Separation Logic

Expressiveness and Copyless Message-Passing

Étienne Lozes

mémoire d'habilitation à diriger des recherches

défendu le 3 juillet 2012
à l'École Normale Supérieure de Cachan
devant le jury composé de

Mmes	Sandrine Blazy	rapporteuses
	Viviana Bono	
	Philippa Gardner	
MM.	Jean-Christophe Filliâtre	examinateurs
	Alain Finkel	
	François Pottier	

À mes petites étoiles et leur super nova.

Foreword

The work presented in this memory covers the most significant topics I have been interested in during my stay at the Laboratoire Spécification and Vérification (LSV) in Cachan. Somehow, these topics all deal with Separation Logic. This apparent coherence hides two lines of works driven by completely opposite points of views. My initial interest for Separation Logic sprang from my phD on spatial logics. I kept addressing on Separation Logic the questions of expressiveness and decidability I had been interested in during my phD. The program of this approach is a kind of deconstruction of Separation Logic that tries to evaluate to which extend Separation Logic connectives and rules can be embedded in previously existing ones. This approach probably hides the enthusiasm I have for all the concepts introduced by Separation Logic, and I am happy that this memory contains a substantial part on applying these concepts in a constructive way.

The work presented in the first two chapters could be ranged in my deconstruction phase. It stemmed from the enthusiasm of Stéphane Demri in learning Separation Logic, and I regret I did not present the first work we did together on mixing temporal logics and Separation Logic [1]. Our collaboration really took off after a question we asked to Rémi Brochenin on the decidability of the list fragment of the assertion language of Separation Logic – a question that, despite its naturality, was never even pointed out as an interesting one, mostly because the magic wand was already abandoned at that time. Rémi Brochenin spent a short but intense time solving this question, and realized that the undecidability proof contained the ingredients for a much more unexpected result, a connection between Separation Logic and Second Order logic. I regret a lot that Remi did not write his phD yet, because I am convinced that the results he obtained could constitute a very valuable phD thesis.

The work presented in the last chapter stemmed from a discussion with Cristiano Calcagno while he visited Cachan at the beginning of Jules Villard's phD thesis. We did not imagine how exciting Cristiano's idea of modeling Sing♯ in Separation Logic would be. Perhaps the best surprise in that work was that it was possible to represent Sing♯ contracts so concisely in SL – we actually first rejected the idea to model them, scared about their complexity, but hopefully changed our mind when we started to understand what they were adding to local reasoning. Formalizing all the intuitions we had took quite a lot of time, and a significant step in that direction was the operational semantics proposed by Jules in his phD thesis. Still, Jules did not have the time to treat the question of determinism, which I hope I have clearly addressed now.

I decided to skip the presentation of the Heap-Hop tool, for which I only contributed in designing and testing some of the examples. In understanding and hacking the original code of Smallfoot, Jules accomplished an impressive work, without even saying a word about it to Cristiano or Josh Berdine, in almost one week. If you don't know Heap-Hop yet, you are probably missing the best part of the work on copyless message-passing, and I recommend you to download it and to play with the examples.

People that know the LSV may wonder why I do not present a memory on automata-based model-checking or security. This memory surveys the results I have obtained on Separation Logic to keep a certain coherence in the presentation, but I wandered around a lot more topics. I do not really regret that when I consider the quantity of models and techniques that I learned at the LSV. I would have loved to make the influence of the LSV more visible in this memory, but this would have revealed too much of the recreative aspect of my research. Out of the work on temporal logics I already mentioned, I worked with Sebastien Bardin and Arnaud Sangnier on an unsuspectedly rich topic initiated by Alain Finkel, namely the analysis of list-manipulating programs through their translation into counter systems [2] – an approach that proved to be extremely successful for applications we absolutely missed at that time. Excited by the work of the security group at LSV, I worked with Jules on a spatial logic for the applied pi-calculus [3], and I discovered with Ralf Treinen and Florent Jacquemard how tree automata techniques could be used to solve some decision problems of this logic [4]. The presence of people like Alain Finkel and Philippe Schnoebelen in the working group of infinite-state model-checking I belonged to is the reason why Jules and I formalized the semantics of Sing♯ contracts in terms of communicating finite state machines and subsequently studied their relation with half-duplex dialogs [5]. The work of Stephane Demri on temporal logics for data words and the work of Luc Segoufin on a similar topic were the sources of my interest for the decidability of Separation Logic over lists with data [6]. And there is probably ten times more ideas that all people at LSV put in my mind and did not concretize yet!

An habilitation defence hopefully is quite rare in a life. And hopefully, a great jury may turn it into something almost pleasant. I am extremly honoured Sandrine Blazy, Viviana Bono and Philippa Gardner accepted to read this memory, came to the defence with so many questions, and helped me so much to improve the first version. I have a special thank to Alain Finkel for accepting both to be part of the jury and to play as the *garant* of this habilitation during all the preparation phase. Finally, I am very grateful to Jean-Christophe Filliâtre and François Pottier for accepting to join the jury and shed other lights to the topic. Each of them brought its own key ingredient to the defence and made it positively unforgettable.

Contents

Table of notations

Symbol	Denotation
$\emptyset$	empty set
$\mathbb{N}$	set of natural numbers
$\mathbb{Z}$	set of integers
i, j, n, m	integers
$a, b, c, \mathsf{cell}, \mathsf{ack}, \ldots$	message identifiers
Σ	set of message identifiers
w	word over alphabet Σ
ϵ	empty word
$x, y, z, e, f, \ldots$	program or first-order variables
X, Y	second-order variables
φ,ψ,...	formula or state property
E	program or logical expression
B	boolean program expression
p	program
v	value
l	cell's location
ε	endpoints' location
σ	program state
s	stack
h	local heap
$\boxed{h}$	global heap
buf	buffer
C	contract
q	control state of a contract
Γ	proof environment

Chapter 1

Introduction

Separation Logic is now a well established program logic for specifying and analysing heap-manipulating programs. Since the foundational papers [7, 8, 9], of O'Hearn, Reynolds and Yang in the early 2000s, it made impressive progresses, essentially in two directions: it demonstrated how the proofs of sequential programs could be fully automated, even for large-scale realistic programs, and it illustrated the strength of local reasoning on challenging small concurrent programs. Good surveys [10, 11] better explain these contributions as we could do. The aim of this introduction is only to collect basic definitions, and to emphasize some specific topics:

1. the completeness of Separation Logic;
2. the reduction of the bi-abduction problem to the entailment problem.

This chapter and the following shows the particular role of the "magic wand" connective in these questions. The magic wand originates from the early works on BI [7], but it is largely absent in the recent works on Separation Logic. We aim to show in this introduction that the magic wand is a powerful and useful feature of the logic, which is probably a victim of carrying such a big power.

1.1 Background on Separation Logic

Memory States

In this manuscript, we adopt the simplified memory model obtained by abstracting away from pointer arithmetic, and considering that all cells are allocated with at most two fields. We assume infinite sets $\mathsf{Loc} = \{l, \dots\}$, $\mathsf{Var} = \{x, y, \dots\}$, and $\mathsf{Val} = \{v, \dots\}$ of respectively locations, variables, and values, that are such that $\mathsf{Val} = \mathsf{Loc} \uplus \text{null}$.

Definition 1.1 (Memory State) *A memory state is a pair $\sigma = (s, h)$ of a stack and a heap such that*

$E :=$		(**Expressions**)
	$x \in \mathsf{Var}$,	(variables)
	$v \in \mathsf{Val}$	(values)
$b :=$		(**Conditions**)
	$E_0 = E_1$, $E_0 \neq E_1$	(equality tests)
$p :=$		(**Programs**)
	$x := \mathrm{new}()$,	(allocation)
	$\mathrm{dispose}(E)$,	(disposal)
	$x := E$,	(affectation)
	$x := E.i$,	(lookup)
	$E.i := E'$	(update)
	skip ,	(neutral)
	$p_0; p_1$,	(sequential composition)
	$p_0 \parallel p_1$,	(parallel composition)
	$\mathrm{while}\ b\ \mathrm{do}\ p$,	(iteration)
	$\mathrm{if}\ b\ \mathrm{then}\ p_0\ \mathrm{else}\ p_1$	(branching)

Figure 1.1: A toy programming language

- $s : \mathsf{Var} {\to} \mathsf{Val}$ *is a total function from variables to values*
- $h : \mathsf{Loc} \rightharpoonup \mathsf{Val} \times \mathsf{Val}$ *is a partial function from locations to pairs of values.*

Two memory states $\sigma_1 = (s_1, h_1)$ and $\sigma_2 = (s_2, h_2)$ are said disjoint, $\sigma_1 \bot \sigma_2$, if $dom(h_1) \cap dom(h_2) = \emptyset$ and $s_1 = s_2 = s$; if so, their composition $\sigma_1 \bullet \sigma_2$ is $(s, h_1 \cup h_2)$. Notations $h_1 \bot h_2$ and $h_1 \bullet h_2$ are defined accordingly. We write $h \leq h'$ if h is a restriction of h', *i.e.* there is h'' such that $h \bullet h'' = h'$.

Programming Language

We will consider a toy programming language modeling heap-manipulating programs, both sequential and concurrent. To keep the presentation simple, we abstract away from common features, like local variables, function calls, or thread creation[1], and consider a rather tiny model (see Figure 1.1). The semantics of this toy programming language is defined by means of a standard small step operational semantics $p, \sigma \rightsquigarrow p', \sigma'$ based on interleaving concurrent threads (see Table 1.2 for the exact definition). A reduction $p, \sigma \rightsquigarrow$ **OwnError** denotes the

[1] These features are known to be more or less easily tractable in SL, and we may freely use them in some examples; their formal treatment is left to the attention to the reader.

situation in which a program runs into an *ownership's violation.* For sequential programs, ownership violations occur only when a non-allocated pointer is either dereferenced, updated, or disposed. For concurrent programs, it also occurs when two threads concurrently access the same variable or the same cell, either by reading or writing it (see Table 1.3 for the exact definition).

Programs without ownership's violation will be sometimes called *selfish.* Note that selfish programs are race-free, but the converse is false: a (race-free) concurrent read to a variable or a cell is not selfish. We focus on this subclass of race-free concurrency with a clear intention: we are convinced that selfish concurrency is worth a special attention, and that selfish programs present several advantages over race-free ones. The thread model introduces a *fiction of sharing* that disappears when looking at lower levels of abstraction. For instance, two threads running on different cores may have each a cached local version of the region of the address space they work with. In this specific case, selfish programs can be expected to be much more efficient than race-free ones, as the coherence of caches is much simpler to guarantee, and the economy of message exchanges across cache managers could be a substantial gain. Another advantage is the simplicity of distributing garbage collection for selfish programs, which we will later discuss in Chapter 3. Despite the interest of selfish concurrency, and its tight connection with separation logic, several questions seem unanswered by the existing literature. We later discuss two such questions, which are the design of a proof theory capturing exactly selfish programs, and the relation between selfishness and determinism for message-passing concurrency.

Assertion Language

We adopt an assertion language of Separation Logic close to the one considered in the early days [7, 9]; in particular, it includes the "magic wand" connective.

Definition 1.2 (Formula) *A formula φ of SL is defined by the grammar:*

$$\boxed{\begin{array}{ll}\varphi,\psi := & x = y \ , \ \mathsf{emp} \ , E_0 \mapsto (E_1, E_2) \ , \ ls(E_0, E_1) \ , \ \varphi \wedge \psi \ , \neg\varphi \ , \\ & \varphi * \psi \ , \ \varphi \mathrel{-\!\!*} \psi \ , \ \exists x.\varphi \end{array}}$$

SL

We adopt the usual notations $\neq$, $\vee$ and $\forall$ for the duals of $=$, $\wedge$ and $\exists$. We use the wildcard notation to abbreviate existentially quantified variables when they occur exactly once in a formula, *i.e.* we write $E \mapsto (-,-)$ for $\exists x_1, x_2.E \mapsto (x_1, x_2)$. We moreover abbreviate $E \mapsto (E', -)$ as $E \mapsto E'$, and $(x \mapsto y, z) * \top$ as $x \hookrightarrow (y, z)$. We also abreviate $ls(x, nil)$ as $ls(x)$. Finally, we write $\varphi \mathbin{-\!\!\circledast} \psi$ for the so-called "septraction" connective, defined as

$$\varphi \mathbin{-\!\!\circledast} \psi \quad \triangleq \quad \neg(\varphi \mathrel{-\!\!*} \neg\psi).$$

$$\frac{h = \{l \mapsto (v_1, v_2)\} \qquad h_f \perp h}{x := \mathrm{new}(), (s, h_f) \quad \rightsquigarrow \quad \mathrm{skip}, (s\{x := l\}, h_f \bullet h)}$$

$$\frac{h = \{\llbracket E \rrbracket s \mapsto (v_1, v_2)\}}{\mathrm{dispose}(E), (s, h_f \bullet h) \quad \rightsquigarrow \quad \mathrm{skip}, (s, h_f)}$$

$$x := E, (s, h_f) \quad \rightsquigarrow \quad \mathrm{skip}, (s\{x := \llbracket E \rrbracket s\}, h_f)$$

$$\frac{h(\llbracket E \rrbracket s) = (v_1, v_2)}{x := E.i, (s, h) \quad \rightsquigarrow \quad \mathrm{skip}, \big(s\{x := v_i\}, h)}$$

$$\frac{h = \{\llbracket E \rrbracket \mapsto (v_1, v_2)\} \qquad h' = \{\llbracket E \rrbracket \mapsto (\llbracket E' \rrbracket, v_2)}{E.1 := E', (s, h_f \bullet h) \quad \rightsquigarrow \quad \mathrm{skip}, (s, h_f \bullet h')}$$

$$\frac{h = \{\llbracket E \rrbracket \mapsto (v_1, v_2)\} \qquad h' = \{\llbracket E \rrbracket \mapsto ([v_1, |E' \rrbracket)}{E.2 := E', (s, h_f \bullet h) \quad \rightsquigarrow \quad \mathrm{skip}, (s, h_f \bullet h')}$$

$$\frac{\llbracket b \rrbracket s = \mathsf{TRUE}}{\mathrm{if}\ \ b\ \mathrm{then}\ p_0\ \ \mathrm{else}\ \ p_1\ ,\ \sigma \quad \rightsquigarrow \quad p_0\ ,\ \sigma}$$

$$\frac{\llbracket b \rrbracket s = \mathsf{FALSE}}{\mathrm{if}\ \ b\ \mathrm{then}\ p_0\ \ \mathrm{else}\ \ p_1\ ,\ \sigma \quad \rightsquigarrow \quad p_1\ ,\ \sigma}$$

$$\frac{p_0 \sim p_0' \qquad p_0', \sigma_0 \rightsquigarrow p_1', \sigma_1' \qquad p_1' \sim p_1}{p_0, \sigma_0 \quad \rightsquigarrow \quad p_1, \sigma_1}$$

$$\frac{p_0, \sigma_0 \rightsquigarrow p_1, \sigma_1}{p_0; p, \sigma_0 \quad \rightsquigarrow \quad p_1; p, \sigma_1} \qquad\qquad \frac{p_0, \sigma_0 \rightsquigarrow p_1, \sigma_1}{p_0 \parallel p, \sigma_0 \quad \rightsquigarrow \quad p_1 \parallel p, \sigma_1}$$

