Towards Gradually Typed Capabilities in the Pi-Calculus

Matteo Cimini
University of Massachusetts Lowell
Lowell, MA, USA
matteo_cimini@uml.edu

Gradual typing is an approach to integrating static and dynamic typing within the same language, and puts the programmer in control of which regions of code are type checked at compile-time and which are type checked at run-time.

In this paper, we focus on the π-calculus equipped with types for the modeling of input-output capabilities of channels. We present our preliminary work towards a gradually typed version of this calculus. We present a type system, a cast insertion procedure that automatically inserts run-time checks, and an operational semantics of a π-calculus that handles casts on channels. Although we do not claim any theoretical results on our formulations, we demonstrate our calculus with an example and discuss our future plans.

1 Introduction

This paper presents preliminary work on integrating dynamically typed features into a typed π-calculus. Pierce and Sangiorgi have defined a typed π-calculus in which channels can be assigned types that express input and output capabilities [16]. Channels can be declared to be input-only, output-only, or that can be used for both input and output operations. As a consequence, this type system can detect unintended or malicious channel misuses at compile-time.

The static typing nature of the type system prevents the modeling of scenarios in which the input-output discipline of a channel is discovered at run-time. In these scenarios, such a discipline must be enforced with run-time type checks in the style of dynamic typing.

Gradual typing provides a natural solution to do just that, as it advocates the integration of dynamic typing and static typing [18, 24]. In gradual typing the programmer is in control of which regions of code are type checked at compile-time and which are given a pass, with the promise to perform checks at run-time when actual values are available.

In this paper, we present our preliminary work to equip Pierce's and Sangiorgi's typed π-calculus with gradual typing. Our contributions are:

- We present a formulation of a gradual type system for capabilities in π-calculus. This type system allows for channels to be declared dynamically typed, that is, their capabilities will be only known at run-time. Accordingly, the type system is permissive when dynamically typed channels occur, but still rejects processes if strong inconsistencies are found at compile-time.
- We offer a cast insertion procedure: A run-time check is inserted at any spot in which the gradual type checker has been optimistic and gave a pass. We make use of casts on channels to perform run-time checks.
- We present a formulation of a π-calculus with cast channels. This is an operational semantics that extends that of the π-calculus to handle casts on channels and detect when they fail or succeed.

M. Bartoletti, L. Henrio, A. Mavridou, A. Scalas (Eds.):
12th Interaction and Concurrency Experience (ICE 2019).
EPTCS 304, 2019, pp. 61–76, doi:10.4204/EPTCS.304.5

© Matteo Cimini
This work is licensed under the
Creative Commons Attribution License.

In this paper we do not claim any theoretical results on our formulations, hence the "*Towards*" part of the title. We offer this preliminary work, demonstrate it with an example, and discuss our future plans.

The following sections are organized as follows. Section 2 reviews the typed π-calculus of Pierce and Sangiorgi and motivates the benefits of adding gradual typing. Section 3 shows our formalisms for gradually typed capabilities in π-calculus. Section 4 applies our formalisms to an example. Section 5 discusses the properties that we plan to establish in the future. Section 6 discusses related work, and Section 7 concludes the paper.

2 Background

We review the type system of Pierce's and Sangiorgi's typed π-calculus with the example that the authors offered in [16]. The scenario is that of a printer server P that interacts with multiple clients $C_1, \ldots, C_n$. The printer provides a channel p over which clients can communicate their job requests. The printing system can be described with the process $\nu p.(P \mid C_1 \mid C_2 \mid \ldots \mid C_n)$. Suppose a client $C_1 \equiv \overline{p}\langle j_1\rangle.\overline{p}\langle j_2\rangle$ requests jobs j_1 and j_2 to be printed. A malicious client $C_2 \equiv p(j).C_2$ could very well interject the job requests of C_1, and pretend to be the printer. In the π-calculus with capabilities this scenario can be prevented by enforcing that clients used p only in output operations. More generally, a channel can be given three kinds of capability types[1]: $\mathtt{o}\cdot(T_1,\ldots,T_n)$, which means that the channel can be used output-only for n channels of types $T_1,\ldots,T_n$ (these types are capabilities themselves), $\mathtt{i}\cdot(T_1,\ldots,T_n)$, which means that the channel can be used input-only for n arguments of types $T_1,\ldots,T_n$, and $\mathtt{either}\cdot(T_1,\ldots,T_n)$, which means that the channel can be used both for input and output operations. If we were to use the type system to the printing example, we would type check the clients separately, and would do so to impose the restriction that p must be of type $\mathtt{o}\cdot(T)$, where T is the type of the print request (irrelevant here). Thanks to this typing discipline the malicious client C_2 would be rejected. The printer system has good guarantees just by knowing that clients type check successfully, without inspecting the code of the clients.

Decisions at Run-time Let us consider the following example. After submitting a tax return, a tax payer either receives a refund or must send a payment to a revenue agency. For the sake of the example, suppose that the revenue agency operates with limited resources and can only share one channel (b below) with the tax payer. This channel is then used to either receive or send a payment. The tax payer is left free to write the application that interacts with the revenue agency. Because of this, we certainly should impose a suitable discipline on channels, or we may leave ourselves open to malicious scenarios akin to those of the printer example. Therefore, we shall use the π-calculus with capability types.

The revenue agency A can be modeled as

$$\begin{aligned} A \equiv\ & \nu(x:\mathtt{o}\cdot T).\overline{r}\langle x\rangle.\overline{x}\langle\$100\rangle \\ & + \\ & \nu(x:\mathtt{i}\cdot T).\overline{r}\langle x\rangle.x(sum) \end{aligned} \tag{1}$$

The channel r sends the newly created channel x to the client. The type T is the type of the channel representing money (irrelevant here).

Remark 2.1. *To simplify our example, we assume that the choice operation $+$ picks the correct branch (the branch for sending a tax refund or the branch for waiting for a payment).*

[1] We use a slightly different notation than that of [16].

In our situation, which branch of the + operation will fire will be known at run-time. A client C can be modeled as follows.

$$C \equiv t(b : \textit{what type?}).\ (\overline{b}\langle\$100\rangle\ +\ b(\textit{sum})) \tag{2}$$

After receiving the channel b, the tax payer client is ready to either send or receive payments, depending on what action the revenue agency makes available.

What type should we assign to the channel b? Were we to declare b to be input-only, the process C would be rejected because it uses b in an output operation (in $\overline{b}\langle\$100\rangle$). Were we to declare b to be output-only, C would be similarly rejected. If we want to use the type system with capabilities and have a chance to pass the type checker we have no choice but to declare channel b as `either`. However, this has the effect to simply leave the tax payer client free to use b with no restrictions. The client application could then exploit unintended mistakes of the revenue agency process. Consider for example the following client.

$$C \equiv t(b : \texttt{either} \cdot T).\ (\overline{b}\langle\$100\rangle\ .b(\textit{sum})\ +\ b(\textit{sum})) \tag{3}$$

This version of the client is well-typed in the typed π-calculus of Pierce and Sangiorgi. Furthermore, it does, in all appearances, what is expected: It does pay the whole sum when asked to, and it does receive the whole sum when asked to. However, when the client is mandated to send payments, it silently also tries to sneak in a reading operation on channel b, on the off chance that the revenue agency reuses the same channel by mistake and some other tax payer would send money over b.

