Journal of Logic and Computation Advance Access published June 6, 2012

Uniqueness typing for resource management in message-passing concurrency

EDSKO DEVRIES, *Computer Science, Department of Trinity College Dublin, College Green, Dublin 2 Ireland.*
E-mail: edsko.de.vries@cs.tcd.ie

ADRIAN FRANCALANZA, *Information and Communication Technology, University of Malta, MSD 2080, Malta.*
E-mail: adrian.francalanza@um.edu.mt

MATTHEW HENNESSY, *Department of Computer Science, Trinity College Dublin, College Green, Dublin 2, Ireland.*
E-mail: matthew.hennessy@cs.tcd.ie

Abstract

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.

Keywords: Message-passing concurrency, type systems, resource management

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 [22] 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 article, 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 resources can be safely deallocated when they are no longer required.

Idiomatic pi-calculus processes are often characterized by wasteful use-once-throw-away channels [21, 22]. Consider the following two pi-calculus process definitions

$$\textsc{timeSrv}(\mathit{getTime}) \triangleq \mathtt{rec}X.\mathit{getTime}?x.x!(\mathit{hr},\mathit{min}).X$$
$$\textsc{dateSrv}(\mathit{getDate}) \triangleq \mathtt{rec}X.\mathit{getDate}?x.x!(\mathit{year},\mathit{mon},\mathit{day}).X$$

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.

doi:10.1093/logcom/exs022

DATESRV is a similar service which returns the current date. An idiomatic pi-calculus client definition is

$$\text{CLIENT}_0 \triangleq \texttt{new}\,ret_1.getTime_1!ret_1 \parallel getTime_2!ret_1 \parallel ret_1?(y_{hr}, y_{min}).ret_1?(y'_{hr}, y'_{min}).\texttt{new}\,ret_2.getDate!ret_2.ret_2?(z_{year}, z_{mon}, z_{day}).P$$

CLIENT_0 first sends a request to two separate time servers in order to obtain an accurate measurement. It does not matter in which order the servers reply, but once they have both replied the client continues to send a request to a date server before continuing as P. It uses two distinct channels ret_1 and ret_2 as return channels to query the time and date servers; 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 for the interactions between this client and the two servers. In particular, since the servers use the reply channel *once* and do not keep any reference to this channel after use, the client can economize on the resources required for these interactions and use *one* channel to communicate sequentially with both the time servers and the date servers.

$$\text{CLIENT}_1 \triangleq \texttt{new}\,ret.getTime_1!ret \parallel getTime_2!ret \parallel ret?(y_{hr}, y_{min}).ret?(y'_{hr}, y'_{min}).\\ getDate!ret.ret?(z_{year}, z_{mon}, z_{day}).P$$

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 a time server and date server in parallel is unsafe:

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

Standard pi-calculus type systems accept only CLIENT_0 and rule out both CLIENT_1 and $\text{CLIENT}_{\text{err}}$. However, CLIENT_1 is safe because, apart from the fact that the time servers use the return channel once, the communication with the date server happens *strictly after* the communication with the time servers.

Adequate resource management also requires precise descriptions of when resources are allocated and existing ones are disposed. The standard scoping construct of the π-calculus $\texttt{new}c.P$ is unfit for this purpose as the extrusion rule means that, for instance, channel ret_2 is allocated before or after the communication with the time server in CLIENT_0. Moreover, the π-calculus does not offer an explicit channel deallocation construct, so that the point of deallocation of channels (garbage collection) is likewise unknown. We address this by introducing an explicit allocation construct $\texttt{alloc}x.P$. When the allocation is executed, a new channel c is created at runtime and the $\texttt{alloc}x.P$ changes to $\texttt{new}c.P\{^{c}/_{x}\}$. Dually, we also extend the calculus with an explicit *deallocation* operator $\texttt{free}c.P$. For example:

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

Inevitably, the added expressiveness of this extended pi-calculus increases the possibilities for runtime errors such as in $\text{CLIENT}_{\text{err2}}$ below. In this client, potential interleavings of the subprocesses $getTime!x.x?(y_{hr}, y_{min}).P_1$ and $getDate!x.x?(z_{year}, z_{mon}, z_{day}).\texttt{free}x.P_2$ can lead to both value mismatch during communication (since the subprocesses are using channel x to communicate values

P, Q	$::= u!\vec{v}.P$	*(output)*	$\mid u?\vec{x}.P$		*(input)*
	$\mid \texttt{nil}$	*(nil)*	$\mid \texttt{if}\ u = v\ \texttt{then}\ P\ \texttt{else}\ Q$		*(match)*
	$\mid \texttt{rec}\, X.P$	*(recursion)*	$\mid X$		*(process variable)*
	$\mid P \parallel Q$	*(parallel)*	$\mid \texttt{new}\, c{:}s.P$		*(stateful scoping)*
	$\mid \texttt{alloc}\, x.P$	*(allocate)*	$\mid \texttt{free}\, u.P$		*(deallocate)*

FIGURE 1. Polyadic resource pi-calculus syntax.

of different types) and also to a premature deallocation of this channel by the second sub-process while it is still in use by the first subprocess.

$$\text{CLIENT}_{\text{err2}} \triangleq \texttt{alloc}x.(\mathit{getTime}!x.x?(y_{hr}, y_{min}).P_1 \parallel \mathit{getDate}!x.x?(z_{year}, z_{mon}, z_{day}).\texttt{free}x.P_2)$$

We define a type system which rejects processes that are unsafe; the type system combines uniqueness typing [5] and affine typing [21], 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 [15], and we make essential use of this duality to keep track of uniqueness across channel-passing parallel processes.

The rest of the article is structured as follows. Section 2 introduces our message-passing language with explicit channel allocation and deallocation. Section 3 describes our type system and outlines how examples are typechecked. Section 4 then details the proof of soundness for our type system. Finally, Section 5 discusses related work and Section 6 concludes with directions for future work.

2 The Resource Pi-Calculus

Figure 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 $c, d \in \text{NAME}$ and variables $x, y \in \text{VAR}$, and lets identifiers $u, v \in \text{NAME} \cup \text{VAR}$ range over both. The input and channel allocation constructs are binders for variables $\vec{x}$ and x *respectively*, whereas scoping is a binder for names (i.e. c). The syntax also assumes a denumerable set of process variables $X, Y \in \text{PVAR}$ which are bound by the recursion construct.

Channels are stateful (allocated, $\top$, or deallocated, $\bot$) and process semantics is defined over systems, $\langle M, P\rangle$ where $M \in \Sigma : \text{CHAN} \rightharpoonup \{\top, \bot\}$ describes the state of the free channels in P, and stateful scoping $\texttt{new}c{:}s.P$ describes the state of scoped channels. A tuple $\langle M, P\rangle$ is a system whenever $\text{fn}(P) \subseteq \text{dom}(M)$ and is denoted as $M \triangleright P$. We say that a system $M \triangleright P$ is closed whenever P does not have any free variables. An example system would be

$$c : \top \triangleright (\texttt{new}d : \top.c!d)$$

whereby, apart from the *visible state* relating to channel c, $M = c : \top$, the system also describes *internal state* through the stateful scoped channel in the process part $P = \texttt{new}d : \top.c!d$

Figure 2 defines contexts over systems where $\mathcal{C}[M \triangleright P]$ denotes the application of a context $\mathcal{C}$ to a system $M \triangleright P$. In the case where a context scopes a name c, the definition extracts the state relating

Contexts

$$\mathcal{C} ::= \quad [-] \quad | \quad \mathcal{C} \parallel P \quad | \quad P \parallel \mathcal{C} \quad | \quad \mathtt{new}\, c.\mathcal{C}$$

$$\begin{array}{rll} [M \triangleright P] & \stackrel{\text{def}}{=} M \triangleright P & \\ \mathcal{C}[M \triangleright P] \parallel Q & \stackrel{\text{def}}{=} M' \triangleright (P' \parallel Q) & \text{if } \mathcal{C}[M \triangleright P] = M' \triangleright P' \\ Q \parallel \mathcal{C}[M \triangleright P] & \stackrel{\text{def}}{=} M' \triangleright (Q \parallel P') & \text{if } \mathcal{C}[M \triangleright P] = M' \triangleright P' \\ \mathtt{new}\, c.\mathcal{C}[M \triangleright P] & \stackrel{\text{def}}{=} M' \triangleright \mathtt{new}\, c{:}s.P' & \text{if } \mathcal{C}[M \triangleright P] = M', c : s \triangleright P' \end{array}$$

Structural equivalence

$$\begin{array}{llll} \textsc{sCom} & P \parallel Q \equiv Q \parallel P & \textsc{sAss} & P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R \\ \textsc{sNil} & P \parallel \mathtt{nil} \equiv P & \textsc{sNm} & \mathtt{new}\, c{:}\bot.\mathtt{nil} \equiv \mathtt{nil} \end{array}$$

$$\begin{array}{lll} \textsc{sSwp} & \mathtt{new}\, c{:}s.(\mathtt{new}\, d{:}s'.P) \equiv \mathtt{new}\, d{:}s'.(\mathtt{new}\, c{:}s.P) & \\ \textsc{sExt} & P \parallel \mathtt{new}\, c{:}s.Q \equiv \mathtt{new}\, c{:}s.(P \parallel Q) & c \notin \mathrm{fn}\, P \end{array}$$

Reduction Rules

$$\frac{M(c) = \top}{M \triangleright c!\vec{d}.P \parallel c?\vec{x}.Q \to M \triangleright P \parallel Q\{\vec{d}/\vec{x}\}}\ \textsc{rCom} \qquad \frac{}{M \triangleright \mathtt{rec}\, X.P \to M \triangleright P\{^{\mathtt{rec}\, X.P}/_X\}}\ \textsc{rRec}$$

$$\frac{}{M \triangleright \mathtt{if}\ c = c\ \mathtt{then}\ P\ \mathtt{else}\ Q \to M \triangleright P}\ \textsc{rThen} \qquad \frac{c \neq d}{M \triangleright \mathtt{if}\ c = d\ \mathtt{then}\ P\ \mathtt{else}\ Q \to M \triangleright Q}\ \textsc{rElse}$$

$$\frac{c \notin \mathrm{dom}(M)}{M \triangleright \mathtt{alloc}\, x.P \to M \triangleright \mathtt{new}\, c{:}\top.P\{c/x\}}\ \textsc{rAll} \qquad \frac{}{M, c{:}\top \triangleright \mathtt{free}\, c.P \to M, c{:}\bot \triangleright P}\ \textsc{rFree}$$

$$\frac{P \equiv P' \qquad M \triangleright P' \to M \triangleright Q' \qquad Q' \equiv Q}{M \triangleright P \to M \triangleright Q}\ \textsc{rStr}$$

