# Dependent Function Types for Higher-Order Interaction

**Alceste Scalas**
Imperial College London, UK

**Nobuko Yoshida**
Imperial College London, UK

## Abstract

We develop a typing discipline for higher-order programs that interact by exchanging values, channels, and mobile code. The key ingredients are channel and process types, subtyping, and a form of *dependent function types* where parameter variables can appear in return types. The main novelty is that, by combining these ingredients, we can accurately describe the behaviour of typed programs. The main result is that we can statically decide safety and liveness properties of programs, by inspecting the labelled transition system of types. Moreover, the key ingredients above are all available (or can be defined) in the type system of the Scala/Dotty programming language: we show that this leads to a direct embedding and implementation of our theory.

**Keywords and phrases** dependent types, behaviroual types, higher-order $\pi$-calculus, temporal logic

## 1 Introduction

Concurrent and distributed applications are widespread, and their correct design is often elusive. Concurrency theory has addressed this challenge by developing various *models* for concurrent and distributed systems, and various forms of *analysis* to ensure that a model enjoys run-time *safety* or *liveness* properties — guaranteeing that unwanted events will never occur, or that desired events will eventually occur [17].

These properties are more difficult to check when dealing with *higher-order* concurrent programs that can exchange *communication channels* and *mobile code*. These features have practical uses in modern applications, and are made available in programming languages and software development toolkits [18, 9, 20]. As a simple example, consider a data analysis server that allows its clients to send custom code for on-the-fly data processing and filtering. The intended behaviour of the client code is:

1. *take the channels "$i_1$", "$i_2$", and "$o$" provided by the server;*
2. *read an integer value $x$ from channel "$i_1$";*
3. *read an integer value $y$ from channel $i_2$";* (∗)
4. *choose one between $x$ and $y$, and forward it through channel "$o$";*
5. *continue from point 2.*

The server, in the notation of the higher-order $\pi$-calculus (HO$\pi$) [23], might look like:

$$(\nu\, i_1, i_2, o)\big(\, cm?(p).p[i_1, i_2, o] \parallel \cdots \big) \tag{1}$$

i.e., the server has three internal channels $i_1, i_2, o$ (restricted by $\nu$), and a client-facing channel $cm$. The server uses $cm$ to receive some mobile code $p$, and then runs $p$ by passing $i_1, i_2, o$ as arguments; thus, $p$ can use such channels to interact with the rest of the system (omitted with "…"). To avoid communication errors, the code in (1) can be checked against *simple channel types*: e.g., if we type-check the server under type assignments $i_1{:}\mathrm{c}^{\mathrm{io}}[\mathrm{int}]$, $i_2{:}\mathrm{c}^{\mathrm{io}}[\mathrm{int}]$, $o{:}\mathrm{c}^{\mathrm{io}}[\mathrm{int}]$, then we ensure that $i_1$, $i_2$ and $o$ have *channel types*, and are only used to input/output integers. The type of $cm$, instead, has the form $\mathrm{c}^{\mathrm{i}}[T]$: a channel allowing to input (*not*

output) values of type $T$, where $T$ is an *abstract process type*, and is also the type of $p$. The form of $T$ depends on how process terms are typed, and tells "how much we know" about the unknown mobile code $p$. In previous works, $T$ can have forms like:

$$T' \;=\; \mathrm{c}^{\mathrm{i}}[\mathrm{int}] \to \mathrm{c}^{\mathrm{o}}[\mathrm{int}] \to \mathrm{c}^{\mathrm{o}}[\mathrm{int}] \to \mathbf{proc} \tag{2}$$

$$T'' \;=\; \Pi(i_1{:}\mathrm{c}^{\mathrm{i}}[\mathrm{int}])\,\Pi(i_2{:}\mathrm{c}^{\mathrm{o}}[\mathrm{int}])\,\Pi(o{:}\mathrm{c}^{\mathrm{o}}[\mathrm{int}])\,\langle i_1{:}\mathrm{c}^{\mathrm{i}}[\mathrm{int}], i_2{:}\mathrm{c}^{\mathrm{i}}[\mathrm{int}], o{:}\mathrm{c}^{\mathrm{o}}[\mathrm{int}]\rangle \tag{3}$$

The type (2) says that $p$ is a function that takes three typed channels, and returns a term of type **proc** — which is inhabited by all type-safe process [23]. The type (3) (proposed in [33, 34, 32]) is a form of *dependent function types*: it says that $p$ takes three named arguments $i_1, i_2, o$ (with the given channel types), and returns a type-safe process term of type $\langle i_1{:}..., i_2{:}...\rangle$: it is inhabited by processes that can use the listed channels, *only*. Clearly, (3) is more informative than (2): it statically limits the resources that $p$ *could* access at run-time. However, (2) and (3) cannot ensure the intended behaviour of $p$ described in $(*)$ above: e.g., they include processes that terminate instead of looping, or write random integers on $o$ instead of forwarding those received from $i_1/i_2$, or discard received values, or spawn a large number of threads (*forkbombs*). Symmetrically, the client-side programmer who writes the actual code of $p$ has no guidance, besides the informal description in $(*)$: more informative types can help both the server in enforcing its intended policies, and the programmer in producing the intended implementation.

**Proposal** We develop a type discipline for higher-order concurrent programs, that can enforce detailed run-time behaviours. For example, we can give to $p$ in (1) a type like:

$$T_m \;=\; \Pi(i_1{:}\mathrm{c}^{\mathrm{i}}[\mathrm{int}])\,\Pi(i_2{:}\mathrm{c}^{\mathrm{i}}[\mathrm{int}])\,\Pi(o{:}\mathrm{c}^{\mathrm{o}}[\mathrm{int}])\,\mu\mathbf{t}.\,\mathbf{i}[i_2, \Pi(x{:}\mathrm{int})\mathbf{i}[i_2, \Pi(y{:}\mathrm{int})\mathbf{o}[o, (x \vee y), \mathbf{t}]\,]\,]$$

that accurately formalises the description in $(*)$: $\mathbf{i}[i_1, \Pi(x{:}\mathrm{int})T]$ is the type of a process that inputs $x$ from channel $i_1$ and continues as $T$, where $x$ can occur as reference to the received value; $\vee$ is the union type, and denotes a choice; and $\mathbf{o}[o, U, \mathbf{t}]$ is the type of a process that outputs on $o$ a value of type $U$ and continues by looping on the recursion variable $\mathbf{t}$.

Our theory leverages: *(i) subtyping*, that we crucially exploit to type internal communication and restricted channels; and *(ii)* a form of dependent function type $\Pi(z{:}U)T$, where the return type $T$ can refer to the parameter $z$ (of type $U$). We develop our theory by formalising a higher-order, concurrent functional calculus, called $\lambda^{\pi\mathsf{D}}_{\leqslant}$; moreover, we take both *(i)* and *(ii)* above from a fragment of the Scala/Dotty programming language [1]. The payoffs of these choices are that (1) $\lambda^{\pi\mathsf{D}}_{\leqslant}$ consistently and naturally uses abstract processes and dependent functions, for both local and mobile code; and (2) $\lambda^{\pi\mathsf{D}}_{\leqslant}$ is remarkably close to an implementation language, easily embedded in Scala: see Fig. 1. Yet, $\lambda^{\pi\mathsf{D}}_{\leqslant}$ supports a simple encoding of HO$\pi$: hence, our theory is also applicable in its canonical setting.

The fine-grained type information shown in $T_m$ above means that we can use *types as behavioural models* [14, 7], by equipping them with a semantics that (over-)approximates the run-time behaviour of programs. The key novelty is that, by using dependent functions types, we can precisely track how values and channels are exchanged.

**Contributions and outline** We present the $\lambda^{\pi\mathsf{D}}_{\leqslant}$ calculus in §2. Its type system is illustrated in §3, where we establish basic type safety (Thm. 11). In §4, we equip our types with a labelled transition system (LTS) semantics, and, with Thm. 14 and Thm. 15, we show the correspondence between the transitions of types and processes. Then, we use such theorems to transfer linear-time $\mu$-calculus judgements from types (where they are decidable, by Lemma 17) to $\lambda^{\pi\mathsf{D}}_{\leqslant}$ processes; as a result, we obtain a method to verify safety and liveness properties of higher-order programs (Thm. 19).

*Due to space limits, proofs and additional materials are in the on-line technical report [25].*

$$\begin{array}{l}\textbf{let } f = \lambda i_1.\lambda i_2.\lambda o.(\\ \quad \textbf{recv}(i_1, \lambda x.(\\ \quad\quad \textbf{recv}(i_2, \lambda y.(\\ \quad\quad\quad \textbf{if } x+y<42 \textbf{ then}\\ \quad\quad\quad\quad \textbf{send}(o, x, \lambda\_.\\ \quad\quad\quad\quad\quad f\, i_1\, i_2\, o)\\ \quad\quad\quad \textbf{else}\\ \quad\quad\quad\quad \textbf{send}(o, y, \lambda\_.\\ \quad\quad\quad\quad\quad f\, i_1\, i_2\, o)\ )))\end{array}$$

$$\begin{array}{l}\textbf{let } f = \lambda i_1.\lambda i_2.\lambda o.(\\ \quad i_1?(x).(\\ \quad\quad i_2?(y).(\\ \quad\quad\quad \textbf{if } x+y<42 \textbf{ then}\\ \quad\quad\quad\quad o!\langle x\rangle.(\\ \quad\quad\quad\quad\quad f\, i_1\, i_2\, o)\\ \quad\quad\quad \textbf{else}\\ \quad\quad\quad\quad o!\langle y\rangle.(\\ \quad\quad\quad\quad\quad f\, i_1\, i_2\, o)\ )))\end{array}$$

```
def f(i1, i2, o) = {
  recv(i1) { x =>
    recv(i2) { y =>
      if (x+y < 42) {
        send(o, x) {
          f(in, out, log) }
      } else {
        send(o, y) {
          f(in, out, log) }}}}}
```

**Figure 1** An implemention of the specification $(*)$ in §1, in three flavours: a $\lambda^{\pi\mathsf{D}}_{\leqslant}$ program typed by $T_m$ in §1 (left); its higher-order $\pi$-calculus version (middle); and the equivalent embedded Scala/Dotty syntax (right). The latter uses syntactic sugar to partially hide the $\lambda$-terms explicitly visible in the $\lambda^{\pi\mathsf{D}}_{\leqslant}$ code: e.g., `recv(i1){x=>...}` is desugared into `recv(i1,λ(x)=>...)`.

$$\begin{array}{llll}\mathbb{B} = \{\mathbf{tt}, \mathbf{ff}\} & \mathbb{C} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots\} & \mathbb{X} = \{x, y, z, \ldots\} & \mathbb{V} \ni u, v, \ldots ::= \mathbb{B} \mid \mathbb{C} \mid \lambda x.t \mid () \mid \mathbf{err}\\ \mathbb{T} \ni t, t', t'', \ldots ::= \mathbb{X} \mid \mathbb{V} & & & \mathbb{P} \ni p, q, r, \ldots ::= \mathbf{end}\\ \quad \neg t \mid \mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 & & & \quad \mathbf{send}(t, t', t'')\\ \quad \mathbf{let}\ x = t\ \mathbf{in}\ t' \mid t\, t' & & & \quad \mathbf{recv}(t, t')\\ \quad \mathbf{chan}() \mid \mathbb{P} & & & \quad t \parallel t'\end{array}$$

**Figure 2** Syntax of $\lambda^{\pi\mathsf{D}}_{\leqslant}$ terms. The elements of $\mathbb{C}$ (highlighted) are part of the run-time syntax.

## 2 The $\lambda^{\pi\mathsf{D}}_{\leqslant}$-Calculus

We now present a $\lambda$-calculus extended with channels, input/output, and parallel composition.

▶ **Definition 1.** The syntax of $\lambda^{\pi\mathsf{D}}_{\leqslant}$ is shown in Fig. 2. Elements of $\mathbb{C}$ are run-time syntax. Free/bound variables $\mathrm{fv}(t)$/$\mathrm{bv}(t)$ are defined as usual. We write $\lambda\_.t$ for $\lambda x.t$, when $x \notin \mathrm{fv}(t)$.

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

▶ **Example 2.** The example below instantiates and interconnects three parallel processes:

$$\begin{array}{l}\textbf{let } \mathit{sender} = \lambda y.\mathbf{send}(y, \texttt{"Hello"}, \lambda\_.\mathbf{end}) \textbf{ in}\\ \quad \textbf{let } \mathit{receiver} = \lambda z.\mathbf{recv}(z, \lambda x'.\mathbf{end}) \textbf{ in}\\ \quad\quad \textbf{let } \mathit{fwd} = \lambda i.\lambda o.\mathbf{recv}(i, \lambda z'.\mathbf{send}(o, z', \lambda\_.\mathit{fwd}\, i\, o)) \textbf{ in}\\ \quad\quad\quad \textbf{let } \mathit{sys} = \lambda y'.\lambda z'.(\ \mathit{sender}\, y' \parallel \mathit{fwd}\, y'\, z' \parallel \mathit{receiver}\, z'\ ) \textbf{ in}\\ \quad\quad\quad\quad \textbf{let } y = \mathbf{chan}() \textbf{ in } \textbf{let } z = \mathbf{chan}() \textbf{ in } \mathit{sys}\ y\ z\end{array}$$

- *sender* is an abstract process that takes a channel $y$, uses it to send `"Hello"`, and **end**s;
- *receiver* takes a channel $z$, uses it to receive a value $x'$, and terminates;
- *fwd* takes two channels $i$ and $o$, recursively reads a message from $i$, and writes it in $o$;
- *sys* takes channels $y'$, $z'$, and uses them to instantiate *sender*, *fwd* and *receiver* in parallel;
- in the last line, *sys* is instantiated with $y$ and $z$, that contain channel instances.

