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

1 Introduction 3

2 Teπc 6
   2.1 Resource-unaware semantics 6
   2.2 Resources 9

3 From Erlang to Teπc 12
   3.1 Foreign values and operations 12
   3.2 Encoding 13
   3.3 Library 14
   3.4 Notes 14
   3.5 Examples 15
   3.6 Resources 16

4 Proving resource-safety 18
   4.1 Resource usage patterns 18
   4.2 Type judgements 19
   4.3 Results 19

5 Conclusions 22
   5.1 Bottom line 22
   5.2 Related works 22
   5.3 Future works 23

A Notes to the introduction 26

B Language 27
   B.1 Substitution and elimination of dead instructions 27
   B.2 Labelled Transition Semantics 28

C From Erlang to Teπc 30
   C.1 Semantics of embedded primitives 31
   C.2 Expressions 31
   C.3 Pattern-matching 32
   C.4 Library 33
D Proving resource-safety
   D.1 Relations ........................................... 35
   D.2 Type system ........................................ 35

E Proving the type system
   E.1 Lemmas ................................................ 38
   E.2 Subject equivalence .................................... 41
      E.2.1 Initialisation .................................... 41
      E.2.2 Induction step .................................... 42
   E.3 Open Subject Reduction ............................... 43
      E.3.1 Initialisation .................................... 44
      E.3.2 Induction step .................................... 47
   E.4 Subject Reduction under attack ....................... 49
   E.5 Safety ................................................ 52
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 $\pi$-calculus\[15\]. We use and extend this last work to offer a denotational semantics of Core Erlang in Ter$\pi$C\[19\], a targettable/extendable $\pi$-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:

```erlang
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 Teₚᵣₒ, one could model the opening of this file as a transition \( i \xrightarrow{*} (νfile = File))j \), 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 \( (νfile = File))k \xrightarrow{*} l \). 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 \( (νfile_1 = File))j_1 \mid (νfile_2 = File))j_2 \mid \cdots \mid (νfile_{n+1} = File))j_{n+1} \) 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:\(^1\)

```erlang
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

\textbf{Teπc}

\textit{Teπc}, the targettable/extendable \(\pi\)-calculus, is an extension of the controlled \(\pi\)-calculus\cite{18}, itself a variant of the \(\pi\)-calculus\cite{14}, a formal language designed for modelling and studying concurrent and distributed systems with dynamic reconfigurations. To this base, the controlled \(\pi\)-calculus adds a notion of \textit{resources} and resource \textit{limitations}. In turn, \textit{Teπc} introduces notions of \textit{foreign operations} and \textit{foreign values}, used to model underlying virtual machine or operating system calls. Numerous aspects of \textit{Teπc} are left parametric, so as to allow targeting \textit{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 \cite{19}.

\section{Resource-unaware semantics}

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

The syntax of \textit{Teπc} is presented on figure 2.1. Weak references are either variables \((x, y \ldots )\), references to either channels or values \((a, b \ldots )\), or the dangling reference \((\text{∅}, \text{or "null"})\). 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-\text{∅} reference may be used as a communication channel.

\textit{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 \(\pi\)-calculus. In the rest of this report, we will assume Barendregt-style convention on references.

| Weak References   | \(m, n\) ∈ \(\text{Weak}\) | ::= | \(u\) | \(\otimes\) |
| Strong References | \(u, v\) ∈ \(\text{Strong}\) | ::= | \(a\) | \(x\) |
| Names             | \(a, b\) ∈ \(N\)         | ::= |
| Variables         | \(x, y\) ∈ \(V\)         | ::= |
| Processes         | \(P, Q\) ∈ \(\text{Proc}\) | ::= | \((\nu a = v)P\) | \(P\) | \(i + j\) | \(i\) |
| Instructions      | \(i, j\) ∈ \(\text{Code}\) | ::= | nothing | foreign \(u = e\) in \(i\) | spawn \(i\) and \(j\) | once \(m\ x\) do \(i\) | on \(m\ x\) do \(i\) | \(m\ n\) | either \(i\) or \(j\) |
| Foreign values    | \(v, w\) ∈ \(\text{Value}\) | ::= | \(E(v)\) |
|                   | \(v'\) ∈ \(\text{Value}\) | ::= | \(a\) | \(\otimes\) | \(v\) |
| Foreign operations| \(e, f\) ∈ \(\text{Op}\) | ::= | \(m\) | \(E(f)\) |
| Process contexts  | \(C[]\) ∈ \(\text{Proc}\) | ::= | \((\nu a = v)C[]\) | \(C[P]\) | \(P\) | \(C[]\) |
| Value context     | \(V[]\) ∈ \(\text{Val}\) | ::= | \(v\) | \(V[]\) | \(V[,\] | \(v\) | \(F(V[])\) |
| Operation context | \(I[]\) ∈ \(\text{Op}\) | ::= | \(e\) | \(I[]\) | \(I[,\] | \(e\) | \(E(I[])\) |

Figure 2.1: Syntax of \(\text{Te}\pi\text{c.}\)

The resource-unaware semantics of \(\text{Te}\pi\text{c}\) essentially extends that of the \(\pi\)-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
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 $a = v \vdash e \xrightarrow{\sigma} e'$ ("assuming that $a$ references value $v$, a foreign operation in state $e$ will progress and reach state $e'$") and $e \xrightarrow{\sigma} v$ ("a foreign operation in state $e$ is complete and yields a result $v$"), respecting the following criteria:

- if $a = v \vdash e \xrightarrow{\sigma} e'$ then $fv(e) = \emptyset$ and $fn(e') \subseteq fn(e) \cup fn(v) \cup \{\sigma\}$
- if $e \xrightarrow{\sigma} v$ then $fv(e) = \emptyset$ and $fn(v) \subseteq fn(e) \cup \{\sigma\}$
- if $e \xrightarrow{\sigma} m$ and $fn(e) \subseteq \{m\} \cup \{\sigma\}$.

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 Te\pi\text{c} 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 Te\pi\text{c} 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 \pi-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 \texttt{if\_then\_else} in Te\pi\text{c}. Equations 2.1, 2.2 and 2.3 specify the behaviour of a
2.1 Specification of foreign operations

\[
\begin{align*}
\text{IfThenElse}(\text{True}, a, b) & \rightarrow a \\
\text{IfThenElse}(\text{False}, a, b) & \rightarrow b \\
a = v \parallel e & \rightsquigarrow f \\
a = v \parallel \text{IfThenElse}(e, b, c) & \rightsquigarrow \text{IfThenElse}(f, b, c)
\end{align*}
\]

2.2 Resources

In addition, the semantics of \( \text{Te\pi c} \) 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”) \((S, \oplus, \preceq, \bot, \top)\) is a parameter of the language, such that \((S, \oplus)\) is a monoid, \(\preceq\) is a partial order on \(S\), with a minimal element \(\bot\) and a maximal element \(\top\) and \(\oplus\) is monotonic with respect to \(\preceq\) (i.e. \(\forall a, b, c, \text{ if } a \preceq b \text{ then } a \oplus c \preceq b \oplus c\)).

From \(\preceq\), we derive relations \(\prec, \succ, \succeq\).

The resource-constrained semantics of Te\(\pi\)c is parametrised by

- the nature of resources being manipulated \(S\), as defined above,
- a reserve of resources \(\text{reserve}\), element of \(S\),
- a resource signature \(\text{res}\), function from \(\text{Value}\) to \(S\), stable by garbage-collection of references – that is, if \(\text{res}(v) = r\) then \(\text{res}(v\{a \leftarrow \circ\}) = r\) – and by \(\alpha\)-conversion – that is, if \(\text{res}(v) = r\) then \(\text{res}(v\{a \leftarrow b\}) = r\).

We extend this definition of \(\text{res}\) to processes, by

- \(\text{res}(i) = \bot\)
- \(\text{res}(i + j) = \bot\)
- \(\text{res}(P|Q) = \text{res}(P) \oplus \text{res}(Q)\)
- \(\text{res}(\mu a = v)P = \text{res}(P) \oplus \text{res}(v)\)

Definition 2 (Resource-aware semantics) The resource-aware semantics of Te\(\pi\)c is defined by

\[
\frac{P \xrightarrow{\tau_{\text{pre}}} Q \quad \text{res}(Q) \preceq r}{P \xrightarrow{_{\text{fix}}} Q} \quad \frac{P \xrightarrow{\tau_{\text{reserve}}} Q \quad \text{res}(Q) \succ \text{reserve}}{P \xrightarrow{\text{reserve}} Q}
\]

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

Definition 4 (Client process) An process \(P\) is client (or unprivileged) if \(\text{res}(P) = \bot\) and for any \(P'\) such that \(P \xrightarrow{\tau_{\text{pre}}} P'\) 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 \(\text{IfThenElse}\) uses resources, process on \(\text{if_then_else}\ e\ a\ b\ \text{do} \ldots\) is privileged, while process \(\text{if_then_else}\ true\ a\ b\) 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. 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), Numbern (constant

1To 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), \textit{Char} (constant characters), \textit{Strings} (constant strings), \textit{Process} (Erlang processes).

