Verifying Message-Passing Programs with Dependent Behavioural Types

Technical report

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

CCS Concepts • Theory of computation → Process calculi; Type structures; Verification by model checking • Software and its engineering → Concurrent programming languages.

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

Short version of this report (accepted at PLDI 2019)
https://doi.org/10.1145/3314221.3322484

1 Introduction

Consider this specification for a payment service with auditing (from a use case for the Akka Typed toolkit [42, 50]):

```
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       }
10   }
```

Figure 1. Implementation of the payment service specification ($\S$1). Although similar to Akka Typed [50], it is written in Dotty and Effpi, described in $\S$5; "\texttt{\text{>>}}" (l.7) means "and then."

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

This can be implemented using various languages and tools for concurrent and distributed programming. E.g., using Scala and Akka Typed [50], a developer can write a solution similar to Fig. 1: payment is an actor, receiving messages of type Pay (line 1); aud is the actor reference of the auditor, used to send messages of type Audit; whenever a pay message is received (line 3), 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 \texttt{send(aud, "Hi")} causes a compilation error. However, the payment service specification is not enforced: e.g., if the developer forgets to write line 7, the code still compiles, but accepted payments are not audited. This is a typical concurrency bug: a missing or out-of-order communication can cause protocol violations, deadlocks, or livelocks. Such bugs are often spotted late, during software testing or maintenance — when they are more difficult to find and fix, and harmful: e.g., what if unaudited payments violate fiscal rules?

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

Step 1: enforcing protocols at compile-time We develop Effpi [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:

\[
\text{In}[\text{Pay}, (p: \text{Pay})] \Rightarrow \begin{cases} \text{Out}[p.\text{replyTo}.\text{type}, \text{Rejected}] & \text{| (Out[\text{Audit}.\text{type}, \text{Audit}[p.\text{type}]] \Rightarrow:} \\
\text{Out}[p.\text{replyTo}.\text{type}, \text{Accepted}] \end{cases}
\]

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

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

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

The issue is that types in \(\lambda^{π}_{≤}\) 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 \(\lambda^{π}_{≤}\) calculus, at the basis of Effpi. §3 presents type system of \(\lambda^{π}_{≤}\). §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 \(\lambda^{π}_{≤}\) 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 \(\lambda^{π}_{≤}\)-Calculus

The theoretical basis of our work is a \(\lambda\)-calculus extended with channels, input/output, and parallel composition, called \(\lambda^{π}_{≤}\). The "π" denotes both: (1) its use of dependent function types, that, together with subtyping \(\ll\), are cornerstones of its typing system (§3); and (2) its connection with the \(π\)-calculus [54, 55, 63]. Indeed, \(\lambda^{π}_{≤}\) is a monadic-style encoding of the higher-order \(π\)-calculus(cf. Ex. 2.6): continuations are \(λ\)-terms, and this will be helpful for typing (§3) and implementation (§5).

Definition 2.1. The syntax of \(\lambda^{π}_{≤}\) is in Fig. 2. Elements of \(C\) are run-time syntax. Free/bound variables \(fv(t)/bv(t)\) are defined as usual. We adopt the Barendregt convention: bound
variables are syntactically distinct from each other, and from free variables. We write $\lambda_{-}t$ for $\lambda x.t$, when $x \notin \text{fv}(t)$.

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

Example 2.2. A ping-pong system in $\lambda_{<}$ is written as:

\[
\begin{align*}
\text{let} & \quad \text{pinger} = \lambda \text{self}. \text{pong}(\cdot) \quad \text{let} \quad \text{ponger} = \lambda \text{self}. \quad \\
\text{send}(\text{pong}, \text{self}, \lambda_{-} \cdot) & \quad \text{recv}(\text{self}, \lambda \text{replyTo}. \quad \\
\text{recv}(\text{self}), \lambda \text{replyTo}. & \quad \text{recv}(\text{replyTo}, "\text{Hi}!", \lambda_{-}(\quad \text{end }))})) \\
\text{let} & \quad \text{sys} = \lambda \text{y}. \lambda \text{z}. (\quad \text{let} \quad \text{main} = \lambda_{-} \quad \text{let} \quad \text{y} = \text{chan}() \quad \text{in} \quad \text{let} \quad \text{z} = \text{chan}() \quad \text{in} \quad \text{sys} \quad \text{y} \quad \text{z} \\
\end{align*}
\]

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

Note that in $\text{pinger}$ and $\text{ponger}$, and also in Ex. 2.4 below, the last argument of $\text{send} / \text{recv}$ is always an abstract process term: this is expected by the semantics (Def. 2.5), and enforced via typing (§3).

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

Example 2.4. This example instantiates and interconnects three parallel processes (we shorten "let..." by omitting "in"):

\[
\begin{align*}
\text{let} & \quad \text{sender} = \lambda \text{y}. \text{send}(\text{y}, "\text{Hello}"), \lambda_{-} \text{end}) \\
\text{let} & \quad \text{receiver} = \lambda \text{z}. \text{recv}(\text{z}, \lambda_{-} \cdot \text{end}) \\
\text{let} & \quad \text{fwd} = \lambda \text{i}. \lambda \text{a}. \text{recv}(\text{i}, \lambda \text{a}. \lambda \text{z}. \text{send}(\text{a}, \text{z}, \lambda_{-} \text{fwd} \text{i} \text{o})) \\
\text{let} & \quad \text{sys} = \lambda \text{y} . \lambda \text{z}. (\quad \text{let} \quad \text{y} = \text{chan}() \quad \text{in} \quad \text{let} \quad \text{z} = \text{chan}() \quad \text{in} \quad \text{sys} \quad \text{y} \quad \text{z} \\
\end{align*}
\]

- $\text{sender}$ is an abstract process that takes a channel $\text{y}$, uses it to send "Hello", and ends;
- $\text{receiver}$ takes a channel $\text{z}$, uses it to receive a value $x'$, and terminates;
- $\text{fwd}$ takes two channels $\text{i}$ and $\text{o}$, recursively reads a message from $x'$ from $\text{i}$, and writes it in $\text{o}$;
- $\text{sys}$ takes channels $\text{y}'$, $\text{z}'$, and uses them to instantiate $\text{sender}$, $\text{fwd}$ and $\text{receiver}$ in parallel;
- in the last line, invoking $\text{main}()$ instantiates sys with $\text{y}$ and $\text{z}$, that contain channel instances.

Definition 2.5 (Semantics of $\lambda_{<}$. Evaluation contexts $E$ and reduction $\Rightarrow$ are illustrated in Fig. 3, where congruence $\equiv$ is defined as: $t_1 \equiv t_2 \equiv t_1$ and $\text{end} \equiv \text{end}$, plus $\alpha$-conversion. We write $\Rightarrow$ for the reflexive and transitive closure of $\Rightarrow$. We say "t has an error" iff $t \equiv \text{E[err]}$ (for some $E$). We say "t is safe" iff $\forall t' : t \Rightarrow t'$ implies $t'$ has no error.

Def. 2.5 is a standard call-by-value semantics, with two rules for concurrency. [R-chan] says that $\text{chan}(l)$ returns a fresh channel instance; [R-Comms] says that the parallel composition $\text{send}(a, u, v_1) \text{recv}(a, v_2)$, where both sides operate on a same channel instance $a$, transfers the value $u$ on the receiver side, yielding $v_1 \equiv v_2 u$: hence, if $v_1$ and $v_2$ are function values, the process keeps running by applying $v_1 ()$ and $v_2 u$ — i.e. the sent value is substituted inside $v_2$. The error rules say how terms can "go wrong:" they include usual type mismatches (e.g., it is an error to apply a non-function value $u$ to any $v$), and three rules for concurrency: it is an error to receive/send data using a value $u$ that is not a channel, and it is an error to put a value in a parallel composition (i.e., only processes from $\mathbb{P}$ in Fig. 2 are safely composed by $\parallel$).

Example 2.6 (Higher-order $\pi$-calculus [79]). HO$\pi$ is easily encoded in $\lambda_{<}$: we render replication $\lambda u.(y).P$ by spawning a replica $z_u$ at every context. The rest is straightforward.

\[
\begin{align*}
\{x\} & = x \quad \{a\} = a \quad \{P_1 \parallel P_2\} = \{P_1\} \parallel \{P_2\} \quad \{\lambda x.P\} = \lambda x.\{P\} \\
\{P_1 \parallel P_2\} & = \{P_1\} \parallel \{P_2\} \quad \{\text{recv}(x)\} = \{x\} = \text{chan}() \quad \{P\} \\
\{u!(y).P\} & = \text{send}(\{u\}, \{c\}, \lambda \text{y}.\{P\}) \\
\{u!(y).P\} & = \text{recv}(\{u\}, \{c\}, \{P\}) \\
\{u!(y).P\} & = \{z_u\} = \lambda \text{recv}(\{u\}, \{y\}, \{P\} \parallel z_u) \} \text{ in } z_u()
\end{align*}
\]

3 Type System

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

Notably, points (1) and (3) establish a similarity between $\lambda_{<}$ 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,

1Except that we do not include polymorphism: it is orthogonal to our aims.
akin to those of $F_\preceq$, 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:

$$
\begin{align*}
\text{bool} & \mid () \mid T \mid T \lor U \mid \Pi(x:U)T \mid \mu_x.T \mid X \\
\text{proc} & \mid \text{nil} \mid \text{o}[S, T, U] \mid \text{i}[S, T] \mid p[T, U]
\end{align*}
$$

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

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

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

$$
\begin{align*}
T \lor U & \equiv U \lor T & S \lor (T \lor U) & \equiv (S \lor T) \lor U & \mu T.\equiv T[\mu T/x] \\
\end{align*}
$$

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

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

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

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

A typing environment $\Gamma$ maps variables (from $X$ in Def. 2.1) to types; the order of the entries of $\Gamma$ is immaterial. All judgements in Fig. 4 are inductive, except subtyping, that is coinductive (hence the double inference lines). Crucially, in Fig. 4 we have two valid type judgements, for two kinds of types: $\Gamma \vdash \Pi.x:T$ type and $\Gamma \vdash \Pi.x:T^-\text{-type}$. The former is standard (except for rule $[\tau\Leftarrow]$, for valid channel types); the latter distinguishes process types. Note that subtyping only relates types of the same kind. Importantly, a typing environment $\Gamma$ can map a variable to a type (rule $[\text{vf}]$), but not to a $\pi$-type; this also means that function arguments cannot be $\pi$-typed. Still, in a function type $\Pi(x:T)U$, the return type $U$ can be a $\pi$-type (rule $[\text{vf}\pi]$): i.e., it is possible to define abstract process types (cf. Ex. 3.4 and 3.5 later). Rules $[\tau\Leftarrow]$ and $[\pi\Leftarrow]$ are based on $[32, \S2]$, and require recursive types to be contractive (e.g., $\mu T_1.\mu T_2.\ldots.\mu T_n.T_1 \lor U$ is not a type; clause $\tau \notin \text{fv}(T)$ means that variable $\tau$ is not bound in negative position in $T$, as in $F_\preceq$ (details: §4A). Recursion is handled by the $[\tau\text{-let}]$: in let $let x = t$ in $t'$, term $t$ can refer to $x$. Rule $[\ll\Leftarrow]$ based on $[9]$, ensures decidability of subtyping $[32, \S1]$; 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 $[\tau\Leftarrow]$, the type $x$ is valid in $\Gamma$; and indeed, by rule $[\tau\Leftarrow]$, we can infer $\Gamma \vdash x : \tau$, i.e., the term $x$ has type $\tau$. Intuitively, this means that $\tau$ is the 'most precise' type for term $x$; this is formally supported by the subtyping rule $[\tau\Leftarrow]$, that says: as $\Gamma$ maps term $x$ to $T$, type $\tau$ is smaller than $T$. To retrieve from $\Gamma$ the information that term $x$ has (also) type $T$, we use subtyping and subsumption (rule $[\ll\Leftarrow]$), as shown here. Since
Verifying Message-Passing Programs with Dependent Behavioural Types (tech. report)

Figure 4. Judgements of the $\lambda^\Pi_\leq$ type system (Def. 3.2). The main concurrency-related rules are highlighted.

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

By $[\llbracket\cdot\rrbracket]$, both sub-terms of $t_1\parallel t_2$ are $\pi$-typed.

By $[r\text{-}\text{send}]$, send$(t_1,t_2,t_3)$ has type $\text{o}[S,T,U]$, under the validity constraints of rule $[\llbracket\cdot\rrbracket]$. 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 $\pi$-type $U'$): i.e., $t_3$ is a process thunk, run by applying $t_1$.

By $[r\text{-}\text{recv}]$, recv$(t_1,t_2)$ has type $\llbracket S,T \rrbracket$, which is well-formed under rule $[\pi\leq]$. Hence, the sub-term $t_1$ must have a channel type with input $U$, while $t_2$ must be an abstract process of type $T = \Pi(x:U')T'$, with $T'$ $\pi$-type. Crucially, by rule $[\pi\leq]$, we have $\Gamma U \leq U'$; 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. 2.4, we have the type assignments:
sender : $T_{end} = \Pi(y: c'[str]) \ o \ y, str, \Pi(\Pi(nil))$
receiver : $T_{rev} = \Pi(z: c'[str]) \ i \ [z, \Pi(x': str)\nil]$
forward : $T_{forward} = \Pi(i: c'[str]) \ (\Pi(x: c'[str]) \ o \ i, \Pi(t))$

sys : $T_{sys} = \Pi(y': c'[str]) \ (\Pi(x': c'[str]) \ p \ [p(T_{end} \ y'), (T_{forward} \ y'), (T_{rev} \ z']])$

Hence, by expanding the type instantiations in $T_{sys}$, we get:

$$T_{sys} = \Pi(y': c'[str]) \ (\Pi(z: c'[str]) \ o \ y', str, \Pi(nil)), \ p \ [p(T_{end} \ y'), (T_{forward} \ y'), (T_{rev} \ z')]$$

where we can observe how $y'$ and $z'$ are used, and how $x$ is received from $y'$, and sent on $z'$.