Note that, in Ex. 2 and Fig. 1, the last argument of **send**/**recv** is always an abstract process term: this is expected by the semantics (Def. 3), and enforced via typing (§3).

$$\mathcal{E} ::= [\,] \mid \neg\mathcal{E} \mid \mathbf{if}\ \mathcal{E}\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \mid \mathbf{let}\ x = \mathcal{E}\ \mathbf{in}\ t \mid \mathbf{let}\ x = w\ \mathbf{in}\ \mathcal{E} \mid \mathcal{E}\,t \mid w\,\mathcal{E}$$
$$\mathbf{send}(\mathcal{E}, t, t') \mid \mathbf{send}(w, \mathcal{E}, t') \mid \mathbf{send}(w, w', \mathcal{E}) \mid \mathbf{recv}(\mathcal{E}, t) \mid \mathbf{recv}(w, \mathcal{E}) \mid \mathcal{E} \parallel t \quad (w, w' \in \mathbb{V} \cup \mathbb{X})$$

$$\frac{t_1' \equiv t_1 \quad t_1 \to t_2 \quad t_2 \equiv t_2'}{t_1' \to t_2'}\ [\text{R-}\equiv] \qquad \frac{t \to t'}{\mathcal{E}[t] \to \mathcal{E}[t']}\ [\text{R-}\mathcal{E}] \qquad \frac{}{\neg\mathbf{tt} \to \mathbf{ff}}\ [\text{R-}\neg\mathbf{tt}] \qquad \frac{}{\neg\mathbf{ff} \to \mathbf{tt}}\ [\text{R-}\neg\mathbf{ff}]$$

$$\frac{}{(\lambda x.t)\,v \to t\{^v/_x\}}\ [\text{R-}\lambda] \qquad \frac{}{\mathbf{if\ tt\ then}\ t_1\ \mathbf{else}\ t_2 \to t_1}\ [\text{R-}\mathbf{if\text{-}tt}] \qquad \frac{}{\mathbf{if\ ff\ then}\ t_1\ \mathbf{else}\ t_2 \to t_2}\ [\text{R-}\mathbf{if\text{-}ff}]$$

$$\frac{w \in \mathbb{V} \cup \mathbb{X}}{\mathbf{let}\ x = w\ \mathbf{in}\ \mathcal{E}[x] \to \mathbf{let}\ x = w\ \mathbf{in}\ \mathcal{E}[w]}\ [\text{R-}\mathbf{let}] \qquad \frac{\mathsf{a}\ \text{fresh}}{\mathbf{chan}() \to \mathsf{a}}\ [\text{R-}\mathbf{chan}()]$$

$$\frac{}{\mathbf{send}(\mathsf{a}, u, v_1) \parallel \mathbf{recv}(\mathsf{a}, v_2) \to v_1\,() \parallel v_2\,u}\ [\text{R-Comm}]$$

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

**Figure 3** Semantics of $\lambda^{\pi\mathsf{D}}_{\leqslant}$: evaluation contexts, reduction rules, and error rules.

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

Def. 3 formalises a standard call-by-value semantics, and two rules for concurrency: [R-$\mathbf{chan}()$] says that $\mathbf{chan}()$ reduces by returning a fresh channel instance; [R-Comm] says that the parallel composition $\mathbf{send}(\mathsf{a}, u, v_1) \parallel \mathbf{recv}(\mathsf{a}, v_2)$, where both sides operate on a same channel instance $\mathsf{a}$, reduces by transferring the value $u$ on the receiver side, obtaining $v_1\,() \parallel v_2\,u$: hence, if $v_1$ and $v_2$ are function values, the process can keep running by applying $v_1$ to $()$ and $v_2$ to $u$ — i.e. the sent value is substituted inside $v_2$. The error rules formalise 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 instance, and it is an error to put a value in a parallel composition (i.e., only process terms from $\mathbb{P}$ in Fig. 2 can be safely composed by $\parallel$).

▶ **Example 4** (Higher-order $\pi$-calculus [33]). HO$\pi$ is easily encoded in $\lambda^{\pi\mathsf{D}}_{\leqslant}$: below, we render replication $*u?(y).P$ by spawning a replica $z_*\,()$ at every input. The rest is straightforward.

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
[\![x]\!] & = & x & [\![\mathsf{a}]\!] & = & \mathsf{a} & P_1\,P_2 & = & [\![P_1]\!]\,[\![P_2]\!] \\
[\![\lambda x.P]\!] & = & \lambda[\![x]\!].[\![P]\!] & [\![P_1 \parallel P_2]\!] & = & [\![P_1]\!] \parallel [\![P_2]\!] & & & \\
[\![u!\langle v\rangle.P]\!] & = & \mathbf{send}([\![u]\!], [\![v]\!], \lambda_.[\![P]\!]) & [\![(\nu x)P]\!] & = & \mathbf{let}\ [\![x]\!] = \mathbf{chan}()\ \mathbf{in}\ [\![P]\!] & & & \\
[\![u?(x).P]\!] & = & \mathbf{recv}([\![u]\!], \lambda[\![x]\!].[\![P]\!]) & [\![*u?(y).P]\!] & = & \mathbf{let}\ z_* = \lambda_.\mathbf{recv}([\![u]\!], \lambda[\![y]\!].([\![P]\!] \parallel z_*\,())\ \mathbf{in}\ z_*() & & &
\end{array}$$

## 3 Type System

We now introduce the type system of $\lambda^{\pi\mathsf{D}}_{\leqslant}$. Our design is based on $F_{<:}$ (System $F$ with subtyping [5]) equipped with equi-recursive types [16], except that *(1)* we do *not* include polymorphism (it is orthogonal to our aims), *(2)* we include union types, *(3)* we add types for channels and processes, and *(4)* we allow types to contain variables from the term syntax (inspired by [1]). The syntax of types is in Def. 5; however, point *(4)* implies that a type $T$ is only valid if its variables exist in the typing environment — which, in turn, must contain valid types. A similar situation arises in $F_{<:}$, where polymorphic types can depend on type variables in the environment; indeed, we use mutually-defined judgements, similar to those of $F_{<:}$, to assess the validity of environments, types, subtyping, and typed terms (Def. 6).

▶ **Definition 5** (Syntax of types). Types are inductively defined by the following productions:

$$S,T,U \quad ::= \quad \mathbf{bool} \mid \bullet \mid \top \mid \bot \mid T \vee U \mid \Pi(\underline{x}{:}T)U \mid \mu\underline{x}.T \mid \underline{x} \\ \mathrm{c}[T_i,T_o] \mid \mathbf{proc} \mid \mathbf{nil} \mid \mathbf{o}[S,T,U] \mid \mathbf{i}[S,T] \mid \mathbf{p}[T,U]$$

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

We write $\Pi()U$ for $\Pi(\underline{x}{:}\bullet)T$ if $\underline{x} \notin \mathrm{fv}(T)$, and visually distinguish recursion variables as $\mathbf{t}, \mathbf{t}', \ldots$ (i.e., we write $\mu\mathbf{t}.T$). We write $\widetilde{T}$ for a sequence of types $T_1, \ldots, T_n$, and $T \in U$ when $T$ syntactically occurs in $U$. Abbreviations: $\mathrm{c}^{\mathrm{i}}[U] = \mathrm{c}[U,\bot]$, $\mathrm{c}^{\mathrm{o}}[U] = \mathrm{c}[\top,U]$, $\mathrm{c}^{\mathrm{io}}[U] = \mathrm{c}[U,U]$.

The first row of productions in Def. 5 includes booleans, the *unit type* $\bullet$, *top/bottom types* $\top/\bot$, the *union type* $T \vee U$, the *dependent function type* $\Pi(\underline{x}{:}T)U$ and the *recursive type* $\mu\underline{x}.T$ (they both bind $\underline{x}$ with scope $T$), and variables $\underline{x}$ (from the set $\mathbb{X}$ in Def. 1): the underlining helps distinguishing $\underline{x}$ in a type context from $x$ in a $\lambda^{\pi\mathsf{D}}_{\leqslant}$ term context.

The second row of Def. 5 contains concurrency-related types. The *channel type* $\mathrm{c}[T_i,T_o]$ describes a channel that can be used to receive values of type $T_i$, or send values of type $T_o$. The *generic process type* **proc** describes any process term; **nil** describes a terminated process; the *output process type* $\mathbf{o}[S,T,U]$ describes a process that sends a $T$-typed value on an $S$-typed channel, and continues as $U$; the *input process type* $\mathbf{i}[S,T]$ describes a process that receives a value from an $S$-typed channel and continues as $T$; the *parallel type* $\mathbf{p}[T,U]$ describes the parallel composition of a $T$-typed process and an $U$-typed process.

▶ **Definition 6** (Judgements). The following judgements are formalised in Fig. 4:

| | |
|---|---|
| $\vdash \Gamma$ env | $\Gamma$ is a valid typing environment |
| $\Gamma \vdash T$ type | $T$ is a valid type in $\Gamma$. We write $\Gamma \vdash \widetilde{T}$ type iff $\forall U \in \widetilde{T} : \Gamma \vdash U$ type |
| $\Gamma \vdash T$ $\pi$-type | $T$ is a valid $\pi$-type in $\Gamma$. We write $\Gamma \vdash \widetilde{T}$ $\pi$-type iff $\forall U \in \widetilde{T} : \Gamma \vdash U$ $\pi$-type |
| $\Gamma \vdash \widetilde{T}$ *-type | when either $\Gamma \vdash \widetilde{T}$ type or $\Gamma \vdash \widetilde{T}$ $\pi$-type holds |
| $\Gamma \vdash T \leqslant U$ | $T$ is subtype of $U$ in $\Gamma$ — provided that $\Gamma \vdash T,U$ *-type holds |
| $\Gamma \vdash t : T$ | $t$ has type $T$ in $\Gamma$ |

A valid typing environment $\Gamma$ is a mapping from variables (from $\mathbb{X}$ in Def. 1) to types. All judgements in Fig. 4 are inductive, *except* for subtyping, that is *co*inductive (hence the double inference lines). Crucially, Fig. 4 has *two* judgements for valid types: $\Gamma \vdash T$ type and $\Gamma \vdash T$ $\pi$-type. The former is standard (except for rule [T-c], discussed later) while the latter distinguishes *process types*. Importantly, a typing environment $\Gamma$ can map a variable to a type (rule [Γ-x]), but *not* to a $\pi$-type; this also means that function arguments cannot be $\pi$-typed. Still, in a function type $\Pi(\underline{x}{:}T)U$, the return type $U$ can be a $\pi$-type (rule [Tπ-Π]): hence, it is possible to define *abstract process types*, like $T_m$ in §1. Rules [T-μ] and [π-μ] are based on [16, §2], and require valid recursive types to be *contractive*: e.g., $\mu\mathbf{t}_1.\mu\mathbf{t}_2.\ldots.\mu\mathbf{t}_n.(\mathbf{t}_1 \vee U)$ is *not* a type; clause "$\underline{x} \notin \mathrm{fv}^{-}(T)$" means that the bound variable $\underline{x}$ is not in negative position in $T$ *(details: §A)*. Recursive terms and types are supported by rule [t-**let**]: in $\mathbf{let}\ x = t\ \mathbf{in}\ t'$, $t$ can refer to $x$. Rule [⩽-Π] is based on [6], to ensure decidability of subtyping [16, §1]: this is often needed in practice, and we need it in §4. The rest of Fig. 4 is mostly standard; we now discuss the main judgements, and their use.