\textbf{arity 1} \textit{Finalise}, \textit{Free} and \textit{Terminate} (used to communicate with the garbage-collector), \textit{Head} and \textit{Tail} (list operations), \textit{Nth} \textit{n} (nth element in a tuple)

\textbf{arity 2} \textit{Cons} (list concatenation), \textit{Same} (name equality predicate)

\textbf{arity n \textit{Tuple\textsubscript{n}}} (tuple constructor)

With the exclusion of \textit{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 Te\pi\textsubscript{c}-Erlang. The encoding itself – essentially source code – is available in the appendices.

For the sake of readability, we extend the syntax of Te\pi\textsubscript{c} to allow polyadic communications, n-ary name creation, as well as n-ary parallel composition

\texttt{spawn \ldots and \ldots and \ldots and choice either \ldots or \ldots or \ldots and a special variable \_\_ (or “don’t care”), which is supposed to never appear free in any term.}

We will also use notation \texttt{new a in \ldots for foreign a = Channel()} in \ldots.

The denotational semantics of modules is a function \(M\\llbracket\cdot\\rrbracket\), while that of functions is \(F\\llbracket\cdot\\rrbracket\), both to the domain of Te\pi\textsubscript{c} instructions. The encoding of expressions with a return channel \(res\) and being evaluated with a process with \texttt{pid self} is a function \(E\\llbracket\cdot\\rrbracket^{res, self}\), also to the domain of Te\pi\textsubscript{c} instructions. These functions, detailed in appendix C.2, are defined much as in \(\pi\)-Erlang, with the following differences:

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

As in \(\pi\)-Erlang, we do not model Erlang’s \texttt{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 \(\pi\)-Erlang we ignore the order of clauses in a \texttt{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 \( C[r,s,i,o,g] \), 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 \( P[in,out] \), the encoding of a pattern matching value in and returning a substitution on channel out, and \( V[\cdot] \), 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 \( \text{Teπ} \) instructions, with the same calling conventions as \( \text{Terπ-} \)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 erlang:exit\_3 and erlang:exit\_4. The encoding then contains a placeholder 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 $E[[e]]_{res,self}$, the executable process for $e$ in LIB is defined as $\text{new } b \in fn(i_e) \rightarrow a \text{ in } (LIB | i_e)$.

**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 $\text{new } \cdot \text{ in } \cdot$, $i_e$ may well be distributed between nodes.

### 3.5 Examples

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

```erlang
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:

```erlang
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
```
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 Te\(\nu\)c 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 \((S_F, \oplus_F, \preceq_F, \bot_F, \top_F)\) as

- \(S_F \triangleq \{ Files(n), n \in \mathbb{N} \cup \{\infty\} \}\)
- \(\forall Files(m), Files(n) \in S_F, Files(m) \oplus_F Files(n) \triangleq Files(m + n)\)
- \(\forall Files(m), Files(n) \in S_F, Files(m) \preceq_F Files(n) \iff m \leq n\)
- \(\bot_F = Files(0), \top_F = Files(\infty).\)

Let us now define the nature of file opening rights \((S_R, \oplus_R, \preceq_R, \bot_R, \top_R)\) as

- \(S_R \triangleq Rights(B)\)
- \(\forall Rights(m), Rights(n) \in S_R, Rights(m) \oplus_R Rights(n) \triangleq Rights(m \lor n)\)
- \(\forall Rights(m), Rights(n) \in S_R, Rights(m) \preceq_R Rights(n) \iff m \Rightarrow n\)
- \(\bot_R = Rights(ff), \top_R = Rights(tt).\)

From these, we derive the definition of set \(S\), the nature of resources for our example, or \((S, \oplus, \preceq, \bot, \top)\), as
• $S \triangleq S_F \times S_R$

• $\forall r = (r_F, r_R), s = (s_F, s_R) \in S_r \triangleq s \triangleq (r_F \oplus s_F, r_R \oplus s_R)$

• $\forall r = (r_F, r_R), s = (s_F, s_R) \in S_r \preceq s \iff r_F \preceq s_F$ and $r_R \preceq s_R$

• $\bot = (\bot_F, \bot_R), \top = (\top_F, \top_R)$.

To this set, we associate a bound 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:

$$res(\text{File}) = 1, tt \quad (3.1)$$

$$res(\text{any other foreign value}) = \bot \quad (3.2)$$

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 \texttt{log\_fragile} and \texttt{log\_robust}, once encoded, define non-exhausting processes. By examining the encoding of \texttt{log\_fragile} and \texttt{log\_robust}, however, we may notice, again, that it is easy to trick \texttt{log\_fragile} into opening any number of files, hence making it non-robust, while \texttt{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 Te\pi\texttt{c}. 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() in i} requires more resources to be execute than \texttt{i}, as it needs to store the result of foreign operation \texttt{E()} – regardless of whether these resources are eventually deallocated. The second pattern is concurrent composition. If two processes \texttt{P} and \texttt{Q} must be executed purely concurrently (i.e. without synchronisations), the system must have enough resources to permit any scheduling of \texttt{P} and \texttt{Q}. By opposition, if only one of two processes \texttt{P} and \texttt{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 \texttt{P \triangleq foreign a = Finalize(b) in i | foreign a' = Finalize(b) in j}, both \texttt{i} and \texttt{j} are triggered only after the deallocation of resources allocated to \texttt{b}. The resources allocated to \texttt{a} must, however, be shared between \texttt{i} and \texttt{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 \textit{cost} in resources, charged to callers whenever they invoke the service. Finally, loss of control permits interaction with untyped processes. Informally, if a channel \texttt{a} may be used to communicate with the outside world, the typed process shouldn’t expect any resource transfer on \texttt{a}, shouldn’t make any assumption about references received from \texttt{a}, and shouldn’t expect that references sent on \texttt{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:

\[
T ::= \text{Bound}(r, \lambda) \\
N ::= \text{Name}(C, r) \\
C ::= \text{Chan}(N, g, d) \\
A ::= \text{Allocation}(r)
\]

Judgement \(\Gamma \vdash P : \text{Bound}(r, \lambda)\) states that, in environment \(\Gamma\), \(P\) 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 \(\lambda\). Judgement \(\Gamma \vdash a : \text{Name}(C, r)\) states that, according to \(\Gamma\), \(a\) is the name of an entity using resource \(r\), with role \(C\). If \(C\) is \(\text{Chan}(N, g, d)\), \(a\) 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 \(\text{Finalizer}\), \(a\) has been declared as foreign \(a = \text{Finalize}(b)\) in \(i\) and should not be used for communication, if \(C\) is \(\text{Unknown}\), \(a\) is untrusted and may serve to communicate with the untyped world, and if \(C\) is \(\text{Ssh}\), \(a\) may not be used to communicate at all. Finally, \(\Gamma \vdash e : \text{Allocation}(r)\) states that, according to \(\Gamma\), \(e\) 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 \(\Gamma \vdash e : \text{Allocation}(r_e)\) and if for some \(a\) and \(v\), we have \(a \equiv v \vdash e \rightsquigarrow f\) then \(\Gamma \vdash f : \text{Allocation}(r_f)\) and \(r_f \leq r_e\). If \(\Gamma \vdash e : \text{Allocation}(r_e)\) and \(e \rightsquigarrow v\) then \(\text{res}(v) \leq r_e\).

With the definition of relations \(\cdot \vdash \cdot \rightsquigarrow \cdot\) and \(\cdot \rightsquigarrow \cdot\) 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 $\Gamma \vdash P : \text{Bound}(t, \lambda)$ and if $(t', \lambda') \succeq (t, \lambda)$ then $\Gamma \vdash P : \text{Bound}(t', \lambda')$.

The lemma is standard and uses standard proofs.

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

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 $\Gamma$ an environment isolating $Q$ and such that $\Gamma \vdash P : \text{Bound}(r, \lambda)$, if there is some $R$ such that $P \mid Q \rightarrow R$ 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

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

In particular, if $Q = \text{nothing}$, it is possible to find $P'$ such that $a : N = \emptyset$ and $Q' = \text{nothing}$. (Subject Reduction without attack). If $\lambda = \bot_N$, we always have $\lambda' = \bot_N$ 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 $a : N = v$ to provide a new type derivation.

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

Informal theorem 3 (Well-typed programs are robust)
Let us consider an Erlang program $e$ and $\text{exe}_e$ its executable. If $\Gamma$ isolates $\text{exe}_e$ and $\Gamma \vdash \text{exe}_e : \text{Bound}(\text{reserve}, \bot_N)$, 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\textunderscore fragile may never be typed in an isolating environment. Informally, an isolating environment requires that name log\textunderscore 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\textunderscore 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 \( \lambda \)-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.

\( \textsc{Te\pi c} \) itself is also related to the Applied \( \pi \)-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

\( \textsc{Te\pi c} \) is a work in progress. In addition to completing the encoding of \( \textsc{Te\pi c} \)-Erlang and to improving the type system and adding real-time constraints, we intend to work on the extraction of \( \textsc{Te\pi c} \) 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


In Erlang/OTP, the definition of \texttt{log\_robust} could be approximated by the following extract:

\begin{verbatim}
log\_robust(Lock, Item) ->
    lock\_acquire(Lock),
    log\_fragile(Item),
    lock\_release(Lock).

start() ->
    Lock = new\_lock(),
    fun (Item) -> log\_robust(Lock, Item) end.
\end{verbatim}
Appendix B

Language

B.1 Substitution and elimination of dead instructions

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

Note that the substitution of $\otimes$ to a name may cause the elimination of a dead instruction, while the substitution of $\otimes$ 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.
\[ ((\nu a = v)P)(m \leftarrow n) \equiv (\nu a = v(m \leftarrow n))(P(m \leftarrow n)) \quad a \notin \{m, n\} \]
\[ (P|Q)(m \leftarrow n) \equiv (P(m \leftarrow n)) | (Q(m \leftarrow n)) \]
\[ \text{nothing}(m \leftarrow n) \equiv \text{nothing} \]
\[ \text{foreign } u = e \text{ in } i(m \leftarrow n) \equiv \text{foreign } u = e\{m \leftarrow n\} \text{ in } (i\{m \leftarrow n\}) \quad u \notin \{m, n\} \]