**Example 3.4.** In Ex. 2.2, we have the type assignments:

$pinger : T_{ping} = \Pi(self: c'[str]) \ o \ \Pi(pong: c'[str])$

$ponger : T_{pong} = \Pi(self: c'[str]) \ o \ \Pi(ping: c'[str])$

$sys : T_{pp} = \Pi(x: c'[str]) \ (\Pi(z: c'[str]) \ o \ T_{ping} \ y \ z, T_{pong} \ z) = \Pi(x: c'[str]) \ o \ T_{ pong} \ y \ z, T_{ ping} \ y \ z$

Notice how $T_{pp}$ captures the ping/pong composition of sys, preserving its channel topology: the type-level applications $T_{ping} \ y \ z$ and $T_{pong} \ z$ (yielded by rule $\{t-app\}$, Fig. 4) substitute $y$ and $z$ 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 3.5 (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 $\Pi x$, the intended behaviour of custom code can be formalised by a type like $T_m$ below: it describes an abstract process, taking two input channels $i_1 / i_2$, and an output channel $o$: it must use $i_1 / i_2$ to input integers $x / y$, and then it must send one of them along $o$ recursively.

$$T_m = \Pi(i_1: c'[int]) \ (\Pi(i_2: c'[int]) \ o \ \Pi(x: c'[int]) \ o \ \Pi(y: c'[int]) \ o \ \Pi(o: c'[int]) \ o \ (o(x \ y), \Pi(t))$$

By inspecting $T_m$, we infer that, e.g., $T_m$-typed terms cannot be forkbombs; also, "$x \ y$ does not allow to send on $out$ a value not coming from $i_1 / i_2$ (we will formalise these intuitions in Ex. 4.12). The terms below implement $T_m$: $m_1$ always sends $x$ received from $i_1$, then recursively calls itself, swapping $i_1 / i_2$; $m_2$ sends the maximum between $x$ and $y$.

let $m_1 = \lambda i_1. \lambda i_2. \lambda o. \ (\Pi(x: c'[int]) \ (\Pi(y: c'[int]) \ o \ (o(x \ y), \Pi(t))))$
let $m_2 = \lambda i_1. \lambda i_2. \lambda o. \ (\Pi(x: c'[int]) \ (\Pi(y: c'[int]) \ o \ (o(x \ y), \Pi(t))))$

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 $z_1 / z_2$:

let $srv = \lambda cm. \lambda out. \ (let \ z_1 = \Pi(cm) = \Pi(out) \ o \ \Pi(prod) \ z_1 \ o \ \Pi(prod) \ z_2 \ )$

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

$$\emptyset \ + \ T_{srv} = \Pi(cm: c'[str]) \ o \ \Pi(out: c'[int]) \ o \ prod_1 \ o \ prod_2 \ o $$

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

$$send(x, t, \_... end) \ + \ o \ send \ (\ client \ sends \ t \ to \ server, \ via \ a)$$

is typable in $\Gamma$ only if $\Gamma + x : c'[str]$ implies $\Gamma + t : T_m$. We can replace $proces$ with a more precise type. If $U_l / U_2$ are types of $prod_1 / prod_2$, the $recv(...)$ sub-term of $srv$ has type:

$$T_{srv} = \Pi(cm: c'[str]) \ o \ (\Pi(out: c'[int]) \ o \ send(x, t, \_... end) \ + \ send(y, "Hi", \_... end))$$

Two $T_p$-typed terms are:

$$t_1 = \lambda \lambda \lambda x. \lambda y. \recv(x, y, \_... end) \ (\ just \ forwards)$$
$$t_2 = \lambda \lambda \recv(x, y, \_... end) \ (\ just \ replies)$$

Their types can be narrower, hence more precise:

$$t_1 : T_p = \Pi(cm: c'[str]) \ o \ (\Pi(out: c'[int]) \ o \ send(x, y, \_... end)) \ + \ send(y, "Hi", \_... end))$$

$$t_2 : T_p = \Pi(cm: c'[str]) \ o \ (\Pi(out: c'[int]) \ o \ send(x, y, \_... end)) \ + \ send(y, "Hi", \_... end))$$

If a term $t_{12}$ interconnects instances of $t_1$ and $t_2$, then its type $T_{12}$ can capture the interconnection. E.g., if we let $t_{12} = t_1 x z \ o \ t_2 z z$, then we have:

$$\Pi(x: c'[str]) \ o \ z: c'[str] \ + \ t_{12} : \Pi(T_{12} x z : T_{12} z z)$$

**Subtyping, subsumption, and private channels** The subtyping rules in Fig. 4 are standard (based on $F_c$; [8, 32]) except the highlighted ones. By rule $\{c < \}_c$, 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 $\{c < \text{proc}\}$ says that $\text{proc}$ is the top
type for $\pi$-types. Rules $[\ell \lessdot o]/[\ell \lessdot i]/[\ell \lessdot p]$ say that types for input/output/parallel processes are covariant in all parameters.

As usual, supertyping / subsumption (rule $\tau \lessdot \kappa$) 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. 3.7: via supertyping, we do not precisely track how private (i.e., bound) channels are used. This information loss is key to type Turing-powerful $\lambda_2^{\tau}$ terms with a non-Turing-complete type language, for the results in §4.

Example 3.7 (Subtyping, binding, and precision loss). Let:

- $t_1 = \text{send}(x, 42, \lambda_{\_\.end}) \parallel \text{recv}(x, \lambda_{\_\.end})$
- $t_2 = \{\text{let } z = \text{chan}(\text{in } \text{send}(z, 42, \lambda_{\_\.end}) \parallel \text{recv}(x, \lambda_{\_\.end})\}$

$T_1 = \lfloor \lambda x, \Pi(\text{nil}) \mid i \lfloor x, (\Pi(\text{nil}) \mid i \lfloor x, \Pi(y\text{int})\lfloor\rfloor\rfloor\rfloor\rfloor$

$T_2 = \lfloor \lambda o \lfloor x, \Pi(\text{int})\rfloor \mid i \lfloor x, (\Pi(y\text{int})\lfloor\rfloor\rfloor\rfloor\rfloor$

Letting $\Gamma = x e^{\Pi}\lfloor \text{int}\rfloor$, we have $\Gamma \vdash x \lessdot e^{\Pi}\lfloor \text{int}\rfloor$ and $\Gamma \vdash T_1 \lessdot T_2$. For $t_1$, we have both $t_1 : T_1$ and $\Gamma \vdash t_1 : T_2$ (by $\tau \lessdot \kappa$): 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 $e^{\Pi}\lfloor \text{int}\rfloor$ 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 $\tau \vdash \text{let}\_{\cdot\cdot}$ (Fig. 4): since $z$ is bound by $\text{let}$,..., its occurrence in $\text{send}(...)$ is typed by a supertype of $z$ that is suitable for both $z$ and $\text{chan}(\cdot)$ in this case, $e^{\Pi}\lfloor \text{int}\rfloor$. Specifically:

$\Gamma \vdash e^{\Pi}\lfloor \text{int}\rfloor \lessdot e^{\Pi}\lfloor \text{int}\rfloor\
\Gamma \vdash z e^{\Pi}\lfloor \text{int}\rfloor + \text{chan}(\cdot) : e^{\Pi}\lfloor \text{int}\rfloor\
\Gamma \vdash e^{\Pi}\lfloor \text{int}\rfloor + \text{send}(z, 42, \lambda_{\_\.end}) \parallel \text{recv}(x, \lambda_{\_\.end})$ [let]

$\Gamma \vdash \text{let } z = \text{chan}(\text{in } \text{send}(z, 42, \lambda_{\_\.end}) \parallel \text{recv}(x, \lambda_{\_\.end})\lfloor\rfloor\rfloor\rfloor\rfloor$

Typing guarantees that well-typed terms never go wrong.

Theorem 3.8 (Type safety). If $\Gamma \vdash t : T$, then $t$ is safe.

Thm. 3.8 follows by: $\Gamma \vdash t : T$ and $t \rightarrow t'$ implies $\exists T'$ such that $\Gamma \vdash t' : T'$ i.e., typed terms only reduce to typed terms, without (un)typable err subtrees. This is expected, as we combine System $F_{\leq}$-style typing rules, and typed I/O channels. In §4, we study how $T$ and $T'$ are related, and how they constrain $t$'s behaviour.

4 Type-Level Model Checking

Our typing discipline guarantees conformance between processes and types (Fig. 4), and of absence run-time errors (Thm. 3.8). 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 $\lambda_2^{\tau}$ terms are allowed to interact.

To avoid this issue, we might want to check whether a process $t$ (possibly consisting of multiple parallel sub-processes) satisfies a property $\phi$ in some temporal logic [73]: $\phi$ could be, e.g., a safety property $\Box(\neg \phi')$ ("$\phi'$ is never true while $t$ runs") or a liveness property $\Diamond \phi'$ ("$\phi'$ will eventually satisfy $\phi'\)"). However, this problem is undecidable (unless $\phi$ is trivial), since $\lambda_2^{\tau}$ 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. 3.4), and (2) mimic the behaviour of processes by giving a semantics to types (as we show in this section). This means that we can ensure that a composition of typed processes $t$ has a desired safety/liveness property, by model-checking its type $T$ (that is not Turing-powerful). Moreover, we do not need to know how $t$ is implemented: we only need to know that it has type $T$. We now illustrate the approach, and its preconditions (roughly: for the verification of liveness properties, we need productivity, and use of open variables).

Outline First, we need to surmount a typical obstacle for behavioural type systems. Ex. 3.7 shows that accurate types require open terms in their typing environment — but Def. 2.5 works on closed terms; so, observing how $T_1$ in Ex. 3.7 uses $x$, we sense that $t_1$ should interact via $x$ — but by Def. 2.5, $t_1$ is stuck. To trigger communication, we may bind $x$ in $t_1$ with a channel instance, e.g., $t'_1 = \text{let } x = \text{chan}(\text{in } t_1 \text{ but } t'_1$’s type cannot mention $x$ hence cannot convey which channel(s) $t'_1$ uses. Thus, we develop a type-based analysis in four steps: (1) we define an over-approximating LTS semantics for typed $\lambda_2^{\tau}$ terms with free variables (Def. 4.1); (2) we define an LTS semantics for types (Def. 4.2); (3) we prove subject transition and type fidelity (Thm. 4.4, 4.5); (4) using them, we show how temporal logic judgements on types transfer to processes.

Definition 4.1 (Labelled semantics of open typed terms). When $\Gamma \vdash t : T$ (for any $\Gamma, t, T$), the judgements $\Gamma \vdash t \rightarrow t'$ and $\Gamma \vdash t \vdash t'$ are inductively defined in Fig. 5.

Unlike Def. 2.5, Def. 4.1 lets an open term like $\neg x$ reduce, by non-deterministically instantiating $x$ to $t$ or $\emptyset$; the assumption $\Gamma \vdash \neg x : T$ ensures that $x$ is a boolean. Rule [SR→] inherits "concrete" reductions from Def. 2.5: if $t \rightarrow t'$ is induced by base rule $[\sigma]$, the transition label is $t$. Rules [SR-send]/[SR-rev] send/receive a value/variable $w'$ using a (channel-typed) value/variable $w$. Note that in [SR-rev], $w'$ is any value/variable of type $T$, which is the input type of $x$ (in $\pi$-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-rot] applies a fun to $w$, yields a term $w[j/y]$ of type $T$; it also records $x$ in the transition label. Rule [SR-env] applies a fun to a variable $x$, with the expected substitution. Rule [SR-let] propagates transitions through contexts, unless labels refer to bound variables. Finally, $\Gamma \vdash t \vdash t'$ holds when $t$ reaches $t'$ via
finite sequence of internal moves excluding interaction: i.e., labels \( w(w'), \overrightarrow{w}(w'), \) and \( \tau[R\text{-Comm}] \) are forbidden.

Using Def. 4.1 on \( t_1 \) from Ex. 3.7, we get the transition
\[ \Gamma \vdash t_1 \text{ end} \mid \text{end} \text{ and we observe the use of } x, \text{ as desired.} \]

**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 \( t_2 \) and \( t_1 \) from Ex. 3.7: we want \( t_2 \) to reduce, simulating the term reduction \( \Gamma \vdash t_1 \text{ end} \mid \text{end} \). This suggests that a type like \( \langle x_1, \ldots, x_n \rangle \) should reduce with a communication on \( x \). But consider \( t_2 \) in Ex. 3.7: \( t_2 \) also types \( t_1 \); hence it should also simulate \( t_1 \)'s reduction — i.e., a type like \( \langle x_0[\text{int}], \ldots, x_n[\text{int}] \rangle \) should reduce, too. In general, we want \( \langle x_0[\text{int}], \ldots, x_n[\text{int}] \rangle \) to reduce if \( S \) and \( T \) "might interact", i.e., they could type a same channel/variable: we formalise this idea as \( \Gamma \vdash S \Rightarrow T \) in Def. 4.2.

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

\[ \Gamma \not\vdash S \parallel T \iff \Gamma \not\vdash S \Rightarrow T \]

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

\[ \{ \} \mid o(E, T, U) \mid o(S, E, U) \mid o(S, T, E) \mid i(E, T) \mid i(S, E) \mid p(E, T) \]

Judgements \( \Gamma \vdash T \Rightarrow T' \) and \( \Gamma \vdash T \Rightarrow T'[u/v] \) are in Fig. 6.

By Def. 4.2, \( \Gamma \vdash S \Rightarrow S' \) holds when \( S \) and \( S' \) have a common subtype besides \( \perp \), i.e., they might type a same term in \( \Gamma \), via rule \( \tau[c] \). The simplest case is when either \( S \) or \( S' \) is a variable \( x \); then, the judgement holds only when \( S \) and \( S' \) are channel types, rule \( \{c\} \) amounts to checking whether \( S \) and \( S' \) have a common valid channel subtype (cf. rule \( \{c\} \), Fig. 4); if such a type exists, then communication might occur, via rules \( \text{R-Comm}[\text{SR-Comm}] \). E.g., the types \( S = e[\text{int}] \) and \( S' = e[\text{int}] \) cannot interact. To see why, assume (by contradiction) that \( S \) and \( S' \) have a common subtype that is not \( \perp \); by \( \{c\} \), such a type be of the form \( c[\text{int}]T_0 \), for some \( T_0 \) such that \( \Gamma \vdash T_0 \leq \leq \text{int} \) and \( \Gamma \vdash T \leq \leq \text{real} \); but for \( T_0 \) to exist, we must have \( \Gamma \vdash T \leq \leq \text{int} \) — contradiction. Instead, if we take \( S = e[\text{real}] \) and \( S' = e[\text{int}] \), then \( \Gamma \vdash S \Rightarrow S' \) holds; in fact, by \( \{c\} \), a common subtype for both \( S \) and \( S' \) is \( e[\text{int}] \). The judgement \( \Gamma \vdash T \Rightarrow T' \) says that \( TVU \) can reduce to \( T \) or \( U \), firing label \( \tau[v] \); type contexts \( \mathcal{E} \) allow, e.g., \( S = S_1 \lor S_2 \) to reduce inside \( p[S, U] \), exposing the prefixes needed by other rules; reductions are up-to congruence \( \equiv \), that can swap \( \lor \) branches, and reorganise \( p[\ldots, \ldots] \) as a commutative monoid, with unit \( \text{nil} \). Rule \( \{\tau[\text{o}] \} \) reduces an output type, recording the used channel \( S \) and payload \( T \) in the transition label. Rule \( \{\tau[\text{i}] \} \) is similar for input types, recording the payload \( T' \); but since \( T' \) is not syntactically part of the type, the rule uses \( \Gamma \) to "guess" it, by accepting:

(a) \( T' = T \), where \( T \) is taken from the continuation type \( \Pi(x:T)U \);

(b) \( T' = z \), for any \( z \in X \). In this case, clause \( \Gamma \vdash T' \leq T \) requires type \( z \) to be compatible with the argument type of the continuation; moreover, it implicitly ensures that \( z \in \text{dom}(\Gamma) \).

