4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Towards a Resource-Safe Erlang

David Teller
LIFO, Université d'Orléans

RR-2007-06

Towards a Resource-Safe Erlang

David Teller

February 3, 2007

Abstract

Slowly but surely, industry is discovering the need for programming languages, runtime environments and methodologies adapted to collaborative and distributed computing platforms. However, current distributed platforms, whether industrial or academic, are generally fragile with respect to resource exhaustion, and can provide, at best, ad hoc solutions to counter accidents or Denial of Service attacks. In this paper, we examine the problem of resource management in Erlang, that is providing services for distant use, while ensuring that untrusted third-parties using the services may not cause the exhaustion of memory, file handles or other limited resources. For this, we provide a formal semantics for a subset of Core Erlang, as well as a model of its library, and we provide a type system for formally proving robustness of services with respect to Denial of Service attacks.

Contents

Chapter 1

Introduction

In the course of the last decades, the landscape of computing has seen a gradual change from purely hierarchical organisations to distributed teams, or even distributed groups of agents, cooperating through decentralised applications, information repositories, online services and now web applications. All these forms of collaborative computing offer numerous opportunities and require new tools and new manners of thinking. In turn, the new tools open the doors to numerous new weaknesses with respect to security and robustness. Indeed, even clean languages designed for distribution, such as Erlang [2] or the strongly-typed JoCaml [7] or Nomadic Pict [21], have little in the manner of protection against malicious or ill-programmed agents.

Of concern, for instance, are Denial-of-Service attacks: can an attacker cause a service to expend all its memory, to open more files than the operating system can handle or to use up all the paper available for its printers ? While dynamic techniques exist – and are included in most modern operating systems and virtual machines – to watch the consumption of resources and terminate ill-behaved processes, they are typically not defined formally, apply to only a specific set of resources, and may only detect these *resource-exhausting* processes when it is too late, especially when the resources under attack are limited and cannot be deallocated. The alternative is to analyse agents at compile-time or launch-time and execute only that do not exhaust resources.

This work presents a step in the development of a resource-safe Erlang, in which both results would be achievable. Erlang is a concurrent, dynamically typed, distributed, purely functional programming language, widely used in the world of telecommunications. While Erlang is not an implementation of formal semantics, several attempts have been made at formalising the semantics of Erlang, either directly[8] or by translating it into π-calculus[15]. We use and extend this last work to offer a denotational semantics of Core Erlang in Teπc[19], a targettable/extendable π-calculus. In addition to capturing the computational behaviour of Erlang, this encoding allows us to observe the resource usage of a program, hence to define a dynamic notion of resource exhaustion and potentially to detect Denial-of-Service attacks. For instance, let us consider the following Erlang extract:

```
log_fragile(Item) ->
    let File = open("log") in
        write(File, Item),
        close(File).
```

The program defines a service `log_fragile`, for use by any agent over the network, whose role it is to write items in a file. The opening of this file is an action which may require resources, such as some space in the file allocation table or some reading privileges. If it is executed, statement `write(file, item)` will run in an environment where these resources are referenced by `file`. Eventually, once the item has been written to the file, that file is closed. Finally, depending on the resources involved, this closing may allow the system to recover some or all of the resources – resources such as memory may be fully recovered while credentials or access logs are more complex.

As it turns out, if the number of files opened at any given time is a critical resource, then this service happens to be fragile: since there is no synchronisation between instances of the service, a malicious agent only has to fork and call `log_fragile` often enough to eventually reach such a state in which resource "file handler" is exhausted.

In Teπc, one could model the opening of this file as a transition $i \longrightarrow^* (\nu file = File())j$, where i is the body of `log_fragile` and j is the remainder of the operations after the opening, that is writing and closing. This transition marks the fact that, while i is executed in an empty environment, j runs with a reference $file$ to some resource $File$. Conversely, closing the file is marked by a transition such as $(\nu file = File())k \longrightarrow^* l$. In particular, the resource usage of these processes differs. If the system is only able to maintain n open files at any given time, a process such as $(\nu file_1 = File())j_1 \mid (\nu file_2 = File())j_2 \mid \cdots \mid (\nu file_{n+1} = File())j_{n+1}$ is exhausting the system's resources and should be stopped.

By opposition, the following listing presents a trivial robust manager for the service, which can be called any number of times and will never use more than one file[1]:

```
let A = new_lock() in
    letrec log_robust = fun(Item) ->
        acquire_lock(A),
        log_fragile(Item),
        release_lock(A)
    in log_robust.
```

The objective of this work is to formalise the difference between the manners of managing resources used in `log_fragile` and `log_robust`, to model resource exhaustion and to provide formally proved tools for guaranteeing a service with

[1]Note that the syntax of Core Erlang is somewhat different from that of Erlang/OTP. This extract complies with the former. A close approximation of this source code in Erlang/OTP is presented in appendix A

respect to Denial of Service attacks. In Section 2, we present Teπc, which we use in Section 3 as the domain of a denotational semantics of a subset of Core Erlang. In Section 4, we make use of this encoding to provide a static type system to guarantee the robustness of a program. We then conclude by an overview of related works and future developments.

Chapter 2

Teπc

Teπc, the targettable/extendable π-calculus, is an extension of the controlled π-calculus[18], itself a variant of the π-calculus[14], a formal language designed for modelling and studying concurrent and distributed systems with dynamic reconfigurations. To this base, the controlled π-calculus adds a notion of *resources* and resource *limitations*. In turn, Teπc introduces notions of *foreign operations* and *foreign values*, used to model underlying virtual machine or operating system calls. Numerous aspects of Teπc are left parametric, so as to allow targetting Teπc for the modelling of specific systems.

We first introduce the syntax and semantics of the core language before completing this definition to take into account constraints on resource usage. A more detailed presentation of this calculus may be found in a (work-in-progress) report on this subject [19] .

2.1 Resource-unaware semantics

The main concepts of Teπc are *names* (references to foreign values and to communication channels), *processes* (the state of a system or subsystem), *instructions* (the code for a system or subsystem), *foreign operations* (calls to primitives of the operating system/virtual machine), *foreign values* (results of foreign operations, e.g. file handlers), *resources* (e.g. amounts of memory) and *deallocation* of names.

The syntax of Teπc is presented on figure 2.1. Weak references are either variables (x, y ...), references to either channels or values (a, b ...), or the dangling reference ($\circledcirc$, or "null"). Note that there is no difference between references and communication channels: any communication channel is also a reference to some (possibly useless) value, while any non-$\circledcirc$ reference may be used as a communication channel.

Processes represent the current state of a system being executed. A process may be an instruction i, the composition $P|Q$ of two processes running concurrently, the choice $i + j$ between two instructions or $(\nu a = v)P$, a process P in which name a is a reference to foreign value v.

Instructions represent code and may state either `nothing` ("do nothing"), `spawn` i `and` j ("execute concurrently i and j"), `foreign` $u = e$ `in` i ("execute foreign operation e, calling the result u"), `either` i `or` j ("execute either i or j, whichever is ready first"), or communications. Instruction $m\ n$ sends one message n on channel m. A messages emitted on channel m will remain available without time limitation, until either it has been received or channel m has been destroyed. Note that messages are names themselves, and may serve to reference channels as well as foreign values. Instruction `once` $m\ x$ `do` i defines a continuation, in other words, expects exactly one message on channel m and, once this message has been received, binds it to x and execute i. This instruction will keep listening for a message on channel m until it receives one such message or until channel m is destroyed. Conversely, `on` $m\ x$ `do` i defines a service, in other words, no matter how many messages are sent on channel m, bind each of these messages to x and execute i. This instruction will keep listening until channel m is destroyed. If two processes send messages along the same channel at the same time or if two processes are listening on the same channel at the same time, the language specifies only that the distribution of messages will involve only one emitter and one receiver at a time.

Finally, both *foreign operations* and *foreign values* are built from weak references and a parametric vocabulary of constructors $E_1, E_2 \ldots$

We do not detail the notions of free names, free variables or free strong references, which are natural extensions of the corresponding notions in the π-calculus. In the rest of this report, we will assume Barendregt-style convention on references.

Weak References	m, n	$\in$ WEAK	$::=$	$u \mid \circledcirc$
Strong References	u, v	$\in$ STRONG	$::=$	$a \mid x$
Names	a, b	$\in \mathcal{N}$		
Variables	x, y	$\in \mathcal{V}$		
Processes	P, Q	$\in$ PROC	$::=$	$(\nu a = v)P \mid P \vert Q \mid i + j \mid i$
Instructions	i, j	$\in$ CODE	$::=$	`nothing` $\mid$ `foreign` $u = e$ `in` i $\mid$
				`spawn` i `and` j $\mid$ `once` $m\ x$ `do` i $\mid$
				`on` $m\ x$ `do` i $\mid$ $m\ n$ $\mid$ `either` i `or` j
Foreign values	v, w	$\in$ VALUE	$::=$	$E(\overrightarrow{v'})$
	v'		$::=$	$a \mid \circledcirc \mid v$
Foreign operations	e, f	$\in$ OP	$::=$	$m \mid E(\overrightarrow{e})$
Process contexts	$C[\cdot]$		$::=$	$(\nu a = v)C[\cdot] \mid C[\cdot] \vert P \mid P \vert C[\cdot] \mid \cdot$
Value context	$V[\cdot]$		$::=$	$v, V[\cdot] \mid V[\cdot], v \mid F(V[\cdot]) \mid \cdot$
Operation context	$I[\cdot]$		$::=$	$e, I[\cdot] \mid I[\cdot], e \mid E(I[\cdot]) \mid \cdot$

Figure 2.1: Syntax of Teπc.

The resource-unaware semantics of Teπc essentially extends that of the π-calculus. The main differences lie in the invocation of foreign operations and the destruction of names, both of which are presented presented on figure 2.2. Evaluation of an foreign operation is handled by rules R-EVALUATE, to initiate

$$\text{R-EVALUATE}\ \texttt{foreign}\ u = e\ \texttt{in}\ i \xrightarrow{e \rightsquigarrow f}_{pre} \texttt{foreign}\ u = f\ \texttt{in}\ i$$

$$\text{R-FETCH}\ \frac{P \xrightarrow{e \rightsquigarrow f}_{pre} P'}{(\nu a = v)P \xrightarrow{\tau}_{pre} (\nu a = v)P'}\ a = v \Vdash e \rightsquigarrow f \qquad fv(e) = fv(f) = \emptyset$$

$$\text{R-DEREF}\ \frac{}{\texttt{foreign}\ x = e\ \texttt{in}\ i \xrightarrow{\tau}_{pre} i\{x \leftarrow b\}} \qquad e \multimap b \qquad fv(e) = \emptyset$$

$$\text{R-STORE}\ \frac{}{\texttt{foreign}\ a = e\ \texttt{in}\ i \xrightarrow{\tau}_{pre} (\nu a = v)i}\ e \multimap v \qquad fv(e) = \emptyset$$

$$\text{R-DEALLOCATE}\ \frac{}{(\nu a = v)P \xrightarrow{\tau}_{pre} P\{a \leftarrow \circledcirc\}} \qquad a = v \vDash P$$

Figure 2.2: Allocation and deallocation of resources in Teπc.

possible transitions, and R-FETCH, to fetch the value of references which may be required for the evaluation. Once the evaluation is complete, it may yield either a reference, which is then dereferenced by R-DEREF, or a foreign value, which is stored in the environment by R-STORE. These rules depend on parametric relations on a parametric relations $a = v \Vdash e \rightsquigarrow e'$ ("assuming that a references valule v, a foreign operation in state e will progress and reach state e'") and $e \multimap v$ ("a foreign operation in state e is complete and yields a result v"), respecting the following criteria:

- if $a = v \Vdash e \rightsquigarrow e'$ then $fv(e) = \emptyset$ and $fn(e') \subseteq fn(e) \cup fn(v) \cup \{\circledcirc\}$
- if $e \multimap v$ then $fv(e) = \emptyset$ and $fn(v) \subseteq fn(e) \cup \{\circledcirc\}$
- if $e \multimap m$ and $fn(e) \subseteq \{m\} \cup \{\circledcirc\}$.

Conversely, destruction of a name is handled by a parametric relation by a parametric relation $a = v \vDash P$ ("in process P, binding $a = v$ should be destroyed") through rule R-DEALLOCATE. Substitution and elimination of dead instructions – two aspects critical to Teπc but not the focus of the present document – are briefly presented in appendix B.1. Appendix B.2 recapitulates the full labelled transition system.

The memory model (and more generally, the resource model) implied by Teπc is that of a direct acyclic graph of references with static single assignment and garbage-collection, in which cycles may, if necessary, be provided by processes.

In particular, note that, as in the π-calculus, every branching, flow control structure or function call is modelled as a sequence of messages along communication channels, possibly accompanied by foreign operations.

The extracts of figure 2.3 demonstrate the definition and usage of a service `if_then_else` in Teπc. Equations 2.1, 2.2 and 2.3 specify the behaviour of a

Specification of foreign operations

$$IfThenElse(True, a, b) \multimap a \tag{2.1}$$

$$IfThenElse(False, a, b) \multimap b \tag{2.2}$$

$$\frac{a = v \Vdash e \rightsquigarrow f}{a = v \Vdash IfThenElse(e, b, c) \rightsquigarrow IfThenElse(f, b, c)} \tag{2.3}$$

Defining and using `if_then_else`

```
on if_then_else e a b do
 foreign go = IfThenElse(e,a,b) in
  go

foreign true  = True
foreign false = False
--
spawn once a do ...
and   once b do ...
and   if_then_else true a b
```

Figure 2.3: Defining and using "if. . . then. . . else" in Teπc

foreign operation *IfThenElse* inside the virtual machine: if e is a foreign value and a and b are two names, after one or more steps of reduction, foreign operation $IfThenElsee, a, b$ will produce either foreign value a or foreign value b, depending on the result of the evaluation of e. Teπc code `on if_then_else e a b do ...` defines a service, listening for requests on channel `if_then_else`. When invoked, this service calls foreign operation *IfThenElse*, binds the result to name *go* and sends a message along this channel *go*. Conversely, Teπc code `spawn ...` spawns three processes: `once a do ...` is a continuation on a, expecting a message along channel a before proceeding, `once b do` is a continuation on b, expecting a message along channel b before proceeding and `if_then_else true a b` invokes service `if_then_else`, hence eventually causing the emission of an empty message on channel a, hence the release on the continuation on a.

2.2 Resources

In addition, the semantics of Teπc is parametrised by resource specifications, determining the *nature* of resources being modelled, the *reserve* of resources available to the system and exactly what kind of resource is occupied by each foreign value. Note that *only foreign values occupy resources* – as we will see later, this is not a limitation of the system.

Definition 1 (Nature of a resource)
The set of resource natures (or "specifications") $(\mathcal{S}, \oplus, \preceq, \bot, \top)$ *is a parameter of the language, such that* $(\mathcal{S}, \oplus)$ *is a monoid,* $\preceq$ *is a partial order on* $\mathcal{S}$*, with a minimal element* $\bot$ *and a maximal element* $\top$ *and* $\oplus$ *is monotonic with respect to* $\preceq$ *(i.e.* $\forall a, b, c$*, if* $a \preceq b$ *then* $a \oplus c \preceq b \oplus c$*).*

From $\preceq$, we derive relations $\prec$, $\succ$, $\succeq$.

The resource-contrained semantics of Teπc is parametrised by

- the nature of resources being manipulated $\mathcal{S}$, as defined above,
- a reserve of resources *reserve*, element of $\mathcal{S}$,
- a resource signature *res*, function from VALUE to $\mathcal{S}$, stable by garbage-collection of references – that is, if $res(v) = r$ then $res(v\{a \leftarrow \circledcirc\}) = r$ – and by α-conversion – that is, if $res(v) = r$ then $res(v\{a \leftarrow b\}) = r$

We extend this definition of *res* to processes, by

$$\begin{array}{lcl} res(i) & = & \bot \\ res(i+j) & = & \bot \\ res(P|Q) & = & res(P) \oplus res(Q) \\ res((\nu a = v)P) & = & res(P) \oplus res(v) \end{array}$$

Definition 2 (Resource-aware semantics) *The resource-aware semantics of Teπc is defined by*

$$\frac{P \xrightarrow{l}_{pre} Q \qquad res(Q) \preceq r}{P \overset{l;r}{\longmapsto} Q} \qquad\qquad \frac{P \overset{\tau;reserve}{\longmapsto} Q}{P \longrightarrow_{reserve} Q}$$

Definition 3 (Exhaustion) *A transition* $P \xrightarrow{l}_{pre} Q$ *is said to* exhaust *the system iff* $res(Q) \succ reserve$*. A process* P *is* non-exhausting *if* $res(P) \preceq reserve$ *and for any* P' *such that* $P \xrightarrow{\tau}_{pre} P'$*,* P' *is a non-exhausting process.*

Definition 4 (Client process) *An process* P *is* client *(or* unpriviledged*) if* $res(P) = \bot$ *and for any* P' *such that* $P \xrightarrow{l}_{pre} P'$ *then* P' *is a client process.*

Informally, a client process is a process which cannot directly allocate resources *inside the system*, although it may possibly cause a privileged process to do so. Typically, processes which are not part of the implementation of the operating system/virtual machine are unprivileged, as they need to request the operating system to allocate resources for them. For instance, assuming for a second that foreign operation *IfThenElse* uses resources, process `on if_then_else e a b do ...` is privileged, while process `if_then_else true a b` is not. In particular, we will consider that all potential Denial-of-Service attackers are client processes, as they may not allocate resources *of the system* directly.

