Xjenza Online - Journal of The Malta Chamber of Scientists
www.xjenza.org
DOI: 10.7423/XJENZA.2014.2.03

Research Article

Uniqueness Typing For A Higher-Order Language

Adrian Francalanza and Melanie Zammit
CS, ICT, University of Malta

Abstract. We investigate type-based analysis for a higher-order channel passing language with strong update, whereby messages of a different kind are communicated over the same channel. In order to reason about such programs, our type system employs the concept of *uniqueness* to be able to assert when it is safe to change the object type a channel. We design a type system based on this concept and prove that our type system is *sound*, meaning that it only accepts programs that do not produce runtime errors.

1 Introduction

Resource usage in settings with limited amounts of resources is an important aspect of computation and one common way how to manage limited resources is through *resource reuse*. In this paper we investigate type-based analysis for resource reuse in concurrent settings; in such settings, the reuse of resources is easy to get wrong because one thread of computation may change the mode of usage of a particular resource while other threads still employ the previous usage mode. In particular, we focus on message-passing programs, such as those written in Go ("The Go Programming Language", n.d.) and Erlang (Armstrong, 2007; Cesarini and Thompson, 2009), where the resources reused are the channels used to transmit the messages on. We carry out our analysis using the pi-calculus(Sangiorgi and Walker, 2003), a standard model for channel-passing computation and, in particular, extend the results obtained in (De Vries et al., 2012) to a higher-order version of the calculus where the values communicated include also the programs themselves.

Consider, as an example, the client-server protocol depicted in Fig. 1. The client sends a program to be executed by the server whereby, for the program to run, it needs to be instantiated with a particular dataset. From the server side, it needs to perform the necessary checks on the code before it runs it whereas the client would ideally not communicate the dataset unless it is certain that the code will be executed (the dataset may be bulky or contain sensitive information).

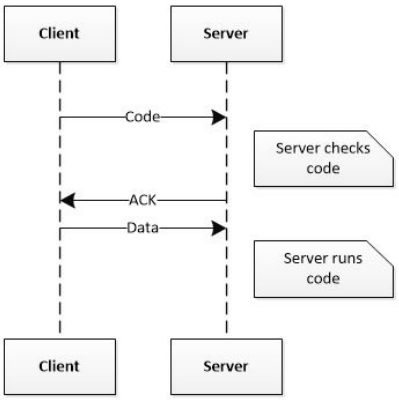

Figure 1: Communication links between a client and a server

Fig. 1 thus describes a two-phase protocol, whereby the client sends the code to the server in the first phase. The server checks the code and acknowledges back if the check is successful and it is prepared to run the code, at which point the client send the data to the server in the second phase of the protocol. Once the server receives the data, it executes the validated code with this data.

A program implementing the protocol of Fig 1 can be expressed as the higher-order pi-calculus parallel composition of a client with a server

$$client \mid server \tag{1}$$

defined as follows:

$$server \stackrel{\text{def}}{=} port_1?(x_{code}, x_{inst}, x_{ack}).\ \texttt{if}\ check(x_{code})\ \texttt{then}\ \texttt{run}\ x_{code}.\ (\texttt{new}\ port_2)(x_{ack}!port_2.port_2?(x_{data}).x_{inst}!x_{data})\ \texttt{else}\ \texttt{nil}$$

$$client \stackrel{\text{def}}{=} (\texttt{new}\ ack, inst)\ \left(port_1!(code, inst, ack) \mid ack?(x_{port}).x_{port}!data\right)$$

$$code \stackrel{\text{def}}{=} \langle inst?(y).P(y)\rangle$$

In (1), the *server* and *client* originally interact only on the port $port_1$. The server inputs three values on this channel,

Correspondence to: Adrian Francalanza (adrian.francalanza@um.edu.mt)

namely the code to check and run, x_{code}, an acknowledgment channel, x_{inst}, and a channel on which to instantiate the code with, x_{inst}. If the check succeeds, it runs the code and acknowledges back to the *client*, sending it a new channel, $port_2$, on which to initiate the second phase of the protocol. Once it receives the data on $port_2$, it forwards it to the executing code through channel x_{inst}. Dually, the *client* first sends the *code* along with two new channels, *ack* and *inst*, on $port_1$ and as soon as it receives an acknowledgment with the channel for the second phase of the protocol, it sends the *data* on this channel. The code consists of a thunked process, $\langle - \rangle$, that inputs its data on channel *inst* once it is executed, and then continues as P with the data inputted, y.