$s\{x := v\}(y)$ is v if $x = y$, otherwise $s(y)$. $\llbracket x \rrbracket s \triangleq s(x)$, and $\llbracket v \rrbracket s \triangleq v$.
$\llbracket E_0 = E_1 \rrbracket s \triangleq \mathsf{TRUE}$ if $\llbracket E_0 \rrbracket s = \llbracket E_1 \rrbracket s$, FALSE otherwise, and $\llbracket E_0 \neq E_1 \rrbracket s \triangleq \neg \llbracket E_0 = E_1 \rrbracket s$.
$\sim$ denotes the smallest program equivalence such that ; is associative with neutral `skip`, $\parallel$ is commutative and associate with neutral `skip`, and

$$\mathrm{while}\ \ b\ \ \mathrm{do}\ p \quad \sim \quad \mathrm{if}\ \ b\ \ \mathrm{then}\ p; \mathrm{while}\ \ b\ \ \mathrm{do}\ p\ \ \mathrm{else}\ \ \mathrm{skip}.$$

■ Figure 1.2: The operational semantics (1/2)

$$\frac{[\![E]\!]s \notin dom(h)}{\mathrm{dispose}(E),(s,h) \rightsquigarrow \textbf{OwnError}}$$

$$\frac{[\![E]\!]s \notin dom(h)}{x := E.i,(s,h) \rightsquigarrow \textbf{OwnError}} \qquad \frac{[\![E]\!]s \notin dom(h)}{E.i := E',(s,h) \rightsquigarrow \textbf{OwnError}}$$

$$\frac{p_i \sim \alpha_i ; p_i' \qquad \mathsf{v}(\alpha_0) \cap \mathsf{v}(\alpha_1) \neq \emptyset}{p_0 \parallel p_1, \sigma \rightsquigarrow \textbf{OwnError}}$$

$$\frac{p_i \sim \alpha_i ; p_i' \qquad \mathsf{address}_s(\alpha_0) \cap \mathsf{address}_s(\alpha_1) \neq \emptyset}{p_0 \parallel p_1, (s,h) \rightsquigarrow \textbf{OwnError}}$$

$$\frac{p \sim p' \qquad p', \sigma \rightsquigarrow \textbf{OwnError}}{p, \sigma \rightsquigarrow \textbf{OwnError}} \qquad \frac{p_0, \sigma_0 \rightsquigarrow \textbf{OwnError}}{p_0 ; p, \sigma_0 \rightsquigarrow \textbf{OwnError}}$$

$$\frac{p_0, \sigma_0 \rightsquigarrow \textbf{OwnError}}{p_0 \parallel p, \sigma_0 \rightsquigarrow \textbf{OwnError}}$$

We range over *atomic* program (allocation, disposal, affectation, lookup, update) with α, and write $\mathsf{v}(\alpha)$ to denote the set of variables occuring in α. We write $\mathsf{address}_s(\alpha)$ to denote $s(\mathsf{v}(\alpha))$ augmented of the locations occuring as plain text in the expressions of α.

Figure 1.3: The operational semantics (2/2)

The semantics of the assertion language is standard (see Table 1.4). Informally, emp denotes a memory state with no cell allocated, $x \mapsto (x_1, x_2)$ denotes a memory state with exactly one cell at location x holding values x_1 and x_2. The precise predicate $ls(x,y)$ denotes an acyclic singly-linked list starting at x and reaching y by following the first field, thus recursively defined by $(x = y \wedge \mathsf{emp}) \vee x \neq y \wedge \exists x'.x \mapsto x' * ls(x', y)$. The separating conjunction $\varphi * \psi$ asserts that φ and ψ hold on disjoint regions of the heap, $\varphi \mathrel{-\!\!\circledast} \psi$ asserts that the heap is a residue of a model of ψ in which a model of φ has been framed out, and $\varphi \mathrel{-\!\!*} \psi$ holds for a state σ_0 such $\sigma \mapsto \sigma \bullet \sigma_0$ transforms any model of φ in a model of ψ.

We say that a formula φ is *precise* if for all (s,h), there is at most one $h' \leq h$ such that $(s,h') \vDash \varphi$. As usual, we say that φ entails ψ, $\varphi \vDash \psi$, if all models of φ are also models of ψ. A formula φ is consistent, or satisfiable, if $\varphi \nvDash \bot$, and φ is valid, $\vDash \varphi$, if $\top \vDash \varphi$.

$$
\begin{array}{rlll}
(s,h) \vDash & E_0 = E_1 & \text{if } [\![E_0]\!]s = [\![E_1]\!]s \\
(s,h) \vDash & \mathsf{emp} & \text{if } \mathit{dom}(h) = \emptyset \\
(s,h) \vDash & E_0 \mapsto (E_1, E_2) & \text{if } \mathit{dom}(h) = \{[\![E_0]\!]s)\} \text{ and } h([\![E_1]\!]s)) = ([\![E_1]\!]s, [\![E_2]\!]s) \\
\sigma \vDash & \varphi \wedge \psi & \text{if } \sigma \vDash \varphi \text{ and } \sigma \vDash \psi \\
\sigma \vDash & \neg\varphi & \text{if } \sigma \nvDash \varphi \\
\sigma \vDash & \varphi_1 * \varphi_2 & \text{if there are } \sigma_1, \sigma_2 \text{ s.t. } \sigma = \sigma_1 \bullet \sigma_2, \sigma_1 \vDash \varphi_1 \;\&\; \sigma_2 \vDash \varphi_2 \\
\sigma \vDash & \varphi_1 \mathrel{-\!\!*} \varphi_2 & \text{if for all } \sigma' \text{ s.t. } \sigma' \bot \sigma, \text{ if } \sigma' \vDash \varphi_1, \text{ then } \sigma \bullet \sigma' \vDash \varphi_2 \\
\sigma \vDash & \exists x.\varphi & \text{if there is } v \text{ s.t. } \sigma \vDash \varphi[x \leftarrow v] \\
\sigma \vDash & ls(E_0, E_1) & \text{if } \sigma \vDash E_0 = E_1 \wedge \mathsf{emp} \text{ or} \\
 & & \text{if } \sigma \vDash E_0 \neq E_1 \wedge \exists x. E \mapsto x * ls(x, E_1)
\end{array}
$$

Figure 1.4: The forcing semantics of Separation Logic

Proof System

As in any program logic, the essential part of Separation logic is its inference rules for inferring Hoare triples $\{\varphi\}\ \mathrm{p}\ \{\psi\}$, where φ is the pre-condition assertion, ψ is the post-condition assertion, and p is a program. In this manuscript, we adopt a standard proof system including the frame rule, the rule for elimination of auxiliary variables, and the parallel rule (see the definition on Figure 1.5). We ignore the rule of conjunction, and the rule of infinite disjunction:

$$
\textsc{Conjunction}\quad \frac{\{\varphi_1\}\ \mathrm{p}\ \{\psi_1\} \qquad \{\varphi_2\}\ \mathrm{p}\ \{\psi_2\}}{\{\varphi_1 \wedge \varphi_2\}\ \mathrm{p}\ \{\psi_1 \wedge \psi_2\}}
\qquad\qquad
\infty\text{-}\textsc{Disjunction}\quad \frac{\{\varphi_i\}\ \mathrm{p}\ \{\psi_i\} \qquad \text{for some } i \in I}{\{\bigvee_{i \in I} \varphi_i\}\ \mathrm{p}\ \{\bigvee_{i \in I} \psi_i\}}
$$

We omit these rules for two reasons: first, we will show that they are not necessary for the completeness of Separation Logic (unlike in the proof given by abstract Separation Logic [12]), and second, taking them into account could be subject to discussions.[2]

We write $\vdash \{\varphi\}\ \mathrm{p}\ \{\psi\}$ when the Hoare triple has a proof using our proof system.

Definition 1.3 (Valid Hoare triple) *A Hoare triple $\{\varphi\}\ \mathrm{p}\ \{\psi\}$ is said valid, denoted $\vDash \{\varphi\}\ \mathrm{p}\ \{\psi\}$, if for all $\sigma \vDash \varphi$,*

1. $p, \sigma \not\rightsquigarrow^* \mathbf{OwnError}$

2. *if $p, \sigma \rightsquigarrow^* \mathrm{skip}, \sigma'$, then $\sigma' \vDash \psi$*

[2] The conjunction rule has been pointed as one possible responsible of Reynolds' paradox [13], and infinite disjunctions are not part of the syntax of the logic.

We write $fv(p)$ and $fv(\psi)$ to denote the set of free variables, *i.e.* occuring in p or φ not under a quantifier, and $mv(p)$ the set of variables modified by p, *i.e.* appearing on a left side of an affectation `x:=`E.

$$\{\mathsf{emp}\}\ \text{x:=new()}\ \{x \mapsto -\} \qquad \{E \mapsto -\}\ \text{dispose(E)}\ \{\mathsf{emp}\}$$

$$\{\mathsf{emp}\}\ \text{x:=E}\ \{x = E \land \mathsf{emp}\}$$

$$\{E \mapsto (E_1, E_2)\}\ \text{x:=E.i}\ \{x = E_i \land E \mapsto (E_1, E_2)\}$$

$$\{E \mapsto (-, E_2)\}\ \text{E.1:=E'}\ \{E \mapsto (E', E_2)\}$$

$$\textsc{If}\ \dfrac{\{\varphi \land b\}\ \text{p}\ \{\psi\} \qquad \{\varphi \land \neg b\}\ \text{p'}\ \{\psi\}}{\{\varphi\}\ \text{if b then p else p'}\ \{\psi\}} \qquad \textsc{While}\ \dfrac{\{\varphi \land b\}\ \text{p}\ \{\varphi\}}{\{\varphi\}\ \text{while b do p}\ \{\varphi \land \neg b\}}$$

$$\textsc{Sequential}\ \dfrac{\{\varphi\}\ \text{p}\ \{\chi\} \qquad \{\chi\}\ \text{p'}\ \{\psi\}}{\{\varphi\}\ \text{p;p'}\ \{\psi\}}$$

$$\textsc{Parallel}\ \dfrac{\{\varphi\}\ \text{p}\ \{\psi\} \qquad \{\varphi'\}\ \text{p'}\ \{\psi'\}}{\{\varphi * \varphi'\}\ \text{p||p'}\ \{\psi * \psi'\}} \quad \begin{array}{lcll} mv(p) & \cap & fv(\varphi', p', \psi') & = \emptyset \\ mv(p') & \cap & fv(\varphi, p, \psi) & = \emptyset \end{array}$$

$$\textsc{Frame}\ \dfrac{\{\varphi\}\ \text{p}\ \{\psi\}}{\{\varphi * \varphi_F\}\ \text{p}\ \{\psi * \varphi_F\}} \quad mv(p) \cap fv(\varphi_F) = \emptyset$$

$$\textsc{Consequence}\ \dfrac{\varphi \vDash \varphi' \qquad \{\varphi'\}\ \text{p}\ \{\psi'\} \qquad \psi' \vDash \psi}{\{\varphi\}\ \text{p}\ \{\psi\}} \qquad \text{AVE}\ \dfrac{\{\varphi\}\ \text{p}\ \{\psi\} \qquad x \notin fv(p)}{\{\exists x.\varphi\}\ \text{p}\ \{\exists x.\psi\}}$$

Figure 1.5: The proof system of Separation Logic

1.2 A Discussion on Completeness

A long standing problem that was first positively solved by Brookes [14] is the soundness of Separation Logic for concurrent programs[3]. Since our toy programming language is a restriction of the one considered by Brookes – which includes conditional critical section – we immediately get the following result:

Theorem 1.1 (Soundness) *For every φ, p, ψ,*

$$\textit{if} \quad \vdash \{\varphi\}\ \mathrm{p}\ \{\psi\} \qquad\qquad \textit{then} \qquad\qquad \vDash \{\varphi\}\ \mathrm{p}\ \{\psi\}.$$

The companion property of the above is the completeness of the proof system, $i.e$ wether $\vDash \{\varphi\}\ \mathrm{p}\ \{\psi\}$ implies $\vdash \{\varphi\}\ \mathrm{p}\ \{\psi\}$. Completeness has been established for sequential programs – we will discuss this again in Section 2.3. We discuss here the completeness of Separation Logic in the concurrent case. The first thing to notice is that all completeness results for program logics dealing with concurrency usually rely on ghost code modeling the interferences from the environment. Admitting the introduction of ghost code might be undesirable, especially if one is interested in the automation of the proofs – it seems that very little is known about how ghost code can be guessed by a programs' prover. In the case of selfish concurrency, however, it could be expected that interferences from the environment may be unnecessary to model, and that the parallel rule would do all the work. This is however not the case, because allocation induces a form of synchronisation between threads: two threads cannot allocate the same cell in parallel. This specialised synchronisation can be exploited to encode spin-locks[4]. The following example program illustrates this phenomenon: the ownership of a cell `z` is transferred by the deallocation of a cell `x`:

```
globals x,x',y,z;
main(){
  x:=new();
  x':=x;
  producer(x)||consumer(x')
```

```
producer(x){
  z:=new();
  // "send z"
  dispose(x);
}
```

```
consumer(x'){
  y:=new();
  while (y!=x') do { // spin-lock
    dispose(y);
    y:=new();
  }
  // "receive z"
  dispose(z);
}
```

[3] See also the work on abstract separation logic [12] and a direct proof of soundness[15] for discussions around Reynold's paradox, the conjunction rule, and precise predicates.

