Uniqueness Typing for Resource Management in Message-Passing Concurrency

Edsko de Vries[*]
Trinity College Dublin, Ireland
Edsko.de.Vries@cs.tcd.ie

Adrian Francalanza
University of Malta
Adrian.Francalanza@um.edu.mt

Matthew Hennessy[*]
Trinity College Dublin, Ireland
Matthew.Hennessy@cs.tcd.ie

We view channels as the main form of resources in a message-passing programming paradigm. These channels need to be carefully managed in settings where resources are scarce. To study this problem, we extend the pi-calculus with primitives for channel allocation and deallocation and allow channels to be reused to communicate values of different types. Inevitably, the added expressiveness increases the possibilities for runtime errors. We define a substructural type system which combines uniqueness typing and affine typing to reject these ill-behaved programs.

1 Introduction

Message-passing concurrency is a programming paradigm whereby shared memory is prohibited and process interaction is limited to explicit message communication. This concurrency paradigm forms the basis for a number of process calculi such as the pi-calculus [15] and has been adopted by programming languages such as the actor based language Erlang [3].

Message-passing concurrency often abstracts away from resource management and programs written at this abstraction level exhibit poor resource awareness. In this paper we study ways of improving this shortcoming. Specifically, we develop a statically typed extension of the pi-calculus in which resources, i.e. channels, can be reused at varying types and unwanted resources can be safely deallocated.

Idiomatic pi-calculus processes are often characterized by wasteful use-once-throw-away channels [15, 14]. Consider the following two pi-calculus process definitions

$$\text{TIMESRV} \triangleq \text{rec}\, X.getTime?x.x!\langle hr, min \rangle.X$$

$$\text{DATESRV} \triangleq \text{rec}\, X.getDate?x.x!\langle year, mon, day \rangle.X$$

TIMESRV defines a server that repeatedly waits on a channel named *getTime* to dynamically receive a channel name, represented by the bound variable x, and then replies with the current time on x. DATESRV is a similar service which returns the current date. An idiomatic pi-calculus client definition is

$$\text{CLIENT}_0 \triangleq (\nu ret_1)\, getTime!\langle ret_1 \rangle.ret_1?(y_{hr}, y_{min}).(\nu ret_2)\, getDate!\langle ret_2 \rangle.ret_2?(z_{year}, z_{mon}, z_{day}).P$$

CLIENT$_0$ uses two distinct channels ret_1 and ret_2 as return channels to query the time and date servers, and then continues as process P with the values obtained. These return channels are scoped (private), to preclude interference from other clients concurrently querying the servers.

From a resource management perspective it makes pragmatic sense to try and reduce the number of channels used and use *one* channel to communicate with both the time server and the date server.

$$\text{CLIENT}_1 \triangleq (\nu ret)\, getTime!\langle ret \rangle.ret?(y_{hr}, y_{min}).getDate!\langle ret \rangle.ret?(z_{year}, z_{mon}, z_{day}).P$$

From a typing perspective, this reuse of the same channel entails *strong update* on the channel: that is, reuse of a channel to communicate values of different types. Strong update must be carefully controlled; for instance, an attempt to use one channel to communicate with both servers in parallel is unsafe:

$$\text{CLIENT}_{err} \triangleq (\nu ret) \left(getTime!\langle ret \rangle.ret?(y_{hr}, y_{min}).P_1 \parallel getDate!\langle ret \rangle.ret?(z_{year}, z_{mon}, z_{day}).P_2 \right)$$

Standard pi-calculus type systems accept only CLIENT_0 and rule out both CLIENT_1 and CLIENT_{err}. However, CLIENT_1 is safe because the communication with the date server happens *strictly after* the communication with the time server, and also because the time server will only use the return channel *once*.

Adequate resource management also requires precise descriptions of when resources are allocated and existing ones are disposed. The characteristic scoping construct $(\nu c)P$ is unfit for this purpose and should be used only as a bookkeeping construct delineating name scoping (which evolves during computation through scope extrusion) . One reason against such an interpretation is the structural rule

$$(\text{SSCP}) \qquad P \equiv (\nu c)P \quad \text{whenever} \quad c \notin \mathbf{fn}(P)$$

whose symmetric nature would entail implicit garbage collection of channels, but also the possibility of unfettered spurious channel allocations. Another reason against this interpretation is the fact that the pi-calculus semantics does not specify whether, in a process such as CLIENT_0, channel ret_2 is allocated before or after the input on ret_1. This problem becomes more acute when scoping occurs inside recursive definitions. We address this shortcoming by introducing an explicit allocation construct $\text{alloc}(x).P$. When the allocation is executed, a new channel c is created at runtime and the $\text{alloc}(x).P$ changes to $(\nu c)P\{c/x\}$. Dually, we also extend the calculus with an explicit *deallocation* operator $\text{free}\,c.P$. We can then rewrite the client as:

$$\text{CLIENT}_2 \triangleq \text{alloc}(x).getTime!\langle x \rangle.x?(y_{hr}, y_{min}).getDate!\langle x \rangle.x?(z_{year}, z_{mon}, z_{day}).\text{free}\,x.P$$

Inevitably, the added expressiveness of this extended pi-calculus increases the possibilities for runtime errors like value mismatch during communication and usage of channels which have been deallocated.

We define a type system which rejects processes that are unsafe; the type system combines uniqueness typing [4] and affine typing [14], while permitting value coercion across these modes through subtyping. Uniqueness typing gives us *global guarantees*, simplifying local reasoning when typing both strong updates and safe deallocations. Uniqueness typing can be seen as dual to affine typing [10], and we make essential use of this duality to allow uniqueness to be temporarily violated.