An alternative server that economises on channels is the one defined below, whereby it *reuses* channel $port_1$ for the second phase of the protocol instead of creating a new channel $port_2$. Note that when $port_1$ is reused, *different* forms of values are communicated on it, namely *data* as opposed to $(code, inst, ack)$. In the literature, this is often referred to as *strong update*(Ahmed et al., 2005).

$$\begin{aligned} srvOpt \stackrel{\text{def}}{=}\ & port_1?(x_{code}, x_{inst}, x_{ack}).\\ & \texttt{if } check(x_{code}) \texttt{ then}\\ & \quad \texttt{run } x_{code}.x_{ack}!port_1.port_1?(x_{data}).x_{inst}!x_{data}\\ & \quad \texttt{else nil} \end{aligned}$$

When we execute

$$client \mid srvOpt \tag{2}$$

it turns out that we obtain the same behaviour as that of (1). However channel reuse is not always safe and can lead to erroneous executions. For instance, executing

$$client \mid srvOpt \mid client$$

does not have the same behaviour as that of its original counterpart $client \mid server \mid client$; in fact, the former results in a runtime error whereby the second phase of the protocol between one *client* and *srvOpt* is interfered with by a message on $port_1$ from the other *client* with *mismatching communicated values* (this is a direct consequence of the strong update on $port_1$). Alternatively, in the thunked process in *code* is instantiated through $port_1$

$$codeErr \stackrel{\text{def}}{=} \langle port_1?(y).P(y) \rangle$$

the channel reuse in *srvOpt* would also be unsafe and generate a runtime error.

In this paper we develop a type system that statically analyses programs and guarantees that type-checked programs do not generate runtime errors. Our type system accepts correct higher-order programs such as (1), but also programs with safe strong updates such as (2), while rejecting all unsafe programs. § 2 describes our language whereas § 3 presents the uniqueness type system. § 4 presents the main soundness results and § 5 concludes.

2 The Language

Fig. 2 presents the higher-order pi-calculus that will be used. The syntax assumes separate denumerable sets for channel names, $c, d \in \textsc{Chan}$, and variables, $x, y \in \textsc{Var}$; identifiers $i, j \in \textsc{Id} = \textsc{Chan} \cup \textsc{Var}$ range over both channels and variables. The main syntactic class is that of processes $P, Q, R \in \textsc{Proc}$ using values, $u, v \in \textsc{Val}$, consisting of identifiers and thunked processes, $\langle P \rangle$. We sometimes use the shorthand notation $c!v$ to denote $c!v.\texttt{nil}$.

The reduction relation ($\longrightarrow$) is defined as the least relation over closed processes satisfying the rules in Figure 2; it assumes a standard structural equivalence relation (Hennessy, 2007) over processes, $\equiv$, that allows us to abstract over aspects such as the commutativity of parallel composition, $P \mid Q \equiv Q \mid P$ (amongst others). The main rules are rCom, describing communication amongst processes, and rRun, describing the spawning of thunked processes using the command `run`. The remaining rules are standard. In what follows, we use $P \longrightarrow^* Q$ to denote the transitive closure of our reduction relation.