Allowing `either` is not an option, and neither it is the use of any of the other types.

Integrating Gradual Typing The problem with the example above is that the type of channel b is not known at compile-time. On the contrary, it can only be discovered at run-time. We therefore integrate Pierce's and Sangiorgi's typed π-calculus with dynamic typing features, where type checking is postponed at run-time. We do so by adopting the gradual typing style, an approach for integrating static and dynamic typing within the same language [18].

We introduce the possibility to assign the dynamic type $\star$ ([8]) to channels. This means that the capability of the channel is not known at compile time. When the channel is used at run-time it will find itself instantiated with a well-determined channel, and the moment this channel is used we verify that its capabilities allow for the operation required.

In our calculus, the client application can be modeled as follows.

$$C \equiv r(b : \star).\ (\overline{b}\langle\$100\rangle\ +\ b(\textit{sum})) \tag{4}$$

When the type checker encounters the sub-expressions $\overline{b}\langle\$100\rangle$ and $b(\textit{sum})$ it essentially gives them a pass, as they cannot be resolved at compile time. However, those checks must be performed at run-time. To perform such checks we follow the standard approach in gradual typing, thus we have a compilation step that inserts run-time checks. A popular mechanism for performing these checks is through the means of casts [19, 22]. This procedure is typically called *cast insertion* in gradual typing. After cast insertion, the client becomes

$$(r(b : \star).\ (\overline{(\ b : \star \Rightarrow \mathtt{o} \cdot T\ )}\langle\$100\rangle\ +\ (\ b : \star \Rightarrow \mathtt{i} \cdot T\ )(\textit{sum})) \tag{5}$$

Cast channels are highlighted: Casts wrap around channels and have the form $a : T_1 \Rightarrow T_2$, which means that the channel a is of type T_1 and is cast to the type T_2. Cast channels are used in lieu of ordinary channels as the subject of input and output operations, and they can also be communicated in output.

Above, $b : \star \Rightarrow \mathtt{o} \cdot T$ means that at compile-time we only knew that b was of dynamic type but at run-time it needs to be a channel with output capabilities. $b : \star \Rightarrow \mathtt{i} \cdot T$ is analogous.

When using Pierce's and Sangiorgi's calculus we frequently type check a part of the process under some capability assignments, and other parts with another. For example, we type check the clients separately. It makes little sense for the printer in the printer example to be type checked with the same assignments as clients, as types must be dual, at least. We inherit the same workflow. However, we need some extra care because the server needs to tell clients that some specific channels must be used with the opposite operation than that which the server performs. The server needs to advertise a reversed type for a channel. To enable this scenario, we augment the π-calculus with a tagged output $\overline{r}^{\mathscr{R}}\langle x \rangle$.

The agency process is then the following.

$$\begin{aligned} A \equiv\ & \nu(x : \mathtt{o} \cdot T).\, \overline{r}^{\mathscr{R}}\langle x \rangle\, .\overline{x}\langle \$100 \rangle \\ & + \\ & \nu(x : \mathtt{i} \cdot T).\, \overline{r}^{\mathscr{R}}\langle x \rangle\, .x(sum) \end{aligned} \tag{6}$$

To make an example, when the first branch of the choice operator fires the channel x is declared output-only. The output $\overline{r}^{\mathscr{R}}\langle x \rangle$ sends the channel x but also informs the receiver that x must be used as input-only. (The second branch follows analogous lines.) We achieve this scenario by having the compilation step inserting a cast to that effect. Intuitively, the agency will have the following casts

$$\begin{aligned} & \nu(x : \mathtt{o} \cdot T).\overline{r}\langle\, (x : \mathtt{i} \cdot T \Rightarrow \star)\, \rangle.\overline{x}\langle \$100 \rangle \\ & + \\ & \nu(x : \mathtt{i} \cdot T).\overline{r}\langle\, (x : \mathtt{o} \cdot T \Rightarrow \star)\, \rangle.x(sum) \end{aligned} \tag{7}$$

Besides the casts, it is to notice that the special reverse output, precious in driving the cast insertion, has been compiled away into an ordinary output.

When we run $(A \mid C)$ with the casts as in (5) and (7), we need to do so in an operational semantics that handles cast channels. We formulate such semantics in Section 3. This formulation is capable of checking when casts succeed or fail at run-time. To make an example, let us suppose that the second branch of (7) occurs, i.e. the tax payer owes a payment. In one step the client would receive a cast channel in input and would become: (As a notational convenience, we collapse sequences of casts such as $(x : \mathtt{o} \cdot T \Rightarrow \star) : \star \Rightarrow \mathtt{o} \cdot T$ with the notation $x : \mathtt{o} \cdot T \Rightarrow \star \Rightarrow \mathtt{o} \cdot T$.

$$\overline{(\, x : \mathtt{o} \cdot T \Rightarrow \star \Rightarrow \mathtt{o} \cdot T\,)}\langle \$100 \rangle \ +\ (\, x : \mathtt{o} \cdot T \Rightarrow \star \Rightarrow \mathtt{i} \cdot T\,)(sum)$$

Notice that the whole channel, with the cast around, has been passed. Now this process must interact with the remaining part of the agency, which is $x(sum)$. However, the subject of the output $x : \mathtt{o} \cdot T \Rightarrow \star \Rightarrow \mathtt{o} \cdot T$ is not a bare channel. Therefore our operational semantics resolves these casts first, and checks that the channel buried inside the casts actually is of type $\mathtt{o} \cdot T$, which is requested at the end target of the cast. This is the case, and our operational semantics simply removes casts, which leads the client to take a step to $\overline{x}\langle \$100 \rangle$. At this point, the ordinary communication rule of the π-calculus can take place. If, by mistake, another client $\overline{x}\langle \$5000 \rangle$ were around, C could not take its payment. Indeed, if the branch $(x : \mathtt{o} \cdot T \Rightarrow \star \Rightarrow \mathtt{i} \cdot T)(sum)$ were selected for a communication, our operational semantics first would detect that an output-only channel is being used for an input operation, and would trigger a cast error at run-time.

The ability of handling capabilities that are unknown at compile-time, established at run-time, and protected during execution, is not possible in the π-calculus of Pierce and Sangiorgi. We have therefore enhanced that calculus with gradual typing in the next section.

3 Gradually Typed Capabilities in the Pi-calculus

3.1 Gradual Type System

In this section we present our formalisms for a π-calculus with gradually typed capabilities. Fig. 1 shows the syntax and the type system of our calculus, $\pi^\star$. The figure highlights the relevant parts of the system. The syntax is based on that of [16] (though we omit recursive types, as discussed in Section 6). The additions are the type $\star$ and the reverse output described in the previous section. The type system has the form $\Gamma \vdash P : \mathtt{ok}$, where Γ contains associations from channels to their capabilities. This typing judgement means that under the assignments of Γ the process P is well-typed. Programmers type check their process P by first providing Γ with the capability associations for free channels. Further associations may be added during type checking when the type system traverses new bindings in restrictions ν and input operations.

