Dynamic Channel Screening in the Higher Order π-Calculus

José Luis Vivas and Nobuko Yoshida

Dept. of Mathematics and Computer Science, University of Leicester, Leicester, UK

Abstract. Recently programming languages have been designed to support mobile code, i.e. higher-order code that is transferred from a remote location or domain and executed within the local environment. This may expose the internal interfaces and objects within a location to attacks by mobile code. In this work, we propose an extension of notations based on the Higher-Order π-calculus with a primitive operator whose role is to protect internal interfaces by dynamically restricting the visibility of channels. The usefulness of these operators is illustrated by applications involving resource access control. We show how restrictions on the behaviour of processes based on the notion of process type [16], and intended to be checked statically, can be enforced dynamically with the aid of the filter operator.

1 Introduction

Recently programming languages have been designed to support mobile code, i.e. higher-order code that is transferred from a remote location or domain and executed within the local environment. This may expose the internal interfaces and objects within a location to attacks by mobile code. There are several techniques for securing mobile code. These techniques are usually based on the administration of access rights and capabilities assigned to processes, and are enforced either statically (during compilation), or dynamically (at runtime).

The technique for statically checking mobile code is based on the notion of typing and type safety. However, in current systems access restrictions are as a rule enforced dynamically by runtime monitoring of process execution. Static checking of properties is usually not sufficient to ensure safety. In Java, for instance, there is a scheme called the Sandbox model of protection against mobile code [9]. This model includes both static and dynamic checking. The type system of Java is used basically as an aid to the security manager, a kind of reference monitor: if the type system is correct, the security manager will be safe. The access control mechanism must nevertheless be ultimately performed by the security manager at runtime. The conclusion is that access control safety can at best be enforced by a mix of static and dynamic techniques.

In the π-calculus and its higher-order extensions, access rights have been largely encoded in terms of types, either channel types denoting capabilities [5, 8, 11], or process types denoting the interface of a process [16]. This discrepancy might be a result of the limitations in the expressiveness of the π-calculus with regard to the notion of locality and to restriction of channel visibility. Apparently these notions can only be encoded if the basic calculus is extended with higher order constructs and locality primitives [1, 4, 5].

The reduction of higher order features to first order ones [12] in the π-calculus is based on the claim that only access to agents need be transmitted, not the agents themselves. This works only because notions of locality are absent from the calculus [13]. In the presence of localities, however, we need to distinguish between agent passing and agent execution. Agents may always be passed by reference, but retrieving an agent or a piece of code from some place in memory and executing it in a determined location - common in object-oriented languages and systems - is a typically higher order operation that cannot be easily reduced to reference passing.

The objective of this work is to investigate how notations based on the π-calculus can be extended with channel screening operators encoding the notion of locality and dynamic binding. We extend the π-calculus with a new operator called polarised filter, or simply filter. The filter operator allows a process within its scope to communicate with the environment only along the channels included in a given finite set of polarised channels. A related type of screening operator is the blocking operator presented in [3, 13], which can also be refined by the addition of polarity [6].

Screening operators have a dual nature. By reducing the visibility of a channel these operators block external communication but permit local communication along screened channels, thus contextualising the meaning of channel names. As a result, they may be used to encode the notions of name space (cf. Java's dynamic linking of applets [14]), encapsulation, and dynamic binding of names, common features in distributed an object-based systems. In fact, it was very early noted that in the object-oriented paradigm "the idea of encapsulation can be modeled by restricting the visibility of operators" (Abramsky [10]). We claim that this is what screening operators do. The restriction operator of the π-calculus does this very imperfectly because of the phenomenon of channel extrusion. Restricting the visibility of names cannot be reduced to the creation of fresh names. Invisible channels may still be named or referenced, cf. method calls in objects, at the same time that encapsulation is guaranteed. The restricted channels of the π-calculus, however, can be extruded, maybe by accident, thus exposing the internal interface.

The usefulness of screening operators are illustrated by applications involving resource access control. We show how type-based restrictions on the behaviour of a process can be encoded with the aid of the filter operator. To this purpose we analyse in detail a typed higher-order calculus based on the λ and the π-calculi, the $\lambda\pi_{\mathrm{v}}$-calculus [16]. In this calculus, the typing system can assign non-trivial *interface* types to many higher-order π-processes, and functional dependent types to arbitrary higher-order abstracted processes. We introduce a simple untyped higher-order π-calculus extended with the filter operator, called the $\lambda\pi\mathrm{F}_{\mathrm{v}}$-calculus, and show how safety requirements can be encoded in the

$\lambda\pi\mathrm{F_v}$-calculus by providing a security preserving translation of processes from a subset of the $\lambda\pi_\mathrm{v}$-calculus to the $\lambda\pi\mathrm{F_v}$-calculus. Based on this result, we can offer an integrated framework to mix both static and dynamic checking coherently.

This paper is organised as follows. In Section 2, we introduce the filter operator in the context of the higher order π-calculus. In Section 3 we give a brief introduction to the $\lambda\pi_\mathrm{v}$-calculus. In Section 4 we present the $\lambda\pi\mathrm{F_v}$-calculus. In Section 5, we show the translation from the $\lambda\pi_\mathrm{v}$-calculus to the $\lambda\pi\mathrm{F_v}$-calculus and analyse its adequacy with regard to safety requirements and process behaviour. In Section 6 we present some conclusions.

2 The Filter Operator

For simplicity we start by introducing the polarised filter operator as an extension to the Higher Order (HO)π-calculus. The role of the filter operator is to block every action which is not explicitly included in a finite set of polarised channel names. We write $P\lceil L$ to denote the process P filtered by L, a finite set of polarised channel names. A positive polarised channel name z is written z^+, and a negative polarised channel name z is written z^-. The meaning of the filter operator is given by the rule:

$$\textbf{FILT:}\ \frac{P \xrightarrow{\mu} P'}{P\lceil L \xrightarrow{\mu} P'\lceil L} \quad \begin{array}{l}\mu \text{ an output along a channel } z \text{ s.t. } z^+ \in L, \text{ or} \\ \mu \text{ an input along a channel } z \text{ s.t. } z^- \in L\end{array}$$

If both $z^+ \in L$ and $z^- \in L$, we write $z \in L$.

2.1 An Example: The Java's Sandbox Model

We show an example in a kind of pseudocode based on the HOπ-calculus, including integer types and integer matching.

Screening primitives like the filter operator are useful for encoding security arrangements such as Java's Sandbox model. The idea behind this model is to provide a safe environment within which unreliable programs hosted by a computer may be confined. Depending on the level of trust associated with it, an imported program may be allowed to access only a restricted number of resources. Different programs may be given different environments, each one defining a distinct set of capabilities for the processes confined within its bounds.

Environments are defined by a filter set, each set establishing the resources that may be accessed. We illustrate this idea by the following example extracted from [9]. A number of different-sized sandboxes are defined. The following resources are part of the system: a CPU *CPU*, a screen *Screen*, a keyboard *Keyboard*, a mouse *Mouse*, the memory *Memory*, a web server *WebServer*, and a local file system *Docs*. Each one of these resources may be defined by an agent guarded by a corresponding channel. Thus, the CPU can be encoded as a replicated input process CPU $\stackrel{\mathrm{def}}{=} *\mathrm{cpu}?(\widetilde{X}).\mathrm{Cpu}$, where the channel "cpu" provides access to the resource and "Cpu" denotes the process representing the CPU.

Similarly, the screen can be encoded by Screen $\stackrel{\text{def}}{=} *\mathrm{scr}?(\widetilde{Y}).\mathrm{Scr}$. The other resources are defined similarly. We assume that incoming code is abstracted on a channel denoting the internal reference to the process, i.e. $\lambda r.P$. Before execution this code is applied to a fresh name.
Four different sandboxes are given:

- Sandbox1: access to the CPU, screen, keyboard, mouse and own memory.
- Sandbox2: access to the CPU, own memory and web server.
- Sandbox3: access to the CPU, own memory, web server, and local files.
- Sandbox4: access to everything (an open sandbox).

The sandboxes can be defined by the following filter sets:

$$\begin{aligned}
\mathrm{Sandbox1} &::= \{\mathrm{cpu}^+, \mathrm{scr}^+, \mathrm{kb}^+, \mathrm{ms}^+, \mathrm{mem}^+, \mathrm{r}^-\}\\
\mathrm{Sandbox2} &::= \{\mathrm{cpu}^+, \mathrm{mem}^+, \mathrm{ws}^+, \mathrm{r}^-\}\\
\mathrm{Sandbox3} &::= \{\mathrm{cpu}^+, \mathrm{mem}^+, \mathrm{ws}^+, \mathrm{docs}^+, \mathrm{r}^-\}\\
\mathrm{Sandbox4} &::= \text{no filter}
\end{aligned}$$

The reference channel r, which may also be used as a reply channel, is intended as a fresh channel and is thus captured by the restriction operator as shown below.