The reduction relation allows us to deduce that

$$client \mid server \longrightarrow^* P(data)$$

and also that

$$client \mid srvOpt \longrightarrow^* P(data)$$

It also allows us to show that, whereas

$$client \mid server \mid client \longrightarrow^* P(data) \mid client$$

is the only possible evaluation, we have the following sequence of reductions for the analogous system that uses the alternative *srvOpt* instead.

$$client \mid srvOpt \mid client \longrightarrow^* \begin{cases} port_1!data \mid \\ (\texttt{new } inst)\big(inst?(y).P(y) \mid port_1?(x_{data}).inst!x_{data}\big) \\ \mid client \end{cases}$$

At this point, the input on channel $port_1$ can react with either the output $port_1!data$ (carrying the right format of values) or with the output $port_1!(code, inst, ack)$ from the second client, which carries values of a different (erroneous) format.

We define a predicate on processes to describe the runtime errors we want to rule out; the rules defining this predicate $P \rightarrow_{\text{ERR}}$ are given in Fig. 2. For example, eOut and eIn state that thunked process values $\langle P \rangle$ cannot be used instead of channels to communicate on, whereas eRun states that we cannot use channel values instead of thunked process values when spawning new executions. eIf1 and eIf2 state that comparisons are only allowed on channel values. The remaining rules are the usual contextual rules.

3 Type System

Although there exist numerous type systems for the various variants pi-calculus, they tend to rule out well-behaved

Syntax:

$$\begin{array}{rcl} u, v \in \textsc{Val} & ::= & i \mid \langle P \rangle \\ P, Q, R \in \textsc{Proc} & ::= & \texttt{nil} \mid P \mid Q \mid v!u.P \mid v?x.P \mid \texttt{if}\ (v{=}u)\ \texttt{then}\ P\ \texttt{else}\ Q \mid \texttt{run}\, v.P \mid (\texttt{new}\, c)P \\ & \mid & \texttt{rec}\, x.P \mid x \end{array}$$

Semantics:

$$\frac{}{c!v.P \mid c?x.Q \longrightarrow P \mid (Q[v/x])}\ \textsc{rCom} \qquad \frac{}{\texttt{run}\,\langle P\rangle.Q \longrightarrow P \mid Q}\ \textsc{rRun} \qquad \frac{}{\texttt{rec}\, x.P \longrightarrow P\{\!|\texttt{rec}\, x.P/x|\!\}}\ \textsc{rRec}$$

$$\frac{}{\texttt{if}\ (c{=}c)\ \texttt{then}\ P\ \texttt{else}\ Q \longrightarrow P}\ \textsc{rThen} \qquad \frac{}{\texttt{if}\ (c{=}b)\ \texttt{then}\ P\ \texttt{else}\ Q \longrightarrow Q}\ \textsc{rElse}$$

$$\frac{P \equiv P' \qquad P' \longrightarrow Q' \qquad Q' \equiv Q}{P \longrightarrow Q}\ \textsc{rStr} \qquad \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}\ \textsc{rPar} \qquad \frac{P \longrightarrow P'}{(\texttt{new}\, c)P \longrightarrow (\texttt{new}\, c)P'}\ \textsc{rRes}$$

Errors:

$$\frac{}{\texttt{if}\ (\langle Q\rangle{=}v)\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \rightarrow_{\textsc{err}}}\ \textsc{eIf1} \qquad \frac{}{\langle P\rangle!v.Q \rightarrow_{\textsc{err}}}\ \textsc{eOut} \qquad \frac{}{\texttt{run}\, c.P \rightarrow_{\textsc{err}}}\ \textsc{eRun} \qquad \frac{P \rightarrow_{\textsc{err}}}{(\texttt{new}\, c)P \rightarrow_{\textsc{err}}}\ \textsc{eRes}$$

$$\frac{}{\texttt{if}\ (v{=}\langle Q\rangle)\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \rightarrow_{\textsc{err}}}\ \textsc{eIf2} \qquad \frac{}{\langle P\rangle?x.Q \rightarrow_{\textsc{err}}}\ \textsc{eIn} \qquad \frac{P \rightarrow_{\textsc{err}}}{P \mid Q \rightarrow_{\textsc{err}}}\ \textsc{ePar} \qquad \frac{P \equiv Q \qquad Q \rightarrow_{\textsc{err}}}{P \rightarrow_{\textsc{err}}}\ \textsc{eStr}$$

Figure 2: Higher-Order Pi-Calculus

processes, such as the client-server example with strong updates of (2). Substructural type systems have been proven useful when reasoning about resource management (Pierce, 2004). In particular, De Vries et al. (2012) have defined a sound substructural type system that can reason about channel reuse for the *basic* pi-calculus, whereby code *cannot* be communicated at values. In this work we extend this type system so as to be able to handle resource reuse in higher-order pi-calculus programs.