**Variables, types, subtyping, and dependencies** The environment $\Gamma = x{:}T$ says that the $\lambda^{\pi\mathsf{D}}_{\leqslant}$ variable $x$ has type $T$. Hence, by rule [T-x], the type $\underline{x}$ is valid in $\Gamma$; and indeed, by rule [t-x], we can infer $\Gamma \vdash x : \underline{x}$, i.e., the term $x$ has type $\underline{x}$. Intuitively, this means that type $\underline{x}$ is the "most precise" type for term $x$; this is formally supported by the subtyping rule [t-x], that says: since $\Gamma$ maps term $x$ to $T$, type $\underline{x}$ is smaller than $T$. To retrieve from $\Gamma$ the information that term $x$ has (also) type $T$, we use subtyping and subsumption (rule [t-⩽]),

**$\vdash \Gamma$ env**

$$\frac{}{\vdash \varnothing \text{ env}}\ [\Gamma\text{-}\varnothing] \qquad \frac{\Gamma \vdash T \text{ type} \quad x \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, x{:}T \text{ env}}\ [\Gamma\text{-}x]$$

**$\Gamma \vdash T$ type**

$$\frac{\vdash \Gamma \text{ env} \quad T \in \{\mathbf{bool}, \bullet, \top, \bot\}}{\Gamma \vdash T \text{ type}}\ [T\text{-BASE}] \qquad \frac{\vdash \Gamma, x{:}T \text{ env}}{\Gamma, x{:}T \vdash \underline{x} \text{ type}}\ [T\text{-}\underline{x}] \qquad \frac{\Gamma, x{:}T \vdash U \text{ type}}{\Gamma \vdash \Pi(\underline{x}{:}T)U \text{ type}}\ [T\text{-}\Pi]$$

$$\frac{\Gamma, x{:}\top \vdash T \text{ type} \quad \underline{x} \notin \mathrm{fv}^{-}(T) \quad T \notin \{\underline{z}, U \vee \underline{z}, \underline{z} \vee U \mid \underline{z} \in \mathbb{X}\}}{\Gamma \vdash \mu\underline{x}.T \text{ type}}\ [T\text{-}\mu] \qquad \frac{\Gamma \vdash T \text{ type} \quad \Gamma \vdash U \text{ type}}{\Gamma \vdash T \vee U \text{ type}}\ [T\text{-}\vee]$$

$$\frac{\Gamma \vdash T_i \text{ type} \quad \Gamma \vdash T_o \text{ type} \quad \Gamma \vdash T_o \leqslant T_i}{\Gamma \vdash \mathrm{c}[T_i, T_o] \text{ type}}\ [T\text{-c}]$$

**$\Gamma \vdash T$ $\pi$-type**

$$\frac{\vdash \Gamma \text{ env} \quad T \in \{\mathbf{nil}, \mathbf{proc}\}}{\Gamma \vdash T\ \pi\text{-type}}\ [\pi\text{-BASE}] \qquad \frac{\Gamma \vdash S \leqslant \mathrm{c}[T_i, T_o] \quad \Gamma \vdash T \leqslant T_o \quad \Gamma \vdash U\ \pi\text{-type}}{\Gamma \vdash \mathbf{o}[S, T, \Pi()U]\ \pi\text{-type}}\ [\pi\text{-}\mathbf{o}]$$

$$\frac{\Gamma \vdash S \leqslant \mathrm{c}[T_i, T_o] \quad \Gamma \vdash T_i \leqslant T \quad \Gamma, x{:}T \vdash U\ \pi\text{-type}}{\Gamma \vdash \mathbf{i}[S, \Pi(\underline{x}{:}T)U]\ \pi\text{-type}}\ [\pi\text{-}\mathbf{i}] \qquad \frac{\Gamma \vdash T\ \pi\text{-type} \quad \Gamma \vdash U\ \pi\text{-type}}{\Gamma \vdash \mathbf{p}[T, U]\ \pi\text{-type}}\ [\pi\text{-}\mathbf{p}]$$

$$\frac{\Gamma, x{:}T \vdash U\ \pi\text{-type}}{\Gamma \vdash \Pi(\underline{x}{:}T)U \text{ type}}\ [T\pi\text{-}\Pi] \qquad \frac{\Gamma, x{:}\top \vdash T\ \pi\text{-type} \quad \underline{x} \notin \mathrm{fv}^{-}(T) \quad T \notin \{\underline{z}, U \vee \underline{z}, \underline{z} \vee U \mid \underline{z} \in \mathbb{X}\}}{\Gamma \vdash \mu\underline{x}.T\ \pi\text{-type}}\ [\pi\text{-}\mu] \qquad \frac{\Gamma \vdash T\ \pi\text{-type} \quad \Gamma \vdash U\ \pi\text{-type}}{\Gamma \vdash T \vee U\ \pi\text{-type}}\ [\pi\text{-}\vee]$$

**$\Gamma \vdash T \leqslant U$**

$$\frac{}{\Gamma \vdash T \leqslant T}\ [\leqslant\text{-REFL}] \qquad \frac{\Gamma \vdash T \leqslant S \quad \Gamma \vdash U \leqslant S}{\Gamma \vdash T \vee U \leqslant S}\ [\leqslant\text{-}\vee\text{L}] \qquad \frac{\Gamma \vdash T \leqslant T'}{\Gamma \vdash T \leqslant T' \vee U}\ [\leqslant\text{-}\vee\text{R1}] \qquad \frac{\Gamma \vdash U \leqslant U'}{\Gamma \vdash U \leqslant T \vee U'}\ [\leqslant\text{-}\vee\text{R2}]$$

$$\frac{\Gamma \vdash T\{\mu\mathbf{t}.T/\mathbf{t}\} \leqslant U}{\Gamma \vdash \mu\mathbf{t}.T \leqslant U}\ [\leqslant\text{-}\mu\text{L}] \qquad \frac{\Gamma \vdash T \leqslant U\{\mu\mathbf{t}.U/\mathbf{t}\}}{\Gamma \vdash T \leqslant \mu\mathbf{t}.U}\ [\leqslant\text{-}\mu\text{R}] \qquad \frac{\Gamma \vdash \Gamma(x) \leqslant T}{\Gamma \vdash \underline{x} \leqslant T}\ [\leqslant\text{-}\underline{x}]$$

$$\frac{\Gamma, x{:}T \vdash U \leqslant U'}{\Gamma \vdash \Pi(\underline{x}{:}T)U \leqslant \Pi(\underline{x}{:}T)U'}\ [\leqslant\text{-}\Pi] \qquad \frac{\Gamma \vdash T_i \leqslant U_i \quad \Gamma \vdash U_o \leqslant T_o}{\Gamma \vdash \mathrm{c}[T_i, T_o] \leqslant \mathrm{c}[U_i, U_o]}\ [\leqslant\text{-c}]$$

$$\frac{\Gamma \vdash T\ \pi\text{-type}}{\Gamma \vdash T \leqslant \mathbf{proc}}\ [\leqslant\text{-}\mathbf{proc}] \qquad \frac{\Gamma \vdash S \leqslant S' \quad \Gamma \vdash T \leqslant T' \quad \Gamma \vdash U \leqslant U'}{\Gamma \vdash \mathbf{o}[S, T, U] \leqslant \mathbf{o}[S', T', U']}\ [\leqslant\text{-}\mathbf{o}]$$

$$\frac{\Gamma \vdash T \leqslant T' \quad \Gamma \vdash U \leqslant U'}{\Gamma \vdash \mathbf{i}[T, U] \leqslant \mathbf{i}[T', U']}\ [\leqslant\text{-}\mathbf{i}] \qquad \frac{\Gamma \vdash S \leqslant S' \quad \Gamma \vdash T \leqslant T'}{\Gamma \vdash \mathbf{p}[T, U] \leqslant \mathbf{p}[T', U']}\ [\leqslant\text{-}\mathbf{p}]$$

**$\Gamma \vdash t : T$**

$$\frac{\vdash \Gamma, x{:}T \text{ env}}{\Gamma, x{:}T \vdash x : \underline{x}}\ [t\text{-}x] \qquad \frac{\vdash \Gamma \text{ env} \quad v \in \mathbb{B}}{\Gamma \vdash v : \mathbf{bool}}\ [t\text{-}\mathbb{B}] \qquad \frac{\vdash \Gamma \text{ env}}{\Gamma \vdash () : \bullet}\ [t\text{-}()] \qquad \frac{\Gamma \vdash t : \mathbf{bool}}{\Gamma \vdash \neg t : \mathbf{bool}}\ [t\text{-}\neg]$$

$$\frac{\Gamma, x{:}U \vdash t : T}{\Gamma \vdash \lambda x^{U}.t : \Pi(\underline{x}{:}U)T}\ [t\text{-}\lambda] \qquad \frac{\Gamma \vdash t : T \quad \Gamma \vdash T \leqslant U}{\Gamma \vdash t : U}\ [t\text{-}\leqslant]$$

$$\frac{\Gamma \vdash T \vee U\ {}^{*}\text{-type} \quad \Gamma \vdash t : \mathbf{bool} \quad \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : U}{\Gamma \vdash \mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 : T \vee U}\ [t\text{-}\mathbf{if}]$$

$$\frac{\Gamma \vdash t_1 : \Pi(\underline{x}{:}U)T \quad \Gamma \vdash t_2 : U' \quad \Gamma \vdash U' \leqslant U}{\Gamma \vdash t_1\, t_2 : T\{{}^{U'}/_{\underline{x}}\}}\ [t\text{-APP}] \qquad \frac{\Gamma, x{:}U \vdash t : U' \quad \Gamma, x{:}U \vdash t' : T \quad \Gamma \vdash U' \leqslant U}{\Gamma \vdash \mathbf{let}\ x^{U} = t\ \mathbf{in}\ t' : T\{{}^{U'}/_{\underline{x}}\}}\ [t\text{-}\mathbf{let}]$$

$$\frac{\Gamma \vdash \mathrm{c}[T_i, T_o] \text{ type}}{\Gamma \vdash \mathsf{a}^{T_i, T_o} : \mathrm{c}[T_i, T_o]}\ [t\text{-}\mathbb{C}] \qquad \frac{\Gamma \vdash \mathrm{c}[T_i, T_o] \text{ type}}{\Gamma \vdash \mathbf{chan}()^{T_i, T_o} : \mathrm{c}[T_i, T_o]}\ [t\text{-}\mathbf{chan}] \qquad \frac{\vdash \Gamma \text{ env}}{\Gamma \vdash \mathbf{end} : \mathbf{nil}}\ [t\text{-}\mathbf{end}]$$

$$\frac{\Gamma \vdash \mathbf{o}[S, T, U]\ \pi\text{-type} \quad \Gamma \vdash t_1 : S \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : U}{\Gamma \vdash \mathbf{send}(t_1, t_2, t_3) : \mathbf{o}[S, T, U]}\ [t\text{-}\mathbf{send}]$$

$$\frac{\Gamma \vdash \mathbf{i}[S, T]\ \pi\text{-type} \quad \Gamma \vdash t_1 : S \quad \Gamma \vdash t_2 : T}{\Gamma \vdash \mathbf{recv}(t_1, t_2) : \mathbf{i}[S, T]}\ [t\text{-}\mathbf{recv}] \qquad \frac{\Gamma \vdash \mathbf{p}[T, U]\ \pi\text{-type} \quad \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : U}{\Gamma \vdash t_1 \parallel t_2 : \mathbf{p}[T, U]}\ [t\text{-}\|]$$

**Figure 4** Judgements of the $\lambda^{\pi\mathsf{D}}_{\leqslant}$ type system (Def. 6). Concurrency-related parts are highlighted.

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

as shown on the left. Since $\underline{x}$ is the smallest type for term $x$, the judgement $\Gamma \vdash t : \underline{x}$ can only hold if $t$ is "something that evaluates to $x$", e.g., $t = x$ or $t =$ **if tt then** $x$ **else** $x$; similarly, the dependent function type $\Pi(\underline{x}{:}\mathrm{bool})\underline{x}$ is inhabited by terms like $\lambda x.x$ or $\lambda x.(\lambda y.y)\,x$. Hence, we can roughly say: if a type $T$ mentions $\underline{x}$, then $T$-typed terms must correspondingly use variable $x$. This is key to track channel usage in process types, as we will see shortly.

**Channels, processes, and their types** By rule [$t$-**chan**], a (type-annotated) term $\mathbf{chan}()^{T_i,T_o}$ has type $\mathrm{c}[T_i, T_o]$, provided that the latter is a valid type: this means, by rule [$T$-c], that $T_o$ must be subtype of $T_i$ — e.g., $\mathrm{c}[\mathrm{int}, \mathrm{real}]$ is not a valid type, as it would describe an unsafe channel that could be used to send real values, but receive them as integers. Rule [$t$-$\mathbb{C}$] is similar, but for channel instances. By rule [$t$-**end**], the terminated process **end** has type **nil**.

By [$t$-**send**], $\mathbf{send}(t_1, t_2, t_3)$ has type $\mathbf{o}[S, T, U]$, which is only well-formed under the constraints of rule [$\pi$-**o**]. Hence, $t_1$ must be a channel type that allows to send values of type $T$, and correspondingly, $t_2$ (the term being sent) must have type $T$; further, $t_3$ must have type $U = \Pi()U'$ (for some $\pi$-type $U'$), hence is a process thunk that runs by applying $t_3\,()$.

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

Finally, rule [$t$-$\|$] types $t_1 \,\|\, t_2$, by ensuring that both sub-terms $t_1$ and $t_2$ are $\pi$-typed.

We now show some examples, and then discuss the remaining part of Fig. 4 (subtyping).