Most of the design of Fig. 1 mirrors the formulation of Pierce and Sangiorgi [16], and much credits go to that formulation. One of the main differences is that our typing rules make use of type consistency $\sim$ ([18, 2]) in rules (T-IN) and (T-OUT), whereas the type system in [16] makes use of subtyping. Type consistency $T_1 \sim T_2$ holds when the two types are recursively the same type modulo the fact that one of them may have $\star$ instead of a concrete type. This is key to model the fact that the type system is liberal at the encounter of $\star$. We have that $\star \sim \mathtt{o}\cdot(T_1,\ldots,T_n)$, as well as $\mathtt{o}\cdot(T_1,\ldots,T_n) \sim \mathtt{o}\cdot(T_1,\ldots,T_n)$. However, the relation is not too liberal, and we have $\mathtt{i}\cdot(T_1,\ldots,T_n) \not\sim \mathtt{o}\cdot(S_1,\ldots,S_n)$. In contrast to subtyping, $\sim$ is symmetric, and is not transitive.

3.2 Cast Insertion Procedure

The gradual type system takes a permissive view at the encounter of $\star$. However, the checks that are not possible at compile-time must take place at run-time. For this reason we compile the program and insert casts. Fig. 2 shows the cast insertion procedure, and highlights the casts that are inserted. This procedure is formulated with a judgement of the form $\Gamma \vdash P \rightsquigarrow P' : \mathtt{ok}$, which means that under the assignments declared in Γ, P is well-typed and compiles into P'. P' is essentially P but contains casts around channels. A cast $a : T_1 \Rightarrow T_2$ means that the channel a is of type T_1 and is cast to the type T_2. Notice that P' is not a process of $\pi^\star$, as $\pi^\star$ does not have a cast operator. P' is a process of the calculus $\pi^\star_c$, which we define in the next section and which does have casts. In a nutshell, $\pi^\star$ acts as the surface language that programmers use. Ideally, its counterpart $\pi^\star_c$ with casts is not visible to programmers, though it is the end calculus that executes programs. Essentially, programmers do not handle casts explicitly.

In Fig. 2 casts are placed when two types are compared by $\sim$ rather than plain equality. Furthermore, the reverse output generates a cast around for the output channel to label it as sending channels with the reverse type. As we shall see, from that point there is a reduction step that moves these casts to the arguments, so that the arguments are set to advertise their reverse type to the receiver. In Fig 2, the operation $\Gamma(a)^{-1}$ provides the reverse type of a capability. Once the cast is generated, the reverse output $\overline{a}^{\mathscr{R}}$ is compiled away and becomes an ordinary output $\overline{a}$. It is worth pointing out that implementations can detect when casts are trivial, i.e. $c : T \Rightarrow T$, and omit generating them.

$$
\begin{array}{lrcl}
\text{Types} & T,S & ::= & I\cdot(T_1,\ldots,T_n) \mid \star \\
\text{Capabilities} & I & ::= & \mathtt{i} \mid \mathtt{o} \\
\text{Processes} & P & ::= & \mathbf{0} \mid\ a(a_1:T_1,\ldots,a_n:T_n).P \mid \overline{a}\langle a_1,\ldots,a_n\rangle.P \mid \overline{a}^{\mathscr{R}}\langle a_1,\ldots,a_n\rangle.P \\
 & & & \mid\ P \mid P \mid\ P+P \mid\ (\nu a:T).P \mid\ !P \\
\text{Type Environment} & \Gamma & ::= & \emptyset \mid \Gamma, a:T
\end{array}
$$

Type System $\boxed{\Gamma \vdash P : \mathtt{ok}}$

$$\Gamma \vdash \mathbf{0} : \mathtt{ok}$$

$$\frac{\Gamma \vdash P_1 : \mathtt{ok} \quad \Gamma \vdash P_2 : \mathtt{ok}}{\Gamma \vdash P_1 + P_2 : \mathtt{ok}} \qquad \frac{\Gamma \vdash P_1 : \mathtt{ok} \quad \Gamma \vdash P_2 : \mathtt{ok}}{\Gamma \vdash P_1 \mid P_2 : \mathtt{ok}}$$

$$\text{(T-RES)}\ \frac{\Gamma, a:T \vdash P : \mathtt{ok}}{\Gamma \vdash (\nu a:T).P : \mathtt{ok}} \qquad \frac{\Gamma \vdash P : \mathtt{ok}}{\Gamma \vdash\ !P : \mathtt{ok}}$$

$$\text{(T-IN)}\ \frac{\Gamma(a) \sim \mathtt{i}\cdot(T_1\cdot\ldots\cdot T_n) \qquad \Gamma, a_1:T_1,\ldots,a_n:T_n \vdash P : \mathtt{ok}}{\Gamma \vdash\ a(a_1:T_1,\ldots,a_n:T_n).P : \mathtt{ok}}$$

$$\text{(T-OUT)}\ \frac{\Gamma(a) \sim \mathtt{o}\cdot(\Gamma(a_1)\cdot\ldots\cdot\Gamma(a_n)) \qquad \Gamma \vdash P : \mathtt{ok}}{\Gamma \vdash\ \{\overline{a}\ \mathtt{or}\ \overline{a}^{\mathscr{R}}\}\langle a_1,\ldots,a_n\rangle.P : \mathtt{ok}}$$

Type Consistency $\boxed{T \sim T}$

$$T \sim \star \qquad \star \sim T$$

$$\frac{T_1 \sim S_1 \quad \cdots \quad T_n \sim S_n}{\mathtt{i}\cdot(T_1,\ldots,T_n) \sim \mathtt{i}\cdot(S_1,\ldots,S_n)}$$

$$\frac{T_1 \sim S_1 \quad \cdots \quad T_n \sim S_n}{\mathtt{o}\cdot(T_1,\ldots,T_n) \sim \mathtt{o}\cdot(S_1,\ldots,S_n)}$$

Figure 1: Syntax and Type System of Gradually Typed Capabilities

Cast Insertion $\boxed{\Gamma \vdash P \rightsquigarrow P : \mathtt{ok}}$

$$\Gamma \vdash \mathbf{0} \rightsquigarrow \mathbf{0} : \mathtt{ok}$$

$$\frac{\Gamma \vdash P_1 \rightsquigarrow P_1' : \mathtt{ok} \qquad \Gamma \vdash P_2 \rightsquigarrow P_2' : \mathtt{ok}}{\Gamma \vdash P_1 \mid P_2 :\rightsquigarrow P_1' \mid P_2'\mathtt{ok}} \qquad \frac{\Gamma \vdash P_1 \rightsquigarrow P_1' : \mathtt{ok} \qquad \Gamma \vdash P_2 \rightsquigarrow P_2' : \mathtt{ok}}{\Gamma \vdash P_1 + P_2 :\rightsquigarrow P_1' + P_2'\mathtt{ok}}$$

$$\frac{\Gamma, a : T \vdash P \rightsquigarrow P' : \mathtt{ok}}{\Gamma \vdash (\nu a : T).P \rightsquigarrow (\nu a : T).P' : \mathtt{ok}} \qquad \frac{\Gamma \vdash P \rightsquigarrow P : \mathtt{ok}}{\Gamma \vdash\, !P \rightsquigarrow\, !P' : \mathtt{ok}}$$