\[ \text{spawn } i \text{ and } j(m \leftarrow n) \equiv \text{spawn } (i\{m \leftarrow n\}) \text{ and } (j\{m \leftarrow n\}) \]
\[ \text{either } i \text{ or } j(m \leftarrow n) \equiv \text{either } (i\{m \leftarrow n\}) \text{ or } (j\{m \leftarrow n\}) \]
\[ \text{on } a \text{ do } i(a \leftarrow \emptyset) \equiv \text{nothing} \]
\[ \text{on } m'\ x \text{ do } i(m \leftarrow n) \equiv \text{on } m'(\{m \leftarrow n\}) \text{ x do } i(m \leftarrow n) \quad \text{otherwise if } x \notin \{m, n\} \]
\[ \text{once } a \text{ do } i(a \leftarrow \emptyset) \equiv \text{nothing} \]
\[ \text{once } m'\ x \text{ do } i(m \leftarrow n) \equiv \text{once } m'(\{m \leftarrow n\}) \text{ x do } i(m \leftarrow n) \quad \text{otherwise if } x \notin \{m, n\} \]
\[ (a\ n')(a \leftarrow \emptyset) \equiv \text{nothing} \]
\[ (m'\ n')(m \leftarrow n) \equiv (m'(\{m \leftarrow n\})) (n'(\{m \leftarrow n\)) \quad \text{otherwise} \]
\[ a(a \leftarrow b) \equiv b \]
\[ a(m \leftarrow n) \equiv a \quad a \neq m \]
\[ a(a \leftarrow \emptyset) \equiv \emptyset \]
\[ x(m \leftarrow n) \equiv x \quad x \neq m \]
\[ x(x \leftarrow n) \equiv n \]
\[ \emptyset\{m \leftarrow n\} \equiv \emptyset \]
\[ E(\overline{v})(m \leftarrow n) \equiv E(v\{m \leftarrow n\}) \]

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 \(\alpha\) conversion of bound names and variables and with the following rules.

\[
\begin{align*}
\text{E-Commut} & \quad (\nu a = v)(\nu b = w)P \equiv (\nu b = w)(\nu a = v)P \\
& \quad a \neq b, a \notin fn(w), b \notin fn(v) \\
\text{E-Scope} & \quad ((\nu a = v)P)|Q \equiv (\nu a = v)(P|Q) \\
& \quad a \notin fr(Q) \\
\text{E-Par} & \quad P \equiv Q \\
& \quad P|R \equiv Q|R \\
\text{E-New} & \quad (\nu a = v)P \equiv (\nu a = v)Q \\
\end{align*}
\]

While most rules of both structural equivalence and labelled transitions are essentially identical to those of the original \(\pi\)-calculus, resource management is more detailed. Indeed, the labelled transition presemantics of Teπc as the smallest relation \(\overset{\text{pre}}{\rightarrow}\) compatible with the following rules

28
\[
\begin{align*}
\text{R-Par} & \quad \frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} & \quad \text{R-Choice} & \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \\
\text{R-Comm} & \quad \frac{P \xrightarrow{a ? m} P' \quad Q \xrightarrow{a ! m} Q'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q'} & & \\
\text{R-Comm-Close} & \quad \frac{P \xrightarrow{a ? m} P' \quad Q \xrightarrow{v b = v a ! m} Q'}{P \parallel Q \xrightarrow{\alpha} (P' \parallel Q')} & & \\
\text{R-Hide} & \quad \frac{P \xrightarrow{\alpha} P'}{(v a = v) P \xrightarrow{\alpha} (v a = v) P'} & & \\
\text{R-Open} & \quad \frac{P \xrightarrow{b \alpha} P' \quad (v a = v) P \xrightarrow{v a = v \beta a} P'}{b \neq a} & & \\
\text{R-Equiv} & \quad \frac{P \equiv P' \quad Q \equiv Q'}{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\alpha} Q'} & & \\
\text{R-Either} & \quad \text{either i or j} \xrightarrow{\tau} i + j & & \\
\text{R-Send} & \quad \text{a m} \xrightarrow{\alpha} \text{nothing} & & \\
\text{R-Once} & \quad \text{once a do} i \xrightarrow{a ? m} \text{i}\{x \leftarrow m\} & & \\
\text{R-On} & \quad \text{on a do} i \xrightarrow{a ? m} \text{i}\{x \leftarrow m\} \mid \text{on a do} i & & \\
\text{R-Deallocate} & \quad (v a = v) P \xrightarrow{\tau} P\{a \leftarrow \bot\} & & a = v \models P \\
\text{R-Evaluate} & \quad \text{foreign u} = e \text{ in i} \xrightarrow{e = f} \text{foreign} \ u = f \text{ in i} & & \\
\text{R-Fetch} & \quad \frac{P \xrightarrow{e = f} P'}{(v a = v) P \xrightarrow{v a = v} P'} & & f v(e) = f v(f) = \emptyset \\
\text{R-Deref} & \quad \text{foreign x} = e \text{ in i} \xrightarrow{\tau} \text{i}\{x \leftarrow b\} & e \rightarrow b & f v(e) = \emptyset \\
\text{R-Store} & \quad \text{foreign a} = e \text{ in i} \xrightarrow{\tau} (v a = v) i \quad e \rightarrow v & f v(e) = \emptyset
\end{align*}
\]
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

\( E() \rightarrow E() \) \hspace{1cm} (C.1)
\( a \rightarrow a \) \hspace{1cm} (C.2)
\( \odot \rightarrow \odot \) \hspace{1cm} (C.3)

\( \text{Finalize}(\odot) \rightarrow \odot \) \hspace{1cm} (C.4)
\( \text{Free}(\odot) \rightarrow \odot \) \hspace{1cm} (C.5)
\( \text{Terminate}(\odot) \rightarrow \odot \) \hspace{1cm} (C.6)

\( l = \text{Cons}(h, t) \vdash \text{Head}(l) \leadsto h \) \hspace{1cm} (C.7)
\( l = \text{Cons}(h, t) \models \text{Tail}(l) \leadsto t \) \hspace{1cm} (C.8)

\( t = \text{Tuple}_n(a_1, a_2, \ldots, a_n) \models \text{Nth}_p(t) \leadsto a_p \quad 1 \leq p \leq n \) \hspace{1cm} (C.9)
\( \text{Same}(a, a) \rightarrow \text{True} \) \hspace{1cm} (C.10)
\( \text{Same}(a, b) \rightarrow \text{False} \quad a \neq b \) \hspace{1cm} (C.11)
\( \text{IfThenElse}(\text{True}, a, b) \rightarrow a \) \hspace{1cm} (C.12)
\( \text{IfThenElse}(\text{False}, a, b) \rightarrow b \) \hspace{1cm} (C.13)
\( a = v \models e \leadsto f \) \hspace{1cm} (C.14)
\( a = v \models \text{IfThenElse}(e, b, c) \leadsto \text{IfThenElse}(f, b, c) \) \hspace{1cm} (C.15)

Rule C.4 specifies the behaviour of finalisation: if \( a \) is not free in \( i \), \textbf{foreign} \( a = \text{Finalise}(b) \) \textbf{in} \( i \) will wait until \( b \) has been garbage-collected before executing \( i \). Constructors \textit{Free} and \textit{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 \textit{number}_z (for any number \( z \)), \textit{char}_c (for any character \( c \)), \textit{string}_s (for any string \( s \)), \textit{nil} and \textit{cons}, all of which are defined in the library, in Section C.4.
C.3 Pattern-matching

This encoding uses reserved names if samethenelse, nthi (for all integers i) and uncons, all of which are defined in the library, in Section C.4.
\[ \mathcal{C} [p \text{ when } e_1 \to e_2]_{\text{res \text{, self \text{, in \text{, success \text{, go}}}}} = \text{new res', res'', structure in once in } x \text{ do} \]
\[ \quad \text{spawn } \mathcal{P} [p']_{\text{res, structure and once structure } y} \text{ do} \]
\[ \quad \text{spawn } \mathcal{E} [e_1]_{\text{res', self and once res' } z \text{ do ifthenelse } z \text{ success } \circ \text{ res'' self}} \]
\[ \quad \text{and once res'' do spawn } \text{next } x \text{ and once go } x \text{ do } \mathcal{E} [e_2]_{\text{res', self}} \]
\[ \quad \text{(where } y = \overleftarrow{x} =\mathcal{V} [p]) \]

\[ \mathcal{P} [p_1, p_2, \ldots, p_n]_{\text{in, out}} = \text{new out}_1, out_2, \ldots, out_n \text{ in} \]
\[ \quad \text{spawn new res}_1 \text{ in} \]
\[ \quad \text{spawn nth}_1 \text{ in res}_1 \text{ self and} \]
\[ \quad \text{once res}_1 \text{ elem}_1 \text{ do } \mathcal{P} [p_1]_{\text{elem}_1, out_1} \]
\[ \quad \text{and ...} \]
\[ \quad \text{and new res}_n \text{ in} \]
\[ \quad \text{spawn nth}_n \text{ in res}_n \text{ self and} \]
\[ \quad \text{once res}_n \text{ elem}_n \text{ do } \mathcal{P} [p_n]_{\text{elem}_n, out_n} \]
\[ \quad \text{and once out}_1 \overrightarrow{x}_1 \text{ do } \ldots \text{ once out}_n \overrightarrow{x}_n \text{ do } \text{out } \overrightarrow{x}_1 \overrightarrow{x}_2 \ldots \overrightarrow{x}_n \]

\[ \mathcal{P} [a]_{\text{in, out}} = \text{if samethenelse } a \text{ in out } \circ \]

\[ \mathcal{P} [x]_{\text{in, out}} = \text{out } x \]

\[ \mathcal{P} [[p_1, p_2, \ldots, p_n]_{\text{p}_{n+1}}]_{\text{in, out}} = \text{new out}_1, out_2, \ldots, out_{n+1} \text{ in} \]
\[ \quad \text{spawn} \]
\[ \quad \text{new hd}_1, tl_1 \text{ in spawn uncons in } \text{hd}_1 \text{ tl}_1 \text{ and once } \text{hd}_1 \text{ h}_1 \text{ do } \mathcal{P} [p_1]_{\text{h}_1, out_1} \]
\[ \quad \text{and once } \text{tl}_1 \text{ t}_1 \text{ do new } \text{hd}_2, \text{tl}_2 \text{ in} \]
\[ \quad \text{spawn uncons } t_1 \text{ hd}_2 \text{ tl}_2 \text{ and once } \text{hd}_2 \text{ h}_2 \text{ do } \mathcal{P} [p_2]_{\text{h}_2, out_2} \]
\[ \quad \ldots \]
\[ \quad \text{ spawn uncons } t_n \text{ hd}_{n+1} \text{ tl}_{n+1} \text{ and once } \text{hd}_{n+1} \text{ h}_{n+1} \text{ do } \mathcal{P} [p_{n+1}]_{\text{h}_{n+1}, out_{n+1}} \]