▶ **Example 7.** Take Ex. 2. We have the following type assignments:

$$\begin{aligned}
\mathit{sender} &: T_{snd} = \Pi(\underline{y}{:}\mathrm{c}^{\mathrm{o}}[\mathrm{str}])\,\mathbf{o}\big[\underline{y}, \mathrm{str}, \Pi()\mathbf{nil}\big] \\
\mathit{receiver} &: T_{rcv} = \Pi(\underline{z}{:}\mathrm{c}^{\mathrm{i}}[\mathrm{str}])\,\mathbf{i}\big[\underline{z}, \Pi(\underline{x'}{:}\mathrm{str})\mathbf{nil}\big] \\
\mathit{fwd} &: T_{fwd} = \Pi(\underline{i}{:}\mathrm{c}^{\mathrm{i}}[\mathrm{str}])\,\Pi(\underline{o}{:}\mathrm{c}^{\mathrm{o}}[\mathrm{str}])\,\mu\mathbf{t}.\mathbf{i}\big[\underline{i}, \Pi(\underline{z'}{:}\mathrm{str})\mathbf{o}\big[\underline{o}, \underline{z'}, \Pi()\mathbf{t}\big]\big] \\
\mathit{sys} &: T_{sys} = \Pi(\underline{y'}{:}\mathrm{c}^{\mathrm{i}}[\mathrm{str}])\,\Pi(\underline{z'}{:}\mathrm{c}^{\mathrm{o}}[\mathrm{str}])\,\mathbf{p}\big[\mathbf{p}\big[(T_{snd}\ \underline{y'}), (T_{fwd}\ \underline{y'}\ \underline{z'})\big], (T_{rcv}\ \underline{z'})\big]
\end{aligned}$$

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

$$T_{sys} = \Pi(\underline{y'}{:}\mathrm{c}^{\mathrm{i}}[\mathrm{str}])\,\Pi(\underline{z'}{:}\mathrm{c}^{\mathrm{o}}[\mathrm{str}])\,\mathbf{p}\left[\begin{array}{l}\mathbf{p}\big[\mathbf{o}\big[\underline{y'}, \mathrm{str}, \Pi()\mathbf{nil}\big],\ \mu\mathbf{t}.\mathbf{i}\big[\underline{y'}, \Pi(\underline{x}{:}\mathrm{str})\mathbf{o}\big[\underline{z'}, \underline{x}, \Pi()\mathbf{t}\big]\big]\big], \\ \mathbf{i}\big[\underline{z'}, \Pi(\underline{x'}{:}\mathrm{str})\mathbf{nil}\big]\end{array}\right]$$

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

▶ **Example 8** (Mobile code)**.** We now reprise the example of §1, and its type $T_m$:

$$T_m = \Pi(\underline{i_1}{:}\mathrm{c}^{\mathrm{i}}[\mathrm{int}])\,\Pi(\underline{i_2}{:}\mathrm{c}^{\mathrm{i}}[\mathrm{int}])\,\Pi(\underline{o}{:}\mathrm{c}^{\mathrm{o}}[\mathrm{int}])\,\mu\mathbf{t}.\mathbf{i}\big[\underline{i_2}, \Pi(\underline{x}{:}\mathrm{int})\mathbf{i}\big[\underline{i_2}, \Pi(\underline{y}{:}\mathrm{int})\mathbf{o}\big[\underline{o}, (\underline{x} \vee \underline{y}), \Pi()\mathbf{t}\big]\big]\big]$$

Like $f$ in Fig. 1, the terms below implement $T_m$: $m_1$ always sends $x$ received from $i_1$, then recursively calls itself by swapping $i_1$ and $i_2$; $m_2$ sends the maximum value between $x$ and $y$.

$$\mathbf{let}\ m_1 = \lambda i_1.\lambda i_2.\lambda o.\big(\mathbf{recv}(i_1, \lambda x.\mathbf{recv}(i_2, \lambda\_.\mathbf{send}(o, x, \lambda\_.m_1\ i_2\ i_1\ o)))\big)$$
$$\mathbf{let}\ m_2 = \lambda i_1.\lambda i_2.\lambda o.\big(\mathbf{recv}(i_1, \lambda x.\mathbf{recv}(i_2, \lambda y.\mathbf{send}(o, (\mathbf{if}\ x{>}y\ \mathbf{then}\ x\ \mathbf{else}\ y), \lambda\_.m_2\ i_1\ i_2\ o)))\big)$$

The following term represents a server that uses channels $cm$ and $out$. It creates two internal channels $z_1$ and $z_2$, uses channel $cm$ to receive and abstract process $p$, and runs it, in parallel with two *prod*ucer processes (definitions omitted) that send values on $z_1$ and $z_2$:

$$\mathbf{let}\ \mathit{srv} = \lambda cm.\lambda out.\ \mathbf{let}\ z_1 = \mathbf{chan}()\ \mathbf{in}\ \ \mathbf{let}\ z_2 = \mathbf{chan}()\ \mathbf{in}\ \ \mathbf{recv}(cm, \lambda p.\ \big(\ p\ z_1\ z_2\ out \parallel \mathit{prod}_1\ z_1 \parallel \mathit{prod}_2\ z_2\ \big)\ )$$

The system works correctly if the received code $p$ is $f$, $m_1$ or $m_2$ above — or any other instance of $T_m$. To ensure that $srv$ receives a suitable process on $cm$, we check its type:

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

and this ensures that, e.g., the parallel composition $\mathbf{send}(x, t, \lambda\_.\mathbf{end}) \parallel srv\ x\ out$ is typable in $\Gamma$ only if $\Gamma \vdash x : \mathrm{c}^{\mathrm{io}}[T_m]$ holds, which implies that $\Gamma \vdash t : T_m$ holds. We can be more detailed. If $U_1/U_2$ are the types of $prod_1/prod_2$, the $\mathbf{recv}(\ldots,\ldots)$ sub-term of $srv$ has type:

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

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

▶ **Example 9** (Channel passing). Below, $T_p$ describes an abstract process that uses channel $x$ to receive a channel $y$, then either replies on $y$ (sending a string), or forwards $y$ through $z$:

$$T_p \;=\; \Pi(\underline{x}{:}\mathrm{c}^{\mathrm{i}}[\mathrm{c}^{\mathrm{o}}[\mathrm{str}]])\,\Pi(\underline{z}{:}\mathrm{c}^{\mathrm{o}}[\mathrm{c}^{\mathrm{o}}[\mathrm{str}]])\,\mathbf{i}\big[\underline{x},\ \Pi(\underline{y}{:}\mathrm{c}^{\mathrm{o}}[\mathrm{str}])\,\mathbf{o}\big[\underline{y}, \mathrm{str}, \Pi()\mathbf{nil}\big] \vee \mathbf{o}\big[\underline{z}, \underline{y}, \Pi()\mathbf{nil}\big]\,\big]$$

Two processes of type $T_p$ are $t_1 = \lambda x.\lambda z.\mathbf{recv}(x, \lambda y.\mathbf{send}(z, y, \lambda\_.\mathbf{end}))$ (always forwards) and $t_2 = \lambda x.\lambda\_.\mathbf{send}(y, \texttt{"Hi"}, \lambda\_.\mathbf{end})$ (always replies). Their types can be more precise:

$$\begin{array}{rcl} t_1 \;:\; T_{p1} \;=\; \Pi(\underline{x}{:}\mathrm{c}^{\mathrm{i}}[\mathrm{c}^{\mathrm{o}}[\mathrm{str}]])\,\Pi(\underline{z}{:}\mathrm{c}^{\mathrm{o}}[\mathrm{c}^{\mathrm{o}}[\mathrm{str}]])\,\mathbf{i}\big[\underline{x},\ \Pi(\underline{y}{:}\mathrm{c}^{\mathrm{o}}[\mathrm{str}])\,\mathbf{o}\big[\underline{z}, \underline{y}, \Pi()\mathbf{nil}\big]\,\big] & \leqslant & T_p \\ t_2 \;:\; T_{p2} \;=\; \Pi(\underline{x}{:}\mathrm{c}^{\mathrm{i}}[\mathrm{c}^{\mathrm{o}}[\mathrm{str}]])\,\Pi(\underline{z}{:}\mathrm{c}^{\mathrm{o}}[\mathrm{c}^{\mathrm{o}}[\mathrm{str}]])\,\mathbf{i}\big[\underline{x},\ \Pi(\underline{y}{:}\mathrm{c}^{\mathrm{o}}[\mathrm{str}])\,\mathbf{o}\big[\underline{y}, \mathrm{str}, \Pi()\mathbf{nil}\big]\,\big] & \leqslant & T_p \end{array}$$

If a term $t_{12}$ interconnects instances of $t_1$ and $t_2$, this is reflected in its type $T_{p12}$:

$$x{:}\mathrm{c}^{\mathrm{io}}[\mathrm{c}^{\mathrm{o}}[\mathrm{str}]]\,,\ z{:}\mathrm{c}^{\mathrm{io}}[\mathrm{c}^{\mathrm{o}}[\mathrm{str}]] \;\vdash\; t_{12} : T_{p12} \quad \text{where} \quad t_{12} = t_1\ x\ z \parallel t_2\ z\ z \quad T_{p12} = \mathbf{p}\big[\,T_{p1}\ \underline{x}\ \underline{z}\,, T_{p2}\ \underline{z}\ \underline{z}\,\big]$$

**Subtyping, subsumption, and private channels** The subtyping rules in Fig. 4 are standard (based on those of $F_{<:}$ [5, 16]) except for the highlighted ones. Rule [⩽-c] says that subtyping for channel types is covariant on input capabilities, and contravariant on output capabilities, as expected [22]: intuitively, channels with smaller types can be used more liberally. Rule [⩽-**proc**] says that **proc** is the top type for $\pi$-types. Rules [⩽-**o**]/[⩽-**i**]/[⩽-**p**] say that the types describing input/output/parallel processes are covariant in all their parameters.

As usual, supertyping and subsumption (rule [$t$-⩽]) enlarge the type of a term, providing flexibility according to the Liskov & Wing's substitution principle [19] (a smaller object can be used when a larger one is required). But in our framework, supertyping also allows to drop information when typing *private* channel, as illustrated in Ex. 10.

▶ **Example 10** (Subtyping, private channels, and precision loss). Consider the terms and types:

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

Letting $\Gamma = x{:}\mathrm{c}^{\mathrm{io}}[\mathrm{int}]$, we have $\Gamma \vdash \underline{x} \leqslant \mathrm{c}^{\mathrm{io}}[\mathrm{int}]$ and $\Gamma \vdash T_1 \leqslant T_2$. For term $t_1$, we have both $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_1 : T_2$ (by [$t$-⩽]). 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: it says that *some* term with type $\mathrm{c}^{\mathrm{io}}[\mathrm{int}]$ is used to send, while $x$ is used to receive.

We also have $\Gamma \vdash t_2 : T_2$; and notably, since $z$ is bound in the "**let** ..." sub-processes, it cannot appear in the type: i.e., we cannot write a more detailed type for $t_2$. This is because:

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

Hence, rule [$t$-**let**] ensures that, if $\underline{z}$ occurs in the type of $\mathbf{send}(\ldots)$, then it is replaced by a (super-)type that is suitable for both $z$ and $\mathbf{chan}()$ — in this case, $\mathrm{c}^{\mathrm{io}}[\mathrm{int}]$.

Ex. 10 shows that the type system cannot precisely track how a term uses its internal channels. This information loss is key to type Turing-powerful $\lambda^{\pi\mathsf{D}}_{\leqslant}$ terms with a non-Turing-complete type language, and will lead to the decidable results presented in §4.

Def. 6 is sufficient to prove that well-typed terms never go wrong.

▶ **Theorem 11** (Type safety). *If* $\Gamma \vdash t : T$, *then* $t$ *is safe (Def. 3).*

Theorem 11 follows by the inductive property: $\Gamma \vdash t : T$ and $t \to t'$ implies $\exists t' : \Gamma \vdash t : T'$ — i.e., typed terms only reduce to typed terms, that cannot contain (untypable) **err** subterms. This is expected, as we combine System $F_{<:}$-style typing rules, and typed I/O channels. In §4, we study how $T$ and $T'$ are related, and how their relation constraints $t$'s behaviour.

## 4 Type-Based Analysis and Behavioural Properties

We now show how behavioural properties of types reflect on typed processes. Ex. 10 reveals that the most precise types require *open* terms in their typing environment — but Def. 3 works on *closed* terms. So, by observing how $T_1$ in Ex. 10 uses $\underline{x}$, we can sense that $t_1$ should communicate via $x$ — but by Def. 3, $t_1$ is stuck. We can trigger communication by replacing $x$ in $t_1$ with a channel instance, e.g., with $t_1' = \mathbf{let}\ x = \mathbf{chan}()\ \mathbf{in}\ t_1$ — but the type of $t_1'$ cannot mention the bound $\underline{x}$, hence does not precisely convey which channel(s) $t_1'$ uses.

With this insight, we develop a type-based analysis in four steps: *(1)* we define a labelled transition semantics for typed $\lambda^{\pi\mathsf{D}}_{\leqslant}$ terms with free variables (Def. 12); *(2)* we define a labelled semantics for types (Def. 13); *(3)* we formalise subject transition and type fidelity (Thm. 14, 15); *(4)* we use them to show how temporal logic judgements on types transfer to processes.

▶ **Definition 12** (Labelled semantics of open typed terms). When $\Gamma \vdash t : T$ (for some $T$), the judgements $\Gamma \vdash t \xrightarrow{\alpha} t'$ and $\Gamma \vdash t \xrightarrow{\tau}{}^{*}\ t'$ are inductively defined as:

$$\frac{t \to t' \text{ by base rule } [\mathrm{R}]}{\Gamma \vdash t \xrightarrow{\tau[\mathrm{R}]} t'}\ [\text{SR-}\to] \qquad \Gamma \vdash \neg x \xrightarrow{\tau[\neg x]} \mathbf{tt} \qquad \Gamma \vdash \neg x \xrightarrow{\tau[\neg x]} \mathbf{ff} \qquad \Gamma \vdash \mathbf{if}\ x\ \mathbf{then}\ t\ \mathbf{else}\ t' \xrightarrow{\tau[\mathbf{if}\ x]} t \qquad \Gamma \vdash \mathbf{if}\ x\ \mathbf{then}\ t\ \mathbf{else}\ t' \xrightarrow{\tau[\mathbf{if}\ x]} t'$$

$$\frac{w, w', w'' \in \mathbb{X} \cup \mathbb{V}}{\Gamma \vdash \mathbf{send}(w, w', w'') \xrightarrow{\overline{w}\langle w' \rangle} w''()}\ [\text{SR-}\mathbf{send}] \qquad \frac{w, w', w'' \in \mathbb{X} \cup \mathbb{V} \quad \Gamma \vdash w : \mathrm{c}[T_i, T_o] \quad \Gamma \vdash w' : T_i}{\Gamma \vdash \mathbf{recv}(w, w'') \xrightarrow{w(w')} w''\, w'}\ [\text{SR-}\mathbf{recv}]$$

$$\frac{\Gamma \vdash t \xrightarrow{\overline{w}\langle w' \rangle} t' \quad \Gamma \vdash t'' \xrightarrow{w(w')} t'''}{\Gamma \vdash t \parallel t'' \xrightarrow{\tau[w]} t' \parallel t'''}\ [\text{SR-Comm}] \qquad \frac{\Gamma \vdash x\, w : T \quad w \in \mathbb{X} \cup \mathbb{V} \quad \Gamma \vdash v\{w/y\} : T}{\Gamma \vdash x\, w \xrightarrow{\tau[x()]} v\{w/y\}}\ [\text{SR-}x()]$$

$$\frac{}{\Gamma \vdash (\lambda y.t)\, x \xrightarrow{\tau[\lambda()]} t\{x/y\}}\ [\text{SR-}\lambda()] \qquad \frac{\Gamma \vdash t' \xrightarrow{\alpha} t'' \quad \mathrm{fv}(\alpha) \cap \mathrm{bv}(\mathcal{E}) = \varnothing}{\Gamma \vdash \mathcal{E}[t] \xrightarrow{\alpha} \mathcal{E}[t']}\ [\text{SR-}\mathcal{E}]$$