(CI-IN)
$$\frac{\Gamma(a) \sim \mathtt{i} \cdot (T_1 \cdot \ldots \cdot T_n) \qquad \Gamma, a_1 : T_1, \ldots, a_n : T_n \vdash P \rightsquigarrow P' : \mathtt{ok}}{\Gamma \vdash a(a_1 : T_1, \ldots, a_n : T_n).P \rightsquigarrow \boxed{(a : \Gamma(a) \Rightarrow \mathtt{i} \cdot (T_1 \cdot \ldots \cdot T_n))}\, (a_1 : T_1, \ldots, a_n : T_n).P' : \mathtt{ok}}$$

(CI-OUT)
$$\frac{\Gamma(a) \sim \mathtt{o} \cdot (\Gamma(a_1) \cdot \ldots \cdot \Gamma(a_n)) \qquad \Gamma \vdash P \rightsquigarrow P' : \mathtt{ok}}{\Gamma \vdash \overline{a}\langle a_1, \ldots a_n \rangle.P \rightsquigarrow \boxed{\overline{(a : \Gamma(a) \Rightarrow \mathtt{o} \cdot (\Gamma(a_1) \cdot \ldots \cdot \Gamma(a_n)))}}\, \langle a_1, \ldots, a_n \rangle.P' : \mathtt{ok}}$$

(CI-ROUT)
$$\frac{\Gamma(a) \sim \mathtt{o} \cdot (\Gamma(a_1) \cdot \ldots \cdot \Gamma(a_n)) \qquad \Gamma \vdash P \rightsquigarrow P' : \mathtt{ok}}{\Gamma \vdash \overline{a}^{\mathscr{R}}\langle a_1, \ldots a_n \rangle.P \rightsquigarrow \boxed{\overline{(a : \Gamma(a) \Rightarrow \mathtt{o} \cdot (\Gamma(a_1)^{-1} \cdot \ldots \cdot \Gamma(a_n)^{-1}))}}\, \langle a_1, \ldots, a_n \rangle.P' : \mathtt{ok}}$$

Figure 2: Cast Insertion for Gradually Typed Capabilities

3.3 Operational Semantics for Channel Casts

Below we show the syntax of $\pi_c^\star$, our π-calculus with cast channels. The most relevant parts are highlighted.

Types	T,S	::=	$I\cdot(T_1,\ldots,T_n) \mid \star$
Capabilities	I	::=	$\mathtt{i} \mid \mathtt{o}$
Cast Channels	c	::=	$a \mid (c : T \Rightarrow T)$
Processes	P,Q	::=	$\mathbf{0} \mid (\nu a : T).P \mid c\,(a_1 : T_1, \ldots a_n : T_n).P \mid \overline{c}\langle c_1,\ldots,c_n\rangle.P$
			$\mid P+P \mid P \mid P \mid !P \mid \mathit{typeError}$
Type Environment	Γ	::=	$\emptyset \mid \Gamma, a : T$

Casts can be wrapped around channels, and, as a matter of fact, around sequences of cast channels. We can therefore have a channel being the subject of several consecutive casts. Cast channels can be used as the subject of input and output operations. Furthermore, channels can be communicated even if they have casts around.

Fig 3 contains the operational semantics of $\pi_c^\star$. This semantics extends that of the π-calculus by adding reduction rules to handle casts on channels. Fig 3 makes use of the structural congruence $\equiv$, which is standard and has been omitted. In the following, we comment on the relevant parts of the operational semantics.

(COMM) is the standard reduction rule for communication. We point out that it can fire only so long as the input and the output have a bare channel (no casts) as subject. Therefore, this rule cannot fire unless casts have been previously resolved by other reduction rules. We also point out that the rule allows for cast channels to be sent, and they are substituted in the body of the receiver with the usual parameter passing mechanism.

(C-SOLVE) applies when the subjects of the input and the output happen not to be ready because wrapped by some casts. In that case, we first need to check that casts are successful. As we shall see in the context of the other reduction rules, handling a successful cast may mean that nested casts must be distributed to the arguments of outputs. Small-step semantics is not ideal because in one step we may choose a communicating partner, and the next step we may choose a completely different output partner. It is incorrect, however, to partially distribute some casts to one process and some others to another. Therefore, (C-SOLVE) makes use of big-step semantics to commit to two communicating partners and resolve their casts in one big step. The `ch` function retrieves the channel name that is used at the bottom of the cast. We use `ch` to select two processes, one in input and one in output, that communicate through the same channel. Afterwards, the big-step reduction relation $\longrightarrow^c$ solves the casts of the output and the casts of the input. These are handled, in specific cases, by the reduction rules that we describe below.

(C-OUT-SUCCEED) applies when the subject of an output is wrapped in a cast. When the channel that is buried inside the cast does have output capabilities the cast is successful. Accordingly, we simply remove the cast. However, we cannot forget that the type of the channel had arguments, which too are supposed to match the requested types. Therefore, we distribute those casts to the arguments of the output. It is interesting to see that the direction of the cast is contravariant. It is so to ensure type preservation, as discussed in Section 5.

(C-OUT-FAIL), tool, applies when the subject of an output is wrapped in a cast. This rule detects a cast failure, that is, the channel that is buried inside the cast is not prescribed for outputs. In this case we trigger a run-time type error.

Reduction Semantics $\boxed{P \longrightarrow P}$

$$\frac{P_1 \longrightarrow P_1'}{P_1 \mid P_2 \longrightarrow P_1' \mid P_2} \qquad \frac{P \longrightarrow P'}{(\nu a : T).P \longrightarrow (\nu a : T).P'}$$

$$\frac{P \equiv P' \qquad P' \longrightarrow Q' \qquad Q' \equiv Q}{P \longrightarrow Q}$$

(COMM)
$$a(a_1 : T_1, \ldots, a_n : T_n).P \mid \overline{a}\langle c_1, \ldots, c_n\rangle.Q \longrightarrow P\{c_1/a_1, \ldots, c_n/a_n\} \mid Q$$

(C-SOLVE)
$$\frac{\begin{array}{c}\textit{either } c_1 \textit{ or } c_2 \textit{ is not a channel (i.e. it has a cast)} \qquad \mathtt{ch}(c_1) = \mathtt{ch}(c_2) \\ \overline{c_2}\langle c_1', \ldots, c_n'\rangle.Q \longrightarrow_{\mathsf{c}} Q' \\ c_1(a_1 : S_1, \ldots, a_n : S_n).P \mid Q' \longrightarrow_{\mathsf{c}} R\end{array}}{c_1(a_1 : S_1, \ldots, a_n : S_n).P \mid \overline{c_2}\langle c_1', \ldots, c_n'\rangle.Q \longrightarrow R}$$

Cast Reduction $\boxed{P \longrightarrow_{\mathsf{c}} P}$

(C-OUT-BASE)
$$\overline{a}\langle c_1, \ldots, c_n\rangle.P \longrightarrow_{\mathsf{c}} \overline{a}\langle c_1, \ldots, c_n\rangle.P$$