Suppose that programs are imported from some trusted channel i together with an indication of the level of trust of the program corresponding to each of the four sandboxes. We could define an agent that imports programs and trust indications, and then execute them in the corresponding environment, as follows:

$$\begin{aligned}
\mathrm{Handler} \stackrel{\text{def}}{=} *i?(X,t).(\nu r)(\ &\text{if } t=1 \text{ then } (Xr)\lceil \mathrm{Sandbox1}\\
&\text{else if } t=2 \text{ then } (Xr)\lceil \mathrm{Sandbox2}\\
&\text{else if } t=3 \text{ then } (Xr)\lceil \mathrm{Sandbox3}\\
&\text{else if } t=4 \text{ then } Xr)
\end{aligned}$$

The whole system may thus be defined as

$$\mathrm{System} \stackrel{\text{def}}{=} (\mathrm{Handler} \mid \mathrm{CPU} \mid \mathrm{Scr} \mid \ldots)\lceil\{i^-\}\ .$$

Pointers to the Web server sending the program may be included as a third argument to i, and dynamically put into the filtering set, thus allowing the imported program to communicate with its original site, a common feature in such systems. The handler becomes now

$$\begin{aligned}
\mathrm{Handler} \stackrel{\text{def}}{=} *i?(X,t,p).(\nu r)(\ &\text{if } t=1 \text{ then } (Xr)\lceil \mathrm{Sandbox1}\\
&\text{else if } t=2 \text{ then } (Xr)\lceil \mathrm{Sandbox2}\\
&\text{else if } t=3 \text{ then } (Xr)\lceil \mathrm{Sandbox3}\\
&\text{else if } t=4 \text{ then } Xr)
\end{aligned}$$

Each sandbox should include now the name p, which becomes bound to the argument p in $i?(X,t,p)$, e.g.

$$\mathrm{Sandbox1} ::= \{\mathrm{cpu}^+, \mathrm{scr}^+, \mathrm{kb}^+, \mathrm{ms}^+, \mathrm{mem}^+, \mathrm{r}^-, \mathrm{p}\}$$

3 The $\lambda\pi_v$-Calculus

The $\lambda\pi_v$-calculus [16] is a typed call-by-value λ-calculus augmented with the π-calculus primitives, in which arbitrary abstractions, including process abstractions, may be the object of output events. In the sequel we will follow the notation for process and types given in that work.

(Type) $\alpha, \beta, \gamma \ldots$

Term :	$\rho ::= \pi \mid \sigma_H$		
Base :	$\sigma_B ::= \mathtt{unit} \mid \mathtt{nat} \mid \ldots$	input only :	$\mathrm{S}^{\mathtt{I}} \stackrel{\text{def}}{=} \langle \mathrm{S}, \bot \rangle$
Process :	$\pi ::= [\Delta] \mid \mathtt{proc}$	output only :	$\mathrm{S}^{\mathtt{O}} \stackrel{\text{def}}{=} \langle \top, \mathrm{S} \rangle$
HO Value :	$\sigma_H ::= \sigma_B \mid \sigma_H \to \rho \mid (x : \sigma) \to \rho$	input/output :	$\mathrm{S}^{\mathtt{IO}} \stackrel{\text{def}}{=} \langle \mathrm{S}, \mathrm{S} \rangle$
Channel :	$\sigma ::= \langle \mathrm{S}_{\mathtt{I}}, \mathrm{S}_{\mathtt{O}} \rangle$		
Value :	$\tau ::= \sigma_H \mid \sigma$	thunk type :	$\langle \Delta \rangle \stackrel{\text{def}}{=} \mathtt{unit} \to [\Delta]$
Sort :	$S ::= (\tau_1, \ldots, \tau_n) \mid \top \mid \bot$		

(Abbreviation)

(Environment)

Channel : $\Delta ::= \emptyset \mid \Delta, u : \sigma$

General : $\Gamma ::= \emptyset \mid \Gamma, x : \tau \mid \Gamma, a : \sigma$

Fig. 1. Types for the $\lambda\pi_v$-Calculus

The types used in $\lambda\pi_v$ are shown in Fig. 1. A set of base types is assumed, as well as an infinite set of channel names and an infinite set of variables. Channel types take the form $\langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle$, where $S_{\mathtt{I}}$ denotes the input sort and $S_{\mathtt{O}}$ the output sort. Two constants $\bot$ and $\top$ denote the least resp. the greatest sort type. Values are of three kinds, base types, channel types, and higher order types. The latter can be either the functional type constructor $\sigma_H \to \rho$ or $(x : \sigma) \to \rho$, where ρ is either a HO-value type or a process type. The type $(x : \sigma) \to \rho$, where σ is a channel type, is a *dependent* functional type, and the type ρ may contain occurrences of the variable x.

Processes may have either type `proc`, the most general process type with no information about the interface, or $[\Delta]$, a finite mapping from channel names and variables to channel types indicating that the process P uses at most the resources in the domain of Δ. Note that this is not true for derivatives of P, i.e. residues or continuations of P after internal or external communication events. By way of illustration, consider a process P which can input, along channel u, a value of type $(\mathtt{int})^{\mathtt{O}}$, the type of a channel with the capability to export values of type `int`:

$$P = u?(x : (\mathtt{int})^{\mathtt{O}})x!\langle 1 \rangle \mathbf{0}$$

This process has type $[u : ((\mathtt{int})^{\mathtt{O}})^{\mathtt{I}}]$, but after importing any channel a of type $(\mathtt{int})^{\mathtt{O}}$ along u, P becomes $a!\langle 1 \rangle \mathbf{0}$, a process that must contain the channel a in the domain of its interface.

(Common)

$$\text{(VAL)}\ \frac{\Gamma, u:\tau, \Gamma' \vdash \texttt{Env}}{\Gamma, u:\tau, \Gamma' \vdash u:\tau} \qquad \text{(CON)}\ \frac{\Gamma \vdash \texttt{Env}}{\Gamma \vdash 1:\texttt{nat}}\ \text{etc.}$$

$$(\text{SUB}_H)\ \frac{\Gamma \vdash P:\rho \quad \Gamma \vdash \rho \leq \rho'}{\Gamma \vdash P:\rho'} \qquad (\text{SUB}_N)\ \frac{\Gamma \vdash u:\sigma \quad \Gamma \vdash \sigma \leq \sigma'}{\Gamma \vdash u:\sigma'}$$

(Function)

$$(\text{ABS}_H)\ \frac{\Gamma, X:\sigma_H \vdash P:\rho}{\Gamma \vdash \lambda(X:\sigma_H)P:\sigma_H \to \rho} \qquad (\text{APP}_H)\ \frac{\Gamma \vdash P:\sigma_H \to \rho \quad \Gamma \vdash Q:\sigma_H}{\Gamma \vdash PQ:\rho}$$

$$(\text{ABS}_N)\ \frac{\Gamma, x:\sigma \vdash P:\rho}{\Gamma \vdash \lambda(x:\sigma)P:(x:\sigma) \to \rho} \qquad (\text{APP}_N)\ \frac{\Gamma \vdash P:(x:\sigma) \to \rho \quad \Gamma \vdash u:\sigma}{\Gamma \vdash Pu:\rho\{u/x\}}$$

(Process)

$$\begin{array}{cccc} \text{(NIL)} & \text{(PAR)} & \text{(REP)} & \text{(RES)} \\ \dfrac{\Gamma \vdash \texttt{Env}}{\Gamma \vdash \mathbf{0}:[\,]} & \dfrac{\Gamma \vdash P_1:\pi\ \Gamma \vdash P_2:\pi}{\Gamma \vdash P_1 \mid P_2:\pi} & \dfrac{\Gamma \vdash P:\pi}{\Gamma \vdash *P:\pi} & \dfrac{\Gamma, a:\sigma \vdash P:\pi}{\Gamma \vdash (\nu a:\sigma)P:\pi/a} \end{array}$$

$$\begin{array}{cc} \text{(OUT)} & \text{(IN)} \\ \dfrac{\begin{array}{l}\pi \vdash_\Gamma u:(\tau_1,\ldots,\tau_n)^{\texttt{O}} \\ \pi \vdash_\Gamma V_i:\tau_i \quad \Gamma \vdash P:\pi\end{array}}{\Gamma \vdash u!\langle V_1,\ldots,V_n\rangle P:\pi} & \dfrac{\begin{array}{l}\pi \vdash_\Gamma u:(\tau_1,\ldots,\tau_n)^{\texttt{I}} \\ \Gamma, x_1:\tau_1,\ldots,x_n:\tau_n \vdash P:\pi, x_1:\tau_1,\ldots,x_n:\tau_n\end{array}}{\Gamma \vdash u?(x_1:\tau_1,\ldots,x_n:\tau_n)P:\pi} \end{array}$$

In (Res), π/a means erasing all occurrences of $a:\sigma$ from π.
In (Out,In), $\pi \vdash_\Gamma V:\tau$ means $\Gamma \vdash [V:\tau] \leq \pi$ if τ is a channel type; else $\Gamma \vdash V:\tau$.
In (In), $\pi, x:\sigma$ is defined as: $\pi, x:\sigma \stackrel{\text{def}}{=} \pi \sqcup [x:\sigma]$, and $\pi, x:\sigma_H \stackrel{\text{def}}{=} \pi$.

Fig. 2. Typing system for the $\lambda\pi_{\mathrm{v}}$-calculus

The inference rules for the typing system are given in Fig. 2. The **(Function)** rules are inherited from typing systems for the λ-calculus, whereas the **(Process)** rules are based on IO-typing systems [5, 11, 15] ($\sqcup$ is a partial least upper bound operator which intuitively calculates the union of the accessibility of two capabilities). See [16] for the details.