Error Reduction Rules

$$\frac{|\vec{d}| \neq |\vec{x}|}{M \triangleright c!\vec{d}.P \parallel c?\vec{x}.Q \xrightarrow{\text{err}}}\ \textsc{eAty} \qquad \frac{M(c) = \bot}{M \triangleright c!\vec{d}.P \xrightarrow{\text{err}}}\ \textsc{eOut} \qquad \frac{M(c) = \bot}{M \triangleright c?\vec{x}.Q \xrightarrow{\text{err}}}\ \textsc{eIn} \qquad \frac{P \equiv Q \quad M \triangleright Q \xrightarrow{\text{err}}}{M \triangleright P \xrightarrow{\text{err}}}\ \textsc{eStr}$$

FIGURE 2. Contexts, reduction rules and error predicates.

to c from M and associates it with the scoping of c. For example,

$$\mathtt{new}c.[(c{:}\top, d{:}\top) \triangleright d!c] = d{:}\top \triangleright (\mathtt{new}c{:}\top.d!c)$$

The reduction relation is defined as the least contextual relation over closed systems satisfying the rules in Figure 2. It relies on a quasi-standard pi-calculus structural equivalence relation over processes ($\equiv$) whereby the only notable difference is that redundant channel scoping can only be discarded when it is deallocated (sNm); this allows us characterize 'memory leakages' during the computation, which take the form of $\mathtt{new}c{:}\top.\mathtt{nil}$. System communication (rCom) requires the communicating channel to be allocated but does not place any constraint on the status of the channels communicated. Allocation (rAll) creates a private allocated channel and substitutes it for the bound variable of the allocation construct in the continuation; the condition $c \notin \mathrm{dom}(M)$ ensures that c is

fresh in P. Deallocation (rFree) is the only construct that changes the visible state of a system, M; through the scoping contextual rule we can then also express changes in the internal state of a system. We can, therefore, express computations such as

$$\begin{aligned} M \triangleright \texttt{alloc}x.(x!c.\texttt{nil} \parallel x?y.\texttt{free}x.P) &\rightarrow M \triangleright \texttt{new}d{:}\top.(d!c.\texttt{nil} \parallel d?y.\texttt{free}c.P) \qquad x \notin \mathrm{fv}(P) \\ &\rightarrow M \triangleright \texttt{new}d{:}\top.(\texttt{free}d.P\{^{\mathrm{c}}/_{y}\}) \\ &\rightarrow M \triangleright \texttt{new}d{:}\bot.(P\{^{\mathrm{c}}/_{y}\}) \equiv M \triangleright P\{^{\mathrm{c}}/_{y}\} \end{aligned}$$

Figure 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 the unconstrained use of strong updates such as in the case of $\text{CLIENT}_{\text{err}}$ in Section 1.

Example 2.1
For $Q = \text{TIMESRV} \parallel \text{DATESRV}$, whenever $M(\mathit{getDate}) = M(\mathit{getTime}) = \top$ we have the following successful reduction sequences:

$$M \triangleright \text{CLIENT}_2 \parallel Q \rightarrow^* M \triangleright P\{^{\text{hr, min}}/_{y_{\text{hr}}, y_{\text{min}}}\}\{^{\text{year, mon, day}}/_{z_{\text{year}}, z_{\text{mon}}, z_{\text{day}}}\} \parallel Q$$
$$M \triangleright \text{CLIENT}_{\text{err2}} \parallel Q \rightarrow^* M \triangleright P_1\{^{\text{hr, min}}/_{y_{\text{hr}}, y_{\text{min}}}\} \parallel P_2\{^{\text{year, mon,day}}/_{z_{\text{year}}, z_{\text{mon}}, z_{\text{day}}}\} \parallel Q$$

However, in the case of $\text{CLIENT}_{\text{err2}}$, we can also have premature deallocation errors—the second subprocess $\mathit{getDate}!x.x?(z_{\text{year}}, z_{\text{mon}}, z_{\text{day}}).\texttt{free}x.P_2$ may deallocate the shared allocated channel, c below, while the right sub-process is still using it.

$$M \triangleright \text{CLIENT}_{\text{err2}} \parallel Q \rightarrow^*$$
$$M \triangleright \texttt{new}c{:}\bot.(c?(y_{\text{hr}}, y_{\text{min}}).P_1 \parallel c!(\text{hr, min}).\text{TIMESRV} \parallel P_2\{^{\text{year, mon, day}}/_{z_{\text{year}}, z_{\text{mon}}, z_{\text{day}}}\}) \parallel \text{DATESRV} \xrightarrow{\text{err}}$$

We can also have arity mismatch errors caused by communication interference on the shared allocated channel c:

$$M \triangleright \text{CLIENT}_{\text{err2}} \parallel Q \rightarrow^* M \triangleright \texttt{new}c{:}\top.\begin{pmatrix} c?(y_{\text{hr}}, y_{\text{min}}).P_1 \parallel c!(\text{year, mon, day}).\text{DATESRV} \\ \parallel c?(z_{\text{year}}, z_{\text{mon}}, z_{\text{day}}).\texttt{free}c.P_2 \parallel c!(\text{hr, min}).\text{TIMESRV} \end{pmatrix} \xrightarrow{\text{err}}$$

Strong updates, as in the case of CLIENT_2, should only be allowed when there is no other processes still using the channel. ■

Example 2.2
Client CLIENT_3 demonstrates channel reuse *across* processes. Rather than allocating a new channel, CLIENT_3 requests a *currently unused* channel from a heap of channels and returns the channel to the heap when it no longer needs it. A heap, in this case, is a designated channel *heap*.

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

This allows us to have multiple replicas of the clients while keeping the cost of the server interactions (in terms of additional channels required) *constant*. Of course, in the arrangement below, client–server interactions are sequentialized for each client, depending on who currently has access to channel c

held on *heap*. But this is necessary so as not to have any interference during the multiple reuses of channel c.

$$M \triangleright \text{CLIENT}_3 \parallel \text{CLIENT}_3 \parallel \ldots \parallel \text{CLIENT}_3 \parallel \texttt{new}c{:}\top.(\mathit{heap}!c) \parallel \text{TIMESRV} \parallel \text{DATESRV} \tag{1}$$

It is instructive to run through an execution of this system and analyse how access to channel c evolves throughout computation, as it turns out that *exclusive access* to such a channel is what enables a process to safely performs operations such as deallocations and strong updates. For instance in (1), after a client inputs on *heap*, it obtains access to c; as standard in the pi-calculus, this is denoted by the scope extrusion of channel c to the client and the system in (1) reduces to

$$M \triangleright \text{CLIENT}_3 \parallel \text{CLIENT}_3 \parallel \ldots \parallel \texttt{new}c{:}\top.(\text{CLIENT}_3') \parallel \text{TIMESRV} \parallel \text{DATESRV} \tag{2}$$

At that point, only CLIENT_3' is in a position to interact with the servers, while the other clients are blocked waiting for the next output on channel *heap*. After CLIENT_3' queries TIMESRV the scope of c is extruded again, this time to TIMESRV, and from (2) we obtain

$$M \triangleright \text{CLIENT}_3 \parallel \text{CLIENT}_3 \parallel \ldots \parallel \texttt{new}c{:}\top.(\text{CLIENT}_3'' \parallel \text{TIMESRV}') \parallel \text{DATESRV} \tag{3}$$

After the interaction with TIMESRV terminates the scope of c can be tightened again to

$$M \triangleright \text{CLIENT}_3 \parallel \text{CLIENT}_3 \parallel \ldots \parallel \texttt{new}c{:}\top.(\text{CLIENT}_3''') \parallel \text{TIMESRV} \parallel \text{DATESRV} \tag{4}$$

In similar fashion the scope of c extrudes to DATESRV and then contracts back again to the client so as to reach the following system from (4).

$$M \triangleright \text{CLIENT}_3 \parallel \text{CLIENT}_3 \parallel \ldots \parallel \texttt{new}c{:}\top.(\mathit{heap}!c.P') \parallel \text{TIMESRV} \parallel \text{DATESRV} \tag{5}$$

At this point, the client $\mathit{heap}!c.P'$ explicitly transfers exclusive access to channel c to some other client.

$$M \triangleright \text{CLIENT}_3 \parallel \texttt{new}c{:}\top.(\text{CLIENT}_3') \parallel \ldots \parallel P' \parallel \text{TIMESRV} \parallel \text{DATESRV} \tag{6}$$

The following variants of client CLIENT_3' are unsafe and lead to errors in the above computational sequence:

$$\text{CLIENT}_3^{\text{err1}} \triangleq \mathit{heap}?x.\mathit{getTime}!x.x?y_{hr}, y_{min}.\mathit{getDate}!x.x?z_{year}, z_{mon}, z_{day}.\mathit{heap}!x.x!.P$$
$$\text{CLIENT}_3^{\text{err2}} \triangleq \mathit{heap}?x.\mathit{getTime}!x.x?y_{hr}, y_{min}.\mathit{getDate}!x.x?z_{year}, z_{mon}, z_{day}.\texttt{free}x.\mathit{heap}!x.P$$

The erroneous client $\text{CLIENT}_3^{\text{err1}}$ highlights the fact that, for the above system to work as intended, it is crucial that the client transfers exclusive access to channel c on *heap*. If, instead, the client attempts to use this channel after transfer, it will lead to an interference that can result in a runtime error as shown earlier in Example 2.1. Although the communication of deallocated channels is permitted in resource pi-calculus, the second erroneous client $\text{CLIENT}_3^{\text{err2}}$ shows how this can indirectly lead to a premature deallocation when this deallocated channel is then transferred to other clients.

3 Type system

We define a type system to statically approximate the class of safe resource pi-calculus systems. The type-level concept that corresponds to 'exclusive access' is *uniqueness*: when a process is typed in a typing environment in which a channel has a unique type, then that process is guaranteed to have exclusive access to that channel. Strong update and deallocation are therefore safe for unique channels.

The type system is an adaptation of uniqueness type systems for lambda calculi [5, 9], but unlike the latter it allows uniqueness to be temporarily violated. As we saw in Example 2.1, a client might have exclusive access to a channel, then use this channel to communicate with a server, and then *regain* exclusive access to this channel, provided that the server does not use the channel anymore.

We must, therefore, generalize uniqueness typing to introduce a type that records that a channel may not be unique now, but it will be after some number i of communication steps ('unique-after-i'); moreover, we need to combine uniqueness typing with *affine* types to limit how often a channel can be used (for instance, we might limit the time server to use a channel at most once).