2 The Resource Pi-Calculus

Fig. 1 shows the syntax for the resource pi-calculus. The language is the standard pi-calculus extended with primitives for explicit channel allocation and deallocation; moreover, channel scoping records whether a channel is allocated ($\top$) or deallocated ($\bot$). The syntax assumes two separate denumerable sets of channel names $c, d \in \text{NAMES}$ and variables $x, y \in \text{VARS}$, and lets identifiers $u, v \in \text{NAMES} \cup \text{VARS}$ range over both. The input and channel allocation constructs are binders for variables $\vec{x}$ and x *resp.*, whereas scoping is a binder for names (*i.e., c*). The syntax also assumes a denumerable set of process variables $X, Y \in \text{PVARS}$ which are bound by the recursion construct.

Channels are stateful (allocated, $\top$, or deallocated, $\bot$) and process semantics is defined over configurations, $\langle \sigma, P \rangle$ where $\sigma \in \Sigma : \text{CHANS} \rightharpoonup \{\top, \bot\}$ describes the state of the free channels in P, and

$$
\begin{array}{llll}
P,Q ::= & u!\vec{v}.P & \textit{(output)} & \mid \quad u?\vec{x}.P & \textit{(input)} \\
& \mid \;\; \text{nil} & \textit{(nil)} & \mid \quad \text{if } u\!=\!v \text{ then } P \text{ else } Q & \textit{(match)} \\
& \mid \;\; \text{rec}\,X.P & \textit{(recursion)} & \mid \quad X & \textit{(process variable)} \\
& \mid \;\; P\,\|\,Q & \textit{(parallel)} & \mid \quad (\nu c\!:\!s)P & \textit{(stateful scoping)} \\
& \mid \;\; \text{alloc}(x).P & \textit{(allocate)} & \mid \quad \text{free}\,u.P & \textit{(deallocate)}
\end{array}
$$

Figure 1: Polyadic resource pi-calculus syntax

stateful scoping $(\nu c\!:\!s)P$ describes the state of scoped channels. A tuple $\langle \sigma, P \rangle$ is a configuration whenever $\mathbf{fn}(P) \subseteq \mathbf{dom}(\sigma)$ and is denoted as $\sigma \triangleright P$. We say that a configuration $\sigma \triangleright P$ is closed whenever P does not have any free variables. Fig. 2 defines contexts over configuration where $\mathscr{C}[\sigma \triangleright P]$ denotes the application of a context $\mathscr{C}$ to a configuration $\sigma \triangleright P$. In the case where a context scopes a name c the definition extracts the state relating to c from σ and associates it with c. For example,

$$
(\nu c)[\,c\!:\!\top, d\!:\!\top \triangleright c!d\,] = d\!:\!\top \triangleright (\nu c\!:\!\top)c!d
$$

The reduction relation is defined as the least contextual relation over closed configurations satisfying the rules in Fig. 2, using a standard pi-calculus structural equivalence relation ($\equiv$). Communication (RCOM) requires the communicating channel to be allocated. Allocation (RALL) creates a private allocated channel and substitutes it for the bound variable of the allocation construct in the continuation; the condition $c \notin \mathbf{dom}(\sigma)$ ensures that c is fresh in P. Deallocation (RFREE) is the only construct that changes the visible state of a configuration, σ.

Fig. 2 also defines error reductions as the least contextual relation satisfying the rules for $\xrightarrow{\text{err}}$. These rules capture errors resulting from arity mismatch and attempts to communicate on deallocated channels. In particular arity mismatch can come from unconstrained use of strong updates such as in the case of CLIENT$_{\text{err}}$ in the Introduction.

3 Type System

3.1 The typing relation

The type language is defined in Figure 3 and the typing rules are given in Figure 4. The typing relation over processes takes the usual shape: $\Gamma \vdash P$ read as "P is well-typed under the typing assumptions in Γ". Typing environments are multisets of pairs of identifiers and types; we do not *a priori* make the assumption that they are partial functions (Section 3.5). This relation is extended to configurations as

$$
\Gamma \Vdash \sigma \triangleright P
$$

by requiring that the process is well-typed in Γ, all channels in Γ are allocated in σ, and Γ is a partial function (*i.e.*, for every identifier in P there is exactly one typing assumption in Γ).

3.2 The type language

The core type of our system is the channel type, $[\overrightarrow{\mathbf{T}}]^a$, consisting of an n-ary tuple of types $\overrightarrow{\mathbf{T}}$ describing the values carried over the channel and an *attribute a* which gives usage information about the channel

Contexts

$$\mathscr{C} ::= \quad [-] \quad | \quad \mathscr{C} \parallel P \quad | \quad P \parallel \mathscr{C} \quad | \quad (\nu c)\mathscr{C}$$

$$
\begin{aligned}
[\sigma \triangleright P] &\stackrel{\text{def}}{=} \sigma \triangleright P \\
\mathscr{C}[\sigma \triangleright P] \parallel Q &\stackrel{\text{def}}{=} \sigma' \triangleright (P' \parallel Q) \quad \text{if } \mathscr{C}[\sigma \triangleright P] = \sigma' \triangleright P' \\
Q \parallel \mathscr{C}[\sigma \triangleright P] &\stackrel{\text{def}}{=} \sigma' \triangleright (Q \parallel P') \quad \text{if } \mathscr{C}[\sigma \triangleright P] = \sigma' \triangleright P' \\
(\nu c)\mathscr{C}[\sigma \triangleright P] &\stackrel{\text{def}}{=} \sigma' \triangleright (\nu c{:}s)P' \quad \text{if } \mathscr{C}[\sigma \triangleright P] = \sigma', c{:}s \triangleright P'
\end{aligned}
$$