4 The $\boldsymbol{\lambda\pi_{\mathrm{F_v}}}$-Calculus

The $\lambda\pi_{\mathrm{F_v}}$-calculus is basically the $\lambda\pi_{\mathrm{v}}$-calculus extended with the filter operator and without type annotations. The syntax for terms of the $\lambda\pi_{\mathrm{F_v}}$-calculus is shown in Fig. 3. Values consist of identifiers and abstractions as in the λ-calculus. Terms include values, application, literals, and π-calculus terms. The definitions of free names and free variables for types and terms are standard.

The reduction semantics (Fig. 4) is based on a structural equivalence relation, denoted $\equiv$, and a reduction relation $P \longrightarrow Q$. The definition of transition must

(Term)

$P, Q, \ldots ::=$	V	value
	$\mid \mathbf{0}$	nil
	$\mid P \mid P$	parallel
	$\mid u!\langle V_1, \ldots, V_n \rangle P$	output
	$\mid u?(x_1, \ldots, x_n) P$	input
	$\mid *P$	replicator
	$\mid (\nu a) P$	restriction
	$\mid PP$	application
	$\mid P \lceil L$	filter

(Identifier)

$u, v, w, \ldots ::=$	l	literal
	$\mid x, y, z, \ldots$	variable
	$\mid a, b, c, \ldots$	channel

(Value)

$V, W, \ldots ::=$	$u, v, w, \ldots$	identifier
	$\mid \lambda(x) P$	abstraction

(Literal)

$l, l', \ldots ::=$	$()$	unit
	$\mid 1, 2, 3, \ldots$	number

(Filter)

$L ::=$ a finite set of polarised channels

Fig. 3. Syntax for the $\lambda\pi\mathrm{F_v}$-calculus

allow actions across the filter operator, and thus labelled transition rules must be introduced. It can be shown that for agents in $\lambda\pi\mathrm{F_v}$ without occurrences of the filter operator, the reduction rules in [16], with type annotations erased, yield the same reduction relation $\longrightarrow$ for agents in $\lambda\pi\mathrm{F_v}$ as the reduction relation obtained by the reduction rules of Fig. 4. If we write $\longrightarrow_{\lambda\pi_\mathrm{v}}$ for the reduction relation obtained from the former, and $\longrightarrow_{\lambda\pi\mathrm{F_v}}$ for the reduction relation obtained from the latter, this result can be stated as follows:

Lemma 1. *For any agents P and P' in $\lambda\pi F_v$ with no occurrences of the filter operator, it holds that $P \longrightarrow_{\lambda\pi_v} P'$ iff $P \longrightarrow_{\lambda\pi F_v} P'$.*

Proof. The *only if* part of the assertion follows easily from the fact that all rules in the $\lambda\pi_\mathrm{v}$ calculus are also rules in the $\lambda\pi\mathrm{F_v}$-calculus. To prove the other direction it is enough to show that (i) whenever $P \stackrel{a?(\widetilde{V})}{\longrightarrow} P'$ then $P \equiv a?(\widetilde{x})P_1 \mid P_2$ and $P' \equiv P_1\{\widetilde{V}/\widetilde{x}\} \mid P_2$ for some agents P_1 and P_2; (ii) whenever $P \stackrel{a!\langle\widetilde{V}\rangle}{\longrightarrow} P'$ then $P \equiv a!\langle\widetilde{V}\rangle P_1 \mid P_2$ and $P' \equiv P_1 \mid P_2$ for some agents P_1 and P_2. The result follows then by application of the structural equivalence rules in $\lambda\pi_\mathrm{v}$, which are identical to the corresponding rules in $\lambda\pi\mathrm{F_v}$ except for the cases involving the filter operator. The proof if by induction on the length of derivation of $P \stackrel{a?(\widetilde{V})}{\longrightarrow} P'$ resp. $P \stackrel{a!\langle\widetilde{V}\rangle}{\longrightarrow} P'$. If the last rule applied is (input) or (output), the result follows immediately. If the last rule is (lpar), the result follows easily from the structural equivalence rules involving composition. If the last rule is (str), the result follows since in this case $P \equiv P'$ and by the induction hypothesis the claim is valid for P'. $\square$

5 Encoding $\lambda\pi_v$ into $\lambda\pi_{F_v}$

We interpret type annotations in higher order input variables as specifications of resource access restrictions. These restrictions can be enforced dynamically by the filter operator. By way of illustration, consider the agent

$$c?(X : \langle\Delta_{ab}\rangle)\, X() \ .$$

This denotes a process willing to import along the channel c any agent of type $\langle\Delta_{ab}\rangle$ where

$$\Delta_{ab} = \{a : (\mathtt{int})^{\mathtt{I}}, b : (\mathtt{int})^{\mathtt{O}}\}$$

i.e. an interface consisting of the channels a and b. Values of type integer can be imported along a, and exported along b. This may be tentatively encoded in $\lambda\pi_{F_v}$ as

$$c?(X)(X())\lceil\{a^-, b^+\} \ .$$

(Reduction)

$$\text{(input)} \quad a?(x_1, \ldots, x_n)P \stackrel{a?(V_1,\ldots,V_n)}{\longrightarrow} P\{V_1/x_1, \ldots, V_n/x_n\}$$

$$\text{(output)} \quad a!\langle V_1, \ldots, V_n\rangle P \stackrel{a!\langle V_1,\ldots,V_n\rangle}{\longrightarrow} P$$

$$\text{(filter)} \ \frac{P \stackrel{\alpha}{\longrightarrow} P'}{P\lceil L \stackrel{\alpha}{\longrightarrow} P'\lceil L} \quad \begin{array}{l}\alpha \text{ void or}\\ \alpha = b!\langle\widetilde{V}\rangle \text{ and } b^+ \in L \text{ or}\\ \alpha = b?(\widetilde{X}) \text{ and } b^- \in L\end{array}$$

$$\text{(lpar)} \ \frac{P \stackrel{\alpha}{\longrightarrow} P'}{P \mid Q \stackrel{\alpha}{\longrightarrow} P' \mid Q} \qquad \text{(com)} \ \frac{P \stackrel{a?(V_1,\ldots,V_n)}{\longrightarrow} P' \quad Q \stackrel{a!\langle V_1,\ldots,V_n\rangle}{\longrightarrow} Q'}{P \mid Q \longrightarrow P' \mid Q'}$$

$$(\beta) \quad (\lambda(x : \tau)P)V \longrightarrow P\{V/x\} \qquad (\text{app}_r) \ \frac{Q \longrightarrow Q'}{PQ \longrightarrow PQ'} \qquad (\text{app}_l) \ \frac{P \longrightarrow P'}{PV \longrightarrow P'V}$$

$$\text{(res)} \ \frac{P \longrightarrow P'}{(\nu a)P \longrightarrow (\nu a)P'} \qquad \text{(str)} \ \frac{P \equiv P' \stackrel{\alpha}{\longrightarrow} Q' \equiv Q}{P \stackrel{\alpha}{\longrightarrow} Q}$$

(Structural Equivalence)

- $P \equiv Q$ if $P \equiv_\alpha Q$.
- $P \mid Q \equiv Q \mid P$, $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$, $P \mid \mathbf{0} \equiv P$, $*P \equiv P \mid *P$
- $(\nu a)\mathbf{0} \equiv \mathbf{0}$, $(\nu a)P \mid Q \equiv (\nu a)(P \mid Q)$ if $a \notin \mathtt{fn}(Q)$
  $(\nu a)(\nu b)P \equiv (\nu b)(\nu a)Q$
- $((\nu a)P)\lceil L \equiv (\nu a)(P\lceil L)$ if $a^+ \notin L$ and $a^- \notin L$, $\quad \mathbf{0}\lceil L \equiv \mathbf{0}$

Note that α may be void in any of the reduction rules.

Fig. 4. Reduction for the $\lambda\pi_{F_v}$-calculus

Pure higher-order passing This technique cannot though be generalised to first-order value passing, cf. the process

$$x?(Y : \langle \Delta_c \rangle)\ Y() \ ,$$

where

$$\Delta_c = \{c : (\mathtt{int}^{\mathtt{O}})^{\mathtt{IO}}\} \ .$$

This agent cannot be adequately encoded by

$$x?(Y).(Y())\lceil\{c^-\}$$

since this would mean that a process instantiating Y, e.g.

$$c?(z : (\mathtt{int}^{\mathtt{O}}))z!\langle 1 \rangle \ ,$$

can only communicate along c, which is not what is expected here. The interface of this process consists rather of input actions along the channel c and output actions along any channel of type $(\mathtt{int})^{\mathtt{O}}$. Given the environment Γ this interface may be extracted. An interpretation of the resource access policy of a process of this kind is that there are no resources of type $(\mathtt{int})^{\mathtt{I}}$ which need protection, and thus any channel of this type may be allowed to perform output actions without damage. As a result this process can be encoded by

$$x?(Y).(Y())\lceil B \ ,$$

where

$$B = \{c^-\} \cup \{a^+ \mid \Gamma \vdash a : (\mathtt{int})^{\mathtt{O}}\} \ .$$

We may thus safely execute the incoming process thunk instantiating the variable Y in a context which filters inputs along c and outputs along any channel $a \in \mathtt{dom}(\Gamma)$ such that

$$\Gamma \vdash a : (\mathtt{int})^{\mathtt{O}} \ .$$

This is the basic idea for the encoding.