(C-OUT-SUCCEED)
$$\frac{\overline{c}\langle (c_1 : S_1 \Rightarrow T_1), \ldots, (c_n : S_n \Rightarrow T_n)\rangle.P \longrightarrow_{\mathsf{c}} R}{\overline{(c : \mathtt{o} \cdot (T_1 \cdot \ldots \cdot T_n) \Rightarrow \mathtt{o} \cdot (S_1 \cdot \ldots \cdot S_n))}\langle c_1, \ldots, c_n\rangle.P \longrightarrow_{\mathsf{c}} R}$$

(C-OUT-FAIL)
$$\overline{(c : \mathtt{i} \cdot (T_1 \cdot \ldots \cdot T_n) \Rightarrow \mathtt{o} \cdot (S_1 \cdot \ldots \cdot S_n))}\langle c_1, \ldots, c_n\rangle.P \longrightarrow_{\mathsf{c}} \textit{typeError}$$

(C-OUT-EXPAND)
$$\frac{\overline{(c : \mathtt{o} \cdot (\star \cdot \ldots \cdot \star) \Rightarrow \mathtt{o} \cdot (S_1 \cdot \ldots \cdot S_n))}\langle c_1, \ldots, c_n\rangle.P \longrightarrow_{\mathsf{c}} R}{\overline{(c : \star \Rightarrow \mathtt{o} \cdot (S_1 \cdot \ldots \cdot S_n))}\langle c_1, \ldots, c_n\rangle.P \longrightarrow_{\mathsf{c}} R}$$

(C-IN-BASE)
$$a(a_1 : S_1, \ldots, a_n : S_n).P \mid \overline{a}\langle c_1, \ldots, c_n\rangle.Q \longrightarrow_{\mathsf{c}} a(a_1 : S_1, \ldots, a_n : S_n).P \mid \overline{a}\langle c_1, \ldots, c_n\rangle.Q$$

(C-IN-SUCCEED)
$$\frac{c(a_1 : T_1, \ldots, a_n : T_n).P \mid \overline{a}\langle (c_1 : S_1 \Rightarrow T_1), \ldots, (c_n : S_n \Rightarrow T_n)\rangle.Q \longrightarrow_{\mathsf{c}} R}{(c : \mathtt{i} \cdot (T_1 \cdot \ldots \cdot T_n) \Rightarrow \mathtt{i} \cdot (S_1 \cdot \ldots \cdot S_n))(a_1 : S_1, \ldots, a_n : S_n).P \mid \overline{a}\langle c_1, \ldots, c_n\rangle.Q \longrightarrow_{\mathsf{c}} R}$$

(C-IN-FAIL)
$$(c : \mathtt{o} \cdot (T_1 \cdot \ldots \cdot T_n) \Rightarrow \mathtt{i} \cdot (S_1 \cdot \ldots \cdot S_n))(a_1 : S_1, \ldots, a_n : S_n).P \mid \overline{a}\langle c_1, \ldots, c_n\rangle.Q \longrightarrow_{\mathsf{c}} \textit{typeError}$$

(C-IN-EXPAND)
$$\frac{(c : \mathtt{i} \cdot (\star \cdot \ldots \cdot \star) \Rightarrow \mathtt{i} \cdot (S_1 \cdot \ldots \cdot S_n))(a_1 : S_1, \ldots, a_n : S_n).P \mid \overline{a}\langle c_1, \ldots, c_n\rangle.Q \longrightarrow_{\mathsf{c}} R}{(c : \star \Rightarrow \mathtt{i} \cdot (S_1 \cdot \ldots \cdot S_n))(a_1 : S_1, \ldots, a_n : S_n).P \mid \overline{a}\langle c_1, \ldots, c_n\rangle.Q \longrightarrow_{\mathsf{c}} R}$$

Figure 3: Operational Semantics for the π-calculus with Cast Channels. Notice the absence of a reduction rule for $P + P'$: this is because, in this preliminary work, we let it reduce to either P or P' as needed (cf. Remark 2.1).

(C-OUT-EXPAND), similarly, applies when the subject of an output is wrapped in a cast. Since channels may be simply declared to be of dynamic type $\star$, we may have casts from $\star$ to $\mathtt{o}\cdot(T_1,\ldots,T_n)$. In this case, we take a step to treat $\star$ as $\mathtt{o}\cdot(\star,\ldots,\star)$, so that (C-OUT-SUCCEED) can take over, succeed for now, but handle the rest of casts. (Casts from $\mathtt{o}\cdot(T_1,\ldots,T_n)$ to $\star$ do not happen in the context of this rule because the top level operation is an output and the cast insertion is guaranteed to produce a cast ending in $\mathtt{o}$.)

Rules (C-IN-*) play the same role as rules (C-OUT-*) but they work with inputs. A key difference is that when casts are resolved their nested casts are transferred to the arguments of the output partner.

4 Example

In this section, we show how our formulation applies to the example of the revenue agency of Section 2.

Gradual Type Checking The revenue agency A and the tax payer client C share the initial channel r. C is type checked under the assignment that r must be used input-only, and used to receive a channel whose capabilities are not known at compile-time, that is, it is of dynamic type. Therefore, we type check C with $\Gamma = r : \mathtt{i}\cdot\star$. Let us recall C. (T is the type of the channel representing \$100, irrelevant here).

$$r(b:\star).\ (\overline{b}\langle\$100\rangle\ +\ b(sum:T)) \tag{8}$$

The first rule that applies is (T-IN) for the top level process $r(b:\star).$*(the rest of the client)*. This rule applies because we are called to check $\mathtt{i}\cdot\star \sim \mathtt{i}\cdot\star$, which certainly holds because $\sim$ is reflexive. Afterwards, Γ is augmented with the information about b and becomes $\Gamma = r : \mathtt{i}\cdot\star, b:\star$. The first branch of the choice operator $\overline{b}\langle\$100\rangle$ type checks successfully because the type $\mathtt{o}\cdot T$ is consistent with the type of b in Γ. This check is $\star \sim \mathtt{o}\cdot T$, which holds because $\star$ is consistent with every type. For analogous reasons, the second branch $b(sum:T)$ type checks successfully because we ultimately need to check $\star \sim \mathtt{i}\cdot T$, which holds.

The agency is type checked under the assignment that r must be used output-only for sending a channel of dynamic type, that is, $\Gamma = r : \mathtt{o}\cdot\star$.

$$\begin{aligned} A \equiv\ & \nu(x:\mathtt{o}\cdot T).\ \overline{r}^{\mathscr{A}}\langle x\rangle.\overline{x}\langle\$100\rangle \\ & + \\ & \nu(x:\mathtt{i}\cdot T).\ \overline{r}^{\mathscr{A}}\langle x\rangle.x(sum) \end{aligned} \tag{9}$$

Let us consider the first branch of the choice operator. (T-RES) introduces x, making $\Gamma = r : \mathtt{o}\cdot\star, x : \mathtt{o}\cdot T$. To type check the reverse output, we need to consider its type $\mathtt{o}\cdot(\mathtt{o}\cdot T)$, where the highlighted part is grabbed from the information of x in Γ. This type must be compared with the type of r which is $r : \mathtt{o}\cdot\star$. The check is $\mathtt{o}\cdot\star \sim \mathtt{o}\cdot(\mathtt{o}\cdot T)$, which holds. Finally, $\overline{x}\langle\$100\rangle$ is type checked successfully because we end up checking $\mathtt{o}\cdot T \sim \mathtt{o}\cdot T$, which holds by reflexivity. The second branch of the choice operator follows similar lines.