3.1 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 describing the values carried over the channel, $\vec{\mathbf{T}}$, and an *attribute* a which can take one of three forms:

- A channel of type $[\,\overrightarrow{\mathbf{T}}\,]^\omega$ is an *unrestricted* channel: it can be used arbitrary often, and provides no information on how many other processes have access to the channel. Such type assumptions correspond to type assumptions of the form $[\,\overrightarrow{\mathbf{T}}\,]$ in standard (non-substructural) type systems.
- A channel of type $[\,\overrightarrow{\mathbf{T}}\,]^1$ is *affine*, and comes with an obligation: it can be used at most once.
- A channel of type $[\,\overrightarrow{\mathbf{T}}\,]^{(\bullet,i)}$ is *unique-after-i* communication actions, comes with a guarantee: a process typed using this assumption will have exclusive access to the channel after i communication actions. We abbreviate the type $[\,\overrightarrow{\mathbf{T}}\,]^{(\bullet,0)}$ of channels that are unique *now* to $[\,\overrightarrow{\mathbf{T}}\,]^\bullet$ and refer to it as simply a *unique channel*.

We give a number of examples illustrating the use of these types with reference to the examples from Section 2.

(1) Unrestricted channels can be used to describe channels such as *getTime* and *getDate* on which replicated servers receive requests.
(2) Affine types can be used to limit how often servers can use a return channel sent to them by a client. For instance, we can give the type $[[\vec{\mathbf{T}}]^1]^\omega$ to *getTime* and *getDate*.
(3) A channel which has just been allocated will have a unique type. We can also use uniqueness to ensure that the heap really carries channels that are unused, by giving the heap channel the type $[[\vec{\mathbf{T}}]^\bullet]^\omega$.
(4) A channel which is unique-after-1 can be used by the client to record that after it has communicated with a server it will once again recover exclusive access to the channel.

3.2 Typing environments

Processes will be typed under a typing environment Γ, which is a multiset of pairs of identifiers and types. Since the type system is substructural, typing assumptions can be used only once in a typing

Type Structure

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

Type Splitting

$$\frac{}{[\vec{\mathbf{T}}]^{\omega} = [\vec{\mathbf{T}}]^{\omega} \circ [\vec{\mathbf{T}}]^{\omega}}\ \textsc{pUnr} \qquad \frac{}{\mathbf{proc} = \mathbf{proc} \circ \mathbf{proc}}\ \textsc{pProc} \qquad \frac{}{[\vec{\mathbf{T}}]^{(\bullet,i)} = [\vec{\mathbf{T}}]^{1} \circ [\vec{\mathbf{T}}]^{(\bullet,i+1)}}\ \textsc{pUnq}$$

Subtyping

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

Structural rules ($\prec$) is the least reflexive transitive relation satisfying

$$\frac{\mathbf{T} = \mathbf{T}_1 \circ \mathbf{T}_2}{\Gamma, u : \mathbf{T} \prec \Gamma, u : \mathbf{T}_1, u : \mathbf{T}_2}\ \textsc{tCon} \qquad \frac{\mathbf{T} = \mathbf{T}_1 \circ \mathbf{T}_2}{\Gamma, u : \mathbf{T}_1, u : \mathbf{T}_2 \prec \Gamma, u : \mathbf{T}}\ \textsc{tJoin}$$

$$\frac{}{\Gamma, u : \mathbf{T} \prec \Gamma}\ \textsc{tWeak} \qquad \frac{\mathbf{T}_1 <: \mathbf{T}_2}{\Gamma, u : \mathbf{T}_1 \prec \Gamma, u : \mathbf{T}_2}\ \textsc{tSub} \qquad \frac{}{\Gamma, u : [\vec{\mathbf{T}}_1]^{\bullet} \prec \Gamma, u : [\vec{\mathbf{T}}_2]^{\bullet}}\ \textsc{tRev}$$

FIGURE 3. Type language and operations.

derivation. For instance, when typing two parallel processes under some environment Γ, assumptions in Γ can be used to type the left process or the right, but never both.

However, we define a number of *structural rules*, shown in Figure 3, which allow to manipulate typing environments. The simplest is *weakening* (TWEAK), which allows to disregard typing assumptions (the type system does not guarantee that a channel *will* be used). We discuss the other rules separately, below.

3.2.1 Splitting and joining

Although typing assumptions can be used only once, typing assumptions can be *split* (logically, contracted) under certain conditions (TCON). We write $\mathbf{T} = \mathbf{T}_1 \circ \mathbf{T}_2$ if an assumption $x : \mathbf{T}$ can be split as two assumptions $x : \mathbf{T}_1$ and $x : \mathbf{T}_2$.

For instance, after allocation CLIENT$_2$ is allowed to split the unique assumption $x : [\vec{\mathbf{T}}]^{\bullet}$ as two separate assumptions $x : [\vec{\mathbf{T}}]^{(\bullet,1)}$ and $x : [\vec{\mathbf{T}}]^{1}$. This allows the client to send x to the server at type $[\vec{\mathbf{T}}]^{1}$ while retaining the assumption $x : [\vec{\mathbf{T}}]^{(\bullet,1)}$ for its own use.

More generally, an assumption $c : [\vec{\mathbf{T}}]^{(\bullet,i)}$ can be split as two assumptions $c : [\vec{\mathbf{T}}]^{(\bullet,i+1)}$ and $c : [\vec{\mathbf{T}}]^{1}$. Unrestricted assumptions can be split as two unrestricted channels, so that unrestricted channels can be used arbitrary often. Finally, affine assumptions cannot be split at all, so that affine channels can be used only once.

Joining (TJOIN) is simply the dual of splitting; it is not strictly necessary to define the type system, but makes the technical development a lot cleaner.

3.2.2 Revision

After the client from Example 2.1 communicates with the time server over channel x, it regains exclusive access to that channel. It is 'therefore' safe to the client to reuse the same channel to communicate with the date server, even though the date server sends values of a different type. This *strong update* is safe only for unique channels, and is embodied by rule tRev.

3.2.3 Subtyping

Since uniqueness is a guarantee that a channel c will be unique after a number of uses, it is safe to weaken this guarantee: a channel at type $[\vec{\mathbf{T}}]^{(\bullet,u)}$ can also be used at type $[\vec{\mathbf{T}}]^{(\bullet,u+1)}$ or even at $[\vec{\mathbf{T}}]^{\omega}$. For example, after getting a unique channel from the heap a process might choose to disregard the uniqueness information about this channel and use it in an unrestricted fashion instead.

Conversely, affinity is an obligation forcing an upper bound on the number of uses. Thus, it is also safe to regard a unique channel or an unrestricted channel as an affine channel. For instance, a client might send an unrestricted channel to a server expecting an affine channel.

We therefore have the following subtyping chain (tSub):[1]

$$[\vec{\mathbf{T}}]^{\bullet} \prec_s [\vec{\mathbf{T}}]^{(\bullet,1)} \prec_s [\vec{\mathbf{T}}]^{(\bullet,2)} \prec_s \cdots [\vec{\mathbf{T}}]^{\omega} \prec_s [\vec{\mathbf{T}}]^{1}$$

Through a combination of subtyping and splitting, we can split a unique-after-i assumption into two unrestricted assumptions, $[\vec{\mathbf{T}}]^{(\bullet,i)} \prec_s [\vec{\mathbf{T}}]^{\omega} = [\vec{\mathbf{T}}]^{\omega} \circ [\vec{\mathbf{T}}]^{\omega}$, and an unrestricted assumption into an unrestricted assumption and an affine assumption, $[\vec{\mathbf{T}}]^{\omega} = [\vec{\mathbf{T}}]^{\omega} \circ ([\vec{\mathbf{T}}]^{\omega} \prec_s [\vec{\mathbf{T}}]^{1})$.

3.2.4 Consistency

The type systems allow for compositional reasoning with respect to uniqueness (we prove formally in Section 4.2). For instance, the clients in Examples 2.1 and 2.2 can be shown to be to be safe using only the typing assumptions assigned to them, and without requiring the full code relating to the servers they interact with.

Such compositional reasoning is safe only if the total type assumptions used to analyse the entire system are consistent among themselves. For example, an environment of the form

$$c : [\mathbf{T}]^{\bullet}, c : [\mathbf{T}]^{1}$$

allows to type a system in which one process deallocates channel c while another, parallel, process attempts to communicate on it. This is clearly unsound.

Clearly an environment with at most one assumption per channel (a partial function) must be consistent, but we have already seen that we sometimes need to split assumptions. We *define* an environment to be consistent if it can be obtained from a partial function by applying any of the structural rules we just defined:

Definition 3.1 (Consistency)
A typing environment Γ is consistent if there exists a partial function Γ' such that $\Gamma' \preceq \Gamma$.

[1] Since we do not consider channel input/output-modalities in this article, channels are invariant with respect to their object types, $\vec{\mathbf{T}}$.

This definition is justified by the soundness theorem, which says that if a system can be typed under a consistent environment, it will not produce any runtime errors. For now, observe that the structural rules have been carefully defined so that if a consistent environment contains an assumption $c : [\vec{\mathbf{T}}]^{(\bullet, i)}$ then it cannot contain any unrestricted assumptions about c, and the number of affine assumptions about c must be less than or equal to i (Lemma 4.1).

3.3 The typing relation

The typing relation over (open and closed) processes is the least relation defined by the rules in Figure 4. It takes the usual shape, $\Gamma \vdash P$, which reads as 'P is well-typed under the typing environment Γ'. It is extended to systems so that $\Gamma \vdash M \triangleright P$ is well-typed if Γ types P and Γ only contains assumptions about channels that are allocated in M.

We do not *a priori* make the assumption that the type environment is consistent (Section 3.5), but soundness of the type system will only be stated with respect to consistent environments.

The rule for parallel (TPAR) requires to divide the assumptions in the typing environment between the left and right process. Restriction introduces a new typing assumption into the typing environment, provided that the channel is allocated (TRST1, TRST2). As expected, allocation introduces unique channels (TALL), and only unique channels can be deallocated (TFREE). We discuss the remaining rules separately, below.

3.3.1 Input and output

In the client–server example, after the client has communicated with the server over channel x, the server is no longer allowed to use x, and the client is guaranteed that it once again has exclusive access to the channel. From a typing perspective, the server had an affine permission $x : [\vec{\mathbf{T}}]^1$ and lost that permission after the communication; the client had a permission $x : [\vec{\mathbf{T}}]^{(\bullet, 1)}$ which became $x : [\vec{\mathbf{T}}]^{\bullet}$ after the communication.

This is expressed in the type system as an operation $\Gamma, c : [\vec{\mathbf{T}}]^{a-1}$ on typing environments (defined in Figure 4); it computes the typing environment after a communication action on c: affine permission are removed, unique-after-$(i+1)$ permissions become unique-after-i and unrestricted permissions are unaffected.

This operation is used in the rules for input and output (TOUT and TIN). Moreover, TOUT requires separate typing assumptions for each of the channels that are sent. The attributes on these channels are not decremented, because no communication 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.

3.3.2 Conditionals

As is standard in substructural (and standard) type systems, the rule for conditions RIF types both branches under the same typing environment, as only one will be executed. It also requires that the process must have a typing assumption for the two channels it is comparing.