Reduction Rules

$$\frac{\sigma(c) = \top}{\sigma \triangleright c!\vec{b} \parallel c?\vec{x}.P \longrightarrow \sigma \triangleright P\{\vec{b}/\vec{x}\}} \text{ RCom} \qquad \frac{}{\sigma \triangleright \operatorname{rec} X.P \longrightarrow \sigma \triangleright P\{\operatorname{rec} X.P/X\}} \text{ RRec}$$

$$\frac{}{\sigma \triangleright \text{if } c = c \text{ then } P \text{ else } Q \longrightarrow \sigma \triangleright P} \text{ RThen} \qquad \frac{c \neq d}{\sigma \triangleright \text{if } c = d \text{ then } P \text{ else } Q \longrightarrow \sigma \triangleright Q} \text{ RElse}$$

$$\frac{c \notin \mathbf{dom}(\sigma)}{\sigma \triangleright \operatorname{alloc}(x).P \longrightarrow \sigma \triangleright (\nu c{:}\top)\big(P\{c/x\}\big)} \text{ RAll} \qquad \frac{}{\sigma, c{:}\top \triangleright \operatorname{free} c.P \longrightarrow \sigma, c{:}\bot \triangleright P} \text{ RFree}$$

$$\frac{P \equiv P' \qquad \sigma \triangleright P' \longrightarrow \sigma \triangleright Q' \qquad Q' \equiv Q}{\sigma \triangleright P \longrightarrow \sigma \triangleright Q} \text{ RStr}$$

Error Reduction Rules

$$\frac{|\vec{d}| \neq |\vec{x}|}{\sigma \triangleright c!\vec{d}.P \parallel c?\vec{x}.Q \xrightarrow{\text{err}}} \text{ EAty} \quad \frac{\sigma(c) = \bot}{\sigma \triangleright c!\vec{d}.P \xrightarrow{\text{err}}} \text{ EOut} \quad \frac{\sigma(c) = \bot}{\sigma \triangleright c?\vec{x}.Q \xrightarrow{\text{err}}} \text{ EIn} \quad \frac{P \equiv Q \quad \sigma \triangleright Q \xrightarrow{\text{err}}}{\sigma \triangleright P \xrightarrow{\text{err}}} \text{ EStr}$$

Figure 2: Contexts, Reduction Rules and Error Predicates

$$
\begin{array}{llll}
\mathbf{T} & ::= & [\overrightarrow{\mathbf{T}}]^a & \text{(channel type)} \\
 & | & \mathbf{proc} & \text{(process)}
\end{array}
\qquad
\begin{array}{llll}
a & ::= & 1 & \text{(affine)} \\
 & | & \omega & \text{(unrestricted)} \\
 & | & (\bullet, i) & \text{(unique after } i \text{ steps, } i \in \mathbb{N})
\end{array}
$$

Figure 3: Type language

(a channel is *used* when communication takes place across the channel). This attribute can take one of three forms:

- A channel of type $[\vec{\mathbf{T}}]^{\omega}$ is an *unrestricted* channel; such type assumptions correspond to type assumptions of the form $[\vec{\mathbf{T}}]$ in non-substructural type systems.

- A channel of type $[\vec{\mathbf{T}}]^1$ is *affine*, and comes with an obligation: it can be used at most once.

- A channel of type $[\vec{\mathbf{T}}]^{(\bullet,i)}$ comes with a guarantee that it is *unique* after i actions; we abbreviate the type $[\vec{\mathbf{T}}]^{(\bullet,0)}$ of channels that are unique *now* to $[\vec{\mathbf{T}}]^{\bullet}$. Unique channels can be used to describe instances where only one process has access to (owns) a channel. Accordingly, strong update and deallocation is safe for unique channels.

3.3 Structural rules

Since the type system is substructural, usage of type assumptions must be carefully controlled, and the logical rules do not allow to use an assumption more than once. Operations on the typing environment are described separately by the structural rules of Fig. 4. Although the subtyping relation is novel because it combines uniqueness subtyping (SUNQ) with affine subtyping (SAFF) making them *dual* with respect to unrestricted types, the subtyping and weakening structural rules are standard.

Rule TCON contracts an assumption of the form $u : \mathbf{T}$, as long as $\mathbf{T}$ can be *split* as $\mathbf{T}_1$ and $\mathbf{T}_2$. Unrestricted assumptions can be split arbitrarily (PUNR and PPROC); affine assumptions cannot be split at all. An assumption $u : [\vec{\mathbf{T}}]^{\bullet}$ about a unique channel can be split as an affine assumption $u : [\vec{\mathbf{T}}]^1$ and an assumption about a channel that is unique after one action $u : [\vec{\mathbf{T}}]^{(\bullet,1)}$—an action using the latter assumption must be coupled with a co-action on the affine channel, and since the affine assumption can only be used once it is sound to assume that the channel is unique again after the action has happened. More generally, a channel which is unique after i actions can be split into an affine assumption and a channel which is unique after $(i+1)$ actions, and splitting is defined in such a way that the number of affine assumptions for a channel never exceeds the index i of the corresponding unique assumption.