[4] We thank Matthew Parkinson for pointing us this problem, which was apparently discovered by Yang some years ago.

To get completeness, it may thus be necessary to acknowledge the power of synchronisation of allocation. For instance, in order to prove the above program, one may attach a form of ownership transfer to allocation/deallocation primitives in the same vein as other synchronisation primitives (like send and receive presented in Chapter 3), and consider global variables as resources. One may also refute this example if one is interest in a program logic with total correctness, because it seems difficult to exploit the synchronisation power of allocation without introducing a potential divergence. We will show in Section 2.4 that even for synchronisation-free, terminating programs, the concurrent separation logic we consider misses extra rules to be complete.

1.3 Proof Checking and Frame Inference

We now consider the following problem

Proof Checking Problem
Input: A proof tree of a Hoare triple $\{\varphi\}$ p $\{\psi\}$.
Question: Does $\vdash \{\varphi\}$ p $\{\psi\}$?

For this problem, a "proof tree" is a tree with edges labeled either with Hoare triples $\{\varphi\}$ p $\{\psi\}$ or with entailments $\varphi \models \psi$, and vertices labeled with the name of one of the proof rules of Figure 1.5. In the case of completeness, the question of this problem is equivalent to "does $\models \{\varphi\}$ p $\{\psi\}$?" This problem is thus undecidable, otherwise the set of programs that are safe and do not terminate on initial empty heaps, *i.e.* the set of programs p such that $\models \{\mathsf{emp}\}$ p $\{\bot\}$, would be recursively enumerable. The undecidability of the proof checking problem is located in the check of the consequence rule, since checking other rules' application goes through rather simple syntactic checks. As a consequence, the proof checking problem can be decided only if one sacrifices completeness. One may consider restricting either the class of programs under consideration, or the proof system, or the logic. But since $\models \{\varphi\}$ skip $\{\psi\}$ is undecidable, as we will see in Section 2.3, it is unavoidable to restrict the logic.

The quest for fragments of SL that are expressive enough for proving common programs, but that also keep a good complexity for the entailment problem, really started with the preliminary theoretical work [16] giving foundations to Smallfoot [17, 18]. Before that work, a first investigation [19] was conducted on fragments of SL that did not include the list predicate[5]. Symbolic heaps are

[5] The list predicate is quite common in proofs of academic examples, but surprisingly rare when verifying large-scale codes. Two facts giving credits to this claim: the analysis of an IMAP server by the Abductor tool required list segments for only 3% of the procedures it could analyse [20], and it was not possible, during the ANR AVERILES project, to find a significant code really in use at EDF that needed to reason on lists. Maybe the efforts that were turned on lists and other recursive predicates will now focus more on efficiency for points-to analysis in SL, as suggested by the conclusion of a recent work [20], and the results of the early days [19] will be reconsidered in this perspective.

defined as pairs $\Pi \wedge \Sigma$ of formulas from the following grammar:

$$\boxed{\begin{array}{lclr} \Pi & := & \Pi \wedge \Pi \ , \ x = y \ , \ x \neq y & \text{(pure formulas)} \\ \Sigma & := & \Sigma * \Sigma \ , \ x \mapsto y \ , \ ls(x,y) \ , \ \mathsf{emp} \ , \ \top & \text{(spatial formulas)} \end{array}}$$

Symbolic Heaps

As it has been recently proved, the entailment problem is polynomial-time for symbolic heaps [21], whereas other studied decidable fragments present always a higher complexity.

The proof checking problem is however a bit too idealised. In practice, proof checkers for SL do not take as input a proof tree, but rather an annotated program, from which they try to reconstruct the proof tree. For this reason, the design of a good assertion language requires more than showing that entailment is decidable. We now review the problems that proof checkers have to face in practice.

All available proof checkers for SL rely on a symbolic execution that generates "proof obligations", *i.e.* instances of the entailment problem. A forward symbolic execution is a function computing, for a loop-free program p and a precondition φ, the smallest post-condition ψ such that $\vDash \{\varphi\}$ p $\{\psi\}$ (backward symbolic execution would work symmetrically). It is then desirable that the fragment of SL manipulated by the proof checker is not only decidable, but also closed under symbolic execution.

Proof checkers often also have to solve the following problem.

<u>Frame Inference Problem</u>
Input Formulas φ, ψ.
Question Is there X such that $\varphi \vDash \psi * X$?

This problem might be worked around in some cases, but it becomes unavoidable if we want to handle function calls. Note that, if the proof checker runs a backward symbolic execution, it has to solve another problem (the *abduction* problem that we will recall soon). A theoretically "complete" fragment of the assertion language of SL, with respect to proof checking, would thus enjoy the following properties:

- decidability of the entailment problem;
- stability by post-conditions (or pre-conditions);
- decidability of the frame inference problem (or the abduction problem).

Symbolic heaps are far from "complete" in that sense: they are for instance not stable by post-conditions. Consider the Hoare triple $\{x \mapsto y * ls(y)\}$ x:=y $\{.\}$. Then the expected postcondition $x = y \wedge \exists x'.x' \mapsto y * ls(y)$ cannot be expressed as a symbolic heap. More generally, symbolic heaps are not closed under existential quantifications, and thus are not closed under either backward or forward symbolic execution. This issue is solved in Smallfoot by coding existentially quantified variables as ghost variables – note that $\varphi \models \psi$ is equivalent to $\exists x.\varphi \models \psi$ when $x \notin fv(\psi)$. But still, symbolic heaps do not permit to solve the frame inference problem in full generality. The fact that symbolic heaps are syntactically too restricted to be a "complete" abstract domain is pointed out by the authors of Smallfoot as follows:

> *"This incompleteness could be dealt with if we instead used the backwards-running weakest preconditions of Separation Logic. Unfortunately, there is no existing automatic theorem prover which can deal with the form of these assertions (which use quantification and the separating implication $\mathrel{-\!\!*}$). If there were such a prover, we would be eager consumers of it."*

The following result gives another more formal explanation of the interest of the magic wand – or more precisely, the septraction – for the purpose of symbolic execution:

Proposition 1.1 *Let φ, ψ be two formulas, and let Sol be the set of solutions of the frame inference problem $\varphi \models \psi * X$. Then, ordering Sol with $\models$, we have :*

1. *if $Sol \neq \emptyset$, then it is closed under infinite $\vee$ and its greatest element is $\top$;*

2. *if moreover ψ is precise, then it is closed under infinite $\wedge$ and its smallest element is $\psi \mathbin{-\!\!\circledast} \varphi$.*

In other words, this result states the following: if we had an abstract domain that included septraction, we could run the forward symbolic execution: starting from φ, a call to a function $f()$ specified as $\{\varphi_1\}$ $\mathtt{f}()$ $\{\varphi_2\}$ would go as follows:

1. check entailment $\varphi \models \varphi_1 * \top$;

2. if the entailment holds, continue with $\varphi_2 * (\varphi_1 \mathbin{-\!\!\circledast} \varphi)$, otherwise report an error.

1.4 Proof Inference and Abduction

We now briefly review the results obtained on proof inference, which could be stated as follows:

Proof Inference Problem
Input A program p.
Question Is there φ, ψ such that φ is consistent and $\{\varphi\}$ $\mathtt{p}$ $\{\psi\}$ is derivable?

In the case of completeness, this problem is equivalent to the safety of p under a non-trivial precondition. This problem is thus undecidable in full generality, but there are several ways to solve it in many practical cases. We mention here two possible approaches. The first one resorts to the shape analysis approach, which roughly consists in a symbolic execution for programs with loops, performing abstractions from time to time to ensure that only finitely many formulas are generated [22]. The ideas behind the abstraction are not explained very conveniently in terms of formulas, and are better explained in terms of graphs (see for instance Sangnier's phD thesis [23]). This abstraction can moreover be kept precise by translating a heap-manipulating program into a bisimilar counter program [24, 2, 25], where counters represent the length of the list segments. Several enhancement of this quantitative shape analysis have been proposed, by Bouajjani & al integrating data in the lists [26], by Cook & al for a lot more fun problems, like proving termination [27] of sequential programs, or progress of non-blocking algorithms [28], and more recently by Magill & al for complex data structures not limited to lists [29].

Another approach is the one based on bi-abduction [30, 31]. This approach is also based on a symbolic execution and abstractions like the ones of shape graphs, but the symbolic execution does not report an error when the precondition φ of p does not permit to execute it safely, and rather try to guess a φ' such that p can be run over $\varphi * \varphi'$. In addition to the frame inference problem we already mentioned, it thus needs to solve the following problem:

Abduction Problem
Input Two formulas φ, ψ.
Question Is there X such that $\varphi * X \models \psi$ and $\varphi * X$ is consistent?

The abduction problem has been advocated to be useful for inferring proofs of not only sequential programs, but also concurrent ones [32]. In a recent work, Gorogiannis studied the complexity of this problem for symbolic heaps [20].

Again, it is possible to clarify the connection of the abduction problem with the magic wand:

Proposition 1.2 *Let φ, ψ be two formulas, and let Sol be the set of solutions of the abduction problem $\varphi * X \models \psi$. Then, ordering Sol with $\models$, we have :*

1. *Sol is closed under infinite disjunctions; it is moreover closed under infinite conjunctions $\wedge_{i \in I} X_i$ when the conjunct is consistent.*

2. *Sol $\neq \emptyset$ if $\varphi * (\varphi \mathrel{-\!\!*} \psi)$ is consistent, and when this is the case, $\varphi \mathrel{-\!\!*} \psi$ is the largest element of Sol.*

Chapter 2

Expressiveness of Separation Logic

In the previous chapter, we saw how the separating conjunction and the magic wand were both needed in the foundations of program verification using Separation Logic: the separating conjunction enables local reasoning through the frame rule and the parallel rule, whereas the magic wand can be used internally to automatically check an annotated programs, and even to infer the annotations.

This chapter presents several results on the expressive power of these two logical connectives. The point of view is not application oriented, and the aim of the presentation is simply to evaluate the expressiveness and complexity of the SL assertion language by relating it to other logics, and not answering the much more delicate question of the design of efficient abstract domains based on Separation Logic.

Problems of interest often include either the entailment problem (given φ, ψ, does $\varphi \models \psi$?), or the validity problem (given φ, does $\models \varphi$?), or the satisfiability problem (given φ, does it have a model?), or the model-checking problem (given (s, h) and φ, does $(s, h) \models \psi$?). These problems are all equivalent: the three first problems are equivalent if formulas are closed under all boolean combinators, which will be the case all the time. The last problem is of a different nature in general, but here it is always a subproblem of the three others, as for every s, h, it is possible to define a formula $\varphi_{(s,h)}$ that precisely captures all models logically equivalent to (s, h). Moreover, if the magic wand is considered, the three first problems may be reduced to a model-checking problem: for instance, if φ is a closed formula, φ is satisfiable if and only if $(s, h) \models \varphi \mathrel{-\!\!\circledast} \top$, whatever (s, h) is.

2.1 Adjunct's Elimination

The first fragment of SL we consider is the one without first-order quantification and list predicates:

$$\boxed{\varphi \ := \ \ \varphi \wedge \varphi \ , \ \neg\varphi \ , \ \varphi * \varphi \ , \ \varphi -\!\!* \ \varphi \ , \ x \mapsto y, z \ , \ x = y \ , \ \mathsf{emp}}$$

SL_{prop}

In the early days of Separation Logic [19], the entailment for this fragment was shown to be decidable but PSPACE-hard. The decidability proof is based on a small model property: a formula φ has a model if and only if it has a model with $n \leq size(\varphi)$ cells, where $size(\varphi)$ is linear in the number of variables plus the number of occurrences of $\mapsto$ and emp [19]. The expressivity of this fragment was later characterized by the following result:

Theorem 2.1 (Adjunct's Elimination for SL) *For every formula φ of the SL_{prop} fragment , there is an equivalent formula in the fragment:*

$$\boxed{\varphi \ := \ \varphi \wedge \varphi \ , \ \neg\varphi \ , (x \hookrightarrow y, z) \ , \ (x \hookrightarrow -) \ , \ heap\_size \geq i \ , \ x = y}$$

A Propositional Logic

where $heap\_size \geq i$ is defined by $\underbrace{\neg\mathsf{emp} * \cdots * \neg\mathsf{emp}}_{i \ times}$.

This result shows that the magic wand can be eliminated, a phenomenon that generalizes to other spatial logics [33, 34, 35, 36] and other adjunct connectives ($-\!\!*$ is here the adjunct of $*$). Moreover, since the use of $*$ is very restricted in the grammar above, SL_{prop} can be translated into a propositional logic. This could have been interesting for deciding SL entailment by reduction to sat/smt solvers, and indeed this idea has been pushed a bit further, improving the translation by coding it into first-order logic [37].