Dependent types Higher order values other than process thunks can be also imported. For instance, the agent

$$c?(X : \tau_x)Xa$$

can be given the type

$$[c : (\tau_x)^{\mathtt{I}},\ a : (\mathtt{int})^{\mathtt{I}}]$$

where

$$\Delta_{xy} = \{x : (\mathtt{int})^{\mathtt{I}}, y : (\mathtt{int})^{\mathtt{O}}\} \quad \text{and} \quad \tau_x = (x : (\mathtt{int})^{\mathtt{I}}) \to [\Delta_{xb}]$$

We have now a channel c that admits incoming agents of the dependent type τ_x. The incoming process must, on being applied to any channel a, yield a process of type Δ_{ab}. This agent can be encoded as

$$c?(X)(Xa)\lceil\{a^-, b^+\} \ .$$

Consider now the process

$$c?(X : \tau_x, y : (\mathtt{int})^{\mathtt{I}})Xy \ .$$

In this case the channel instantiating x in Δ_{xb} is sent along c together with an agent of type τ_x. We encode this process as:

$$c?(X, y)(Xy)\lceil\{y^-, b^+\} \ .$$

Wrapping higher-order variables The scheme discussed above is still not sufficiently general. Consider the process

$$c?(X : \tau_x, Z : (\tau_x \to \rho))ZX \ .$$

In this case, X is given as argument to an unknown agent Z. In order to encode the agent X in a way that preserves the desired behaviour in all contexts, we have to "wrap up" X as follows

$$c?(X, Z)Z(\lambda(y)(Xy)\lceil\{y^-, b^+\})$$

By wrapping X according to its type, the interface of Xy becomes of the desired form. The variable Z must be given the same treatment. This technique may be generalised to any number and type of arguments. For instance, consider a process of the form

$$c?(X : \tau_{xZy}, Y : (\tau_{xZy} \to \rho))YX$$

where

$$\tau_{xZy} \stackrel{\text{def}}{=} (x : (\mathtt{int})^{\mathtt{I}}) \to \sigma_H \to (y : (\mathtt{int})^{\mathtt{O}}) \to [\Delta]$$

for suitable types σ_H, ρ, and $[\Delta]$. As a first approximation this process can be encoded by

$$c?(X, Y)Y(\lambda(x)\lambda(Z)\lambda(y)(XxZy)\lceil F)$$

where $F = \mathcal{I}_\Gamma([\Delta])$. Basically, $\mathcal{I}_\Gamma([\Delta])$ denotes the polarised channels in the closure of Δ, i.e. the extension of Δ with all channels in Γ that may be imported by the process instantiating X or any of its derivatives. However, the variable Y should also be wrapped according to its type σ_H. The formalisation of these ideas are given in the next section.

5.1 The Translation

The translation from $\lambda\pi_{\mathrm{v}}$ to $\lambda\pi\mathrm{F}_{\mathrm{v}}$, $\mathcal{T}_\Gamma$, is given in Fig. 5. We assume that bound names have been conveniently renamed in order to avoid name clashes. The translation extends to abstractions. Basically $\mathcal{T}_\Gamma$ is a homomorphism over all process operators and abstractions. The interesting point is the translation of an agent variable x, which depends on its type τ, and is defined as $\mathcal{F}_\Gamma(x,\tau)$. The role of the latter is roughly to wrap up the higher order variable x according to its type τ, and at the same time to force x to execute within the scope of a filter operator embodying the restrictions on x's behaviour expressed by the type τ.

The predicate $\mathcal{C}_\Gamma([\Delta])$, defined in Fig. 6, denotes the closure of the process type $[\Delta]$. It can be shown that $\mathcal{C}_\Gamma(\pi)$ is the minimal closure of π, i.e. if $u^+ \in \mathtt{dom}(\mathcal{C}_\Gamma(\pi))$ (resp. $u^- \in \mathtt{dom}(\mathcal{C}_\Gamma(\pi))$) then there is a process P such that $\Gamma \vdash P : \pi$ and for some derivative P' of P it holds that P' commits to $u!$ (resp. $u?$). The notion of derivative is formalised in Section 5.2 (cf. [7] Section 2.6). The predicate $\mathcal{I}_\Gamma([\Delta])$ denotes the set of channels occurring in the domain of $\mathcal{C}_\Gamma([\Delta])$ together with their corresponding polarities. Thus, if

$$\Gamma \vdash [u : (S)^{\mathtt{I}}] \leq \mathcal{C}_\Gamma([\Delta])$$

then $u^- \in \mathcal{I}_\Gamma([\Delta])$. Analogously, if

$$\Gamma \vdash [u : (S)^{\mathtt{O}}] \leq \mathcal{C}_\Gamma([\Delta])$$

then $u^+ \in \mathcal{I}_\Gamma([\Delta])$.

We can establish the following important results:

Lemma 2. *The operator $\mathcal{C}_\Gamma$ is monotonic increasing and a closure operator on well-formed types in Γ, i.e.*

- $\Gamma \vdash \pi \leq \mathcal{C}_\Gamma(\pi)$
- $\Gamma \vdash \pi \leq \pi'$ *implies* $\Gamma \vdash \mathcal{C}_\Gamma(\pi) \leq \mathcal{C}_\Gamma(\pi')$
- $\Gamma \vdash \mathcal{C}_\Gamma(\pi) = \mathcal{C}_\Gamma(\mathcal{C}_\Gamma(\pi))$

Before proving this lemma, we need an auxiliary result:

Lemma 2A (i) $\mathcal{C}_\Gamma([\Delta,\Delta']) = \mathcal{C}_\Gamma([\Delta]) \sqcup \mathcal{C}_\Gamma([\Delta'])$ for any channel environment $[\Delta,\Delta']$; (ii) for any $u \in \mathtt{dom}(\Gamma)$ and any σ, σ' such that $\Gamma \vdash \Gamma(u) \leq \sigma$ and $\Gamma \vdash \Gamma(u) \leq \sigma'$, it holds that

$$\mathcal{C}_\Gamma([u:\sigma] \sqcup [u:\sigma']) = \mathcal{C}_\Gamma([u:\sigma]) \sqcup \mathcal{C}_\Gamma([u:\sigma'])$$

Proof of Lemma 2A. Claim (i) follows easily from the definition of $\mathcal{C}_\Gamma([\Delta])$. We prove (ii) by induction on the formation of $\Gamma(u)$.

Assume $\sigma = \langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle$, $\sigma' = \langle S'_{\mathtt{I}}, S'_{\mathtt{O}}\rangle$, and $\Gamma(u) = \sigma'' = \langle S''_{\mathtt{I}}, S''_{\mathtt{O}}\rangle$. Hence $\Gamma \vdash S''_{\mathtt{I}} \leq S_{\mathtt{I}}$ and $\Gamma \vdash S''_{\mathtt{I}} \leq S'_{\mathtt{I}}$. Since $[u:\sigma] \sqcup [u:\sigma'] = [u:\sigma \sqcap \sigma']$ and $\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle \sqcap \langle S'_{\mathtt{I}}, S'_{\mathtt{O}}\rangle = \langle S_{\mathtt{I}} \sqcap S'_{\mathtt{I}}, S_{\mathtt{O}} \sqcup S'_{\mathtt{O}}\rangle$, we have to show that

$$\mathcal{C}_\Gamma([u : \langle S_{\mathtt{I}} \sqcap S'_{\mathtt{I}}, S_{\mathtt{O}} \sqcup S'_{\mathtt{O}}\rangle]) = \mathcal{C}_\Gamma([u : \langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle]) \sqcup \mathcal{C}_\Gamma([u : \langle S'_{\mathtt{I}}, S'_{\mathtt{O}}\rangle])$$

Since the left term above reduces to $\mathcal{D}_\Gamma(S_{\mathtt{I}} \sqcap S'_{\mathtt{I}}, S''_{\mathtt{I}}) \sqcup [u : \langle S_{\mathtt{I}} \sqcap S'_{\mathtt{I}}, S_{\mathtt{O}} \sqcup S'_{\mathtt{O}}\rangle]$, and the right term to $\mathcal{D}_\Gamma(S_{\mathtt{I}}, S''_{\mathtt{I}}) \sqcup \mathcal{D}_\Gamma(S'_{\mathtt{I}}, S''_{\mathtt{I}}) \sqcup [u : \langle S_{\mathtt{I}} \sqcap S'_{\mathtt{I}}, S_{\mathtt{O}} \sqcup S'_{\mathtt{O}}\rangle]$, what we have to show reduces to

$$\mathcal{D}_\Gamma(S_{\mathtt{I}} \sqcap S'_{\mathtt{I}}, S''_{\mathtt{I}}) = \mathcal{D}_\Gamma(S_{\mathtt{I}}, S''_{\mathtt{I}}) \sqcup \mathcal{D}_\Gamma(S'_{\mathtt{I}}, S''_{\mathtt{I}})$$

Assume $S_{\mathtt{I}} = \top$. Hence $S_{\mathtt{I}} \sqcap S'_{\mathtt{I}} = S'_{\mathtt{I}}$. Thus $\mathcal{D}_\Gamma(S_{\mathtt{I}}, S''_{\mathtt{I}}) \sqcup \mathcal{D}_\Gamma(S'_{\mathtt{I}}, S''_{\mathtt{I}}) = [] \sqcup \mathcal{D}_\Gamma(S'_{\mathtt{I}}, S''_{\mathtt{I}}) = \mathcal{D}_\Gamma(S'_{\mathtt{I}}, S''_{\mathtt{I}}) = \mathcal{D}_\Gamma(S_{\mathtt{I}} \sqcap S'_{\mathtt{I}}, S''_{\mathtt{I}})$ as desired.