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

Unlike Def. 3, Def. 12 lets an open term like $\neg x$ reduce, by non-deterministically instantiating $x$ to **tt** or **ff**; the assumption $\Gamma \vdash \neg x : T$ ensures that $x$ is a boolean. Rule [SR-→] inherits "concrete" reductions from Def. 3: if $t \to t'$ is induced by base rule [R], the transition label is $\tau[\mathrm{R}]$. Rules [SR-**send**]/[SR-**recv**] send/receive a value/variable $w'$ using a (channel-typed) value/variable $w$. Note that in [SR-**recv**], $w'$ is *any* value/variable of type $T_i$, which is the input type of $x$ (in $\pi$-calculus jargon, it is an *early* semantics). Rule [SR-Comm] lets processes exchange a payload $w'$ via a channel/variable $w$, recording $w$ in the transition label. Rule [SR-$x()$] "applies" $x$ by instantiating it with any suitably-typed $v$ (i.e., $v$ must be a function), and recording $x$ in the transition label. Rule [SR-$\lambda()$] applies a function to a variable $y$, with the expected substitution. Rule [SR-$\mathcal{E}$] propagates transitions through contexts, unless labels refer to bound variables. Finally, $\Gamma \vdash t \xrightarrow{\tau}{}^{*}\ t'$ holds when $t$ reaches $t'$ via a *finite* sequence of internal moves *excluding interaction*: labels $w(w')$, $\overline{w}\langle w' \rangle$, $\tau[w]$, and $\tau[\text{R-Comm}]$ are forbidden.

Using Def. 12 on $t_1$ from Ex. 10, we obtain $\Gamma \vdash t_1 \xrightarrow{\tau[x]} \mathbf{end} \parallel \mathbf{end}$, as desired.

We now introduce a labelled transition semantics for types (Def. 13 below). We want type transitions to abstract the possible interactions of typed processes: therefore, a type like $\mathbf{p}[\mathbf{o}[\underline{x}, \ldots], \mathbf{i}[\underline{x}, \ldots]]$ should reduce with a communication on $\underline{x}$. This intuition applies,

e.g., to $T_1$ in Ex. 10, and mimics the term reduction $\Gamma \vdash t_1 \xrightarrow{\tau[x]} \mathbf{end} \parallel \mathbf{end}$. But consider $T_2$ in Ex. 10: since it also types $t_1$ in $\Gamma$, it should also mimic $t_1$'s reduction — hence, a type like $\mathbf{p}[\mathbf{o}[\mathrm{c}^{\mathrm{io}}[\mathrm{int}],\ldots]\,,\mathbf{i}[\underline{x},\ldots]]$ should reduce, too. In general, we want $\mathbf{p}[\mathbf{o}[S,\ldots],\mathbf{i}[T,\ldots]]$ to reduce whenever $S$ and $T$ "might interact", in the sense that they could type a same channel/variable: we formalise this idea as the judgement $\Gamma \vdash S \bowtie T$ in Def. 13 below.

▶ **Definition 13** (Type semantics). Let $S \sqcap_\Gamma T$ and $S \sqcup_\Gamma T$ be the greatest subtype / least supertype of $S$ and $T$ in $\Gamma$. The judgement $\Gamma \vdash S \bowtie T$ ("$S$ and $T$ might interact") is:

$$\frac{\Gamma \vdash \underline{x} \leqslant T}{\Gamma \vdash \underline{x} \bowtie T}\ [\bowtie\text{-}x\mathrm{L}] \qquad \frac{\Gamma \vdash \underline{x} \leqslant S}{\Gamma \vdash S \bowtie \underline{x}}\ [\bowtie\text{-}x\mathrm{R}] \qquad \frac{\Gamma \vdash \mathrm{c}[\,S_i \sqcap_\Gamma T_i\,,\, S_o \sqcup_\Gamma T_o\,]\ \mathrm{type}}{\Gamma \vdash \mathrm{c}[S_i,S_o] \bowtie \mathrm{c}[T_i,T_o]}\ [\bowtie\text{-c}]$$

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

$$T \vee U \equiv U \vee T \quad \mathbf{p}[T,U] \equiv \mathbf{p}[U,T] \quad \mathbf{p}[S,\mathbf{p}[T,U]] \equiv \mathbf{p}[\mathbf{p}[S,T],U] \quad \mathbf{p}[T,\mathbf{nil}] \equiv T$$

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

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

The judgements $\Gamma \vdash T \xrightarrow{\alpha} T'$ and $\Gamma \vdash T \xrightarrow{\tau[\vee\mu]}{}^{*}\, T'$ are inductively defined as:

$$\frac{}{\Gamma \vdash T \vee U \xrightarrow{\tau[\vee]} T} \qquad \frac{}{\Gamma \vdash \mu\mathbf{t}.T \xrightarrow{\tau[\mu]} T\{\mu\mathbf{t}.T/\mathbf{t}\}} \qquad \frac{\Gamma \vdash T \xrightarrow{\alpha} T'}{\Gamma \vdash \mathcal{E}[T] \xrightarrow{\alpha} \mathcal{E}[T']} \qquad \frac{T' \equiv T \quad \Gamma \vdash T \xrightarrow{\alpha} U \quad U \equiv U'}{\Gamma \vdash T' \xrightarrow{\alpha} U'}$$

$$\frac{}{\Gamma \vdash \mathbf{o}[S,T,\Pi()U] \xrightarrow{\overline{S}\langle T\rangle} U}\ [T\to\mathbf{o}] \qquad \frac{\Gamma \vdash T' \leqslant T \qquad T' = T \text{ or } T' \in \mathbb{X}}{\Gamma \vdash \mathbf{i}[S,\Pi(\underline{x}{:}T)U] \xrightarrow{S(T')} U\{T'/\underline{x}\}}\ [T\to\mathbf{i}]$$

$$\frac{\Gamma \vdash U \xrightarrow{\overline{S}\langle \underline{x}\rangle} U' \quad \Gamma \vdash U'' \xrightarrow{S'(\underline{x})} U''' \quad \Gamma \vdash S \bowtie S'}{\Gamma \vdash \mathbf{p}[U,U''] \xrightarrow{\tau[S,S']} \mathbf{p}[U',U''']}\ [T\to\mathbf{io}x]$$

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

$$\frac{}{\Gamma \vdash T \xrightarrow{\tau[\vee\mu]}{}^{*}\, T} \qquad \frac{\Gamma \vdash T \xrightarrow{\tau[\vee\mu]}{}^{*}\, T' \qquad \Gamma \vdash T' \xrightarrow{\tau[\vee]} T'' \text{ or } \Gamma \vdash T' \xrightarrow{\tau[\mu]} T''}{\Gamma \vdash T \xrightarrow{\tau[\vee\mu]}{}^{*}\, T''}$$

By Def. 13, $\Gamma \vdash S \bowtie S'$ holds when $S$ and $S'$ are channel types (or channel-typed variables) that might type a same term in $\Gamma$, via rule [$t$-$\leqslant$]. When either $S$ or $S'$ is a variable $\underline{x}$ (rules [$\bowtie$-$x$L]/[$\bowtie$-$x$R]), we just check whether the other is a supertype. Otherwise, rule [$\bowtie$-c] checks whether $S$ and $S'$ have a common valid channel subtype, whose input capabilities are smaller than both, and output capabilities larger than both (cf. rule [$\leqslant$-c], Fig. 4); if such a type exists, then communication *might* occur, via rules [R-Comm]/[SR-Comm]. E.g., $S = \mathrm{c}[\mathrm{int},\mathrm{int}]$ and $S' = \mathrm{c}[\mathrm{real},\mathrm{real}]$ *cannot* interact: letting $U_i = \mathrm{int} \sqcap_\Gamma \mathrm{real} = \mathrm{int}$ and $U_o = \mathrm{int} \sqcup_\Gamma \mathrm{real} = \mathrm{real}$, we get $\Gamma \nvdash U_o \leqslant U_i$, hence $\Gamma \nvdash \mathrm{c}[U_i,U_o]$ type (cf. rule [$T$-c], Fig. 4), and conclude $\Gamma \nvdash S \bowtie S'$. Instead, $S = \mathrm{c}[\top,\mathrm{int}]$ and $S' = \mathrm{c}[\mathrm{real},\bot]$ *might* interact: letting $U_i = \top \sqcap_\Gamma \mathrm{real} = \mathrm{real}$ and $U_o = \mathrm{int} \sqcup_\Gamma \bot = \mathrm{int}$, we get $\Gamma \vdash U_o \leqslant U_i$, hence $\Gamma \vdash \mathrm{c}[U_i,U_o]$ type, and conclude $\Gamma \vdash S \bowtie S'$.

The judgement $\Gamma \vdash T \xrightarrow{\alpha} T'$ says that $T \vee U$ can reduce to either $T$ or $U$; recursive types reduce by unfolding; type contexts $\mathcal{E}$ allow, e.g., $S = \mu\mathbf{t}.S'$ to unfold inside $\mathbf{p}[S,U]$, exposing the prefix needed by other rules; reductions are up-to congruence $\equiv$, that can swap $\vee$ branches, and reorganise $\mathbf{p}[\ldots,\ldots]$ as a commutative monoid, with unit $\mathbf{nil}$. Rule [$T\to\mathbf{o}$] reduces an output type, recording the used channel type $S$ and the payload $T$ in the transition label. Rule [$T\to\mathbf{i}$] works similarly for input types, and records the received payload type $T'$; 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 $\Pi(\underline{x}{:}T)U$; or
*(b)* $T' = \underline{z}$, for any $z \in \mathbb{X}$. In this case, clause $\Gamma \vdash T' \leqslant T$ requires type $\underline{z}$ to be compatible with the argument type of the continuation; moreover, it implicitly ensures that $\underline{z} \in \mathrm{dom}(\Gamma)$.

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 $\underline{z}$ through the dependent function type $\Pi(\underline{x}{:}T)U$. Crucially, *(a)* and *(b)* imply that rule $[T{\to}\mathbf{i}]$ is *finite-branching* (unlike rule [SR-**recv**] in Def. 12). There are two communication rules:

- $[T{\to}\mathbf{io}x]$ fires when, in $\mathbf{p}[U, U']$, we have $\Gamma \vdash U \xrightarrow{\overline{S}\langle \underline{x} \rangle} U'$ and $\Gamma \vdash U'' \xrightarrow{S'(\underline{x})} U'''$, and $S, S'$ might interact. In this case, the type reduces to $\mathbf{p}[U', U''']$. Note that, by $[T{\to}\mathbf{i}]$, the payload $\underline{x}$ sent by $U$ is substituted in $U'''$, hence it can appear in its future transitions. The rule produces a transition label $\tau[S, S']$ that records which channel types were used;
- $[T{\to}\mathbf{io}]$ is similar, but fires if the output payload $T$ is *not* a type variable. Note that clause $\Gamma \vdash S \bowtie S'$ ensures that $U''$ has a $S'(T')$-transition with $\Gamma \vdash T \leqslant T'$, and the rule fires it.

E.g., by rule $[T{\to}\mathbf{io}x]$, in Ex. 10 we have $\Gamma \vdash T_1 \xrightarrow{\tau[\underline{x},\underline{x}]} \mathbf{p}\big[\mathbf{nil}, \mathbf{nil}\big\{{}^{\mathrm{int}}/_{\underline{y}}\big\}\big]$ and $\Gamma \vdash T_2 \xrightarrow{\tau[\mathrm{c}^{\mathrm{io}}[\mathrm{int}],\underline{x}]} \mathbf{p}\big[\mathbf{nil}, \mathbf{nil}\big\{{}^{\mathrm{int}}/_{\underline{y}}\big\}\big]$ — and the transition of $T_1$ shows that *precisely* channel $\underline{x}$ is used. Note that if a type $\mathbf{p}[\mathbf{o}[S, \ldots], \mathbf{i}[S', \ldots]]$ reduces via label $\tau[S, S']$, then it enables *either* $[T{\to}\mathbf{io}x]$ *or* $[T{\to}\mathbf{io}]$, but *not* both rules. Finally, $\Gamma \vdash T \xrightarrow{\tau[\vee\mu]}{}^{*} T'$ holds when $T$ reaches $T'$ through a finite sequence of internal transitions $\tau[\vee]$ and $\tau[\mu]$, thus *excluding interaction*.

**Subject transition and type fidelity** Using the labelled semantics presented of Def. 12, we can prove a subject transition result that yields Thm. 11 as a corollary.