\[ \quad \text{and once out}_1 \overrightarrow{x}_1 \text{ do } \ldots \text{ once out}_{n+1} \overrightarrow{x}_{n+1} \text{ do } \text{out } \overrightarrow{x}_1 \overrightarrow{x}_2 \ldots \overrightarrow{x}_{n+1} \]

\[ \mathcal{V} [p_1, p_2, \ldots, p_n] = \mathcal{V} [p_1] \mathcal{V} [p_2] \ldots \mathcal{V} [p_n] \]

\[ \mathcal{V} [v] = \{v\} \]

\[ \mathcal{V} [a] = \{\} \]

\[ \mathcal{V} [[p_1, p_2]] = \mathcal{V} [p_1] \mathcal{V} [p_2] \]

### C.4 Library

This is an abridged version of the library.
• Message emission
  on erlang : \texttt{send}\, x_1\, x_2\, res, self \to spawn\, res\, x_2\, and\, x_1\, x_2
• Identification of the current process
  on erlang : \texttt{self}\, res\, self \to do\, res\, self
• Process spawning
  on erlang : \texttt{spawn}\, \_\, arg\, res\, self \to do\, new\, res'\, in\, foreign\, self' = Process() \in
  \texttt{spawn}\, \_\, arg\, self'\, res'\, and\, res\, self' \to and\, once\, res' \to do\, nothing
• Exceptions/killing
  on erlang : \texttt{exit}\, 3\, reason\, res\, self \to do\, spawn\, kill\, self\, reason\, and\, res\, true
  on erlang : \texttt{exit}\, 4\, pid\, reason\, res\, self \to do\, spawn\, kill\, pid\, reason\, and\, res\, true
  on kill\, pid\, reason\, do\, new\, do_kill, trap\, exit\, in\, foreign\, next = \texttt{IfThenElse}(False,\, trap\, exit,\, do\, kill) \in
  \texttt{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 \to do\, foreign\, file = File() \in\, res\, file
  on close\, name\, res\, self \to do\, spawn\, free\, file \and\, foreign\, a = Finalize(file) \in\, res\, true
  on write\, name\, content\, res\, self \to do\, res\, true
• Waitlines (simplified)
  on new\, lock\, res\, self \to do\, foreign\, a = Lock() \in\, spawn\, a \and\, res\, a
  on acquire\, lock\, l\, res\, self \to do\, once\, l \to do\, res\, true
  on release\, lock\, l\, res\, self \to do\, spawn\, l \and\, res\, true
• Primitive values
  on number\, z\, res\, self \to do\, foreign\, a = Number_z() \in\, res\, a
  on string\, s\, res\, self \to do\, foreign\, a = String_s() \in\, res\, a
  on char\, c\, res\, self \to do\, foreign\, a = Char_c() \in\, res\, a
  on nil\, res\, self \to do\, foreign\, a = Nil() \in\, res\, a
  on cons\, h\, t\, res\, self \to 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\, x_1\, x_2\ldots\, x_n\, res\, self \to do\, foreign\, a = Tuple_n(x_1,\ldots,\, x_n) \in\, res\, a
  on nth\, x\, res\, self \to do\, foreign\, a = Nth(x) \in\, res\, a
  on ifthenelse\, x\, y\, z\, res\, self \to do\, foreign\, next = \texttt{IfThenElse}(x,\, y,\, z) \in\, res\, next
  on ifsamethenelse\, w\, x\, y\, z\, res\, self \to do\, foreign\, next = \texttt{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() \equiv foreign\, c = Terminate(b) \in\, i \mid P
  b = File() \equiv foreign\, a = Free(b) \in\, i \mid P
Appendix D

Proving resource-safety

D.1 Relations

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

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

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

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

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 $\text{foreign } a : N = e \text{ in } \cdots$ and $(\nu a : N = v)$. When necessary, we write $a \mapsto r$ for the function defined on $N$ whose value is $r$ for $a$ and $\perp$ for everything else, $\lambda\{x\}$ for the function defined on $N$ whose value on $x$ is $\perp$ and identical to that of $\lambda$ everywhere else, and $\perp_N$ for the function defined on $N$ whose value is $\perp$ everywhere. We extend the definition to always have $\lambda(\ominus) = \perp$.

Rule T-Nil states that the terminated process is always typable. Rule T-Repl permits the typing of $\text{on } \cdot \text{ do } \cdots$ 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
Figure D.1: Type system for resource guarantees (non-foreign terms).
T-Alloc-Value \quad \Gamma \vdash v : Allocation(r) \quad \text{res}(v) \trianglelefteq r

T-Alloc-Nothing \quad \Gamma \vdash e : Allocation(r) \quad e \notin \text{Value}, \quad fn(e) \subseteq \Gamma

T-Finalize \quad \Gamma, a : N \vdash i : Bound(t, \lambda) \quad N = Name(Finalize, \bot) \quad t_r \oplus r \trianglerighteq t \quad (t', \lambda') \trianglelefteq (t, \lambda) + m \mapsto r \quad \lambda(a) = \bot

\Gamma \vdash \text{foreign } a : N \trianglelefteq \text{Finalize}(m) \in i : Bound(t', \lambda')

\Gamma \vdash m : N \quad \Gamma \vdash n : N \quad (t', \lambda') \trianglerighteq (t, \lambda \setminus \{a\}) \quad \ominus(m \mapsto \lambda(a)) \quad \ominus(n \mapsto \lambda(a))

T-If \quad \Gamma \vdash \text{foreign } a : N \trianglelefteq \text{IfThenElse}(e, m, n) \in i : Bound(t', \lambda')

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 be 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 $\text{T}_{\pi c}$, 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 $\Gamma, b : T_b \vdash e[a \leftarrow b] : T$.

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

- **T-Null** We have $e = \emptyset = e[a \leftarrow b]$. Trivial.
- **T-Ref** We have $e = a$ and $T = T_a$. Trivial.
- **T-Alloc-Nothing** We have any $T$. Trivial.

$\square$

**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 $c\pi$.

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 $\Gamma, x : N \vdash P : Bound(t, \lambda)$ and $\Gamma \vdash m : N$ then $\Gamma \vdash P\{x \leftarrow m\} : Bound(t, \lambda\{x\}) \oplus m \rightarrow \lambda(x)$

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

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

We have the following type derivation

<table>
<thead>
<tr>
<th>Typing $P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma, x : \text{Name}(., r_a), a : M \vdash i$ :</td>
</tr>
<tr>
<td>$\Rightarrow \Gamma, x : \text{Name}(., r_a) \vdash \text{foreign } a : M = \text{Finalize}(x) \text{ in } i : Bound(t, \lambda)$</td>
</tr>
<tr>
<td>Where</td>
</tr>
<tr>
<td>$t \geq t_i$</td>
</tr>
<tr>
<td>$\lambda' \geq \lambda_i \oplus a \rightarrow r$</td>
</tr>
<tr>
<td>$t_r \oplus r \geq t_i$</td>
</tr>
<tr>
<td>$\lambda_i(a) = \perp$</td>
</tr>
</tbody>
</table>

Therefore, we may have

<table>
<thead>
<tr>
<th>Typing $i{x \leftarrow \ominus}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma, x : \text{Name}(., r_a), a : M \vdash i$ :</td>
</tr>
<tr>
<td>$\Rightarrow \Gamma, a : M \vdash i{x \leftarrow \ominus} : Bound(t_i, \lambda_i)$</td>
</tr>
<tr>
<td>See above</td>
</tr>
<tr>
<td>$\Rightarrow \Gamma \vdash \text{foreign } a : M = \text{Finalize}(\ominus) \text{ in } i{x \leftarrow \ominus} : Bound(t''', \lambda''')$</td>
</tr>
<tr>
<td>From T-Finalize</td>
</tr>
<tr>
<td>Where</td>
</tr>
<tr>
<td>$t''' = t_r \oplus r$</td>
</tr>
<tr>
<td>$t''_r = t \oplus \lambda_i(x)$</td>
</tr>
<tr>
<td>$\lambda''' = \lambda_i{x}$</td>
</tr>
</tbody>
</table>

As $t''' = t \oplus \lambda_i(x) \oplus r$, and $\lambda(x) \geq \lambda_i \oplus a \rightarrow r$, we have $t''' \geq t \oplus \lambda(x)$.

In addition, $\lambda''' = \lambda_i\{x\} = (\lambda_i \oplus (x \rightarrow r))\{x\} \leq \lambda'$.

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

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 $\Gamma \vdash P : Bound(t, \lambda)$ and $\lambda' \geq \lambda$ then $\Gamma \vdash P : Bound(t, \lambda')$.
Lemma 8 (Strengthening deallocations) If $\Gamma \vdash P : \text{Bound}(t, \lambda)$ and $a \notin fn(P)$ then $\Gamma \vdash P : \text{Bound}(t, \lambda \setminus \{a\})$.

Trivial. □

Lemma 9 (Foreign operation progress) Let $e$ and $e'$ be two foreign operations such that $\_ \vdash e \rightsquigarrow e'$. Let $\Gamma$ be an environment such that $\Gamma \vdash \text{foreign } x : N = e \text{ in i : } T$. 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 $\_ \vdash \_ \rightarrow \_ \rightarrow \_.$

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