Definition 5 (Robustness) *A process P is robust with respect to denials of service attacks if, for any client process Q, $P|Q$ is a non-exhausting process.*

Chapter 3

From Erlang to Teπc

In this section, we introduce Teπc-Erlang, the encoding of a subset of Core Erlang [4] by translation to Teπc. This encoding is a vastly enriched superset of π-Erlang [15], and the core model we use in this work as a *definition* of the semantics of Erlang[1] As space constraints do not permit it, we cannot present Erlang itself in this document. Rather, the bibliography suggests further readings [4] on the subject for the interested reader.

As any language definition in Teπc, the specification of Teπc-Erlang involves the following steps: defining the vocabulary of foreign values and foreign operations, the semantics of foreign operations, the semantics of garbage-collection and – at last – the encoding of Erlang expressions in Teπc. To this, we add a model of a few functions/services of Erlang's standard library, including representation of primitive values, process spawning, some aspects of exceptions, mutexes, some file management, etc. In order to ensure some clarity of the coding, we make sure that these services are the only *privileged* processes on the system. In addition, so as to permit controlling the number of processes being spawned, we make sure that each process holds a reference to a foreign value, which we will take into account during the analysis of resource usage, in Section 4.

3.1 Foreign values and operations

In our translation of Erlang, we will make use of the following constructors:

arity 0 $Channel$ (regular communication channels), $File$ (files), $Lock$ (mutexes), Nil (empty list), $True$, $False$ (booleans), $Number_n$ (constant

[1] To the best of our knowledge Erlang has already three different – and contradictory – formal semantics, including π-Erlang. None of these semantics seems to prove any meaningful abstraction result, nor to clarify to which version of Erlang – themselves incompatible – it is related. We decided to use this enriched π-Erlang as it forms the semantics to which our theories apply best.

numbers), $Char_c$ (constant characters), $String_s$ (constant strings), $Process$ (Erlang processes)

arity 1 $Finalise$, $Free$ and $Terminate$ (used to communicate with the garbage-collector), $Head$ and $Tail$ (list operations), Nth_n (nth element in a tuple)

arity 2 $Cons$ (list concatenation), $Same$ (name equality predicate)

arity n $Tuple_n$ (tuple constructor)

With the exclusion of $Channel$, these foreign values are manipulated exclusively by the library. The semantics of all the corresponding foreign operations is detailed in appendix C.1.

3.2 Encoding

In this section, we introduce informally various aspects of Teπc-Erlang. The encoding itself – essentially source code – is available in the appendices.

For the sake of readability, we extend the syntax of Teπc to allow polyadic communications, n-ary name creation, as well as n-ary parallel composition `spawn` $\cdots$ `and` $\cdots$`and`$\cdots$ and choice `either` $\cdots$ `or` $\cdots$`or`$\cdots$ and a special variable __ (or "don't care"), which is supposed to never appear free in any term. We will also use notation `new` a `in` $\cdots$ for `foreign` $a = Channel()$ `in` $\cdots$.

The denotational semantics of modules is a function $\mathcal{M}[\![\cdot]\!]$, while that of functions is $\mathcal{F}[\![\cdot]\!]$, both to the domain of Teπc instructions. The encoding of expressions with a return channel res and being evaluated with a process with pid $self$ is a function $\mathcal{E}[\![\cdot]\!]^{res,self}$, also to the domain of Teπc instructions. These functions, detailed in appendix C.2, are defined much as in π-Erlang, with the following differences:

- names are kept private to their module unless specifically exported
- primitive values are encoded rather than marked as `unknown`
- arguments of function calls are evaluated
- message reception does not differentiate between finite and infinite timeouts
- message reception is defined from pattern-matching
- Teπc-Erlang supports definition of local functions
- the translation of `case` involves actual pattern-matching rather than pure non-determinism.

As in π-Erlang, we do not model Erlang's `primop` (i.e. calls to implementation-dependent primitives which may depend on the whole state of the system and/or have side effects), links (i.e. process failures), module attributes, or module importation. As in π-Erlang we ignore the order of clauses in a `case` statement.

The encoding of expressions relies on the translation of pattern-matching, as detailed in appendix C.3. This translation is more complex than that of π-Erlang, and takes the form of a function $\mathcal{C}\llbracket\cdot\rrbracket^{r,s,i,o,g}$, the encoding of a clause with return channel r, being evaluated within a process with pid s, to match a value received on channel i, returning the substitution on channel o for management by the `case` statement itself, and expecting a message on g to actually proceed with the evaluation. In turn, this function uses $\mathcal{P}\llbracket\cdot\rrbracket^{in,out}$, the encoding of a pattern matching value in and returning a substitution on channel out, and $\mathcal{V}\llbracket\cdot\rrbracket$, the vector of free variables in a pattern.

3.3 Library

A specification of Erlang is incomplete without a model or encoding of the library. Although a full model is beyond the scope of this paper, we have studied a few key functions to represent primitive values, process spawning, some aspects of exceptions, etc. These entries in the library are defined directly as Teπc instructions, with the same calling conventions as Teπc-Erlang functions. The model is detailed in appendix C.4. Both message emission and process identification are essentially identical to their π-Erlang counterparts. Process spawning creates a depositary $Process$, which may be used later, if necessary, to control the number of processes running. Process suicide or termination, which doesn't appear in π-Erlang, is invoked respectively by channels $\texttt{erlang}:\texttt{exit}_3$ and $\texttt{erlang}:\texttt{exit}_4$. The encoding then contains a place-holder to check whether the target process traps exits. In this version of the encoding, it is impossible to trap exits and the request for termination is forwarded to some predefined channel *terminate*. File opening and closing, as well as waitlines, have been simplified for the sake of examples. File closing sends a message to the garbage-collector on some predefined channel $free$, then waits for garbage-collection to take place before returning *true*. Although the specifications of the garbage-collector are far from complete – and well beyond the scope of this paper – we specify that both *terminate* and $free$ trigger destruction, respectively of process identifiers and of files.

3.4 Notes

As in π-Erlang, we do not model Erlang's `primop` (i.e. calls to implementation-dependent primitives which may depend on the whole state of the system and/or have side effects), links (i.e. process failures), module attributes, or module importation. Also, as in π-Erlang we ignore the order of clauses in a `case` statement.

From this encoding, we define a notion of executable system and a notion of robustness.

Informal definition 1 (Executable) *If LIB is the model of Erlang's standard library, if e is an Erlang program in which names $a_1, a_2 \ldots a_n$ denote ser-*

vices accessible to the public, and if i_e is the result of the encoding of e in Teπc-Erlang, the executable process *for e in LIB is defined as the Teπc process offering LIB and i_e aa*

and if i_e is the instruction $\mathcal{E}\llbracket e \rrbracket^{res,self}$, the executable process for e in LIB is defined as `new` $\overrightarrow{b \in fn(i_e) \backslash \overrightarrow{a}}$ `in` $(LIB \mid i_e)$.

Informal definition 2 (Robust) *An Erlang program e is robust with respect to Denial of Service if its executable process is robust with respect to Denial of Service.*

Note that this notion of robustness is compatible with distributed applications: as long as the communications between the nodes are kept private by `new` $\cdot$ `in` $\cdot$, i_e may well be distributed between nodes.

3.5 Examples

From this definition of Teπc-Erlang, we may encode the example `log_fragile` as:

```
on log_fragile item res self do
 new res' in
  spawn new res_1 in
   spawn string_"log" res_1 self
   and once res_1 x do open x res' self
  and once res' x do new res'' in
   new res_1, res_2 in
    spawn res_1 x
    and   res_2 item
    and once res_1 x_1 do once res_2 x_2 do
     write x_1 x_2 res'' self
    and once res'' _ do
    new res_3 in
     spawn res_3 x
     and   once res_3 x_3 do
      close x3 res self
```

Conversely, the encoding of `log_robust` yields

```
new res' in
 spawn new res_1 in
  spawn new_lock res_1 self in
  and once res_1 x do
   new log_robust in

spawn on log_robust item res self do
 new res' in
  spawn new res_2 in
   spawn res_2 x
   and   once res_2 x do acquire_lock x res' self
  and once res' x do new res'' in
   new res_3 in
    spawn res_3 item
    and once res_3 x_3 do
     log_fragile x3 res'' self
    and once res'' _ do
    new res_3 in
     spawn res_3 x
     and   once res_3 x_3 do
      release_lock x3 res self
and res log_robust
```

3.6 Resources

Note that we have not yet specified the nature of resources. Indeed, while guaranteeing that an Erlang service is robust with respect to Denial-of-Service requires a precise definition of which resources are critical and what operations require such resources, neither the specifications of Erlang nor the translation to Teπc define resource constraints. For instance, it is imaginable that an implementation of Erlang for Linux-based embedded systems could use one Linux kernel thread for each Erlang thread. This implementation would be limited to the simultaneous execution of a few hundred threads, by opposition to the nearly-limitless concurrency of the standard distribution of Erlang/OTP. Similarly, an Erlang program running on sensor networks will be faced with the problems of managing energy reserves, whereas the critical resources for an application server would presumably be memory or bandwidth.

In order to obtain a set of resources measuring opened files and file opening rights, let us first define the nature of opened files $(\mathcal{S}_F, \oplus_F, \preceq_F, \bot_F, \top_F)$ as

- $\mathcal{S}_F \triangleq \{Files(n), n \in \mathbf{N} \cup \{\infty\}\}$
- $\forall Files(m), Files(n) \in \mathcal{S}_F, Files(m) \oplus_F Files(n) \triangleq Files(m+n)$
- $\forall Files(m), Files(n) \in \mathcal{S}_F, Files(m) \preceq_F Files(n) \iff m \leq n$
- $\bot_F = Files(0)$, $\top_F = Files(\infty)$.

Let us now define the nature of file opening rights $(\mathcal{S}_R, \oplus_R, \preceq_R, \bot_R, \top_R)$ as

- $\mathcal{S}_R \triangleq Rights(\mathbf{B})$
- $\forall Rights(m), Rights(n) \in \mathcal{S}_R, Rights(m) \oplus_R Rights(n) \triangleq Rights(m \vee n)$
- $\forall Rights(m), Rights(n) \in \mathcal{S}_R, Rights(m) \preceq_R Rights(n) \iff m \Rightarrow n$
- $\bot_R = Rights(ff)$, $\top_R = Rights(tt)$.

From these, we derive the definition of set $\mathcal{S}$, the nature of resources for our example, or $(\mathcal{S}, \oplus, \preceq, \bot, \top)$, as

- $\mathcal{S} \triangleq \mathcal{S}_F \times \mathcal{S}_R$
- $\forall r = (r_F, r_R), s = (s_F, s_R) \in \mathcal{S} r \oplus s \triangleq (r_F \oplus s_F, r_R \oplus s_R)$
- $\forall r = (r_F, r_R), s = (s_F, s_R) \in \mathcal{S} r \preceq s \iff r_F \preceq s_F$ and $r_R \preceq s_R$
- $\bot = (\bot_F, \bot_R)$, $\top = (\top_F, \top_R)$.

To this set, we associate a bound $\mathit{reserve} \triangleq 1, tt$ (i.e. only one file may be opened at any time). We complete this definition by the following resource signature res:

$$\mathit{res}(\mathit{File}) = 1, tt \tag{3.1}$$
$$\mathit{res}(\text{any other foreign value}) = \bot \tag{3.2}$$

Informally, our Erlang program will therefore be robust to Denial-of-Service if no external manipulation may cause the opening of more than one file.

In our example, both services `log_fragile` and `log_robust`, once encoded, define non-exhausting processes. By examining the encoding of `log_fragile` and `log_robust`, however, we may notice, again, that it is easy to trick `log_fragile` into opening any number of files, hence making it non-robust, while `log_robust` may only open one file at any time. We will now formalise and mechanise this reasoning and introduce a proof system to prove stronger properties.

Chapter 4

Proving resource-safety

In this section, we introduce a type system designed to produce proofs of robustness of Erlang programs compiled in Teπc. As the encoding itself, this type system is parametrised by the set of resources.

4.1 Resource usage patterns

This type system is designed to recognise a number of orthogonal patterns: resource allocation, concurrent composition, exclusive choice, resource finalisation, resource transmission and loss of control.

The first pattern is resource allocation. An instruction $\texttt{foreign}\ a = E()\ \texttt{in}\ i$ requires more resources to be execute than i, as it needs to store the result of foreign operation $E()$ – regardless of whether these resources are eventually deallocated. The second pattern is concurrent composition. If two processes P and Q must be executed purely concurrently (i.e. without synchronisations), the system must have enough resources to permit any scheduling of P and Q. By opposition, if only one of two processes P and Q is to be executed, the system only needs enough resources to execute this process. Finalisation of resources permits safely reusing deallocated resources: in a process such as $P \triangleq \texttt{foreign}\ a = \mathit{Finalize}(b)\ \texttt{in}\ i \mid \texttt{foreign}\ a' = \mathit{Finalize}(b)\ \texttt{in}\ j$, both i and j are triggered only after the deallocation of resources allocated to b. The resources allocated to a must, however, be shared between i and j. Another pattern is that of runtime resource transmission: the amount of resources required for the execution of a service depends on what happens whenever the service is invoked, as well as on how many times the service is indeed invoked. In our analysis, this translates to a static *cost* in resources, charged to callers whenever they invoke the service. Finally, loss of control permits interaction with untyped processes. Informally, if a channel a may be used to communicate with the outside world, the typed process shouldn't expect any resource transfer on a, shouldn't make any assumption about references received from a, and shouldn't expect that references sent on a will be properly used. In other words, a should be considered "untrusted".

By searching for these patterns, the type system extracts a set of linear inequations upon the set of resources being manipulated.

4.2 Type judgements

Type judgements are expressed with the following grammar:

$$\begin{array}{lcll} T & ::= & \mathit{Bound}(r,\lambda) & r \in \mathcal{S}, \lambda : \textsc{Weak} \longrightarrow \mathcal{S} \\ N & ::= & Name(C,r) & r \in \mathcal{S} \\ C & ::= & Chan(N,g,d) & g,d \in \mathcal{S} \\ & | & Finalizer \ \mid \ Ssh \ \mid \ Unknown & \\ A & ::= & \mathit{Allocation}(r) & \end{array}$$

Judgement $\Gamma \vdash P : \mathit{Bound}(r,\lambda)$ states that, in environment Γ, P can be evaluated as a process which may be executed without starvation using only resources included in r and may have reused resources of external entities as specified by λ. Judgement $\Gamma \vdash a : Name(C,r)$ states that, according to Γ, a is the name of an entity using resource r, with role C. If C is $Chan(N,g,d)$, a is a communication channel, which can be used to communicate names of type N, to transfer resource g from the sender to the receiver, d of which can be reused by the receiver after deallocation. Conversely, if C is $Finalizer$, a has been declared as `foreign` $a = Finalize(b)$ `in` i and should not be used for communication, if C is $\mathit{Unknown}$, a is untrusted and may serve to communicate with the untyped world, and if C is Ssh, a may not be used to communicate at all. Finally, $\Gamma \vdash e : \mathit{Allocation}(r)$ states that, according to Γ, e is a foreign expression, and, assuming that the evaluation succeeds, the result of the evaluation of e will require at most r resources.

The set of rules defining the type system is presented in appendix D.2.

While this type system we present deals only with monadic Teπc and does not take into account n-ary operators, it may easily be extended to do so, at the expense of readability. In the rest of this document, we assume such an extension.

4.3 Results

Lemma 1 (Embedded Subject Reduction)
If $\Gamma \vdash e : \mathit{Allocation}(r_e)$ and if for some a and v, we have $a = v \Vdash e \rightsquigarrow f$ then $\Gamma \vdash f : \mathit{Allocation}(r_f)$ and $r_f \preceq r_e$. If $\Gamma \vdash e : \mathit{Allocation}(r_e)$ and $e \multimap v$ then $res(v) \preceq r_e$.