▶ **Theorem 14** (Subject transition). *Assume $\Gamma \vdash t : T$. If $\Gamma \vdash T$ type, then $\Gamma \vdash t \xrightarrow{\alpha} t'$ implies $\Gamma \vdash t' : T$. Otherwise, when $\Gamma \vdash T$ $\pi$-type, we have:*

1. *$\Gamma \vdash t \xrightarrow{\alpha} t'$ with $\alpha \in \{\tau[\neg x], \tau[\mathbf{if}\ x], \tau[x()], \tau[\lambda()], \tau_{[\mathrm{R}]} \mid x \in \mathbb{X}, {}_{[\mathrm{R}]} \neq {}_{[\text{R-Comm}]}\}$ implies $\Gamma \vdash t' : T$;*
2. *$\Gamma \vdash t \xrightarrow{\alpha} t'$ with $\alpha \in \{\overline{x}\langle w \rangle, x(w), \tau[x], \tau_{[\text{R-Comm}]} \mid x \in \mathbb{X}, w \in \mathbb{V} \cup \mathbb{X}\}$ implies either:*
   - **a.** *$\Gamma \vdash t' : T$ and $\mathbf{proc} \in T$;*
   - **b.** *$\alpha = \overline{x}\langle w \rangle$ and $\exists S, U, T' : \Gamma \vdash x : S, w : U, t' : T'$ and $\Gamma \vdash T \xrightarrow{\tau[\vee\mu]}{}^{*} \xrightarrow{\overline{S}\langle U \rangle} T'$;*
   - **c.** *$\alpha = x(w)$ and $\exists S, U, T' : \Gamma \vdash x : S, w : U, t' : T'$ and $\Gamma \vdash T \xrightarrow{\tau[\vee\mu]}{}^{*} \xrightarrow{S(U)} T'$;*
   - **d.** *$\alpha = \tau[x]$ and $\exists S, S', T' : \Gamma \vdash x : S, x : S', t' : T'$ and $\Gamma \vdash T \xrightarrow{\tau[\vee\mu]}{}^{*} \xrightarrow{\tau[S,S']} T'$;*
   - **e.** *$\alpha = \tau_{[\text{R-Comm}]}$ and $\exists S, S', T' : \{S, S'\} \notin \mathbb{X}$, $\Gamma \vdash t' : T'$ and $\Gamma \vdash T \xrightarrow{\tau[\vee\mu]}{}^{*} \xrightarrow{\tau[S,S']} T'$.*

Assume $\Gamma \vdash t : T$, with $t$ reducing to $t'$. Thm 14 says that when the reduction is caused by the functional fragment of $\lambda^{\pi\mathsf{D}}_{\leqslant}$ (hypothesis $\Gamma \vdash T$ type, or case 1), then $t'$ has the same type $T$. Instead, if the reduction is caused by input, output or interaction events (which means that $t$ is a process term, and $\Gamma \vdash T$ $\pi$-type), then we can observe a corresponding labelled transition in the type, possibly after some $\tau[\vee]/\tau[\mu]$ moves (cases 2b–2e); the only exception is case 2a: if $t'$ keeps the same type $T$, it means that $T$ syntactically contains $\mathbf{proc}$, which types a reducing sub-term of $t$ both before and after its reduction (via rule $[t{\text{-}}\leqslant]$).

We can also prove the opposite direction of Thm. 14: *if type $T$ interacts, then a typed term $t$ interacts accordingly.* This intuition holds under two conditions, leading to Thm. 15:

1. we only use *productive* $\lambda^{\pi\mathsf{D}}_{\leqslant}$ terms, i.e., all functions must be total (always return a value or process when applied). This means that, e.g., if $\Gamma \vdash t : \mathbf{o}[\underline{x}, \mathrm{int}, T']$, then $t$ *will* output on $x$; cases like $t = \mathbf{if}\ \omega\ \mathbf{then}\ \mathbf{send}(x, 42, t')\ \mathbf{else}\ \mathbf{send}(x, 43, t'')$ (where $\omega = (\lambda y.\, y\, y)\, (\lambda z.\, z\, z)$) are excluded. Productivity is obtained with many methods from literature (e.g., [11, 26]);
2. the input/output/interaction transitions of $T$ must operate on type variables: this allows to precisely relate them to occurrences of (open) variables in $t$.

▶ **Theorem 15** (Type fidelity). *Within a productive fragment of $\lambda^{\pi\mathsf{D}}_{\leqslant}$, assume $\Gamma \vdash t : T$. Then:*

1. *$\Gamma \vdash T \xrightarrow{\overline{x}\langle U \rangle} T'$ implies $\exists w, t' : \Gamma \vdash w : U, t' : T'$ and $\Gamma \vdash t \xrightarrow{\tau}{}^{*} \xrightarrow{\overline{x}\langle w \rangle} t'$;*
2. *$\Gamma \vdash T \xrightarrow{x(U)} T'$ implies $\forall w :$ if $\Gamma \vdash w : U$, then $\exists t' : \Gamma \vdash t' : T'$ and $\Gamma \vdash t \xrightarrow{\tau}{}^{*} \xrightarrow{x(w)} t'$;*

3. $\Gamma \vdash T \xrightarrow{\tau[\underline{x},\underline{x}]} T'$ *implies* $\exists t' :\ \Gamma \vdash t' : T'$ *and* $\Gamma \vdash t \xrightarrow{\tau}{}^{*} \xrightarrow{\tau[x]} t'$;
4. $\Gamma \vdash T \xrightarrow{\tau[\vee]}$ *implies either:* **a.** $\exists T' : \Gamma \vdash T \xrightarrow{\tau[\vee]} T'$ *and* $\Gamma \vdash t : T'$; *or,* **b.** $\exists t' : \Gamma \vdash t \xrightarrow{\alpha} t'$ *with* $\alpha \in \{\tau[\neg x], \tau[\mathbf{if}\ x], \tau[x()], \tau[\lambda()], \tau_{[\mathrm{R}]} \mid x \in \mathbb{X}, [\mathrm{R}] \neq [\text{R-Comm}]\}$ *and* $\Gamma \vdash t' : T$;
5. $\Gamma \vdash T \xrightarrow{\tau[\mu]} T'$ *implies* $\Gamma \vdash t : T'$.

Items 1–3 of Thm. 15 say that if $T$ can input/output/interact, then $t$ can do the same, possibly after a sequence of $\tau$-steps (without communication, cf. Def. 12); the $\tau$-sequence is finite, since $t$ is productive by hypothesis. By item 4, if $T$ can make a choice ($\vee$), then $t$ could have already chosen one option (case 4a), or could choose later (case 4b); productivity implies that $t$ *will* choose, after a *finite* number $\tau$-steps. Item 5 unfolds recursive types.

**From type verification to process verification** We now show that, by exploiting the correspondence between process and type reductions of Thm. 14 and 15, we can transfer (decidable) verification results from types to processes. To this purpose, we analyse the labelled transition systems (LTSs) of both types and processes using the linear-time $\mu$-calculus [10, §3] — chosen because it leads to Lemma 17 without relevant restrictions on our types.

▶ **Definition 16** (Linear-time $\mu$-calculus)**.** Given a *set of actions* $\mathbb{A}\mathrm{ct}$ ranged over by $\alpha$, the *linear-time $\mu$-calculus formulas* are defined as follows (where $\mathbb{A}$ is a subset of $\mathbb{A}\mathrm{ct}$):

*Basic formulas:* $\phi ::= \mathsf{Z} \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid (\alpha)\phi \mid \nu\mathsf{Z}.\phi$

*Derived formulas:* $\top \mid \bot \mid \phi_1 \vee \phi_2 \mid \phi_1 \Rightarrow \phi_2 \mid \mu\mathsf{Z}.\phi \mid (\mathbb{A})\phi \mid (-\mathbb{A})\phi \mid \phi_1 \,\mathsf{U}\, \phi_2 \mid \Box\phi \mid \Diamond\phi$

In Def. 16, a formula $\phi$ describes accepted sequences of actions; $\phi$ can be a variable $\mathsf{Z}$, negation, conjunction, prefixing $(\alpha)\phi$ ("accept a sequence if it starts with $\alpha$, and then $\phi$ holds"), or greatest fixed point $\nu\mathsf{Z}.\phi$. Basic formulas are enough [27, 3] to derive true/false (i.e., accept any/no sequence of actions), disjunction, implication, least fixed points $\mu\mathsf{Z}.\phi$; $(\mathbb{A})\phi$ accepts sequences that start with any $\alpha \in \mathbb{A}$, then satisfy $\phi$; $(-\mathbb{A})\phi$ is dual, as it requires $\alpha \in \mathbb{A}\mathrm{ct} \setminus \mathbb{A}$. We also derive usual temporal formulas $\phi_1 \,\mathsf{U}\, \phi_2$ ("$\phi_1$ holds, until $\phi_2$ eventually holds"), $\Box\phi$ ("$\phi$ is always true"), and $\Diamond\phi$ ("$\phi$ is eventually true"). Given a process $p$ with LTS semantics and labels $\mathbb{A}\mathrm{ct}$, a *run of* $p$ is a finite or infinite sequence of labels fired along a complete execution of $p$; we write $p \vDash \phi$ when $\phi$ accepts all runs of $p$. *(Details: §B)*

Now, we would like to check whether a term $t$ (with the LTS of Def. 12) satisfies a property $\phi$, that could be, e.g., a *safety property* $\Box(\neg\phi')$ ("$\phi'$ is never true while $t$ runs") or a *liveness property* $\Diamond\phi'$ ("$t$ will eventually satisfy $\phi'$"); however, the judgement $t \vDash \phi$ is generally undecidable, since $\phi$ has least/greatest fixpoints, and $\lambda^{\pi\mathsf{D}}_{\leqslant}$ is Turing-complete (even in its productive fragment, due to recursion and channel creation [4]). Luckily, we can decide $\phi$ on a *guarded type* $T$, as shown in Lemma 17 below. To this purpose, we instantiate $\mathbb{A}\mathrm{ct}$ in Def. 16 as $\mathbb{A}_\Gamma(T)$, which is the set of labels fired along $T$'s transitions in $\Gamma$, by Def. 13; notably, $\mathbb{A}_\Gamma(T)$ is *finite* and can be determined syntactically. *(Details: §B.2)*

▶ **Lemma 17.** *Given* $\Gamma$*, we say that* $T$ is guarded *iff, for all* $\pi$*-type subterms* $\mu\mathbf{t}.U$ *of* $T$*,* $\mathbf{t}$ *can occur in* $U$ *only as subterm of* $\mathbf{i}[...]$ *or* $\mathbf{o}[...]$*; then, if* $T$ *is guarded,* $T \vDash \phi$ *is decidable.*

Lemma 17 holds because guarded $\pi$-types are encodable in Petri nets (similarly to [12, §4.1]), for which linear-time $\mu$-calculus is decidable [10]. Notably, Lemma 17 covers *infinite-state* types (with $\mathbf{p}[...,...]$ under $\mu\mathbf{t}....$), that type $\lambda^{\pi\mathsf{D}}_{\leqslant}$ terms with unbounded parallel components.

We now show that, when $\Gamma \vdash t : T$, we can decide a linear-time $\mu$-calculus formula $\phi$ on $T$, and ensure that a related formula holds for $t$. This requires to take into account the loss of precision of type semantics w.r.t. process semantics, with the following intuitions:

(i1) if we do *not* want $t$ to ever perform some action on channel $x$, we must check that $T$ never performs a *potential use* of type variable $\underline{x}$;

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

We formalise such intuitions in three cases, in Thm. 19; but first, we need the tools of Def. 18.

▶ **Definition 18.** The *input / output uses of* $\underline{x}$ *by* $T$ *in* $\Gamma$, written $\mathbb{U}^{\mathsf{i}}_{\Gamma,\mathrm{T}}(\underline{x})$ / $\mathbb{U}^{\mathsf{o}}_{\Gamma,\mathrm{T}}(\underline{x})$, are:

$$\mathbb{U}^{\mathsf{i}}_{\Gamma,\mathrm{T}}(\underline{x}) = \{S'(U') \in \mathbb{A}_\Gamma(T) \mid \Gamma \vdash \underline{x} \leqslant S'\} \qquad \mathbb{U}^{\mathsf{o}}_{\Gamma,\mathrm{T}}(\underline{x}) = \{\overline{S'}\langle U'\rangle \in \mathbb{A}_\Gamma(T) \mid \Gamma \vdash \underline{x} \leqslant S'\}$$

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

$$\frac{\Gamma \vdash T \xrightarrow{\alpha} T' \quad \alpha \in \{\underline{x}(-), \overline{\underline{x}}\langle - \rangle \mid \underline{x} \in \mathbb{Y}\}}{T \upharpoonright_\Gamma \mathbb{Y} \xrightarrow{\alpha} T' \upharpoonright_\Gamma \mathbb{Y}} \qquad \frac{\Gamma \vdash t \xrightarrow{\alpha} t' \quad \alpha \in \{x(-), \overline{x}\langle - \rangle \mid x \in \mathbb{Y}\}}{t \upharpoonright_\Gamma \mathbb{Y} \xrightarrow{\alpha} t' \upharpoonright_\Gamma \mathbb{Y}}$$

▶ **Theorem 19.** *Within productive* $\lambda^{\pi\mathsf{D}}_{\leqslant}$*, assume* $\Gamma \vdash t : T$*, with* $\Gamma \vdash T$ $\pi$-type, $\mathbf{proc} \notin T$*. For* $\mu$*-calculus judgements on* $T$*, let* $\mathbb{A}\mathrm{ct} = \mathbb{A}_\Gamma(T)$*, and* $\mathbb{A}_\tau = \{\tau[S, S'] \in \mathbb{A}_\Gamma(T) \mid \{S, S'\} \notin \mathrm{dom}(\Gamma)\}$.