On the other hand, SL_{prop} is not a very rich fragment of SL, and a natural question was whether it could be extended to other fragments of SL. Despite a proposal for a form of adjunct's elimination in a procedure for stabilization under interferences in RGSep [38], procedures for the elimination of $-\!\!*$ in SL are not well-studied. One reason is that adjunct's elimination is not always possible in richer fragments of SL. The first negative result was obtained by Yang (and communicated privately only). Yang showed that the full SL assertion language of Definition 1.2 does not have the adjunct's elimination. Let us show now a counter-example adapted from Yang's original one. We focus on the following property – we will later reuse this property in many examples:

$$\varphi_{=}(x, x', y, y') \ \triangleq \qquad \big(ls(x, x') * ls(y, y')\big) \ \wedge \ \text{“}\mathsf{dist}(x, x') = \mathsf{dist}(y, y')\text{”}$$

From the embedding of $\mathsf{FO}_{LS}(*)$, the fragment of SL without magic wand, into monadic second-order logic (see next section), we know that $\varphi_{=}(x, x', y, y')$ is

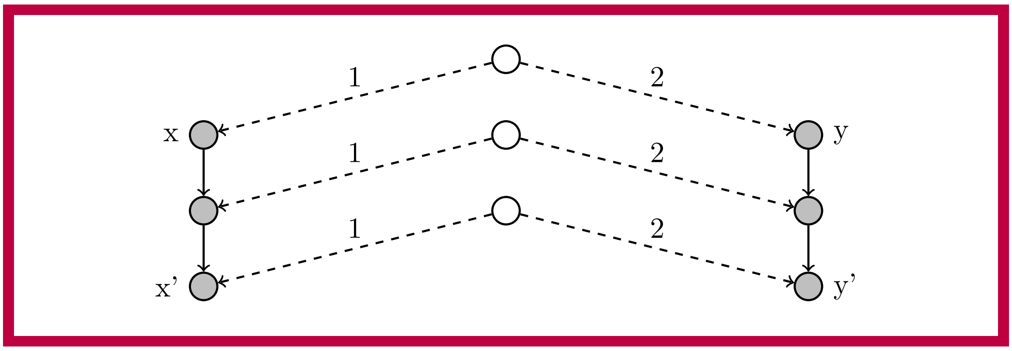


Figure 2.1: Principle of the encoding of $\varphi_=(x,x',y,y')$ with two selectors.

not expressible in $\mathsf{FO}_{LS}(*)$. It is thus sufficient to show that $\varphi_=(x,x',y,y')$ is expressible in SL to deduce that the adjunct's elimination does not hold in SL.

Consider the following formulas:

$$\begin{aligned}
\mathit{marker}(z) &\triangleq (z \mapsto -,-) * \top \ \wedge\ z \neq x \ \wedge\ z \neq y \ \wedge\ \neg\,(- \mapsto z,-) * \top\\
\mathit{marked}1(z) &\triangleq \exists t.\mathit{marker}(t) \wedge (t \mapsto z,-)\\
\mathit{marked}2(z) &\triangleq \exists t.\mathit{marker}(t) \wedge (t \mapsto -,z)\\
\mathit{marked}(z) &\triangleq \mathit{marked}1(z) \vee \mathit{marked}2(z)\\
\mathit{marked\_list}1 &\triangleq \forall z.\ \ ls(x,z) * \top \Leftrightarrow \mathit{marked}1(z)\\
\mathit{marked\_list}2 &\triangleq \forall z.\ \ ls(y,z) * \top \Leftrightarrow \mathit{marked}2(z)
\end{aligned}$$

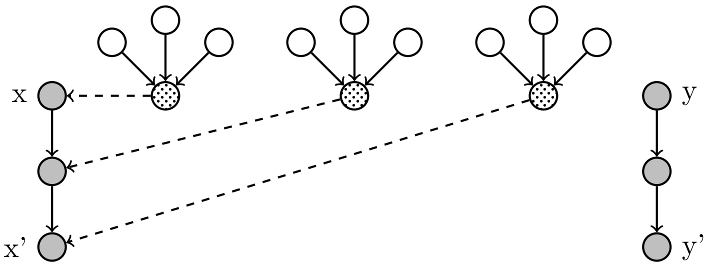
Then $\varphi_=(x,x',y,y')$ is equivalent to the formula $\big(ls(x,x') * ls(y,y')\big) \wedge \varphi'$, with φ' the formula

$$\big(\neg\exists z.\mathit{marked}(z) * \mathit{marked}(z) * \top\big) \mathrel{-\!\circledast} \big(\mathit{marked\_list}1 \ \wedge\ \mathit{marked\_list}2\big).$$

To see why, observe that this formula encodes the scenario depicted on Figure 2.1: cells called "markers" are allocated by the magic wand. Each marker cell points to a cell of the x list through its first selector, and points to a cell of the second list through its second selector; these are precisely the marked cells, and no cell can be marked twice. Such a set of marker cells can be allocated if and only the two lists $ls(x,x')$ and $ls(y,y')$, which shows the claim.

This counter-example has a limitation: it strongly relies on cells having two successors, and the question of adjunct's elimination for cells with one selector is worth to be raised, as many example programs only manipulate lists. Perhaps surprisingly, $\varphi_=(x,x',y,y')$ can be expressed in the Separation Logic for heaps with pure lists.

Theorem 2.2 (Brochenin's Counter-Example) *Consider* SL_{LS} *the sepa-*



Marking locations (here dotted) are created out of to the two lists by one application of $\mathrel{-\!\!*}$. If there are as many marking locations as the length of a list plus one, this list is markable, *i.e.* marking locations can be allocated to form a marking of the list (dashed arrows). The two lists x and y have the same length if the same set of marking locations make them markable.

Figure 2.2: Principle of Brochenin's counter-example.

ration logic over memory states with one selector, i.e. *the fragment*[1]

$$\boxed{\varphi, \psi \quad := x = y \ , \ \mathsf{emp} \ , x \mapsto y \ , \ ls(x,y) \ , \ \varphi \wedge \psi \ , \neg\varphi \ , \ \varphi * \psi \ , \ \varphi \mathrel{-\!\!*} \psi \ , \ \exists x.\varphi.}$$

SL_{LS}

Then SL_{LS} *can express* $\varphi_{=}(x, x', y, y')$.

Proof sketch. Consider the formulas:

$$\begin{aligned}
marking(x) &\triangleq \quad (- \mapsto x) * (- \mapsto x) * (- \mapsto x) * \top \\
marked(x) &\triangleq \quad \exists y.marking(y) * (y \mapsto x) * \top \\
marked\_list(x) &\triangleq \quad \forall y. \ \ \big(ls(x,y) * \top\big) \ \Leftrightarrow \ marked(y) \\
&\quad\ \wedge \neg\exists x, y, z.marking(x) * marking(y) * x \mapsto z * y \mapsto z \\
&\quad\ \wedge \forall y.marking(y) \Rightarrow (y \hookrightarrow -) \\
markable(x) &\triangleq \quad \big(\neg\exists x.(- \mapsto x) * (- \mapsto x) * \top\big) \mathbin{-\!\!\circledast} marked\_list(x) \\
markers &\triangleq \quad \forall x.(- \mapsto x) * \top \ \Rightarrow \ marker(x)
\end{aligned}$$

Then the claim is that the formula

$$\big(ls(x,x') * ls(y,y')\big) \ \wedge \ markers \mathbin{-\!\!\circledast} \big(markable(x) \wedge markable(y)\big) \tag{2.1}$$

is equivalent to $\varphi_{=}(x,y)$, and cannot thus be expressed without magic wand.

[1] This result may be strengthened to the equally expressive fragment

$$\boxed{\varphi, \psi \quad := x = y \ , \ x \hookrightarrow y \ , \ \varphi \wedge \psi \ , \ \neg\varphi \ , \ \varphi \mathrel{-\!\!*} \psi \ , \ \exists x.\varphi}$$

as explained in Section 2.3

Let us explain all of these formulas step by step, taking intuition from Figure 2.2: a location is *marking* if it is pointed by at least three cells, and a location is *marked* if it is pointed by a marking cell (note how this definition ensures that original list cells are not considered as marking). The aim of the final formula is to assert that the length of the two lists is n for a same n, by introducing $n+1$ unallocated marking locations. We say that a list is *marked* if all its locations are marked injectively, and these locations are the only marked ones. We say that the list is *markable* if it is possible to add disjoint list segments (hence forming no new marking location) in such a way that the list becomes marked. In other words, if all marking locations are unallocated, a list is markable if and only if there are as many marking locations as the length of the list plus one.

Now the formula 2.1 reads as "there is a set of marking locations disjoint from the locations of $ls(x)$ and $ls(y)$, such that both $ls(x)$ and $ls(y)$ are markable", which ends the proof.

Several other results strengthen the idea that adjunct's elimination and decidability of logics with the $\mathrel{-\!\!*}$ connective are rare, in particular in the field of Context Logic [39] and boolean BI [40]. The major difference with these results is that they rely on atomic formulas being not specific predicates, but propositions denoting sets of elements from an arbitrary monoid. These similarities may not be incidental, but due to the very different semantics of these logics, it seems hard to formally unify these results – as a matter of fact, all existing proofs of such results largely differ.

2.2 Separation Logic and Monadic Second Order Logic

It seems now quite admitted that symbolic heaps are the fragment of SL that should be used for automation purpose. But understanding the role of each of the restrictions they impose on formulas, and how it helps to solve the entailment problem, should be precised. We already noticed that $\mathrel{-\!\!*}$ added a facility for comparing the lengths of two lists, a property that, due to Iosif's results [41], quickly leads to undecidability. Before delving more into Brochenin's results in the next section, let us take a detour to SL_{LS} without magic wand:

$$\boxed{\varphi \ := \ \ \varphi \wedge \varphi \ , \ \neg\varphi \ , \ \varphi * \varphi \ , \ x \hookrightarrow y \ , \ x = y \ , \ \forall x.\varphi}$$

$\mathsf{FO}_{LS}(*)$

The following result, largely inspired by a result of Dawar, Ghelli and Gardner [42], shows that $\mathsf{FO}_{LS}(*)$ is strictly more expressive than first-order logic, or in other words, it does not have the elimination of the separating conjunction.

Theorem 2.3 (List Predicate's Encoding) *The predicate $ls(x, y)$ can be expressed in $\mathsf{FO}_{LS}(*)$, hence $\mathsf{FO}_{LS}(*)$ is more expressive than FO_{LS}.*

Proof sketch. Consider the formulas

$$\begin{aligned}
\mathit{extremities}(x,y) \;&\triangleq\; \quad (- \mapsto y) \wedge \neg(y \mapsto -) \\
&\quad\; \wedge \forall z. z \neq x \Rightarrow z \neq y \Rightarrow \big(\;\; (- \mapsto z) \Leftrightarrow (z \mapsto -) \;\;\big) \\
\mathit{precisely}(\varphi) \;&\triangleq\; \varphi \;\wedge\; \neg(\varphi * \neg \mathsf{emp}) \\
\varphi(x,y) \;&\triangleq\; (x = y \wedge \mathsf{emp}) \vee \mathit{precisely}\big(\mathit{extremities}(x,y)\big)
\end{aligned}$$

The formula $\mathit{extremities}(x,y)$ expresses that y is the end of a list, and the only possible end of any list, whereas x is the only possible head of any list. In particular, $\mathit{extremities}(x,y) \vDash \mathit{ls}(x,y) * \top$. The construct $\mathit{precisely}(\varphi)$ turns φ into a precise formula: it characterizes the minimal models of φ for the division ordering. For this reason, $\varphi(x,y)$ is equivalent to $\mathit{ls}(x,y)$.

Since reachability cannot be expressed in first-order logic, the previous example complicates a bit the resolution of the entailment problem for $\mathsf{FO}_{LS}(*)$ – for instance, if we were to transfer it to an SMT solver, we would have to check how it could deal with reachability. Making a step beyond FO_{LS}leads us to weak monadic second order logic over heaps with lists (MSO_{LS}), where some second-order variables quantify over finite sets of locations.

$$\boxed{\varphi \quad := \quad \varphi \wedge \varphi \;, \neg\varphi \;,\; \exists x.\varphi \;,\; \exists X.\varphi \;,\; x \hookrightarrow y \;,\; X(x)}$$

MSO_{LS}

Proposition 2.1 **([43])** *For all formula φ of $\mathsf{FO}_{LS}(*)$, there is an equivalent closed formula ψ of MSO_{LS} that can be computed in logarithmic space.*

The encoding of $\mathsf{FO}_{LS}(*)$ in MSO_{LS} is quite simple (it suffices to observe that the semantics of $\mathsf{FO}_{LS}(*)$ can be expressed in MSO_{LS}), and it immediately gives the decidability of the logic due to an old result of Rabin [44].

Proposition 2.2 **([43])** *The satisfiability problem of $\mathsf{FO}_{LS}(*)$ is decidable.*

Resorting to MSO_{LS} may however look like a bit drastic, and one may look for something a bit more efficient. This is however not so easy for at least two reasons. First, the entailment problem for $\mathsf{FO}_{LS}(*)$ already is quite high in the complexity hierarchy, due to a reduction to the satisfiability of FO over finite words [45].

Proposition 2.3 **([43])** *The satisfiability problem of $\mathsf{FO}_{LS}(*)$ is non-elementary.*

The second reason why translating $\mathsf{FO}_{LS}(*)$ into MSO_{LS} would probably be close to the optimal way of dealing with it is that $\mathsf{FO}_{LS}(*)$ may express lots of properties that belong to MSO_{LS}, as pointed by Dawar, Ghelli and Gardner in their study of the spatial logic for graphs [42]. Perhaps the most significant example in that respect is their encoding of regular languages. Switching to lists with values, we assume[2] that instead of the $(x \hookrightarrow y)$ predicate of pure lists, we have a predicate $(x \hookrightarrow y, a)$ for each letter a of a finite alphabet Σ.

[2] This is just for convenience, it would also be possible to represent finite words in pure lists, although in a more complex way [43].