In particular when the index is 0 no other assumptions about that channel can exist in the typing environment. This means that if a process can be typed using a unique assumption for a channel, no other process has access to that channel. The last structural rule, TREV, makes use of this fact to allow strong updates ("revision") to channels as long as they are unique.

3.4 Logical rules

The rules for input and output (TOUT and TIN) decrement the attribute of the channel, *i.e.*, they *count* usage. This operation is denoted by $\Gamma, c : [\vec{\mathbf{T}}]^{a-1}$ in Fig. 4 and states that affine assumptions can only be used once, unrestricted assumptions can be used an arbitrary number of types, and if a channel is unique after $i+1$ actions, then it will be unique after i actions once the action has been performed.

Moreover, TOUT requires separate typing assumptions for each of the channels that are sent. The attributes on these channels are not decremented, because no action has been performed on them; instead, the corresponding assumptions are handed over to the parallel process receiving the message. If the sending process wants to use any of these channels in its continuation (P) it must split the corresponding assumptions first.

Recursive processes must be typed in an environment that contains only unrestricted channels (TREC). This is reasonable since recursion can be used to define processes with an unbounded number of parallel

Logical rules

$$\frac{\Gamma,u:[\overrightarrow{\mathbf{T}}]^{a-1},\overrightarrow{x:\mathbf{T}}\vdash P}{\Gamma,u:[\overrightarrow{\mathbf{T}}]^{a}\vdash u?\vec{x}.P}\text{ TIN} \qquad \frac{\Gamma,u:[\overrightarrow{\mathbf{T}}]^{a-1}\vdash P}{\Gamma,u:[\overrightarrow{\mathbf{T}}]^{a},\overrightarrow{v:\mathbf{T}}\vdash u!\vec{v}.P}\text{ TOUT} \qquad \frac{\Gamma_1\vdash P \qquad \Gamma_2\vdash Q}{\Gamma_1,\Gamma_2\vdash P\|Q}\text{ TPAR}$$

$$\frac{u,v\in\Gamma \quad \Gamma\vdash P \quad \Gamma\vdash Q}{\Gamma\vdash \text{if }u=v\text{ then }P\text{ else }Q}\text{ TIF} \qquad \frac{\Gamma^{\omega},X:\mathbf{proc}\vdash P}{\Gamma^{\omega}\vdash \text{rec}X.P}\text{ TREC} \qquad \frac{}{X:\mathbf{proc}\vdash X}\text{ TVAR}$$

$$\frac{\Gamma,x:[\overrightarrow{\mathbf{T}}]^{\bullet}\vdash P}{\Gamma\vdash \text{alloc}(x).P}\text{ TALL} \qquad \frac{\Gamma\vdash P}{\Gamma,u:[\overrightarrow{\mathbf{T}}]^{\bullet}\vdash \text{free}\,u.P}\text{ TFREE} \qquad \frac{}{\emptyset\vdash \text{nil}}\text{ TNIL}$$

$$\frac{\Gamma,c:\mathbf{T}\vdash P}{\Gamma\vdash (\nu c:\top)P}\text{ TRST1} \qquad \frac{\Gamma\vdash P}{\Gamma\vdash (\nu c:\bot)P}\text{ TRST2}$$

where Γ^{ω} is an environment containing only unrestricted assumptions and all bound variables are fresh.

Structural rules

$$\frac{\mathbf{T}=\mathbf{T}_1\circ\mathbf{T}_2 \quad \Gamma,u:\mathbf{T}_1,u:\mathbf{T}_2\vdash P}{\Gamma,u:\mathbf{T}\vdash P}\text{ TCON} \qquad \frac{\Gamma\vdash P}{\Gamma,u:\mathbf{T}\vdash P}\text{ TWEAK}$$

$$\frac{\Gamma,u:\mathbf{T}_2\vdash P \quad \mathbf{T}_1<:\mathbf{T}_2}{\Gamma,u:\mathbf{T}_1\vdash P}\text{ TSUB} \qquad \frac{\Gamma,u:[\overrightarrow{\mathbf{T}_2}]^{\bullet}\vdash P}{\Gamma,u:[\overrightarrow{\mathbf{T}_1}]^{\bullet}\vdash P}\text{ TREV}$$

Typing configurations

$$\frac{\forall c\in\mathbf{dom}(\Gamma).\,\sigma(c)=\top \qquad \Gamma\vdash P \qquad \Gamma\text{ is a partial map}}{\Gamma\Vdash \sigma\triangleright P}\text{ TCONF}$$

Channel usage

$$\Gamma,c:[\overrightarrow{\mathbf{T}}]^{a-1}\overset{\mathbf{def}}{=}\begin{cases}\Gamma & \text{if }a=1\\ \Gamma,c:[\overrightarrow{\mathbf{T}}]^{\omega} & \text{if }a=\omega\\ \Gamma,c:[\overrightarrow{\mathbf{T}}]^{(\bullet,i)} & \text{if }a=(\bullet,i+1)\end{cases}$$

Type splitting

$$\frac{}{[\overrightarrow{\mathbf{T}}]^{\omega}=[\overrightarrow{\mathbf{T}}]^{\omega}\circ[\overrightarrow{\mathbf{T}}]^{\omega}}\text{ PUNR} \qquad \frac{}{\mathbf{proc}=\mathbf{proc}\circ\mathbf{proc}}\text{ PPROC} \qquad \frac{}{[\overrightarrow{\mathbf{T}}]^{(\bullet,i)}=[\overrightarrow{\mathbf{T}}]^{1}\circ[\overrightarrow{\mathbf{T}}]^{(\bullet,i+1)}}\text{ PUNQ}$$