With the definition of relations $\cdot \Vdash \cdot \rightsquigarrow \cdot$ and $\cdot \multimap \cdot$ used in our encoding, this lemma is trivial. It is, of course, possible to find other relations which make this lemma false.

Lemma 2 (Weakening)
If $\Gamma \vdash P : Bound(t, \lambda)$ and if $(t', \lambda') \succeq (t, \lambda)$ then $\Gamma \vdash P : Bound(t', \lambda')$.

The lemma is standard and uses standard proofs.

Definition 6 (Isolation) *An environment Γ is said to* isolate *Q if and only if for any name a in $fn(Q)$, $\Gamma(a) = Name(Unknown, \_)$.*

In other words, Q is isolated if the hypothesis used by the type system correctly labels any communication with Q world as untrusted.

Informal theorem 1 (Subject Reduction under attack) *If P is a process, Q a client Γ an environment isolating Q and such that $\Gamma \vdash P : Bound(r, \lambda)$, if there is some R such that $P|Q \longrightarrow R$ then we may find two processes P' and Q' and a set of names, types and values $a_1 : N_1 = v_1, \ldots, a_n : N_n, v_n$ such that*

- $R \equiv (\nu \overrightarrow{a = v})(P' \mid Q')$
- $\Gamma, \overrightarrow{a : N} \vdash P' : Bound(t', \lambda')$
- $\Gamma, \overrightarrow{a : N}$ *isolates* Q'
- $\Sigma_{b \in \mathcal{N}} \lambda'(b) \preceq \Sigma_{b \in \mathcal{N}} \lambda(b)$
- $t' \oplus \Sigma_{b \in \mathcal{N}} \lambda'(b) \preceq t \oplus \Sigma_{b \in \mathcal{N}} \lambda(b)$
- $\forall i \in 1..n$, *if* $N_i = Name(\_, r_i)$ *then* $\lambda'(a_i) \preceq r_i$
- Q' *is a client*

In particular, if $Q =$ `nothing`, it is possible to find P' such that $\overrightarrow{a : N = v} = \emptyset$ and $Q' =$ `nothing`. (Subject Reduction without attack). If $\lambda = \bot_{\mathcal{N}}$, we always have $\lambda' = \bot_{\mathcal{N}}$ and $t' \preceq t$ (Closed Subject Reduction).

We prove this theorem by examining transitions internal to P, transitions internal to Q and communications between P and Q, taking advantage of existing type information $\overrightarrow{a : N = v}$ to provide a new type derivation.

Informal theorem 2 (Well-typed terms behave)
If $\Gamma \vdash P : Bound(reserve, \bot_{\mathcal{N}})$ then P is non-exhausting.

Informal theorem 3 (Well-typed programs are robust)
Let us consider an Erlang program e and exe_e its executable. If Γ isolates exe_e and $\Gamma \vdash exe_e : Bound(reserve, \bot_{\mathcal{N}})$, then e is robust.

Both theorems are corollaries of theorem 1. Note that the compositionality of this type system makes it possible to type separately the library and the encoding of e. However, the absence of polymorphism limits the usability of this compositionality.

Informal theorem 4 (Inference)
For any set of resources such that (constructive) satisfiability of sets of linear inequations is decidable, type inference is decidable.

This may be proved by induction on the structure of a term, by building a set of systems of linear inequations on resources such that the term may be typed if and only if one of the systems has a solution. Types of the term may then be deduced from the solutions of the system.

This theorem states that, for simple sets of resource – including the set used here as an example and, indeed, all the sets of resources we have used to this day, including amounts of memory or hard-drive, file handlers, cpus or secrets – we may derive a an algorithm for automatic analysis from this type system.

Corollaire 1 (Examples) *It is easy to prove that the encoding of* `log_fragile` *may never be typed in an isolating environment. Informally, an isolating environment requires that name log_fragile may not charge resources to callers – as these callers are not trusted to deliver the resources – while message* `open` *requires one file handler at each call. As there is no synchronisation between instances of this service, it is not sure that file handlers may be finalised and reused between instances.*

Conversely, typing the encoding of `log_robust` *is possible in an isolating environment and yields a proof that the service will use at most one file handler, no matter how many instances are being executed. Informally, these instances synchronise through the shared lock and may therefore finalise and reuse one common file handler.*

Note that it is easy to write more complex type-checked resource-bounded loops, for more complex sets of resources. For instance, it is quite possible to have several services share resources, as long as they have some manner of synchronising upon their common heap of resources. For more examples, see [18].

Chapter 5

Conclusions

5.1 Bottom line

This work is part of an ongoing work on the application of process calculi to operating system-level problems. Using Teπc, we have formalised the semantics of a large fragment of Core Erlang, as well as the usage of resources by an Erlang program, a notion of resource exhaustion and of robustness with respect to Denial-of-Service attacks. We have then produced a proof technique to guarantee that an Erlang program and run-time environment is robust, and applied this technique to a few simple examples.

Our encoding of Core Erlang is not complete, in particular with respect to error-handling, physical locations or migrations. In particular, the examples used throughout this document made use of a simplified version of some library functions, so as to avoid error management. Similarly, it is difficult to impose a static type system to a dynamically typed (indeed, in some circumstances, sometimes weakly typed) programming language. Our type system is itself therefore limited, in particular with respect to polymorphism, pattern-matching or process-to-process communication. However, we believe that our work is promising, and that it should be possible to extend it to a useful subset of Erlang – or other languages for concurrency and distribution. Indeed, we are currently working on a more generic type system for Teπc, with polymorphism and dependent types, which we hope will be able to remove most of the current restrictions and to extend the degree of control to other aspects of resource-management, and on the integration of real-time constraints in Teπc, which we hope will be applicable to Erlang programs.

5.2 Related works

Teπc-Erlang is directly related to π-Erlang [15], which it extends. Other attempts have been made to formalise Erlang, using operational semantics rather than encoding [8, 5]. While these last works are more complete, they are also more self-contained and harder to extend in a convincing manner to model resource usage or non-well-behaved agents. Other type systems have also been offered for Erlang [12, 13, 16] but with objectives distant from resource management and have, to the best of our knowledge, no proof of subject reduction.

A few languages and models have been designed to permit control of resources. Camelot [9] is a variant of ML with safe explicit deallocation of memory, in which the type system is able to infer bounds on memory usage. While this type system may express more precise bounds than ours, it is limited to memory and the language is strictly non-concurrent. Similarly, the ULM model [3] permits the description of systems from the point of view of their resource consumption, but in a strongly synchronous model, which prevents modelling any form of multitasking and without any management of resource deallocation. Finally, some works on process algebras [1][20][10]or on the λ-calculus [11] attempt to provide a formalisation of resource control, but either require much more abstract settings or fail to take into account concurrency or deallocation of resources. In particular, none of these languages or models provides a formalisation of Denial of Service attacks or a notion of resisting to such attacks.

Teπc itself is also related to the Applied π-calculus [6]. While both formalism is intended for proofs of protocols, its design makes it more adapted to the examination of cryptography and less to the examination of resource-related properties, in particular resource limitations, which are at the core of our work.

5.3 Future works

Teπc is a work in progress. In addition to completing the encoding of Teπc-Erlang and to improving the type system and adding real-time constraints, we intend to work on the extraction of Teπc into Erlang, with guarantees of preservation of the semantics and types of programs. We also intend to merge our work on resources, foreign operations and foreign values to the ongoing development of the Kell platform [17], a process algebra/language/virtual machine designed for the construction of component-based distributed applications.

Bibliography

[1] F. Barbanera, M. Bugliesi, M. Dezani, and V. Sassone. A calculus of bounded capacities. In *Proceedings of Advances in Computing Science, 9th Asian Computing Science Conference, ASIAN'03*, volume 2896 of *Lecture Notes in Computer Science*. Springer, 2003.

[2] J. Barklund and R. Virding. Erlang 4.7.3 reference manual, 1999.

[3] G. Boudol. Ulm: a core programming model for global computing. In *ESOP 04*, 2004.

[4] R. Carlsson. An introduction to core erlang. In *Proceedings of the PLI'01 Erlang Workshop*, 2001.

[5] K. Claessen and H. Svensson. A semantics for distributed erlang. In *ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 78–87, New York, NY, USA, 2005. ACM Press.

[6] C. Fournet and M. Abadi. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, 2001.

[7] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory*. Springer-Verlag, 1996.

[8] A. Fredlund. *A Framework for Reasoning about ERLANG*. PhD thesis, Royal Institute of Technology, 2001.

[9] M. Hofmann. A type system for bounded space and functional in-place update–extended abstract. *Nordic Journal of Computing*, 7(4), Autumn 2000. An earlier version appeared in ESOP2000.

[10] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2001.

[11] A. Igarashi and N. Kobayashi. Resource usage analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2002.

[12] T. Lindahl and K. Sagonas. Typer: a type annotator of erlang code. In *ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 17–25, New York, NY, USA, 2005. ACM Press.

[13] S. Marlow and P. Wadler. A practical subtyping system for erlang. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 136–149, New York, NY, USA, 1997. ACM Press.

[14] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. Technical Report -86, Penn State University, 1989.

[15] T. Noll and C. Roy. Modeling erlang in the pi-calculus. In *Proceedings of the ACM SIGPLAN 2005 Erlang Workshop (Erlang '05)*, pages 72–77. ACM, 2005.

[16] S.-O. Nystr&#246;m. A soft-typing system for erlang. In *ERLANG '03: Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, pages 56–71, New York, NY, USA, 2003. ACM Press.

[17] A. Schmitt and J.-B. Stefani. Towards a Calculus for Distributed Components. In *5th International Symposium on Formal Methods for Components and Objects (FMCO 2006)*, Amsterdam, Netherlands, Nov. 2006.

[18] D. Teller. Resources, garbage-collection and the pi-calculus. In *International Colloquium on Automata, Languages and Programming (submitted)*, January 2006.

[19] D. Teller. Tepic: A targettable, extendable pi-calculus. Technical Report (in progress), University of Sussex, 2006.

[20] D. Teller, P. Zimmer, and D. Hirschkoff. Using Ambients to Control Resources. In *Proceedings of the 13th International Conference on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[21] P. Wojciechowski and P. Sewell. Nomadic pict: Language and infrastructure design for mobile agents. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.

Appendix A

Notes to the introduction

In Erlang/OTP, the definigion of `log_robust` could be approximated by the following extract :

```
log_robust(Lock, Item) ->
    lock_acquire(Lock),
    log_fragile(Item),
    lock_release(Lock).

start() ->
    Lock = new_lock (),
    fun (Item) -> log_robust(Lock, Item) end.
```

Appendix B

Language

B.1 Substitution and elimination of dead instructions

The substitution relation of the traditional π-calculus is extended to support elimination of dead instructions. Intuitively, an instruction is considered dead if it has the form $\circledcirc\ u$ (sending a message on $\circledcirc$), `on` $\circledcirc x$ `do` i or `once` $\circledcirc x$ `do` i (receiving a message on $\circledcirc$).

Note that the substitution of $\circledcirc$ to a name may cause the elimination of a dead instruction, while the substitution of $\circledcirc$ to a variable will generally be considered unsafe. This is a design choice, as the second kind of substitution maps to the unsafe passing of `null` references/pointers in numerous general-purpose programming languages.

$$
\begin{array}{lcll}
((\nu a = v)P)\{m \leftarrow n\} & = & (\nu a = v\{m \leftarrow n\})(P\{m \leftarrow n\}) & a \notin \{m,n\} \\
(P|Q)\{m \leftarrow n\} & = & (P\{m \leftarrow n\}) \mid (Q\{m \leftarrow n\}) & \\
\texttt{nothing}\{m \leftarrow n\} & = & \texttt{nothing} & \\
(\texttt{foreign}\ u = e\ \texttt{in}\ i)\{m \leftarrow n\} & = & \texttt{foreign}\ u = e\{m \leftarrow n\}\ \texttt{in}\ (i\{m \leftarrow n\}) & u \notin \{m,n\} \\
(\texttt{spawn}\ i\ \texttt{and}\ j)\{m \leftarrow n\} & = & \texttt{spawn}\ (i\{m \leftarrow n\})\ \texttt{and}\ (j\{m \leftarrow n\}) & \\
(\texttt{either}\ i\ \texttt{or}\ j)\{m \leftarrow n\} & = & \texttt{either}\ (i\{m \leftarrow n\})\ \texttt{or}\ (j\{m \leftarrow n\}) & \\
(\texttt{on}\ a\ x\ \texttt{do}\ i)\{a \leftarrow \circledcirc\} & = & \texttt{nothing} & \\
(\texttt{on}\ m'\ x\ \texttt{do}\ i)\{m \leftarrow n\} & = & \texttt{on}\ m'\{m \leftarrow n\}\ x\ \texttt{do}\ i\{m \leftarrow n\} & \text{otherwise if } x \notin \{m,n\} \\
(\texttt{once}\ a\ x\ \texttt{do}\ i)\{a \leftarrow \circledcirc\} & = & \texttt{nothing} & \\
(\texttt{once}\ m'\ x\ \texttt{do}\ i)\{m \leftarrow n\} & = & \texttt{once}\ m'\{m \leftarrow n\}\ x\ \texttt{do}\ i\{m \leftarrow n\} & \text{otherwise if } x \notin \{m,n\} \\
(a\ n')\{a \leftarrow \circledcirc\} & = & \texttt{nothing} & \\
(m'\ n')\{m \leftarrow n\} & = & (m'\{m \leftarrow n\})\ (n'\{m \leftarrow n\}) & \text{otherwise} \\
a\{a \leftarrow b\} & = & b & \\
a\{m \leftarrow n\} & = & a & a \neq m \\
a\{a \leftarrow \circledcirc\} & = & \circledcirc & \\
x\{m \leftarrow n\} & = & x & x \neq m \\
x\{x \leftarrow n\} & = & n & \\
\circledcirc\{m \leftarrow n\} & = & \circledcirc & \\
E(\overrightarrow{v})\{m \leftarrow n\} & = & E(\overrightarrow{v\{m \leftarrow n\}}) &
\end{array}
$$

B.2 Labelled Transition Semantics

The semantics of Teπc relies on a structural equivalence, the smallest equivalence law $\equiv$ such that $|$ and $+$ are commutative and associative, compatible with α conversion of bound names and variables and with the following rules.

$$\textsc{E-Commut}\ (\nu a = v)(\nu b = w)P \equiv (\nu b = w)(\nu a = v)P \qquad a \neq b, a \notin fn(w), b \notin fn(v)$$

$$\textsc{E-Scope}\ ((\nu a = v)P)|Q \equiv (\nu a = v)(P|Q) \qquad a \notin fr(Q) \qquad\qquad \textsc{E-Par}\ \frac{P \equiv Q}{P|R \equiv Q|R}$$

$$\textsc{E-New}\ \frac{P \equiv Q}{(\nu a = v)P \equiv (\nu a = v)Q}$$

While most rules of both structural equivalence and labelled transitions are essentially identical to those of the original π-calculus, resource management is more detailed. Indeed, the labelled transition presemantics of Teπc as the smallest relation $\stackrel{\cdot}{\longrightarrow}_{pre}$ compatible with the following rules

$$\text{R-PAR}\ \frac{P \xrightarrow{\alpha}_{pre} P'}{P|Q \xrightarrow{\alpha}_{pre} P'|Q} \qquad \text{R-CHOICE}\ \frac{P \xrightarrow{\alpha}_{pre} P'}{P+Q \xrightarrow{\alpha}_{pre} P'}$$

$$\text{R-COMM}\ \frac{P \xrightarrow{a?m}_{pre} P' \qquad Q \xrightarrow{a!m}_{pre} Q'}{P|Q \longrightarrow_{pre} P'|Q'}$$

$$\text{R-COMM-CLOSE}\ \frac{P \xrightarrow{a?b}_{pre} P' \qquad Q \xrightarrow{\nu b=v.a!b}_{pre} Q'}{P|Q \longrightarrow_{pre} (\nu b = v)(P'|Q')}$$

$$\text{R-HIDE}\ \frac{P \xrightarrow{\alpha}_{pre} P'}{(\nu a = v)P \xrightarrow{\alpha}_{pre} (\nu a = v)P'}\ a \notin fr(\alpha)$$

$$\text{R-OPEN}\ \frac{P \xrightarrow{b!a}_{pre} P'}{(\nu a = v)P \xrightarrow{\nu a=v.b!a}_{pre} P'}\ b \neq a$$

$$\text{R-EQUIV}\ \frac{P \equiv P' \qquad Q' \equiv Q \qquad P' \xrightarrow{\alpha}_{pre} Q'}{P \xrightarrow{\alpha}_{pre} Q} \qquad \begin{array}{l}\text{R-SPAWN}\\ \texttt{spawn}\ i\ \texttt{and}\ j \xrightarrow{\tau}_{pre} i|j\end{array}$$

$$\begin{array}{l}\text{R-EITHER}\\ \texttt{either}\ i\ \texttt{or}\ j \xrightarrow{\tau}_{pre} i+j\end{array} \qquad \text{R-SEND}\ a\ m \xrightarrow{a!m}_{pre} \texttt{nothing}$$

$$\text{R-ONCE}\ \texttt{once}\ a\ x\ \texttt{do}\ i \xrightarrow{a?m}_{pre} i\{x \leftarrow m\}$$

$$\text{R-ON}\ \texttt{on}\ a\ x\ \texttt{do}\ i \xrightarrow{a?m}_{pre} i\{x \leftarrow m\} \mid \texttt{on}\ a\ x\ \texttt{do}\ i$$

$$\text{R-DEALLOCATE}\ \frac{}{(\nu a = v)P \xrightarrow{\tau}_{pre} P\{a \leftarrow \circledcirc\}} \qquad a = v \vDash P$$

$$\text{R-EVALUATE}\ \frac{}{\texttt{foreign}\ u = e\ \texttt{in}\ i \xrightarrow{e \rightsquigarrow f}_{pre} \texttt{foreign}\ u = f\ \texttt{in}\ i}$$

$$\text{R-FETCH}\ \frac{P \xrightarrow{e \rightsquigarrow f}_{pre} P'}{(\nu a = v)P \xrightarrow{\tau}_{pre} (\nu a = v)P'}\ a = v \Vdash e \rightsquigarrow f \qquad fv(e) = fv(f) = \emptyset$$

$$\text{R-DEREF}\ \frac{}{\texttt{foreign}\ x = e\ \texttt{in}\ i \xrightarrow{\tau}_{pre} i\{x \leftarrow b\}} \qquad e \multimap b \qquad fv(e) = \emptyset$$

$$\text{R-STORE}\ \frac{}{\texttt{foreign}\ a = e\ \texttt{in}\ i \xrightarrow{\tau}_{pre} (\nu a = v)i}\ e \multimap v \qquad fv(e) = \emptyset$$

Appendix C

From Erlang to Teπc

All the extracts involved in this encoding use Haskell/Python-style bi-dimensional syntax, i.e. tabulations are part of the syntax and influence syntactic priority of operators, as do parenthesis in most languages.