When the rule fires, \( T' \) is substituted in the continuation type; hence, case (a) gives a (safe) approximation for the continuation, while case (b) faithfully propagates \( z \) through the dependent function type \( \Pi(x:T)U \). Crucially, (a) and (b) imply that rule \( \{\tau[\text{i}] \} \) is finite-branching (unlike rule \( \text{SR-rev} \)) in Def. 4.1). We have two communication rules:

- **[\{\tau[\text{o}] \}]** if \( p(U', U') \), there might be an interaction with a type variable \( x \) as payload. More precisely, the rule fires when we have \( \Gamma \vdash U \xrightarrow{\tau[\text{o}]} U' \) and \( \Gamma \vdash U'' \xrightarrow{\tau[\text{o}]} U''' \), and \( S, S' \) might interact. In this case, the type reduces to \( p(U', U'') \). Note that, by \( \{\tau[\text{o}] \} \), the \( x \) sent by \( U \) is substituted in \( U'' \), hence it can appear in its future transitions. The rule yields a transition label \( r[S, S'] \), recording which channel types were used:

- **[\{\tau[\text{i}] \}]** is similar, but fires if the payload \( T \) is a not a variable. Note that clause \( \Gamma \vdash S \Rightarrow S' \) ensures that \( U'' \) has a \( T' \) transition with \( \Gamma \vdash T' \leq T' \), and the rule fires it. Note that if a type reduces with label \( r[S, S'] \), then it enables either \([\{\tau[\text{o}] \}] \) or \([\{\tau[\text{i}] \}] \), but not both. Finally, \( \Gamma \vdash T \xrightarrow{\tau[v]} T' \)
Verifying Message-Passing Programs with Dependent Behavioural Types (tech. report)

Example 4.3. Take syst from Ex. 2.2, \( \text{TPpp from Ex. 3.4. Let:} \)

\[ \Gamma = y \in \text{sys} \alpha, z \in \text{sys} \alpha \]

\[ T = \text{TPpp } \alpha \gamma z \]

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

\[ \Gamma \vdash t \mathbin{ \vdash_{\ast} } y \mathbin{ \vdash_{\ast} } (\text{recv}(y, \ldots) \implies |\text{send}(y, \text{“Hi!”}, \ldots)\rangle) \mathbin{ \vdash_{\ast} } \end{end} \]

By Def. 4.2, applying rule \( \tau \to \text{iox} \) twice, we get:

\[ \Gamma \vdash T \mathbin{ \vdash_{\ast} } y \mathbin{ \vdash_{\ast} } (\text{recv}(y, \ldots) \implies |\text{send}(y, \text{“Hi!”}, \ldots)\rangle) \mathbin{ \vdash_{\ast} } \end{end} \]

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

Subject transition and type fidelity  With the semantics of Def. 4.1, we prove a result yielding Thm. 3.8 as a corollary.

Theorem 4.4 (Subject transition). Assume \( \Gamma \vdash t : T \). If \( \Gamma \vdash t \mathbin{ \vdash_{=\tau} } t' \) implies \( \Gamma \vdash t' : T ' \). Otherwise, \( \Gamma \vdash T \pi \text{-type}, we have:

1. \( \Gamma \vdash t \mathbin{ \vdash_{=\tau} } t' \) with \( \mathit{t^*}(\alpha) \) (Fig. 5) implies \( \Gamma \vdash t' : T ' \);
2. \( \Gamma \vdash t \mathbin{ \vdash_{=\tau} } t' \) and \( \alpha \in \{x(w), x(w), \tau[x], \tau[\mathit{RComm}]\} \) implies one:

- a. \( \alpha = T ' \) and \( \mathit{proc} \in \mathbb{E} \);
- b. \( \alpha = \mathbb{X}(w) \) and \( \exists S, U, T ' : \Gamma \vdash x : S, w : U, t' : T ' \) and
- c. \( \alpha = \mathbb{X}(w) \) and \( \exists S, U, T ' : \Gamma \vdash x : S, w : U, t' : T ' \) and
- d. \( \alpha = \tau[x] \) and \( \exists S, S', T ' : \Gamma \vdash x : S, x : S', t' : T ' \) and
- e. \( \alpha = \tau[\mathit{RComm}] \) and \( \exists S, S', T ' : \{S, S' \} \not\subseteq \mathbb{X}, \Gamma \vdash t' : T ' \) and
- "End of the list."
**Process verification via type verification** By exploiting the correspondence between process/term reductions in Thm. 4.4 and 4.5, we can transfer (decidable) verification results from types to processes. To this purpose, we analyse the labelled transition systems (LTSs) of types and processes using the linear-time \( \mu \)-calculus [20, §3]. We chose it for two reasons: (1) the open term/term semantics (Def. 4.1/4.2) are over-approximating, and a linear-time logic is a natural tool to ensure that all possible executions ("real" or approximated) satisfy a formula; and (2) linear-time \( \mu \)-calculus is decidable for our types, with minimal restrictions (Lemma 4.7).

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

- **Basic formulas:** \( \phi := Z \mid \neg \phi \mid \phi_1 \land \phi_2 \mid (\alpha)\phi \mid vZ.\phi \)
- **Derived formulas:** \( (A)\phi \mid (\neg A)\phi \mid (\alpha) \phi \mid (\perp)\phi \mid (\mu Z.\phi) \)

In Def. 4.6, \( \phi \) describes accepted sequences of actions; \( \phi \) can be a variable \( Z \), negation, conjunction, prefixing (\( \alpha \)) ("accept a sequence if it starts with \( \alpha \), and then \( \phi \) holds"), or greatest fixed point \( vZ.\phi \). Basic formulas are enough [6, 73] to derive true/false (accept any/no sequence of actions), disjunction, implication, least fixed points \( \mu Z.\phi \); \( (A)\phi \) accepts sequences that start with any \( A \), then \( \phi \) s; dually, \( (\neg A)\phi \) requires \( A \) in \( \mathcal{Act} \setminus A \). We also derive usual temporal formulas \( (\alpha) \phi \mid (\neg A)\phi \mid (\alpha) \phi \mid (\perp)\phi \mid (\mu Z.\phi) \) (\( \phi \) is always true), and \( (\phi) \) (\( \phi \) is eventually true). Given a process \( p \) with LTS of labels \( \mathcal{Act} \), a run of \( p \) is a finite or infinite sequence of labels fired along a complete execution of \( p \); we write \( p \vdash \phi \) if \( \phi \) accepts all runs of \( p \). (Details: §B)

We can decide \( \phi \) on a guarded type \( T \), as shown in Lemma 4.7. Here, we instantiate \( \mathcal{Act} \) (Def. 4.6) as \( \mathcal{Act}_T \), which is the set of labels fired along \( T \)'s transitions in \( T \), (Def. 4.2); notably, \( \mathcal{Act}_T \) is finite and syntactically determined. (Details: §B.2)

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

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

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

- (i) if we do not want \( t \) to perform an action on channel \( x \), we check that \( T \) never potentially uses type variable \( x \).
- (ii) if we want \( t \) to eventually perform an action on channel \( x \), we need \( t \) productive, and check that \( T \) eventually uses \( x \) — without doing "imprecise" actions before.

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

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

- **input uses:** \( \mathcal{U}^\text{in}_{\Gamma,T}(x) = \{S(U') \in \mathcal{A}_T(T) \mid \Gamma \vdash x \leq S'\} \)
- **output uses:** \( \mathcal{U}^\text{out}_{\Gamma,T}(x) = \{\mathcal{S}^\text{in}(U') \in \mathcal{A}_T(T) \mid \Gamma \vdash x \leq S'\} \)

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

\[
\Gamma \vdash T \xrightarrow{\alpha} T' \quad \forall S, U : \alpha \in (S(U), \mathcal{S}(U)) \text{ implies } S \in \Upsilon \]

\[
\Gamma \vdash t \xrightarrow{\alpha} t' \quad \forall w, w' : \alpha \in (w(w'), \mathcal{S}(w')) \text{ implies } w \in \Upsilon \]

**Theorem 4.10.** Within productive \( \lambda^\Upsilon_{\xi} \), assume \( \Gamma \vdash t : T \), with \( \Gamma \vdash T \pi \text{-type, } \text{proc} \notin T \). Also assume, for all \( i[S, \Pi(x;U') \in T \text{ occurring in } T \), that there is \( y \) such that \( \Gamma \vdash y : U \) holds.²

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

Assume \( \Gamma \vdash t : T \). The sets \( \mathcal{U}^\text{in}_{\Gamma,T}(x) / \mathcal{U}^\text{out}_{\Gamma,T}(x) \) 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 \( \downarrow_\Gamma \{x_i \mid i \leq n\} \) (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 \( \tau \)-actions; i.e., \( x_1, \ldots, x_n \) are interfaces to other types/processes, and are "probed" for verification (this is common in model checking tools).

**Example 4.11.** Consider the type:

\[
T = i \left[x, \Pi(y ; \text{int}) p \left[\mu t.0 \begin{array}{c} z \in \text{int} \Pi(t), \\ \mu t'.1 \begin{array}{c} z \in \text{int} \Pi(t'), \Rightarrow \text{int}(x, \text{int}, \Pi(t') \end{array}\end{array}\right]\right]\[

It processes that receive an integer \( y \) on channel \( x \), then spawn two threads that recursively exchange \( y \) on channel \( z \), and at each loop, send an integer on channel \( x \). We have \( \Gamma \vdash T \pi \text{-type, with } \Gamma = \chi = \text{int}[\text{int}], \chi = \text{int}[\text{int}] \). By Def. 4.2, \( \Gamma \) has the transitions in Fig. 8 (top), i.e., \( T \) can receive an integer on \( x \) and move to \( T' \), where it could perform either an output on \( z \), an input on \( z \), or a synchronisation on \( z \) in the last two cases, the type performs an output on \( x \) and loops back to \( T' \).

When analysing the behaviour of \( T \), we might be interested in verifying that an initial input on \( x \) is eventually followed by an output on \( x \), without need of further external interactions. To this purpose, we want to focus our analysis on the inputs/outputs on \( x \) and let \( T \) reduce autonomously on any other channel (in this case, the only other channel is \( z \)): this amounts to pruning the input/output transitions not occurring on channel \( x \), while keeping all synchronisation

²This implicitly requires \( \Gamma \vdash T \pi \text{-type, hence } \text{fv}(U) \cap \text{btv}(T) = \emptyset \); 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 \( \not\exists y \) such that \( \Gamma \vdash y : U \); we can pick \( y \not\in \text{dom}(\Gamma) \), extend \( \Gamma \) as \( \Gamma = \Gamma, y' : U \); and get \( \Gamma \vdash t : T \).

Alceste Scalas, Nobuko Yoshida, and Elias Benussi
Transitions. This is achieved by observing the \{x\}_{i \in 1..n} -limited transitions of \(T\), i.e., \(\mathcal{T} \mapsto \{x\}_{i \in 1..n}\) (Def. 4.9), that yields the LTS in Fig. 8 (bottom). The same reasoning can be applied on the transitions of \(\lambda_{\Sigma}\) terms: by observing the transitions of \(\mathcal{T} \mapsto \{x\}_{i \in 1..n}\), we only focus on the inputs/outputs of \(T\) channel \(x\), while other channels are only used for synchronisation.

In Thm. 4.10, item (1) can be seen as a case of intuition (i1) above: if \(T\) never fires a label (\(\square\)) that is a potential output use of \(x_i\) (\(i \in 1..n\)), then \(T\) never uses \(x_i\) for output. The “potential output use”, by Def. 4.8, is any label \(S'U'(\tau)\) fired by \(T\) where \(S'\) is a supertype of \(x\): this accounts for “imprecise typing”, discussed in Ex. 3.7. 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 \(\Sigma(U)\); moreover, we check that \(T\) never fires any label in \(\mathcal{A}_\tau\), until \(U\) is output \(\Sigma(U)\) occurs. The set \(\mathcal{A}_\tau\) 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 \(\mathcal{A}_\tau\), then \(T\) might deadlock and never perform \(\Sigma(U)\) (for any \(U\)). Finally, item (4) combines the intuitions of both previous cases: we want to ensure that whenever \(T\) receives \(z\) on channel \(x\), then it eventually forwards \(z\) through channel \(y\), without doing other inputs on \(x\) before; to this purpose, we check that whenever \(T\) inputs \(z\) on a channel \(S\) (representing a potential use of \(x\)), then \(T\) eventually fires \(\Sigma(z)\) — without doing potential inputs on \(x\), nor firing any label in \(\mathcal{A}_\tau\), before.

Example 4.12. Take \(\Gamma, \tau, T\) in Ex. 4.3. To ensure that \(T\) eventually uses \(y\) to output a message, we check \(\mathcal{T} \mapsto \{\tau\} \models \phi\) with \(\phi\) in Fig. 7(3) (right).

Take ponger (Ex. 2.2), \(T_{pong}\) (Ex. 3.4), and \(\Gamma = z : \text{str} \rightarrow [\text{str}][\text{str}]\). To ensure that the term \(\text{pong} z\) is responsive on \(z\), we check \(\mathcal{T} \mapsto \{\tau\} \models \phi\) with \(\phi\) in Fig. 7(6) (right).
and the same mechanism tracks aliased channels across communications (since \texttt{send}(z, x, ...) / \texttt{send}(z, y, ...) have different types, and \(x/y\) are substituted in the receiver’s type).

5 Implementation and Evaluation

We designed \(\lambda T\) 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 foundation \(D<\) [2]. This naturally leads to a three-step implementation strategy: (1) internal embedding of \(\lambda T\); (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://alceste.github.io/effpi

5.1 Implementation

A payoff of the \(\lambda T\) design is that we can implement it as an internal embedded domain-specific language (EDSL) in Dotty; i.e., we can reuse Dotty’s syntax and type system, to define: (1) typed communication channels, (2) dedicated methods to render the \(\lambda T\) concurrency primitives (\texttt{send}, \texttt{recv} \[\parallel\], \texttt{end}), and (3) dedicated classes to render their types (\texttt{int}, \texttt{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 \texttt{receive(c)} \{ \texttt{x := P} \} does not cause an input from channel \(c\). Instead, the receive method returns an object of type \texttt{In[...]} (corresponding to \texttt{i[...]}} in Def. 3.1), which describes the act of using \(c\) to receive a value \(v\), and continue as \texttt{P\{v|x\}}. Such objects are executed by the Effpi interpreter, according to the \(\lambda T\) semantics (Def. 2.5).

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

\[
\Pi(x:T) o x \equiv (x:T) \Rightarrow \text{Out}[y.type, x.type, T^']
\]

Thus, the Scala compiler can check the program syntax (§2) and perform type checking (§3), ensuring type safety (Thm. 3.8). Dotty also supports (local) type inference.

For better usability, Effpi also provides some extensions over \(\lambda T\), like buffered channels, and a sequencing operator \("\Rightarrow\)" (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 \(\text{Par}[\text{Out}[x.type, Int], \text{Out}[x.type, Int]]\).