Definition 2.1 (Definable language) *Let Σ be a finite alphabet, $\Sigma \subseteq \mathsf{Val}$. The list encoding h_w of a word $w = w_1 \dots w_n \in \Sigma^*$ is the list of length n such that the ith cell of the list contains w_i in its second field.*

We say that a language $L \subseteq \Sigma^*$ is definable in $\mathsf{FO}_{LS}(*)$ (resp. MSO_{LS}) if there is a formula φ of $\mathsf{FO}_{LS}(*)$ (resp. MSO_{LS}) such that for all $w \in \Sigma^*$, $(s, h_w) \vDash \varphi$ if and only if $w \in L$.

Proposition 2.4 (Dawar & al [42]) *Let $L \subseteq \Sigma^*$. Then the following are equivalent:*
L is definable in $\mathsf{FO}_{LS}(*)$ $\Leftrightarrow$ L is regular $\Leftrightarrow$ L is definable in MSO_{LS}.

This result shows that it is not completely clear whether $\mathsf{FO}_{LS}(*)$ is strictly less expressive than MSO_{LS}. A similar problem was proposed by Dawar, Ghelli and Gardner for the graph logic [42], and was partly answered by a result of Marcinkowski [46]. The proof technique of Marcinkowski cannot be adapted to the SL_{LS} case, and the conjecture of the strictness of the inclusion of SL_{LS} into MSO_{LS} was settled by Antonopoulos and Dawar only quite recently:

Theorem 2.4 (Antonopoulos & Dawar [47]) *$\mathsf{FO}_{LS}(*)$ is strictly less expressive than MSO_{LS}.*

Antonopoulos & Dawar's result might be expected, this is nonetheless a precious result, especially if one tried to prove it and experienced how it was embarrassingly difficult. One might be tempted to conclude that the $\mathsf{FO}_{LS}(*)$ fragment has only bad properties: it has a very high complexity (much higher than for instance symbolic heaps), but its expressivity is significantly limited with respect to other well-known decidable formalisms.

2.3 The Almighty Wand

We now turn our attention again on SL_{LS}, hence to the expressive power of the magic wand. First, it must be observed that the semantics of $-\!\!*$ is not a sentence of MSO_{LS}, and Brochenin's counter-example shows that there is no way to encode SL_{LS} in MSO_{LS}. But the semantics of SL_{LS} is a second-order sentence, since it quantifies only over first-order functions. We may thus consider how it related to weak[3] second-order logic (SO), where second-order variables quantify over finite sets of tuples of locations.

$$\boxed{\varphi \quad := \quad \varphi \wedge \varphi \ , \neg\varphi \ , \ \exists x.\varphi \ , \ \exists X(x_1, \dots, x_n).\varphi \ , \ x \hookrightarrow y \ , \ X(x_1, \dots, x_n)}$$
SO_{LS}

The fact the whole semantics of SL_{LS} is a second-order sentence entails the following result: for any formula φ of SL_{LS}, there is a closed formula ψ of SO_{LS}

[3] Note that weak SO is not full SO: second-order variables can only denote finite sets of tuples.

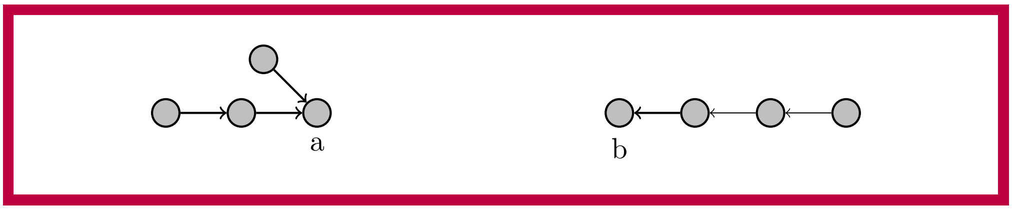


Figure 2.3: An encoding of the binary relation $\{(a,a),(a,b)\}$.

that has exactly the same models as φ. Brochenin's *tour de force* was to show that the converse is true[4], and even more. Let $\mathsf{FO}_{LS}(\mathrel{-\!\!*})$ denote the fragment of SL_{LS} without $*$, i.e.

$$\boxed{\varphi \ := \ \ \varphi \land \varphi \ , \ \neg\varphi \ , \ \varphi \mathrel{-\!\!*} \varphi \ , \ x \hookrightarrow y \ , \ x = y \ , \ \forall x.\varphi.}$$

$\mathsf{FO}_{LS}(\mathrel{-\!\!*})$

Theorem 2.5 (Star elimination [43]) *SL_{LS}, SO_{LS}, and $\mathsf{FO}_{LS}(\mathrel{-\!\!*})$ have the same expressive power, and translations among them are logarithmic-space.*

The proof of this result is quite tedious. One of the key idea was already exposed in the counter-example to adjunct's elimination, when we defined a formula $\varphi_{=}(x,y)$ that characterises heaps where $ls(x,x')$ and $ls(y,y')$ have the same length. We illustrate the other key idea of the translation on one example:

Example 1 (Encoding a SO quantification) *Assume a formula $\varphi_0(X)$ of pure* FO*, without any occurrence of $(. \hookrightarrow .)$, or* emp*, and with a free* SO *variable X of arity 2 (for instance, $\varphi_0(X)$ could be the property "if X is an order relation, then it has a smallest element"). Let us express $\exists X.\varphi_0(X) \land \mathsf{emp}$ as a formula of $\mathsf{FO}_{LS}(\mathrel{-\!\!*})$. Building on the counter-example for adjunct's elimination, we can define a formula "$d(x,x') = d(y,y') + 1$", that expresses that the distance from x to x' is one more than from y to y'. Consider now the formulas:*

$$\begin{aligned} max\_ls(x',x) \ &\triangleq \ \ ls(x',x) * \top \ \land \ \neg\big((- \hookrightarrow x') \lor (x \hookrightarrow -)\big) \\ pair(x,y) \ &= \ \ \exists x',y'.max\_ls(x',x) \land max\_ls(y',x) \land d(x',x) = d(y,y) + 1. \end{aligned}$$

The formula $pair(x,y)$ asserts that x and y are the tail of maximal lists of length n and $n+1$ respectively. This property associates a binary relation $pair_h \subseteq \mathsf{Loc}^2$ to any heap h. Moreover, any binary relation can be realized by $pair_h$ for some adequate choice of h. For instance the binary relation $\{(a,a),(a,b)\}$ over the set $\{a,b\}$ can be represented as the heap of Figure 2.3. Thanks to this encoding of binary relations as heaps, the formula $\mathsf{emp} \land \big(\top \mathrel{-\!\!\circledast} \varphi_0(pair)\big)$ – where $\varphi_0(pair)$ is defined by replacing $X(x,y)$ with $pair(x,y)$ in $\varphi_0(X)$ – expresses $\exists X.\varphi(X) \land \mathsf{emp}$.

[4] A correspondence between SL and SO was already stated by Kunczak and Rinard [48], but the separation logic they consider separates n-ary relations, and not heaps. In their case, all of the expressivity of SO could then be encoded in a separation logic *without* magic wand.

This example shows how it is possible to encode a second-order quantification inside the heap, based on arithmetic relations. This is of course a very specific case of second-order quantification, and the generalisation of this idea has to face several problems that are detailed in the journal version [49].

Note that the result holds for SL, FO($-\!\!*$), and SO, *i.e.* for the same logics over heaps with two selector cells and the predicate $(x \hookrightarrow y, z)$. The proof of Theorem 2.5 indeed adapts very well to the case of several selectors – a direct proof would also be worth to derive, and would probably be much simpler.

2.4 Back to Completeness

A remarkable consequence of the equivalence between SL and SO is the following result:

Proposition 2.5 (Weakest precondition) *Let p be a sequential program, and φ a formula of* SL*. Then there is a formula ψ of* SL *such that $\{\psi\}$ p $\{\varphi\}$, and for any other ψ' such that $\{\psi'\}$ p $\{\varphi\}$, $\psi' \vDash \psi$.*

Proof sketch. s, h is in the weakest precondition of p, φ if and only if:

- there is not a finite run $p, s, h \rightsquigarrow^* \mathtt{skip}, s', h'$ such that $s', h' \vDash \neg\varphi$, and
- there is not a finite run $p, s, h \rightsquigarrow^*$ **OwnError**.

Let us sketch how to express this in SO. First, observe that a finite sequence $(p_i, s_i, h_i)_{i=0\ldots l}$ can be represented as some term tree (considering that only finitely many variables should be cared about for s, a finite run can be represented as a finite term). Obviously, the quantification over a term can be expressed in weak SO (this is a particular case of a graph). Then, the check that a term denotes a valid run can be easily expressed in SO, as the definition of $\rightsquigarrow$ is definable in SO. Finally, two more things need to be expressible in SO: whether s_0, h_0 is isomorphic to s, h, and whether $s_l, h_l \vDash \varphi$. This can be done for any reasonable encoding of a pair (s, h) as a term.

The expressibility of weakest preconditions is the key of the completeness of Hoare logic. The proof technique can be easily adapted to Separation Logic, and yields the following result:

Theorem 2.6 (Completeness) *Let p be a sequential program, and φ, ψ be* SL *formulas. Then the following three statements are equivalent:*

1. $\vDash \{\varphi\}$ p $\{\psi\}$.
2. $\vdash \{\varphi\}$ p $\{\psi\}$ *according to the rules of Figure 1.5.*
3. $\{\varphi\}$ p $\{\psi\}$ *is derivable without the rules for auxiliary variables and the frame rule, using instead the backward axioms of Figure 2.4.*

$$\{\forall x.\ ((x \mapsto -) \mathrel{-\!\!*} \varphi)\}\ \text{x:=new()}\ \{\varphi\} \qquad \{(x \mapsto -) * \varphi\}\ \text{dispose(x)}\ \{\varphi\}$$

$$\{\exists x.x = y \wedge \varphi\}\ \text{x:=y}\ \{\varphi\} \qquad \{\exists x.(y \hookrightarrow (x, -)) \wedge \varphi\}\ \text{x:=y.1}\ \{\varphi\}$$

$$\{\exists x.(y \hookrightarrow (-, x)) \wedge \varphi\}\ \text{x:=y.2}\ \{\varphi\}$$

$$\{\exists x_1, x_2.(x \mapsto (x_1, x_2)) * \Big((x \mapsto (y, x_2)) \mathrel{-\!\!*} \varphi\Big)\}\ \text{x.1:=y}\ \{\varphi\}$$

$$\{\exists x_1, x_2.(x \mapsto (x_1, x_2)) * \Big((x \mapsto (x_1, y)) \mathrel{-\!\!*} \varphi\Big)\}\ \text{x.2:=y}\ \{\varphi\}$$

where x_1 and x_2 are "fresh" variables, *i.e.* do not occur in φ and differ from x and y.

Figure 2.4: Axioms for weakest preconditions of atomic commands

Proof sketch. (2) ⇒ (1) is by soundness of SL. Backward axioms are derivable from the small axioms using the frame rule and the rule for auxiliary variables, hence (3) ⇒ (2). The proof of (1) ⇒ (3) goes as follows: it is proved by induction on p that $\{\varphi\}$ p $\{\psi\}$ is derivable in the particular case where φ is the weakest precondition. Then the result holds for the general case by consequence rule. The inductive cases (while, if, sequence) are by definition of the weakest precondition and rely on the consequence rule. The base cases are by hypothesis.

This result strengthens the one from [12], which rely on infinite disjunctions. The elimination of the AVE and frame rules could probably be obtained in a more effective way, for instance by means of a rewrite system on proof trees that postpone these rules in a similar way as the cut rule in sequent calculi[5].

This completeness result is still... incomplete. First, because it does not say anything about programs with function calls or concurrency, and second because it does not evaluate how the application of the frame rule may reduce the size of a proof, which are precisely the features that are expected to make the strength of Separation Logic.

We observed in Section 1.2 that SL was incomplete for concurrent selfish programs due to possible synchronisations on cell's allocation and deallocation, and raised the question of the completeness of SL for synchronisation-free programs (in particular, terminating programs). We conclude this section by an example

[5] The possibility of postponing the frame rule seems a very general property, and would probably also hold for the parallel rule, the rule for function calls, the anti-frame rule, etc. On the other hand, the possibility of postponing the rule of auxiliary variables seems much more dependent of the proof system: for instance, it cannot be posponed after the parallel rule.

program that illustrate the difficulties in deriving the completeness of SL even in this case.

Consider a program that :

1. first starts with two disjoint lists x and y of the same length,

2. then traverses each of the lists in parallel,

3. then traverses again the two lists simultaneously, in a same loop, halting when the last cell of x has been reached.

More precisely, we consider the following program:

```
1  traversal(z){
2    local z';
3    z':=z;
4    while(x'!=NULL) z':=z'.1;
5  }
6  main{
7    {traversal(x) || traversal(y)}
8    while(x!=NULL) {
9      x:=x.1; y:=y.1;
10   }
11 }
```

Let us introduce the property

$$\varphi_{=}(x,y) \triangleq \big(ls(x) * ls(y)\big) \ \wedge \ \text{“}\mathsf{length}(x) = \mathsf{length}(y)\text{”},$$

hence stating that the heap contains precisely two allocated lists of the same length, starting at `x` and `y` respectively, and sharing no cells. Then

$$\{\varphi_{=}(x,y)\} \text{ main } \{y = \text{null}\} \tag{2.2}$$

is valid – observe that the precondition $\varphi_{=}(x,y)$ is crucial, because the test in the while loop on line 7 does not suffice to prevent from an illegal dereferencing or a memory leak.

To prove the valid Hoare triple (2.2), it seems to be hard to avoid to introduce a ghost variable, if we want to use a proof system that is sound for our selfish semantics – by this, we exclude proof systems that allow concurrent read accesses, like the ones using permissions or rely-guarantee.