Cast Insertion We start with the client. For the top level process $r(b:\star).$*(the rest of the client)*, the relation $\mathtt{i}\cdot\star \sim \mathtt{i}\cdot\star$ induces the cast $(r : \mathtt{i}\cdot\star \Rightarrow \mathtt{i}\cdot\star)(b:\star).$*(the rest of the client)*. However, this is a trivial cast and we omit it, as implementations in fact do skip trivial casts. The first branch of the choice operator $\overline{b}\langle\$100\rangle$ prescribes checking $\star \sim \mathtt{o}\cdot T$, which induces the cast $\overline{(b:\star \Rightarrow \mathtt{o}\cdot T)}\langle\$100\rangle$. The second

branch of the choice operator prescribes checking $\star \sim \mathtt{i} \cdot T$, which induces the cast $(b : \star \Rightarrow \mathtt{i} \cdot T)(sum))$. The result of the compilation for the client is:

$$C' \equiv r(b : \star).\ (\overline{(b : \star \Rightarrow \mathtt{o} \cdot T)}\langle \$100 \rangle \ +\ (b : \star \Rightarrow \mathtt{i} \cdot T)(sum))$$

Let us consider the compilation for the first branch of the choice operator of the agency. Going through a restriction ν does not generate new casts. Afterwards, the use of the reverse output does generate casts. The checking that takes place is $\mathtt{o} \cdot \star \sim \mathtt{o} \cdot (\mathtt{o} \cdot T)$. Therefore, we apply the cast insertion rule for reverse output and we have $\overline{(r : \mathtt{o} \cdot \star \Rightarrow \mathtt{o} \cdot (\mathtt{i} \cdot T))}\langle x \rangle$. Notice the highlighted input capability. This type comes from $\Gamma(a)^{-1}$ when applied to $(\mathtt{o} \cdot T)$, which is $(\mathtt{i} \cdot T)$. Notice also that the reverse output became an ordinary output. The compilation for the second branch of the choice operator is similar. Ultimately, the compilation of the agency process is the following.

$$\begin{aligned} A' \equiv\ & \nu(x : \mathtt{o} \cdot T).(\overline{r : \mathtt{o} \cdot \star \Rightarrow \mathtt{o} \cdot (\mathtt{i} \cdot T)})\langle x \rangle.\overline{x}\langle \$100 \rangle \\ & + \\ & \nu(x : \mathtt{i} \cdot T).(\overline{r : \mathtt{o} \cdot \star \Rightarrow \mathtt{o} \cdot (\mathtt{o} \cdot T)})\langle x \rangle.x(sum) \end{aligned} \tag{10}$$

These casts are slightly different from those of the example in Section 2. One reduction step will distribute these casts to the arguments, reaching that form. We shall see this aspect in the next paragraph.

Execution Let us consider the execution of the compiled agency and the compiled client in parallel, i.e., $(A' \mid C')$. Let us suppose that we are in the scenario in which the tax payer must send a payment[2]. Structural congruence brings together the two processes so that ultimately the rule (COMM) would apply:

$$r(b : \star).\ (\overline{(b : \star \Rightarrow \mathtt{o} \cdot T)}\langle \$100 \rangle \ +\ (b : \star \Rightarrow \mathtt{i} \cdot T)(sum)) \mid (\overline{r : \mathtt{o} \cdot \star \Rightarrow \mathtt{o} \cdot (\mathtt{o} \cdot T)})\langle x \rangle.x(sum)$$

However, (COMM) cannot apply just yet because the rule works only when the subject of inputs and outputs are bare channels. Instead, rule (C-SOLVE) applies. This rule has the effect of applying (C-OUT-SUCCEED) to the output process. This step detects that the top level cast is an o-to-o match, and so it removes the cast and distributes the nested casts to the argument:

$$(\overline{r : \mathtt{o} \cdot \star \Rightarrow \mathtt{o} \cdot (\mathtt{i} \cdot T)})\langle x \rangle.\overline{x}\langle \$100 \rangle \longrightarrow_{\mathsf{c}} \overline{r}\langle (x : \mathtt{o} \cdot T \Rightarrow \star) \rangle.x(sum)$$

Therefore, in one step we have a process that is in a form that (COMM) handles:

$$r(b : \star).\ (\overline{(b : \star \Rightarrow \mathtt{o} \cdot T)}\langle \$100 \rangle \ +\ (b : \star \Rightarrow \mathtt{i} \cdot T)(sum)) \mid \overline{r}\langle (x : \mathtt{o} \cdot T \Rightarrow \star) \rangle.x(sum)$$

And in one step we obtain:

$$\overline{(x : \mathtt{o} \cdot T \Rightarrow \star \Rightarrow \mathtt{o} \cdot T)}\langle \$100 \rangle \ +\ (x : \mathtt{i} \cdot T \Rightarrow \star \Rightarrow \mathtt{i} \cdot T)(sum) \mid x(sum)$$

At this point, structural congruence brings the output on x and its input $x(sum)$ together. Since the output on x has casts, we have that (C-SOLVE) applies. This rule has the effect to first apply (C-OUT-EXPAND) because the top level cast is from $\star$ to $\mathtt{o} \cdot T$. This application of (C-OUT-EXPAND) expands the inner cast and turns the output into $\overline{(x : \mathtt{o} \cdot T \Rightarrow \mathtt{o} \cdot \star \Rightarrow \mathtt{o} \cdot T)}\langle \$100 \rangle$. At this point, two applications of (C-OUT-SUCCEED) push these casts to the argument. This is possible because all casts are an o-to-o match,

[2]Recall that, for simplicity, we assume that the $+$ operator will select the right branch for us, cf. Remark 2.1.

and therefore the rule applies. Ultimately, we end up with the following process (isolated by structural congruence for execution).

$$\overline{x}\langle \$100 : T \Rightarrow \star \Rightarrow T \rangle \mid x(sum)$$

Here, (COMM) applies as usual. Notice that $\$100 : T \Rightarrow \star \Rightarrow T$ is not resolved immediately. It will be checked when $\$100 : T \Rightarrow \star \Rightarrow T$ is going to be used, *and if* it is going to be used. This is in line with the dynamic typing style. Such event does not occur in our example, but we could imagine that happening if $x(sum)$ had a continuation process.

5 Properties (Not Addressed in This Preliminary Work)

In this paper, we do not claim any theoretical results about our formalisms, hence the "*Towards*" part of the title. In future we would like to prove some key properties of our formulations. The properties that we plan on attacking can be divided into two kinds. The first kind concerns the typical properties at play for typed calculi: the *progress theorem* and the *type preservation theorem*. The second kind of properties are specific to the domain of gradual typing. In this section we discuss some of these properties.

Progress For the progress theorem, we want to prove that if a process P is well-typed then either $P = typeError$, or P is stuck in a situation in which no communication can take place (as in $a(x).P \mid b(x).P$), or $P \longrightarrow P'$, for some P'. To prove this property we need to check that all behavior is covered. We make an example of the reasoning that applies in our context.