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 \(\lambda T\), input/output actions and their continuations are represented by \(\lambda\)-terms (closures), that can be easily stored away (e.g., when waiting for an input from a channel), and executed later (e.g., when the desired input becomes available). Thus, we implemented a non-preemptive scheduling system partly inspired by Akka dispatchers [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 \(\lambda T\) 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 \(c[T]\) (Def. 3.1). Hence, \(A\) can only use \(m\) to receive messages of type \(T\) — i.e., \(m\) is \(A\)’s mailbox;
Verifying Message-Passing Programs with Dependent Behavioural Types (tech. report)

- A receives $T$-typed messages by calling read — which is syntactic sugar for `recv(m, ...)` (see Fig. 1, and notice that the input channel $m$ is left implicit);
- other processes/actors can send messages to $A$ through its `ActorReference $r$` — which is just the output endpoint of its channel/mailbox $m$. The type of $r$ is $\text{co}[T]$ (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 $\lambda^\text{eff}_{\Pi}$ 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`:

```plaintext
@effpi.verifier.verify(\phi)
def f(x: ..., y: ...): T = ...
```

Such annotations ask to check if a program of type $T$ satisfies $\phi$, which is a conjunction/disjunctions of the properties from Fig. 7 (left). Note that $\top$ 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 $\lambda^\text{eff}_{\Pi}$ type $T'$, as per Def. 3.1;
2. checks if $T \models \phi'$ holds — where $\phi'$ is the companion formula of $\phi$ in Fig. 7 (right). This step uses the mCRL2 model checker [23]: we encode $T$ into an mCRL2 process, and check if $\phi'$ holds;
3. returns an error (located at the code annotation) if steps 1 or 2 fail. Otherwise, the compilation proceeds.

When compilation succeeds, any program of return type $T$ (including $f$ above) enjoys the property $\phi$ at run-time, by Thm. 4.10. This works both when $f$ is implemented, and when it is an unimplemented stub (i.e., when $f$ is defined as “???” in Dotty). This allows to compose the types/protocols of multiple services, and verify their interactions, even without their full implementation. E.g., consider Ex. 2.2, 3.4, and 4.12: 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. 3.4), 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 accepts payments after notifying on aud (with a variation of properties (5), (4) in Fig. 7, right).

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

3To obtain an mCRL2 encoding of $T$ with semantics adhering to Def. 4.2, we use the encoding into CCS (without restriction) mentioned after Lemma 4.7.
It does not check productivity of annotated code: such checks are unsupported in Dotty, and in most programming languages. Hence, programmers must ensure that all functions invoked from their Effpi code eventually return a value — otherwise, liveliness properties might not hold at run-time (cf. condition (c1) in §4).

2. It does not verify processes with unbounded parallel components (i.e., with parallel composition under recursion), hence, it rejects types having $\mu \pi$ under $\eta$. This does not impact the examples in this paper.

3. It uses iso-recursive types [60, Ch. 21] because, unlike $\lambda \pi_0$ (Def. 3.2), Dotty does not have equi-recursive types.

Limitations 1 and 3 might be avoided by implementing $\lambda \pi_0$ 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.

\section{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.

\subsection*{Run-time benchmarks}

We adopted a set of benchmarks from the Savina suite [31], with diverse interaction patterns:

- \textit{chamenoes}: $n$ actors ("chamenoes") connect to a central broker, who picks pairs and sends them their respective ActorReferences, so they can interact peer-to-peer [34];
- \textit{counting}: actor $A$ sends $n$ numbers to $B$, who adds them;
- \textit{fork-join} — \textit{creation (FJ-C)}: creation of $n$ new actors, who signal their readiness to interact;
- \textit{fork-join} — \textit{throughput (FJ-T)}: creation of $n$ new actors, and transmission of a sequence of messages to each.
- \textit{ping-pong}: $n$ pairs of actors exchange requests–responses;
- \textit{ring}: $n$ actors, connected in a ring, pass each other a token;
- \textit{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:

- \textit{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);
- \textit{memory vs. size}: how many times the JVM garbage collector runs, depending on the size of the benchmark — and also the maximum memory used before collection.

The results are in Fig. 9: 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 "chamenoes" (Effpi is $\sim 2x$ slower); the positive exceptions are fork-join throughput (Effpi is $\sim 2x$ faster), and the ring variants (Akka has exponential slowdown).

\subsection*{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. 10. 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 fuse dependent function types, in order to (1) connect a type variable \( x \) to a process variable \( x \), and (2) gain a form of type-level substitution (Def. 3.1).

Item (2), in particular, is not present in previous work; we take advantage of it to compose protocols (Ex. 3.4) 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–channel passing, thus covering an important feature of modern programming languages; (2) higher-order processes that send/receive mobile code, thus covering an important feature of modern programming toolkits (Ex. 3.5, 4.12). 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, 80]: 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.

Various \( \pi \)-calculus type systems specialise on accurate deadlock-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 \( \mu \)-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): the intuition is that, by using infinite-state types with unbounded parallel sub-terms (i.e., with \( \mu \)-types under \( \mu \)-types), we could model any number of unordered pending messages waiting to be received. In practice, this requires a linear-time \( \mu \)-calculus model checker that supports the resulting infinite-state systems, and we are not currently aware of any such tool (see footnote 4).

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., \( T = \{ p \mid o[x, y, T, a], o[x, z, T', b], \sum_{i} a \Pi(z,e^{\alpha}[int])U \} \) types parallel processes with a race on channel \( x \); we can verify such processes, capturing that either \( y \) or \( z \) may replace \( z \) in the \( U \)-typed continuation. This covers locking/mutex protocols, allowing, e.g., to implement and verify Dijkstra’s dining philosopher problem (mentioned in Fig. 10). [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 \( \lambda ^{\pi} \) 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 Brc [33] (that does not cover the linear-time \( \mu \)-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 Effp1 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

Verifying Message-Passing Programs with Dependent Behavioural Types (tech. report)
A \( \lambda_t \)-Calculus and Type System

The definition below is reprised from [32, §2].

**Definition A.1** (Positive/negative position of a type variable). We define the polarised free variables of \( T \), written \( \text{fv}^+(T) \) and \( \text{fv}^-(T) \), as follows:

\[
\begin{align*}
\text{fv}^+(T) & = \emptyset \\
\text{fv}^+(\bot) & = \emptyset \\
\text{fv}^+(\text{bool}) & = \emptyset \\
\text{fv}^+(()) & = \emptyset \\
\text{fv}^+(c[T]) & = \text{fv}^+(c^o[T]) = \text{fv}^+(T) \\
\text{fv}^+(\text{nil}) & = \emptyset \\
\text{fv}^+(S(T)) & = \text{fv}^+(S) \cup \text{fv}^+(T) \\
\text{fv}^+(S, T) & = \text{fv}^+(S) \cup \text{fv}^+(T) \cup \text{fv}^+(U) \\
\text{fv}^+(p(T, U)) & = \text{fv}^+(T) \cup \text{fv}^+(U) \\
\text{fv}^+(T \cup U) & = \text{fv}^+(T) \cup \text{fv}^+(U) \\
\text{fv}^+(\Pi(x:T) U) & = \text{fv}^+(T) \cup (\text{fv}^+(U) \setminus x) \\
\text{fv}^+(\mu_x.T) & = \text{fv}^+(T) \setminus x \\
\text{fv}^+(x) & = \{x\} \\
\text{fv}^-(x) & = \emptyset
\end{align*}
\]

B Linear-time \( \mu \)-calculus and type/process verification

This appendix contains additional definitions complementing §4.

B.1 Linear-time \( \mu \)-calculus

The definitions and notation below are mainly reprised from [20, §3], and [6, 73].

**Definition B.1** (Words over a set). Given a set \( \mathcal{Y} \), we define \( \mathcal{Y}^* \) and \( \mathcal{Y}^\omega \) as the sets of finite and infinite words over \( \mathcal{Y} \), respectively; we also define \( \mathcal{Y}^{\omega \omega} = \mathcal{Y}^* \cup \mathcal{Y}^\omega \). Given a word \( \sigma = a_1a_2a_3 \ldots \in \mathcal{Y}^{\omega \omega} \), we define \( \text{hd}(\sigma) = a_1 \), and \( \text{tl}(\sigma) = a_2a_3 \ldots \); we denote the empty word as \( \epsilon \), and leave \( \text{hd}(\epsilon) \) and \( \text{tl}(\epsilon) \) undefined.

**Definition B.2** (Semantics). Given a set of actions \( \text{Act} \), a valuation \( \mathcal{V} \) is a partial mapping from propositional variables to sets of words over \( \text{Act} \) — i.e., if \( Z \in \text{dom}(\mathcal{V}) \), then \( \mathcal{V}(Z) \subseteq \text{Act}^{\omega} \); given a set of words \( \mathcal{W} \subseteq \text{Act}^{\omega} \), let \( \mathcal{V}_{\mathcal{W}/z} \) be the valuation such that \( \mathcal{V}_{\mathcal{W}/z}(Z) = \mathcal{W} \) and \( \mathcal{V}_{\mathcal{W}/z}(Z') = \mathcal{V}(Z') \) (when \( Z' \neq Z \)). The *denotation of a linear-time \( \mu \)-calculus formula \( \phi \) under valuation \( \mathcal{V} \)*, written \( \|\phi\|_{\mathcal{V}} \), is the set of words of \( \text{Act}^{\omega} \) inductively defined as:

\[
\begin{align*}
\|Z\|_{\mathcal{V}} & = \mathcal{V}(Z) \\
\|\neg \phi\|_{\mathcal{V}} & = \text{Act}^{\omega} \setminus \|\phi\|_{\mathcal{V}} \\
\|\phi_1 \land \phi_2\|_{\mathcal{V}} & = \|\phi_1\|_{\mathcal{V}} \cap \|\phi_2\|_{\mathcal{V}} \\
\|\alpha\phi\|_{\mathcal{V}} & = \{\sigma \in \text{Act}^{\omega} \mid \text{hd}(\sigma) = \alpha \text{ and } \text{tl}(\sigma) \in \|\phi\|_{\mathcal{V}}\} \\
\|v Z.\phi\|_{\mathcal{V}} & = \bigcup \{\mathcal{W} \subseteq \text{Act}^{\omega} \mid \mathcal{W} \subseteq \|\phi\|_{\mathcal{V}_{\mathcal{W}/z}}\}
\end{align*}
\]

Given a labelled transition system \( T \) with initial state \( s_0 \) and labels in \( \text{Act} \), we say that \( T \) satisfies \( \phi \), written \( T \models \phi \), iff every run\(^5\) of \( T \) belongs to \( \|\phi\|_0 \).

**Definition B.3** (Extended constructs). Using the basic linear-time \( \mu \)-calculus productions (left-hand side of Def. 4.6), we define the following extended formulas (right-hand side of Def. 4.6),

\(^5\)A run of \( T \) is a (finite or infinite) sequence of transition labels obtained by starting from the initial state \( s_0 \), until a state without outgoing transitions is reached.
Verifying Message-Passing Programs with Dependent Behavioural Types (tech. report)

<table>
<thead>
<tr>
<th>Formula</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊤</td>
<td>νZ.Z</td>
<td>true (denotation is Act^o)</td>
</tr>
<tr>
<td>⊥</td>
<td>¬⊤</td>
<td>false (denotation is Ø)</td>
</tr>
<tr>
<td>φ₁ ∨ φ₂</td>
<td>¬((¬φ₁ ∧ ¬φ₂)</td>
<td>φ₁ holds, or φ₂ holds</td>
</tr>
<tr>
<td>φ₁ ⇒ φ₂</td>
<td>¬φ₁ ∨ φ₂</td>
<td>if φ₁ holds, then φ₂ holds</td>
</tr>
<tr>
<td>μZ.φ</td>
<td>¬νZ.¬φ{¬Z/z}</td>
<td>least fixed point (denotation is a set of words of finite length)</td>
</tr>
<tr>
<td>(A)φ</td>
<td>∨_α∈Α (α)φ</td>
<td>after some action α in Α, φ holds</td>
</tr>
<tr>
<td>(¬A)φ</td>
<td>∨_α∈Α∩(Α) (α)φ</td>
<td>after some action α not in Α, φ holds</td>
</tr>
<tr>
<td>φ₁ U φ₂</td>
<td>μZ.φ₁ φ₂ ∨ (μZ ∧ (Act)Z)</td>
<td>φ₁ holds (for a finite number of actions), until φ₂ holds</td>
</tr>
<tr>
<td>◯φ</td>
<td>T U φ</td>
<td>φ eventually holds, after a finite number of actions</td>
</tr>
<tr>
<td>□φ</td>
<td>¬◯(¬φ)</td>
<td>φ always holds</td>
</tr>
</tbody>
</table>

B.2 Actions of a type

The following is the definition of the set of actions A_T(T), that completes Def. 4.8.

Definition B.4 (Actions of a π-type). The basic actions of a π-type in Γ are defined as:

\[ B_Γ(nil) = B_Γ(t) = θ \quad B_Γ(t.Τ) = B_Γ(T) \quad B_Γ(\langle T, U \rangle) = B_Γ(T) \cup B_Γ(U) \]

\[ B_Γ(T \lor U) = \{ τ[V] \} \cup B_Γ(T) \cup B_Γ(U) \quad B_Γ(σ[S, T, Π]U) = \{ S(T) \} \cup B_Γ(U) \]

The (complete) actions of a π-type in Γ are defined as:

\[ A_Γ(T) = B_Γ(T) \cup \left\{ τ[S_S'] \mid S_Γ(T) \cap S_Γ(U) \in B_Γ(T) \text{ and } S_Γ(U') \in B_Γ(T) \text{ and } Γ + S \triangleright S' \right\} \]

The input and output uses of S by π-type T in Γ, written ⅈ_Γ.T.(S) and ⅉ_Γ.T.(S), are:

\[ ⅈ_Γ.T.(S) = \{ S_Γ(U') \in A_Γ(T) \mid Γ + S \triangleleft S' \} \quad ⅉ_Γ.T.(S) = \{ S_Γ(U') \in A_Γ(T) \mid Γ + S \triangleleft S' \} \]

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

\[ Γ + T \xrightarrow{a'} T' \quad (α = x(U) \text{ or } α = x ⟨U⟩) \quad \text{implies } x \in Y \]

\[ Γ + t \xrightarrow{a} t' \quad (α = x(w) \text{ or } α = x ⟨w⟩) \quad \text{implies } x \in Y \]

Intuitively, Def. B.4 computes the possible actions of a π-type T in two steps:

1. first, it computes the set of basic actions B_Γ(T), by performing a simple syntactic traversal of T. Some care is required to compute the actions of an input type T_in = [S, Π(x; T)U], that by Def. 4.2, could take different paths by firing different actions S(T′) for various payload types T′. For this reason,
   a. all possible payload types T′, according to the premises of rule [T→i], are collected in the set Y. Note that Y is always finite: it can contain at most T and all variables in Γ;
   b. then, for each T′ ∈ Y, the action S(T′) is added to B_Γ(T_in), together with the basic actions of the continuation U(T′)
2. then, it computes the (complete) set of actions A_Γ(T) by combining:
   a. B_Γ(T), and
   b. all possible communication actions τ[S, S'] obtained by pairing the actions in B_Γ(T), whenever they involve channel types that might communicate (T + S ↠ S" ⊢ T′, Def. 4.2).

Notably, to compute A_Γ(T) we need to compare types via subtyping, and thus, we need the judgement Γ + U ⊆ U' to be decidable (hence the remark about rule [≤ ii] in §3).