case $t = \text{Tuple}_n(a_1, a_2, \ldots, a_n) \vdash \text{Nth}(t) \rightsquigarrow a_p, 1 \leq p \leq n$: everything has type $\text{Allocation}(\_)$

case $a = v \vdash e \rightsquigarrow f$

$\quad a = v \vdash \text{IfThenElse}(e, b, c) \rightsquigarrow \text{IfThenElse}(f, b, c)$: everything has type $\text{Allocation}(\_)$

□

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

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

We prove this by examining relations $\_ \vdash \_ \rightsquigarrow \_ \rightsquigarrow \_ \rightsquigarrow \_.$

• case $a \rightsquigarrow a$: trivial

• case $\text{IfThenElse}(\text{True}, a, b) \rightsquigarrow a$: trivial

• case $\text{IfThenElse}(\text{False}, a, b) \rightsquigarrow b$: trivial

□

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 $\Gamma$ 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. □
E.2 Subject equivalence

Proposition 1 (Subject equivalence) If \( \Gamma \) 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:

\[
\begin{align*}
E-Trans & \quad \frac{P \equiv Q}{P \equiv R} \quad \frac{Q \equiv R}{P \equiv P} \\
E-Par-Commut & \quad \frac{P \equiv Q \equiv P}{P \equiv Q \\
E-Par-Assoc & \quad \frac{(P|Q)|R \equiv P|(Q|R)}{P \equiv P \equiv P} \\
E-Choice-Commut & \quad \frac{P + Q \equiv Q + P}{P \equiv P \\
E-Choice-Assoc & \quad \frac{(P + Q) + R \equiv P + (Q + R)}{P \equiv P} \\
E-Alpha & \quad \frac{(\nu a = v)P \equiv (\nu b = v)P[a \leftarrow b]}{b \notin fn(P)} \quad b \notin fn(P)
\end{align*}
\]

E.2.1 Initialisation

E-Refi 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:

\[
\text{Typing } ((\nu a : \text{Name}(C, r_a) = v)|P)
\]
Γ, a : Name(C, r_a) ⊢ P : Bound(t_P, λ_P)  
\implies Γ ⊢ (νa : Name(C, r_a) = v) P : Bound(t_1, λ_1) 
Where  
res(v) ≤ r_a  
λ_P(a) ≤ r_a  
λ_1 ≥ λ\{a\}  
t_1 ≥ r_a ⊕ t_P

By hypothesis  
⇒ Γ ⊢ (νa : Name(C, r_a) = v) P : Bound(t_1, λ_1)

Typing ((νa = v : N)P)|Q

Γ ⊢ (νa : Name(C, r_a) = v) P : Bound(t_1, λ_1)  
⇒ Γ ⊢ (νa = v : N)P)|Q : Bound(t_2, λ_2) 
Where  
t_2 ≥ t_1 ⊕ t_Q  
λ_2 ≥ λ_1 ⊕ λ_Q

By lemma 8, as a /∈ fn(Q), we may assume that λ_Q(a) = ⊥. Consequently, we may also derive:

Typing Q

Γ ⊢ Q : Bound(t_Q, λ_Q)  
⇒ Γ, a : Name(C, r_a) ⊢ Q : Bound(t_Q, λ_Q)

Typing P|Q

Γ, a : Name(C, r_a) ⊢ P : Bound(t_P, λ_P)  
Γ, a : Name(C, r_a) ⊢ Q : Bound(t_Q, λ_Q)  
⇒ Γ, a : Name(C, r_a) ⊢ P|Q : Bound(t_3, λ_3) 
Where  
t_3 = t_P ⊕ t_Q  
λ_3 = λ_P ⊕ λ_Q

Gamma ⊢ (νa = v : N)(P|Q) : Bound(t_2, λ_2)  
⇒ Γ ⊢ (νa = v : N)(P|Q) : Bound(t_2, λ_2)

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

### E.3 Open Subject Reduction

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

- if $l = \tau$ or $l = e \rightarrow f$, we have $\Gamma \vdash A' : \text{Bound}(r', \lambda')$, where $\Sigma_{c \in \mathbb{N}} \lambda'(c) \leq \Sigma_{c \in \mathbb{N}} \lambda'(c)$, $r' \oplus \Sigma_{c \in \mathbb{N}} \lambda'(c) \leq r \oplus \Sigma_{c \in \mathbb{N}} \lambda'(c)$

- if $l = a?b$
  - either $\Gamma \vdash a : \text{Name}(\text{Chan}(N, g_a, d_a), \_)$, $\Gamma, b : N \vdash A' : \text{Bound}(r', \lambda')$, $r' \leq r \oplus g_a \oplus d_a$ and $\lambda' \leq \lambda$.
  - or $\Gamma \vdash a : \text{Name}(\text{Unknown}, \_)$, $\Gamma, b : \text{Name}(\text{Unknown}, \_ \vdash A' : \text{Bound}(r', \lambda')$, $r' \leq r$ and $\lambda' \leq \lambda$

- if $l = a?\oplus$, then
  - either $a \in \Gamma$, $\Gamma \vdash a : \text{Name}(\text{Chan}(N, g_a, d_a), \_)$, $r' \leq r \oplus g_a \oplus d_a$ and $\lambda' \leq \lambda$.
  - or $\Gamma \vdash a : \text{Name}(\text{Unknown}, \_)$, $r' \leq r$ and $\lambda' \leq \lambda$

- if $l = a!b$, then
  - either $\Gamma \vdash a : \text{Name}(\text{Chan}(N, g_a, d_a), \_)$, $\Gamma \vdash b : N$, $r' \oplus g_a \leq r$, $\lambda' \leq \lambda$.
  - or $\Gamma \vdash a : \text{Name}(\text{Unknown}, \_)$, $\Gamma \vdash b : (\text{Unknown}, \_)$, $r' \leq r$ and $\lambda' \leq \lambda$

- if $l = a!\oplus$, then
  - either $\Gamma a \vdash \text{Name}(\text{Chan}(\_, g_a, d_a), \_)$, $r' \oplus g_a \oplus d_a \leq r$, $\lambda' \leq \lambda$
  - or $\Gamma \vdash a : \text{Name}(\text{Unknown}, \_)$, $r' \leq r$ and $\lambda' \leq \lambda$

- if $l = vb : N = v.a!b$, then
  - either $\Gamma a \vdash \text{Name}(\text{Chan}(N, g_a, d_a), \_)$, $N = \text{Name}(\_, r_b)$, $r_b \geq \text{res}(v)$, $r' \oplus g_a \oplus r_b \leq r$, $\lambda'(b) \oplus d_a \leq r_b$.
  - or $\Gamma \vdash a = \text{Name}(\text{Unknown}, \_)$, $N = \text{Name}(\text{Unknown}, r_b)$, $r_b \geq \text{res}(v)$, $r' \oplus r_b \leq r$, $\lambda' \leq \lambda$ and $\lambda(b) \leq r_b$.

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

43
E.3.1 Initialisation

R-Spawn Label is \( \tau \). Rules T-SPAWN and T-PAR produce the same typings from the same set of hypothesis.

R-Either Label is \( \tau \). Rules T-EITHER and T-CHOICE produce the same typings from the same set of hypothesis.

R-Deallocate Label is \( \tau \). We have \( A = (\nu a : \text{Name}(C, r_a) = v)P \) and \( A' = P[a \leftarrow \odot] \). By T-New, we also have \( \Gamma, a : \text{Name}(C, r_a) \vdash A : \text{Bound}(r_P, \lambda_P) \), with \( r \geq r_P \oplus r_a \), \( \lambda \geq \lambda_P \setminus \{a\} \), \( \lambda_P(a) \leq r_a \) and \( \text{res}(v) \leq r_a \). By lemma 5, if \( \Gamma, a : N \vdash P : \text{Bound}(r_P, \lambda_P) \), then \( \Gamma \vdash P[a \leftarrow \odot] : \text{Bound}(r_P', \lambda_P') \), where \( \Sigma_{b \in N} \lambda_P'(b) \leq \Sigma_{b \in N} \lambda_P(b) \) and \( r_P' \oplus \Sigma_{b \in N} \lambda_P'(b) \leq r_P \Sigma_{b \in N} \lambda_P(b) \).
Which proves the case.

R-Deref Label is \( \tau \). We have \( A = \text{foreign} \ x : N = e \ in \ i \) and \( A' = i\{x \leftarrow m\} \) where \( m = \odot \) or \( m \not\in \text{fn}(i) \) and \( N = \text{Name}_{\odot}r_e \)

Let us consider the typings of \( A \) and deduce a typing of \( A' \)

\[
\begin{array}{ll}
\text{Typing } A \\
\Gamma \vdash e : \text{Allocation}(r_e) \\
\Gamma, x : N \vdash i : \text{Bound}(t_i, \lambda_i) \\
\Rightarrow \Gamma \vdash A : \text{Bound}(t, \lambda) \quad \text{From T-LET} \\
\text{Where} \\
t \geq t_i \oplus r_e \\
\lambda \geq \lambda_i \setminus \{x\} \\
\lambda_i(x) \leq r_e \\
\end{array}
\]

\[
\begin{array}{ll}
\text{Typing } A' \text{ case } m = b \neq \odot \\
\Gamma, x : N \vdash i : \text{Bound}(t_i, \lambda_i) \quad \text{By hypothesis} \\
\Rightarrow \Gamma \vdash i\{x \leftarrow b\} : \text{Bound}(t_i, \lambda_i \setminus \{x\}) \quad \text{From Substitution} \\
\text{Since} \\
\Gamma(b) = N \quad \text{From Lemma 10} \\
\Rightarrow \Gamma \vdash A' : \text{Bound}(t, \lambda) \quad \text{From Weakening} \\
\text{Since} \\
t \geq t_i \oplus r_e \\
\lambda \geq \lambda_i \setminus \{x\} \\
\end{array}
\]

\[
\begin{array}{ll}
\text{Typing } A' \text{ case } m = \odot \\
\Gamma, x : N \vdash i : \text{Bound}(t_i, \lambda_i) \quad \text{By hypothesis} \\
\Rightarrow \Gamma \vdash i\{x \leftarrow \odot\} : \text{Bound}(t_i \oplus \lambda_i(x), \lambda_i \setminus \{x\}) \quad \text{From Substitution} \\
\Rightarrow \Gamma \vdash A' : \text{Bound}(t, \lambda) \quad \text{From Weakening} \\
\text{Since} \\
\lambda_i(x) \leq r_e \\
\lambda \geq \lambda_i \setminus \{x\} \\
\end{array}
\]

\( \odot \)

Which proves the case.

R-Store Label is \( \tau \). We have \( A = \text{foreign} \ a : N = e \ in \ i \) and \( A' = (\nu a : N = \)
We have $r_\text{o}$, where $N = \text{Name}(\_, r_e)$. Let us consider the typings of $A$ and deduce a typing of $A'$

<table>
<thead>
<tr>
<th>Typing $A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash e : \text{Allocation}(r_e)$</td>
</tr>
<tr>
<td>$\Gamma, a : N \vdash i : \text{Bound}(t_i, \lambda_i)$</td>
</tr>
<tr>
<td>$\Rightarrow \Gamma \vdash A : \text{Bound}(t, \lambda)$ From T-LET</td>
</tr>
<tr>
<td>Where $t \geq t_i \oplus r_e$</td>
</tr>
<tr>
<td>$\lambda \geq \lambda_i \setminus {a}$</td>
</tr>
<tr>
<td>$\lambda_i(a) \preceq r_e$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Typing $A'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma, a : N \vdash i : \text{Bound}(t_i, \lambda_i)$ By hypothesis</td>