The only additions to ordinary π-calculus is that we can have casts in 3 places:

- a cast for a channel being sent, as in $\overline{a}\langle c : \star \Rightarrow \mathtt{o} \rangle.P$. If this output does not have an input partner it would be correctly stuck in our calculus. If it does have a communicating partner then the ordinary communication rule of the π-calculus takes place with no restriction, therefore it progresses.
- a cast for a channel that is subject of an output, as in $\overline{a : list\ of\ casts}\langle c \rangle.P$. In this case, rules (C-OUT-SUCCEEDS), (C-OUT-FAIL), and (C-OUT-EXPAND) of Fig. 4 apply in all possible cases, depending on the casts. Indeed, a cast can only be o-to-o, handled by (C-OUT-SUCCEEDS), i-to-o, handled by (C-OUT-FAIL), or $\star$-to-o, handled by (C-OUT-EXPAND), with the latter leading directly to a form handled by (C-OUT-SUCCEEDS). Casts to i or $\star$ cannot happen because the cast insertion always places a cast to o in the context of an output.
- a cast for a channel that is subject of an input, as in $(a : list\ of\ casts)(b : T).P$. Here casts are resolved in essentially the same way as in the previous case, though dually.

Type Preservation We would like to prove that if a process P is well-typed and $P \longrightarrow P'$ then P' is well-typed, as well. This would require a case analysis on the steps that P can take. Especially, the new reduction rules that handle casts will be the relevant part of this proof. We make an example of the reasoning that applies in our context, just for one reduction rule: (C-OUT-SUCCEED). Let us consider the step that (C-OUT-SUCCEED) provides. Suppose a is an output channel that sends one channel of dynamic type, and b is an input channel. Therefore, we have $\Gamma = a : \mathtt{o} \cdot \star, b : \mathtt{i}$. Suppose we have the process $(\overline{a : \mathtt{o} \cdot \star \Rightarrow \mathtt{o} \cdot \mathtt{i}})\langle b \rangle$. This is a well-typed process because the cast channel used in output is ultimately of type $\mathtt{o} \cdot \mathtt{i}$ (the target of the cast), and b is indeed of type i. Therefore, types all match. After the step of (C-OUT-SUCCEED) we have the cast pushed into the argument: $\overline{a}\langle b : \mathtt{i} \Rightarrow \star \rangle$. We have to

check that this process is well-typed. It is, indeed: a is a channel of type $\mathtt{o} \cdot \star$ and it is used in output. We have to check that the sent channel is of type $\star$. This is the case because the sent argument is $b : \mathtt{i} \Rightarrow \star$, which ultimately is of type $\star$ because of the cast to $\star$.

Gradual Typing Properties In future, we would like to address the correctness criteria of gradual typing as summarized and delineated in [21]. In this paragraph, we discuss only some relevant ones.

A property that we want to prove is that our type system is a conservative extension of that of Pierce and Sangiorgi (when `either` is not used). We expect this property to hold because the type consistency $\sim$ is designed to cover the same cases of Pierce's and Sangiorgi's type system, and *additionally* some more. We have to be careful, however, that the additional processes that we can type check are not incorrectly deemed well-typed. It would be incorrect for example to type check $a(b : \mathtt{o}).b(z)$. This would not happen in our calculus despite $\sim$ being a liberal relation because it still mandates $\mathtt{i} \not\sim \mathtt{o}$, and thus would reject that process.

Similarly, also the operational semantics should be a conservative extension of that of the π-calculus. That is, processes that do no contain casts at all should be executed as they were in the ordinary π-calculus. We expect this property to hold because when casts are not around then the plain reduction rules of the π-calculus, and solely those, apply.

Another property that we would like to prove is that our gradualized type system is monotonic w.r.t. the amount of type annotations that are turned into the dynamic type. This is a fundamental property in gradual typing: To make an example, if we were in the λ-calculus, this property would entail that we can write a generalized identity function $(\lambda x : \star.x)$ and it *must* be well-typed (as we expect), because $(\lambda x : \mathtt{Int}.x)$ is well-typed, and the former simply turns a type into the dynamic type. We expect this property to hold in our type system because the type consistency $\sim$ is indeed designed to allow type checking to succeed when the dynamic type is met.

The converse monotonic theorem, in which we turn dynamic types into specific types and expect to preserve typeability, does not, and should not, hold: it is not guaranteed that if a programmer inserts new type annotations then the process is well-typed. Indeed, the programmer might insert a wrong type.

We would like to address the blame theorem [1, 28, 24]. This property states that run-time cast errors can happen only in the regions of code that are dynamically typed. To address this property we will need to modify our calculus to keep, at run-time, the information of a label on casts. This label pinpoints the origin of the cast in the source code. Furthermore, our operational semantics must be equipped with a suitable mechanism for blame tracking.

We also would like to address the gradual guarantee. This property states that adding types to a gradually typed process has the effect that the new version of the process can only i) end up ill-typed at compile-time, ii) behave the same way as the previous version, or iii) end up in a cast error at run-time (if the new type inserted was wrong but this could not be detected at compile-time). This is an important property that provides programmers with the guarantee that adding types would not corrupt unpredictably the execution of processes. As a consequence, gradual typing could be used to evolve a process into a more and more statically typed version over time without disrupting the workflow of the programmer. We expect this property to be challenging, as it has been proven challenging in several contexts [21, 11, 6].

6 Related Work and (More) Future Work

The work that is most related to that of this paper is the typed π-calculus of Pierce and Sangiorgi [16]. We have delineated the major differences with our calculus in Section 2 and Section 3. An additional difference is that the typed π-calculus makes use of recursive types to type check processes such as $\overline{a}\langle a\rangle$. We have omitted recursive types but our type system can still type check those cases by giving the channel a the type $\star$. However, adding recursive types is part of our future work so that we would be able to describe more precisely cases of the like. Recursive types have been addressed previously in gradual typing implementations [9, 25, 12, 13], and in semantics formulations [20, 15]. We plan on building on this body of work.

We have removed the type `either` altogether, which means that our channels can be used input-only or output-only. The feature that `either` offers cannot be recovered completely with $\star$ because once the input or output capability of a channel has been established at run-time it cannot be used for the opposite operation. We therefore plan to extend our formulations to have both `either` and $\star$.

There has been considerable work in gradual typing both from industry [3, 9, 27, 4] and academia [18, 24, 26, 14, 17, 1]. The challenge in capturing a calculus such as the π-calculus is that several output processes may compete for a single input. We, however, must stick to one such communicating process after initiating a communication, or otherwise we would transfer some casts to a process and some others to another. Because of this, we have used big-step semantics to commit to a communicating process throughout solving all casts. To our knowledge, this is an aspect that has not been previously addressed in gradual typing.

Gradual typing has been applied to process calculi previously, and in particular to gradual session types [10, 23]. The major differences between gradual session types and the work of this paper are:

- Types in [10, 23] are set to enforce session fidelity: every send is matched with a receive, every select is matched with an offer, and so on, and deadlock freedom is ensured. The type system we focus on is derived from that of Pierce and Sangiorgi, which does not strictly structure the communication taking place. It simply keeps track of the capabilities of channels. Therefore, our type system is more permissive, and describes more computations, but is also less safe. In our calculus we can describe a race condition with one input and several output partners, for example. Similarly, we can type check a single process in input hanging on forever without outputs. These scenarios are rejected in gradual session types but are common in Pierce's and Sangiorgi's calculus because, as exemplified in Section 2, the type system is frequently used to type check only a part of the whole process.