C Type system properties

C.1 \[λ^π_S\] as a specialisation of F_c:

The judgements in Fig. 4 can be reconnected to those of F_c [8] under an intuition based on the following encoding from \[λ^π_S\] to F_c:
Environments
\[
\begin{align*}
\Gamma \vdash \{ \emptyset \} & = \emptyset \\
\Gamma \vdash \{ x : T, \Gamma \} & = \{ x : T \} \cup \{ x : T, \Gamma \}
\end{align*}
\]

Types
\[
\begin{align*}
\Gamma \vdash \{ \Pi(x : T) U \} & = \forall (X, \Gamma \vdash \{ T \}) \rightarrow \Gamma \vdash \{ U \} \\
\Gamma \vdash \{ x \} & = \Gamma \vdash \{ x \}
\end{align*}
\]

Terms
\[
\begin{align*}
\Gamma \vdash \{ \lambda x : T. t \} & = \lambda (X, \Gamma \vdash \{ T \}) \rightarrow \Gamma \vdash \{ t \}
\end{align*}
\]

The idea is that:

1. a typing environment entry \( x : T \in \Gamma \) in \( \lambda F \Gamma \) corresponds to two typing environment entries in \( F_{\lessgtr} \): a type variable \( X \), with bound \( \{ T \} \), and a term variable \( x \) with type \( X \);
2. a dependent function type \( \Pi(x : T) U \) corresponds to a (non-dependent) function type \( X \rightarrow \{ U \} \), under the bounded quantification \( \forall (X, \Gamma \vdash \{ T \}) \rightarrow \Gamma \vdash \{ U \} \); a term variable \( x \) with type \( X \);
3. an occurrence of \( x \) in a \( \lambda F \Gamma \) type \( T \) corresponds to an occurrence of \( X \), in the encoded \( F_{\lessgtr} \)-type \( \{ T \} \); a term variable \( x \) with type \( X \).

Under the correspondence above, we can notice that:

- the typing rule \( \{ t : x \} \) in Fig. 4, that, inverts \( \Gamma \vdash x : x \), is an instance of rule (Val x) in \( [8] \), that, inverts \( \{ t \} \vdash x : X \);
- the subtyping rule \( \{ \leq \} \) in Fig. 4, that, inverts \( \Gamma \vdash x : T \), is an instance of rule (Sub X) in \( [8] \), that, inverts \( \{ \Gamma \} \vdash x : T \).

Indeed, all judgements in Fig. 4 (except for the highlighted, concurrency-related ones) are developed from those in \( [8] \) by following the correspondence above. For example, the \( \lambda F \Gamma \) typing rule

\[
\frac{\Gamma \vdash t_1 : \Pi(x : T) U \quad \Gamma \vdash t_2 : U \quad \Gamma \vdash U \leq U}{\Gamma \vdash t_1 t_2 : T \{ U \}}
\]

is obtained as an instance of (Val appl) and (Val appl2) in \( [8] \):

\[
\frac{\{ \Gamma \} \vdash \{ t_1 \} \quad \{ \Pi(x : T) U \} \vdash \{ U \} \quad \{ \Gamma \} \vdash \{ U \} \leq \{ U \} \quad \{ \Gamma \} \vdash \{ t_2 \} \quad \{ U \} \vdash \{ U \}}{\{ \Gamma \} \vdash \{ t_1 t_2 \} \quad \{ U \} \vdash \{ U \} \}
\]

i.e., the typing of a dependent function application \( t_1 t_2 \) in \( \lambda F \Gamma \) (via rule \( \{ t \} \)) corresponds, in \( F_{\lessgtr} \), to typing an application of bounded quantification (rule (Val appl2)), term \( \{ t_1 \} \vdash \{ U \} \), in turn is applied to \( \{ t_2 \} \vdash \{ U \} \) (rule (Val appl)). Note, in particular, that the application of bounded quantification if \( F_{\lessgtr} \) is responsible for the type-level substitution \( \{ \Pi(x : T) U \} \rightarrow \{ U \rightarrow \} \), that corresponds, to \( T \{ U \} \) in \( \lambda F \Gamma \).

The above correspondence also guides in adapting the results and proofs from \( F_{\lessgtr} \) to \( \lambda F \Gamma \), leading to the results in §C.2.

### C.2 Properties

**Lemma C.1.** \( \leq \) is a preorder, i.e.:

1. if \( \Gamma \vdash T \)-type, then \( \Gamma \vdash T \leq T \);
2. if \( \Gamma \vdash S, T, U \)-type, then \( \Gamma \vdash S \leq T \) and \( \Gamma \vdash T \leq U \) implies \( \Gamma \vdash S \leq U \).

**Proof.** Item 1 is immediate. For item 2, given \( \Gamma \), we build a relation

\[
\mathcal{R} = \{ (S, U) \mid \Gamma \vdash S \leq T \text{ and } \Gamma \vdash T \leq U \}
\]

and by inspecting each pair \((S, U) \in \mathcal{R}\), we prove that the judgement \( \Gamma \vdash S \leq U \) by some rule in Fig. 4. In most cases, this holds similar to \([32, \text{Prop. 2 and 3}]\), (cf. §C.1); the remaining cases are union types, channel types, and \( \pi \)-types: they are all easy. \( \Box \)

**Proposition C.2.** Assume \( \Gamma \vdash t : T \). Then, \( \text{fv}(t) \in \text{dom}(\Gamma) \).

**Proof.** By induction on the typing derivation. \( \Box \)

**Proposition C.3.** If \( t \in \text{env} \) and \( \Gamma(x) = T \), then \( x \notin \text{fv}(T) \).
We now prove that:

\[ \text{Proposition C.4. If } \Gamma \vdash \text{env and } \Gamma(x) = T, \text{ then } \Gamma \vdash x : T. } \]

Proof:

\[
\dfrac{
\Gamma \vdash x : T \quad \Gamma \vdash \text{env}
}{
\Gamma \vdash \text{env} \quad [t \cdot x]
\}
\]

\[ \vdash \text{env} \quad [t \cdot x]
\]

\[ \Gamma \vdash x : T \quad \Gamma \vdash x \leq T
\]

\[ \Gamma \vdash \text{env} \quad [\leq \text{refl}]
\]

\[ \Gamma \vdash x \leq T \quad \Gamma \vdash \text{env}
\]

\[ \Gamma \vdash \text{env} \quad [\leq \text{g}]
\]

\[ \Gamma \vdash x \leq T \quad [t \cdot]
\]

\[ \Gamma \vdash x : T
\]

\[ \Box
\]

Proposition C.5. If \( \Gamma \not\vdash T \leq \bot \), then \( \Gamma \vdash T \leq \bot \) implies \( \Gamma \vdash \bot \leq T \).

Proof: Observe that the premise \( \Gamma \vdash T \leq \bot \) can only hold under a derivation composed by rules \([\leq \text{refl}]\) and \([\leq \text{vL}]\); then, we can prove \( \Gamma \vdash \bot \leq T \) with a symmetric derivation, where each instance of \([\leq \text{vL}]\) is replaced with \([\leq \text{vR}]\).

Lemma C.6 (Typing inversion). Assume \( \Gamma \vdash t : T \) with \( \Gamma \vdash T \text{ type} \). Then, one of the following cases holds:

1. \( t = x \), and either:
   a. \( \Gamma \vdash T \leq x \); or
   b. \( \Gamma \vdash x \vdash \text{bool} \leq T \);
2. \( t = \nu \in \mathbb{E} \), and \( \Gamma \vdash \text{bool} \leq T \);
3. \( t = () \), and \( \Gamma \vdash () \leq T \);
4. \( t = \neg t' \) with \( \Gamma \vdash t' \vdash \text{bool} \), and \( \Gamma \vdash \text{bool} \leq T \);
5. \( t = \lambda x : U . t \) with \( \Gamma \vdash U \text{ type} \), and for some \( T' \) such that \( \Gamma, x : U \vdash t' : T' \) and \( \Gamma \vdash \Pi(x : U) \Gamma' \leq T \);
6. \( t = \text{if } t' \text{ then } t_1 \text{ else } t_2 \), and for some \( T_1, T_2 \) such that \( \Gamma \vdash T_1, T_2 \text{ type} \), we have \( \Gamma \vdash t' : \text{bool} \), and \( \Gamma \vdash t_1 : T_1 \) and \( \Gamma \vdash T_2 : T_2 \) and \( \Gamma \vdash T_1 \lor T_2 \leq T \);
7. \( t = t_1 t_2 \), and for some \( U, U', T' \) such that \( \Gamma \vdash U, U', T' \text{ type} \), we have \( \Gamma \vdash t_1 : \Pi(x : U) \Gamma' \) and \( \Gamma \vdash t_2 : U' \) and \( \Gamma \vdash U' \leq U \) and \( \Gamma \vdash T'(U'/2) \leq T \);
8. \( t = \text{let } x = x' = t_1 \text{ in } t_2 \) with \( \Gamma \vdash U \text{ type} \), and for some \( U', T' \) such that \( \Gamma \vdash U', T' \text{ type} \), we have \( \Gamma, x : U \vdash t_1 : U' \) and \( \Gamma, x : U \vdash t_2 : T' \) and \( \Gamma \vdash U' \leq U \) and \( \Gamma \vdash T'(U'/2) \leq T \);
9. \( t = a^T \) with \( \Gamma \vdash T \text{ type} \), and \( \Gamma \vdash \text{e}^0[T'] \leq T \);
10. \( t = \text{chan}(T') \) with \( \Gamma \vdash T' \text{ type} \), and \( \Gamma \vdash \text{e}^0[T'] \leq T \).

Proof: First, observe that the assumption implies \( T \neq \bot \) (otherwise, we would not have a typed term \( t \)).

Item 1. Assume \( \Gamma \vdash x : T \). Then, for some finite \( n \geq 0 \), letting \( T_0 = x \) and \( T_n = T \), the judgement can only be the conclusion of a derivation of the following form (where \( P \) denotes the premises of a judgement):

\[
\dfrac{
\Gamma \vdash x : T_0 \quad P_1 \quad P_2 \quad \cdots \quad P_{n-1} \quad P_n
}{
\Gamma \vdash x : T_n \quad [? n-1]
\}
\]

This is because:

- the instance of \([t \cdot x]\) on the top left is the only possible base case for a judgement of the form \( \Gamma \vdash x : U \);
- the \( i \)-th application of rule \([t \cdot \leq]\) requires an application of some subtyping rule \([? i]\) with (coinductive) premises \( P_i \), allowing to get \( \Gamma \vdash x \leq T_i \).

We now prove that:

\[ \forall i \in 1..n : \quad \Gamma \vdash T_i \leq x \text{ or } \Gamma \vdash \Gamma(x) \leq T_i \] (1)

We proceed by induction on \( i \):

- base case \( i = 0 \). Then, we have \( T_0 = x \) and conclude \( \Gamma \vdash T_0 \leq x \) by \([\leq \text{refl}]\) (Fig. 4);
- inductive case \( i = j + 1 \). Then, we must have \( \Gamma \vdash T_j \leq T_i \) for some subtyping rule \([? i]\). We have two possibilities:
  1. \([? j] = [\leq \text{g}]\). This implies \( T_j = x \) and \( P_j = \Gamma \vdash \Gamma(x) \leq T_j \). Hence, we conclude \( \Gamma \vdash \Gamma(x) \leq T_i \);
  2. \([? j] \neq [\leq \text{g}]\). By the induction hypothesis, we have either:
a. \( \Gamma \vdash T_j \leq x \). Observe that, from the premise of the \( i \)-th application of rule \([r \leq \cdot]\), we have \( \Gamma \vdash x \leq T_i \). Therefore, by Prop. C.5, we conclude \( \Gamma \vdash T_i \leq x \);

b. \( \Gamma \vdash \Gamma(x) \leq T_j \). Then, by Lemma C.1(2) (subtyping transitivity) we have \( \Gamma \vdash x \leq T_i \); thus, we conclude \( \Gamma \vdash \Gamma(x) \leq T_i \).

Now, having proved (i), and reminding \( T_n = T \), we obtain that either \( \Gamma \vdash T \leq x \) or \( \Gamma \vdash \Gamma(x) \leq T \) holds—which is the thesis.

\textbf{Items 2–10.} By cases on the rule concluding \( \Gamma \vdash t : T \).

\begin{proof}

\textbf{Lemma C.7 (Typing inversion for \( \pi \)-types). Assume} \( \Gamma \vdash t : T \) \textit{with} \( \Gamma \vdash T \text{-} \pi \text{-type. Then, one of the following cases holds:}

\begin{enumerate}
\item \( t = \text{end} \), and \( \Gamma \vdash \text{nil} \leq T \);
\item \( t = \text{send}(t_1, t_2, \lambda x^1(t_3)) \) and for some \( S', T_1, T_n, U', T' \) such that \( \Gamma \vdash S' \leq c^\th(T_n) \) and \( \Gamma \vdash T' \text{-type} \) and \( \Gamma \vdash T' \leq T_n \) and \( \Gamma \vdash U' \text{-}\pi \text{-type} \), we have \( \Gamma \vdash t_1 : S' \) and \( \Gamma \vdash t_2 : T' \) and \( \Gamma \vdash t_3 : U' \) and \( \Gamma \vdash \mathbf{a}(S', T, \Pi(U')) \leq T \);
\item \( t = \text{recv}(t_1, \lambda x^1(T_2)) \) with \( \Gamma \vdash T' \text{-type} \) and for some \( S', T_1, T_n, U' \) such that \( \Gamma \vdash S' \leq c^\th(T_n) \) and \( \Gamma \vdash T_1 \leq T' \) and \( \Gamma, x : T' \vdash U' \text{-}\pi \text{-type} \), we have \( \Gamma \vdash t_1 : S' \) and \( \Gamma, x : T' \vdash t_2 : U' \) and \( \Gamma \vdash [S', \Pi(x : T')U'] \leq T \);
\item \( t = \text{if} \ t' \text{ then } t_1 \text{ else } t_2 \), and for some \( T_1, T_2 \) such that \( \Gamma \vdash T_1, T_2 \text{-}\pi \text{-type} \), we have \( \Gamma \vdash t' : \text{bool} \) and \( \Gamma \vdash t_1 : T_1 \) and \( \Gamma \vdash t_2 : T_2 \) and \( \Gamma \vdash T_1 \land T_2 \leq T \);
\item \( t = \text{let} \ x^U = t_1 \text{ in } t_2 \), and for some \( U, U', T' \) such that \( \Gamma \vdash U, U' \text{-}\pi \text{-type} \) and \( \Gamma \vdash T' \text{-}\pi \text{-type} \), we have \( \Gamma \vdash t_1 : \Pi(x : U)U' \) and \( \Gamma \vdash t_2 : U' \) and \( \Gamma \vdash U' \leq U \) and \( \Gamma \vdash T' \leq \{U'/x\} \leq T \);
\item \( t = \text{let} \ x^U = t_1 \text{ in } t_2 \), and for some \( U', T' \) such that \( \Gamma \vdash U' \text{-}\pi \text{-type} \) and \( \Gamma \vdash T' \text{-}\pi \text{-type} \), we have \( \Gamma \vdash x : U \) and \( \Gamma, x : U \vdash t_2 : T' \) and \( \Gamma \vdash U' \leq U \) and \( \Gamma \vdash T' \leq \{U'/x\} \leq T \).
\end{enumerate}

\textbf{Proof.} By cases on the rule concluding \( \Gamma \vdash t : T \).