Subtyping

$$\frac{}{(\bullet,i)<:(\bullet,i+1)}\text{ SINDX} \qquad \frac{}{(\bullet,i+1)<:\omega}\text{ SUNQ} \qquad \frac{}{\omega<:1}\text{ SAFF} \qquad \frac{a_1<:a_2}{[\overrightarrow{\mathbf{T}}]^{a_1}<:[\overrightarrow{\mathbf{T}}]^{a_2}}\text{ STYP}$$

Figure 4: Typing rules

uses of some channel. Nevertheless, it is not as serious a restriction as it may seem ,as recursive processes can still send, receive and allocate unique channels. For instance, the following process models an "infinite heap" that keeps allocating new channels and sends them across a channel $heap : [[\mathbf{T}]^\bullet]^\omega$:

$$\textsc{InfHeap} \triangleq \mathsf{rec}\, X.\mathsf{alloc}(x).heap!x.X$$

As expected, allocation introduces unique channels (TALLOC) and only unique channels can be deallocated (TFREE). Finally, a typing assumption is introduced in the typing environment for locally scoped names only if the corresponding channel is allocated (TRST1 and RRST2).

3.5 Consistency

When we take a bird's eye-view of a system, every channel has exactly one type. In the definition of $\Gamma \Vdash \sigma \triangleright P$, we therefore restrict the environment to have at most one assumption about every channel: we require that Γ is a partial function. Consequently, we state type safety and subject reduction lemmas with respect to environments that are partial functions.

Nevertheless, when two processes both need a typing assumption relating to the same channel c, there need to be two separate assumptions in the typing environment (*cf.* TPAR in Fig. 4). These two assumptions need not be identical; for example, an assumption $c : []^\bullet$ can be split as two assumptions

$$c : []^{(\bullet,1)}, c : []^1$$

A *consistent* environment is defined to be an environment that can be obtained by applying any of the structural rules (contraction, weakening, subtyping or revision) from an environment which is a partial function. It follows that any process that can be typed in a consistent environment can also be typed in a environment that is a partial function. The reader may therefore wonder why we do not restrict the typing relation to partial functions. It turns out that even if a process can be typed in a consistent environment, some of its subprocesses might have to be typed in an inconsistent environment. As an example, consider the typing derivation

$$
\cfrac{
\cfrac{}{a : [[]^1]^\omega, u : []^\bullet \vdash a!u} \text{\small TOUT}
\qquad
\cfrac{
\cfrac{
\cfrac{a : [[]^1]^\omega, u : []^\bullet, x : []^1 \vdash \mathsf{free}\, u \,\|\, a!x}{a : [[]^1]^\omega, u : []^{(\bullet,1)}, x : []^1 \vdash u?().(\mathsf{free}\, u \,\|\, a!x)} \text{\small TIN}
}{a : [[]^1]^\omega, u : []^{(\bullet,1)} \vdash a?x.u?().(\mathsf{free}\, u \,\|\, a!x)} \text{\small TIN}
}
{a : [[]^1]^\omega, a : [[]^1]^\omega, u : []^1, u : []^{(\bullet,1)} \vdash a!u \,\|\, a?x.u?().(\mathsf{free}\, u \,\|\, a!x)} \text{\small TPAR}
}
{a : [[]^1]^\omega, u : []^\bullet \vdash a!u \,\|\, a?x.u?().(\mathsf{free}\, u \,\|\, a!x)} \text{\small TCON (twice)}
$$

This is a valid typing derivation, and moreover the typing environment used at every step is consistent. But now consider what happens after this process takes a reduction step:

$$
\cfrac{
\cfrac{u : []^\bullet, u : []^1 \vdash \mathsf{free}\, u \,\|\, a!u}{u : []^{(\bullet,1)}, u : []^1 \vdash u?().(\mathsf{free}\, u \,\|\, a!u)} \text{\small TIN}
}
{u : []^\bullet \vdash u?().(\mathsf{free}\, u \,\|\, a!u)} \text{\small TCON}
$$

The tail of this process looks suspicious as it attempts to free u while simultaneously sending it on a. Indeed, $\mathsf{free}\, u \,\|\, a!u$ can only be typed in an inconsistent environment $u : []^\bullet, u : []^1$. Nevertheless, the fact that this process is typeable is not a violation of type safety. The assumption $u : []^\bullet$ tells us that there are no processes that output on u so that the input on u is blocked: the tail of the process will never execute.

Thus, when an environment

$$\Gamma, c : [\vec{\mathbf{T}}]^a, c : [\vec{\mathbf{T}}]^{a'}$$

(e.g., $u : []^{(\bullet, 1)}, u : []^1$) is consistent, it may be the case that

$$\Gamma, c : [\vec{\mathbf{T}}]^{a-1}, c : [\vec{\mathbf{T}}]^{a'}$$

(e.g., $u : []^\bullet, u : []^1$) is *inconsistent*: this means that the tails of input or output processes may have to be typed under inconsistent environments, even when the larger process is typed in a consistent environment.

However, communication in the pi-calculus provides synchronization points: when a communication happens, two processes will *both* start executing their tail processes. The following lemma says that if this happens, the resulting overall system (of both processes) is still typeable in a consistent environment; in fact, both must be derivable from the *same* partial function. This lemma is crucial in the subject reduction proof.