If $S'_{\mathtt{I}} = \top$, the argument is the same but with the roles of the sort types $S_{\mathtt{I}}$ and $S'_{\mathtt{I}}$ inverted.

Assume thus that $S_{\mathtt{I}} = (\tau_1, \ldots, \tau_n)$ and $S'_{\mathtt{I}} = (\tau'_1, \ldots, \tau'_n)$. Since $\Gamma \vdash S''_{\mathtt{I}} \leq S_{\mathtt{I}}$, $S''_{\mathtt{I}} \neq \top$. Assume thus that $S''_{\mathtt{I}} = (\tau''_1, \ldots, \tau''_n)$, where necessarily $\Gamma \vdash \tau''_i \leq \tau_i$ and $\Gamma \vdash \tau''_i \leq \tau'_i$ for every i. By the definition of $\mathcal{D}_\Gamma$ it suffices to show that for every i it holds that

$$\mathcal{D}_\Gamma(\tau_i \sqcap \tau'_i, \tau''_i) = \mathcal{D}_\Gamma(\tau_i, \tau''_i) \sqcup \mathcal{D}_\Gamma(\tau'_i, \tau''_i)$$

$$
\begin{aligned}
\mathcal{T}_\Gamma(P) &= \mathcal{T}^{\emptyset}_\Gamma(P) \\
\mathcal{T}^S_\Gamma(P \mid P) &= \mathcal{T}^S_\Gamma(P) \mid \mathcal{T}^S_\Gamma(P) \\
\mathcal{T}^S_\Gamma(u!\langle V_1, \ldots, V_n\rangle.P) &= u!\langle \mathcal{T}^{\emptyset}_\Gamma(V_1), \ldots, \mathcal{T}^{\emptyset}_\Gamma(V_n)\rangle.\mathcal{T}^S_\Gamma(P) \\
\mathcal{T}^S_\Gamma(u?(x_1 : \tau_1, \ldots, x_n : \tau_n)P) &= u?(x_1, \ldots, x_n)\mathcal{T}^{S'}_{\Gamma'}(P) \\
& \quad S' = S \cup \{x_i : \tau_i \text{ s.t. } \tau_i \text{ is a HO Value},\ i = 1, \ldots, n\} \\
& \quad \Gamma' = \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \\
\mathcal{T}^S_\Gamma(*P) &= *\mathcal{T}^S_\Gamma(P) \\
\mathcal{T}^S_\Gamma((\nu a : \sigma)P) &= (\nu a)\mathcal{T}^S_{\Gamma'}(P),\ \Gamma' = \Gamma, a : \sigma \\
\mathcal{T}^S_\Gamma(PP) &= \mathcal{T}^S_\Gamma(P)\mathcal{T}^S_\Gamma(P) \\
\mathcal{T}^S_\Gamma(u) &= \text{if } u \in \mathtt{dom}(S) \text{ then } \mathcal{F}_\Gamma(u, S(u)) \text{ else } u \\
\mathcal{T}^S_\Gamma(\lambda(x : \tau)P) &= \lambda(x)\mathcal{T}^S_{\Gamma'}(P),\ \Gamma' = \Gamma, x : \tau
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}_\Gamma(P, \sigma_B) &= P \\
\mathcal{F}_\Gamma(P, \mathtt{proc}) &= P \\
\mathcal{F}_\Gamma(P, [\Delta]) &= P\lceil \mathcal{I}_\Gamma([\Delta]) \\
\mathcal{F}_\Gamma(P, (y : \sigma) \to \rho) &= \lambda(y)(\mathcal{F}_{\Gamma'}(Py, \rho)),\ \Gamma' = \Gamma, y : \sigma \\
\mathcal{F}_\Gamma(P, \sigma_H \to \rho) &= \lambda(y)\mathcal{F}_\Gamma(Py, \rho))
\end{aligned}
$$

Fig. 5. Translation

$$
\begin{aligned}
\mathcal{C}_\Gamma([]) &= [] \\
\mathcal{C}_\Gamma([\Delta, u : \sigma]) &= \mathcal{C}_\Gamma([u : \sigma]) \sqcup \mathcal{C}_\Gamma([\Delta]) \\
\mathcal{C}_\Gamma([u : \langle S_\mathtt{I}, S_\mathtt{O} \rangle]) &= [u : \langle S_\mathtt{I}, S_\mathtt{O} \rangle] \sqcup \mathcal{D}_\Gamma(S_\mathtt{I}, S'_\mathtt{I}) \text{ (where } \Gamma(u) = \langle S'_\mathtt{I}, S'_\mathtt{O} \rangle) \\[1ex]
\mathcal{D}_\Gamma(\top, S) &= [] \\
\mathcal{D}_\Gamma((\tau_1, \ldots, \tau_n), (\tau'_1, \ldots, \tau'_n)) &= \mathcal{D}_\Gamma(\tau_1, \tau'_1) \sqcup \ldots \sqcup \mathcal{D}_\Gamma(\tau_n, \tau'_n) \\
\mathcal{D}_\Gamma(\sigma_H, \sigma'_H) &= [] \\
\mathcal{D}_\Gamma(\sigma, \sigma') &= \mathcal{C}(\Gamma, \sigma, \sigma') \\[1ex]
\mathcal{C}(\emptyset, \sigma, \sigma') &= [] \\
\mathcal{C}((\Gamma, x : \sigma_H), \sigma, \sigma') &= \mathcal{C}(\Gamma, \sigma, \sigma') \\
\mathcal{C}((\Gamma, u : \sigma''), \sigma, \sigma') &= \textbf{ if } \Gamma \vdash \sigma'' \leq \sigma' \textbf{ then } \mathcal{C}(\Gamma, \sigma, \sigma') \sqcup \mathcal{C}_\Gamma([u : \sigma]) \textbf{ else } \mathcal{C}(\Gamma, \sigma, \sigma') \\
\mathcal{I}_\Gamma([\Delta]) &= \mathrm{interface}(\mathcal{C}_\Gamma([\Delta])) \\[1ex]
\mathrm{interface}([]) &= \emptyset \\
\mathrm{interface}([\Delta, a : \langle S_\mathtt{I}, S_\mathtt{O} \rangle]) &= \mathrm{interface}([\Delta]) \cup \mathcal{P}_-(a, S_\mathtt{I}) \cup \mathcal{P}_+(a, S_\mathtt{O}) \\[1ex]
\mathcal{P}_-(u, S) &= \textbf{ if } S \neq \top \textbf{ then } \{u^-\} \textbf{ else } \emptyset \\
\mathcal{P}_+(u, S) &= \textbf{ if } S \neq \bot \textbf{ then } \{u^+\} \textbf{ else } \emptyset
\end{aligned}
$$

Fig. 6. Closure and Interface

If $\tau_i = \sigma_H$ then both sides reduce to $[]$. Assume thus that $\tau_i = \sigma_i$ for some channel type σ_i. Then there exits channel types σ'_i and σ''_i such that $\tau'_i = \sigma'_i$ and $\tau''_i = \sigma''_i$. But then $\mathcal{D}_\Gamma(\tau_i, \tau''_i) = \mathcal{D}_\Gamma(\sigma_i, \sigma''_i) = \mathcal{C}(\Gamma, \sigma_i, \sigma''_i)$. Likewise, $\mathcal{D}_\Gamma(\tau'_i, \tau''_i) = \mathcal{D}_\Gamma(\sigma'_i, \sigma''_i) = \mathcal{C}(\Gamma, \sigma'_i, \sigma''_i)$ and $\mathcal{D}_\Gamma(\tau_i \sqcap \tau'_i, \tau''_i) = \mathcal{D}_\Gamma(\sigma_i \sqcap \sigma'_i, \sigma''_i) = \mathcal{C}(\Gamma, \sigma_i \sqcap \sigma'_i, \sigma''_i)$. Thus, what we have to show reduces to

$$\mathcal{C}(\Gamma, \sigma_i \sqcap \sigma'_i, \sigma''_i) = \mathcal{C}(\Gamma, \sigma_i, \sigma''_i) \sqcup \mathcal{C}(\Gamma, \sigma'_i, \sigma''_i)$$

By the definition of $\mathcal{C}$ it suffices to show that for every $x : \sigma \in \Gamma$ such that $\Gamma \vdash \sigma \leq \sigma''_i$ we have $\mathcal{C}_\Gamma([x : \sigma_i \sqcap \sigma'_i]) = \mathcal{C}_\Gamma([x : \sigma_i]) \sqcup \mathcal{C}_\Gamma([x : \sigma'_i])$. Since $[x : \sigma_i \sqcap \sigma'_i] = [x : \sigma_i] \sqcup [x : \sigma'_i]$ this reduces to showing that

$$\mathcal{C}_\Gamma([x : \sigma_i] \sqcup [x : \sigma'_i]) = \mathcal{C}_\Gamma([x : \sigma_i]) \sqcup \mathcal{C}_\Gamma([x : \sigma'_i])$$

But this follows from the induction hypothesis, and the proof is complete. □

We turn now to the proof of Lemma 2:

Proof of Lemma 2. By Lemma 2A it is enough to show that the claims above are true for singleton channel environments $[u : \sigma]$.
Claim (i) follows easily from the fact that by definition $\mathcal{C}_\Gamma([u : \langle S_\mathtt{I}, S_\mathtt{O} \rangle]) = [u : \langle S_\mathtt{I}, S_\mathtt{O} \rangle] \sqcup \mathcal{D}_\Gamma(S_\mathtt{I}, S'_\mathtt{I})$, where we assume $u : \langle S'_\mathtt{I}, S'_\mathtt{O} \rangle \in \Gamma$, and obviously $\Gamma \vdash [u : \langle S_\mathtt{I}, S_\mathtt{O} \rangle] \leq [u : \langle S_\mathtt{I}, S_\mathtt{O} \rangle] \sqcup \mathcal{D}_\Gamma(S_\mathtt{I}, S'_\mathtt{I})$.