\end{proof}

\textbf{Proposition C.8. Assume} \( \Gamma \vdash x : T \). \textbf{Then,} \( \Gamma \vdash x \leq T \).

\textbf{Proof.} By Lemma C.6(1), we have two cases:

- \( \Gamma \vdash T \leq x \). Then, we conclude by Prop. C.5:
  \( \Gamma \vdash \Gamma(x) \leq T \).

- \( \Gamma \vdash \Gamma(x) \leq T \). Then, we conclude by \( \Gamma \vdash x \leq T \) \( [\leq \cdot] \).

\end{proof}

\textbf{Proposition C.9. Assume} \( \Gamma \vdash E[t] : T \) \textbf{with} \( \Gamma \vdash T \text{-type} \). \textbf{Then,} \( \exists \Gamma', T' \text{ such that} \Gamma \leq \Gamma' \) \textbf{and} \( \Gamma' \vdash t : T' \).

\textbf{Proof.} By induction on \( E \) and the derivation of \( \Gamma \vdash E[t] : T \), using Lemma C.6.

\end{proof}

\textbf{Proposition C.10. Assume} \( \Gamma \vdash E[t] : T \), \textbf{with} \( t = \text{send}(t_1, t_2, t_3) \) \textbf{or} \( t = \text{recv}(t_1, t_2) \) \textbf{or} \( t = t_1 \mid t_2 \), and \( \Gamma \vdash T \text{-}\pi \text{-type} \). \textbf{Then,} \( \exists E', T' \text{ such that} \Gamma \vdash T' \text{-}\pi \text{-type} \), \( \Gamma \vdash E[t] : E'[T'] \), \( \Gamma \vdash E'[T'] \leq T \), \textbf{and} \( \Gamma' \vdash t : T' \).

\textbf{Proof.} By induction on \( E \) and the derivation of \( \Gamma \vdash E[t] : T \), using Lemma C.7(4).

\end{proof}

\textbf{Lemma C.11 (Substitution). Assume} \( \Gamma, x : U \vdash t : T \) \textbf{and} \( \Gamma \vdash w : U \) \textbf{(with} \( w \in \forall \cup \exists \)). \textbf{Then,} \( \Gamma \vdash t[w/x] : T \{U/x\} \).

\textbf{Proof.} By induction on the derivation of \( \Gamma, x : U \vdash t : T \).

\end{proof}

\textbf{Proposition C.12. For all} \( t, T, T' \text{, } T'' \text{ such that} \Gamma \vdash T \text{-}\pi \text{-type} \), \( \Gamma \vdash T' \text{-}\pi \text{-type} \) \textbf{and} \( \Gamma \vdash T'' \text{-}\pi \text{-type} \), \textbf{if} \( \Gamma \vdash t : T \) \textbf{and} \( T' \equiv \text{proc} \lor T'' \), \textbf{then} \( \Gamma \vdash t : T' \).

\textbf{Proof.}

\begin{align*}
\frac{\Gamma \vdash t : T \quad \Gamma \vdash T \leq \text{proc} \quad \Gamma \vdash \text{proc} \leq T'}{\Gamma \vdash t : T'} & \quad [\leq \text{proc}] \\
& \quad \text{by [\leq \text{max}] and Lemma C.1(2)} \\
& \quad \text{by \([\leq \cdot \leq \cdot]\)} \\
& \quad \text{[\leq \cdot]} \\
\end{align*}
D Subject transition (Thm. 4.4)

Proposition D.1. Assume \( \Gamma \vdash \mathcal{E}[t] : T \), and
\[
\Gamma \vdash t \triangleright_{\alpha} t' \quad \text{with} \quad \alpha \in \left( \{ \text{[R-λ]}, \text{[R-let]}, \text{[R-chan]}, \text{[R-if]}, \text{[R-if-if]}, \text{[R-if-if]} \} \cup \{ \tau[x], \tau[\text{if } x], \tau[\text{if } o], \tau[\text{if } o] \mid x \in \mathbb{X} \} \right)
\]

Then, \( \Gamma \vdash \mathcal{E}[t] \models \mathcal{E}[t'] \) and \( \Gamma \vdash \mathcal{E}[t'] : T \).

Proof: By rule \([sr]_{\mathcal{E}}\) in Def. 4.1, we obtain \( \Gamma \vdash \mathcal{E}[t] \models \mathcal{E}[t'] \).

We now prove \( \Gamma \vdash \mathcal{E}[t'] : T \). We have two possibilities:

1. \( \Gamma \vdash T \) type. First, using Prop. C.9, we show that there are \( \Gamma' \), \( T' \) such that \( \Gamma' \subseteq \Gamma \) and \( \Gamma' \vdash t : T' \). Then, by induction on the derivation of \( \Gamma + t \triangleright_{\alpha} t' \) using Lemma C.6 (inversion of typing), we prove \( \Gamma' + t' : T' \). Finally, using Lemma C.11, we conclude \( \Gamma \vdash \mathcal{E}[t'] : T \).

2. \( \Gamma \vdash T \pi\text{-type} \). Then, \( \mathcal{E} = \mathcal{P} \) for some \( \mathcal{P} \) (Def. E.1), and thus, \( t \) occurs within \( \text{send}(\_, \_, \_\_\_) \) or \( \text{recv}(\_, \_\_) \), possibly inside some instances of \( \_ \). Hence, using Prop. C.10, we show that there are \( \mathcal{E}'', T'' \) such that \( \Gamma + \mathcal{E}'', T'' \leq T \) and \( \Gamma \vdash \mathcal{P}[t] : \mathcal{E}''[T''] \) and \( \Gamma \vdash t : T'' \). Then, we proceed as in case 1 above to show that, after \( t \) reduces to \( t' \), we have \( \Gamma \vdash t' : T'' \). Finally, using Lemma C.11 again, we conclude \( \Gamma \vdash \mathcal{E}[t'] : T \).

\( \square \)

Proposition D.2. Assume \( \Gamma \vdash t : T \) with \( \Gamma \vdash T \pi\text{-type} \). Then, \( \Gamma \vdash t \overset{\tau[w]}{\rightarrow} t' \) implies either:

1. \( \Gamma \vdash t' : T \) and \( \text{proc} \in T \); or

2. \( \exists S, U, T', \Gamma \vdash x : S, w : U, t' : T' \) and \( \Gamma \vdash T \overset{\tau[v]}{\rightarrow} S(T) \overset{T'}{\rightarrow} T' \).

Proof: Assume \( \Gamma \vdash t \overset{\tau[w]}{\rightarrow} t' \); by inversion of the derivation of the transition, we have \( t = \mathcal{E}[\text{send}(x, w, t'')] \), for some \( \mathcal{E}, t'' \). By Prop. C.10, \( \exists S', T_0 \) such that \( \Gamma + \mathcal{E}'[T_0] \leq T \), \( \Gamma \vdash t : \mathcal{E}'[T_0] \) and \( \Gamma \vdash \mathcal{E}[\text{send}(x, w, t'')] : T_0 \). By Lemma C.7(2), \( t'' = \lambda x^{(i)}.t''' \), hence we know that \( t' = \mathcal{E}[(\lambda x^{(i)}.t''')] \); moreover, again by Lemma C.7(2), for some \( S'', T_1, T_2, t'', U'' \), such that \( \Gamma + S'' \leq S_0[T_0] \) and \( \Gamma + U'' \text{-type} \) and \( \Gamma \vdash U'' \leq T_0 \) and \( \Gamma \vdash T'' \pi\text{-type} \), we have \( \Gamma \vdash x : S'' \) and \( \Gamma \vdash w : U'' \) and \( \Gamma \vdash t'' : T'' \) and \( \Gamma \vdash \text{proc}[S'', U'', \Pi(T'')] \leq T_0 \). We now have two possibilities:

- if \( T_0 \equiv \text{proc} \vee T_1 \) with \( \Gamma \vdash S'' \leq S_1, U'' \leq U_1, T'' \leq T_1 \), then we have \( \Gamma \vdash \mathcal{E}'[T_0] \overset{S(T)}{\rightarrow} \mathcal{E}'[T_1] \), and two more sub-cases:
  - if \( T_0 \equiv \text{proc} \vee T_2 \) with \( \Gamma \vdash S \leq S_2, U \leq U_2, T \leq T_2 \), then by letting \( S = S_2, U = U_2 \) and \( T' = T_2 \), we get
    \[
    \Gamma \vdash x : S, w : U, t' : T' \quad \text{(by \([t \leq i]\))} \quad \text{and} \quad \Gamma \vdash T \overset{\tau[v]}{\rightarrow} S(T) \overset{T'}{\rightarrow} T', \]
    and we conclude by obtaining item 2;
  - otherwise, we have \( T = \text{proc} \vee T_2 \). In this case, we get \( \Gamma \vdash t : T \) (by Prop. C.12) and \( \text{proc} \in T \), and conclude by obtaining item 2.

- otherwise, we have \( T_0 \equiv \text{proc} \vee T_1 \). In this case, we get \( \Gamma \vdash t'' : T_0 \) (by Prop. C.12), and thus, \( \Gamma \vdash t' : \mathcal{E}'[T_0] \); hence, since \( \Gamma \vdash \mathcal{E}'[T_0] \leq T \), we must have \( \text{proc} \in T \). Therefore, we conclude by obtaining item 2.

\( \square \)

Proposition D.3. Assume \( \Gamma \vdash t : T \) with \( \Gamma \vdash T \pi\text{-type} \). Then, \( \Gamma \vdash t \overset{\tau[x]}{\rightarrow} t' \) implies either:

1. \( \Gamma \vdash t' : T \) and \( \text{proc} \in T \); or

2. \( \exists S, U, T', \Gamma \vdash x : S, x : S', t' : T' \) and \( \Gamma \vdash T \overset{\tau[v]}{\rightarrow} S(S') \overset{T'}{\rightarrow} T' \).

Proof: Similar to the proof of Prop. D.2, but using Lemma C.7(3).

\( \square \)

Proposition D.4. Assume \( \Gamma \vdash t : T \) with \( \Gamma \vdash T \pi\text{-type} \). Then, \( \Gamma \vdash t \overset{\tau[x]}{\rightarrow} t' \) implies either:

1. \( \Gamma \vdash t' : T \) and \( \text{proc} \in T \); or

2. \( \exists S, S', T', \Gamma \vdash x : S, x : S', t' : T' \) and \( \Gamma \vdash T \overset{\tau[v]}{\rightarrow} S(S') \overset{T'}{\rightarrow} T' \).

Proof: Similar to the proof of Prop. D.2, but using Lemma C.7(4).

\( \square \)

Proposition D.5. Assume \( \Gamma \vdash t : T \) with \( \Gamma \vdash T \pi\text{-type} \). Then, \( \Gamma \vdash t \overset{\tau[R-\text{Comm}]}{\rightarrow} t' \) implies either:

1. \( \Gamma \vdash t' : T \) and \( \text{proc} \in T \); or

2. \( \exists S, S', T' : S, S' \neq x, \Gamma \vdash t' : T' \) and \( \Gamma \vdash T \overset{\tau[v]}{\rightarrow} S(S') \overset{T'}{\rightarrow} T' \).

Proof: Similar to the proof of Prop. D.4.

\( \square \)
D.1 Proof of subject transition (Thm. 4.4)

Proof. Assume $\Gamma \vdash t : T$. If $\Gamma \vdash T$ type, then $\Gamma \vdash t \xrightarrow{\alpha} t'$ follows by Prop. D.1.

Now, assume $\Gamma \vdash T \pi$-type.

Item 1. Follows by Prop. D.1.

Item 2. By cases on $\alpha$, the result follows by either Prop. D.2, D.3, D.4, or D.5.

□

E Type Fidelity (Thm. 4.5)

Definition E.1 (Process evaluation context). A process evaluation context $\mathcal{P}$ is a restricted case of evaluation context $\mathcal{E}$ (Def. 2.5):

$$\mathcal{P} ::= [] \mid \text{send}(\mathcal{P}, t, t') \mid \text{send}(\mathcal{P}, w, t) \mid \text{send}(\mathcal{P}, w, t') \mid \text{recv}(\mathcal{P}, t) \mid \text{recv}(\mathcal{P}, w) \mid \mathcal{P} \parallel t \quad (w, w' \in \mathbb{V} \cup \mathbb{X})$$

E.1 Proof of type fidelity (Thm. 4.5)

Proof. Assume $\Gamma \vdash T \xrightarrow{\alpha} T''$ for some $\alpha$ matching one of items 1–4, and for some $T''$. By Def. 4.2 and inversion of the transition, for some $\mathcal{E}, T_0$, we have $T = \mathcal{E}[T_0]$, and $\Gamma \vdash T_0 \xrightarrow{\alpha} T''_0$. Moreover, since $\Gamma \vdash t : \mathcal{P}[T_0]$ for some $t_0, \mathcal{P}$ (Def. E.1) we have $t = \mathcal{P}[t_0]$ and $\Gamma \vdash t_0 : T_0$. Then, we have the following possibilities:

Item 1 ($\alpha = \overline{x}(U)$). Then, in the statement we have $T'' = \mathcal{E}[T''_0]$. By inversion of the transition $\Gamma \vdash T_0 \overline{x}(U) \xrightarrow{T''_0}$, we have $T_0 = o[x, U, \Pi(T'_0)]$. Therefore, by the productivity hypothesis, $\Gamma \vdash t_0 \xrightarrow{t''} T''_0$; by Thm. 4.4, we get $\Gamma \vdash t'' : T_0$; hence, we have the following possibilities:

1. $t'' \in \mathbb{V}$. Impossible, because it would imply $\Gamma \not\vdash t'' : T_0$, leading to the contradiction $\Gamma \not\vdash t : T$;
2. $t'' = z \in \mathbb{X}$. Impossible, because it would require $z\mathcal{T}'_0 \in \Gamma$ for some $T_0$ such that $\Gamma \vdash T_1 \leq T_0$; but then, $\Gamma \vdash T_0 \pi$-type, we would also have $\Gamma \vdash T_1 \pi$-type, that would imply $\Gamma \vdash \Gamma \text{env}$, leading to the contradiction $\Gamma \not\vdash t : T$;
3. $t'' \in \mathbb{P}$. Then, by Lemma C.7, we must have $t'' = \text{send}(x, w, w')$, with $\Gamma \vdash w : U$, and $\Gamma \vdash w' : \Pi(T'_0)$. Moreover, since $\Gamma \vdash U$ type and $\Gamma \vdash \Pi(T'_0)$ type (by $\langle \text{send} \rangle$), we also have $w, w' \in \mathbb{V} \cup \mathbb{X}$ (otherwise, we would contradict $t'' \not\vdash \mathcal{P}$); hence, by rule $\text{[SR-send]}$ (Def. 4.1), $\Gamma \vdash t'' \overline{\text{send}}(w')$, $w'(\), and by Thm. D.4(2b), $\Gamma \vdash w'() : T'_0$. Hence, we get $t' = \mathcal{P}[w']$ and $\Gamma \vdash t' : \mathcal{E}[T'_0] = T'$, which concludes the proof;
4. $t'' \in \mathbb{V}$. Similar to the proof for item 1 above, but concluding via Thm. D.4(2c);
5. $t'' \in \mathbb{P}$. Similar to the proofs for items 1 and 2 above, but concluding via Thm. D.4(2d);
6. $t'' \in \mathbb{P}$. Then, $T_0 \equiv T_1 \vee T_2$, and for some $i \in \{1, 2\}, \Gamma \vdash T_i \vee T_2 \overline{\text{t}} \rightarrow T_i$ and $\Gamma \vdash T'' \vdash \mathcal{E}[T_i]$. We now have two possibilities:

1. $\Gamma \vdash T_1 \lor T_2$ type. Then, by Lemma C.6, we have two cases:
   a. for some $i \in \{1, 2\}$, $\Gamma \vdash t_0 : T_i$. This means that $\Gamma \vdash t_0 : T_1 \lor T_2$ holds by an instance of $[\vee \leftarrow \in \mathcal{E}[T_i]]$. Then, by letting $T' = \mathcal{E}[T_i]$, we get $\Gamma \vdash t \overline{\text{t}} \rightarrow t''$ and $\Gamma \vdash t : T''$: hence, we conclude by obtaining item (a);
   b. for all $i \in \{1, 2\}, \Gamma \not\vdash t_0 : T_i$. Then, either:
      (a) the reducing $\vee$-type is introduced by a subterm of $t_0$ of the form if $t'_0$ then $t'_0$ else $t'_0$, possibly combined with instances of if ... then ... else ..., let ... = ... in ... or function application. Thus, $t_0$ can reduce as:
          $$\Gamma \vdash t_0 \xrightarrow{\alpha} t'' \quad \text{with} \quad \alpha \in \{\tau[-x], \tau[\text{if } x], \tau[\text{let } x], \tau[\lambda], \tau[x] \mid x \in \mathbb{X}, \alpha \neq \tau[\text{Coun}]\}$$
          and thus, by $\text{[SR-E]}$, we also have $\Gamma \vdash t \xrightarrow{\tau} \mathcal{P}[t''];$ moreover, by Thm. 4.4, we have $\Gamma \vdash t'' : T_1 \lor T_2$, which implies $\Gamma \vdash \mathcal{P}[t''] = T$. Therefore, letting $t' = \mathcal{P}[t'']$, we get $\Gamma \vdash t \xrightarrow{\alpha} t' \quad \Gamma \vdash t' : T$, and conclude by obtaining item (b);
      (b) the reducing $\vee$-type is not introduced by a subterm of $t_0$ of the form if $t'_0$ then $t'_0$ else $t'_0$. Then, it must be due to some variable $z$ of type $T'_0 \lor T'_2$ that might occur either:
         • within some instances of if ... then ... else ..., let ... = ... in ... or function application. Then, $t_0$ can reduce as $\Gamma \vdash t_0 \xrightarrow{\alpha} t''$ similarly to case (a), we conclude by obtaining item (b);
         • directly as $z$, hence $t = \mathcal{P}[z]$. Then, $t$ has a top-level send/recv term (possibly within $\mathcal{P}$), and correspondingly, by Lemma C.7, $\mathcal{E}$ has a top-level $\alpha$-term (possibly within $\mathcal{P}$). Therefore, $T$ has an enabled transition for input/output/interaction with a label $\alpha \neq \tau[\vee]$; hence, we conclude by obtaining item (c);
2. $\Gamma \vdash T_1 \lor T_2$ $\pi$-type. Then, we have $\Gamma \vdash T_1, T_2$ $\pi$-type, and using Lemma C.7, we find two cases, corresponding to either case 1a or 1b(a) above and conclude similarly.

□
Verifying Message-Passing Programs with Dependent Behavioural Types (tech. report)

F Proof of Lemma 4.7

We prove the thesis in three steps:

1. we develop a calculus akin to CCS [53], but without restrictions nor relabeling, called \( CCST \). Its syntax is based on our \( \pi \)-types, and its labelled semantics match Def. 4.2. We also show that our \( \pi \)-types are encodable in \( CCST \) (§F.1);
2. since \( CCST \) has no name restriction nor relabeling, we show that it can be encoded into Petri nets with a minor variation of [22, §4.1] (§F.2);
3. from this, it follows that linear-time \( \mu \)-calculus formulas are decidable for \( CCST \) terms, and thus, for our types (§F.3).

F.1 Encoding of \( \pi \)-types into \( CCST \)

Definition F.1. \( CCST \) terms have the following syntax:

\[
\mathcal{T}, \mathcal{U} ::= \mathcal{S}(\mathcal{T}).\mathcal{T} \mid \sum_{i \in \mathcal{T}} S_i(\mathcal{T}_i).\mathcal{T}_i \mid \mathcal{T} \parallel \mathcal{U} \mid \mathcal{T} \lor \mathcal{U} \mid \mathcal{T} + \mathcal{U} \mid \mu t.\mathcal{T} \mid t \mid \text{nil}
\]

The congruence \( \equiv \) between \( CCST \) terms is defined as:

\[
\mathcal{T} \parallel \mathcal{U} \equiv \mathcal{U} \parallel \mathcal{T} \quad (\mathcal{T}_1 \parallel \mathcal{T}_2) \parallel \mathcal{T}_3 \equiv \mathcal{T}_1 \parallel (\mathcal{T}_2 \parallel \mathcal{T}_3) \quad \mathcal{T} \parallel \text{nil} \equiv \mathcal{T} \quad \mu t.\mathcal{T} \equiv \mathcal{T}\{\mu t.\mathcal{T}/t\}
\]

Given a typing environment \( \Gamma \), the semantics of \( CCST \) terms is defined as:

\[
\mathcal{T} \overset{\alpha}{\to} \mathcal{T}' \quad \text{for some } i \in \{1, 2\} \quad \mathcal{T} \parallel \mathcal{U} \overset{\mathcal{S}(\mathcal{T})}{\to} \mathcal{U}' \quad \Gamma \vdash S \land S' \quad \Gamma \vdash T \quad T \not\in \mathcal{X}
\]

\[
\mathcal{T} \overset{\mathcal{T} \parallel \mathcal{U} \overset{\mathcal{S}(\mathcal{T})}{\to} \mathcal{U}' \quad \Gamma \vdash S \land S' \quad \Gamma \vdash T \leq \mathcal{T}' \quad T \not\in \mathcal{X}
\]

\[
\mathcal{U} \overset{\alpha}{\to} \mathcal{U}' \quad \mathcal{T} \parallel \mathcal{U} \overset{\mathcal{T} \parallel \mathcal{U} \overset{\mathcal{S}(\mathcal{T})}{\to} \mathcal{U}' \quad \Gamma \vdash S \land S' \quad \Gamma \vdash T \leq \mathcal{T}' \quad T \not\in \mathcal{X}
\]

\[
\mathcal{T} \overset{\mathcal{T} \parallel \mathcal{U} \overset{\mathcal{S}(\mathcal{T})}{\to} \mathcal{U}' \quad \Gamma \vdash S \land S' \quad \Gamma \vdash T \leq \mathcal{T}' \quad T \not\in \mathcal{X}
\]

Definition F.2. We write \( \Gamma \vdash T_1 \leq \Gamma T_2 \) if \( \Gamma \vdash T_1 \leq T_2 \) and \( \Gamma \vdash T_2 \leq T_1 \).

We can now define an encoding of our \( \pi \)-types into \( CCST \) (Def. F.5 below). The encoding is straightforward, except for one detail:

\((*)\) given \( \Gamma \), by Def. 4.2 the \( \pi \)-type \( \mathcal{T}_o = o[S \lor S', T, \Pi(U)] \) can reduce as follows:

\[
\Gamma \vdash o[S \lor S', T, \Pi(U)] \overset{\mathcal{S}(\mathcal{T})}{\to} U
\]

where the second and third transition are due to the contextual rule, allowing to reduce \( \lor \)-types inside \( o \)-types;

To simplify our encoding of \( \pi \)-types in \( CCST \), it is convenient to remove the transitions of \( \lor / \mu \)-types inside \( o \)-types; to this purpose, we expand a type, with the rewriting outlined below:

\((**\ast)\) we bring \( \lor \)-types at the top-level, removing the need of expanding them inside \( o \)-types. For this purpose, we introduce an "expanded or" type "+" defined exactly like \( \lor \), except that its semantic rules are:

\[
\Gamma \vdash T \overset{\alpha}{\to} T' \quad \Gamma \vdash U \overset{\alpha}{\to} U'
\]

\[
\Gamma \vdash T \overset{\alpha}{\to} T' \quad \Gamma \vdash U \overset{\alpha}{\to} U'
\]

i.e., + does not introduce a \( \mathcal{T}(\mathcal{T}) \) transition when choosing one of its two options.

Then, the expansion of type \( \mathcal{T}_o = o[S \lor S', T, \Pi(U)] \) above is the type:

\[
\mathcal{T}_o' = o[S \lor S', T, \Pi(U)] + (o[S, T, \Pi(U)] \lor o[S', T, \Pi(U)]
\]
We write $\pi$ in §F.2. Moreover, the encoding in §F.2 yields Petri nets that are strongly bisimilar to their originating $\mu$-CCS terms, as formalised in Prop. F.4 below.

Definition F.3 (Type bisimulation). We say that a relation $R_F$ between valid $\pi$-types in $\Gamma$ is a type bisimulation iff, whenever $(U_1, U_2) \in R_F$:

1. $\Gamma \vdash U_1 \xrightarrow{S(T)} U_1'$ implies $\exists S', T', U_2': \Gamma \vdash S \leq S', \Gamma \vdash T \leq T', \Gamma \vdash U_2 \xrightarrow{S(T')} U_2'$ and $(U_1', U_2') \in R_F$;
2. $\Gamma \vdash U_1 \xrightarrow{S(T)} U_1'$ implies $\exists S', T', U_2': \Gamma \vdash S \leq S', \Gamma \vdash T \leq T', \Gamma \vdash U_2 \xrightarrow{S(T')} U_2'$ and $(U_1', U_2') \in R_F$;
3. $\Gamma \vdash U_1 \xrightarrow{r[S,T]} U_1'$ implies $\exists S', T', U_2': \Gamma \vdash S \leq S', \Gamma \vdash T \leq T', \Gamma \vdash U_2 \xrightarrow{r[S',T']} U_2'$ and $(U_1', U_2') \in R_F$;
4. $\Gamma \vdash U_1 \xrightarrow{r[v]} U_1'$ implies $\exists U_2': \Gamma \vdash U_2 \xrightarrow{r[v]} U_2'$ and $(U_1', U_2') \in R_F$;
5. the converse of clauses 1–4, on the transitions emanating from $U_2$.

We write $\Gamma \vdash U_1 \sim U_2$ iff, for some type bisimulation $R_F$, we have $U_1 R_F U_2$.

Proposition F.4. For all $\Gamma, T$ such that $\Gamma \vdash T \pi$-type, $\Gamma \vdash T \sim \exp(T)$.

Definition F.5 ($CCS^T$ encoding of $\pi$-types). For all $\Gamma, T$ such that $\Gamma \vdash T \pi$-type, the $CCS^T$ encoding of $T$ in $\Gamma$ is defined as $\exp(T)_\Gamma$, where:

- $\exp(\text{nil})_\Gamma = \text{nil}$
- $\exp(\mu t.T)_\Gamma = \mu t.\exp(T)_\Gamma$
- $\exp(t)_\Gamma = t$
- $\exp(T \lor U)_\Gamma = \exp(T)_\Gamma \lor \exp(U)_\Gamma$
- $\exp(T \land U)_\Gamma = \exp(T)_\Gamma \land \exp(U)_\Gamma$
- $\exp(p[U,T])_\Gamma = \exp(T)_\Gamma \land \exp(U)_\Gamma$
- $\exp(o[S,T,\Pi(U)]_\Gamma = \exp(S(T)_\Gamma) \land \exp(U)_\Gamma$
- $\exp(\Pi(T\{x\to U\})_\Gamma = \sum_{T' \in \mathcal{Y}} \exp(S(T')) \land \exp(U/T')_\Gamma$

where $\mathcal{Y}$ contains all possible payload types $T'$, according to the premises of rule $[T \rightarrow t]$. As discussed in §B.2, the set $\mathcal{Y}$ is always finite, hence the summation has a finite number of branches.

Proposition F.6. For all $\Gamma, T$ such that $\Gamma \vdash T \pi$-type, $\Gamma \vdash T \sim \exp(T)_\Gamma$.

Proof. By Def. F.5, we can verify that $\Gamma \vdash \exp(T) \sim \exp(T)_\Gamma$, where the judgement stands for strong bisimilarity, and is defined as expected. Then, we conclude by Prop. F.4.

F.2 Encoding of $CCS^T$ into Petri nets

Following Def. F.1, we can encode $CCS^T$ terms into a Petri net with a minor variation of the encoding in [22, §4.1]. The key restrictions for such an encoding is that it only applies to finite-branching and guarded $CCS^T$ terms (i.e., in a recursive term $\mu t.T$, the recursion variable $t$ can only appear in $T$ as subterm of $S(U).T'$ or $S(U).T'$). By Def. F.5, both restrictions are satisfied by $CCS^T$ terms obtained by encoding guarded $\pi$-types (hence the requirement in Lemma 4.7).

Besides this, the only differences w.r.t. [22, §4.1] are that:

1. in $CCS^T$ we have two kinds of internal transitions: $r[v]$ and $r[S,S']$ — whereas in CCS, only one $r$-transition covers all cases. Such different internal transitions must be kept distinguished in the labels of the encoded Petri net;
2. to generate a synchronisation label $r[S,S']$ in the encoded Petri net, we must apply the (decidable) checks of the corresponding semantic rules in Def. F.1, which include subtyping-based comparisons — whereas CCS uses simpler duality checks (e.g., the CCS label $a$ only synchronises with $\overline{a}$, and vice versa).

F.3 Decidability of linear-time $\mu$-calculus judgements

By [20, §3], linear-time $\mu$-calculus judgements are decidable on Petri nets — and this includes those generated with the encoding in §F.2. Moreover, the encoding in §F.2 yields Petri nets that are strongly bisimilar to their originating $CCS^T$ terms, which in turn are strongly bisimilar to their originating $\pi$-types (Prop. F.6). Therefore, if we have a $\pi$-type $T$, and a linear-time $\mu$-calculus formula $\phi$, we obtain that $\phi$ holds for $T$’s Petri net if and only if it holds for $T$; and since we can decide whether $\phi$ holds for $T$’s Petri net, we have obtained a decision procedure of $\phi$ for $T$. 
G Process verification via type verification

G.1 Basic properties

Lemma G.1. If \( \Gamma \vdash T \xrightarrow{a} T' \), then \( \mathbb{B}_T(T') \subseteq \mathbb{B}_T(T) \).

Proof. By induction on the derivation of the transition \( \Gamma \vdash T \xrightarrow{a} T' \).