</tr>
<tr>
<td>Since $\lambda \geq \lambda_i \setminus {a}$</td>
</tr>
<tr>
<td>$\lambda_i(a) \preceq r_e$</td>
</tr>
<tr>
<td>$t \geq t_i \oplus r_e$</td>
</tr>
<tr>
<td>$\Rightarrow \Gamma \vdash (\nu a : N = v)i : \text{Bound}(t, \lambda)$ From T-New</td>
</tr>
</tbody>
</table>

Which proves the case.

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

- If $\Gamma \vdash a : \text{Name} \bigl( \text{Chan}(N, g_a, d_a), \_ \bigr)$ and $m \neq \varnothing$, we know that $t \geq g_a$ and $\lambda \geq b \mapsto d_a$. As nothing may have any $\text{Bound}(\_ \bigr)$ type, $\text{Bound}(\bot, \bot_N)$ 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 $\Gamma \vdash a : \text{Name} \bigl( \text{Chan}(N, g_a, d_a), \_ \bigr)$ and $m = \varnothing$, we know that $t \geq g_a \oplus d_a$. As nothing may have any $\text{Bound}(\_ \bigr)$ type, $\text{Bound}(\bot, \bot_N)$ is a possible choice, and it matches the property we need to prove.

- if $\Gamma \vdash a : \text{Name}(\text{Unknown}, \_ \bigr)$ and $m \neq \varnothing$, we use $\text{Bound}(\bot, \bot_N)$ In addition, by T-SEND\text{UNKNOWN}, we have $\Gamma \vdash b : \text{Name}(\text{Unknown}, \_ \bigr)$.

- if $\Gamma(a) = \text{Name}(\text{Unknown}, \_ \bigr)$ and $m = \varnothing$, we use again $\text{Bound}(\bot, \bot_N)$.

Which proves the case.

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

Case 1 If $\Gamma \vdash a : \text{Name} \bigl( \text{Chan}(N, g_a, d_a), \_ \bigr)$ and $m \neq \varnothing$, we have $\Gamma, x : N \vdash i : \text{Bound}(t_i, \lambda_i)$, where $t \oplus g_a \geq t_i$, $\lambda \geq \lambda_i \setminus \{x\}$ and $d_a \geq \lambda_i(x)$.
By hypothesis, we have $\Gamma \vdash b : N$. Hence, by lemma 5 (substitution), we conclude that $\Gamma \vdash i(x \leftarrow b) : \text{Bound}(t_i, \lambda_i \setminus \{x\}) \oplus (b \mapsto \lambda_i(x))$. With $t' = t_i$ and $\lambda' = \lambda_i \setminus \{x\} \oplus (b \mapsto \lambda_i(x))$, we conclude that $t' \sqsubseteq t \oplus g_a$ and $\lambda' \leq \lambda \oplus b \mapsto d_a$, which proves the case.

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

**Case 3** If $\Gamma \vdash a : \text{Name}(\text{Unknown}, \_)$ and $m \neq \odot$, by $\text{T-RECEIVE}\text{UNKNOWN}$, we have $\Gamma, x : \text{Name}(\text{Unknown}, \_), i : \text{Bound}(t_i, \lambda_i)$, where $t \geq t_i$, $\lambda \geq \lambda_i$ and $\lambda_i(x) = \perp$. By hypothesis, we have $\Gamma \vdash b : \text{Name}(\text{Unknown}, \_)$.

Hence, by lemma 5 (substitution), we conclude that $\Gamma \vdash i(x \leftarrow b) : \text{Bound}(t_i, \lambda_i \setminus \{x\}) \oplus (b \mapsto \lambda_i(x))$. With $t' = t_i$ and $\lambda' = \lambda_i \setminus \{x\} \oplus (b \mapsto \lambda_i(x))$, we conclude that $t' \leq t$ and $\lambda' \leq \lambda \setminus \{x\}$, which proves the case.

**Case 4** If $\Gamma \vdash a : \text{Name}(\text{Unknown}, \_)$ and $m = \odot$, we have $\Gamma, x : \text{Name}(\text{Unknown}, \_), i : \text{Bound}(t_i, \lambda_i)$, where $t \geq t_i$ and $\lambda \geq \lambda_i$ and $\lambda(x) = \perp$. Hence, by lemma 5 (substitution), we conclude that $\Gamma \vdash i(x \leftarrow b) : \text{Bound}(t_i, \lambda_i \setminus \{x\}) \oplus (b \mapsto \lambda_i(x))$. With $t' = t_i$ and $\lambda' = \lambda_i \setminus \{x\} \oplus (b \mapsto \lambda_i(x))$, we conclude that $t' \leq t$ and $\lambda' \leq \lambda \setminus \{x\}$, which proves the case.

**R-On** We have $A = \text{on} a x \text{ do } i$ and $A' = i\{x \leftarrow \} \mid \text{on} a x \text{ do }$. We also have $\Gamma \vdash A : \text{Bound}(t, \lambda)$ hence either $\Gamma \vdash a : \text{Name}(\text{Chan}(N, g_a, d_a), \_)$ for some $g_a$ and $d_a$ (by $\text{T-RECEIVE}\text{UNKNOWN}$) or $\Gamma \vdash a : \text{Name}(\text{Unknown}, \_)$ (by $\text{T-RECEIVE}\text{UNKNOWN}$). In addition, by $\text{T-REPL}$, we have $\Gamma \vdash \text{once} a x \text{ do } i : \text{Bound}(\perp, \perp_N)$.

- If $\Gamma \vdash a : \text{Name}(\text{Chan}(N, g_a, d_a), \_)$, $m \neq \odot$ and $\Gamma \vdash b : N$, we have $\Gamma, x : N \vdash i : \text{Bound}(\perp, \perp_N)$. Hence, by lemma 5 (substitution), we have $\Gamma \vdash i(x \leftarrow b) : \text{Bound}(\perp, \perp_N)$. By $\text{T-PAR}$, we deduce that $\Gamma \vdash A' : \text{Bound}(t, \lambda)$. Which proves the case.

- If $\Gamma \vdash a : \text{Name}(\text{Chan}(N, g_a, d_a), \_)$ and $m = \odot$, we have $\Gamma, x : N \vdash i : \text{Bound}(\perp, \perp_N)$. Hence, by lemma 5 (substitution), we have $\Gamma \vdash i(x \leftarrow m) : \text{Bound}(\perp, \perp_N)$. By $\text{T-PAR}$, we deduce that $\Gamma \vdash A' : \text{Bound}(t, \lambda)$. Which proves the case.

- If $\Gamma \vdash a : \text{Name}(\text{Unknown}, \_)$, $m \neq \odot$ and $\Gamma \vdash b : \text{Name}(\text{Unknown}, \_)$, we have $\Gamma, x : \text{Name}(\text{Unknown}, \_), i : \text{Bound}(\perp, \perp_N)$. Hence, by lemma 5 (substitution), we have $\Gamma \vdash i(x \leftarrow b) : \text{Bound}(\perp, \perp_N)$. By $\text{T-PAR}$, we deduce that $\Gamma \vdash A' : \text{Bound}(t, \lambda)$. Which proves the case.

- If $\Gamma \vdash a : \text{Name}(\text{Unknown}, \_)$ and $m = \odot$, we have $\Gamma, x : \text{Name}(\text{Unknown}, \_), i : \text{Bound}(\perp, \perp_N)$. Hence, by lemma 5 (substitution), we have $\Gamma \vdash i(x \leftarrow m) : \text{Bound}(\perp, \perp_N)$. By $\text{T-PAR}$, we deduce that $\Gamma \vdash A' : \text{Bound}(t, \lambda)$. Which proves the case.

46
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

\[
P \xrightarrow{a|m_{pr}} P', Q \xrightarrow{a|m_{pr}} Q', \Gamma \vdash P : \text{Bound}(t_p, \lambda_p), \Gamma \vdash Q : \text{Bound}(t_q, \lambda_q),
\]

\[
\Gamma \vdash a : \text{Name}(C, \_). \text{ and } m \neq \emptyset.
\]

Let us write \( b = m \).

By induction hypothesis, we have \( \Gamma \vdash b : N, \Gamma \vdash a : \text{Name}(Chan(N, g_a, d_a), \_), \Gamma, b : N \vdash P' : \text{Bound}(t'_p, \lambda'_p), t'_p \preceq t_p \oplus g_a, \lambda'_p \preceq \lambda_p \oplus b \mapsto d_a \).

\[
t'_q \oplus g_a \preceq t_q, \lambda'_q \preceq b \mapsto d_a = \lambda_q.
\]

By T-PAR, we deduce \( \Gamma \vdash A' : \text{Bound}(t'_p \oplus t'_q, \lambda'_p \oplus \lambda'_q) \). 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 \( \lambda'_q \preceq b \mapsto d_a = \lambda_q \), 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

\[
P \xrightarrow{a|m_{pr}} P', Q \xrightarrow{a|m_{pr}} Q', \Gamma \vdash P : \text{Bound}(t_p, \lambda_p), \Gamma \vdash Q : \text{Bound}(t_q, \lambda_q),
\]

\[
\Gamma \vdash a : \text{Name}(C, \_). \text{ and } m = \emptyset.
\]

By induction hypothesis, we have \( \Gamma \vdash a : \text{Name}(Chan(N, g_a, d_a), \_), \Gamma \vdash P' : \text{Bound}(t'_p, \lambda'_p), t'_p \preceq t_p \oplus g_a \oplus d_a, \lambda'_p \preceq \lambda_p \oplus \lambda'_q \preceq t_q \) and \( \lambda'_q \preceq \lambda_q \).

By T-PAR, we deduce \( \Gamma \vdash A' : \text{Bound}(t'_p \oplus t'_q, \lambda'_p \oplus \lambda'_q) \). 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 \( \lambda'_p \preceq \lambda_p \) 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

\[
P \xrightarrow{a|m_{pr}} P', Q \xrightarrow{a|m_{pr}} Q', \Gamma \vdash P : \text{Bound}(t_p, \lambda_p), \Gamma \vdash Q : \text{Bound}(t_q, \lambda_q),
\]

\[
\Gamma \vdash a : \text{Name}(Unknown, \_). \text{ and } m \neq \emptyset.
\]

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

\[
P \xrightarrow{a|m_{pr}} P', Q \xrightarrow{a|m_{pr}} Q', \Gamma \vdash P : \text{Bound}(t_p, \lambda_p), \Gamma \vdash Q : \text{Bound}(t_q, \lambda_q),
\]

\[
\Gamma \vdash a : \text{Name}(C, \_). \text{ and } m = \emptyset.
\]

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