Lemma 1. *Let $\Gamma, u : [\vec{\mathbf{T}}]^{a_1}, u : [\vec{\mathbf{T}}]^{a_2}$ be a consistent environment, i.e., derivable from a partial function Γ' by applying structural rules. Then $\Gamma, u : [\vec{\mathbf{T}}]^{a_1-1}, u : [\vec{\mathbf{T}}]^{a_2-1}$ is derivable from the same environment Γ', and is therefore consistent.*

3.6 Soundness

We prove soundness of the type system in the usual way.

Theorem 2 (Type safety). *If $\Gamma \Vdash \sigma \triangleright P$ then $P \not\rightarrow^{err}$.*

Theorem 3 (Subject reduction). *If $\Gamma \Vdash \sigma \triangleright P$ and $\sigma \triangleright P \rightarrow \sigma' \triangleright P'$ then there exists a environment Γ' such that $\Gamma' \Vdash \sigma' \triangleright P'$.*

Taken together these two theorems imply that a well-typed process will not have any runtime errors. The proofs of these theorems can be found in the accompanying technical appendix [21].

3.7 Examples

The systems $\text{CLIENT}_i \parallel \text{TIMESRV} \parallel \text{DATESRV}$ for $i \in \{0, 1, 2\}$ can all be typed in our type system, whereas CLIENT_{err} is rejected because type splitting enforces a common object type (*cf.* PUNR, PUNQ in Fig. 4.) For convenience, we here recall CLIENT_2 and consider how it is typed, assuming x is not free in P:

$$\text{CLIENT}_2 \triangleq \mathsf{alloc}(x).getTime!\langle x \rangle.x?(y_{hr}, y_{min}).getDate!\langle x \rangle.x?(z_{year}, z_{mon}, z_{day}).\mathsf{free}\,x.P$$

Assuming an environment with $getTime : [[\mathbf{T}_1, \mathbf{T}_2]^1]^\omega$ and $getDate : [[\mathbf{T}_3, \mathbf{T}_4, \mathbf{T}_5]^1]^\omega$, CLIENT_2 types as follows. Rule TALL assigns the unique type $[\mathbf{T}_1, \mathbf{T}_2]^\bullet$ to variable x and the structural rule TCON then splits this unique assumption in two using PUNQ. Rule TOUT uses the affine assumption for x for the output argument and the unique-after-one assumption to type the continuation. Rule TIN restores the uniqueness of x for the continuation of the input after decrementing the uniqueness index, at which point TREV is applied to change the object type of x from pairs of integers (for time) to triples of integers (for dates). The pattern of applying TCON, TOUT and TIN repeats, at which point x is unique again and can be safely deallocated by TFREE.

Uniqueness allows us to typecheck a third client variation manifesting (explicit) ownership transfer. Rather than allocating a new channel, CLIENT_3 requests a channel from a heap of channels and returns the channel to the heap when it no longer needs it, thereby reusing channels *across* clients.

$$\text{CLIENT}_3 \triangleq heap?x.getTime!\langle x \rangle.x?(y_{hr}, y_{min}).getDate!\langle x \rangle.x?(z_{year}, z_{mon}, z_{day}).heap!x.P$$

With the assumption $heap : [[\mathbf{T}]^{\bullet}]^{\omega}$ typing the system below is analogous to the previous client typings.

$$\text{CLIENT}_3 \parallel \text{CLIENT}_3 \parallel \text{CLIENT}_3 \parallel \text{TIMESRV} \parallel \text{DATESRV} \parallel (vc)heap!\langle c \rangle \qquad (1)$$

In CLIENT$_2$ and CLIENT$_3$, substituting $x!().Q$ for P makes the clients unsafe (they perform *resp.* deallocated-channel usage and mismatching communication). Both clients would be rejected by the type system because the use of x in $x!().Q$ requires a split for the assumption of x, and it is not possible to split any assumption into a unique assumption and any other assumption.

Finally, system (1) above can be safely extended with processes such as CLIENT$_4$ which uses unique channels obtained from the heap in unrestricted fashion. Our type system accepts CLIENT$_4$ by applying *subtyping* from unique to unrestricted on the channel x obtained from *heap*.

$$\text{CLIENT}_4 \triangleq heap?x.\mathrm{rec}\,X.(getTime!\langle x \rangle.x?(y_{hr}, y_{min}).P \parallel X)$$

4   Related Work

The literature on using substructural logics to support destructive or strong updates is huge and we can give but a brief overview here. More in-depth discussions can be found in [20, 17].

Resources and pi-calculus   Resource usage in a pi-calculus extension is studied in [19] but it differs from our work in many respects. For a start, their scoping construct assumes an allocation semantics while we tease scoping and allocation apart as separate constructs. The resource reclamation construct in [19] is at a higher level of abstraction than $free\,c.P$, and acts more like a "resource finaliser" leading to garbage collection. Resource reclamation is implicit in [19], permitting different garbage collection policies for the same program whereas in the resource pi-calculus resource reclamation is explicit and fixed for every program. The main difference however concerns the aim of the type systems: our type system ensures safe channel deallocation and reuse; the type system in [19] statically determines an upper bound for the number of resources used by a process and does not use substructural typing.

Linearity versus Uniqueness   In the absence of subtyping, affine typing and uniqueness typing coincide but when subtyping is introduced they can be considered dual [10]. For linear typing, the subtyping relation allows coercing non-linear assumptions into a linear assumptions, *i.e.,* $!U \to U$, but for uniqueness typing, the subtyping relation permits coercing unique assumptions into non-unique assumptions. Correspondingly, the interpretation is different: linearity (*resp.* affinity) is a *local obligation* that a channel must be used exactly (*resp.* at most) once, while uniqueness is a *global guarantee* that no other processes have access to the channel. Combining both subtyping relations as we have done in this paper appears to be novel. The usefulness of the subtyping relation for affine or linear typing is well-known (e.g., see [12]); subtyping for unique assumptions allows to "forget" the uniqueness guarantee; CLIENT$_4$ above shows one scenario where this might be useful.