We show now (ii), monotonicity, i.e. $\Gamma \vdash \mathcal{C}_\Gamma([u:\sigma]) \le \mathcal{C}_\Gamma([u:\sigma'])$ whenever $\Gamma \vdash \sigma' \le \sigma$. We show this by induction on the formation of σ'', where $\sigma'' = \Gamma(u)$ and hence necessarily $\Gamma \vdash \sigma'' \le \sigma' \le \sigma$. Assume $\sigma = \langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle$, $\sigma' = \langle S'_{\mathtt{I}}, S'_{\mathtt{O}}\rangle$ and $\sigma'' = \langle S''_{\mathtt{I}}, S''_{\mathtt{O}}\rangle$. We must have $\Gamma \vdash S''_{\mathtt{I}} \le S'_{\mathtt{I}} \le S_{\mathtt{I}}$. By the definition of $\mathcal{C}_\Gamma$, what we have to prove reduces to showing that $\Gamma \vdash \mathcal{D}_\Gamma(S_{\mathtt{I}}, S''_{\mathtt{I}}) \le \mathcal{D}_\Gamma(S'_{\mathtt{I}}, S''_{\mathtt{I}})$. If $S_{\mathtt{I}} = \top$ the result follows since in this case $\mathcal{D}_\Gamma(S_{\mathtt{I}}, S''_{\mathtt{I}}) = []$. Assume that $S_{\mathtt{I}} = (\tau_1, \ldots, \tau_n)$ and $S''_{\mathtt{I}} = (\tau''_1, \ldots, \tau''_n)$. In this case we must also have $S'_{\mathtt{I}} = (\tau'_1, \ldots, \tau'_n)$ for some τ'_i such that $\Gamma \vdash \tau''_i \le \tau'_i \le \tau_i$ for each i. The result will follow if we can show that for every i it holds that that $\Gamma \vdash \mathcal{D}_\Gamma(\tau_i, \tau''_i) \le \mathcal{D}_\Gamma(\tau'_i, \tau''_i)$. If $\tau_i = \sigma_H$ the result follows immediately. Assume thus that $\tau_i = \langle \widetilde{S}_{\mathtt{I}}, \widetilde{S}_{\mathtt{O}}\rangle$, $\tau'_i = \langle \widetilde{S}'_{\mathtt{I}}, \widetilde{S}'_{\mathtt{O}}\rangle$, and $\tau''_i = \langle \widetilde{S}''_{\mathtt{I}}, \widetilde{S}''_{\mathtt{O}}\rangle$. Then $\Gamma \vdash \widetilde{S}''_{\mathtt{I}} \le \widetilde{S}'_{\mathtt{I}} \le \widetilde{S}_{\mathtt{I}}$. By the definition of $\mathcal{D}_\Gamma$, it is enough to show that $\Gamma \vdash \mathcal{C}_\Gamma([x:\langle \widetilde{S}_{\mathtt{I}}, \widetilde{S}_{\mathtt{O}}\rangle]) \le \mathcal{C}_\Gamma([x:\langle \widetilde{S}'_{\mathtt{I}}, \widetilde{S}'_{\mathtt{O}}\rangle])$ for every $x:\tilde{\sigma} \in \Gamma$ such that $\Gamma \vdash \tilde{\sigma} \le \langle \widetilde{S}''_{\mathtt{I}}, \widetilde{S}''_{\mathtt{O}}\rangle$. But this follows from the induction hypothesis, and the proof of (ii) is thus complete.

We show now (iii). By (i) and (ii) it suffices to show that $\mathcal{C}_\Gamma(\mathcal{C}_\Gamma([u:\sigma])) \le \mathcal{C}_\Gamma([u:\sigma])$. Suppose that $[\Delta] = \mathcal{C}_\Gamma([u:\sigma])$. The result will follow if we can show that for every $x:\sigma' \in \Delta$ we have $\Gamma \vdash \mathcal{C}_\Gamma([x:\sigma']) \le \mathcal{C}_\Gamma([u:\sigma])$. This follows from the fact that the only way that x may have been included in the domain of $\mathcal{C}_\Gamma([\Delta])$ is by expressions of type $\mathcal{C}_\Gamma([x:\tilde{\sigma}])$ which may occur (see Figure 15) in either an equality of the form

$$\mathcal{C}_\Gamma([\Delta, x:\tilde{\sigma}]) = \mathcal{C}_\Gamma([\Delta]) \sqcup \mathcal{C}_\Gamma([x:\tilde{\sigma}])$$

or of the form

$$\mathcal{C}((\Gamma, x:\tilde{\sigma}'), \tilde{\sigma}, \tilde{\sigma}'') = \textbf{ if } \Gamma \vdash \tilde{\sigma}' \le \tilde{\sigma}'' \textbf{ then } \mathcal{C}(\Gamma, \tilde{\sigma}, \tilde{\sigma}'') \sqcup \mathcal{C}_\Gamma([x:\tilde{\sigma}]) \textbf{ else } \mathcal{C}(\Gamma, \tilde{\sigma}, \tilde{\sigma}'')$$

Now $x:\sigma'$ must be the result of the union of the set of all occurrences of expressions $[x:\sigma_1], \ldots, [x:\sigma_n]$ for some $n \ge 1$ in the definition of $\mathcal{C}_\Gamma([u:\sigma])$, i.e. $[x:\sigma'] = [x:\sigma_1] \sqcup \ldots \sqcup [x:\sigma_n] = [x:\sigma_1 \sqcap \ldots \sqcap \sigma_n]$, hence $\sigma' = \sigma_1 \sqcap \ldots \sqcap \sigma_n$. Since, as noted above, each occurrence of $[x:\sigma_i]$ must have been derived from an occurrence of the expression $\mathcal{C}_\Gamma([x:\sigma_i])$, this implies that also that $\Gamma \vdash \mathcal{C}_\Gamma([x:\sigma_1]) \sqcup \ldots \sqcup \mathcal{C}_\Gamma([x:\sigma_n]) \le \mathcal{C}_\Gamma([u:\sigma])$, and since by Lemma 2A $\mathcal{C}_\Gamma([x:\sigma_1]) \sqcup \ldots \sqcup \mathcal{C}_\Gamma([x:\sigma_n]) = \mathcal{C}_\Gamma([x:\sigma_1] \sqcup \ldots \sqcup [x:\sigma_n]) = \mathcal{C}_\Gamma([x:\sigma_1 \sqcap \ldots \sqcap \sigma_n]) = \mathcal{C}_\Gamma([x:\sigma'])$, we obtain $\Gamma \vdash \mathcal{C}_\Gamma([x:\sigma']) \le \mathcal{C}_\Gamma([u:\sigma])$ as desired and the proof is thus complete. $\square$

Lemma 3. *If $\Gamma(u) = \langle(\tau_1, \ldots, \tau_n), S_{\mathtt{O}}\rangle$, $\tau_k = \sigma$ for some $k \in \{1, \ldots, n\}$, and $\Gamma \vdash a:\sigma$, then $\Gamma \vdash [a:\sigma] \le \mathcal{C}_\Gamma([u:(\tau_1, \ldots, \tau_n)^{\mathtt{I}}])$.*

Proof. By the definition of $\mathcal{C}_\Gamma$, $\Gamma \vdash \mathcal{D}_\Gamma(\sigma, \sigma) \le \mathcal{C}_\Gamma([u:(\tau_1, \ldots, \tau_n)^{\mathtt{I}}])$. By the definition of $\mathcal{D}_\Gamma$ we get

$$\Gamma \vdash [a:\sigma] \le \mathcal{D}_\Gamma(\sigma, \sigma) \le \mathcal{C}_\Gamma([u:(\tau_1, \ldots, \tau_n)^{\mathtt{I}}]$$

as desired. $\square$

5.2 Adequacy of the Translation

We show that the behaviours of P and $\mathcal{T}_\Gamma(P)$ are indiscernible within contexts that are translations of well-typed terms in $\lambda\pi_v$. This follows basically from the fact that occurrences of the filter operator in $\mathcal{T}_\Gamma(P)$ are "transparent" in the sense that actions executed by $\mathcal{T}_\Gamma(P)$, or any of its derivatives, within contexts that are translations of contexts in $\lambda\pi_v$, are never blocked by these operators. We consider here only *value-passing* processes [2]. This means that if $\Gamma \vdash P : \pi$ and V is any value that may occur as the object of a communication event during P's execution, then V is typable in Γ. Extrusion of restricted channels is thus not allowed (See Section 6).