In the following, we will use v for Erlang variables, f for Erlang function names, d for Erlang function definitions, z for Erlang numbers, a for Erlang atoms (i.e. names), c for Erlang characters, s for Erlang strings, e for Erlang expressions and p for Erlang patterns. Finally, we assume the existence of a name $true$ referencing depositary $True$.

C.1 Semantics of embedded primitives

$$E() \multimap E() \tag{C.1}$$

$$a \multimap a \tag{C.2}$$

$$\circledcirc \multimap \circledcirc \tag{C.3}$$

$$Finalize(\circledcirc) \multimap \circledcirc \tag{C.4}$$

$$Free(\circledcirc) \multimap \circledcirc \tag{C.5}$$

$$Terminate(\circledcirc) \multimap \circledcirc \tag{C.6}$$

$$l = Cons(h,t) \Vdash Head(l) \rightsquigarrow h \tag{C.7}$$

$$l = Cons(h,t) \Vdash Tail(l) \rightsquigarrow t \tag{C.8}$$

$$t = Tuple_n(a_1, a_2, \ldots, a_n) \Vdash Nth_p(t) \rightsquigarrow a_p \qquad 1 \leq p \leq n \tag{C.9}$$

$$Same(a,a) \multimap True \tag{C.10}$$

$$Same(a,b) \multimap False \qquad a \neq b \tag{C.11}$$

$$IfThenElse(True, a, b) \multimap a \tag{C.12}$$

$$IfThenElse(False, a, b) \multimap b \tag{C.13}$$

$$\frac{a = v \Vdash e \rightsquigarrow f}{a = v \Vdash IfThenElse(e,b,c) \rightsquigarrow IfThenElse(f,b,c)} \tag{C.14}$$

$$\tag{C.15}$$

Rule C.4 specifies the behaviour of finalisation: if a is not free in i, `foreign` $a = Finalise(b)$ `in` i will wait until b has been garbage-collected before executing i. Constructors $Free$ and $Terminate$ have the same definition but will be used in a different manner later. The other rules model trivial depositaries, usual data structures (tuples and lists) and simple flow control statements.

C.2 Expressions

This encoding uses reserved names $number_z$ (for any number z), $char_c$ (for any character c), $string_s$ (for any string s), nil and $cons$, all of which are defined in the library, in Section C.4.

$$
\begin{array}{lcl}
\mathcal{M}\llbracket \mathtt{module}\ a[\mathtt{f}_1,\ldots,\mathtt{f}_\mathtt{p}]fd_1\ldots fd_n\rrbracket & = & \mathtt{new}\ \overrightarrow{b\in(\bigcup_i fn(fd_i))\backslash\{f_i\}}\ \mathtt{in\ spawn}\ \mathcal{F}\llbracket fd_1\rrbracket\ \mathtt{and}\ldots\mathtt{and}\ \mathcal{F}\llbracket fd_n\rrbracket \\
\mathcal{F}\llbracket f=\mathtt{fun}(v_1,\ldots,v_n)- >e\rrbracket & = & \mathtt{on}\ f\ v_1\ v_2\ \ldots\ v_n\ res\ \mathtt{do}\ \mathcal{E}\llbracket e\rrbracket^{res,self} \\
\mathcal{E}\llbracket v\rrbracket^{res,self} & = & res\ v \\
\mathcal{E}\llbracket f\rrbracket^{res,self} & = & res\ f \\
\mathcal{E}\llbracket z\rrbracket^{res,self} & = & number_z\ res\ self \\
\mathcal{E}\llbracket a\rrbracket^{res,self} & = & res\ a \\
\mathcal{E}\llbracket c\rrbracket^{res,self} & = & char_c\ res\ self \\
\mathcal{E}\llbracket s\rrbracket^{res,self} & = & string_s\ res\ self \\
\mathcal{E}\llbracket []\rrbracket^{res,self} & = & nil\ res\ self \\
\mathcal{E}\llbracket [e_1|e_2]\rrbracket^{res,self} & = & \mathtt{new}\ res_1, res_2\ \mathtt{in} \\
 & & \mathtt{spawn}\ \mathcal{E}\llbracket e_1\rrbracket^{res_1,self}\ \mathtt{and}\ \mathcal{E}\llbracket e_2\rrbracket^{res_2,self} \\
 & & \ \mathtt{and\ once}\ res_1\ x_1\ \mathtt{do\ once}\ res_2\ x_2\ \mathtt{do}\ cons\ x_1\ x_2\ res\ self \\
\mathcal{E}\llbracket \{e_1,\ldots,e_n\}\rrbracket^{res,self} & = & \mathtt{new}\ res_1, res_2\ldots, res_n\ \mathtt{in} \\
 & & \quad \mathtt{spawn}\ \mathcal{E}\llbracket e_1\rrbracket^{res_1,self}\ \mathtt{and}\ \ldots\ \mathtt{and}\ \mathcal{E}\llbracket e_n\rrbracket^{res_n,self} \\
 & & \quad \mathtt{and\ once}\ res_1\ x_1\ \mathtt{do}\ \ldots\mathtt{once}\ res_n\ x_n\ \mathtt{do}\ tuple_n\ x_1\ \ldots\ x_n\ res\ self \\
\mathcal{E}\llbracket \mathtt{foreign}\ x=e_1\ \mathtt{in}\ e_2\rrbracket^{res,self} & = & \mathtt{new}\ res'\ \mathtt{in\ spawn}\ \mathcal{E}\llbracket e_1\rrbracket^{res',self}\ \mathtt{and\ once}\ res'\ x\ \mathtt{do}\ \mathcal{E}\llbracket e_2\rrbracket^{res,self} \\
\mathcal{E}\llbracket \mathtt{apply}\ f(e_1,\ldots,e_n)\rrbracket^{res,self} & = & \mathtt{new}\ res_1, res_2,\ldots, res_n\ \mathtt{in} \\
 & & \quad \mathtt{spawn}\ \mathcal{E}\llbracket e_1\rrbracket^{res_1,self}\ \mathtt{and}\ \ldots\ \mathtt{and}\ \mathcal{E}\llbracket e_n\rrbracket^{res_n,self} \\
 & & \quad \mathtt{and\ once}\ res_1\ x_1\ \mathtt{do}\ \ldots\mathtt{once}\ res_n\ x_n\ \mathtt{do}\ f\ x_1\ \ldots\ x_n\ res\ self \\
\mathcal{E}\llbracket \mathtt{letrec} f_1\ d_1\ \ f_2\ d_2\ \ \ldots f_n\ d_n\ \mathtt{in}\ e\rrbracket^{res,self} & & \\
 & = & \mathtt{new}\ f_1,\ldots,f_n\ \mathtt{in} \\
 & & \mathtt{spawn}\ \mathcal{F}\llbracket f_1 d_1\rrbracket\ \mathtt{and}\ \ldots\ \mathtt{and}\ \mathcal{F}\llbracket f_n d_n\rrbracket\ \mathtt{and}\ \mathcal{E}\llbracket e\rrbracket^{res,self} \\
\mathcal{E}\llbracket \mathtt{case}\ e\ \mathtt{of}\ c_1\ldots c_n\rrbracket^{res,self} & = & \mathtt{new}\ test_1, go_1, success_1, test_2, go_2, success_2\ldots, test_n, go_n, success_n\ \mathtt{in} \\
 & & \ \mathtt{new}\ res'\ \mathtt{in} \\
 & & \quad \mathtt{spawn}\ \mathcal{C}\llbracket c_1\rrbracket^{res,self,test_i,success_i,go_i}\ \mathtt{and}\ \ldots \\
 & & \qquad \mathtt{and}\ \mathcal{C}\llbracket c_1\rrbracket^{res,self,test_i,success_i,go_i} \\
 & & \quad \mathtt{and}\ \mathcal{E}\llbracket e\rrbracket^{res',self} \\
 & & \quad \mathtt{and\ once}\ res'\ x\ \mathtt{do}\ (\mathtt{spawn}\ test_1\ x\ \mathtt{and}\ \ldots test_n\ x) \\
 & & \quad \mathtt{and} \\
 & & \quad\ \mathtt{either\ once}\ success_1\ \mathcal{V}\llbracket c_1\rrbracket\ \mathtt{do}\ go_1\ \overrightarrow{x_1} \\
 & & \quad\ \ldots \\
 & & \quad\ \mathtt{or\ once}\ success_n\ \mathcal{V}\llbracket c_n\rrbracket\ \mathtt{do}\ go_n\ \overrightarrow{x_n} \\
\mathcal{E}\llbracket \mathtt{receive}\ c_1\ldots c_n\ \mathtt{after}\ e_1- >e_2\rrbracket^{res,self} & & \\
 & = & \mathtt{once}\ self\ x\ \mathtt{do}\ \mathcal{E}\llbracket \mathtt{case}\ e\ \mathtt{of}\ c_1\ldots c_n\ _- >e_2\rrbracket^{res,self}
\end{array}
$$

C.3 Pattern-matching

This encoding uses reserved names $ifsamethenelse$, nth_i (for all integers i) and $uncons$, all of which are defined in the library, in Section C.4.

$$
\begin{array}{lcl}
\mathcal{C}\,[\![p \text{ when } e_1 - > e_2]\!]^{res,self,in,success,go} & & \\
 & = & \texttt{new } res', res'', structure \texttt{ in once } in\ x \texttt{ do} \\
 & & \quad \texttt{spawn } \mathcal{P}\,[\![p]\!]^{x,structure} \texttt{ and once } structure\ \overrightarrow{y} \texttt{ do} \\
 & & \quad\ \texttt{spawn } \mathcal{E}\,[\![e_1]\!]^{res',self} \texttt{ and once } res'\ z \texttt{ do } ifthenelse\ z\ success\ \circledcirc\ res''\ self \\
 & & \quad\ \texttt{and once } res'' \texttt{ do spawn } next\ \overrightarrow{z} \texttt{ and once } go\ \overrightarrow{x} \texttt{ do } \mathcal{E}\,[\![e_2]\!]^{res,self} \\
 & & \qquad\qquad (\text{where } \overrightarrow{y} = \overrightarrow{x} = \mathcal{V}\,[\![p]\!]) \\
\mathcal{P}\,[\![p_1, p_2, \ldots, p_n]\!]^{in,out} & = & \texttt{new } out_1, out_2 \ldots, out_n \texttt{ in} \\
 & & \texttt{spawn new } res_1 \texttt{ in} \\
 & & \quad \texttt{spawn } nth_1\ in\ res_1\ self \texttt{ and} \\
 & & \quad \texttt{once } res_1\ elem_1 \texttt{ do } \mathcal{P}\,[\![p_1]\!]^{elem_1,out_1} \\
 & & \quad \texttt{and } \ldots \\
 & & \qquad \texttt{and new } res_n \texttt{ in} \\
 & & \qquad\quad \texttt{spawn } nth_n\ in\ res_n\ self \texttt{ and} \\
 & & \qquad\quad \texttt{once } res_n\ elem_n \texttt{ do } \mathcal{P}\,[\![p_n]\!]^{elem_n,out_n} \\
 & & \texttt{and once } out_1\ \overrightarrow{x_1} \texttt{ do } \ldots \texttt{once } out_n\ \overrightarrow{x_n} \texttt{ do } out\ \overrightarrow{x}\overrightarrow{x_2}\cdots\overrightarrow{x_n} \\
\mathcal{P}\,[\![a]\!]^{in,out} & = & ifsamethenelse\ a\ in\ out\ \circledcirc \\
\mathcal{P}\,[\![v]\!]^{in,out} & = & out\ x \\
\mathcal{P}\,[\![[p_1, p_2, \ldots, p_n | p_{n+1}]]\!]^{in,out} & & \\
 & = & \texttt{new } out_1, out_2 \ldots, out_{n+1} \texttt{ in} \\
 & & \texttt{spawn} \\
 & & \quad \texttt{new } hd_1, tl_1 \texttt{ in spawn } uncons\ in\ hd_1\ tl_1 \texttt{ and once } hd_1\ h_1 \texttt{ do } \mathcal{P}\,[\![p_1]\!]^{h_1,out_1} \\
 & & \quad\ \texttt{and once } tl_1\ t_1 \texttt{ do new } hd_2, tl_2 \texttt{ in} \\
 & & \qquad \texttt{spawn } uncons\ t_1\ hd_2\ tl_2 \texttt{ and once } hd_2\ h_2 \texttt{ do } \mathcal{P}\,[\![p_2]\!]^{head_2,out_2} \\
 & & \qquad\ \ldots \\
 & & \qquad\quad \texttt{spawn } uncons\ t_n\ hd_{n+1}\ tl_{n+1} \texttt{ and once } hd_{n+1}\ h_{n+1} \texttt{ do } \mathcal{P}\,[\![p_{n+1}]\!]^{h_{n+1},out_{n+1}} \\
 & & \texttt{and once } out_1\ \overrightarrow{x_1} \texttt{ do } \ldots \texttt{once } out_{n+1}\ \overrightarrow{x_{n+1}} \texttt{ do } out\ \overrightarrow{x_1}\overrightarrow{x_2}\cdots\overrightarrow{x_{n+1}} \\
\mathcal{V}\,[\![p_1, p_2, \ldots, p_n]\!] & = & \mathcal{V}\,[\![p_1]\!]\,\mathcal{V}\,[\![p_2]\!] \ldots \mathcal{V}\,[\![p_n]\!] \\
\mathcal{V}\,[\![v]\!] & = & \{v\} \\
\mathcal{V}\,[\![a]\!] & = & \{\} \\
\mathcal{V}\,[\![[p_1 | p_2]]\!] & = & \mathcal{V}\,[\![p_1]\!]\,\mathcal{V}\,[\![p_2]\!]
\end{array}
$$

C.4 Library

This is an abriged version of the library.