Assume by absurd that there is such a proof of (2.2) without a ghost variable, *i.e.* without applying the AVE rule. Then it must induce a proof of

$$\{\varphi_{=}(x,y)\} \ \text{traversal}\,(\text{x},\text{x}',\text{t}\,)||\ \text{traversal}\,(\text{y},\text{y}',\text{u}) \ \{\varphi_{=}(x,y)\} \tag{2.3}$$

But such a proof is impossible: the only rules that can be applied first are the frame rule, the consequence rule, or the parallel rule. Clearly, no cells can be framed out, nor any pure assertion, so the only possible application of a frame rule is with emp as a frame, and can be eliminated. Then a proof of (2.3) starting with the application of the consequence rule (possibly without changing pre and post conditions), followed by the parallel rule requires to exhibit φ_1, φ_2 such that

$$\begin{array}{rll} \models & (\varphi_1 * \varphi_2) \Leftrightarrow \varphi_{=}(x,y) & (a) \\ \varphi_1 \models & ls(x) * \top & (b) \\ \varphi_2 \models & ls(y) * \top & (c) \end{array}$$

– (b) and (c) comes from the right formulas being the weakest preconditions of the traversal procedure. It can be seen then that $x \neq y \wedge x \mapsto \text{null}$ and $x \neq y \wedge x \neq y' \wedge (y \mapsto y' * y' \mapsto \text{null})$ should then respectively entail φ_1 and φ_2, hence the contradiction with (a).

If we use the AVE rule, if we assume that variables can store naturals, and if we allow some more expressivity of the logic to talk about these naturals, then a proof seems amenable. Let us introduce the property

$$\psi_{=}(x,z) \triangleq ls(x) \wedge \text{“}\mathsf{length}(x) = z\text{”},$$

hence expressing that the heap precisely contains a list starting at x, and that this has the same length as the natural number stored in z. Then, in order to prove (2.3), we may introduce a ghost variable z using AVE and prove

$$\{\psi_{=}(x,z) * \psi_{=}(y,z)\} \text{ traversal (x,x',t)|| traversal (y,y',u) } \{\psi_{=}(x,z) * \psi_{=}(y,z)\} \tag{2.4}$$

which can be proved applying the parallel rule – note that sharing a ghost variable or a ghost cell is sound with respect to the selfish semantics, because concurrent read accesses to such resources do not really occur at runtime.

All the assumptions that permitted this proof are quite reasonable, but it might be interesting to reject the hypothesis that $\psi_{=}(x,z)$ is expressible in the assertion language. For instance, we may consider proving this program in a tool that does not support anything else than pure list heaps. One may consider to encode the counter z via a ghost list of the same length as the two other lists, and then share this list when the parallel rule is applied – note again that sharing a ghost list is sound. But now, in order to allocate this ghost list, it seems we need to allow to introduce ghost code.

One may expect that SL extended with a rule for ghost code introduction would be complete, due to similar results for Owicki-Gries and rely-guarantee program logics. This may be a bit disapointing, because ghost code seems pretty hard to guess automatically, and delicate to check being actually “ghost” in some circonstances, and moreover because ghost code may be thought as required in other proof systems to better handle interferences from the environment, which do not occur for selfish programs. All in all, it seems hard to design a proof system that exactly captures selfish programs[6], despite the interest of this subclass of race-free programs.

[6] At best, one can conjecture that RGSep, restricted to read interferences, would capture exactly the race-free programs.

2.5 Extensions with Data

So far, we exploited very little of the second field of cells. This second field has several interests: the modeling of doubly-linked lists, the modeling of non-linear recursive data structures (like trees), but also the modeling of simple lists with data. We discuss here how to deal with this last kind of heaps. We hence consider heaps $h : \mathsf{Loc} \rightharpoonup \mathsf{Val} \times \mathsf{Val}$ of lists with data, *i.e.* heaps such that the second fields contain invalid heap addresses only, or, in other words, states s, h such that $s, h \models \forall x.(- \hookrightarrow -, x) \Rightarrow \neg(x \hookrightarrow -, -)$. We also consider the extension of $\mathsf{FO}_{LS}(*)$ with the predicate $(x \hookrightarrow y, z)$ dealing with the second field:

$$\boxed{\varphi \ := \ \varphi \wedge \varphi \ , \ \neg\varphi \ , (x \hookrightarrow y, z) \ , \ x = y \ , \varphi * \varphi \ , \exists x.\varphi}$$

$\mathsf{FO}(*)$

and write $\mathsf{FO}_{dat}(*)$ to denote the theory of $\mathsf{FO}(*)$ over heaps of lists with data. The following result shows that this small extension already provides too much expressive power:

Theorem 2.7 **([6])** *The satisfiability problem is undecidable for* $\mathsf{FO}_{dat}(*)$.

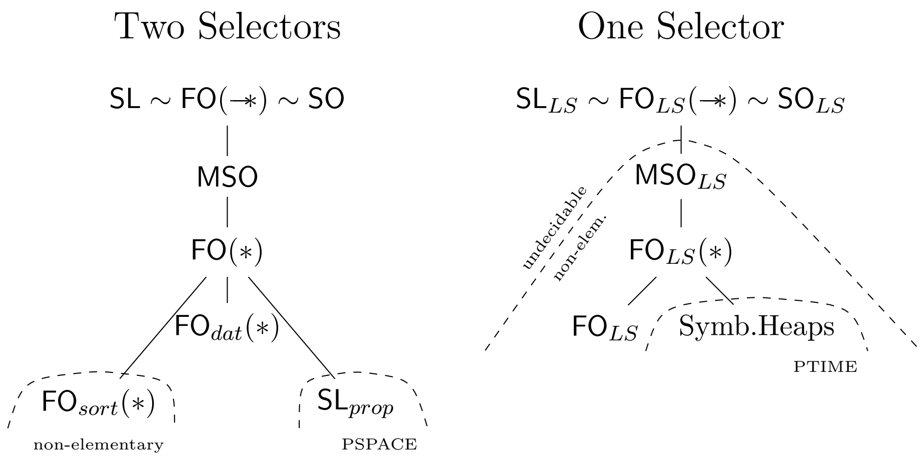
This result comes from the ability to encode the FO theory of finite totally ordered sets equipped with an equivalence relation $(S, <, \sim)$, or equivalently the FO theory of finite *data* words. Even the first-order theory of these structured is undecidable – it is unclear whether restricting the logics to two variables would make the logic decidable, like in the FO case [50]. It is however desirable to provide an extension of SL that could deal with lists with data, and more precisely sorted lists, and programs manipulating them. Having this purpose in mind, one may assume that $\mathsf{Val} = \mathbb{N}$ (or $\mathsf{Val} = \mathbb{Z}$), and seek a fragment of SL that may compare values that appear in the second fields of cells. The design of such a fragment should moreover allow to express properties that may be needed by the proofs or example programs. We introduced the following fragment:

$$\boxed{\begin{array}{rcl} pred & := & (x \hookrightarrow y, z) \ , x \overset{\geq}{\hookrightarrow} y \ , \ x \overset{\leq}{\hookrightarrow} y \ , \ x \leq val(\underline{y}) \ , x \geq val(\underline{y}) \\ \varphi & := & pred \ , \ \varphi \wedge \varphi \ , \ \neg\varphi \ , \ \varphi * \varphi \ , \exists x.\varphi \end{array}}$$

$\mathsf{FO}_{sort}(*)$

where underlined variables denote variables that are not quantified in any context. The new predicates allow to compare the values of one cell with either the value of its successor's cell ($\overset{\geq}{\hookrightarrow}$, $\overset{\leq}{\hookrightarrow}$) or with the value of a cell at a program's variable ($x \leq val(\underline{y})$, $x \geq val(\underline{y})$). The formal semantics of these new predicates is:

$$\begin{array}{lll} s, h \models & x \overset{\geq}{\hookrightarrow} y & \text{if } s(x), s(y) \in dom(h) \text{ and } snd(h(x)) \geq snd(h(y)) \\ s, h \models & x \geq val(\underline{y}) & ifs(y) \in dom(h) \text{ and } s(x) \geq snd(h(y)) \end{array}$$



Elimination Properties

Adjunct Elimination	SL_{prop}
Star Elimination	SL,SL_{LS},MSO, MSO_{LS}, SL_{prop}
List Predicate's Elimination	SL_{LS}, $\mathsf{FO}_{LS}(-\!\!*)$, MSO_{LS},$\mathsf{FO}_{LS}(*)$

Figure 2.5: Summary of the results of this chapter

Example of properties that can be stated in $\mathsf{FO}_{sort}(*)$ are thus "there is an ordered list starting at x", or "all values of the sorted list starting at x are smaller than the ones of the ordered list starting at y", but it cannot be said that a list contains pairwise distinct values. $\mathsf{FO}_{sort}(*)$ seems actually expressive enough to run a symbolic execution of a program merging two sorted lists [6]. It is also a decidable logic:

Theorem 2.8 ([6]) *The satisfiability problem for* $\mathsf{FO}_{sort}(*)$ *is decidable.*

The decidability proof goes by reduction to MSO_{LS} – note that the actual values of the second field can be abstracted, since only their order constraints do matter. This is probably yet not very satisfactory, as the complexity of the problem is, like in the case of pure list, non-elementary.

Chapter 3

Copyless Message Passing

Sing♯ is an experimental programming language developed for the design of the Singularity operating system [51]. All kernel processes in Sing♯ share their address space with each others, but a form of memory protection of each process is enforced at compile-time. These so-called "software isolated processes" (SIP) are thus a bit like selfish threads, in the sense of selfishness we introduced in Chapter 1. Sharing the address space among processes has several benefits over a paranoid hardware memory protection. The one that interests us is the following: message-passing across processes can be implemented in a copyless way, in the sense that, whenever a message is sent from one process to an other, a mere pointer to the memory region of the content of the message is inserted in the communication buffer.

This form of message-passing gives much better performances than a copying approach of message-passing, but it complicates the static analysis of the selfishness of the programs at compile-time. To help the compiler in his task, Sing♯ presents an interesting mechanism of contracts. These contracts may express conditions like "if I receive a message *order*, then I will answer it with a message *acknowledgment*", and are thus a human-readable, specialized form of rely-guarantee reasoning for message exchanges.

We intend now to model these two aspects of Sing♯, copyless message-passing and contracts, in the framework of Separation Logic.

3.1 A model of Sing♯

In Sing♯, processes communicate through channels. A channel is composed of a pair of two endpoints. An endpoint can be used to send messages to its peer endpoint, or to receive messages from it, exactly like UNIX sockets. Communications are asynchronous FIFO like in TCP. Endpoints are heap objects like others, which implies two things: they are dynamically allocated and disposed, and they can be sent in messages as any other heap objects, yielding a form of mobility.

$$
\begin{array}{lll}
m \in \Sigma \ ::= & \mathrm{cell}\,,\ \mathrm{list}\,, \mathrm{ack}, \mathrm{close\_me}, \ldots & \text{(Message Tags)} \\
p \ ::= & & \text{(Programs)} \\
& \ x := \mathsf{new}()\ , \mathsf{dispose}(E)\ ,\ x := E\ , & \\
& \ x := E.i\ ,\ E.i := E'\ ,\ \mathsf{skip}\ , & \text{(see Chapter 1)} \\
& \ \mathsf{while}\ b\ \mathsf{do}\ p\ ,\ \ \mathsf{if}\ \ b\ \mathsf{then}\ p_0\ \ \mathsf{else}\ \ p_1\ , & \\
& \ p_0; p_1\ , p_0 \parallel p_1\ , & \\
& (e,f) = \mathsf{open}(C) & \text{(allocation)} \\
& \mathsf{close}\,(E,F)\ , & \text{(disposal)} \\
& \mathsf{send}(m, E, \vec{E})\ , & \text{(sending)} \\
& \vec{x} := \mathsf{receive}\,(m, E)\ , & \text{(reception)} \\
& \mathsf{switch} \left\{ \begin{array}{c} \vec{x}_1 := \mathsf{receive}\,(m_1, E_1) : p_1 \\ \ldots \\ \vec{x}_n := \mathsf{receive}\,(m_n, E_n) : p_n \end{array} \right\} & \text{(scanning)}
\end{array}
$$

Figure 3.1: Syntax of a toy programming language with message passing

3.1.1 A Toy Programming Language

We consider the toy programming language of Figure 3.1, obtained by adding four new primitives and a new language construct to the toy programming language of Chapter 1. The primitive $(e, f) = \mathsf{open}(C)$ allocates a channel and stores its two endpoints in the variables e and f; this channel is ruled by a contract C, which will be later explained. The primitive $\mathsf{close}\,(e, f)$ deallocates the channel (e, f) in a symmetric way[1]. The command $\mathsf{send}(m, \vec{E})$ sends a message tagged with m, over endpoint E, and with as many extra parameters $\vec{E} = E_1, \ldots, E_n$ as required by the arity of the tag m. Symmetrically, the command $\vec{x} = \mathsf{receive}\,(m, E)$ receives a message tagged with m over endpoint E, and store the value of its parameters in $\vec{x}$. The `switch` construct scans the incomming buffers of the E_i and branches on one of the case $\vec{x}_i = \mathsf{receive}\,(m_i, E_i) : \{p\}$ such that m_i is the tag of the first out message of the incoming buffer of E_i.

Example 2 *The following program illustrates a very simple usage of the communication primitives*[2]*:*

[1] This treatment of closure differs from Sing♯, where each endpoint is "closed" separately: in Sing♯, the first call to $\mathsf{close}\,(e)$ sends a special message towards the other endpoint, and the second $\mathsf{close}\,(f)$ really closes the channel. Example 3 shows how this mechanism can still be represented in our model.

[2] For the sake of this example, we admit here and elsewhere in other examples a standard construct for local variables, which we omitted from the formal syntax for conciseness reason.

```
local e,f in
  (e,f)=open(C);
  send(string,e,x);
  y = receive(string,f);
  close(e,f);
```

This program is operationally equivalent to $y := x$. Note how the copyless aspect manifests itself here: x and y point to the same array of characters. By contrast, for a copying semantics, the above program would be equivalent to something like $y :=$ string_copy(x).