Linearity in functional programming   In pure functional programming languages, data structures are always persistent and destructive updates are not supported: mapping a function f across a list $[x_1, \ldots, x_n]$ yields a *new* list $[f\,x_1, \ldots, f\,x_n]$, leaving the old list intact. However, destructive updates cannot always be avoided (*e.g.,* when modelling system I/O [1]) and are sometimes required for efficiency (*e.g.,* updating arrays). Substructural type systems can be used to support destructive update without losing referential transparency: destructive updates are only allowed on terms that

are not shared. Both uniqueness typing [4] and linear typing have been used for this purpose, although even some proponents of linear typing agree that the match is not perfect [23, Section 3].

In functional languages with side effects, substructural type systems have been used to support strong (type changing) updates. For instance, Ahmed *et al.* have applied a linear type system to support "strong" (type changing) updates to ML-style references [2] in a setting with no subtyping.

It has been recognized early on that it is useful to allow the uniqueness of an object to be temporarily violated. In functional languages, this typically takes the form of a sequential construct that allows a unique object (such as an array) to be regarded as non-unique to allow multiple non-destructive accesses (such as multiple reads) after which the uniqueness is recovered again. Wadler's `let!` construct [22] (or the equivalent Clean construct `#!`) and observer types [16] both fall into this category, and this approach has also been adopted by some non-functional languages where it is sometimes called *borrowing* [7]. It is however non-trivial to extend this approach to a concurrent setting with a partial order over execution steps; our approach can be regarded as one preliminary attempt to do so.

Strong update in the presence of sharing There is substantial research on type systems that allow strong update even in the presence of sharing; the work on alias types and Vault [18, 24, 8] and on CQual [9] are notable examples of this. These type systems do explicit alias analysis by reflecting memory locations at the type level through singleton types. This makes it possible to track within the type system that a strong (type changing) update to one variable changes the type of all its aliases. The interpretation of unique (or linear) in these systems is different: a unique reference (typically called a *capability* in this context) does not mean that there is only a single reference to the object, but rather that all its aliases are known. For non-unique reference not all aliases are known and so strong update is disallowed.

These systems are developed for imperative languages. They are less useful for functional languages because they cannot guarantee referential transparency, and they appear to be even less useful for concurrent languages: even if we track the effect of a strong update on a shared object on all its aliases, this is only useful if we know *when* the update happens. In an inherently non-deterministic language such as the pi-calculus this is usually hard to know before execution.

Linearity in the pi-calculus Linear types for the pi-calculus were introduced by Kobayashi *et al.* [14] but do not employ any subtyping. Moreover, their system cannot be used as a basis for strong update or channel deallocation; although they split a bidirectional linear ("unique") channel into a linear input channel and a linear output channel (*cf.* Definition 2.3.1 for the type combination operator $(+)$) these parts are never "collected" or "counted". The more refined type splitting operation we use in this paper, combined with the type decrement operation (which has no equivalent in their system) is key to make uniqueness useful for strong updates and deallocation. Our system can easily be extended to incorporate modalities but it does not rely on them; in our case, channel modalities are an orthogonal issue.

Fractional permissions and permission accounting Boyland [6] was one of the first to consider splitting permissions into *fractional* permissions which allow linearity or uniqueness to be temporarily violated. Thus, strong update is possible only with a full permission, whereas only passive access is permitted with a "fraction" of a permission. When all the fractions have been reassembled into one whole permission, strong update is once again possible.

Boyland's suggestion has been taken up by Bornat *et al.* [5], who introduce both fractional permissions and "counting" permissions to separation logic. Despite of the fact that their model

of concurrency is shared-memory, their mechanism of permission splitting and counting is surprisingly similar to our treatment of unique assumptions. However, while their resource reading of semaphores targets *implicit* ownership-transfer, uniqueness typing allow us to reason about explicit ownership-transfer. Moreover, subtyping from unique to unrestricted types provides the flexibility of not counting assumptions whenever this is not required, simplifying reasoning for resources that are not deallocated or strongly updated.

Session types Session types [11] and types with CCS-like usage annotations [12] are used to describe channels which send objects of different types. However, these types give detailed information on how channels are used, which makes modular typing difficult. For example, the *heap* channel used by CLIENT_3 cannot be given a type without knowing all the processes that use the heap.

5   Conclusions and Future Work

We have extended ideas from process calculi, substructural logics and permission counting to define a type system for a the pi-calculus extended with primitives for channel allocation and deallocation, where strong update and channel deallocation is deemed safe for unique channels.

The purpose of our type system is not to ensure that every resource that is allocated will also be deallocated (*i.e.,* the absence of memory leaks). This is difficult to track in a type system. For instance, consider

$$\text{alloc}(x).\big(\ c!d_1.d_1!\langle\rangle.\text{nil} \quad \| \quad c!d_2.\text{nil} \quad \| \quad c?y.y?().\text{free}\,x\ \big)$$

Statically, it is hard to determine whether the third parallel process will eventually execute the $\text{free}\,x$ operation. This is due to the fact that it can non-deterministically react with either the first or second parallel process and, should it react with the second process, it will block at $d_2?().\text{free}\,x$. In order to reject this process as ill-typed, the type-system needs to detect potential deadlocks. This can be done [13], but requires a type system that is considerably more complicated than ours. We leave the responsibility to deallocate to the user, but guarantee that resources once deallocated will no longer be used.