• Message emission

on erlang : send x_1 x_2 $res, self$ do spawn res x_2 and x_1 x_2

• Identification of the current process

on erlang : self res $self$ do res $self$

• Process spawning

on erlang : spawn f arg res $self$ do new res' in foreign $self' = Process()$ in
  spawn f arg $self'$ res' and res $self'$ and once res' _ do nothing

• Exceptions/killing

on erlang : exit$_3$ $reason$ res $self$ do spawn $kill$ $self$ $reason$ and res $true$

on erlang : exit$_4$ pid $reason$ res $self$ do spawn $kill$ pid $reason$ and res $true$

on $kill$ pid $reason$ do new $do\_kill, trap\_exit$ in foreign $next = IfThenElse(False, trap\_exit, do\_kill)$ in
  spawn $next$ and either once do_kill do $terminate$ pid or once $trap\_exit$ do pid $reason$

• File management (simplified)

on $open$ $name$ res $self$ do foreign $file = File()$ in res $file$

on $close$ $name$ res $self$ do spawn $free$ $file$ and foreign $a = Finalize(file)$ in res $true$

on $write$ $name$ $content$ res $self$ do res $true$

• Waitlines (simplified)

on new_lock res $self$ do foreign $a = Lock()$ in spawn a and res a

on $acquire\_lock$ l res $self$ do once l do res $true$

on $release\_lock$ l res $self$ do spawn l and res $true$

• Primitive values

on $number_z$ res $self$ do foreign $a = Number_z()$ in res a

on $strings_s$ res $self$ do foreign $a = Strings_s()$ in res a

on $char_c$ res $self$ do foreign $a = Char_c()$ in res a

on nil res $self$ do foreign $a = Nil()$ in res a

on $cons$ h t res $self$ do foreign $a = Cons(h, t)$ in res a

on $uncons$ x h t $self$ do foreign $a = Head(x)$ in foreign $b = Tail(x)$ in spawn h a and t b

on $tuple_n$ x_1 x_2 $\ldots$ x_n res $self$ do foreign $a = Tuple_n(x_1, \ldots, x_n)$ in res a

on nth_i x res $self$ do foreign $a = Nth_i(x)$ in res a

on $ifthenelse$ x y z res $self$ do foreign $next = IfThenElse(x, y, z)$ in res $next$

on $ifsamethenelse$ w x y z res $self$ do foreign $next = IfThenElse(Same(w, x), y, z)$ in res $next$

• Garbage-collection

on $free$ x do foreign $a = Free(x)$ in nothing

on $terminate$ x do foreign $a = Terminate(x)$ in nothing

$b = Process() \vDash$ foreign $c = Terminate(b)$ in $i \mid P$

$b = File() \vDash$ foreign $a = Free(b)$ in $i \mid P$

Appendix D

Proving resource-safety

D.1 Relations

If $\oplus$ is a function from $\mathcal{S}^2$ to $\mathcal{S}$ and $\preceq$ is a relation on $\mathcal{S}$, we extend the definition of both symbols to functions from $\mathcal{N}$ to $\mathcal{S}$ by

- $\forall \lambda, \mu \in \mathcal{S}^{\mathcal{N}}, \forall a \in \mathcal{N}, (\lambda \oplus \mu)(a) \triangleq \lambda(a) \oplus \mu(a)$
- $\forall \lambda, \mu \in \mathcal{S}^{\mathcal{N}}, \lambda \preceq \mu \iff \forall a \in \mathcal{N}, \lambda(a) \preceq \mu(a)$

We also extend the definition of $\oplus$ and $\preceq$ to pairs in $\mathcal{S} \times \mathcal{S}^{\mathcal{N}}$ by

- $(t, \lambda) \oplus u \mapsto r \triangleq (t, \lambda \oplus (u \mapsto r))$
- $(t, \lambda) \oplus \circledcirc \mapsto r \triangleq (t \oplus r, \lambda)$
- $(t, \lambda) \preceq (t', \lambda') \iff t \preceq \lambda \wedge \lambda \preceq \lambda'$.

D.2 Type system

Figures D.1 and D.2 present the type system. For the sake of readability, we slightly alter the syntax to allow writing `foreign` $a : N = e$ `in` $\cdots$ and $(\nu a : N = v)$. When necessary, we write $a \mapsto r$ for the function defined on $\mathcal{N}$ whose value is r for a and $\perp$ for everything else, $\lambda \backslash \{x\}$ for the function defined on $\mathcal{N}$ whose value on x is $\perp$ and identical to that of λ everywhere else, and $\perp_{\mathcal{N}}$ for the function defined on $\mathcal{N}$ whose value is $\perp$ everywhere. We extend the definition to always have $\lambda(\circledcirc) = \perp$.

Rule T-Nil states that the terminated process is always typable. Rule T-Repl permits the typing of `on` $\cdot$ `do` ... services, as long as they require no resources or that the resources are provided by the caller. Rules T-New and T-Op count the allocation of resources to a name a and the possible reuse of these resources by a process, while rule T-Finalize redistributes some of these resources to an instruction i. Rule T-Par adds the costs and resource reuse of

$$\textsc{T-Nil}\ \Gamma \vdash \texttt{nothing} : T \qquad \textsc{T-Repl}\ \frac{\Gamma \vdash \texttt{once}\ m\ x\ \texttt{do}\ i : \mathit{Bound}(\bot, \bot_{\mathcal{N}})}{\Gamma \vdash \texttt{on}\ m\ x\ \texttt{do}\ i : T}$$

$$\textsc{T-New}\ \frac{\Gamma, a : \mathit{Name}(C, r_a) \vdash P : \mathit{Bound}(t_P, \lambda)}{\Gamma \vdash (\nu a : \mathit{Name}(C, r_a) = v)P : \mathit{Bound}(t', \lambda')} \quad \begin{array}{ll} \mathit{res}(v) \preceq r_a & \lambda(a) \preceq r_a \\ \lambda' \succeq \lambda \backslash \{a\} & t' \succeq r_a \oplus t_P \end{array}$$

$$\textsc{T-Op}\ \frac{\begin{array}{c}\Gamma, a : N \vdash i : \mathit{Bound}(t, \lambda) \\ \Gamma \vdash e : \mathit{Allocation}(r_e)\end{array}}{\Gamma \vdash \texttt{foreign}\ a : N = e\ \texttt{in}\ i : \mathit{Bound}(t', \lambda')} \quad \begin{array}{ll} N = \mathit{Name}(\_, r_e) & \lambda(a) \preceq r_e \\ \lambda' \succeq \lambda \backslash \{a\} & t' \succeq r_e \oplus t \end{array}$$

$$\textsc{T-Par}\ \frac{\begin{array}{c}\Gamma \vdash P : \mathit{Bound}(t_P, \lambda_P) \\ \Gamma \vdash Q : \mathit{Bound}(t_Q, \lambda_Q)\end{array}}{\Gamma \vdash P|Q : \mathit{Bound}(t', \lambda')} \ t' \succeq t_P \oplus t_Q \qquad \lambda' \succeq \lambda_P \oplus \lambda_Q$$

$$\textsc{T-Spawn}\ \frac{\begin{array}{c}\Gamma \vdash i : \mathit{Bound}(t_i, \lambda_i) \\ \Gamma \vdash j : \mathit{Bound}(t_j, \lambda_j)\end{array}}{\Gamma \vdash \texttt{spawn}\ i\ \texttt{and}\ j : \mathit{Bound}(t', \lambda')} \ t' \succeq t_i \oplus t_j \qquad \lambda' \succeq \lambda_i \oplus \lambda_j$$

$$\textsc{T-Choice}\ \frac{\begin{array}{c}\Gamma \vdash P : \mathit{Bound}(t_P, \lambda_P) \\ \Gamma \vdash Q : \mathit{Bound}(t_Q, \lambda_Q)\end{array}}{\Gamma \vdash P + Q : \mathit{Bound}(t', \lambda')} \ t' \succeq t_P \qquad t' \succeq t_Q \qquad \lambda' \succeq \lambda_P \qquad \lambda' \succeq \lambda_Q$$

$$\textsc{T-Either}\ \frac{\begin{array}{c}\Gamma \vdash i : \mathit{Bound}(t_i, \lambda_i) \\ \Gamma \vdash j : \mathit{Bound}(t_j, \lambda_j)\end{array}}{\Gamma \vdash \texttt{either}\ i\ \texttt{or}\ j : \mathit{Bound}(t', \lambda')} \ t' \succeq t_i \qquad t' \succeq t_j \qquad \lambda' \succeq \lambda_i \qquad \lambda' \succeq \lambda_j$$

$$\textsc{T-Receive}\ \frac{\begin{array}{c}\Gamma \vdash m : \mathit{Name}(\mathit{Chan}(N, g, d), \_) \\ \Gamma, x : N \vdash i : \mathit{Bound}(t, \lambda)\end{array}}{\Gamma \vdash \texttt{once}\ m\ x\ \texttt{do}\ i : \mathit{Bound}(t', \lambda')} \ t' \oplus g \succeq t \qquad \lambda' \succeq \lambda \backslash \{x\} \qquad d \succeq \lambda(x)$$

$$\textsc{T-ReceiveUnknown}\ \frac{\begin{array}{c}\Gamma \vdash m : \mathit{Name}(\mathit{Unknown}, \_) \\ \Gamma, x : \mathit{Name}(\mathit{Unknown}, \_) \vdash i : \mathit{Bound}(t, \lambda)\end{array}}{\Gamma \vdash \texttt{once}\ m\ x\ \texttt{do}\ i : \mathit{Bound}(t', \lambda')} \ t' \succeq t \qquad \lambda' \succeq \lambda \qquad \lambda(x) = \bot$$

$$\textsc{T-Send}\ \frac{\Gamma \vdash m : \mathit{Name}(\mathit{Chan}(N, g, d), \_) \qquad \Gamma \vdash n : N}{\Gamma \vdash m\ n : \mathit{Bound}(t', \lambda')} \ (t', \lambda') \succeq (g, \bot_{\mathcal{N}}) \oplus n \mapsto d$$

$$\textsc{T-SendUnknown}\ \frac{\Gamma \vdash m : \mathit{Name}(\mathit{Unknown}, \_) \qquad \Gamma \vdash n : \mathit{Name}(\mathit{Unknown}, \_)}{\Gamma \vdash m\ n : T}$$

$$\textsc{T-Ref}\ \frac{\Gamma(a) = T}{\Gamma \vdash a : T} \qquad \textsc{T-Null}\ \frac{}{\Gamma \vdash \circledcirc : T}$$

Figure D.1: Type system for resource guarantees (non-foreign terms).

$$\textsc{T-Alloc-Value}\ \frac{}{\Gamma \vdash v : \mathit{Allocation}(r)}\ res(v) \preceq r$$

$$\textsc{T-Alloc-Nothing}\ \frac{}{\Gamma \vdash e : \mathit{Allocation}(r)}\ e \notin \textsc{Value},\ fn(e) \subseteq \Gamma$$

$$\textsc{T-Finalize}\ \frac{\Gamma, a : N \vdash i : \mathit{Bound}(t, \lambda)}{\Gamma \vdash \texttt{foreign}\ a : N = \mathit{Finalize}(m)\ \texttt{in}\ i : \mathit{Bound}(t', \lambda')}\ \begin{array}{l} N = Name(Finalizer, \bot) \\ t_r \oplus r \succeq t \\ (t', \lambda') \succeq (t_r, \lambda) \oplus m \mapsto r \\ \lambda(a) = \bot \end{array}$$

$$\textsc{T-If}\ \frac{\begin{array}{c}\Gamma, a : N \vdash i : \mathit{Bound}(t, \lambda) \\ \Gamma \vdash m : N \qquad \Gamma \vdash n : N\end{array}}{\Gamma \vdash \texttt{foreign}\ a : N = IfThenElse(e, m, n)\ \texttt{in}\ i : \mathit{Bound}(t', \lambda')}\ \ (t', \lambda') \succeq (t, \lambda \backslash \{a\}) \quad \begin{array}{l} \oplus (m \mapsto \lambda(a)) \\ \oplus (n \mapsto \lambda(a)) \end{array}$$

Figure D.2: Type system for resource guarantees (foreign terms and terms specific to this work).

two concurrent processes, while rule T-Choice only considers a boundary for any of two processes which might be executed. Rules T-Receive and T-Send permit the typing of communications, including the transfer of resources, while rules T-ReceiveUnknown and T-SendUnknown forbid transfer of resources or any assumption on names which are received from the outside world or sent to the outside world, in which case no guarantee can made regarding their usage. Rule T-If permits typing flow control statements, while T-Alloc-Value and T-Alloc-Nothing compute bounds on the resource usage of an allocator. Finally, T-Null states that the dangling reference may have any type and T-Ref permits fetching the type of a reference from the environment.

Note that, while T-Alloc-Nothing and T-Alloc-Value are valid with respect to the encoding of Erlang presented in this paper, they represent a simplification of the generic rule for Teπc, and may easily be rendered invalid by enriching the language with well-chosen allocators and/or resource signatures. Also note that this type system makes no attempt to type intelligently lists or tuples, as they are not the focus of this study, or non-trivial process-to-process communication, as they are inherently untyped in Erlang.

Appendix E

Proving the type system

E.1 Lemmas

Lemma 3 (α-conversion in foreign operations) *If* $\Gamma, a : T_a \vdash e : B$ *and* $b \notin fn(e)$ *then* $\Gamma, b : T_a \vdash e\{a \leftarrow b\} : T$.

We prove this lemma by structural congruence upon a proof of $\Gamma, a : r \vdash e : T$.

T-Null We have $e = \circledcirc = e\{a \leftarrow b\}$. Trivial.

T-Ref We have $e = a$ and $T = T_a$. Trivial.

T-Alloc-Nothing We have any T. Trivial.

□

Note This lemma is heavily dependent on the set of foreign operations permitted.

Lemma 4 (α-conversion) *If* $\Gamma, a : r \vdash P : Bound(t_P, \lambda_P)$ *and* $b \notin fn(P)$ *then* $\Gamma, b : r \vdash P\{a \leftarrow b\} : Bound(t_P, \lambda_{\{}a \leftarrow b\})$.

We prove this lemma by structural congruence upon a proof of $\Gamma, a : r \vdash P : T$. Most cases are identical to their counterpart in cπ .

The key differences are in the handling of

T-New As *res* is stable by α-conversion and by induction hypothesis, we may apply again T-New, with the same hypothesis and the same result.

T-Op By lemma 3, we have $\Gamma \vdash e\{a \leftarrow b\} : Allocation(r_e)$. From this and by induction hypothesis, the hypothesis of T-Op are stable by α-conversion. Therefore, we may again apply T-Op and obtain the same result.

T-ReceiveUnknown Trivial.

T-SendUnknown Trivial.

□

Lemma 5 (Substitution) *If* $\Gamma, x : N \vdash P : \mathit{Bound}(t, \lambda)$ *and* $\Gamma \vdash m : N$ *then* $\Gamma \vdash P\{x \leftarrow m\} : \mathit{Bound}(t, \lambda \backslash \{x\}) \oplus m \mapsto \lambda(x)$

This is proved by induction on the structure of a proof of $\Gamma, x : N \vdash P : \mathit{Bound}(t, \lambda)$.

The only tricky case is $P = \texttt{foreign}\ a : M = \mathit{Foreign}(x)\ \texttt{in}\ i$ with $m = c$, where $M = \mathit{Name}(\mathit{Finalizer}, \bot)$.

We have the following type derivation

Typing P		
$\Gamma, x : \mathit{Name}(\_, r_a), a : M \vdash i :$	$\mathit{Bound}(t_i, \lambda_i)$	By hypothesis
$\Rightarrow \Gamma, x : \mathit{Name}(\_, r_a) \vdash \texttt{foreign}\ a : M = \mathit{Finalize}(x)\ \texttt{in}\ i :$	$\mathit{Bound}(t, \lambda)$	From T-Finalize
Where	$t \succeq t_r$	
	$\lambda' \succeq \lambda_i \oplus a \mapsto r$	
	$t_r \oplus r \succeq t_i$	
	$\lambda_i(a) = \bot$	

$\bigcirc$

Therefore, we may have

Typing $i\{x \leftarrow \circledcirc\}$		
$\Gamma, x : \mathit{Name}(\_, r_a), a : M \vdash i :$	$\mathit{Bound}(t_i, \lambda_i)$	See above
$\Rightarrow \Gamma, a : M \vdash i\{x \leftarrow \circledcirc\} :$	$\mathit{Bound}(t_i \oplus \lambda_i(x), \lambda_i \backslash \{x\})$	From *Induction*

Typing P		
$\Gamma, a : M \vdash i\{x \leftarrow \circledcirc\} :$	$\mathit{Bound}(t_i \oplus \lambda_i(x), \lambda_i \backslash \{x\})$	See above
Since	$(\lambda_i \backslash \{x\})(a) = \bot$	
	$t \oplus \lambda(x) \succeq t_r \oplus \lambda_i(x) \oplus r$	
	$t_r \oplus \lambda_i(x) \oplus r \succeq t_i \oplus \lambda_i(x)$	
$\Rightarrow \Gamma \vdash \texttt{foreign}\ a : M = \mathit{Finalize}(\circledcirc)\ \texttt{in}\ i\{x \leftarrow \circledcirc\} :$	$\mathit{Bound}(t'', \lambda'')$	From T-Finaliz
Where	$t'' = t'_r \oplus r$	
	$t'_r = t \oplus \lambda_i(x)$	
	$\lambda'' = \lambda_i \backslash \{x\}$	

$\bigcirc$

As $t'' = t \oplus \lambda_i(x) \oplus r$, and $\lambda(x) \succeq \lambda_i \oplus a \mapsto r$, we have $t'' \preceq t \oplus \lambda(x)$.