Let us now consider a relatively more complex example that illustrates the interaction between heap-manipulation and message-passing, as well as the interest of the switch construct.

Example 3 *The following program features a producer/consumer general schema:*

```
1  send_and_dipose_list(x){
2   local e,f;
3   (e,f) := open(C);
4   producer(x,e) || consumer(f)
5  }

6  producer(x,e){
7   local t;
8   while x != null {
9     t := x->next;
10    send(cell,e,x);
11    x := t;
12    receive(ack,e);
13  }
14  send(close_me,e,e);
15 }
16
17

18 consumer(f){
19  local y,e;
20  while true {
21   switch {
22    y := receive(cell,f)     : {
23      dispose(y); send(ack,f);}
24    e := receive(close_me,f) : {
25       break }
26   }
27  }
28  close(e,f);
29 }
```

The left thread sends the list $ls(x)$ cell by cell to the right thread, and this one immediately deallocates the cells it receives, and sends an acknowledgment. When the end of the list is reached, the left thread sends its endpoint to the other thread to inform it that the traversal is finished and that it can close the channel.

Interactions between message-passing and heap manipulations are subtle. Let us observe a few facts about Example 3. On the one hand, it can be observed that the two instructions dispose(x) and send(ack,f) on line 23 can be executed in any order. But one should be careful when commuting message-passing and heap-manipulating instructions. Consider on the other hand the variant of the program of Example 3 where the local variable t is omitted, and the lines 9...11 become

```
9  send(cell,e,x);
10 x := x->next;
11
```

Then this program is not safe any more, as the following scenario is possible: the producer executes line 9 in this new code, then the consumer receives the cell x and dispose it, and finally the producer dereferences a non-allocated pointer on line 10.

The notion of ownership better explains the difference between the two messages: the sending of the `cell` message transfers the ownership of the cell `x` from the producer to the consumer, and for this reason, it would not be possible to dereference `x` in the producer after the `cell` message has been sent. We say that the heap region consisting of the cell is the *footprint* of the message. By contrast, the `ack` message has an empty footprint, transfers no ownership, and acts as a pure synchronization. It can thus freely commute with the disposal. It may further be observed that one may simplify the program and not exchange a `ack` message[3] on lines 12 and 23.

3.1.2 Ownership is not just in the Eye of the Asserter

Remember that Sing♯ has to deal with processes, and not threads. In particular, it must be possible for the operating system to reclaim the memory used by a process when this one dies, even if other processes keep running and try to interact with this process (note that processes can be killed at any point in time). The operating system should thus garbage collect the cells that were owned by the dead process, but not the ones that may be used by other processes in the future.

In order to determine ownership, Sing♯ cannot rely only on the static analysis at compile-time (again, think of processes being abruptly killed). Enough runtime information must thus be maintained to determine ownership – for instance, each cell can be tagged with a process-id.

When a piece of heap is sent, its ownership information must be updated; and if this piece of heap contains endpoints, then the owner of the messages that are waiting to be received on these endpoints should be updated as well, because of the following *transitivity rule*:

Transitivity Rule *when a process owns an endpoint, it also owns the messages that are waiting to be received on that endpoint.*

Note that the transitivy rule may potentially slow down message exchanges significantly. Indeed, when an endpoint is passed, the update of the ownership information requires to recursively examine all messages of all queues whose

[3] In Sing♯, the ack message is required in order to ensure that the queue of incoming messages over f is bounded, and can thus be allocated once for all at channel's allocation – but other implementations of channels, *e.g.* based on lists, would not show this limitation.

```
 1 main(){
 2   local e,f,e',f';
 3   (e,f) := open(C);
 4   (e',f') := open(C);
 5   producer(e,f,e') || consumer(f')
 6 }

 7 producer(e,e',f') {             13 consumer(f) {
 8   local x;                      14   local f',x;
 9   x:=new();                     15   f' := receive(endpoint,f);
10   send(cell,e',x);              16   x  := receive(cell,f');
11   send(endpoint,e,f');          17   dispose(x);
12 }                               18 }
```

Figure 3.2: Producer/Consumer with one indirection: an example of the recursivity of ownership transfer.

```
mem_leak_generator(){             mem_leak_gen2(){
  local x,e,f;                     local e,f,e',f';
  x:=new();                        (e,f) = open(C);
  (e,f) = open(C);                 (e',f') = open(C');
  send(cell,e,x);                  send(endpoint,e,f');
  send(endpoint,e,f);              send(endpoint,e',f);
}                                 }
```

Figure 3.3: Two examples of distributed memory leaks

ownership changes. For instance, on the program of Figure 3.2, when the ownership of the endpoint f' is passed to the consumer (line 11), the ownership of the cell x should be immediately transferred to the consumer by application of the transitivity rule.

3.1.3 Shallow Ownership Transfer

In some situations, it is impossible to determine how the ownership information should be updated. This occurs for instance in the program `mem_leak_generator` of Figure 3.3. This program sends two messages in the queue of f: first the cell x, then the endpoint f. Applying the transitivity rule does not help deciding who is the owner of the endpoint f at the end of the program. The same occurs in the program `mem_leak_gen2()` above, illustrating how circular reasoning with the transitivity rule are not restricted to the sole case of sending an endpoint over its peer. Note also that such situations of undetermined ownership are memory leakages, since they will not be handled by any process-wide garbage collector.

These ownerless cells could be declared owned by the operating system and treated by a system-wide garbage collector. But Sing♯ avoid a system-wide garbage collection for such cells, as endpoint mobility is restricted as follows:

Shallow Ownership Transfer *when an endpoint ε is sent, its incoming queue should be empty, and should remain empty until ε has been received.*

Shallow ownership transfers could be also stated as follows: "the transitivity rule never needs to be applied". Observe that the examples on Figure 3.3 are not shallow ownership tranfers, nor the example on Figure 3.2, nor the same example in which the two send operations would be commuted.

Aside the problem of ownerless cells, another motivation for adopting shallow ownership transfers is probably efficiency: the shallow ownership transfer hypothesis ensures that sending is constant-time.

Shallowness of ownership transfers would be hard to ensure by static analysis without any further restrictions. To check that an ownership transfer is shallow, Sing♯ compiler needs indeed to ensure a prophecy on the uses of the peer ε' of the endpoint ε that is sent: no message can be sent on ε' until ε will be received. Ensuring this prophecy directly is hard, especially with local reasoning, because ε' is in general owned by a process p' that differs from the process p that sends ε. A solution is to rely on *communication contracts*, but before we explain this solution, we should first explain the main purposes of communication contracts.

3.1.4 Communication Contracts

Communication contracts are similar to session types [52], and like other types, their purpose range from documentation of the code to runtime-safety of some specific operations – in the case of contracts, communications.

Let us briefly review some communication errors that contracts can prevent. First of all, contracts intend to prevent deadlocks. Defining deadlocks as a unique notion, like "absence of progress", is quite subtle and not fully informative. We rather provide some examples of deadlocks on Figure 3.4 and will later formalize some of these deadlocks.

Circular Wait This is probably the most familiar deadlock: a group of processes wait for each others, and no leader emerges.

Head-to-Head The head-to-head deadlock finds its name from MPI programming. It occurs when two processes simultaneously send a message to each other, which get stuck if message-passing is implemented by a rendez-vous synchronisation. It tends to be avoided in asynchronous message-passing as well to ease the portability of the code; the absence of head-to-head deadlocks corresponds to half-duplex communications, and will be later studied in Section 3.3.

Unspecified Reception A process can get stuck in a switch construct if the tag of the first available message in the queue is not one of those that are listed in the switch construct[4].

Buffer Overflow Deadlocks may happen if send operations are blocking until a receive operation frees a slot for a new message in the queue[5]

To complete the description of the problems that contracts might help to solve, we should mention **orphan messages** (see Figure 3.4), which occur when a channel is closed although it contains some pending messages.

Let us now introduce some intuitions on how these problems can be solved by communication contracts in Sing♯. Contracts themselves are quite simple: they are finite-state automata labeled with send and receive actions describing the protocol of the conversations occuring on the channels they type.

Definition 3.1 (Contracts) *A contract is a tuple $C = (Q, \Sigma, \delta, q_0, F)$ such that:*

- *Q is a finite set of* states*;*
- *Σ is a finite set of* message tags*;*
- *$\Delta \subseteq Q \times \{!, ?\} \times \Sigma \times Q$ is the set of* transitions*;*
- *$q_0 \in Q$ is a distinguished* initial state*;*
- *$F \subseteq Q$ is a set of distinguished* final states.

The dual *$\overline{C}$ of C is the contract obtained by swapping $!$ and $?$ in all transitions.*

Example 4 *Consider again the program of Example 3. Then a contract describing the underlying protocol could be the following:*

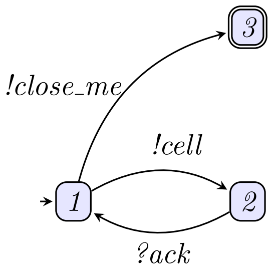
[4] If we were not to forbid unspecified receptions, we could consider two ways of handling these "deadlocks": either the process stops waiting and escapes as soon as it can determine that the pattern-matching is not exhaustive, but then the programmer should have written some code to catch the exception, or it accepts to pick a message that is not the first one in the queue, like in Erlang [53], but then it is possible to wait for such a message forever. All in all, preventing unspecified receptions seems a pretty clean and efficient discipline.

[5] This of course depends on whether the implementation of the queue forces it to be bounded, and how overflows are prevented. Due to slack *inelasticity* [54], deadlocks might be caused by adopting too large buffer bounds. An important benefit of contracts is not just that they ensure buffer boundedness, but also that one may derive from them the exact buffer bounds.

```
circular_wait() {
 (e,f) := open(C);
  {
    receive(m1,e); [..]
  ||
    receive(m2,f); [..]
  }
}

head_to_head() {
  (e,f) := open(C);
  {
    send(m1,e); [..]
  ||
    send(m2,f); [..]
  }
}

orphan_message(x) {
  (e,f) := open(C);
  send(cell,e,x);
  close(e,f);
}

unspecified_reception(x) {
  (e,f) := open(C);
  send(list,e,x)||consumer(f);
}
consumer(f) {
  local x,e;
  switch {
    x:=receive(cell,f):{
      dispose(x);}
    e:=receive(close_me,f):{
      close(e,f);}
  }
}

buffer_overflow(x,y) {
  (e,f) := open(C);
  // buffer_size(e,f)= 1 msg
  send(cell,e,x);
  send(cell,e,y);
}
```

Figure 3.4: Different forms of communication errors in message-passing programs

```
contract C {
  initial state 1 {
    ! cell → 2;
    !close_me → 3;
  }
  state 2 {
    ?ack → 1;
  };
  final state 3 {};
};
```



Preparing the next sections, let us give now some intuitions on how contracts are used to prevent communication errors. Basically, Sing♯'s compiler executes the two following verifications:

1. All the contracts provided by the programmer are *admissible*. A contract is admissible if it satisfies some syntactical restrictions, which mainly ensures that the contract as a standalone executable object is safe. Contracts can

indeed be executed as communicating finite state machines (CFSM), and may or may not feature some of the communication errors we mentioned.

2. The program is *contract obedient*: communications inside the program follow the protocol prescribed by the contract: essentially, this means that the CFSM semantics is a sound over-approximation of the program's semantics.

Formalizing these two conditions is the purpose of two sections: the check for contract obedience will be explained by means of an extension of the proof system of SL in Section 3.2, and the admissibility will be explained in terms of CFSM in Section 3.3.

3.1.5 A word on Merro's locality condition

Let us now look back at the problem of ensuring that ownership transfers are shallow. Since contracts avoid full-duplex communications, the shallowness of ownership transfer is equivalent to the following property:

Half-Duplex, Shallow Ownership Transfer *after receiving an endpoint, the first action on it must be a send action.*

Half-Duplex shallow ownership transfers are thus a weakening of the *locality condition* in the π-calculus [55], which imposes that for ever after an endpoint has been received, it can only be used for sending. The two conditions can be considered as motivated by the same problem: determining to whom a sent message should be addressed. There is however a slight difference: the locality condition in π-calculus ensures that there is always *at most one* receiver on a given channel – which is the case in Sing♯ even without shallow ownership transfer, due to selfishness. However, shallow ownership transfers ensure that there is always *at least one* receiver on a given channel – a problem specific to the asynchrony of communications in Sing♯.

The restriction to shallow ownership transfers raises the question of the expressive power that is lost by adopting this restriction. Interestingly, Merro showed that all choice-free, guard-free processes of the asynchronous π-calculus can be represented in the local fragment using a form of linear forwarders [56]. A similar result would probably hold for Sing♯ purged of some problematic features, like equality tests on endpoints. On the other hand, if we only consider selfish programs, the results we present in forthcoming Section 3.5 suggest that non-shallow ownership transfers increase the expressive power of the programming language.

3.2 A Proof System for Copyless Message-Passing

3.2.1 Assertion Language

In order to extend the proof system of SL to our model of Sing♯, we first need to extend the assertion language so as to deal with endpoints. Two informations may be required about a given endpoint e: first, its peer endpoint f, and second, its contract C and the contract's state q in which it currently is. We thus extend the assertion language with the predicate

$$e \overset{ep}{\mapsto} (C\{q\}, f)$$

that characterises such an endpoint. That's all we need. We could also have considered keeping some information about the content of the queues, but this is submitted to interferences from the environment when a thread only owns one of the two endpoints, and a closure of the queues' contents under interferences *à la* RGSep would be necessary[6]; contracts just do this in a more specialized and user-friendly way.