In the sequel we use the notation $\widetilde{V}$ and $\widetilde{\tau}$ to denote either a vector or a set of values $\widetilde{V}$ resp. types $\widetilde{\tau}$, and $\widetilde{V} : \widetilde{\tau}$ to denote a vector or set of bindings, where the arities of both terms are assumed to be identical. Likewise, we write $P \backslash \widetilde{a}$ for $P \backslash a_1 \backslash \ldots \backslash a_n$ if $\widetilde{a} = \{a_1, \ldots, a_n\}$. Also, $\Gamma \vdash \widetilde{V} : \widetilde{\tau}$ denotes $\Gamma \vdash V : \tau$ for every $V : \tau \in \widetilde{V} : \widetilde{\tau}$.

Definition 1 (Γ-Derivative). Suppose $\Gamma \vdash P : \pi = [\Delta]$. Then we say that P' is a *Γ-derivative* of P if either $P \Longrightarrow P'$ or one of the following two conditions hold:

(i) $P \Longrightarrow P''$, $P'' \equiv (\nu\widetilde{a} : \widetilde{\sigma_a})(u?(\widetilde{x} : \widetilde{\tau}).Q \mid R)$, $\Delta(u) = \langle(\widetilde{\tau'}), S\rangle$, $\Gamma \vdash \widetilde{V} : \widetilde{\tau'}$, $\Gamma, \widetilde{a} : \widetilde{\sigma_a} \vdash \widetilde{\tau'} \leq \widetilde{\tau}$, and P' is a Γ-derivative of $(\nu\widetilde{a} : \widetilde{\sigma_a})(Q\{\widetilde{V}/\widetilde{x}\} \mid R)$;

(ii) $P \Longrightarrow P''$, $P'' \equiv (\nu\widetilde{a} : \widetilde{\sigma_a})(u!\langle\widetilde{V}\rangle).Q \mid R)$, $\Gamma \vdash \widetilde{V} : \widetilde{\tau}$, and P' is a Γ-derivative of $(\nu\widetilde{a} : \widetilde{\sigma_a})(Q \mid R)$.

Note that in the definition of Γ-derivative we require that the objects of communication V are typable in Γ, thus precluding channel extrusion.

Lemma 4. *Let P be any value-passing process such that $\Gamma \vdash P : \mathcal{C}_\Gamma(\pi)$ for some π. Then, for any Γ-derivative P' of P, it holds that $\Gamma \vdash P' : \mathcal{C}_\Gamma(\pi)$.*

Proof. By the subject reduction lemma, it is enough that $\mathcal{C}_\Gamma(\pi)$ is preserved in the cases specified in conditions (i) and (ii) in Definition 1.

We show case (i). We have to show that if

$$\Gamma \vdash (\nu\widetilde{a} : \widetilde{\sigma})(u?(\widetilde{x} : \widetilde{\tau}).Q \mid R) : \mathcal{C}_\Gamma(\pi) \tag{1}$$

then

$$\Gamma \vdash (\nu\widetilde{a} : \widetilde{\sigma})(Q\{\widetilde{V}/\widetilde{x}\} \mid R) : \mathcal{C}_\Gamma(\pi)$$

where by definition of Γ-derivative

$$\Gamma \vdash \widetilde{V} : \widetilde{\tau'} \tag{2}$$

where $\mathcal{C}_\Gamma(\pi) = [\Delta]$ and $\Delta(u) = \langle(\tau'_1, \ldots, \tau'_n), S\rangle$, and thus $\Gamma, \widetilde{a} : \widetilde{\sigma} \vdash \widetilde{\tau'} \leq \widetilde{\tau}$. By (RES) and (PAR) we obtain from (1)

$$\Gamma, \widetilde{a} : \widetilde{\sigma} \vdash u?(\widetilde{x} : \widetilde{\tau}).Q : \pi' \tag{3}$$

and

$$\Gamma, \widetilde{a} : \widetilde{\sigma} \vdash R : \pi' \tag{4}$$

for some π' such that $\pi' \backslash \widetilde{a} = \mathcal{C}_\Gamma(\pi)$. From (3), by (IN)

$$\Gamma, \widetilde{a} : \widetilde{\sigma}, \widetilde{x} : \widetilde{\tau} \vdash Q : \pi', \widetilde{x} : \widetilde{\tau} \tag{5}$$

Obviously

$$\Gamma \vdash [u : (\widetilde{\tau'})^{\mathtt{I}}] \leq \mathcal{C}_\Gamma(\pi)$$

By Lemma 3, we deduce from this and (2) that for every $V : \tau' \in \widetilde{V} : \widetilde{\tau'}$ such that τ' is a channel type

$$\Gamma \vdash [V : \tau'] \leq \mathcal{C}_\Gamma(\pi) \tag{6}$$

By the substitution lemma, subsumption, (2) and (5) we obtain

$$\Gamma, \widetilde{a} : \widetilde{\sigma} \vdash Q\{\widetilde{V}/\widetilde{x}\} : \pi', \widetilde{V} : \widetilde{\tau}$$

By subsumption

$$\Gamma, \widetilde{a} : \widetilde{\sigma} \vdash Q\{\widetilde{V}/\widetilde{x}\} : \pi', \widetilde{V} : \widetilde{\tau'} \tag{7}$$

By subsumption we also get from (4)

$$\Gamma, \widetilde{a} : \widetilde{\sigma} \vdash R : \pi', \widetilde{V} : \widetilde{\tau'} \tag{8}$$

By (PAR) and repeated use of (RES) we obtain from (7) and (8)

$$\Gamma \vdash (\nu \widetilde{a} : \widetilde{\sigma})(Q\{\widetilde{V}/\widetilde{x}\} \mid R) : \mathcal{C}_\Gamma(\pi), \widetilde{V} : \widetilde{\tau'}$$

since no $a \in \widetilde{a}$ occur in $\widetilde{V}, \widetilde{\tau'}$ and thus $(\pi', \widetilde{V} : \widetilde{\tau'}) \backslash \widetilde{a} = \pi' \backslash \widetilde{a}, \widetilde{V} : \widetilde{\tau'} = \mathcal{C}_\Gamma(\pi), \widetilde{V} : \widetilde{\tau'}$. From this and from (6) we obtain

$$\Gamma \vdash (\nu \widetilde{a} : \widetilde{\sigma})(Q\{\widetilde{V}/\widetilde{x}\} \mid R) : \mathcal{C}_\Gamma(\pi)$$

as desired.

We show now (ii). We have to show that if

$$\Gamma \vdash (\nu \widetilde{a} : \widetilde{\sigma})(u!\langle \widetilde{V} \rangle.Q \mid R) : \mathcal{C}_\Gamma(\pi) \tag{9}$$

then

$$\Gamma \vdash (\nu \widetilde{a} : \widetilde{\sigma})(Q \mid R) : \mathcal{C}_\Gamma(\pi)$$

By definition of Γ-derivative

$$\Gamma \vdash \widetilde{V} : \widetilde{\tau}$$

By (RES) and (PAR) we obtain from (9)

$$\Gamma, \widetilde{a} : \widetilde{\sigma} \vdash u!\langle \widetilde{V} \rangle.Q : \pi' \tag{10}$$

and

$$\Gamma, \widetilde{a} : \widetilde{\sigma} \vdash R : \pi' \tag{11}$$

where $\pi' \backslash \widetilde{a} = \mathcal{C}_\Gamma(\pi)$. From (10), by (OUT)

$$\Gamma, \widetilde{a} : \widetilde{\sigma} \vdash Q : \pi'$$

Hence, by (PAR) and repeated use of (RES) we obtain from (10) and (11)

$$\Gamma \vdash (\nu \widetilde{a} : \widetilde{\sigma})(Q\{\widetilde{V}/\widetilde{x}\} \mid R) : \mathcal{C}_\Gamma(\pi)$$

as desired. □

Lemma 4 guarantees that the type $\mathcal{C}_\Gamma(\pi)$ is preserved for the derivatives of any value-passing process.

We can now show that for any value-passing process all occurrences of the filter operator in $\mathcal{T}_\Gamma(P)$ are transparent. In order to do this, we introduce $\mathcal{T}'_\Gamma$, an operator from processes in $\lambda\pi_\mathrm{v}$ to processes in $\lambda\pi\mathrm{F}_\mathrm{v}$, which is defined as $\mathcal{T}_\Gamma$ with the sole exception that the rule

$$\mathcal{T}^S_\Gamma(u) = \text{if } u \in \mathtt{dom}(S) \text{ then } \mathcal{F}_\Gamma(u, S(u)) \text{ else } u$$

in Fig. 5 is substituted by the rule

$$\mathcal{T}'^S_\Gamma(u) = \text{if } u \in \mathtt{dom}(S) \text{ then } \mathcal{F}'_\Gamma(u, S(u)) \text{ else } u$$

where $\mathcal{F}'_\Gamma$ is defined exactly as $\mathcal{F}_\Gamma$, except that the rule

$$\mathcal{F}_\Gamma(P, [\Delta]) = P\lceil \mathcal{I}_\Gamma([\Delta])$$

in Fig. 5 is substituted by the rule

$$\mathcal{F}'_\Gamma(P, [\Delta]) = P.$$

Hence the filter operator does not occur in $\mathcal{T}'_\Gamma(P)$. In order to show that $\mathcal{T}_\Gamma(P)$ and $\mathcal{T}'_\Gamma(P)$ have identical behaviour within contexts that are translations of contexts in $\lambda\pi_\mathrm{v}$ we have only to show that all occurrences of the filter operator in $\mathcal{T}_\Gamma(P)$ are transparent. A labelled transition system for the value-passing calculus may be defined simply by labelling the transitions in both the premiss and the conclusion of the rule (res) in Fig. 4, with the requirement that the subject, if any, of the transition label is not identical to the restricted channel. The labelled transition trees thus obtained for $\mathcal{T}_\Gamma(P)$ and $\mathcal{T}'_\Gamma(P)$ are isomorphic, in the sense that there is a bijection between the set of labelled transitions for each pair of identified nodes. The transition labels may not be identical, and two labels should be identified if one is equal to $\mathcal{T}_\Gamma(\widetilde{V})$ and the other to $\mathcal{T}'_\Gamma(\widetilde{V})$, for any $\widetilde{V} \in \lambda\pi_{\mathrm{v}}$.