In addition, $\lambda'' = \lambda_i \backslash \{x\} = (\lambda_i \oplus (x \mapsto r)) \backslash \{x\} \preceq \lambda'$.

By weakening, we conclude that $\Gamma \vdash P\{x \leftarrow m\} : \mathit{Bound}(t, \lambda \backslash \{x\}) \oplus m \mapsto \lambda(x)$

Lemma 6 (Weakening) *If* $\Gamma \vdash P : T$ *and* $a \notin fn(P)$ *then* $\Gamma, a : N \vdash P : T$.

Trivial. □

Lemma 7 (Weakening deallocations) *If* $\Gamma \vdash P : \mathit{Bound}(t, \lambda)$ *and* $\lambda' \succeq \lambda$ *then* $\Gamma \vdash P : \mathit{Bound}(t, \lambda')$.

Trivial. $\square$

Lemma 8 (Strengthening deallocations) *If $\Gamma \vdash P : Bound(t, \lambda)$ and $a \notin fn(P)$ then $\Gamma \vdash P : Bound(t, \lambda \backslash \{a\})$.*

Trivial. $\square$

Lemma 9 (Foreign operation progress) *Let e and e' be two foreign operations such that $\_ \Vdash e \rightsquigarrow e'$. Let Γ be an environment such that $\Gamma \vdash \texttt{foreign}\ x : N = e\ \texttt{in}\ i : T$. Then we have $\Gamma \vdash e : T_e$ for some T_e. In addition, we have $\Gamma \vdash e : T'_e$.*

We prove this by examining relations $\cdot \Vdash \cdot \rightsquigarrow \cdot$.

cases $l = Cons(h,t) \Vdash Head(l) \rightsquigarrow h$ and $l = Cons(h,t) \Vdash Tail(l) \rightsquigarrow t$: everything has type $Allocation(\_)$

case $t = Tuple_n(a_1, a_2, \ldots, a_n) \Vdash Nth_p(t) \rightsquigarrow a_p \quad 1 \leq p \leq n$: everything has type $Allocation(\_)$

case $\dfrac{a = v \Vdash e \rightsquigarrow f}{a = v \Vdash IfThenElse(e,b,c) \rightsquigarrow IfThenElse(f,b,c)}$: everything has type $Allocation(\_)$

$\square$

Lemma 10 (Foreign operation dereference) *Let e be a foreign operation and b a name such that $e \multimap b$. Let Γ be an environment such that $\Gamma \vdash \texttt{foreign}\ x : N = e\ \texttt{in}\ i : T$. Then we have $b \in \Gamma$ and $\Gamma(b) = x$.*

The fact that $b \in \Gamma$ stems from the definition of $e \multimap b$ and T-Alloc-Nothing: since $b \in fn(e)$ and $fn(e) \subseteq \Gamma$.

We prove this by examining relations $\cdot \Vdash \cdot \rightsquigarrow \cdot$ and $\cdot \multimap \cdot$.

- case $a \multimap a$: trivial
- case $IfThenElse(True, a, b) \multimap a$: trivial
- case $IfThenElse(False, a, b) \multimap b$: trivial

$\square$

Note This lemma is quite dependent on the set of allowed foreign operations. However, it seems quite likely that it could be generalised by adding a type discipline to foreign operations themselves.

Lemma 11 (Pseudo-weakening environment) *If Γ is an environment, if we have a process A, a name b and types T, N such that $\Gamma, b : N \vdash A : T$ and $\Gamma \vdash b : N$ then we also have $\Gamma \vdash A : T$.*

This lemma is trivial. $\square$

E.2 Subject equivalence

Proposition 1 (Subject equivalence) *If Γ is an environment, if A is a process such that $\Gamma \vdash A : T$ and if B is a process such that $B \equiv A$ then we also have $\Gamma \vdash B : T$.*

By induction upon a proof of $A \equiv B$, we prove that $\Gamma \vdash A : T$ if and only if $\Gamma \vdash B : T$ We may have $A \equiv B$ either by an application of rules E-Commut, E-Scope, E-Par or E-New, or by one of the following pseudo-rules:

$$\textsc{E-Trans}\ \frac{P \equiv Q \qquad Q \equiv R}{P \equiv R} \qquad \textsc{E-Refl}\ \frac{}{P \equiv P} \qquad \textsc{E-Sym}\ \frac{P \equiv Q}{Q \equiv P}$$

$$\textsc{E-Par-Commut}\ \frac{}{P|Q \equiv Q|P} \qquad \textsc{E-Par-Assoc}\ \frac{}{(P|Q)|R \equiv P|(Q|R)}$$

$$\textsc{E-Choice-Commut}\ \frac{}{P + Q \equiv Q + P}$$

$$\textsc{E-Choice-Assoc}\ \frac{}{(P + Q) + R \equiv P + (Q + R)}$$

$$\textsc{E-Alpha}\ \frac{}{(\nu a = v)P \equiv (\nu b = v)P\{a \leftarrow b\}}\ b \notin fn(P)$$

E.2.1 Initialisation

E-Refl Trivial.

E-Par-Commut We have $A = P|Q$ and $B = Q|P$. As rule T-Par is symmetrical with respect to A and B, the case is trivial.

E-Par-Assoc We have $A = (P|Q)|R$ and $B = P|(Q|R)$. It is easy to check that, by two applications of rule T-Par, we obtain the same constraints for minimal types of A and B. Which proves the case.

E-Choice-Commut As E-Par-Commut.

E-Choice-Assoc As E-Par-Assoc.

E-Alpha By lemma 4.

E-Scope Let us consider $A = ((\nu a = v : N)P)|Q$ such that $\Gamma \vdash A : T$ and $a \notin fn(Q)$. Then, we necessarily have the following derivation:

Typing $(\nu a : Name(C, r_a) = v)P$

$\Gamma, a : Name(C, r_a) \vdash P :$	$Bound(t_P, \lambda_P)$	By hypothesis
$\Rightarrow \Gamma \vdash (\nu a : Name(C, r_a) = v)P :$	$Bound(t_1, \lambda_1)$	From T-New
Where	$res(v) \preceq r_a$	
	$\lambda_P(a) \preceq r_a$	
	$\lambda_1 \succeq \lambda \backslash \{a\}$	
	$t_1 \succeq r_a \oplus t_P$	
Typing $((\nu a = v : N)P)\|Q$		
$\Gamma \vdash (\nu a : Name(C, r_a) = v)P :$	$Bound(t_1, \lambda_1)$	See above
$\Gamma \vdash Q :$	$Bound(t_Q, \lambda_Q)$	By hypothesis
$\Rightarrow \Gamma \vdash ((\nu a = v : N)P)\|Q :$	$Bound(t_2, \lambda_2)$	From T-Par
Where	$t_2 \succeq t_1 \oplus t_Q$	
	$\lambda_2 \succeq \lambda_1 \oplus \lambda_Q$	

$\bigcirc$

By lemma 8, as $a \notin fn(Q)$, we may assume that $\lambda_Q(a) = \bot$.

Consequently, we may also derive :

Typing Q		
$\Gamma \vdash Q :$	$Bound(t_Q, \lambda_Q)$	By hypothesis
$\Rightarrow \Gamma, a : Name(C, r_a) \vdash Q :$	$Bound(t_Q, \lambda_Q)$	From *lemma 6*
Typing $P\|Q$		
$\Gamma, a : Name(C, r_a) \vdash P :$	$Bound(t_P, \lambda_P)$	By hypothesis
$\Gamma, a : Name(C, r_a) \vdash Q :$	$Bound(t_Q, \lambda_Q)$	See above
$\Rightarrow \Gamma, a : Name(C, r_a) \vdash P\|Q :$	$Bound(t_3, \lambda_3)$	From T-Par
Where	$t_3 = t_P \oplus t_Q$	
	$\lambda_3 = \lambda_P \oplus \lambda_Q$	
Typing $(\nu a = v : N)(P\|Q)$		
$\Gamma, a : Name(C, r_a) \vdash P\|Q :$	$Bound(t_3, \lambda_3)$	See above
Since	$\lambda_3 = \lambda_P \oplus \lambda_Q$	
	$\lambda_P \preceq r_a$	
we deduce	$\lambda_3(a) \preceq r_a$	
Since	$t_2 \succeq t_1 \oplus t_Q$	
	$t_1 \succeq r_a \oplus t_P$	
	$\lambda_2 \succeq \lambda_1 \oplus \lambda_Q$	
	$\lambda_1 \succeq \lambda \backslash \{a\}$	
$\Rightarrow \Gamma \vdash (\nu a = v : N)(P\|Q) :$	$Bound(t_2, \lambda_2)$	From T-New

$\bigcirc$

Which proves one implication. The other way is similar.

E-New Trivial.

E.2.2 Induction step

E-Par Directly by induction hypothesis.

E-Trans Directly by induction hypothesis.

E-Sym Directly by induction hypothesis.

E-Commut Directly by induction hypothesis.

We conclude by induction that the proposition has been proved.$\square$

E.3 Open Subject Reduction

Proposition 2 (Open Subject Reduction) *If A is a process, Γ an environment such that $\Gamma \vdash A : Bound(r, \lambda)$ and if there is some process A', a label l and some resource r such that $A \xrightarrow{l}_{pre} A'$ then*

- *if $l = \tau$ or $l = e \rightsquigarrow f$, we have $\Gamma \vdash A' : Bound(r', \lambda')$, where $\Sigma_{c\in\mathbf{N}}\lambda'(c) \preceq \Sigma_{c\in\mathbf{N}}\lambda'(c)$ $r' \oplus \Sigma_{c\in\mathbf{N}}\lambda'(c) \preceq r \oplus \Sigma_{c\in\mathbf{N}}\lambda'(c)$*
- *if $l = a?b$*
  - *either $\Gamma \vdash a : Name(Chan(N, g_a, d_a), \_)$, $\Gamma, b : N \vdash A' : Bound(r', \lambda')$, $r' \preceq r \oplus g_a$ and $\lambda' \preceq \lambda \oplus b \mapsto d_a$.*
  - *or $\Gamma \vdash a : Name(Unknown, \_)$, $\Gamma, b : Name(Unknown, \_) \vdash A' : Bound(r', \lambda')$, $r' \preceq r$ and $\lambda' \preceq \lambda$*
- *if $l = a?\circledcirc$, then*
  - *either $a \in \Gamma$, $\Gamma \vdash a : Name(Chan(N, g_a, d_a), \_)$, $r' \preceq r \oplus g_a \oplus d_a$ and $\lambda' \preceq \lambda$.*
  - *or $\Gamma \vdash a : Name(Unknown, \_)$, $r' \preceq r$ and $\lambda' \preceq \lambda$*
- *if $l = a!b$, then*
  - *either $\Gamma \vdash a : Name(Chan(N, g_a, d_a), \_)$, $\Gamma \vdash b : N$, $r' \oplus g_a \preceq r$, $\lambda' \oplus b \mapsto d_a \preceq \lambda$*
  - *or $\Gamma \vdash a : Name(Unknown, \_)$, $\Gamma \vdash b : (Unknown, \_)$, $r' \preceq r$ and $\lambda' \preceq \lambda$*
- *if $l = a!\circledcirc$, then*
  - *either $\Gamma a \vdash Name(Chan(\_, g_a, d_a), \_)$, $r' \oplus g_a \oplus d_a \preceq r$, $\lambda' \preceq \lambda$*
  - *or $\Gamma \vdash a : Name(Unknown, \_)$, $r' \preceq r$ and $\lambda' \preceq \lambda$*
- *if $l = \nu b : N = v.a!b$, then*
  - *either $\Gamma a \vdash Name(Chan(N, g_a, d_a), \_)$, $N = Name(\_, r_b)$, $r_b \succeq res(v)$, $r' \oplus g_a \oplus r_b \preceq r$, $\lambda'_(b) \oplus d_a \preceq r_b$.*
  - *or $\Gamma \vdash a = Name(Unknown, \_)$, $N = Name(Unknown, r_b)$, $r_b \succeq res(v)$, $r' \oplus r_b \preceq r$, $\lambda' \preceq \lambda$ and $\lambda(b) \preceq r_b$.*

We prove this proposition by induction on the structure of a proof of $A \longrightarrow_{pre} A'$.

E.3.1 Initialisation

R-Spawn Label is τ. Rules T-Spawn and T-Par produce the same typings from the same set of hypothesis.

R-Either Label is τ. Rules T-Either and T-Choice produce the same typings from the same set of hypothesis.

R-Deallocate Label is τ. We have $A = (\nu a : Name(C, r_a) = v)P$ and $A' = P\{a \leftarrow \circledcirc\}$. By T-New, we also have $\Gamma, a : Name(C, r_a) \vdash A : Bound(r_P, \lambda_P)$, with $r \succeq r_P \oplus r_a$, $\lambda \succeq \lambda_P \backslash \{a\}$, $\lambda_P(a) \preceq r_a$ and $res(v) \preceq r_a$. By lemma 5, if $\Gamma, a : N \vdash P : Bound(r_P, \lambda_P)$, then $\Gamma \vdash P\{a \leftarrow \circledcirc\} : Bound(r'_P, \lambda'_P)$, where $\Sigma_{b \in \mathcal{N}} \lambda'_P(b) \preceq \Sigma_{b \in \mathcal{N}} \lambda'_P(b)$ and $r'_P \oplus \Sigma_{b \in \mathcal{N}} \lambda'_P(b) \preceq r_P \Sigma_{b \in \mathcal{N}} \lambda'_P(b)$.

Which proves the case.

R-Deref Label is τ. We have $A =$ `foreign` $x : N = e$ `in` i and $A' = i\{x \leftarrow m\}$ where $m = \circledcirc$ or $m \notin fn(i)$ and $N = Name(\_, r_e)$ Let us consider the typings of A and deduce a typing of A'

Typing A		
$\Gamma \vdash e :$	$Allocation(r_e)$	
$\Gamma, x : N \vdash i :$	$Bound(t_i, \lambda_i)$	
$\Rightarrow \Gamma \vdash A :$	$Bound(t, \lambda)$	From T-Let
Where	$t \succeq t_i \oplus r_e$	
	$\lambda \succeq \lambda_i \backslash \{x\}$	
	$\lambda_i(x) \preceq r_e$	

Typing A' case $m = b \neq \circledcirc$		
$\Gamma, x : N \vdash i :$	$Bound(t_i, \lambda_i)$	By hypothesis
$\Rightarrow \Gamma \vdash i\{x \leftarrow b\} :$	$Bound(t_i, \lambda_i \backslash \{x\})$	From *Substitution*
Since	$\Gamma(b) = N$	From *Lemma 10*
$\Rightarrow \Gamma \vdash A' :$	$Bound(t, \lambda)$	From *Weakening*
Since	$t \succeq t_i \oplus r_e$	
	$\lambda \succeq \lambda_i \backslash \{x\}$	

Typing A' case $m = \circledcirc$		
$\Gamma, x : N \vdash i :$	$Bound(t_i, \lambda_i)$	By hypothesis
$\Rightarrow \Gamma \vdash i\{x \leftarrow \circledcirc\} :$	$Bound(t_i \oplus \lambda_i(x), \lambda_i \backslash \{x\})$	From *Substitution*
$\Rightarrow \Gamma \vdash A' :$	$Bound(t, \lambda)$	From *Weakening*
Since	$\lambda_i(x) \preceq r_e$	
	$\lambda \succeq \lambda_i \backslash \{x\}$	

○

Which proves the case.

R-Store Label is τ. We have $A =$ `foreign` $a : N = e$ `in` i and $A' = (\nu a : N =$

$v)i$, where $N = Name(\_, r_e)$. Let us consider the typings of A and deduce a typing of A'

Typing A			
$\Gamma \vdash e :$		$Allocation(r_e)$	
$\Gamma, a : N \vdash i :$		$Bound(t_i, \lambda_i)$	
$\Rightarrow \Gamma \vdash A :$		$Bound(t, \lambda)$	From T-Let
	Where	$t \succeq t_i \oplus r_e$	
		$\lambda \succeq \lambda_i \backslash \{a\}$	
		$\lambda_i(a) \preceq r_e$	

Typing A'			
$\Gamma, a : N \vdash i :$		$Bound(t_i, \lambda_i)$	By hypothesis
	Since	$\lambda \succeq \lambda_i \backslash \{a\}$	
		$\lambda_i(a) \preceq r_e$	
		$t \succeq t_i \oplus r_e$	
$\Rightarrow \Gamma \vdash (\nu a : N = v)i :$		$Bound(t, \lambda)$	From T-New

○

Which proves the case.