- Both [10] and [23] only focus on dyadic session types, that is, 2 dually communicating processes. Our calculus allows for an unrestricted number of processes in parallel. This raised the challenge described above on committing to a communicating process, which we have solved with a big-step semantics.

There has been some work in automating the shift to gradual typing [5, 6, 7]. We have found out that because of the particular challenge with committing to a communicating process these frameworks could not be applied. That is to say that our formulations could not be generated automatically with current automated techniques.

7 Conclusions

We have presented our preliminary work towards a calculus of gradual capabilities for the π-calculus. We have formulated a gradual type system, a cast insertion procedure, and a π-calculus that can handle casts on channels. We do not claim any theoretical results about our formalisms at this stage. We have shown how our formalisms apply to an example that requires the discovery of channel capabilities at run-time. We also have discussed our future plans, especially w.r.t. the key properties that a gradually typed calculus such as ours should afford.

References

[1] Ahmed, A., Findler, R.B., Siek, J.G., Wadler, P.: Blame for all. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 201–214 (2011). https://doi.org/10.1145/1926385.1926409, https://doi.org/10.1145/1926385.1926409

[2] Anderson, C., Drossopoulou, S.: Babyj: from object based to class based programming via types. Electr. Notes Theor. Comput. Sci. **82**(7), 53–81 (2003). https://doi.org/10.1016/S1571-0661(04)80802-8, https://doi.org/10.1016/S1571-0661(04)80802-8

[3] Bierman, G.M., Abadi, M., Torgersen, M.: Understanding typescript. In: ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings. pp. 257–281 (2014). https://doi.org/10.1007/978-3-662-44202-9_11, https://doi.org/10.1007/978-3-662-44202-9_11

[4] Chaudhuri, A.: Flow: a static type checker for javascript, http://flowtype.org

[5] Cimini, M., Siek, J.G.: The gradualizer: A methodology and algorithm for generating gradual type systems. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 443–455. POPL '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837632, http://doi.acm.org/10.1145/2837614.2837632

[6] Cimini, M., Siek, J.G.: Automatically generating the dynamic semantics of gradually typed languages. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 789–803. POPL 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009863, http://doi.acm.org/10.1145/3009837.3009863

[7] Garcia, R., Clark, A.M., Tanter, E.: Abstracting gradual typing. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 429–442. POPL '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837670, http://doi.acm.org/10.1145/2837614.2837670

[8] Harper, R.: Practical Foundations for Programming Languages. Cambridge University Press, New York, NY, USA (2012)

[9] Hejlsberg, A.: Introducing TypeScript. Microsoft Channel 9 Blog (2012)

[10] Igarashi, A., Thiemann, P., Vasconcelos, V.T., Wadler, P.: Gradual session types. Proc. ACM Program. Lang. **1**(ICFP), 38:1–38:28 (Aug 2017). https://doi.org/10.1145/3110282, http://doi.acm.org/10.1145/3110282

[11] Igarashi, Y., Sekiyama, T., Igarashi, A.: On polymorphic gradual typing. PACMPL **1**(ICFP), 40:1–40:29 (2017). https://doi.org/10.1145/3110284, https://doi.org/10.1145/3110284

[12] Kuhlenschmidt, A.: The grift compiler (2019), https://github.com/Gradual-Typing/Grift

[13] Kuhlenschmidt, A., Almahallawi, D., Siek, J.G.: Toward efficient gradual typing for structural types via coercions. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language

Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 517–532 (2019). https://doi.org/10.1145/3314221.3314627, https://doi.org/10.1145/3314221.3314627

[14] Lehmann, N., Tanter, É.: Gradual refinement types. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 775–788 (2017). https://doi.org/10.1145/3009837, http://dl.acm.org/citation.cfm?id=3009856

[15] New, M.S., Licata, D.R., Ahmed, A.: Gradual type theory. PACMPL **3**(POPL), 15:1–15:31 (2019). https://doi.org/10.1145/3290328, https://dl.acm.org/citation.cfm?id=3290328

[16] Pierce, B.C., Sangiorgi, D.: Typing and subtyping for mobile processes. Mathematical Structures in Computer Science **6**(5), 409–453 (1996). https://doi.org/10.1017/S096012950007002X

[17] Schwerter, F.B., Garcia, R., Tanter, É.: Gradual type-and-effect systems. J. Funct. Program. **26**, e19 (2016). https://doi.org/10.1017/S0956796816000162, https://doi.org/10.1017/S0956796816000162

[18] Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop. pp. 81–92 (September 2006)

[19] Siek, J.G., Thiemann, P., Wadler, P.: Blame and coercion: together again for the first time. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015. pp. 425–435 (2015). https://doi.org/10.1145/2737924.2737968, https://doi.org/10.1145/2737924.2737968

[20] Siek, J.G., Tobin-Hochstadt, S.: The recursive union of some gradual types. In: A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday. pp. 388–410 (2016). https://doi.org/10.1007/978-3-319-30936-1_21, https://doi.org/10.1007/978-3-319-30936-1_21

[21] Siek, J.G., Vitousek, M.M., Cimini, M., Boyland, J.T.: Refined criteria for gradual typing. In: 1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA. pp. 274–293 (2015). https://doi.org/10.4230/LIPIcs.SNAPL.2015.274, https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

[22] Siek, J.G., Wadler, P.: Threesomes, with and without blame. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010. pp. 365–376 (2010). https://doi.org/10.1145/1706299.1706342, https://doi.org/10.1145/1706299.1706342

[23] Thiemann, P.: Session types with gradual typing. In: Trustworthy Global Computing - 9th International Symposium, TGC 2014, Rome, Italy, September 5-6, 2014. Revised Selected Papers. pp. 144–158 (2014). https://doi.org/10.1007/978-3-662-45917-1_10, https://doi.org/10.1007/978-3-662-45917-1_10

[24] Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: From scripts to programs. In: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications. pp. 964–974. OOPSLA '06, ACM, New York, NY, USA (2006). https://doi.org/10.1145/1176617.1176755, http://doi.acm.org/10.1145/1176617.1176755

[25] Tobin-Hochstadt, S., St-Amour, V., Dobson, E., Takikawa, A.: The typed racket guide (2019)

[26] Toro, M., Garcia, R., Tanter, É.: Type-driven gradual security with references. ACM Trans. Program. Lang. Syst. **40**(4), 16:1–16:55 (2018). https://doi.org/10.1145/3229061, https://dl.acm.org/citation.cfm?id=3229061

[27] Verlaguet, J.: Facebook: Analyzing PHP statically. In: Commercial Users of Functional Programming (CUFP) (2013), http://cufp.org/2013/julien-verlaguet-facebook-analyzing-php-statically.html

[28] Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings. pp. 1–16 (2009). https://doi.org/10.1007/978-3-642-00590-9_1, https://doi.org/10.1007/978-3-642-00590-9_1