The simplicity of our type-system makes it easily extensible. For instance, one useful extension would be that of input/output modalities, which blend easily with the affine/unique duality. Presently, when a server process splits a channel $c : [\mathbf{T}]^{\bullet}$ into one channel of type $[\mathbf{T}]^{(\bullet,2)}$ and two channels of type $[\mathbf{T}]^1$ to be given to two clients, the clients can potentially use this channel to communicate amongst themselves instead of the server. Modalities are a natural mechanism to preclude this from happening.

We are currently investigating ways how uniqueness types can be used to refine existing equational theories, so as to be able to equate processes such as CLIENT_1 and CLIENT_0. This will probably require us to establish a correspondence between uniqueness at the type level and restriction at the term level.

References

[1] P. M. Achten & M. J. Plasmeijer (1995): *The ins and outs of Clean I/O*. Journal of Functional Programming 5(1), pp. 81–110.

[2] Amal Ahmed, Matthew Fluet & Greg Morrisett (2005): *A Step-Indexed Model of Substructural State*. In: *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, ACM, pp. 78–91.

[3] Joe Armstrong (2007): *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf.

[4]   Erik Barendsen & Sjaak Smetsers (1996): *Uniqueness Typing for Functional Languages with Graph Rewriting Semantics*. Mathematical Structures in Computer Science 6, pp. 579–612.

[5]   Richard Bornat, Cristiano Calcagno, Peter O'Hearn & Matthew Parkinson (2005): *Permission accounting in separation logic*. SIGPLAN Not. 40(1), pp. 259–270.

[6]   John Boyland (2003): *Checking Interference with Fractional Permissions*. In: R. Cousot, editor: *Static Analysis: 10th International Symposium*, Lecture Notes in Computer Science 2694, Springer, pp. 55–72.

[7]   Dave Clarke & Tobias Wrigstad (2003): *External Uniqueness Is Unique Enough*. In: *ECOOP 2003 – Object-Oriented Programming*, Lecture Notes in Computer Science 2743, Springer Berlin / Heidelberg, pp. 59–67.

[8]   Manuel Fähndrich & Robert DeLine (2002): *Adoption and focus: practical linear types for imperative programming*. In: *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, ACM, pp. 13–24.

[9]   Jeffrey S. Foster, Tachio Terauchi & Alex Aiken (2002): *Flow-sensitive type qualifiers*. In: *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, ACM, pp. 1–12.

[10] Dana Harrington (2006): *Uniqueness logic*. Theoretical Computer Science 354(1), pp. 24–41.

[11] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: *ESOP '98: Proceedings of the 7th European Symposium on Programming*, Springer-Verlag, pp. 122–138.

[12] Naoki Kobayashi (2003): *Type Systems for Concurrent Programs*. In: *Formal Methods at the Crossroads: From Panacea to Foundational Support*, LNCS 2757, Springer Berlin / Heidelberg, pp. 439–453.

[13] Naoki Kobayashi (2006): *A New Type System for Deadlock-Free Processes*. In: *CONCUR 2006 Concurrency Theory*, Lecture Notes in Computer Science 4137, Springer Berlin / Heidelberg, pp. 233–247.

[14] Naoki Kobayashi, Benjamin C. Pierce & David N. Turner (1999): *Linearity and the pi-calculus*. ACM Trans. Program. Lang. Syst. 21(5), pp. 914–947.

[15] Robin Milner, Joachim Parrow & David Walker (1992): *A calculus of mobile processes, parts I and II*. Inf. Comput. 100(1), pp. 1–40.

[16] Martin Odersky (1992): *Observers for Linear Types*. In: Bernd Krieg-Brückner, editor: *Proceedings of the 4th European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science 582, Springer-Verlag, pp. 390–407.

[17] François Pottier (2007). *Wandering through linear types, capabilities, and regions*. Survey talk given at INRIA, Rocquencourt, France.

[18] Frederick Smith, David Walker & J. Gregory Morrisett (2000): *Alias Types*. In: *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, Springer-Verlag, pp. 366–381.

[19] David Teller (2004): *Recollecting resources in the pi-calculus*. In: *Proceedings of IFIP TCS 2004*, Kluwer Academic Publishing, pp. 605–618.

[20] Edsko de Vries (2008): *Making Uniqueness Typing Less Unique*. Ph.D. thesis, Trinity College Dublin, Ireland.

[21] Edsko de Vries, Adrian Francalanza & Matthew Hennessy (2009): *Uniqueness Typing for Resource Management in Message-Passing Concurrency—Technical Appendix*. Technical Report, Trinity College Dublin, Ireland. Available from `http://www.cs.tcd.ie/Edsko.de.Vries/pub`.

[22] Philip Wadler (1990): *Linear types can change the world!* In: M. Broy & C. B. Jones, editors: *Proceedings of the IFIP TC2 WG 2.2/2.3 Working Conference on Programming Concepts and Methods*, North-Holland, pp. 561–581.

[23] Philip Wadler (1991): *Is there a use for linear logic?* In: *Proceedings of the 2nd ACM SIGPLAN symposium on Partial Evaluation and semantics-based program manipulation (PEPM)*, ACM, pp. 255–273.

[24] David Walker & J. Gregory Morrisett (2001): *Alias Types for Recursive Data Structures*. In: *TIC '00: Third International Workshop on Types in Compilation*, Springer-Verlag, pp. 177–206.