1. (Non-usage) *The judgement* $t \upharpoonright_\Gamma \{x_i\}_{i\in 1..n} \vDash \Box(\neg(\bigvee_{i\in 1..n}(\{\overline{x_i}\langle w\rangle \mid w \in \mathbb{V}\cup\mathbb{X}\})\top))$ *holds if* $T \upharpoonright_\Gamma \{\underline{x_i}\}_{i\in 1..n} \vDash \Box\big(\neg\big(\bigvee_{i\in 1..n}(\mathbb{U}^{\mathsf{o}}_{\Gamma,\mathrm{T}}(\underline{x_i}))\top\big)\big)$
2. (Eventual usage) *The judgement* $t \upharpoonright_\Gamma \{x_i\}_{i\in 1..n} \vDash \Diamond(\bigvee_{i\in 1..n}(\{\overline{x_i}\langle w\rangle \mid w \in \mathbb{V}\cup\mathbb{X}\})\top)$ *holds if* $T \upharpoonright_\Gamma \{\underline{x_i}\}_{i\in 1..n} \vDash (-\mathbb{A}_\tau)\top \,\mathsf{U}\, (\bigvee_{i\in 1..n}(\{\overline{\underline{x_i}}\langle U\rangle \mid \overline{\underline{x_i}}\langle U\rangle \in \mathbb{U}^{\mathsf{o}}_{\Gamma,\mathrm{T}}(\underline{x_i})\})\top)$
3. (Forwarding) $t \upharpoonright_\Gamma \{x, y, z\} \vDash \Box((\{x(z)\})\top \Rightarrow ((-\{x(w) \mid w \in \mathbb{V}\cup\mathbb{X}\})\top \,\mathsf{U}\, (\overline{y}\langle z\rangle)\top))$ *holds if* $T \upharpoonright_\Gamma \{\underline{x}, \underline{y}, \underline{z}\} \vDash \Box\big((\{S(\underline{z}) \mid S(\underline{z}) \in \mathbb{U}^{\mathsf{i}}_{\Gamma,\mathrm{T}}(\underline{x})\})\top \Rightarrow \big((-(\mathbb{A}_\tau \cup \mathbb{U}^{\mathsf{i}}_{\Gamma,\mathrm{T}}(\underline{x})))\top \,\mathsf{U}\, (\overline{\underline{y}}\langle \underline{z}\rangle)\top\big)\big)$

Item 1 of Thm. 19 can be seen as a case of intuition (i1) above: if $T$ never fires a label ($\Box(\neg\ldots)$) that is a *potential output use* of $\underline{x_i}$ ($i \in 1..n$), then $t$ never uses $x_i$ for output. The "potential output use", by Def. 18, is any label $\overline{S'}\langle U'\rangle$ fired by $T$ where $S'$ is a supertype of $\underline{x}$: this accounts for "imprecise typing", discussed in Ex. 10. Note that item 1 limits the observed inputs/outputs of $T/t$ to $x_i$, using $\upharpoonright_\Gamma \{x_i\}_{i\in 1..n}$ (Def. 18): other (open) channels are ignored, and can only reduce by synchronising. Item 2 of Thm. 19 is a case of intuition (i2): to ensure that $t$ eventually outputs on $x_i$ ($i \in 1..n$), we check that $T$ eventually fires a label $\overline{\underline{x}}\langle U\rangle$; moreover, we check $T$ does *not* fire any label in $\mathbb{A}_\tau$, until ($\mathsf{U}$) the output $\overline{\underline{x}}\langle U\rangle$ occurs. The set $\mathbb{A}_\tau$ contains all "imprecise" synchronisation labels $\tau[S, S']$ where either $S$ or $S'$ is *not* a type variable: we exclude such labels because, if $T$ fires one, then we cannot use Thm. 15(3) to ensure that $t$ reduces accordingly; i.e., if we do *not* exclude $\mathbb{A}_\tau$, then $t$ might deadlock and never perform $\overline{x_i}\langle w\rangle$. Finally, item 3 combines the intuitions of both previous cases: we want to ensure that whenever $t$ receives $z$ on channel $x$, then it eventually forwards $z$ through channel $y$, without doing other inputs on $x$ before; to this purpose, we check that whenever $T$ inputs $\underline{z}$ on a channel $S$ (representing a *potential* use of $\underline{x}$), then $T$ eventually fires $\overline{\underline{y}}\langle \underline{z}\rangle$ — without doing *potential* inputs on $\underline{x}$, nor firing any label in $\mathbb{A}_\tau$, before.

▶ **Example 20.** We can use Thm. 19 to verify that a typed $\lambda^{\pi\mathsf{D}}_{\leqslant}$ function implements a desired behaviour. Take $T_{fwd}$ (Ex. 7): letting $\Gamma = i{:}\mathrm{c}^{\mathrm{i}}[\mathrm{str}], o{:}\mathrm{c}^{\mathrm{o}}[\mathrm{str}], z{:}\mathrm{str}$, we apply Thm. 19(3) to: *(1)* verify that the "body" of $T_{fwd}$, $T = \mu\mathbf{t}.\mathbf{i}[\underline{i}, \Pi(\underline{z'}{:}\mathrm{str})\mathbf{o}[\underline{o}, \underline{z'}, \Pi()\mathbf{t}]]$, forwards $\underline{z}$ through $\underline{o}$, when received on $\underline{i}$; and *(2)* conclude that all functions of type $T_{fwd}$ yield a $T$-typed process with the desired forwarding behaviour; one such functions is *fwd* in Ex. 2. Also, with small variations of the formulas of Thm. 19(3), we can decide that all $T_m$-typed functions (§1, Ex. 8) yield processes that eventually forward "with a choice"; hence, all typed mobile code received via channel *cm* has this property. Similarly, in Ex. 9 we can prove that any $T_{p12}$-typed term (including $t_{12}$) eventually outputs on a channel $z'$, after $z'$ is received via $x$.

▶ **Example 21** (Channel aliasing)**.** The verification approach above assumes that *distinct channel-typed variables represent distinct channel instances*. E.g., assume $\Gamma \vdash t : T$ with $t = \mathbf{send}(x, 42, t_1) \parallel \mathbf{recv}(y, t_2)$ and $T = \mathbf{p}\big[\mathbf{o}[\underline{x}, \mathrm{int}, T_1], \mathbf{i}\big[\underline{y}, T_2\big]\big]$: Def. 12 and 13 do *not* let $t$ and $T$ reduce by synchronising, since $x \neq y$; hence, the $\mu$-calculus analysis of $t$ and $T$ does

not "see" any communication. However, in the well-typed term $t' = (\mathbf{let}\ y = x\ \mathbf{in}\ t)$, term $t$ *does* communicate, because $y$ becomes an alias of $x$. Still, we can correctly analyse $t'$, because $t$ and $t'$ *have different types*: the latter has type $T' = \mathbf{p}[\mathbf{o}[\underline{x}, \mathrm{int}, T'], \mathbf{i}[\underline{x}, T'']]$. This is because rule [$t$-**let**] reflects aliasing through the type-level substitution $T' = T\{\underline{x}/\underline{y}\}$ (also seen in Ex. 10): hence, we correctly detect the communication in $t'$ and $T'$. The same type-level substitution occurs in rule [$t$-APP], and thus, terms $t_f\, x\, x$ and $t_f\, x\, y$ have different types; and the same mechanism tracks aliased channels across communications (since $\mathbf{send}(z, x, ...)$ / $\mathbf{send}(z, y, ...)$ have different types, and $\underline{x}/\underline{y}$ are substituted in the receiver's type).

## 5 Conclusion and Related Work

We presented a typing discipline for higher-order concurrent programs, allowing to decide safety/liveness properties of processes by checking the LTS of types. The cornerstones are behavioural *process types* leveraging *channel types*, *subtyping*, and *dependent function types*.

**Implementation** Our theory relies on subtyping and dependent function types, chosen to closely correspond to (a fragment of) Scala/Dotty, and their foundation System $D_{<:}$ [1]. A payoff of this choice is that we can implement $\lambda^{\pi\mathsf{D}}_{\leqslant}$ as a *deeply-embedded domain-specific language*, by reusing the existing Scala syntax and type system, defining new functions for the concurrency primitives (**send**, **recv**, ‖, **end**), and new classes for their types (**o**[...], **i**[...], **p**[...], **nil**) — with the well-formedness and subtyping constraints in Fig. 4. The resulting programs look like Fig. 1; the types are isomorphic, with type $\underline{x}$ rendered as `x.type` — e.g., $\Pi(\underline{x}{:}T)\mathbf{o}[\underline{y}, \underline{x}, T']$ becomes `(x:T) => Out[y.type, x.type, T']`. Thus, the Scala compiler can check the syntax and types of programs (§2 and §3), ensuring type safety (Thm. 11). With this strategy, we implemented a working prototype, available in [24]. A next step is implementing the type-based analysis in §4: it requires writing a compiler plugin.

**Related work** [28] introduced value dependencies for *session types*, based on their linear logical foundations; their types can state properties of exchanged data values. [31] extends the theory, combining processes and higher-order functions, through a contextual monad introduced in [29]. All of the above calculi are inherently *deterministic* and *strongly normalising*; instead, typed $\lambda^{\pi\mathsf{D}}_{\leqslant}$ processeses can be non-deterministic; e.g., our type $T = \mathbf{p}[\mathbf{p}[\mathbf{o}[\underline{x}, \underline{y}, T], \mathbf{o}[\underline{x}, \underline{z}, T']], \mathbf{i}[\underline{x}, \Pi(\underline{z'}{:}\mathrm{c}^{\mathrm{io}}[\mathrm{int}])U]]$ types parallel processes with two concurrent outputs on $\underline{x}$, and the LTS semantics of $T$ captures that either $\underline{y}$ or $\underline{z}$ could replace $\underline{z'}$ in the $U$-typed continuation. [2] studies a non-conservative extension of linear logic-based session types with sharing, allowing non-determinism. Their work includes dependent quantification with shared channels, but their type syntax does *not* include free type variables, so the actual type dependencies do not arise (see [2, 37:8]).

Multiparty *indexed* session types are studied in [8], and implemented as a protocol description language in [21]: indexed types can represent an arbitrary number of session *participants*. [30] extends [28] to multiparty sessions, treating value dependency across participants. None of [8, 30] considers higher-order processes.

[34] studies a dependent type system where the type of processes is a mapping $\Delta$ from channels $x$ to channel types $T$. The dependency is specified as $\Pi(x{:}T)\Delta$, representing a channel abstraction of the environment. This notion is extended to an existential channel dependency type $\Sigma(x{:}T)\Delta$ to address fresh name creation [32, 13]. As discussed in §1, [34, 32, 13] study channel dependencies from resources in the environment; instead, our dependent types model process behaviour and channel usage. The behavioural nature of our types reminds [15], that also uses LTS-based type analysis; but unlike our work, [15] does *not* track type-level dependencies on received values, *nor* addresses higher-order interaction.

## References


1. Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 249–272, 2016. doi:10.1007/978-3-319-30936-1_14.
2. Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. *PACMPL*, 1(ICFP):37:1–37:29, 2017.
3. Julian Bradfield and Colin Stirling. Modal $\mu$-calculi. In Patrick Blackburn, Johan Van Benthem, and Frank Wolter, editors, *Handbook of Modal Logic*, volume 3 of *Studies in Logic and Practical Reasoning*, pages 721 – 756. Elsevier, 2007. doi:https://doi.org/10.1016/S1570-2464(07)80015-2.
4. Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. On the expressive power of recursion, replication and iteration in process calculi. *Mathematical Structures in Computer Science*, 19(6):1191–1222, 2009. doi:10.1017/S096012950999017X.
5. L. Cardelli, S. Martini, J.C. Mitchell, and A. Scedrov. An extension of system f with subtyping. *Information and Computation*, 109(1):4 – 56, 1994. doi:https://doi.org/10.1006/inco.1994.1013.
6. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985. doi:10.1145/6041.6042.
7. Davide Ancona *et al.* Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages*, 3(2-3), 2017. doi:10.1561/2500000031.
8. Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. *Logical Methods in Computer Science*, 8(4), 2012. URL: http://dx.doi.org/10.2168/LMCS-8(4:6)2012, doi:10.2168/LMCS-8(4:6)2012.
9. Ericsson. The Erlang/OTP language and toolkit, 2018. http://erlang.org/.
10. Javier Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2), Feb 1997. doi:10.1007/s002360050074.
11. Jürgen Giesl, Matthias Raffelsieper, Peter Schneider-Kamp, Stephan Swiderski, and René Thiemann. Automated termination proofs for haskell by term rewriting. *ACM Trans. Program. Lang. Syst.*, 33(2):7:1–7:39, February 2011. doi:10.1145/1890028.1890030.
12. Ursula Goltz. CCS and Petri nets. In Irène Guessarian, editor, *Semantics of Systems of Concurrent Processes*, pages 334–357, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.
13. Matthew Hennessy, Julian Rathke, and Nobuko Yoshida. safeDpi: a language for controlling mobile code. *Acta Inf.*, 42(4-5):227–290, 2005.
14. Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 128–141, New York, NY, USA, 2001. ACM. doi:10.1145/360204.360215.
15. Atsushi Igarashi and Naoki Kobayashi. A generic type system for the $\pi$-calculus. *TCS*, 311(1), 2004. doi:http://dx.doi.org/10.1016/S0304-3975(03)00325-6.
16. Alan Jeffrey. A symbolic labelled transition system for coinductive subtyping of $F_{\mu<}$ types. In *LICS*, 2001. doi:10.1109/LICS.2001.932508.
17. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977. doi:10.1109/TSE.1977.229904.
18. Lightbend, Inc. The Akka toolkit and runtime, 2018. http://akka.io/.
19. Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *TOPLAS*, 16(6), 1994. doi:10.1145/197320.197383.
20. Heather Miller, Philipp Haller, and Martin Odersky. Spores: A type-based foundation for closures in the age of concurrency and distribution. In *ECOOP 2014 - Object-Oriented*

*Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, pages 308–333, 2014. doi:10.1007/978-3-662-44202-9_13.