We claimed that after a communication with the server over channel x the client regains exclusive access to that channel. At the type level, the server looses its affine permission for x after the communication, as we saw above. The side condition on the rule for conditionals implies that this

Logical rules

$$\frac{\Gamma, u : [\vec{\mathbf{T}}]^{a-1}, \overrightarrow{x : \mathbf{T}} \vdash P}{\Gamma, u : [\vec{\mathbf{T}}]^{a} \vdash u?\vec{x}.P}\ \textsc{tIn} \qquad \frac{\Gamma, u : [\vec{\mathbf{T}}]^{a-1} \vdash P}{\Gamma, u : [\vec{\mathbf{T}}]^{a}, \overrightarrow{v : \mathbf{T}} \vdash u!\vec{v}.P}\ \textsc{tOut} \qquad \frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1, \Gamma_2 \vdash P \parallel Q}\ \textsc{tPar}$$

$$\frac{u, v \in \Gamma \qquad \Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash \mathtt{if}\ u = v\ \mathtt{then}\ P\ \mathtt{else}\ Q}\ \textsc{tIf} \qquad \frac{\Gamma^{\omega}, X : \mathbf{proc} \vdash P}{\Gamma^{\omega} \vdash \mathtt{rec}\, X.P}\ \textsc{tRec} \qquad \frac{}{X : \mathbf{proc} \vdash X}\ \textsc{tVar}$$

$$\frac{\Gamma, x : [\vec{\mathbf{T}}]^{\bullet} \vdash P}{\Gamma \vdash \mathtt{alloc}\, x.P}\ \textsc{tAll} \qquad \frac{\Gamma \vdash P}{\Gamma, u : [\vec{\mathbf{T}}]^{\bullet} \vdash \mathtt{free}\, u.P}\ \textsc{tFree} \qquad \frac{}{\emptyset \vdash \mathtt{nil}}\ \textsc{tNil}$$

$$\frac{\Gamma, c : \mathbf{T} \vdash P}{\Gamma \vdash \mathtt{new}\, c{:}\top.P}\ \textsc{tRst1} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash \mathtt{new}\, c{:}\bot.P}\ \textsc{tRst2} \qquad \frac{\Gamma' \vdash P \qquad \Gamma \prec \Gamma'}{\Gamma \vdash P}\ \textsc{tStr}$$

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

Channel Usage

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

Typing systems

$$\frac{\Gamma \vdash P \qquad \mathrm{dom}(\Gamma) \subseteq \mathrm{alloc}(M)}{\Gamma \vdash M \triangleright P}\ \textsc{tSys}$$

FIGURE 4. Typing rules.

means that the server cannot even use x in a comparison anymore, thus guaranteeing the client true exclusive access to the channel.

3.3.3 Recursion

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 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 $\mathit{heap} : [[\mathbf{T}]^{\bullet}]^{\omega}$:

$$\textsc{infHeap} \triangleq \mathtt{rec}X.\mathtt{alloc}x.\mathit{heap}!x.X$$

3.4 Examples

The systems $\textsc{client}_i \parallel \textsc{timeSrv} \parallel \textsc{dateSrv}$ for $i \in \{0, 1, 2\}$ can all be typed in our type system, whereas $\textsc{client}_{\text{err}}$ is rejected because type splitting enforces a common object type (cf. pUNR, pUNQ in Figure 4.) The derivation below outlines how $\textsc{client}_2$ from the introduction is typed, where we recall

that x is not free in P. It assumes an environment $\Gamma = \mathit{getTime} : [[\mathbf{T}_1, \mathbf{T}_2]^1]^\omega, \mathit{getDate} : [[\mathbf{T}_3, \mathbf{T}_4, \mathbf{T}_5]^1]^\omega$ and proceeds as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x : [\mathbf{T}_3, \mathbf{T}_4, \mathbf{T}_5]^\bullet, y_{hr} : \mathbf{T}_1, y_{min} : \mathbf{T}_2 \vdash \mathit{getDate}!x.x?(z_{year}, z_{mon}, z_{day}).\texttt{free}x.P}\ \textsc{tStr}}{\Gamma, x : [\mathbf{T}_1, \mathbf{T}_2]^\bullet, y_{hr} : \mathbf{T}_1, y_{min} : \mathbf{T}_2 \vdash \mathit{getDate}!x.x?(z_{year}, z_{mon}, z_{day}).\texttt{free}x.P}\ \textsc{tIn}}{\Gamma, x : [\mathbf{T}_1, \mathbf{T}_2]^{(\bullet,1)} \vdash x?(y_{hr}, y_{min}).\mathit{getDate}!x.x?(z_{year}, z_{mon}, z_{day}).\texttt{free}x.P}\ \textsc{tOut}}{\Gamma, x : [\mathbf{T}_1, \mathbf{T}_2]^{(\bullet,1)}, x : [\mathbf{T}_1, \mathbf{T}_2]^1 \vdash \mathit{getTime}!x.x?(y_{hr}, y_{min}).\mathit{getDate}!x.x?(z_{year}, z_{mon}, z_{day}).\texttt{free}x.P}\ \textsc{tStr}}{\Gamma, x : [\mathbf{T}_1, \mathbf{T}_2]^\bullet \vdash \mathit{getTime}!x.x?(y_{hr}, y_{min}).\mathit{getDate}!x.x?(z_{year}, z_{mon}, z_{day}).\texttt{free}x.P}\ \textsc{tAll}}{\Gamma \vdash \texttt{alloc}x.\mathit{getTime}!x.x?(y_{hr}, y_{min}).\mathit{getDate}!x.x?(z_{year}, z_{mon}, z_{day}).\texttt{free}x.P}$$

Rule tAll assigns the unique type $[\mathbf{T}_1, \mathbf{T}_2]^\bullet$ to variable x. Using tStr with tCon, this unique assumption is split in two using pUnq. Rule tOut uses the affine assumption for x for the output argument and the unique-after-1 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 the environment through tStr to change the object type of x from pairs of values (for time) to triples (for dates). The pattern of applying tStr with tCon, tOut and tIn repeats, at which point x is unique again and can be safely deallocated by tFree.

Through the type environment Γ below, we can type system (1) from Example 2.2 as well.

$$\Gamma = \mathit{getTime} : [[\mathbf{T}_1, \mathbf{T}_2]^1]^\omega, \mathit{getDate} : [[\mathbf{T}_3, \mathbf{T}_4, \mathbf{T}_5]^1]^\omega, \mathit{heap} : [[\vec{\mathbf{T}}]^\bullet]^\omega$$

Since the type for channel *heap* is unrestricted, we can split this assumption for every occurrence of $\textsc{client}_3$ in (1). When typechecking each instance of this client, we obtain a unique channel assumption as soon as we apply tIn for the input on channel *heap*. The rest of the type derivation then proceeds in similar fashion to that of $\textsc{client}_2$ discussed above. It is, however, instructive to see why $\textsc{client}_3^{\text{err1}}$ and $\textsc{client}_3^{\text{err2}}$, also from Example 2.2, cannot be typed. Through the application of tStr with tCon, tOut and tIn we reach the two type judgements below for $\textsc{client}_3^{\text{err1}}$ and $\textsc{client}_3^{\text{err2}}$, respectively:

$$\Gamma, x : [\mathbf{T}_3, \mathbf{T}_4, \mathbf{T}_5]^\bullet, y_{hr} : \mathbf{T}_1, y_{min} : \mathbf{T}_2, z_{year} : \mathbf{T}_3, z_{mon} : \mathbf{T}_4, z_{day} : \mathbf{T}_5 \vdash \mathit{heap}!x.x!.P \tag{7}$$

$$\Gamma, x : [\mathbf{T}_3, \mathbf{T}_4, \mathbf{T}_5]^\bullet, y_{hr} : \mathbf{T}_1, y_{min} : \mathbf{T}_2, z_{year} : \mathbf{T}_3, z_{mon} : \mathbf{T}_4, z_{day} : \mathbf{T}_5 \vdash \texttt{free}x.\mathit{heap}!x.P \tag{8}$$

In the case of type derivation (7), even though we can revise the object type of the type assumption for x from $\mathbf{T}_3, \mathbf{T}_4, \mathbf{T}_5$ to $\vec{\mathbf{T}}$, so as to enable the typing of the output of x on channel *heap*, this would leave us with no remaining type assumptions for channel x, precluding the typing of the remaining process $x!.P$. Similarly, in the case of type derivation (8), typing the deallocation of x would consume all the type assumptions relating to x, which rules out any possible typing for the remaining process $\mathit{heap}!x.P$.

Finally, system (1) in Example 2.2 can be safely extended with processes such as $\textsc{client}_4$ which uses unique channels obtained from the heap in unrestricted fashion. Our type system accepts $\textsc{client}_4$ by applying *subtyping* from unique to unrestricted on the channel x obtained from *heap*.

$$\textsc{client}_4 \triangleq \mathit{heap}?x.\texttt{rec}X.(\mathit{getTime}!x.x?(y_{hr}, y_{min}).P \parallel X)$$

3.5 (In)consistency

Due to our substructural treatment of assumption, when two processes both need a typing assumption relating to the same channel c, TPAR from Figure 4 forces us to have two separate assumptions in the typing environment. As we discussed already, these two assumptions need not be identical and can be derived from a single assumption; for example, an assumption $c : []^{\bullet}$ can be split as two assumptions

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

As mentioned, soundness of the type system is stated only with respect to *consistent* environments. However, in the definition of the typing relation we allow processes to be typed under arbitrary (possibly inconsistent) environments. In this section, we explain why we cannot restrict the typing relation to consistent environments.

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

$$\dfrac{\dfrac{}{a : [[]^{1}]^{\omega}, u : []^{1} \vdash a!u}\ \textsc{tOut} \qquad \dfrac{\dfrac{\dfrac{}{a : [[]^{1}]^{\omega}, u : []^{\bullet}, x : []^{1} \vdash \texttt{free}u \parallel a!x}\ \textsc{tIn}}{a : [[]^{1}]^{\omega}, u : []^{(\bullet,1)}, x : []^{1} \vdash u?().\texttt{free}u \parallel a!x}\ \textsc{tIn}}{a : [[]^{1}]^{\omega}, u : []^{(\bullet,1)} \vdash a?x.u?().\texttt{free}u \parallel a!x}\ \textsc{tPar}}{\dfrac{a : [[]^{1}]^{\omega}, a : [[]^{1}]^{\omega}, u : []^{1}, u : []^{(\bullet,1)} \vdash a!u \parallel a?x.u?().\texttt{free}u \parallel a!x}{a : [[]^{1}]^{\omega}, u : []^{\bullet} \vdash a!u \parallel a?x.u?().\texttt{free}u \parallel a!x}\ \textsc{tStr (twice)}}$$

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:

$$\dfrac{\dfrac{\dfrac{}{u : []^{\bullet}, u : []^{1} \vdash \texttt{free}u \parallel a!u}\ \textsc{tIn}}{u : []^{(\bullet,1)}, u : []^{1} \vdash u?().\texttt{free}u \parallel a!u}\ \textsc{tCon}}{u : []^{\bullet} \vdash u?().\texttt{free}u \parallel a!u}$$

The continuation of this process looks suspicious as it attempts to free u while simultaneously sending it on a. Indeed, $\texttt{free}u \parallel a!u$ 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 continuation 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

two processes communicate, *both* will start executing their continuations. It is an important property of our typing environments that if $\Gamma, c:[\vec{\mathbf{T}}]^{a}, c:[\vec{\mathbf{T}}]^{a'}$ is consistent, then

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

will also be consistent (Lemma 4.1). This is crucial when proving subject reduction, which says that if a system is typeable in a consistent environment, it will remain typeable in a consistent environment.