R-Send We have $A = a\ m$, $A' =$ `nothing` and $l = a!m$. We also have $\Gamma \vdash A : Bound(t, \lambda)$ hence either $\Gamma \vdash a : Name(Chan(N, g_a, d_a), \_)$ for some g_a and d_a. (by T-Send) or $\Gamma \vdash a : Name(Unknown, \_)$ (by T-SendUnknown).

- If $\Gamma \vdash a : Name(Chan(N, g_a, d_a), \_)$ and $m \neq \circledcirc$, we know that $t \succeq g_a$ and $\lambda \succeq b \mapsto d_a$. As `nothing` may have any $Bound(\_)$ type, $Bound(\bot, \bot_{\mathcal{N}})$ is a possible choice, and it matches the property we need to prove. In addition, by T-Send, we have $\Gamma \vdash b : N$.
- if $\Gamma \vdash a : Name(Chan(N, g_a, d_a), \_)$ and $m = \circledcirc$, we know that $t \succeq g_a \oplus d_a$. As `nothing` may have any $Bound(\_)$ type, $Bound(\bot, \bot_{\mathcal{N}})$ is a possible choice, and it matches the property we need to prove.
- if $\Gamma \vdash a : Name(Unknown, \_)$ and $m \neq \circledcirc$, we use $Bound(\bot, \bot_{\mathcal{N}})$ In addition, by T-SendUnknown, we have $\Gamma \vdash b : Name(Unknown, \_)$.
- if $\Gamma(a) = Name(Unknown, \_)$ and $m = \circledcirc$, we use again $Bound(\bot, \bot_{\mathcal{N}})$.

Which proves the case.

R-Once We have $A =$ `once` $a\ x$ `do` i and $A' = i\{x \leftarrow m\}$. We also have $\Gamma \vdash A : Bound(t, \lambda)$ hence either $\Gamma \vdash a : Name(Chan(N, g_a, d_a), \_)$ for some g_a and d_a (by T-Receive) or $\Gamma \vdash a : Name(Unknown, \_)$ (by T-ReceiveUnknown). Therefore, depending on whether a is tagged as *Unknown* and on whether m is $\circledcirc$, we have 4 cases to consider.

Case 1 If $\Gamma \vdash a : Name(Chan(N, g_a, d_a), \_)$ and $m \neq \circledcirc$, we have $\Gamma, x : N \vdash i : Bound(t_i, \lambda_i)$, where $t \oplus g_a \succeq t_i$, $\lambda \succeq \lambda_i \backslash \{x\}$ and $d_a \succeq \lambda_i(x)$.

By hypothesis, we have $\Gamma \vdash b : N$. Hence, by lemma 5 (substitution), we conclude that $\Gamma \vdash i\{x \leftarrow b\} : \mathit{Bound}(t_i, \lambda_i \backslash \{x\}) \oplus (b \mapsto \lambda_i(x))$. With $t' = t_i$ and $\lambda' = \lambda_i \backslash \{x\} \oplus (b \mapsto \lambda_i(x))$, we conclude that $t' \preceq t \oplus g_a$ and $\lambda' \preceq \lambda \oplus b \mapsto d_a$, which proves the case.

Case 2 If $\Gamma \vdash a : \mathit{Name}(\mathit{Chan}(N, g_a, d_a), \_)$ and $m = \circledcirc$, we have $\Gamma, x : N \vdash i : \mathit{Bound}(t_i, \lambda_i)$, where $t \oplus g_a \succeq t_i$, $\lambda \succeq \lambda_i \backslash \{x\}$ and $d_a \succeq \lambda_i(x)$. Hence, by lemma 5 (substitution), we conclude that $\Gamma \vdash i\{x \leftarrow b\} : \mathit{Bound}(t_i, \lambda_i \backslash \{x\}) \oplus (b \mapsto \lambda_i(x))$. With $t' = t_i \oplus \lambda_i(x)$ and $\lambda' = \lambda_i \backslash \{x\}$, we conclude that $t' \preceq t \oplus g_a \oplus d_a$ and $\lambda' \preceq \lambda$, which proves the case.

Case 3 If $\Gamma \vdash a : \mathit{Name}(\mathit{Unknown}, \_)$ and $m \neq \circledcirc$, by T-ReceiveUnknown, we have $\Gamma, x : \mathit{Name}(\mathit{Unknown}, \_) \vdash i : \mathit{Bound}(t_i, \lambda_i)$, where $t \succeq t_i$, $\lambda \succeq \lambda_i$ and $\lambda_i(x) = \bot$. By hypothesis, we have $\Gamma \vdash b : \mathit{Name}(\mathit{Unknown}, \_)$. Hence, by lemma 5 (substitution), we conclude that $\Gamma \vdash i\{x \leftarrow b\} : \mathit{Bound}(t_i, \lambda_i \backslash \{x\}) \oplus (b \mapsto \lambda_i(x))$. With $t' = t_i$ and $\lambda' = \lambda_i \backslash \{x\} \oplus (b \mapsto \lambda_i(x))$, we conclude that $t' \preceq t$ and $\lambda' \preceq \lambda \backslash \{x\}$, which proves the case.

Case 4 If $\Gamma \vdash a : \mathit{Name}(\mathit{Unknown}, \_)$ and $m = \circledcirc$, we have $\Gamma, x : \mathit{Name}(\mathit{Unknown}, \_) \vdash i : \mathit{Bound}(t_i, \lambda_i)$, where $t \succeq t_i$ and $\lambda \succeq \lambda_i$ and $\lambda(x) = \bot$. Hence, by lemma 5 (substitution), we conclude that $\Gamma \vdash i\{x \leftarrow b\} : \mathit{Bound}(t_i, \lambda_i \backslash \{x\}) \oplus (b \mapsto \lambda_i(x))$. With $t' = t_i$ and $\lambda' = \lambda_i \backslash \{x\} \oplus (b \mapsto \lambda_i(x))$, we conclude that $t' \preceq t$ and $\lambda' \preceq \lambda$, which proves the case.

R-On We have $A = \texttt{on}\ a\ x\ \texttt{do}\ i$ and $A' = i\{x \leftarrow\} \mid \texttt{on}\ a\ x\ \texttt{do}\ .$ We also have $\Gamma \vdash A : \mathit{Bound}(t, \lambda)$ hence either $\Gamma \vdash a : \mathit{Name}(\mathit{Chan}(N, g_a, d_a), \_)$ for some g_a and d_a (by T-Receive) or $\Gamma \vdash a : \mathit{Name}(\mathit{Unknown}, \_)$ (by T-ReceiveUnknown). In addition, by T-Repl, we have $\Gamma \vdash \texttt{once}\ a\ x\ \texttt{do}\ i : \mathit{Bound}(\bot, \bot_{\mathcal{N}})$.

- If $\Gamma \vdash a : \mathit{Name}(\mathit{Chan}(N, g_a, d_a), \_)$, $m \neq \circledcirc$ and $\Gamma \vdash b : N$, we have $\Gamma, x : N \vdash i : \mathit{Bound}(\bot, \bot_{\mathcal{N}})$. Hence, by lemma 5 (substitution), we have $\Gamma \vdash i\{x \leftarrow b\} : \mathit{Bound}(\bot, \bot_{\mathcal{N}})$. By T-Par, we deduce that $\Gamma \vdash A' : \mathit{Bound}(t, \lambda)$. Which proves the case.
- If $\Gamma \vdash a : \mathit{Name}(\mathit{Chan}(N, g_a, d_a), \_)$ and $m = \circledcirc$, we have $\Gamma, x : N \vdash i : \mathit{Bound}(\bot, \bot_{\mathcal{N}})$. Hence, by lemma 5 (substitution), we have $\Gamma \vdash i\{x \leftarrow m\} : \mathit{Bound}(\bot, \bot_{\mathcal{N}})$. By T-Par, we deduce that $\Gamma \vdash A' : \mathit{Bound}(t, \lambda)$. Which proves the case.
- If $\Gamma \vdash a : \mathit{Name}(\mathit{Unknown}, \_)$, $m \neq \circledcirc$ and $\Gamma \vdash b : \mathit{Name}(\mathit{Unknown}, \_)$, we have $\Gamma, x : \mathit{Name}(\mathit{Unknown}, \_) \vdash i : \mathit{Bound}(\bot, \bot_{\mathcal{N}})$. Hence, by lemma 5 (substitution), we have $\Gamma \vdash i\{x \leftarrow b\} : \mathit{Bound}(\bot, \bot_{\mathcal{N}})$. By T-Par, we deduce that $\Gamma \vdash A' : \mathit{Bound}(t, \lambda)$. Which proves the case.
- If $\Gamma \vdash a : \mathit{Name}(\mathit{Unknown}, \_)$ and $m = \circledcirc$, we have $\Gamma, x : \mathit{Name}(\mathit{Unknown}, \_) \vdash i : \mathit{Bound}(\bot, \bot_{\mathcal{N}})$. Hence, by lemma 5 (substitution), we have $\Gamma \vdash i\{x \leftarrow m\} : \mathit{Bound}(\bot, \bot_{\mathcal{N}})$. By T-Par, we deduce that $\Gamma \vdash A' : \mathit{Bound}(t, \lambda)$. Which proves the case.

R-Evaluate This is a direct corollary of lemma 9.

E.3.2 Induction step

R-Comm 1 Let us consider the case where $A = P|Q$, $A' = P'|Q'$, where $P \xrightarrow{a?m}_{pre} P'$, $Q \xrightarrow{a!m}_{pre} Q'$, $\Gamma \vdash P : \mathit{Bound}(t_P, \lambda_P)$, $\Gamma \vdash Q : \mathit{Bound}(t_Q, \lambda_Q)$, $\Gamma \vdash a : \mathit{Name}(C, \_)$. and $m \neq \circledcirc$. Let us write $b = m$.

By induction hypothesis, we have $\Gamma \vdash b : N$, $\Gamma \vdash a : \mathit{Name}(\mathit{Chan}(N, g_a, d_a), \_)$, $\Gamma, b : N \vdash P' : \mathit{Bound}(t'_P, \lambda'_P)$, $t'_P \preceq t_P \oplus g_a$, $\lambda'_P \preceq \lambda_P \oplus b \mapsto d_a$. $t'_Q \oplus g_a \preceq t_Q$, $\lambda'_Q \oplus b \mapsto d_a \preceq \lambda_Q$.

By T-Par, we deduce $\Gamma \vdash A' : \mathit{Bound}(t'_P \oplus t'_Q, \lambda'_P \oplus \lambda'_Q)$. As $t'_P \preceq t_P \oplus g_a$ and $t'_Q \oplus g_a \preceq t_Q$, we deduce $t'_P \oplus t'_Q \preceq t_P \oplus t_Q \preceq t$. As $\lambda'_P \preceq \lambda_P \oplus b \mapsto d_a$. and $\lambda'_Q \oplus b \mapsto d_a \preceq \lambda_Q$, we deduce $\lambda'_P \oplus \lambda'_Q \preceq \lambda_P \oplus \lambda_Q \preceq \lambda$.

Which proves the case.

R-Comm 2 Let us consider the case where $A = P|Q$, $A' = P'|Q'$, where $P \xrightarrow{a?m}_{pre} P'$, $Q \xrightarrow{a!m}_{pre} Q'$, $\Gamma \vdash P : \mathit{Bound}(t_P, \lambda_P)$, $\Gamma \vdash Q : \mathit{Bound}(t_Q, \lambda_Q)$, $\Gamma \vdash a : \mathit{Name}(C, \_)$. and $m = \circledcirc$.

By induction hypothesis, we have $\Gamma \vdash a : \mathit{Name}(\mathit{Chan}(N, g_a, d_a), \_)$, $\Gamma \vdash P' : \mathit{Bound}(t'_P, \lambda'_P)$, $t'_P \preceq t_P \oplus g_a \oplus d_a$, $\lambda'_P \preceq \lambda_P$, $t'_Q \oplus g_a \oplus d_a \preceq t_Q$ and $\lambda'_Q \preceq \lambda_Q$

By T-Par, we deduce $\Gamma \vdash A' : \mathit{Bound}(t'_P \oplus t'_Q, \lambda'_P \oplus \lambda'_Q)$. As $t'_P \preceq t_P \oplus g_a \oplus d_a$ and $t'_Q \oplus g_a \oplus d_a \preceq t_Q$, we deduce $t'_P \oplus t'_Q \preceq t_P \oplus t_Q \preceq t$. As $\lambda'_P \preceq \lambda_P$. and $\lambda'_Q \lambda_Q$, we deduce $\lambda'_P \oplus \lambda'_Q \preceq \lambda_P \oplus \lambda_Q \preceq \lambda$.

Which proves the case.

R-Comm 3 Let us consider the case where $A = P|Q$, $A' = P'|Q'$, where $P \xrightarrow{a?m}_{pre} P'$, $Q \xrightarrow{a!m}_{pre} Q'$, $\Gamma \vdash P : \mathit{Bound}(t_P, \lambda_P)$, $\Gamma \vdash Q : \mathit{Bound}(t_Q, \lambda_Q)$, $\Gamma \vdash a : \mathit{Name}(\mathit{Unknown}, \_)$. and $m \neq \circledcirc$. Let us write $b = m$.

This case is identical to R-Comm 1, with $d_a = r_a = \bot$.

R-Comm 4 Let us consider the case where $A = P|Q$, $A' = P'|Q'$, where $P \xrightarrow{a?m}_{pre} P'$, $Q \xrightarrow{a!m}_{pre} Q'$, $\Gamma \vdash P : \mathit{Bound}(t_P, \lambda_P)$, $\Gamma \vdash Q : \mathit{Bound}(t_Q, \lambda_Q)$, $\Gamma \vdash a : \mathit{Name}(C, \_)$. and $m = \circledcirc$.

This case is identical to R-Comm 2, with $d_a = r_a = \bot$.

R-Comm-Close 1 Let us consider the case where $A = P|Q$, $A' = (\nu b : N = v)(P'|Q')$, where $P \xrightarrow{a?m}_{pre} P'$, $Q \xrightarrow{(\nu b):N=v.a!m}_{pre} Q'$, $\Gamma \vdash P : \mathit{Bound}(t_P, \lambda_P)$, $\Gamma \vdash Q : \mathit{Bound}(t_Q, \lambda_Q)$, $\Gamma \vdash a : \mathit{Name}(C, \_)$. and $m \neq \circledcirc$. Let us write $b = m$ and $N = \mathit{Name}(\_, r_b)$.

By induction hypothesis, we have $r_b \succeq \mathit{res}(v)$, $\Gamma \vdash a : \mathit{Name}(\mathit{Chan}(N, g_a, d_a), \_)$, $\Gamma, b : N \vdash P' : \mathit{Bound}(t'_P, \lambda'_P)$, $t'_P \preceq t_P \oplus g_a$, $\lambda'_P \preceq \lambda_P \oplus b \mapsto d_a$, $t'_Q \oplus g_a \oplus r_b \preceq t_Q$, $\lambda'_Q(b) \oplus d_a \preceq r_b$.

As $b \notin fn(P)$, by lemma 8 (strengthening deallocation), we may assume that $\lambda_P(b) = \bot$. Therefore, as $\lambda'_P \preceq \lambda_P \oplus b \mapsto d_a$, we have $\lambda'_P(b) \preceq d_a$. As $\lambda'_Q(b) \oplus d_a \preceq r_b$, we deduce that $(\lambda'_P \oplus \lambda'_Q)(b) \preceq r_b$. Similarly, we deduce that $t'_P \oplus t'_Q \oplus r_b \preceq t_P \oplus t_Q$.

By T-Par and T-Res, we may therefore prove that $\Gamma \vdash A' : Bound(t', \lambda')$, with $t' = t'_P \oplus t'_Q \oplus r_b$ and $\lambda' = (\lambda'_P \oplus \lambda'_Q)\backslash\{b\}$. As $\lambda'_P(b) \preceq d_a$ and $\lambda'_Q 9b) \oplus d_a \preceq r_b$, we further deduce that $\lambda' \preceq \lambda$ and $t' \oplus \Sigma_{c\in\mathcal{N}}\lambda'(c) \preceq t \oplus \Sigma_{c\in\mathcal{N}}\lambda(c)$.

Which proves the case.