21 Nicholas Ng and Nobuko Yoshida. Pabble: parameterised Scribble. *Service Oriented Computing and Applications*, 9(3-4):269–284, 2015.

22 Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.

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

24 Alceste Scalas. `effpi`: typed concurrent programming based on $\lambda_{\leqslant}^{\pi\mathsf{D}}$ (prototype), 2018. Git repository. URL: `https://www.doc.ic.ac.uk/~ascalas/effpi/effpi.git`.

25 Alceste Scalas and Nobuko Yoshida. Dependent Function Types for Higher-Order Interaction, 2018. Technical report. URL: `https://www.doc.ic.ac.uk/~ascalas/effpi/tech-report.pdf`.

26 Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total haskell is reasonable coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 14–27, New York, NY, USA, 2018. ACM. doi:10.1145/3167092.

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

28 B. Toninho, L. Caires, and F. Pfenning. Dependent session types via intuitionistic linear type theory. In *PPDP'11*, pages 161–172, 2011.

29 Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP*, pages 350–369, 2013.

30 Bernardo Toninho and Nobuko Yoshida. Certifying data in multiparty session types. *Journal of Logical and Algebraic Methods in Programming*, 90(C):61–83, 2017.

31 Bernardo Toninho and Nobuko Yoshida. Depending on session-typed processes. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 128–145, Cham, 2018. Springer International Publishing.

32 Nobuko Yoshida. Channel dependent types for higher-order mobile processes. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 147–160, 2004. doi:10.1145/964001.964014.

33 Nobuko Yoshida and Matthew Hennessy. Assigning types to processes. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*, pages 334–345, 2000. doi:10.1109/LICS.2000.855782.

34 Nobuko Yoshida and Matthew Hennessy. Assigning types to processes. *Information and Computation*, 174(2):143 – 179, 2002. doi:https://doi.org/10.1006/inco.2002.3113.

# A $\lambda^{\pi\mathsf{D}}_{\leqslant}$-Calculus and Type System

The definition below is reprised from [16, §2].

▶ **Definition 22** (Positive/negative position of a type variable). We define the *polarised free variables of* $T$, written $\mathrm{fv}^+(T)$ and $\mathrm{fv}^-(T)$, as follows:

$$
\begin{array}{rcl}
\mathrm{fv}^{\pm}(\top) & = & \varnothing \\
\mathrm{fv}^{\pm}(\bot) & = & \varnothing \\
\mathrm{fv}^{\pm}(\mathrm{bool}) & = & \varnothing \\
\mathrm{fv}^{\pm}(\bullet) & = & \varnothing \\
\mathrm{fv}^{\pm}(\mathrm{c}[T,U]) & = & \mathrm{fv}^{\pm}(T) \cup \mathrm{fv}^{\pm}(U) \\
\mathrm{fv}^{\pm}(\mathbf{nil}) & = & \varnothing \\
\mathrm{fv}^{\pm}(\mathbf{i}[S,T]) & = & \mathrm{fv}^{\pm}(S) \cup \mathrm{fv}^{\pm}(T) \\
\mathrm{fv}^{\pm}(\mathbf{o}[S,T,U]) & = & \mathrm{fv}^{\pm}(S) \cup \mathrm{fv}^{\pm}(T) \cup \mathrm{fv}^{\pm}(U) \\
\mathrm{fv}^{\pm}(\mathbf{p}[T,U]) & = & \mathrm{fv}^{\pm}(T) \cup \mathrm{fv}^{\pm}(U) \\
\mathrm{fv}^{\pm}(T \vee U) & = & \mathrm{fv}^{\pm}(T) \cup \mathrm{fv}^{\pm}(U) \\
\mathrm{fv}^{\pm}(\Pi(\underline{x}{:}T)U) & = & \mathrm{fv}^{\mp}(T) \cup (\mathrm{fv}^{\pm}(U) \setminus \underline{x}) \\
\mathrm{fv}^{\pm}(\mu\underline{x}.T) & = & \mathrm{fv}^{\pm}(T) \setminus \underline{x} \\
\mathrm{fv}^{+}(\underline{x}) & = & \{\underline{x}\} \\
\mathrm{fv}^{-}(\underline{x}) & = & \varnothing
\end{array}
$$

# 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 [10, §3], and [27, 3].

▶ **Definition 23** (Words over a set). Given a set $\mathbb{Y}$, we define $\mathbb{Y}^*$ and $\mathbb{Y}^\omega$ as the sets of finite and infinite words over $\mathbb{Y}$, respectively; we also define $\mathbb{Y}^\infty = \mathbb{Y}^* \cup \mathbb{Y}^\omega$. Given a word $\sigma = \alpha_1\alpha_2\alpha_3\ldots \in \mathbb{Y}^\infty$, we define $\mathsf{hd}(\sigma) = \alpha_1$, and $\mathsf{tl}(\sigma) = \alpha_2\alpha_3\ldots$; we denote the empty word as $\epsilon$, and leave $\mathsf{hd}(\epsilon)$ and $\mathsf{tl}(\epsilon)$ undefined.

▶ **Definition 24** (Semantics). Given a set of actions $\mathbb{Act}$, a *valuation* $\mathcal{V}$ is a partial mapping from propositional variables to sets of words over $\mathbb{Act}$ — i.e., if $\mathsf{Z} \in \mathrm{dom}(\mathcal{V})$, then $\mathcal{V}(\mathsf{Z}) \in \mathbb{Act}^\infty$; given a set of words $\mathbb{W} \subseteq \mathbb{Act}^\infty$, let $\mathcal{V}\{\mathbb{W}/\mathsf{z}\}$ be the valuation such that $\mathcal{V}\{\mathbb{W}/\mathsf{z}\}(\mathsf{Z}) = \mathbb{W}$ and $\mathcal{V}\{\mathbb{W}/\mathsf{z}'\}(\mathsf{Z}') = \mathcal{V}(\mathsf{Z}')$ (when $\mathsf{Z}' \neq \mathsf{Z}$). The *denotation of a linear-time $\mu$-calculus formula $\phi$ under valuation $\mathcal{V}$*, written $\|\phi\|_{\mathcal{V}}$, is the set of words of $\mathbb{Act}^\infty$ inductively defined as:

$$
\begin{array}{rcl}
\|\mathsf{Z}\|_{\mathcal{V}} & = & \mathcal{V}(\mathsf{Z}) \\
\|\neg\phi\|_{\mathcal{V}} & = & \mathbb{Act}^\infty \setminus \|\phi\|_{\mathcal{V}} \\
\|\phi_1 \wedge \phi_2\|_{\mathcal{V}} & = & \|\phi_1\|_{\mathcal{V}} \cap \|\phi_2\|_{\mathcal{V}} \\
\|(\alpha)\phi\|_{\mathcal{V}} & = & \{\sigma \in \mathbb{Act}^\infty \mid \mathsf{hd}(\sigma) = \alpha \text{ and } \mathsf{tl}(\sigma) \in \|\phi\|_{\mathcal{V}}\} \\
\|\nu\mathsf{Z}.\phi\|_{\mathcal{V}} & = & \bigcup \left\{ \mathbb{W} \subseteq \mathbb{Act}^\infty \;\middle|\; \mathbb{W} \subseteq \|\phi\|_{\mathcal{V}\{\mathbb{W}/\mathsf{z}\}} \right\}
\end{array}
$$

Given a *labelled transition system* $\mathcal{T}$ with *initial state* $s_0$ and *labels* in $\mathbb{Act}$, we say that $\mathcal{T}$ *satisfies* $\phi$, written $\mathcal{T} \vDash \phi$, iff every run[1] of $\mathcal{T}$ belongs to $\|\phi\|_{\varnothing}$.

[1] A run of $\mathcal{T}$ 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.

▶ **Definition 25** (Extended constructs). Using the basic linear-time $\mu$-calculus productions (left-hand side of Def. 16), we define the following extended formulas (right-hand side of Def. 16),

| Formula | Definition | Description |
|---|---|---|
| $\top$ | $\nu \mathsf{Z}.\mathsf{Z}$ | true (denotation is $\mathbb{A}\mathrm{ct}^\infty$) |
| $\bot$ | $\neg\top$ | false (denotation is $\varnothing$) |
| $\phi_1 \vee \phi_2$ | $\neg(\neg\phi_1 \wedge \neg\phi_2)$ | $\phi_1$ holds, or $\phi_2$ holds |
| $\phi_1 \Rightarrow \phi_2$ | $\neg\phi_1 \vee \phi_2$ | if $\phi_1$ holds, then $\phi_2$ holds |
| $\mu \mathsf{Z}.\phi$ | $\neg\nu \mathsf{Z}.\neg\phi\{^{\neg\mathsf{Z}}/_{\mathsf{Z}}\}$ | least fixed point (denotation is a set of words of finite length) |
| $(\mathbb{A})\phi$ | $\bigvee_{\alpha\in\mathbb{A}} (\alpha)\phi$ | after some action $\alpha$ in $\mathbb{A}$, $\phi$ holds |
| $(-\mathbb{A})\phi$ | $\bigvee_{\alpha\in(\mathbb{A}\mathrm{ct}\setminus\mathbb{A})} (\alpha)\phi$ | after some action $\alpha$ *not* in $\mathbb{A}$, $\phi$ holds |
| $\phi_1 \,\mathsf{U}\, \phi_2$ | $\mu \mathsf{Z}.\phi_2 \vee (\phi_1 \wedge (\mathbb{A}\mathrm{ct})\mathsf{Z})$ | $\phi_1$ holds (for a finite number of actions), until $\phi_2$ holds |
| $\Diamond\phi$ | $\top \,\mathsf{U}\, \phi$ | $\phi$ eventually holds, after a finite number of actions |
| $\Box\phi$ | $\neg\Diamond(\neg\phi)$ | $\phi$ always holds |

## B.2 Actions of a type

The following is the definition of the set of actions $\mathbb{A}_\Gamma(T)$, that completes Def. 18.

▶ **Definition 26** (Actions of a $\pi$-type). The *basic actions of a $\pi$-type* in $\Gamma$ are defined as:

$$\mathbb{B}_\Gamma(\mathbf{nil}) = \mathbb{B}_\Gamma(\mathbf{t}) = \varnothing \qquad \mathbb{B}_\Gamma(\mu\mathbf{t}.T) = \{\tau[\mu]\} \cup \mathbb{B}_\Gamma(T) \qquad \mathbb{B}_\Gamma(\mathbf{p}[T,U]) = \mathbb{B}_\Gamma(T) \cup \mathbb{B}_\Gamma(U)$$
$$\mathbb{B}_\Gamma(T \vee U) = \{\tau[\vee]\} \cup \mathbb{B}_\Gamma(T) \cup \mathbb{B}_\Gamma(U) \qquad \mathbb{B}_\Gamma(\mathbf{o}[S,T,\Pi()U]) = \{\overline{S}\langle T\rangle\} \cup \mathbb{B}_\Gamma(U)$$

$$\mathbb{B}_\Gamma(\mathbf{i}[S,\Pi(\underline{x}{:}T)U]) = \{S(T') \mid T' \in \mathbb{Y}\} \cup \{\mathbb{B}_\Gamma(U\{^{T'}/_{\underline{x}}\}) \mid T' \in \mathbb{Y}\} \quad \text{where } \mathbb{Y} = \left\{ T' \;\middle|\; \begin{array}{l} \Gamma \vdash T' \leqslant T \text{ and} \\ (T' = T \text{ or } T' \in \mathbb{X}) \end{array} \right\}$$

The *(complete) actions of a $\pi$-type* in $\Gamma$ are defined as:

$$\mathbb{A}_\Gamma(T) \;=\; \mathbb{B}_\Gamma(T) \cup \left\{ \tau[S,S'] \;\middle|\; \overline{S}\langle U\rangle \in \mathbb{B}_\Gamma(T) \text{ and } S'(U') \in \mathbb{B}_\Gamma(T) \text{ and } \Gamma \vdash S \bowtie S' \right\}$$

Intuitively, Def. 26 computes the possible actions of a $\pi$-type $T$ in two steps:

1. first, it computes the set of basic actions $\mathbb{B}_\Gamma(T)$, by performing a simple syntactic traversal of $T$. Some care is required to compute the actions of an input type $T_{in} = \mathbf{i}[S,\Pi(\underline{x}{:}T)U]$, that by Def. 13, 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 [$T{\to}\mathbf{i}$], are collected in the set $\mathbb{Y}$. Note that $\mathbb{Y}$ is always finite: it can contain *at most* $T$ and all variables in $\Gamma$;
   b. then, for each $T' \in \mathbb{Y}$, the action $S(T')$ is added to $\mathbb{B}_\Gamma(T_{in})$, together with the basic actions of the continuation $U\{^{T'}/_{\underline{x}}\}$;
2. then, it computes the (complete) set of actions $\mathbb{A}_\Gamma(T)$ by combining:
   a. $\mathbb{B}_\Gamma(T)$, and
   b. all possible communication actions $\tau[S,S']$ obtainined by pairing the actions in $\mathbb{B}_\Gamma(T)$, whenever they involve channel types that might communicate ("$\Gamma \vdash S \bowtie S'$", Def. 13).

Notably, to compute $\mathbb{A}_\Gamma(T)$ we need to compare types via subtyping, and thus, we need the judgement $\Gamma \vdash U \leqslant U'$ to be decidable (hence the remark about rule [$\leqslant$-$\Pi$] in §3).