Note that allowing for inconsistent environments during type checking is not unusual for these kinds of type systems; for instance, in the session type system with subtyping of Gay and Hole a very similar issue arises [13, Section 5.2] (they use the word 'unbalanced' instead of inconsistent).

4 Soundness

We show that our type system is sound: well-typed programs do not go wrong. The presentation of this proof proceeds as follows: Section 4.1 outlines some preliminary properties of consistent environments and the typing relation, and states the necessary substitution lemmas; Section 4.2 details the subterm typing lemma, which turns out to be the most involved aspect of the soundness proof; Section 4.3 finally states and proves the standard safety and subject reduction theorems: well-typed programs do not have runtime errors and remain well-typed when they reduce.

4.1 Preliminaries

We first state some properties of consistent environments.

LEMMA 4.1 (Consistent environments)
Let Γ be a consistent environment. Then:

(1) If $\Gamma = \Gamma', c:[\vec{\mathbf{T}}]^{a}, c:[\vec{\mathbf{T}}']^{a'}$ then $\vec{\mathbf{T}} = \vec{\mathbf{T}}'$.
(2) If $\Gamma = \Gamma', c:[\overrightarrow{\mathbf{T}}]^{\bullet}$ then $c \notin \mathrm{dom}(\Gamma')$.
(3) If $\Gamma \preceq \Gamma', c:[\overrightarrow{\mathbf{T}}]^{\bullet}$ then $\exists \Gamma''$ such that $\Gamma = \Gamma'', c:[\overrightarrow{\mathbf{T}'}]^{\bullet}$ and $\Gamma'' \preceq \Gamma'$.
(4) If $c \in \mathrm{dom}(\Gamma)$ and $\Gamma' \preceq \Gamma$ then $c \in \mathrm{dom}(\Gamma')$.
(5) If $\Gamma = \Gamma_1, \Gamma_2$ then both Γ_1 and Γ_2 are consistent.
(6) If Γ contains only unrestricted assumptions then $\Gamma \preceq \Gamma, \Gamma$.
(7) If $\Gamma = \Gamma', u:[\vec{T}]^{a_1}, u:[\vec{T}]^{a_2}$ then

$$\Gamma', u:[\vec{T}]^{a_1}, u:[\vec{T}]^{a_2} \preceq \Gamma', u:[\vec{T}]^{a_1-1}, u:[\vec{T}]^{a_2-1}$$

PROOF. Properties (1)–(6) are easily verified. For the proof of (7), we do case analysis on a_1 and a_2. If a_1, a_2 are affine or unrestricted, the lemma follows from weakening (unrestricted assumptions are not affected by the decrement operation). The only interesting case is where $a_1 = (\bullet, i)$ and $a_2 = 1$ (or vice versa), in which case it follows from TJOIN. ■

One of the advantages of phrasing the structural rules of the type system using an auxiliary transitive relation ($\preceq$) is that we can conveniently state inversion principles. For example,

LEMMA 4.2 (Inversion for output)

If $\Gamma \vdash c!\vec{d}.P$ then

$$\Gamma \preceq \Gamma', u : [\vec{T}]^a, \overrightarrow{d : T}$$

and $\Gamma', u : [\vec{T}]^{a-1} \vdash P$.

The inversion lemmas for the other constructs are similar.

As usual, we need two substitution lemmas; however, the typing rules have been set up in such a way that these are easy to prove.

LEMMA 4.3 (Process substitution)
If $\Gamma_1, X : \mathbf{proc} \vdash P$ and $\Gamma_2^{\omega} \vdash Q$ then $\Gamma_1, \Gamma_2^{\omega} \vdash P\{^{Q}/_{X}\}$.

PROOF. By induction on P followed by inversion on the typing relation. The only tricky case is the case for parallel, where we might duplicate the process. However, since the process is typed under an environment containing only unrestricted assumptions, this follows from Lemma 4.1(6). ■

LEMMA 4.4 (Identifier substitution)
If $\Gamma, \overrightarrow{x : T} \vdash P$, where the $\vec{x}$ are pairwise disjoint and do not occur in the domain of Γ, then $\Gamma, \overrightarrow{u : T} \vdash Q\{\vec{u}/\vec{x}\}$.

PROOF. This is a simple renaming of variables throughout the typing derivation. ■

LEMMA 4.5 (Preservation of types under structural equivalence)
If $\Gamma \vdash P$ and $P \equiv P'$ then $\Gamma \vdash P'$.

PROOF. Since the typing relation is not sensitive to the order of the assumptions in the typing environment, reordering parallel processes and extrusion do not affect typing. Removing or adding nil processes and adding or removing a restriction around the `nil` process do not affect typing because `nil` can be typed in any environment. Finally, alpha-renaming bound names does not affect typing because those bound names are not in the (original) typing environment. ■

4.2 Subterm typing lemma

The preservation theorem (Section 4.3) states that if a system $M \triangleright P$ is typed under some environment Γ, and $M \triangleright P$ takes a step to $M' \triangleright P'$, then this new system is typeable under some typing environment Γ' such that such that $\Gamma \preceq \Gamma'$. An important case in the proof of this theorem is the case for contexts: i.e. *part* of the system takes a step, while the rest of the system remains the same. This case is dealt with by the subterm typing lemma, which we state and prove in this section. The subterm typing lemma is important, because it shows that the type system is *compositional*: in particular, if P and Q are two parallel processes in some system, and P deallocates a channel, then this will not affect Q.

Unfortunately, the subterm typing lemma is also a little technical to state and non-trivial to prove. We will, therefore, give the proof in detail, after we first give a number of small definitions and lemmas that we will require in the proof.

DEFINITION 4.6 (Restriction of a typing environment to a resource environment)

$$\Gamma|_M = \{d : \mathbf{T} \mid d : \mathbf{T} \in \Gamma \wedge M(d) = \top\}$$

DEFINITION 4.7 (Applying a context to a typing environment)

We define an operation $\mathcal{C}(\Gamma)$ which removes all assumptions about channels that are restricted in $\mathcal{C}$ from Γ:

$$[](\Gamma) = \Gamma \qquad (\mathcal{C} \parallel P)(\Gamma) = \mathcal{C}(\Gamma)$$
$$(\mathtt{new}c.\mathcal{C})(\Gamma) = \mathcal{C}(\Gamma)\backslash c \qquad (P \parallel \mathcal{C})(\Gamma) = \mathcal{C}(\Gamma)$$

DEFINITION 4.8 (Subenvironment)
M' is a subenvironment of M with respect to a typing environment Γ, denoted as $M \preceq_\Gamma M'$, if

(1) $\mathrm{dom}(M') = \mathrm{dom}(M)$;
(2) If $M(c) = \top$ and $M'(c) = \bot$ then $c : [\vec{\mathbf{T}}]^\bullet \in \Gamma$.

Intuitively, $M \preceq_\Gamma M'$ if we might end up in state M' by running a process typed under Γ in state M. This is formalized by the following lemma:

LEMMA 4.9 (Subenvironments and Reduction)
$\Gamma \vdash M \triangleright P$ and $M \triangleright P \to M' \triangleright P'$ then $M \preceq_\Gamma M'$

We will also require the following result which relates subenvironments and the operation on typing environments that we defined above.

LEMMA 4.10
Let $M_1 \preceq_\Gamma M_2$. Suppose $\mathcal{C}[M_1 \triangleright P_1] = M_1' \triangleright P_1'$ and $\mathcal{C}[M_2 \triangleright P_2] = M_2' \triangleright P_2'$ for some P_1 and P_2. Then $M_1' \preceq_{\mathcal{C}(\Gamma)} M_2'$.

We can now state the subterm typing lemma:

LEMMA 4.11 (Subterm typing)
Suppose $\Gamma_1^{\mathcal{C}} \vdash \mathcal{C}[M_1 \triangleright P_1]$ ($\Gamma_1^{\mathcal{C}}$ consistent). Then the following hold.

(1) There exist some Γ_1^p such that $\Gamma_1^{\mathcal{C}} \preceq \mathcal{C}(\Gamma_1^p)$ and $\Gamma_1^p \vdash M_1 \triangleright P_1$.
(2) For all systems $M_2 \triangleright P_2$ where $M_1 \preceq_{\Gamma_1^p} M_2$ and for all environments Γ_2^p where $\Gamma_1^p \preceq \Gamma_2^p$, if $\Gamma_2^p \vdash M_2 \triangleright P_2$ then there exist a $\Gamma_2^{\mathcal{C}}$ such that $\Gamma_1^{\mathcal{C}} \preceq \Gamma_2^{\mathcal{C}}$ and $\Gamma_2^{\mathcal{C}} \vdash \mathcal{C}[M_2 \triangleright P_2]$.

In words: if a system $M_1 \triangleright P_1$ is placed in some context $\mathcal{C}$, and the whole thing is typeable under some environment $\Gamma_1^{\mathcal{C}}$, then

(1) There is some environment Γ_1^p which types $M_1 \triangleright P_1$; the assumptions in this environment come from $\Gamma_1^{\mathcal{C}}$, with the exception of the assumptions for channels restricted by $\mathcal{C}$, and
(2) If we replace $M_1 \triangleright P_1$ by $M_2 \triangleright P_2$, typed under an environment Γ_2^p obtained by applying structural rules to Γ_1^p, then—provided there are unique permissions in Γ_1^p for the channels that are allocated in M_1 but deallocated in M_2—the whole thing is typeable under an environment $\Gamma_2^{\mathcal{C}}$, obtained by applying structural rules to $\Gamma_1^{\mathcal{C}}$.

PROOF. By induction on $\mathcal{C}$. The base case ($\mathcal{C} = []$) is trivial; remains to consider case for parallel and the case for restriction.