3.1 Types and Type Environments

Uniqueness is a concept of exclusive access (De Vries et al., 2012), meaning that if a channel has a unique type, then it is guaranteed that only the process typed with that channel type has access to use that channel. Thus, strong update is safe for unique channels. However, even though a channel may not be unique at this instant, it may become unique after n computations. We describe these channels by the channel type attribute, *unique-after-i*. Uniqueness can also be represented as type annotations and reasoned about statically. More precisely, following (De Vries et al., 2012), a channel type has the syntactic form $[C]^a$ and describes two things: C is the *object type* and denotes the values that are allowed to be communicated over the channel; a denotes a type attribute (defined in Fig. 3). Thus, we say that

- An *Affine* channel ($[C]^{\mathbf{1}}$) is a channel with an *access restriction* that limits the channel to only *one* use at most.
- A *Unique-after-n* channel ($[C]^{(\bullet,n)}$) is a channel with an *access guarantee* that the channel will become unique after it is used for n times.
- An *Unrestricted* channel ($[C]^{\omega}$) is a channel with no usage restrictions or guarantees.

We note that unique channel types (De Vries et al., 2012) are in a sense, dual to affine channels (Kobayashi et al., 1999): whereas one yield a guarantee, the other imposes a restriction on usage. However, in order for the guarantees to be sound, we shall need to impose global restrictions on our type analysis (*c.f.* Def. 1) As an example of this, consider two parallel processes, A and B, that are using a channel c exclusively (*i.e.*, channel c is *unique* for process $A \mid B$):

- We would typically type-check one process, say A, with respect to a number, say m, of affine channel types for channel c. This restricts A to use c m times *at most*.
- Dually we would type-check the other process, B , with respect to a unique-after-n channel type for c. At each use, the count is reduced by 1, because this would mean that A used up one of its restrictions. Once 0 is reached, this would mean that A has used up all of its usage permissions, and since access was restricted to the process $A \mid B$, this would mean that B has exclusive access.
- Crucially, however, for the uniqueness guarantee to be sound, we need to require that $m \leq n$; otherwise we would reach 0 while A would still have further usage permissions.

Equipped with this analogy, we present the full type structure and operations for environment manipulation in

Type Structure:

$$\begin{array}{rcl} a & ::= & \mathbf{1} \mid \omega \mid (\bullet, n) \\ A, B \in \textsc{bTyp} & ::= & \texttt{int} \mid \texttt{bool} \\ C, D \in \textsc{cTyp} & ::= & B \mid [C]^a \mid \langle L \rangle \\ L, K \in \textsc{pTyp} & ::= & \epsilon \mid i = C, L \\ S, T \in \textsc{Typ} & ::= & C \mid \{L\} \end{array}$$

Type Splitting:

$$\frac{}{B = B \circ B}\ \textsc{pBase} \qquad \frac{}{[C]^\omega = [C]^\omega \circ [C]^\omega}\ \textsc{pUnr} \qquad \frac{}{[C]^{(\bullet,n)} = [C]^{\mathbf{1}} \circ [C]^{(\bullet,n+1)}}\ \textsc{pUnq}$$

Subtyping:

$$\frac{}{(\bullet, n) <: (\bullet, n+1)}\ \textsc{sIndx} \qquad \frac{}{(\bullet, i) <: \omega}\ \textsc{sUnq} \qquad \frac{}{\omega <: \mathbf{1}}\ \textsc{sAff} \qquad \frac{a_1 <: a_2}{[C]^{a_1} <: [C]^{a_2}}\ \textsc{sTyp}$$

