Cosimo Laneve. 2017. Deadlock analysis of unbounded process networks. *Information and Computation* 252 (2017). https://doi.org/10.1016/j.ic.2016.03.004

[39] Naoki Kobayashi and Davide Sangiorgi. 2010. A hybrid type system for lock-freedom of mobile processes. *TOPLAS* 32, 5 (2010). https://doi.org/10.1145/1745312.1745313

[40] Morten Krogh-Jespersen, Amin Timany, and Lars Birkedal Marit Edna Ohlenbusch. 2018. Aneris: A Logic for Node-Local, Modular Reasoning of Distributed Systems. https://iris-project.org/pdfs/2019-aneris-submission.pdf. Unpublished draft.

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

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

[43] Leslie Lamport. 1977. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering* SE-3, 2 (March 1977). https://doi.org/10.1109/TSE.1977.229904

[44] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2017. Fencing off go: liveness and safety for channel-based programming. In *POPL*. https://doi.org/10.1145/3093333.3009847

[45] Julien 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

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

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

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

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

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

[51] Barbara H. Liskov and Jeannette M. Wing. 1994. A Behavioral Notion of Subtyping. *TOPLAS* 16, 6 (1994). https://doi.org/10.1145/197320.197383

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

[53] Robin Milner. 1989. *Communication and Concurrency*. Prentice-Hall, Inc.

[54] Robin Milner. 1999. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press.

[55] Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, Parts I and II. *Information and Computation* 100, 1 (1992). https://doi.org/10.1016/0890-5401(92)90008-4

[56] Hanne Riis Nielson and Flemming Nielson. 1994. Higher-order Concurrent Programs with Finite Communication Topology (Extended Abstract). In *POPL*. https://doi.org/10.1145/174675.174538

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

[58] Luca Padovani. 2014. Deadlock and lock freedom in the linear π-calculus. In *CSL-LICS*. https://doi.org/10.1145/2603088.2603116

[59] Dmitry Petrashko, Ondřej Lhoták, and Martin Odersky. 2017. Miniphases: Compilation Using Modular and Efficient Tree Transformations. In *PLDI*. https://doi.org/10.1145/3062341.3062346

[60] Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.

[61] Benjamin C. Pierce and Davide Sangiorgi. 1996. Typing and Subtyping for Mobile Processes. *Mathematical Structures in Computer Science* 6, 5 (1996).

[62] Sriram K. Rajamani and Jakob Rehof. 2001. A Behavioral Module System for the Pi-Calculus. In *SAS*. https://doi.org/10.1007/3-540-47764-0_22

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

[64] Alceste Scalas, Elias Benussi, and Nobuko Yoshida. 2019. Effpi website. https://alcestes.github.io/effpi.

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

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

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

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

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

[70] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Verifying message-passing programs with dependent behavioural types. https://www.doc.ic.ac.uk/research/technicalreports/2019/#1 DoC Technical report 2019/1.

[71] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and proving with distributed protocols. *PACMPL* 2, POPL (2018). https://doi.org/10.1145/3158116

[72] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is Reasonable Coq. In *CPP*. https://doi.org/10.1145/3167092

[73] Colin Stirling. 2001. *Modal and Temporal Properties of Processes*. Springer-Verlag New York, Inc., New York, NY, USA.

[74] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018. Modularity for decidability of deductive verification with applications to distributed systems. In *PLDI*. https://doi.org/10.1145/3192366.3192414

[75] Bernardo. Toninho, Luís. Caires, and Frank Pfenning. 2011. Dependent session types via intuitionistic linear type theory. In *PPDP*. https://doi.org/10.1145/2003476.2003499

[76] Bernardo Toninho and Nobuko Yoshida. 2017. Certifying data in multiparty session types. *Journal of Logical and Algebraic Methods in Programming* 90, C (2017). https://doi.org/j.jlamp.2016.11.005

[77] Bernardo Toninho and Nobuko Yoshida. 2018. Depending on Session-Typed Processes. In *FoSSaCS*. https://doi.org/10.1007/978-3-319-89366-2_7

[78] Nobuko Yoshida. 2004. Channel dependent types for higher-order mobile processes. In *POPL*. https://doi.org/10.1145/964001.964014

[79] Nobuko Yoshida and Matthew Hennessy. 2002. Assigning Types to Processes. *Information and Computation* 174, 2 (2002). https://doi.org/10.1006/inco.2002.3113