(1) Case $\mathcal{C} = \mathcal{C}' \parallel Q$. Let $\mathcal{C}'[M_1 \triangleright P_1] = M_1' \triangleright P_1'$. Then $\mathcal{C}[M_1 \triangleright P_1] = M_1' \triangleright P_1' \parallel Q$. By inversion we get

$$\dfrac{\dfrac{\Gamma_1^{\mathcal{C}'} \vdash P_1' \qquad \Gamma^q \vdash Q}{\Gamma_1^{\mathcal{C}} \vdash P_1' \parallel Q}\ \Gamma_1^{\mathcal{C}} \preceq \Gamma_1^{\mathcal{C}'}, \Gamma^q \qquad \mathrm{dom}(\Gamma_1^{\mathcal{C}}) \subseteq \mathrm{alloc}(M_1')}{\Gamma_1^{\mathcal{C}} \vdash \mathcal{C}[M_1 \triangleright P_1] = M_1' \triangleright P_1' \parallel Q} \tag{9}$$

It is not hard to see that

$$\dfrac{\Gamma_1^{\mathcal{C}'} \vdash P_1' \qquad \mathrm{dom}(\Gamma_1^{\mathcal{C}'}) \subseteq \mathrm{alloc}(M_1')}{\Gamma_1^{\mathcal{C}'} \vdash \mathcal{C}'[M_1 \triangleright P_1] = M_1' \triangleright P_1'}$$

so that by the induction hypothesis there exist a $\overset{\text{IH}}{\Gamma}{}_1^p$ such that $\Gamma_1^{\mathcal{C}'} \preceq \mathcal{C}'(\overset{\text{IH}}{\Gamma}{}_1^p)$ and $\overset{\text{IH}}{\Gamma}{}_1^p \vdash M_1 \triangleright P_1$. Choose $\Gamma_1^p = \overset{\text{IH}}{\Gamma}{}_1^p$. Clearly $\Gamma_1^{\mathcal{C}} \preceq \Gamma_1^{\mathcal{C}'} \preceq \mathcal{C}'(\Gamma_1^p) = \mathcal{C}(\Gamma_1^p)$.

Suppose we have a system $M_2 \triangleright P_2$ where $M_1 \preceq_{\Gamma_1^p} M_2$ and an environment Γ_2^p where $\Gamma_1^p \preceq \Gamma_2^p$, such that $\Gamma_2^p \vdash M_2 \triangleright P_2$. By the induction hypothesis, there exist a $\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}}$ such that $\Gamma_1^{\mathcal{C}'} \preceq \overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}}$ and $\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}} \vdash \mathcal{C}'[M_2 \triangleright P_2]$.

Let $\mathcal{C}[M_2 \triangleright P_2] = M_2' \triangleright P_2' \parallel Q$ where $\mathcal{C}'[M_2 \triangleright P_2] = M_2' \triangleright P_2'$. By inversion we have

$$\dfrac{\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}} \vdash P_2' \qquad \mathrm{dom}(\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}}) \subseteq \mathrm{alloc}(M_2')}{\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}} \vdash \mathcal{C}'[M_2 \triangleright P_2] = M_2' \triangleright P_2'} \tag{10}$$

By (9) we have $\Gamma^q = \Gamma^q|_{M_1'}$, and by Lemma 4.10 we have $M_1' \preceq_{\mathcal{C}(\Gamma_1^p)} M_2'$. Hence, channels allocated in M_1' but deallocated in M_2' must have a unique permission in $\mathcal{C}(\Gamma_1^p)$. Since $\Gamma_1^{\mathcal{C}'} \preceq \mathcal{C}(\Gamma_1^p)$ they must therefore have a unique permission in $\Gamma_1^{\mathcal{C}'}$. Finally, since $\Gamma_1^{\mathcal{C}} \preceq (\Gamma_1^{\mathcal{C}'}, \Gamma^q)$ and $\Gamma_1^{\mathcal{C}}$ is assumed consistent, this means that there cannot be any assumptions about these channels in Γ^q. Hence $\Gamma^q = \Gamma^q|_{M_1'} = \Gamma^q|_{M_2'}$. Finally, we have $\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}} = \overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}}|_{M_2'}$ by (10).

Choose $\Gamma_2^{\mathcal{C}} = \Gamma_1^{\mathcal{C}}|_{M_2'}$ (clearly $\Gamma_1^{\mathcal{C}} \preceq \Gamma_2^{\mathcal{C}}$). We have

$$\Gamma_2^{\mathcal{C}} \preceq (\Gamma_1^{\mathcal{C}'}, \Gamma^q)|_{M_2'} \preceq (\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}}, \Gamma^q)|_{M_2'} = \overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}}, \Gamma^q$$

The proof is completed by

$$\dfrac{\dfrac{\dfrac{\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}} \vdash P_2' \qquad \Gamma^q \vdash Q}{\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}}, \Gamma^q \vdash P_2' \parallel Q}\ \textsc{tPar}}{\Gamma_2^{\mathcal{C}} \vdash P_2' \parallel Q}\ \textsc{tStr} \qquad \mathrm{dom}(\Gamma_2^{\mathcal{C}}) \subseteq \mathrm{alloc}(M_2')}{\Gamma_2^{\mathcal{C}} \vdash \mathcal{C}[M_2 \triangleright P_2] = M_2' \triangleright P_2' \parallel Q}\ \textsc{tSys}$$

(2) Case $\mathcal{C} = \texttt{new}c.\mathcal{C}'$. Let $\mathcal{C}'[M_1 \triangleright P_1] = M_1', c:s_1 \triangleright P_1'$. Then $\mathcal{C}[M_1 \triangleright P_1] = M_1' \triangleright \texttt{new}c:s_1.P_1'$. We take cases on s_1.

(a) Case $s_1 = \top$. By inversion we get

$$\frac{\dfrac{\Gamma_1^{\mathcal{C}'}, c:\mathbf{T} \vdash P_1'}{\Gamma_1^{\mathcal{C}} \vdash \texttt{new}c:\top.P_1'}\ \Gamma_1^{\mathcal{C}} \preceq \Gamma_1^{\mathcal{C}'} \qquad \mathrm{dom}(\Gamma_1^{\mathcal{C}}) \subseteq \mathrm{alloc}(M_1')}{\Gamma_1^{\mathcal{C}} \vdash \mathcal{C}[M_1 \triangleright P_1] = M_1' \triangleright \texttt{new}c:\top.P_1'} \tag{11}$$

where $c \notin \mathrm{dom}(\Gamma_1^{\mathcal{C}}) \supseteq \mathrm{dom}(\Gamma_1^{\mathcal{C}'})$ by Barendregt. It is not hard to see that

$$\frac{\Gamma_1^{\mathcal{C}'}, c:\mathbf{T} \vdash P_1' \qquad \mathrm{dom}(\Gamma_1^{\mathcal{C}'}, c:\mathbf{T}) \subseteq \mathrm{alloc}(M_1', c:\top)}{\Gamma_1^{\mathcal{C}'}, c:\mathbf{T} \vdash \mathcal{C}'[M_1 \triangleright P_1] = M_1', c:\top \triangleright P_1'}$$

so that by the induction hypothesis, there exist a $\overset{\text{IH}}{\Gamma}{}_1^p$ such that $\Gamma_1^{\mathcal{C}'}, c:\mathbf{T} \preceq \mathcal{C}'(\overset{\text{IH}}{\Gamma}{}_1^p)$ and $\overset{\text{IH}}{\Gamma}{}_1^p \vdash M_1 \triangleright P_1$. Choose $\Gamma_1^p = \overset{\text{IH}}{\Gamma}{}_1^p$. We have

$$\Gamma_1^{\mathcal{C}} \preceq \left(\Gamma_1^{\mathcal{C}'} = (\Gamma_1^{\mathcal{C}'}, c:\mathbf{T}) \backslash c\right) \preceq \left(\mathcal{C}'(\Gamma_1^p) \backslash c = \mathcal{C}(\Gamma_1^p)\right)$$

Suppose we have a system $M_2 \triangleright P_2$ where $M_1 \preceq_{\Gamma_1^p} M_2$ and an environment Γ_2^p where $\Gamma_1^p \preceq \Gamma_2^p$, such that $\Gamma_2^p \vdash M_2 \triangleright P_2$. By the induction hypothesis, there exist a $\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}}$ such that $(\Gamma_1^{\mathcal{C}'}, c:\mathbf{T}) \preceq \overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}}$ and $\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}} \vdash \mathcal{C}'[M_2 \triangleright P_2]$.

Let $\mathcal{C}[M_2 \triangleright P_2] = M_2' \triangleright \texttt{new}c:s_2.P_2'$ where $\mathcal{C}'[M_2 \triangleright P_2] = M_2', c:s_2 \triangleright P_2'$. By inversion we have

$$\frac{\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}} \vdash P_2' \qquad \mathrm{dom}(\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}}) \subseteq \mathrm{alloc}(M_2', c:s_2)}{\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}} \vdash \mathcal{C}'[M_2 \triangleright P_2] = M_2', c:s_2 \triangleright P_2'} \tag{12}$$

Choose $\Gamma_2^{\mathcal{C}} = \Gamma_1^{\mathcal{C}}|_{M_2'}$ (clearly $\Gamma_1^{\mathcal{C}} \preceq \Gamma_2^{\mathcal{C}}$). Note that since $\Gamma_1^{\mathcal{C}} \preceq \Gamma_1^{\mathcal{C}'}$ by (11) we have $\Gamma_2^{\mathcal{C}} \preceq \Gamma_1^{\mathcal{C}'}|_{M_2'}$, and moreover $\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}} = \overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}}|_{M_2', c:s_2}$ by (12).
We take cases on s_2.

(i) Case $s_2 = \top$. We have $\Gamma_2^{\mathcal{C}}, c:\mathbf{T} \preceq \left((\Gamma_1^{\mathcal{C}'}|_{M_2'}, c:\mathbf{T}) = (\Gamma_1^{\mathcal{C}'}, c:\mathbf{T})|_{(M_2', c:\top)}\right) \preceq \overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}}$.
The proof is completed by

$$\dfrac{\dfrac{\dfrac{\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}} \vdash P_2'}{\Gamma_2^{\mathcal{C}}, c:\mathbf{T} \vdash P_2'}\ \textsc{tStr}}{\Gamma_2^{\mathcal{C}} \vdash \texttt{new}c:\top.P_2'}\ \textsc{tRst1} \qquad \mathrm{dom}(\Gamma_2^{\mathcal{C}}) \subseteq \mathrm{alloc}(M_2')}{\Gamma_2^{\mathcal{C}} \vdash \mathcal{C}[M_2 \triangleright P_2] = M_2' \triangleright \texttt{new}c:\top.P_2'}\ \textsc{tSys}$$

(ii) Case $s_2 = \bot$. We have $\Gamma_2^{\mathcal{C}} \preceq \Gamma_1^{\mathcal{C}'}|_{M_2'} \preceq \overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}}$ since we must have $c \notin \mathrm{dom}(\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}})$. Hence, the proof can be completed as above using TRST2.