Structural Manipulations for Environments:

$$\frac{}{\Gamma, v : T \ \prec\ \Gamma}\ \textsc{tWeak} \qquad \frac{T_1 <: T_2}{\Gamma, v : T_1 \ \prec\ \Gamma, v : T_2}\ \textsc{tSub} \qquad \frac{}{\Gamma, c : [C_1]^\bullet \ \prec\ \Gamma, c : [C_2]^\bullet}\ \textsc{tRev}$$

$$\frac{T = T_1 \circ T_2}{\Gamma, v : T \ \prec\ \Gamma, v : T_1, v : T_2}\ \textsc{tCon} \qquad \frac{T = T_1 \circ T_2}{\Gamma, v : T_1, v : T_2 \ \prec\ \Gamma, v : T}\ \textsc{tJoin}$$

Figure 3: Type Structure and Operations

our type system, given in Fig.3. We have three classes of types:

- The *base* types ($A, B \in \textsc{bTyp}$) can either be integers (`int`) or boolean values (`bool`).
- The *communicative* types ($C, D \in \textsc{cTyp}$) represent those types that can be sent over a channel, namely base types, channel types and process abstraction types, $\langle L \rangle$ (representing thunked processes).
- The *general* types ($S, T \in type$), made of communicative type and active process types $\{L\}$.

Our type system is substructural(Pierce, 2004), whereby type assumptions, mapping identifiers to types, are used in a controlled manner. We therefore represent our type environments, Γ, as *lists* of type assumptions whereby in particular, we can have duplicate multiple types assigned to an identifier. Type environments are manipulated using the environment structural relation $\Gamma_1 \prec \Gamma_2$ (Fig. 3) which rely on the type splitting $T_1 \circ T_2$ and the subtyping relations $T_1 <: T_2$.

Typing assumptions can be split and re-joined under certain conditions. Active processes, process abstractions and affine channels cannot be split. This way, they can only be used once. However, basic types and unrestricted assumptions can be duplicated, (pBase and pUnr), while $(\bullet, i)$ assumptions can be split into an affine assumption and a $(\bullet, i+1)$ assumption (pUnq), manifesting the duality discussed earlier. Joining is the dual of splitting.

To enhance the expressivity of our type system, we use also a notion of subtyping. When we define a type S of being a subtype of type T ($S <: T$), we mean that we can at any time use the value of type S instead of a value of type T (Pierce, 2002). The rules in Fig. 3 allow us to deduce the following subtyping chain:

$$[C]^\bullet <: [C]^{(\bullet,1)} <: [C]^{(\bullet,2)} <: \ldots <: [C]^\omega <: [C]^{\mathbf{1}}$$

We can there allow an unrestricted channel to be used in place of an affine one (but not vice-versa, as this would break it restriction constraints). As a result of subtyping combined with splitting, we can use a unique-after-n assumption as an unrestricted assumption, and then split it into 2 unrestricted assumptions ($([C]^{(\bullet,n)} <: [C]^\omega) = [C]^\omega \circ [C]^\omega$). We also can split a unique-after-n assumption into and affine assumption and a unique-after-n+1 assumption, and use the latter instead of an affine assumption ($([C]^{(\bullet,n)} = [C]^{\mathbf{1}} \circ ([C]^{(\bullet,n+1)} <: [C]^{\mathbf{1}})$).

The structural relation for environments ($\prec$) is the least reflexive transitive relation satisfying the rules in Fig. 3. The most obvious rule is weakening (tWeak), which allows extra fresh identifiers to be mapped in the environment and still type-check the process. Splitting (tCon), joining (tJoin) and subtyping (tSub) are other ways of modifying the environment in a safe way. The most important is the rule that allows revision or strong update (tRev). Since a channel is unique, it means that only one process has access to it and therefore, we can change the

type of that channel to allow different types to be communicated over it.