R-Comm-Close 2 Let us consider the case where $A = P|Q$, $A' = (\nu b : N = v)(P'|Q')$, where $P \xrightarrow{a?m}_{pre} P'$, $Q \xrightarrow{(\nu b):N=v.a!m}_{pre} Q'$, $\Gamma \vdash P : Bound(t_P, \lambda_P)$, $\Gamma \vdash Q : Bound(t_Q, \lambda_Q)$, $\Gamma \vdash a : Name(Unknown, \_)$. and $m \neq \circledcirc$. This case is essentially identical to R-Comm-Close 1, with $d_a = g_a = \bot$.

R-Fetch We have $A = (\nu a : N = v)P$, $A' = (\nu a : N = v)P'$ and $P \overset{e \rightsquigarrow f}{\longmapsto} P'$. By induction hypothesis, the type of P and the type of P' are the same. Which proves the case.

R-Par We have $A = P|Q$ and $A' = P'|Q$ where $P \xrightarrow{\alpha}_{pre} P'$. By examining all possible configurations of α, invoking the corresponding induction hypothesis and pasting the result in T-Par, we prove the case.

R-Choice As R-Par, just easier.

R-Hide As R-Par.

R-Equiv This is a direct corollary of lemma 1 (Subject Equivalence).

R-Open 1 Let us consider the case where $A = (\nu b : Name(\_, r_b) = v)P$, $A' = P'$, $P \xrightarrow{a!b}_{pre} P'$ and $\Gamma \vdash a : Name(C, \_)$.

As A may be typed in Γ, by T-New, have $\Gamma, b : N \vdash P : Bound(t_P, \lambda_P)$, where $t \succeq t_P \oplus r_b$, $\lambda \succeq \lambda_P\backslash\{b\}$, $\lambda_P(b) \preceq r_b$ and $r_b \succeq res(v)$.

By induction hypothesis, we also have $\Gamma \vdash a : Name(Chan(N, g_a, d_a), \_)$, $\Gamma \vdash b : N$, $t'_P \oplus g_a \preceq t_P$ and $\lambda'_P \oplus b \mapsto d_a \preceq \lambda_P$.

Consequently, we have $\lambda'_P(b) \oplus d_a \preceq \lambda_P(b) \preceq r_b$ and $t'_P \oplus g_a \oplus r_b \preceq t_P \oplus r_b \preceq t$.

Therefore, we have $\Gamma \vdash A' : Bound(t', \lambda')$, with $\lambda'_(b) \oplus d_a \preceq r_b$ and $t' \oplus g_a \oplus r_b \preceq t$, which proves the case.

R-Open 2 Let us consider the case where $A = (\nu b : Name(\_, r_b) = v)P$, $A' = P'$, $P \xrightarrow{a!b}_{pre} P'$ and $\Gamma \vdash a : Name(Unknown, \_)$.

This case is essentially identical to R-Open 1, with $g_a = d_a = \bot$.

Proposition 3 (Subject Reduction) *If A is a process, Γ an environment such that $\Gamma \vdash A : Bound(r, \lambda)$ and if there is some process A' such that $A \longrightarrow_{pre} A'$, then we also have $\Gamma \vdash A' : Bound(r', \lambda')$, where $r' \preceq r$ and $\lambda' \preceq \lambda$.*

This is a direct corollary of the Open Subject Reduction.

E.4 Subject Reduction under attack

Proposition 4 (Subject Reduction under attack) *If A is a process, B a client and Γ an environment such that, Γ isolates both A and B and such that $\Gamma \vdash A : Bound(r, \lambda)$, if there is some C such that $A|B \longrightarrow C$ then we may find two processes A' and B' and a set of names, types and values $a_1 : N_1 = v_1, \ldots, a_n : N_n, v_n$ such that*

1. $C \equiv (\nu \overrightarrow{a = v})(A' \mid B')$
2. $\Gamma, \overrightarrow{a : N} \vdash A' : Bound(t', \lambda')$
3. $\Gamma, \overrightarrow{a : N}$ *isolates* B'
4. $\Sigma_{b \in \mathcal{N}} \lambda'(b) \preceq \Sigma_{b \in \mathcal{N}} \lambda(b)$
5. $t' \oplus \Sigma_{b \in \mathcal{N}} \lambda'(b) \preceq t \oplus \Sigma_{b \in \mathcal{N}} \lambda(b)$
6. $\forall i \in 1..n$, *if* $N_i = Name(\_, r_i)$ *then* $\lambda'(a_i) \preceq r_i$
7. $t' \oplus \Sigma_{i \in 1..n} r_i \Sigma_{b \in \mathcal{N}} \lambda' \setminus \overrightarrow{a}(b) \preceq t \oplus \Sigma_{b \in \mathcal{N}} \lambda(b)$
8. B' *is a client*

In particular, if $B =$ `nothing`, it is possible to find A' such that $\overrightarrow{a : N = v} = \emptyset$ and $B' =$ `nothing`. (Subject Reduction without attack). If $\lambda = \bot_{\mathcal{N}}$, we always have $\lambda' = \bot_{\mathcal{N}}$ and $t' \oplus \Sigma_{i \in 1..n} r_i \preceq t$ (Subject Reduction with closed garbage-collection).

- if $A \longrightarrow_{pre} A'$ – by Open Subject Reduction, we have $\Gamma \vdash A' : Bound(r', \lambda')$, where $\Sigma_{c \in \mathbf{N}} \lambda'(c) \preceq \Sigma_{c \in \mathbf{N}} \lambda'(c)$ $r' \oplus \Sigma_{c \in \mathbf{N}} \lambda'(c) \preceq r \oplus \Sigma_{c \in \mathbf{N}} \lambda'(c)$. With $B' = B$, we have
  1. $C = A'|B'$
  2. $\Gamma \vdash A' : Bound(t', \lambda')$
  3. Γ isolates B'
  4. $\Sigma_{b \in \mathcal{N}} \lambda'(b) \preceq \Sigma_{b \in \mathcal{N}} \lambda(b)$
  5. $t' \oplus \Sigma_{b \in \mathcal{N}} \lambda'(b) \preceq t \oplus \Sigma_{b \in \mathcal{N}} \lambda(b)$
  6. as we have added no name to Γ, $\forall i \in 1..0$, if $N_i = Name(\_, r_i)$ then $\lambda'(a_i) \preceq r_i$

7. as we have added no name to Γ, $t' \oplus \Sigma_{i\in 1..n} r_i \Sigma_{b\in\mathcal{N}} \lambda' \backslash \overrightarrow{a}(b) \preceq t \oplus \Sigma_{b\in\mathcal{N}} \lambda(b)$
8. B' is a client

which proves the case.

- if $B \longrightarrow_{pre} B'$ – we have changed nothing to the typable contents.

- if $A \xrightarrow{a!b}_{pre} A'$ and $B \xrightarrow{a?b}_{pre} B'$ – by hypothesis, since Γ isolates B, we have $\Gamma \vdash a : Name(Unknown, \_)$. By Open Subject Reduction, we may also check that we have $\Gamma \vdash A' : Bound(t', \lambda')$, where $t' \preceq t$ and $\lambda' \preceq \lambda$. In addition, by Open Subject Reduction, we have $\Gamma \vdash b : Name(Unknown, \_)$. As Γ isolates B, we deduce that Γ isolates B' Therefore, we have

  1. $C = A'|B'$
  2. $\Gamma \vdash A' : Bound(t', \lambda')$
  3. Γ isolates B'
  4. as $\lambda' \preceq \lambda$, $\Sigma_{b\in\mathcal{N}} \lambda'(b) \preceq \Sigma_{b\in\mathcal{N}} \lambda(b)$
  5. as $\lambda' \preceq \lambda$ and $t' \preceq t$, $t' \oplus \Sigma_{b\in\mathcal{N}} \lambda'(b) \preceq t \oplus \Sigma_{b\in\mathcal{N}} \lambda(b)$
  6. as we have added no name to Γ, $\forall i \in 1..0$, if $N_i = Name(\_, r_i)$ then $\lambda'(a_i) \preceq r_i$
  7. as we have added no name to Γ, $t' \oplus \Sigma_{i\in 1..n} r_i \Sigma_{b\in\mathcal{N}} \lambda' \backslash \overrightarrow{a}(b) \preceq t \oplus \Sigma_{b\in\mathcal{N}} \lambda(b)$
  8. as $B \xrightarrow{a?b}_{pre} B'$, by definition, B' is a client

  which proves the case.

- if $A \xrightarrow{a!\circledcirc}_{pre} A'$ and $B \xrightarrow{a?\circledcirc}_{pre} B'$ – by hypothesis, since Γ isolates A and B, we have $\Gamma \vdash a : Name(Unknown, \_)$. By Open Subject Reduction, we may also check that we have $\Gamma \vdash A' : Bound(t', \lambda')$, where $t' \preceq t$ and $\lambda' \preceq \lambda$. As Γ isolates B, we deduce that Γ isolates B' Therefore, we have

  1. $C = A'|B'$
  2. $\Gamma \vdash A' : Bound(t', \lambda')$
  3. Γ isolates B'
  4. as $\lambda' \preceq \lambda$, $\Sigma_{b\in\mathcal{N}} \lambda'(b) \preceq \Sigma_{b\in\mathcal{N}} \lambda(b)$
  5. as $\lambda' \preceq \lambda$ and $t' \preceq t$, $t' \oplus \Sigma_{b\in\mathcal{N}} \lambda'(b) \preceq t \oplus \Sigma_{b\in\mathcal{N}} \lambda(b)$
  6. as we have added no name to Γ, $\forall i \in 1..0$, if $N_i = Name(\_, r_i)$ then $\lambda'(a_i) \preceq r_i$
  7. as we have added no name to Γ, $t' \oplus \Sigma_{i\in 1..n} r_i \Sigma_{b\in\mathcal{N}} \lambda' \backslash \overrightarrow{a}(b) \preceq t \oplus \Sigma_{b\in\mathcal{N}} \lambda(b)$
  8. as $B \xrightarrow{a?\circledcirc}_{pre} B'$, by definition, B' is a client

which proves the case.

- if $A \xrightarrow{a?b}_{pre} A'$ and $B \xrightarrow{a!b}_{pre} B'$ – by hypothesis, since Γ isolates A and B, we have $\Gamma \vdash a : Name(Unknown, \_)$. By Open Subject Reduction, we deduce that $\Gamma, b : Name(Unknown, \_) \vdash A' : Bound(t', \lambda')$, where $t' \preceq t$ and $\lambda' \preceq \lambda$. As Γ isolates B, we deduce that Γ isolates B'. Therefore, we have

  1. $C = A'|B'$
  2. $\Gamma \vdash A' : Bound(t', \lambda')$
  3. Γ isolates B'
  4. as $\lambda' \preceq \lambda$, $\Sigma_{b\in\mathcal{N}}\lambda'(b) \preceq \Sigma_{b\in\mathcal{N}}\lambda(b)$
  5. as $\lambda' \preceq \lambda$ and $t' \preceq t$, $t' \oplus \Sigma_{b\in\mathcal{N}}\lambda'(b) \preceq t \oplus \Sigma_{b\in\mathcal{N}}\lambda(b)$
  6. as we have added no name to Γ, $\forall i \in 1..0$, if $N_i = Name(\_, r_i)$ then $\lambda'(a_i) \preceq r_i$
  7. as we have added no name to Γ, $t' \oplus \Sigma_{i\in 1..n} r_i \Sigma_{b\in\mathcal{N}} \lambda' \backslash \overrightarrow{a}(b) \preceq t \oplus \Sigma_{b\in\mathcal{N}}\lambda(b)$
  8. as $B \xrightarrow{a!b}_{pre} B'$, by definition, B' is a client

  which proves the case.

- if $A \xrightarrow{a?\circledcirc}_{pre} A'$ and $B \xrightarrow{a!\circledcirc}_{pre} B'$ – by hypothesis, since Γ isolates A and B, we have $\Gamma \vdash a : Name(Unknown, \_)$. By Open Subject Reduction, we deduce that $r' \preceq r$ and $\lambda' \preceq \lambda$. As previously, the case is proved.

- if $A \xrightarrow{\nu b:N=v.a!b}_{pre} A'$ and $B \xrightarrow{a?b}_{pre} B'$ – by hypothesis, since Γ isolates A and B, we have $\Gamma \vdash a : Name(Unknown, \_)$. By Open Subject Reduction, we deduce that $N = Name(Unknown, r_b)$, $r_b \succeq res(v)$, $t' \oplus r_b \preceq t$, $\lambda' \preceq \lambda$ and $\lambda(b) \preceq r_b$.

  Therefore, we have

  1. $C = (\nu b : N = v)(A'|B')$
  2. $\Gamma, b : N \vdash A' : Bound(t', \lambda')$
  3. $\Gamma, b : N$ isolates B'
  4. as $\lambda' \preceq \lambda$, $\Sigma_{b\in\mathcal{N}}\lambda'(b) \preceq \Sigma_{b\in\mathcal{N}}\lambda(b)$
  5. as $\lambda' \preceq \lambda$ and $t' \preceq t$, $t' \oplus \Sigma_{b\in\mathcal{N}}\lambda'(b) \preceq t \oplus \Sigma_{b\in\mathcal{N}}\lambda(b)$
  6. as $\lambda(b) \preceq r_b$, we may write that $\forall i \in 1..1$, if $N_i = Name(\_, r_i)$ then $\lambda'(a_i) \preceq r_i$
  7. $t' \oplus \Sigma_{i\in 1..n} r_i \Sigma_{c\in\mathcal{N}} \lambda' \backslash \overrightarrow{a}(c) \preceq t' \oplus r_b \oplus \Sigma_{c\in\mathcal{N}} \lambda' \backslash \{b\}(c) \preceq t \oplus \Sigma_{c\in\mathcal{N}\backslash\{b\}} \lambda'(c) \preceq t \oplus \Sigma_{c\in\mathcal{N}\backslash\{b\}} \lambda(c) \preceq t \oplus \Sigma_{c\in\mathcal{N}} \lambda(c)$
  8. as $B \xrightarrow{a?b}_{pre} B'$, by definition, B' is a client.

  Which proves the case.

- if $A \xrightarrow{a?b}_{pre} A'$ and $B \xrightarrow{\nu b:N=v.a!b}_{pre} B'$ – by hypothesis, since Γ isolates A and B, we have $\Gamma \vdash a : Name(Unknown, \_)$. By Open Subject Reduction, we deduce that $\Gamma, b : Name(Unknown, \_) \vdash A' : Bound(t', \lambda')$, where $t' \preceq t$ and $\lambda' \preceq \lambda$. In addition, by definition, since B is a client, $res(B) = \bot$. Consequently, we have $res(v) = \bot$.

  Therefore, we have

  1. $C = (\nu b : N = v)(A'|B')$
  2. $\Gamma, b : N \vdash A' : Bound(t', \lambda')$
  3. $\Gamma, b : N$ isolates B'
  4. as $\lambda' \preceq \lambda$, $\Sigma_{b\in\mathcal{N}}\lambda'(b) \preceq \Sigma_{b\in\mathcal{N}}\lambda(b)$
  5. as $\lambda' \preceq \lambda$ and $t' \preceq t$, $t' \oplus \Sigma_{b\in\mathcal{N}}\lambda'(b) \preceq t \oplus \Sigma_{b\in\mathcal{N}}\lambda(b)$
  6. as $\lambda(b) \preceq r_b$, we may write that $\forall i \in 1..1$, if $N_i = Name(\_, r_i)$ then $\lambda'(a_i) \preceq r_i$
  7. $t' \oplus \Sigma_{i\in 1..n} r_i \Sigma_{c\in\mathcal{N}}\lambda' \backslash \overrightarrow{a}(c) = t' \oplus \Sigma_{c\in\mathcal{N}}\lambda' \backslash \{b\}(c) \preceq t \oplus \Sigma_{c\in\mathcal{N}\backslash\{b\}}\lambda'(c) \preceq t \oplus \Sigma_{c\in\mathcal{N}}\lambda(c)$
  8. as $B \xrightarrow{a?b}_{pre} B'$, by definition, B' is a client.

  Which proves the case.

Which concludes the theorem. □

E.5 Safety

Lemma 12 (Well-typed terms may run) *If $\Gamma \vdash P : Bound(reserve, \bot_{\mathcal{N}})$ then $res(P) \preceq reserve$.*

This lemma may be proved by induction on the structure of a proof of $\Gamma \vdash P : Bound(reserve, \bot_{\mathcal{N}})$.
□

Proposition 5 (Well-typed terms behave) *If $\Gamma \vdash P : Bound(reserve, \bot_{\mathcal{N}})$ then P is non-exhausting.*

This is a direct corollary of lemma 12 and theorem 3. □

Proposition 6 (Well-typed terms are robust) *If Γ isolates P and $\Gamma \vdash P : Bound(reserve, \bot_{\mathcal{N}})$ then P is robust.*

This is a direct corollary of lemma 12 and Subject Reduction with closed garbage-collection (theorem 4).