(b) Case $s_1 = \bot$. By inversion we get

$$\frac{\dfrac{\Gamma_1^{\mathcal{C}'} \vdash P_1'}{\Gamma_1^{\mathcal{C}} \vdash \mathtt{new}c{:}\bot.P_1'}\ \Gamma_1^{\mathcal{C}} \preceq \Gamma_1^{\mathcal{C}'} \qquad \mathrm{dom}(\Gamma_1^{\mathcal{C}}) \subseteq \mathrm{alloc}(M_1')}{\Gamma_1^{\mathcal{C}} \vdash \mathcal{C}[M_1 \triangleright P_1] = M_1' \triangleright \mathtt{new}c{:}\bot.P_1'} \tag{13}$$

where $c \notin \mathrm{dom}(\Gamma_1^{\mathcal{C}}) \supseteq \mathrm{dom}(\Gamma_1^{\mathcal{C}'})$ by Barendregt. It is not hard to see that

$$\frac{\Gamma_1^{\mathcal{C}'} \vdash P_1' \qquad \mathrm{dom}(\Gamma_1^{\mathcal{C}'}) \subseteq \mathrm{alloc}(M_1', c{:}\bot)}{\Gamma_1^{\mathcal{C}'} \vdash \mathcal{C}'[M_1 \triangleright P_1] = M_1', c{:}\bot \triangleright P_1'}$$

so that by the induction hypothesis there exist a $\overset{\text{IH}}{\Gamma}{}_1^{p}$ such that $\Gamma_1^{\mathcal{C}'} \preceq \mathcal{C}'(\overset{\text{IH}}{\Gamma}{}_1^{p})$ and $\overset{\text{IH}}{\Gamma}{}_1^{p} \vdash M_1 \triangleright P_1$. Choose $\Gamma_1^{p} = \overset{\text{IH}}{\Gamma}{}_1^{p}$. Since $c \notin \mathrm{dom}\,\Gamma_1^{\mathcal{C}'} \supseteq \mathrm{dom}\mathcal{C}'(\overset{\text{IH}}{\Gamma}{}_1^{p})$ we have

$$\Gamma_1^{\mathcal{C}} \preceq \Gamma_1^{\mathcal{C}'} \preceq \left(\mathcal{C}'(\overset{\text{IH}}{\Gamma}{}_1^{p}) = \mathcal{C}'(\overset{\text{IH}}{\Gamma}{}_1^{p}) \backslash c\right) \preceq \mathcal{C}(\Gamma_1^{p})$$

Suppose we have a system $M_2 \triangleright P_2$ where $M_1 \preceq_{\Gamma_1^{p}} M_2$ and an environment Γ_2^{p} where $\Gamma_1^{p} \preceq \Gamma_2^{p}$, such that $\Gamma_2^{p} \vdash M_2 \triangleright P_2$. By the induction hypothesis, there exist a $\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}}$ such that $\Gamma_1^{\mathcal{C}'} \preceq \overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}}$ and $\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}} \vdash \mathcal{C}'[M_2 \triangleright P_2]$.

Let $\mathcal{C}[M_2 \triangleright P_2] = M_2' \triangleright \mathtt{new}c{:}s_2.P_2'$ where $\mathcal{C}'[M_2 \triangleright P_2] = M_2', c{:}s_2 \triangleright P_2'$. By inversion we have

$$\frac{\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}} \vdash P_2' \qquad \mathrm{dom}(\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}}) \subseteq \mathrm{alloc}(M_2', c{:}s_2)}{\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}} \vdash \mathcal{C}'[M_2 \triangleright P_2] = M_2', c{:}s_2 \triangleright P_2'} \tag{14}$$

Choose $\Gamma_2^{\mathcal{C}} = \Gamma_1^{\mathcal{C}}|_{M_2'}$ (clearly $\Gamma_1^{\mathcal{C}} \preceq \Gamma_2^{\mathcal{C}}$). As above, since $\Gamma_1^{\mathcal{C}} \preceq \Gamma_1^{\mathcal{C}'}$ by (13) we have $\Gamma_2^{\mathcal{C}} \preceq \Gamma_1^{\mathcal{C}'}|_{M_2'}$, and moreover $\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}} = \overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}}|_{M_2', c:s_2}$ by (14). In addition, we have $c \notin \mathrm{dom}(\overset{\text{IH}}{\Gamma}{}_2^{\mathcal{C}})$ since $c \notin \Gamma_1^{\mathcal{C}'}$. The proof therefore is completed as above.

■

4.3 *Safety and preservation*

We can now state and prove the main theorems of this article: type safety and safety reduction.

THEOREM 4.12 (Type safety)
If $\Gamma \vdash M \triangleright P$ (Γ consistent) then $M \triangleright P \not\twoheadrightarrow^{\mathrm{err}}$.

PROOF. By induction on $P \xrightarrow{\text{err}}$ followed by inversion on $\Gamma \vdash M \triangleright P$. We do not show the entire proof here as it is reasonably straightforward. The case for EATY relies on Lemma 4.1(1), the case for ESTR relies on Lemma 4.5, and the case for contexts relies on Lemma 4.11. ■

THEOREM 4.13 (Subject Reduction)
If $\Gamma \vdash M \triangleright P$ (Γ consistent) and $M \triangleright P \rightarrow M' \triangleright P'$ then there exists an Γ' such that $\Gamma \preceq \Gamma'$ and $\Gamma' \vdash M' \triangleright P'$.

PROOF. By induction on $M \triangleright P \rightarrow M' \triangleright P'$. We show only a number of the cases; of the omitted cases, RREC relies on the process substitution lemma, RTHEN and RELSE are trivial, RSTR follows from the induction hypothesis and preservation of types under structural equivalence (Lemma 4.5), and the case for contexts finally is dealt with by the subterm typing lemma (Section 4.2).

(1) Case $\dfrac{M(c) = \top}{M \triangleright c!\vec{d}.P \parallel c?\vec{x}.Q \rightarrow M \triangleright P \parallel Q\{\vec{d}/\vec{x}\}}$ RCOM
By inversion on the typing relation we get $\text{dom}(\Gamma) \subseteq \text{alloc}(M)$ and

$$\dfrac{\dfrac{\Gamma_1, c : [\vec{\mathbf{T}}]^{a_1 - 1} \vdash P}{\Gamma_p \vdash c!\vec{d}.P}\ \Gamma_p \preceq \Gamma_1, c : [\vec{\mathbf{T}}]^{a_1}, \overrightarrow{d : T} \qquad \dfrac{\Gamma_2, c : [\vec{\mathbf{T}}]^{a_2 - 1}, \overrightarrow{x : T} \vdash Q}{\Gamma_q \vdash c?\vec{x}.Q}\ \Gamma_q \preceq \Gamma_2, c : [\vec{\mathbf{T}}]^{a_2}}{\Gamma \vdash c!\vec{d}.P \parallel c?\vec{x}.Q}\ \Gamma \preceq \Gamma_p, \Gamma_q$$

where Lemma 4.1(1) allows us to conclude that the two assumptions $c : [\vec{\mathbf{T}}_1]^{a_1}$ and $c : [\vec{\mathbf{T}}_2]^{a_2}$, used to type the left and right process, respectively, must refer to the same object type $\vec{\mathbf{T}}_1 = \vec{\mathbf{T}} = \vec{\mathbf{T}}_2$. By the identifier substitution lemma (Lemma 4.4), we have that

$$\Gamma_2, c : [\vec{\mathbf{T}}]^{a_2 - 1}, \overrightarrow{d : T} \vdash Q\{\vec{d}/\vec{x}\}$$

Pick $\Gamma' = \Gamma$. Clearly $\text{dom}(\Gamma') \subseteq \text{alloc}(M)$. We construct the required type derivation as follows:

$$\dfrac{\dfrac{\dfrac{}{\Gamma_1, c : [\vec{\mathbf{T}}]^{a_1 - 1} \vdash P} \qquad \dfrac{}{\Gamma_2, c : [\vec{\mathbf{T}}]^{a_2 - 1}, \overrightarrow{d : T} \vdash Q\{\vec{d}/\vec{x}\}}}{(\Gamma_1, c : [\vec{\mathbf{T}}]^{a_1 - 1}), (\Gamma_2, c : [\vec{\mathbf{T}}]^{a_2 - 1}, \overrightarrow{d : T}) \vdash P \parallel Q\{\vec{d}/\vec{x}\}}\ \text{TPAR} \quad (*)}{\Gamma \vdash P \parallel Q\{\vec{d}/\vec{x}\}}\ \text{TSTR}$$

where the side condition $(*)$ is

$$\Gamma \preceq (\Gamma_1, c : [\vec{\mathbf{T}}]^{a_1 - 1}), (\Gamma_2, c : [\vec{\mathbf{T}}]^{a_2 - 1}, \overrightarrow{d : T})$$

and follows from Lemma 4.1(7), since environment are unordered.

(2) Case $\dfrac{c \notin \text{dom}(M)}{M \triangleright \texttt{alloc}x.P \rightarrow M \triangleright \texttt{new}c : \top.P\{^{c}/_{x}\}}$ RALL
By inversion we get $\text{dom}(\Gamma) \subseteq \text{alloc}(M)$ and

$$\dfrac{\Gamma_1, x : [\overrightarrow{\mathbf{T}}]^{\bullet} \vdash P}{\Gamma \vdash \texttt{alloc}x.P}\ \Gamma \preceq \Gamma_1$$

We have $c \notin \text{dom}(M) \supseteq \text{alloc}(M) \supseteq \text{dom}(\Gamma) \supseteq \text{dom}(\Gamma_1)$ and hence $c \notin \text{fn}(P)$.

Choose $\Gamma' = \Gamma$. Clearly $\Gamma \preceq \Gamma'$ and $\mathrm{dom}(\Gamma') \supseteq \mathrm{alloc}(M)$. The proof is completed by

$$\dfrac{\dfrac{\Gamma_1, c:[\overrightarrow{\mathbf{T}}]^\bullet \vdash P\{c/x\} \qquad \Gamma', c:[\overrightarrow{\mathbf{T}}]^\bullet \preceq \Gamma_1, c:[\overrightarrow{\mathbf{T}}]^\bullet}{\Gamma', c:[\overrightarrow{\mathbf{T}}]^\bullet \vdash P\{c/x\}}\ \textsc{tStr}}{\Gamma' \vdash \mathtt{new}c{:}\top.P\{c/x\}}\ \textsc{tRst1}$$

where the premise is established by the identifier substitution lemma (Lemma 4.4).