When there is only one type assumption for each identifier in a type environment, we say that the type environment denotes a partial function (from identifiers to types). We define a condition on type environments, consistency, that ensures that multiple type assumptions are not in conflict (this ensures that the guarantees given by unique types are indeed sound, as discussed earlier). In particular, we need to check this also for thunked process types, that may eventually be executed.

Definition 1 (Consistency)**.** A typing environment Γ is consistent if:

1. There exists a partial function Γ' such that $\Gamma' \prec \Gamma$
2. $\Gamma = \Gamma_1, x : \langle \Gamma_2 \rangle$ **implies** Γ_1, Γ_2 is consistent

♦

3.2 The Typing Relation

The typing relation is defined as two separate, mutually dependent relations. One relation is defined over values ($\vdash_v$) and the other is defined over processes ($\vdash_p$). The rules for typing relation over values are given in Fig. 4 whereas those for the typing relation over processes are given in Fig. 5.

$$\frac{}{\Gamma, i : C \vdash_v i : C}\ \textsc{tvId} \qquad \frac{\Gamma_1, \Gamma_2 \vdash_p P}{\Gamma_1 \vdash_v \langle P \rangle : \langle \Gamma_2 \rangle}\ \textsc{tvProc}$$

Figure 4: Value Typing

To type-check an identifier there needs to be a mapping for it in the typing environment (tvId). A new rule (tvProc) is introduced to type-check process abstractions with respect to some environment ($\Gamma_1 \vdash_v \langle P \rangle : \langle \Gamma_2 \rangle$). In order to do so, we need to ensure that the spawned process, P, type-checks with respect to the existing environment extended with the mappings of the thunked process type, Γ_1, Γ_2.

In Fig. 5, we have three separate rules for typing input processes ($c?x.P$) that differ only in the channel type. If c had no restrictions, (rule tpInW) we can continue to use it in an unrestricted fashion in P . If c had an affine assumption (tpInA), we can no longer use it in the continuation P. Moreover, if it has a unique-after-n assumption (tpInU), we need to account for this one communication and update the guarantee for P accordingly. Therefore, we can now guarantee that c will be unique for process P after $n - 1$ other communications. Apart from updating the channel's assumption, we also need to include the variable's type in the environment. The type will correspond to the type that channel c is allowed to communicate.

Similar reasoning is used to type-check output processes ($c!v.P$). However, we also need different rules for when we communicate an identifier or a process abstraction. When communicating a channel (rules tpOutIA, tpOutIW and tpOutIU), there should be a respective identifier type mapping in the type environment; importantly, the type-checking for the continuation process P can no longer use this type assumption. Once again we will need three separate rules for different channel assumptions.

On the other hand, we cannot use the same logic to type-check higher-order output processes ($c!\langle Q \rangle.P$). This is because the process abstraction is not in the environment. Therefore, we will need to type-check it before sending it to another process. We employ three additional rules for outputting process abstractions. The logic behind channel assumptions is the same as before. However, the process abstraction type is matched to the channel's type by value typing the process (using tvProc) with respect to an empty environment. By doing so, we are not only making sure that Q is being type-checked, but we are also making sure that it is self-contained (the environment abstracted in its type is enough to type-check it).

The rule for conditionals (tpIf) type checks both possible processes (*i.e.,* P and Q) with respect to the same typing environment as only one of them will be executed. The same environment must also type-check the values we are comparing to channel types. Even though this rule only matches channel names (as is standard in other work in process-calculi), extending the typing rule to more generic boolean conditions is straightforward.

We have two different rules for running a process (`run` $v.Q$). One of them (tpRun1) covers the case when v is a variable. This rule requires the environment to include a mapping for this variable, with its type corresponding to a process abstraction type. The rest of the environment should be able to type-check the continuation process (Q). The second rule (tpRun2) covers the case when $v = \langle P \rangle$. This time, we do not have a mapping for the process abstraction in the environment. So we divide the environment in two, part of it type checks the continuation process (Q), and the other part should be used in value typing the thunked process with respect to an empty environment.