R-Comm-Close 1 Let us consider the case where \( A = P|Q, \ A' = (vb : N = v)(P'|Q') \), where \( P \xrightarrow{a|m_{pr}} P', Q \xrightarrow{(vb) N = v.a|m_{pr}} Q', \Gamma \vdash P : \text{Bound}(t_p, \lambda_p), \Gamma \vdash Q : \text{Bound}(t_q, \lambda_q), \Gamma \vdash a : \text{Name}(C, \_). \) and \( m \neq \emptyset \).

Let us write \( b = m \) and \( N = \text{Name}(\_r_b) \).

By induction hypothesis, we have \( r_b \preceq \text{res}(v), \Gamma \vdash a : \text{Name}(Chan(N, g_a, d_a), \_), \Gamma, b : N \vdash P' : \text{Bound}(t'_p, \lambda'_p), t'_p \preceq t_p \oplus g_a, \lambda'_p \preceq \lambda_p \oplus b \mapsto d_a, \)

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

By T-PAR and T-RES, we may therefore prove that $\Gamma \vdash A' : Bound(t', \lambda')$, with $t' = t'_P + t'_Q \preceq r_b$ and $\lambda' = (\lambda'_P + \lambda'_Q)\{b\}$. As $\lambda'_P(b) \preceq d_a$ and $\lambda'_Q(b) \preceq d_a \preceq r_b$, we further deduce that $\lambda' \preceq \lambda$ and $t' \preceq \sum_{c \in N} \lambda'(c) \preceq t \preceq \sum_{c \in N} \lambda(c)$.

Which proves the case.

**R-Comm-Close 2** Let us consider the case where $A = P\downarrow_Q, A' = (vb : N = v)P$, $A' = (vb : N = v)P'$ and $P\stackrel{\alpha}{\longrightarrow} P'$. By induction hypothesis, the type of $P$ and the type of $P'$ are the same. Which proves the case.

**R-Fetch** We have $A = (va : N = v)P, A' = (va : N = v)P'$ and $P\stackrel{\alpha}{\longrightarrow} P'$. By induction hypothesis, the type of $P$ and the type of $P'$ are the same. Which proves the case.

**R-Par** We have $A = P\downarrow_Q$ and $A' = P\downarrow_Q$ where $P\stackrel{\alpha}{\longrightarrow} P'$. By examining all possible configurations of $\alpha$, 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 $A = (vb : Name(\_, r_b) = v)P, A' = P', P\stackrel{ab}{\longrightarrow} P'$ and $\Gamma \vdash a : Name(C, \_)$.

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

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

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

**R-Open 2** Let us consider the case where $A = (vb : Name(\_, r_b) = v)P, A' = P', P\stackrel{ab}{\longrightarrow} P'$ and $\Gamma \vdash a : Name(Unknown, \_)$.

This case is essentially identical to R-Open 1, with $g_a = d_a = \bot$. 48
Proposition 3 (Subject Reduction) If \( A \) is a process, \( \Gamma \) an environment such that \( \Gamma \vdash A : \text{Bound}(r, \lambda) \) and if there is some process \( A' \) such that \( A \rightarrow_{pre} A' \), then we also have \( \Gamma \vdash A' : \text{Bound}(r', \lambda') \), 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 \( \Gamma \) an environment such that, \( \Gamma \) isolates both \( A \) and \( B \) and such that \( \Gamma \vdash A : \text{Bound}(r, \lambda) \), if there is some \( C \) such that \( A \parallel B \rightarrow C \) 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. \( C \equiv \langle \nu \vec{a} = \vec{v} \rangle (A' \mid B') \)
2. \( \bar{\Gamma}, a : \overline{N} \vdash A' : \text{Bound}(t', \lambda') \)
3. \( \bar{\Gamma}, a : \overline{N} \) isolates \( B' \)
4. \( \Sigma_{b \in N'} \lambda'(b) \preceq \Sigma_{b \in N} \lambda(b) \)
5. \( t' + \Sigma_{b \in N'} \lambda'(b) \preceq t + \Sigma_{b \in N} \lambda(b) \)
6. \( \forall i \in 1..n, \text{if } N_i = \text{Name}(\_, r_i) \text{ then } \lambda'(a_i) \preceq r_i \)
7. \( t' + \Sigma_{i \in 1..n} r_i \Sigma_{b \in N} \lambda'(b) \preceq t + \Sigma_{b \in N} \lambda(b) \)
8. \( B' \) is a client

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

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

8. $B'$ is a client

which proves the case.

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

• if $A \xrightarrow{ab_{pre}} A'$ and $B \xrightarrow{a?b_{pre}} B'$ – by hypothesis, since $\Gamma$ isolates $B$, we have $\Gamma \vdash a : \text{Name}(\text{Unknown}, \_)$.

  By Open Subject Reduction, we may also check that we have $\Gamma \vdash A' : \text{Bound}(t', \lambda')$, where $t' \preceq t$ and $\lambda' \preceq \lambda$. In addition, by Open Subject Reduction, we have $\Gamma \vdash b : \text{Name}(\text{Unknown}, \_)$.

  As $\Gamma$ isolates $B$, we deduce that $\Gamma$ isolates $B'$

  Therefore, we have

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

  which proves the case.

• if $A \xrightarrow{a@_{pre}} A'$ and $B \xrightarrow{a@_{pre}} B'$ – by hypothesis, since $\Gamma$ isolates $A$ and $B$, we have $\Gamma \vdash a : \text{Name}(\text{Unknown}, \_)$.

  By Open Subject Reduction, we may also check that we have $\Gamma \vdash A' : \text{Bound}(t', \lambda')$, where $t' \preceq t$ and $\lambda' \preceq \lambda$.

  As $\Gamma$ isolates $B$, we deduce that $\Gamma$ isolates $B'$

  Therefore, we have

  1. $C = A'|B'$
  2. $\Gamma \vdash A' : \text{Bound}(t', \lambda')$
  3. $\Gamma$ 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 $\Gamma$, $\forall i \in 1..0$, if $N_i = \text{Name}(\_, r_i)$ then $\lambda'(a_i) \preceq r_i$
  7. as we have added no name to $\Gamma$, $t' \oplus \Sigma_{i \in 1..n} r_i \Sigma_{b \in \mathcal{N}} \lambda' \setminus \overline{\lambda}(b) \leq t \oplus \Sigma_{b \in \mathcal{N}} \lambda(b)$
  8. as $B \xrightarrow{a@_{pre}} B'$, by definition, $B'$ is a client
which proves the case.

- if $A \xrightarrow{\text{a,\text{b}}} B$ and $B \xrightarrow{\text{a,\text{b}}} B'$ - by hypothesis, since $\Gamma$ isolates $A$ and $B$, we have $\Gamma \vdash a : \text{Name(Unknown, r)}$. By Open Subject Reduction, we deduce that $\Gamma, b : \text{Name(Unknown, r)} \vdash A' : \text{Bound}(t', \lambda')$, where $t' \preceq t$ and $\lambda' \preceq \lambda$. As $\Gamma$ isolates $B$, we deduce that $\Gamma$ isolates $B'$. Therefore, we have
  1. $C = A'|B'$
  2. $\Gamma \vdash A' : \text{Bound}(t', \lambda')$
  3. $\Gamma$ isolates $B'$
  4. as $\lambda' \preceq \lambda$, $\Sigma_{b \in N} \lambda'(b) \preceq \Sigma_{b \in N} \lambda(b)$
  5. as $\lambda' \preceq \lambda$ and $t' \preceq t$, $t' \oplus \Sigma_{b \in N} \lambda'(b) \preceq t \oplus \Sigma_{b \in N} \lambda(b)$
  6. as we have added no name to $\Gamma$, $\forall i \in 1..0$, if $N_i = \text{Name(\_\_}_r_i)$ then $\lambda'(a_i) \preceq r_i$
  7. as we have added no name to $\Gamma$, $t' \oplus \Sigma_{i \in 1..n} r_i \Sigma_{b \in N} \lambda' \ominus \bar{a}(b) \preceq t \oplus \Sigma_{b \in N} \lambda(b)$
  8. as $B \xrightarrow{\text{a,\text{b}}} B'$, by definition, $B'$ is a client

which proves the case.

- if $A \xrightarrow{\text{a,\text{b}}} B$ and $B \xrightarrow{\text{a,\text{b}}} B'$ - by hypothesis, since $\Gamma$ isolates $A$ and $B$, we have $\Gamma \vdash a : \text{Name(Unknown, r)}$. By Open Subject Reduction, we deduce that $\Gamma, b : \text{Name(Unknown, r)} \vdash A' : \text{Bound}(t', \lambda')$, where $t' \preceq t$ and $\lambda' \preceq \lambda$. As previously, the case is proved.

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

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

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

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 $\lambda' \preceq \lambda$, $\Sigma_{b \in N} \lambda'(b) \preceq \Sigma_{b \in N} \lambda(b)$
5. as $\lambda' \preceq \lambda$ and $t' \preceq t$, $t' \oplus \Sigma_{b \in N} \lambda'(b) \preceq t \oplus \Sigma_{b \in N} \lambda(b)$
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. $t' \oplus \Sigma_{i \in 1..n} r_i \Sigma_{c \in N} \lambda'(c) = t' \oplus \Sigma_{c \in N} \lambda'(\{b\})(c) \preceq t \oplus \Sigma_{c \in N} \lambda(c)$
8. as $B \xrightarrow{a\tau b}_{pre} B'$, 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 $\Gamma \vdash P : \text{Bound}(\text{reserve}, \bot_N)$ then $\text{res}(P) \preceq \text{reserve}$.

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

□

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

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

Proposition 6 (Well-typed terms are robust) If $\Gamma$ isolates $P$ and $\Gamma \vdash P : \text{Bound}(\text{reserve}, \bot_N)$ then $P$ is robust.

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