(3) Case $\dfrac{}{M, c:\top \triangleright \mathtt{free}c.P \to M, c:\bot \triangleright P}\ \textsc{rFree}$

By inversion we get $\mathrm{dom}(\Gamma) \subseteq \mathrm{alloc}(M, c:\top)$ and

$$\dfrac{\Gamma_1' \vdash P}{\Gamma \vdash \mathtt{free}u.P}\ \Gamma \preceq \Gamma_1', u:[\overrightarrow{\mathbf{T}}]^\bullet$$

By Lemma 4.1(3) there exist an Γ_1 such that $\Gamma = \Gamma_1, c:[\overrightarrow{\mathbf{T}'}]^\bullet$ and $\Gamma_1 \preceq \Gamma_1'$.
Choose $\Gamma' = \Gamma_1$. Clearly $\Gamma \preceq \Gamma'$. Moreover $\mathrm{dom}(\Gamma') \subseteq \mathrm{alloc}(M, c:\bot)$ since $c \notin \mathrm{dom}(\Gamma')$ by Lemma 4.1(2). The proof is completed by

$$\dfrac{\Gamma_1' \vdash P \qquad \Gamma' \preceq \Gamma_1'}{\Gamma' \vdash P}\ \textsc{tStr}$$

■

5 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 [9, 24].

Resources and pi-calculus: resource usage in a pi-calculus extension is studied in [26] 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 [26] is at a higher level of abstraction than $\mathtt{free}c.P$, and acts more like a 'resource finalizer' leading to garbage collection. Resource reclamation is implicit in [26], 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 [26] 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 [15]. 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 (*respectively* affinity) is a *local obligation* that a channel must be used exactly (*respectively* 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 article appears to be novel. The usefulness of the subtyping relation for affine or linear typing is well-known (e.g. see [19]); 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 [5] and linear typing have been used for this purpose, although even some proponents of linear typing agree that the match is not perfect [29, 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 [28] (or the equivalent Clean construct `#!`) and observer types [23] both fall into this category, and this approach has also been adopted by some non-functional languages where it is sometimes called *borrowing* [8]. 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 [11, 25, 31] and on CQual [12] 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.* [21] 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 article, 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 [7] 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.* [6], 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 [16] and types with CCS-like usage annotations [19] are used to describe channels which send objects of different types. However, these types give detailed prescriptions on how channels can be used, constraining them to follow specific protocols determined at compile-time; this 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. Moreover, most session type systems describe the interaction between *two* parties, and cannot describe the interaction of a client with two time servers, as in example CLIENT_1; multi-party session types [17] lift this restriction but at the expense of a significant increase of the complexity of the type system.

6 Conclusions and future work

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

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

Statically, it is hard to determine whether the third parallel process will eventually execute the $\texttt{free}x$ 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 $d_2?.\texttt{free}x$. In order to reject this process as ill-typed, the type-system needs to detect potential deadlocks. This can be done [20], 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.

One can envision various extensions of and variations on the type system we have presented. One obvious generalization is to consider a pair of attributes $\langle u, a\rangle$ indicating when the channel will be

unique (u) and how often the channel can be used (a). This leads to an infinite space of attributes of the form:

$$\begin{array}{ccccccc}
\langle\omega,\omega\rangle & \cdots & \langle 2,\omega\rangle & & \langle 1,\omega\rangle & & \langle 0,\omega\rangle \\
\vdots & \ddots & & & & & \vdots \\
\langle\omega,2\rangle & & \ddots & & & & \vdots \\
\langle\omega,1\rangle & & & & \ddots & & \vdots \\
\langle\omega,0\rangle & \cdots & \cdots & & \cdots & & \langle 0,0\rangle
\end{array}$$

The highlighted pairs are the pairs that we represent in our type system: $\langle\omega,\omega\rangle$ we denote simply as ω (unrestricted), $\langle u,\omega\rangle$ we represent as $(\bullet,u)$ or just $\bullet$ if $u=0$ (unique), and $\langle\omega,1\rangle$ we represent as 1 (affine). We could have taken ω to be syntactic shorthand for $(\bullet,\omega)$, which too would correspond to $\langle\omega,\omega\rangle$ in this matrix; although this would technically simplify the type language, we felt it would be confusing to describe unrestricted channels as a special kind of unique channels (i.e. channels that will be unique after an infinite number of steps).

Of the entries in the matrix that we do not represent, attributes of the form $\langle\omega,a\rangle$ for other values of a would certainly be of use and it should be straightforward to extend the type system to support them. However, entries of the form $\langle u,a\rangle$ with $u,a<\omega$ are less useful. If $a<u$, we know the channel will be unique after u steps but we are not allowed to take u steps, so we can consider the channel to have attribute $\langle\omega,a\rangle$ instead. Similarly, if $u<a$ we will only be able to do a restricted number of actions on the channel after it becomes unique, but (by definition of uniqueness) no other process will benefit from this restriction, so we can consider the channel to have attribute $\langle u,\omega\rangle$. Hence, the natural space of attributes is one-dimensional:

$$\begin{array}{ccccccccc}
\langle\omega,1\rangle & \langle\omega,2\rangle & \cdots & \langle\omega,\omega\rangle & \cdots & \langle 2,\omega\rangle & \langle 1,\omega\rangle & \langle 0,\omega\rangle \\
(\quad 1 & & & \omega & \cdots & (\bullet,2) & (\bullet,1) & \bullet \quad)
\end{array}$$

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 among themselves instead of the server. Modalities are a natural mechanism to preclude this from happening. For instance, we could generalize the splitting rule so that a unique channel $c:[\mathbf{T}]^{\bullet}$ can be split into two affine output channels of type $[\mathbf{T}]^{1}_{\text{out}}$ and an input channel of type $[\mathbf{T}]^{(\bullet,2)}_{\text{in}}$ which will be unique after two inputs.

One might even envision *two* usage attributes, one for input and one for output, so that we can have a channels that can be limited to be used for x inputs and y outputs, or dually channels that can are guaranteed to be unique after x inputs and y outputs. It is not clear at present whether this additional expressivity would be useful, however; if one wants to be more precise about communication protocols, session types [16] are probably a better choice.

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.

We have not designed a type checker for our type system. Type checking for substructural language without subtyping is well-understood (for instance, see the textbook [30]). Type checking for a substructural type system with subtyping is a less well-charted territory, but here too there is a lot of work, especially for functional languages [4, 5, 14, 18, 27] but also for imperative languages [10] and for the π-calculus [13].

Funding

Edsko de Vries and Matthew Hennessy are supported by SFI project [SFI 06 IN.1 1898].

References

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

[2] A. Ahmed, M. Fluet and G. Morrisett. A step-indexed model of substructural state. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, ACM, pp. 78–91, 2005.

[3] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[4] P. Baillot and M. Hofmann. Type inference in intuitionistic linear logic. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming* (New York, NY, USA, 2010), PPDP '10, ACM, pp. 219–230.

[5] E. Barendsen and S. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, **6**, 579–612, 1996.

[6] R. Bornat, C. Calcagno, P. O'Hearn and M. Parkinson. Permission accounting in separation logic. In *POPL '05 Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles Of Programming Languages*, Pages 259–270, ACM New York, NY, USA.

[7] J. Boyland. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium*, R. Cousot, ed., vol. 2694 of *LNCS*, Springer, pp. 55–72, 2003.

[8] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP 2003 – Object-Oriented Programming* (2003), vol. 2743 of *Lecture Notes in Computer Science*, Springer, pp. 59–67.

[9] E. de Vries. *Making Uniqueness Typing Less Unique*. PhD thesis, Trinity College Dublin, 2008.

[10] M. Degen, P. Thiemann and S. Wehr. Tracking linear and affine resources with java(x). In *ECOOP 2007 Object-Oriented Programming*, E. Ernst, ed., vol. 4609 of *Lecture Notes in Computer Science*. Springer, 2007, pp. 550–574.

[11] M. Fähndrich and R. DeLine. 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* (2002), ACM, pp. 13–24.

[12] J. S. Foster, T. Terauchi and A. Aiken. Flow-sensitive type qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (2002), ACM, pp. 1–12.

[13] S. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, **42**, 191–225, 2005.

[14] J. Hage, S. Holdermans and A. Middelkoop. A generic usage analysis with subeffect qualifiers. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, ACM, pp. 235–246, 2007.

[15] D. Harrington. Uniqueness logic. *Theoretical Computer Science*, **354**, 24–41, 2006.
[16] K. Honda, V. T. Vasconcelos and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP '98: Proceedings of the 7th European Symposium on Programming*, Springer, pp. 122–138, 1998.
[17] K. Honda, N. Yoshida and M. Carbone. Multiparty asynchronous session types. In *POPL '08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, pp. 273–284, 2008.
[18] O. Kiselyov and C.-C. Shan. A substructural type system for delimited continuations. In *Typed Lambda Calculi and Applications*, S. Della Rocca, ed., vol. 4583 of *Lecture Notes in Computer Science*. Springer, pp. 223–239, 2007.
[19] N. Kobayashi. Type systems for concurrent programs. In *Formal Methods at the Crossroads: From Panacea to Foundational Support*, vol. 2757 of *LNCS*, Springer, pp. 439–453, 2003.
[20] N. Kobayashi. A new type system for deadlock-free processes. In *CONCUR 2006 âŁ" Concurrency Theory*, vol. 4137 of *Lecture Notes in Computer Science*, Springer, pp. 233–247, 2006.
[21] N. Kobayashi, B. C. Pierce and D. N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **21**, 914–947, 1999.
[22] R. Milner, K. Parrow and D. Walker. A calculus of mobile processes, parts i and ii. *Information and Computation*, **100**, 1–40; (part ii, 41–47). 1992.
[23] M. Odersky. Observers for linear types. In *Proceedings of the 4th European Symposium on Programming (ESOP)*, B. Krieg-Brückner, ed., vol. 582 of *Lecture Notes in Computer Science*, Springer, pp. 390–407, 1992.
[24] F. Pottier. Wandering through linear types, capabilities, and regions, 2007. Survey talk given at INRIA, Rocquencourt, France.
[25] F. Smith, D. Walker and J. G. Morrisett. Alias types. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, Springer, pp. 366–381, 2000.
[26] D. Teller. Recollecting resources in the pi-calculus. In *Proceedings of IFIP TCS 2004*, Kluwer Academic Publishing, pp. 605–618, 2004.
[27] T. Terauchi and A. Aiken. Witnessing side-effects. *SIGPLAN Not.*, **40**, 105–115, 2005.
[28] P. Wadler. Linear types can change the world! In *Proceedings of the IFIP TC2 WG 2.2/2.3 Working Conference on Programming Concepts and Methods*, M. Broy and C. B. Jones, eds. North-Holland, pp. 561–581, 1990.
[29] P. Wadler. Is there a use for linear logic? In *PEPM*, ACM New York, NY, USA, pp. 255–273, 1991.
[30] D. Walker. Substructural type systems. In *Advanced Topics in Types and Programming Languages*, B. C. Pierce, ed., The MIT Press, 2005.
[31] D. Walker and J. G. Morrisett. Alias types for recursive data structures. In *TIC '00: Third International Workshop on Types in Compilation*, Springer, pp. 177–206, 2001.

Received October 2010, revised July 2011, accepted February 2012