In order to type-check a recursive process (`rec` $x.P$) we need to add the recursive variable x to the environment and use it to type-check the continuation processes P. The variable should be an active process type with the same environment used to type-check the original process. Then, when we need to type check a recursion variable, we just need to make sure that the environment contains a map for the variable that corresponds to the rest of the environment. All this is represented in the rules tpRec and tpVar.

The remaining constructs are standard. `nil` always type-checks, with any environment (tpNil). The rule for parallel processes (tpPar) needs to divide the assumptions into two environments, to type-check the parallel processes separately. If both processes need to use the same channel, it is first split using the rules in Fig. 3. The rule for channel name creation (tpRes) introduces new channels with a unique access guarantee. Finally, in tpStr, if process P type-checks with respect to an environment Γ, it should still type-check after Γ has been restructured using rules in Fig 3.

$$\frac{\Gamma \vdash_p P}{\Gamma, c : [C]^{\mathbf{1}}, j : C \vdash_p c!j.P}\ \textsc{tpOutIA} \qquad \frac{\emptyset \vdash_v \langle Q\rangle : \langle \Gamma_2\rangle \qquad \Gamma_1 \vdash_p P}{\Gamma_1, c : [\langle \Gamma_2\rangle]^{\mathbf{1}} \vdash_p c!\langle Q\rangle.P}\ \textsc{tpOutPA}$$

$$\frac{\Gamma, c : [C]^{\omega} \vdash_p P}{\Gamma, c : [C]^{\omega}, j : C \vdash_p c!j.P}\ \textsc{tpOutIW} \qquad \frac{\emptyset \vdash_v \langle Q\rangle : \langle \Gamma_2\rangle \qquad \Gamma_1, c : [\langle \Gamma_2\rangle]^{\omega} \vdash_p P}{\Gamma_1, c : [\langle \Gamma_2\rangle]^{\omega} \vdash_p c!\langle Q\rangle.P}\ \textsc{tpOutPW}$$

$$\frac{\Gamma, c : [C]^{(\bullet, n-1)} \vdash_p P}{\Gamma, c : [C]^{(\bullet, n)}, j : C \vdash_p c!j.P}\ \textsc{tpOutIU} \qquad \frac{\emptyset \vdash_v \langle Q\rangle : \langle \Gamma_2\rangle \qquad \Gamma_1, c : [\langle \Gamma_2\rangle]^{(\bullet, n-1)} \vdash_p P}{\Gamma_1, c : [\langle \Gamma_2\rangle]^{(\bullet, n)} \vdash_p c!\langle Q\rangle.P}\ \textsc{tpOutPU}$$

$$\frac{\Gamma, x : C \vdash_p P}{\Gamma, c : [C]^{\mathbf{1}} \vdash_p c?x.P}\ \textsc{tpInA} \qquad \frac{\Gamma, c : [C]^{\omega}, x : C \vdash_p P}{\Gamma, c : [C]^{\omega} \vdash_p c?x.P}\ \textsc{tpInW} \qquad \frac{\Gamma, c : [C]^{(\bullet, n-1)}, x : C \vdash_p P}{\Gamma, c : [C]^{(\bullet, n)} \vdash_p c?x.P}\ \textsc{tpInU}$$

$$\frac{}{\Gamma \vdash_p \mathtt{nil}}\ \textsc{tpNil} \qquad \frac{\Gamma_1 \vdash_p P}{\Gamma_1, x : \langle \Gamma_2\rangle \vdash_p \mathtt{run}\, x.P}\ \textsc{tpRun1} \qquad \frac{\Gamma_1 \vdash_p Q \qquad \emptyset \vdash_v \langle P\rangle : \langle \Gamma_2\rangle}{\Gamma_1, \Gamma_2 \vdash_p \mathtt{run}\, \langle P\rangle.Q}\ \textsc{tpRun2}$$