It is in this very strict sense that we define the notion of LT-bisimilarity here:

Definition 2 (LT-equivalence). Two agents P and Q in the $\lambda\pi\mathrm{F}_{\mathrm{v}}$-calculus which are both translations from value-passing agents in the $\lambda\pi_{\mathrm{v}}$-calculus, are LT-bisimilar, briefly $P \sim Q$, if their corresponding labelled transitions trees are isomorphic, where two labels are identified if for any $\widetilde{V}$ in $\lambda\pi_{\mathrm{v}}$ one of them is equal to $\mathcal{T}_\Gamma(\widetilde{V})$ and the other is equal to $\mathcal{T}'_\Gamma(\widetilde{V})$.

We define Erase(P), for any $P \in \lambda\pi_{\mathrm{v}}$, as the agent obtained from P by erasing all type annotations in P. Note that Erase(P) is a term in $\lambda\pi\mathrm{F}_{\mathrm{v}}$. We say that $P \approx_\eta Q$, where $P \in \lambda\pi_{\mathrm{v}}$ and $Q \in \lambda\pi\mathrm{F}_{\mathrm{v}}$, if there is a η-reduction identifying Erase(P) and Q. We define η-reduction as in the λ-calculus, i.e. $\lambda x.Mx = M$. Finally, we write $P =_\eta Q$ if $P \approx_\eta\sim Q$.

Lemma 5. *For any Γ and P such that $\Gamma \vdash P : \pi$, it holds that $P \approx_\eta \mathcal{T}'_\Gamma(P)$.*

Proof. Since $\mathcal{T}'_\Gamma(P)$ is a homomorphism over all operators, the only case that needs attention is when P is a higher-order variable. This boils down to showing that $\mathcal{F}_\Gamma(P,\tau) \approx_\eta P$. The only case that does not follow directly by definition is when the type τ of P is either $\sigma_H \to \rho$ or $(y:\sigma) \to \rho$. We show the latter case, the former being similar. The proof in by induction on the formation of τ. Since $\mathcal{F}_\Gamma(P,\sigma_H \to \rho) = \lambda(y)\mathcal{F}_\Gamma(Py,\rho)$, we obtain $\mathcal{F}_\Gamma(Py,\rho) \approx_\eta Py$, either by the induction hypothesis if τ is a HO-Value type $(y:\sigma) \to \rho$ or $\sigma_H \to \rho$, or otherwise by definition if τ is a process type π or a base type σ_B. Hence $\lambda(y)\mathcal{F}_\Gamma(Py,\rho) \approx_\eta \lambda(y)Py \approx_\eta P$ as desired. □

Lemma 6. *For any value-passing process P in $\lambda\pi_v$ it holds that $\mathcal{T}_\Gamma(P) \sim \mathcal{T}'_\Gamma(P)$.*

Proof. The result follows from Lemma 4. □

Basically, $P \sim P\lceil L$ if P never executes an action along a channel that is not specified in L. The relation $\sim$ is a congruence in the value-passing process since in this setting variables are never extruded. This congruence extends thus to terms in the object position of output actions.

We obtain thus the main result of this section:

Theorem 1. *For any process P in the value-passing $\lambda\pi F_v$, it holds that $P =_\eta \mathcal{T}_\Gamma(P)$.*

In this sense we may assert that the behaviours of P and $\mathcal{T}_\Gamma(P)$ are equivalent within contexts that are well-typed in Γ. Any reasonable notion of behaviour equivalence should relate P and $\mathcal{T}_\Gamma(P)$. Of course, the encoding is not fully abstract, nor it is intended to be. The objective was to show that $\mathcal{T}_\Gamma(P)$ behaves as P in any context consisting of agents that are translations of agents in $\lambda\pi_\mathrm{v}$, but that the security requirements expressed by type annotations in P are preserved by $\mathcal{T}_\Gamma(P)$ in any arbitrary environment in $\lambda\pi\mathrm{F}_\mathrm{v}$, reflecting the fact that dynamic control allows more processes to execute than static control.

6 Conclusion

The main result of this paper concerns only value-passing agents, i.e. agents in which channels are not extruded. As an illustration of the kind of difficulties involved in channel extrusion, consider an agent P and an environment Γ such that $\Gamma \vdash P : [u : (((\mathtt{int})^{\mathtt{IO}})^{\mathtt{I}})^{\mathtt{O}}]$. Then we may have

$$P = (\nu b : ((\mathtt{int})^{\mathtt{IO}})^{\mathtt{I}})(u!\langle b\rangle.b?(x : (\mathtt{int})^{\mathtt{IO}})(x!\langle 3\rangle \mid x?(z)\mathbf{0})).$$

Hence any channel of type $(\mathtt{int})^{\mathtt{IO}}$ might become part of the interface of a derivative of P, a fact which is not immediately reflected by the type $[u : (((\mathtt{int})^{\mathtt{IO}})^{\mathtt{I}})^{\mathtt{O}}]$.

This restriction is not a limitation of the translation, but rather inherent to the typing of $\lambda\pi_\mathrm{v}$-terms. The typing rules of the $\lambda\pi_\mathrm{v}$ calculus eliminate information about channels bound by input prefix or by restriction (see Discussion in [16]). Only channels that can be referenced may become part of the interface of a process. Restricted channels, by their very nature, cannot be referenced by name, and thus cannot become part of the interface. Analogously, the filter operator cannot filter actions along channels that cannot be referenced by name.

The typing system of the $\lambda\pi_\mathrm{v}$-calculus can be refined with a notion of interface type where information is kept about the type of channels that may be imported via input actions. Indeed, the notion of closure that we introduce in this paper is nothing more than an indirect way of partially recovering this kind of information. Ideally, the type of a process should be invariant with regard to any kind of transitions, including input events. The result should be a general version of the subject reduction theorem that extends to any kind of labelled transitions. We leave this for future investigation.

It is important to note that the translation is not intended to be fully abstract. The objective of the translation is to show how security requirements that have been determined in terms of typing constraints may be dynamically enforced with the aid of screening operators. A decidable type system will necessarily reject processes that are totally harmless. For instance, the channel b belongs to the interface of the agent $(\nu a)a!\langle b\rangle$, although this agent can be identified with $\mathbf{0}$. The translation yields an agent that never rejects any agent on account of its interface type, but simply execute it in a safe environment that dynamically screens out undesirable actions.

References

1. Luca Cardelli and Andrew D. Gordon. Mobile ambients. In Maurice Nivat, editor, *Proceedings of FoSSaCS '98*, volume 1378 of *LNCS*, pages 140–155. Springer, 1998.

2. Takis Hartonas and Matthew Hennessy. Full abstractness for a functional/concurrent language with higher-order value-passing. Journal of Functional Programming.
3. Matthew Hennessy. Fully abstract denotational model for higher-order processes. Info. & Comp., 112:55–95, 1994.
4. Matthew Hennessy and James Riely. Trust and partial typing in open systems of mobile agents. In *POPL99*, pages 93–104.
5. Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. In *Proceedings of HLCL '98*, volume 16.3 of *ENTCS*, pages 3–17. Elsevier Science Publishers, 1998. Full version to appear in *Info. &. Comp.*
6. José Luis Vivas. *Dynamic Binding of Names in Calculi for Mobile Processes.* PhD thesis, Royal Institute of Technology, 2001.
7. R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24.
8. De Nicola, Ferrari, Pugliese, and Venneri. Types for access control. *TCS*, 240, 2000.
9. Scott Oaks. *Java Security.* O'Reilly & Associates, 2nd edition, 2001.
10. Panel. Foundations of object-based concurrent programming. In *Addendum to the proceedings, OOPSLA/ECOOP-90: Conference on Object-Oriented Programming: Systems*, pages 9–14, Languages, and Applications / European Conference on Object-Oriented Programming, Ottawa, Canada, 1990.
11. Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.
12. Davide Sangiorgi. From π-calculus to Higher-Order π-calculus — and back. In Marie-Claude Gaudel and Jean-Pierre Jouannaud, editors, *Proceedings of TAPSOFT '93*, volume 668 of *LNCS*, pages 151–166. Springer, 1993.
13. Jose-Luis Vivas and Mads Dam. From higher-order pi-calculus to pi-calculus in the presence of static operators. In *Proceedings of CONCUR '98*, volume 1466 of *LNCS*, pages 115–130. Springer, September 1998.
14. Dan Seth Wallach. *A New Approach to Mobile Code Security.* PhD thesis, Princeton University, 1999.
15. Nobuko Yoshida and Matthew Hennessy. Subtyping and locality in distributed higher order processes. In *CONCUR'99*, volume 1664 of *LNCS*, pages 557–572. Springer, August 1999.
16. Nobuko Yoshida and Matthew Hennessy. Assigning types to processes. *To appear in Info. & Comp*, 2002. An extended abstract appeared in the *LICS'01*, 334–345, 2001.