Corollary G.2. If \( \Gamma \vdash T \xrightarrow{a} T' \), then \( \mathcal{A}_T(T') \subseteq \mathcal{A}_T(T) \).

Proof. Direct consequence of Lemma G.1 and Def. B.4.

Corollary G.3. If \( \Gamma \vdash T \xrightarrow{a} T' \), then for all \( S, \mathcal{U}_{\Gamma,T}(S) \subseteq \mathcal{U}_{\Gamma,T}(S) \) and \( \mathcal{U}_{\Gamma,T}(S) \subseteq \mathcal{U}_{\Gamma,T}(S) \).

Proof. Direct consequence of Cor. G.2 and Def. B.4.

G.2 Proof of Theorem 4.10

Definition G.4. Assume \( \Gamma \vdash t : T \), with \( \Gamma \vdash T \pi\text{-type} \). The set of actions \( \text{of} \ t \) in \( \Gamma \) is:

\[ \mathcal{A}_T(t) = \{ \beta | \Gamma \vdash t \xrightarrow{\alpha_1 \ldots \alpha_n} \beta \} \]

Theorem 4.10. Within productive \( \lambda^T \), assume \( \Gamma \vdash t : T \), with \( \Gamma \vdash T \pi\text{-type} \), \( \text{proc} \not\in T \). Also assume, for all \( i[S, \Pi (x) U'] \) occurring in \( T \), that there is \( y \) such that \( \Gamma \vdash y : U \) holds. For \( \mu \)-calculus judgements on \( T \), let \( \mathcal{A}_T = \mathcal{A}_{\mathcal{T}(T)} \), and \( \mathcal{A}_T = \{ \tau[S,S'] \in \mathcal{A}_T(T) | \{ S, S' \} \not\subseteq \text{dom}(\Gamma) \} \). Then, the implications in Fig. 7 hold.

Proof. We develop some interesting cases in Fig. 7 (the remaining ones are similar).

**Item (1) (non-usage).** Let \( \phi = \square \neg \{ \bigvee \{ i \in 1..n \} \mathcal{U}_{\Gamma,T}(x_i) \} \).

By Def. B.3 and Def. B.2, the denotation of \( \phi \) is:

\[
\| \phi \|_0 = \bigcup \left\{ W \subseteq \mathcal{A}_T(T) \mid \begin{array}{l}
W \subseteq \left( \{ \sigma \in \mathcal{A}_T(T) | \text{hd}(\sigma) \in \bigcup \{ \mathcal{U}_{\Gamma,T}(x_i) \} \} \right) \\
\cap \left( \{ \sigma \in \mathcal{A}_T(T) | \text{tl}(\sigma) \in \bigcup \{ \mathcal{U}_{\Gamma,T}(x_i) \} \} \right)
\end{array} \right\}
\]

and therefore, for all finite or infinite words \( \alpha_1 \alpha_2 \ldots \in \mathcal{A} \),

\[
\alpha_1 \alpha_2 \ldots \in \| \phi \|_0 \quad \text{iff} \quad \forall j \in 1,2,\ldots: \alpha_j \not\in \bigcup \{ \mathcal{U}_{\Gamma,T}(x_i) \}
\]

which implies that, by the hypothesis \( T \vdash \{ x_i \}_{i \in 1..n} \models \phi \) and Def. B.2,

\[
T \vdash \{ x_i \}_{i \in 1..n} \xrightarrow{\alpha_1 \alpha_2 \ldots} \quad \text{implies} \quad \forall j \in 1,2,\ldots: \alpha_j \not\in \bigcup \{ \mathcal{U}_{\Gamma,T}(x_i) \}
\]

Now, taking any \( t \) such that \( \Gamma \vdash t : T \), we prove that:

\[
t \vdash \{ x_i \}_{i \in 1..n} \xrightarrow{\beta_1 \beta_2 \ldots} \quad \text{implies} \quad \forall j \in 1,2,\ldots: \beta_j \not\in \bigcup \{ \mathcal{U}_{\Gamma,T}(x_i) \}
\]

We proceed by contradiction. Assume that (4) is false, i.e., that \( \exists k \in 1,2,\ldots \) such that:

\[
t \vdash \{ x_i \}_{i \in 1..n} \xrightarrow{\beta_1 \beta_2 \ldots \beta_k} \quad \text{implies} \quad \forall j \in 1,2,\ldots: \beta_j \not\in \bigcup \{ \mathcal{U}_{\Gamma,T}(x_i) \}
\]

where

\[
\beta_k = \pi(w) \quad \text{for some} \quad y \in \{ x_i \}_{i \in 1..n} \quad \text{and} \quad w \in \mathcal{U}_{\Gamma,T}(x_i)
\]

This implicitly requires \( \Gamma \vdash U \pi\text{-type} \), hence \( \text{fv}(U) \cap \text{bv}(T) = \emptyset \): 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 \( y \not\in T \), we can pick \( y \not\in \text{dom}(\Gamma) \), extend \( \Gamma \) as \( \Gamma' = \Gamma, y' : U \), and get \( \Gamma \vdash y' : U \) and \( \Gamma \vdash t : T \).
Then, letting \( t_0 = t \) and \( T_0 = T \), by induction on \( k \) (using Thm. 4.4) we can prove:

\[
\exists T_1, \ldots, T_k : \\
\forall l \in 0..k - 1 : T_l = T_{l+1} \\
\forall l \in 0..k - 1 : \exists a'_{l+1} : \Gamma \vdash T_l \rightarrow a'_{l+1} T_{l+1} \quad \text{if } \beta_{l+1} = \tau[x] \text{ with } [x] \neq [R:\text{Cons}], \text{ or } \beta_{l+1} = \tau[x()] \\
\text{ otherwise}
\] (7)

and in particular, from the definition of \( \beta_k \) in (6), we have:

\[
\exists S, U : \Gamma \vdash y : S, w : U \quad \text{and} \quad a'_k = \bar{S}(U) \\
\Gamma \vdash y \subseteq S \\
a'_k \in \mathbb{U}^0_{T_0}(y) \\
a'_k \in \mathbb{U}^0_{T}(y) \\
\text{by (5), (6) and Thm. 4.4} \quad (8)
\]

(9)

(10)

Hence, summing up:

\[
\exists a''', a''''\ldots, a''' : \\
\quad a''' = a'_k \\
\quad T \models \{ x_1 \}_{i \in 1..n} \quad \text{by (11), (7) and (6)}
\]

but this contradicts (3), and thus, the hypothesis \( T \models \{ x_1 \}_{i \in 1..n} \models \phi \). Therefore, (5)/(6) must be false, and we obtain that (4) holds.

Now, by (4), we have that all the runs of \( t \models \{ x_i \}_{i \in 1..n} \) belong to:

\[
\bigcup \left\{ \mathcal{W} \subseteq A_T(t) \bigg| \mathcal{W} \subseteq \left\{ \sigma \bigg| \sigma = \epsilon \text{ or } \text{hd}(\sigma) \notin \bigcup_{i \in 1..n} \{ \overline{\text{U}}(w) \mid w \in \mathcal{W} \} \right\} \text{ and } \text{tl}(\sigma) \notin \mathcal{W} \right\} \\
= \bigcup \left\{ \mathcal{W} \subseteq A_T(t) \bigg| \mathcal{W} \subseteq \left\{ \sigma \bigg| \sigma \in A_T(t) \text{ and } \text{hd}(\sigma) \notin \bigcup_{i \in 1..n} \{ \overline{\text{U}}(w) \mid w \in \mathcal{W} \} \right\} \right\} \\
= \| \neg (\forall \in 1..n (\overline{\text{U}}(w) \cap T)) \|_0
\]

(12)

(13)

(14)

where we get the equality from (12) to (13) through a series of rewritings similar to the ones in (2) above (in reverse order), and the equality from (13) to (14) by Def. B.2. Therefore, by (14) and Def. B.2, we conclude \( t \models \{ x_i \}_{i \in 1..n} \models \phi \).

**Item (3) (eventual usage).** Let \( \phi = (\neg A_T) \cup \bigcup_{i \in 1..n} \{ \overline{\text{U}}(U') \mid \text{any } U' \cap T \} \). By Def. B.3 and Def. B.2, the denotation of \( \phi \) is:

\[
\| \phi \|_0 = A_T(t) \setminus \bigcup \left\{ \mathcal{W} \subseteq A_T(t) \bigg| \mathcal{W} \subseteq \left\{ \sigma \bigg| \sigma \in A_T(t) \setminus \bigcup_{i \in 1..n} \{ \overline{\text{U}}(U') \mid \text{hd}(\sigma) = \overline{\text{U}}(U) \} \right\} \right\} \\
= A_T(t) \setminus \bigcup \left\{ \mathcal{W} \subseteq A_T(t) \bigg| \mathcal{W} \subseteq \left\{ \sigma \bigg| \sigma \in A_T(t) \setminus \bigcup_{i \in 1..n} \{ \overline{\text{U}}(U') \mid \text{hd}(\sigma) = \overline{\text{U}}(U) \} \right\} \right\} \\
= A_T(t) \setminus \bigcup \left\{ \mathcal{W} \subseteq A_T(t) \bigg| \mathcal{W} \subseteq \left\{ \sigma \bigg| \sigma \in A_T(t) \setminus \bigcup_{i \in 1..n} \{ \overline{\text{U}}(U') \mid \text{hd}(\sigma) = \overline{\text{U}}(U) \} \right\} \right\}
\]
Verifying Message-Passing Programs with Dependent Behavioural Types (tech. report)

First, observe that for any run

\[ (*) \]

We proceed by contradiction. Assume that \((*)\) is false, i.e., that there is a run of

\[ (\sigma, \tau) \]

such that

\[ \sigma \text{ is a run of } T \]

and

\[ \tau \text{ is a run of } T' \]

Then, by (17) we have two possibilities:

1. \( t \models \{ x_i \}_{i \in 1..n} \]

2. \( t \models \{ x_i \}_{i \in 1..n} \]

\[ \sigma \text{ does not contain any } x_i \]

\[ \tau \text{ does not contain any } x_i \]

\[ \text{and } \forall i \in 1..n, \exists w \in \mathbb{V} \cup \mathbb{X} \]

\[ x_i \text{ never fires } \]

\[ x_i \text{ can perform further actions} \]

\[ \text{and since it} \]

\[ \text{does not belong to } \parallel \phi \parallel_\emptyset \]

\[ \text{we obtain the contradiction } \]

\[ t \models \{ x_i \}_{i \in 1..n} \neq \phi \]
where we get the equality from (19) to (20) through a series of rewritings similar to the ones in (15) above (in reverse order),

Finally, we show that all runs of \( t \) that does not contain \( \overline{\tau}(w) \) (for any \( i \in 1..n \) and \( w \in \mathcal{V} \cup \mathcal{X} \)). Since \( t \) is productive (by hypothesis), \( \sigma_t \) cannot contain an infinite sequence of transitions only using labels of the form \( \tau[s] \) (with \([s] \neq [\text{R-Cons}]\)), \( \tau[\neg x] \), \( \tau[\text{if} \, x \, \text{then} \, \tau[x] \, \text{else} \, \tau[]] \) (for some \( x \)), or \( \tau\{ \text{run} \} \). Therefore, \( \sigma_t \) contains an infinite number of actions of the form \( \tau[w] \) (for some \( x \) and \( w \in \mathcal{V} \cup \mathcal{X} \)), \( \tau[x] \) (for some \( x \)), or \( \tau[\text{R-Cons}] \); hence, by Thm. 4.4 (subject reduction), the sequence of type actions \( \sigma_T \) built from \( \sigma_t \) is also infinite; therefore, \( \sigma_T \) is an (infinite) run of \( t \) that does not contain any \( \overline{\tau}(U) \), hence does not belong to \( \mathcal{V}_\phi \) — i.e., we obtain the contradiction \( T \vdash \{ x \}_{i \in 1..n} \not\models \phi \).

Summing up: if we assume (18), we contradict the hypothesis \( T \vdash \{ x \}_{i \in 1..n} \models \phi \). Therefore, (18) must be false, hence (17) holds.

Now, by (17), we have that all the runs of \( t \) that belong to:

\[
A_T(t)^{\omega} \cap \bigcup \mathcal{W} \subseteq \left\{ \sigma \mid \begin{array}{l}
\text{hd}(\sigma) \not\in \{ \overline{\tau}(w) \mid i \in 1..n, \ w \in \mathcal{V} \cup \mathcal{X} \} \text{ and } tl(\sigma) \in \mathcal{W} \end{array} \right\} \tag{19}
\]

\[
= A_T(t)^{\omega} \cap \bigcup \mathcal{W} \subseteq \left\{ \sigma \mid \begin{array}{l}
\text{hd}(\sigma) \not\in \{ A_T(t)^{\omega} \cap \bigcup \mathcal{W} \mid i \in 1..n, \ w \in \mathcal{V} \cup \mathcal{X} \} \text{ and } tl(\sigma) \in \mathcal{W} \end{array} \right\} \tag{20}
\]

\[
= || \mathcal{V} \cup \{ \overline{\tau}(w) \mid w \in \mathcal{V} \cup \mathcal{X} \} ||_{\phi} \tag{21}
\]

\[
= || \bigl( \mathcal{V} \cup \{ \overline{\tau}(w) \mid w \in \mathcal{V} \cup \mathcal{X} \} \bigl) \bigr) ||_{\phi} \tag{22}
\]

where we get the equality from (19) to (20) through a series of rewritings similar to the ones in (15) above (in reverse order), the equality from (20) to (21) by Def. B.2, and finally we obtain (22) by Def. B.3. Therefore, by (22) and Def. B.2, we conclude \( t \models \{ x \}_{i \in 1..n} \not\models \phi \).

Item (4) (forwarding). Letting \( \phi = \square\left( \left( S(z) \land S(z) \in U_{T,T}(x) \right) \right) \Rightarrow \left( \left( -A_T \cup U_{T,T}(x) \right) \cup \overline{\tau}(z) \right) \) \), the proof follows the same strategy of the proofs for items (1) and (3): use the shape of all runs of \( T \vdash \{ x, y, z \} \) (belonging to \( \mathcal{V}_\phi \)) to determine the shape of the runs of \( t \vdash \{ x, y, z \} \)

- first, we use a strategy similar to item (1) to show that any action \( x(z) \) in a run of \( t \vdash \{ x, y, z \} \) is matched by an action \( a \) in the run of \( T \vdash \{ x, y, z \} \) such that \( a \in \{ S(z) \mid S(z) \in U_{T,T}(x) \} \);
- then, we use a strategy similar to item (3) to show that the action \( x(z) \) above must be followed by \( \overline{\tau}(z) \), that is the only term action that can correspond to the type action \( \overline{\tau}(z) \).

Finally, we show that all runs of \( t \vdash \{ x, y, z \} \) belong to the denotation

\[
\square((x(z)) \Rightarrow ((\neg x(w)) \cup \overline{\tau}(z)))) \not\models_\phi
\]

from which, by Def. B.2, we conclude \( t \vdash \{ x, y, z \} \models \square((x(z)) \Rightarrow ((\neg x(w)) \cup \overline{\tau}(z)))) \).