$$\frac{\Gamma, c : [C]^{\bullet} \vdash_p P}{\Gamma \vdash_p (\mathtt{new}\, c)P}\ \textsc{tpRes} \qquad \frac{\Gamma_1 \vdash_p P \quad \Gamma_2 \vdash_p Q}{\Gamma_1, \Gamma_2 \vdash_p P \mid Q}\ \textsc{tpPar} \qquad \frac{\Gamma, x : \{\Gamma\} \vdash_p P}{\Gamma \vdash_p \mathtt{rec}\, x.P}\ \textsc{tpRec} \qquad \frac{}{\Gamma, x : \{\Gamma\} \vdash_p x}\ \textsc{tpVar}$$

$$\frac{\Gamma \prec \Gamma' \qquad \Gamma' \vdash_p P}{\Gamma \vdash_p P}\ \textsc{tpStr} \qquad \frac{\Gamma \vdash_v v : [C_1]^{a_1} \qquad \Gamma \vdash_v u : [C_2]^{a_2} \qquad \Gamma \vdash_p P \qquad \Gamma \vdash_p Q}{\Gamma \vdash_p \mathtt{if}\ (v = u)\ \mathtt{then}\ P\ \mathtt{else}\ Q}\ \textsc{tpIf}$$

Figure 5: Process Typing

4 Results

We prove soundness for our type system with respect to the errors formalised in Fig. 2. As is standard, we do so by proving Subject Reduction (Thm. 1) and Safety (Thm. 2)

Theorem 1 (Subject Reduction)**.**

if Γ **is consistent and** $\Gamma \vdash_p P$ **and** $P \longrightarrow P'$ **implies there exists** Γ' **such that** Γ' **is consistent and** $\Gamma' \vdash_p P'$

Proof. By rule induction on $\Gamma \vdash_p P$. See (Zammit, 2012) □

Theorem 2 (Type Safety)**.**

$$\Gamma \vdash_p P \textbf{ implies } P \not\longrightarrow_{\text{ERR}}$$

Proof. By rule induction on $\Gamma \vdash_p P$. See (Zammit, 2012) □

We are also able to type-check the programs (1) and (2) discussed in the introduction and reject the erroneous programs discussed there. Repeated applications of Thm. 1 ensure that typed programs will remain typed when they compute, whereas Thm. 2 ensures that as long as they type-check, programs never produce an error.

5 Conclusion and Future Work

We have presented a sound type system for reasoning statically about higher-order pi-calculus programs using strong updates. This extended the work in De Vries et al. (2012), which did not consider higher-order communications.

In De Vries et al. (2012) (of which, this work is an immediate extension) the authors give an extensive discussion of related work and type systems. One possible avenue for future work is that of extending our type systems with input/output modalities as in Hennessy (2007), yielding richer notions of channel subtyping. One could also extend uniqueness typing to session types (Honda et al., 1998).

References

Ahmed, A., Fluet, M. and Morrisett, G. (2005). A step-indexed model of substructural state. In *Acm sigplan notices* (Vol. 40, pp. 78–91). ACM.

Armstrong, J. (2007). *Programming erlang*. The Pragmatic Bookshelf.

The Go Programming Language. (nodate). `http://golang.org/`.

Cesarini, F. and Thompson, S. (2009). *Erlang programming*. O'Reilly.

De Vries, E., Francalanza, A. and Hennessy, M. (2012). Uniqueness typing for resource management in message-passing concurrency. *J. Logic. Computation.* 24(3), 531–556.

Hennessy, M. (2007). *A distributed pi-calculus*. Cambridge University Press.

Honda, K., Vasconcelos, V. T. and Kubo, M. (1998). Language primitives and type disciplines for structured communication-based programming. In *Esop* (Vol. 1381, pp. 22–138). LNCS. Springer.

Kobayashi, N., Pierce, B. and Turner, D. (1999). Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21(5), 914–947.

Pierce, B. (2002). *Types and programming languages*. MIT press.

Pierce, B. (2004). *Advanced topics in types and programming languages*. MIT press.

Sangiorgi, D. and Walker, D. (2003). *The pi-calculus: a theory of mobile processes*. Cambridge University Press.

Zammit, M. (2012). *A type system for a higher-order language*. University of Malta.