3.2.2 Proof Rules

We need now to provide new inference rules for the four new instructions of our toy programming language. Allocation and disposal of a channel are rather straightforward: when the endpoints are allocated, they are placed in the initial state of the contract, and can be assumed as peer of each other. Conversely, for closing a channel, the two endpoints must be peer of each other, and be in a same final state of the contract.

$$\frac{q = \mathit{init}(C)}{\{\mathsf{emp}\}\ (\mathrm{e,f}){:=}\mathrm{open}(\mathrm{C})\ \{e \overset{ep}{\mapsto} (C\{q\}, f) * f \overset{ep}{\mapsto} (\overline{C}\{q\}, e)\}}$$

$$\frac{q \in \mathit{finals}(C)}{\{E \overset{ep}{\mapsto} (C\{q\}, F) * F \overset{ep}{\mapsto} (\overline{C}\{q\}, E)\}\ \mathrm{close}\,(\mathrm{E,F})\ \{\mathsf{emp}\}}$$

Sends and receives are a bit more complicated. We need to handle two things: the ownership transfer and the obedience to the contract. Regarding ownership transfer, it is necessary to precise which part of the heap is transferred. For this, we consider that each message tag m will be associated with a formula $\varphi_m(\vec{x})$ of the same arity as m that describes the footprint of the message. Then a send acts as deallocating the footprint in the local state, and conversely a receive corresponds to an allocation of the footprint in the local state. For what

[6] Instead of interferences, we could also relax the notion of program "state" and keep some information on the past send messages or the future received messages. This is the approach followed by Appel & al [57].

concerns contract obedience, it is checked that the contract of the endpoint contains a transition tagged like the message that is sent or received. This leads to the following proof rules:

$$\frac{q \xrightarrow{!m} q' \ \in \ \mathit{transitions}(C)}{\{E \overset{ep}{\mapsto}(C\{q\},F) * \varphi_m(\vec{E})\}\ \mathsf{send}(m,E,\vec{E})\ \{E \overset{ep}{\mapsto}(C\{q'\},F)\}}$$

$$\frac{q \xrightarrow{?m} q' \ \in \ \mathit{transitions}(C)}{\{E \overset{ep}{\mapsto}(C\{q\},F)\}\ \vec{x} := \mathsf{receive}(m,E)\ \{E \overset{ep}{\mapsto}(C\{q'\},F) * \varphi_m(\vec{x})\}}$$

Example 5 *The variation around Example 2 described below can be proved with the new small axioms:*

```
1  message cell(x) {$x \mapsto -$}
2  contract C {
3    initial state 1 {!cell->2;};
4    final state 2 {};
5  }
6  main(){
7    {emp}
8    x:=new();
9    {$x \mapsto -$}
10   (e,f)=open(C);
11   {$(x \mapsto -) * e \overset{ep}{\mapsto}(C\{1\},f) * f \overset{ep}{\mapsto}(\overline{C}\{1\},e)$}
12   send(cell,e,x);
13   {$e \overset{ep}{\mapsto}(C\{2\},f) * f \overset{ep}{\mapsto}(\overline{C}\{1\},e)$}
14   y := receive(cell,f);
15   {$(y \mapsto -) * e \overset{ep}{\mapsto}(C\{2\},f) * f \overset{ep}{\mapsto}(\overline{C}\{2\},e)$}
16   close(e,f);
17   {$y \mapsto -$}
18   dispose(y)
19   {emp}
20 }
```

On Example 5, note how the contract's state of an endpoint evolve after every operation. The above contract forbids head-to-head deadlocks (messages cannot be sent on f), buffer overflow (only one message can be sent, thus allocating a buffer of size one never causes a buffer overflow), and message orphans (if the cell message were not received before closure, f would not be in the final state when the axiom of `close` is applied). However, the following example shows that circular waits are not ruled out by the contract in our proof system:

```
main(){
  {emp}
  (e1,f1):=open(C);
  (e2,f2):=open(C);
  foo(e1,f2) || foo(e2,f1);
  close(e1,f1);
  close(e2,f2)
  {emp}
}

foo(e,f) {
  {$e \overset{ep}{\mapsto}(C\{1\},-) * f \overset{ep}{\mapsto}(\overline{C}\{1\},)$}
  _ := receive(cell,f);
  send(cell,e,new());
  {$e \overset{ep}{\mapsto}(C\{2\},-) * f \overset{ep}{\mapsto}(\overline{C}\{2\},)$}
}
```

Worst, even with one channel, circular waits are possible (consider for instance swapping lines 12 and 14 in Example 5). Circular waits just cannot be ruled out by contracts[7]. But the good news is that unspecified receptions *can*: whenever a switch construct occurs, we can check that the scanned endpoint[8] is in a contract's state that guarantees that no message is missed by the listed cases[9]. The rule below formalizes that check:

$$\frac{\begin{array}{c}\forall q \xrightarrow{m} q' \ \in \ \mathit{transitions}(C), \quad m \in \{m_1, \ldots, m_n\} \\ \forall i = 1 \ldots n, \quad \{E \overset{ep}{\mapsto} (C\{q\}, -) * \varphi\} \ \vec{x}_i := \mathsf{receive}(m_i, E); p_i \ \{\psi\}\end{array}}{\{E \overset{ep}{\mapsto} (C\{q\}, -) * \varphi\} \ \mathsf{switch}\{\ldots \vec{x}_i := \mathsf{receive}(m_i, E) : p_i \ldots\} \ \{\psi\}}$$

Example 6 *Consider the annotations for the program of Example 3:*

```
send_and_dipose_list(x){
  $\{ls(x, nil)\}$
  local e,f;
  (e,f) := open(C);
  producer(x,e) || consumer(f)
  {emp}
}
```

```
producer(x,e){
  $\{ls(x, nil) * e \overset{ep}{\mapsto} (C\{1\}, -)\}$
  local t;
  while x != null {
    t := x->next;
    $\{x \mapsto t * ls(t, nil) * e \overset{ep}{\mapsto} (C\{1\}, -)\}$
    send(cell,e,x);
   $\{ls(t, nil) * e \overset{ep}{\mapsto} (C\{2\}, -)\}$
    x := t;
    receive(ack,e);
  }
  $\{e \overset{ep}{\mapsto} (C\{1\}, -)\}$
  send(close_me,e,e);
  {emp}
}
```

```
consumer(f){
 local y,e;
 while true {
   $\{f \overset{ep}{\mapsto} (\overline{C}\{1\}, -)\}$
   switch {
     y := receive(cell,f) : {
       $\{(x \mapsto -) * f \overset{ep}{\mapsto} (\overline{C}\{2\}, -)\}$
       dispose(y); send(ack,f);}
     e := receive(close_me,f) : {
       $\{f \overset{ep}{\mapsto} (\overline{C}\{3\}, e) * e \overset{ep}{\mapsto} (C\{3\}, f)\}$
       break }
   }
 }
 $\{f \overset{ep}{\mapsto} (\overline{C}\{3\}, e) * e \overset{ep}{\mapsto} (C\{3\}, f)\}$
 close(e,f);
 {emp}
}
```

Then the loop invariant of the consumer ensures that f is in contract's state 1 at the entry of the switch construct, thus by contract C either a `cell` *or* `close_me` *message can be received, but no other message, and thus no unspecified reception can occur.*

[7] One way of preventing circular waits is to forbbid one thread to own more than one endpoint at a time, which is arguably a bit drastic. Another, richer, solution is to introduce a hierarchy on channels, see Leino and Muller [58].

[8] Well, we did not prevent from scanning *several* endpoints in a same switch construct, but we will never meet this situation in this presentation; see Villard's thesis [59] for a possible treatment of switches on a group of endpoints.

[9] Our rule for the switch construct differs in its form from small axioms – it embeds some of the frame rule in it. Handling a more algebraic *external* choice $p \oplus p'$ would surely be cleaner, but we leave open the question of such a cleaner formalization.

The careful reader will have noticed that some ingredients are missing in our proof system to build a proof out of the annotations of Example 6. The first problem is the precondition of `close(e,f)` in consumer's code. We really need to know that e and f are peers of each other, but the side condition of the parallel rule that prevents variable sharing forces us to forget the peer information at the entry of functions `producer` and `consumer`. The fix is simple: the footprint of the `close_me` message must not only say that an endpoint is transferred, it should also precise that this endpoint is the peer of the endpoint that will receive the message. We achieve that with a special keyword dst allowed in message footprints only:

```
message cell(x) {x ↦ −}
message ack {emp}
message close_me(e) {e ↦ep (C{3}, dst)}
```

The second problem is a more severe one: the Hoare triple of `send(close_me,e,e)` (line 18) is not derivable from the send rule! We must fix this with a second rule to handle the specific case of an endpoint sent over itself. As a consequence, we need to update the rules of send and receive as follows:

$$\frac{q \xrightarrow{?m} q' \ \in\ \mathit{transitions}(C)}{\{E \overset{ep}{\mapsto}(C\{q\},F)\}\ \vec{x} := \mathrm{receive}(m,E)\ \{E \overset{ep}{\mapsto}(C\{q'\},F) * \varphi_m(\vec{x})[\mathsf{dst}\leftarrow E]\}}$$

$$\frac{q \xrightarrow{!m} q' \ \in\ \mathit{transitions}(C)}{\{E \overset{ep}{\mapsto}(C\{q\},F) * \varphi_m(\vec{E})[\mathsf{dst}\leftarrow F]\}\ \mathrm{send}(m,E,\vec{E})\ \{E \overset{ep}{\mapsto}(C\{q'\},F)\}}$$

$$\frac{q \xrightarrow{!m} q' \ \in\ \mathit{transitions}(C)}{\{E \overset{ep}{\mapsto}(C\{q\},F) * \big(E \overset{ep}{\mapsto}(C\{q'\},F) \mathrel{-\!\!*} \varphi_m(\vec{E})[\mathsf{dst}\leftarrow F]\big)\}\ \mathrm{send}(m,E,\vec{E})\ \{\mathsf{emp}\}}$$

Provided our semantics will validate the entailment

$$e \overset{ep}{\mapsto}(C\{q\},f) * f \overset{ep}{\mapsto}(C'\{q'\},e') \vDash e = e'$$

these new rules are enough for the examples we presented so far. But we may be a bit worried atfer missing such subtelties in our first trial: are these rules "complete" in some sense? An answer is "no", as we will see in Section 3.5.

3.2.3 Soundness

The first check we should have is that our proof system is at least sound, in the same sense as for Theorem 1.1. There are several ways of reaching such a goal. Probably the most attractive one would be to establish soundness by means of Rely-Guarantee Separation Logic. To do this, we would have to consider a particular implementation of queues in the heap, a particular implementation of open, send, receive, and close, a particular definition of a concurrent abstract predicate $E \overset{ep}{\mapsto} (C\{q\}, -)$, etc. This is surely possible, and it would probably be interesting if we were to prove a given implementation of Sing♯ channels, but it also has a lot of pitfalls, like the modeling of the switch construct, or the rule for sending an endpoint over itself. Out of that, there is also an important reason for *not* proving soundness by means of RGSep: we expect to eventually prove more than the soundness of our proof system. We indeed want to show that this proof system can be used to guarantee the *safety* of provable code, which includes absence of memory violations, races, memory leaks, and deadlocks. The soundness of RGSep would guarantee the absence of memory violations, but even inferring the absence of races from RGSep would require some extra work, not talking about memory leaks and deadlocks.

In addition to RGSep, we could consider proving the soundness of our proof system by means of abstract separation logic [12]; this is indeed the first proof of soundness that we developed for our proof system [60]. This semantics has some interesting aspects – for instance, it introduces some "tags" on cells that can be thought as an abstraction of the ownership information that exists in the internals of Sing♯. But it is arguably less concrete than Villard's one, due to the algebraic formulation needed by abstract separation logic. Villard's semantics also presents the advantage of modeling switch receives, whereas the first semantics did not distinguish internal and external choices. We choose to present here the second proof of soundness developed by Jules Villard in his phD thesis [59].

Villard defined an operational semantics that is based on an extension of the state model of Chapter 1. This state model contains on the one hand logical informations and on the other hand the information on the contents of the queues. Since a message may contain endpoints, and the state of an endpoint should describe the messages in its queues, it could be expected that our state model would have to cope with a certain circularity similar to the state models for locks in the heap [61, 62]. Although this circularity could be treated with domain theory, or Appel & al indirection's theory [63], Villard followed a different approach that does not show any circularity in the definition of the state model.

Let us now recall Villard's semantics. For presentation purpose, we adopt different names and notations (inspired by RGSep), and do not consider variables as resources. In addition to the sets Loc of cell locations, and Var of variables, we assume infinite sets $\mathsf{Endpoint} = \{\varepsilon, \dots\}$ of endpoint locations, $\mathsf{MsgTag} = \{m, a, b, \dots\}$ of message tags, Ctt of contract identifiers, and $\mathsf{CttState} = \{q, \dots\}$ of identifiers of contract's states.

The state model is formally defined on Figure 3.5. A state is a tuple $\sigma = (s, h, \boxed{h})$ of a stack s, a "local" heap h, and a "global" heap $\boxed{h}$. The local heap is partitioned into a cell heap and an endpoint heap. The endpoint heap contains the logical information about a given endpoint – namely its peer, its contract,

$$\begin{array}{lcll}
\mathsf{State} & \triangleq & \mathsf{Stack} \times \mathsf{Heap} \times \boxed{\mathsf{Heap}} & \text{(states)} \\
\mathsf{Stack} & \triangleq & \mathsf{Var} \to \mathsf{Val} & \text{(stacks)} \\
\mathsf{Heap} & \triangleq & \mathsf{CHeap} \times \mathsf{EHeap} & \text{(local heaps)} \\
\mathsf{CHeap} & \triangleq & \mathsf{Loc} \rightharpoonup_{\mathit{fin}} \mathsf{Val} \times \mathsf{Val} & \text{(cell heaps)} \\
\mathsf{EHeap} & \triangleq & \mathsf{Endpoint} \rightharpoonup_{\mathit{fin}} \mathsf{Endpoint} \times \mathsf{Ctt} \times \mathsf{CttState} & \text{(endpoint heaps)} \\
\boxed{\mathsf{Heap}} & \triangleq & \mathsf{Endpoint} \to \mathsf{Endpoint} \times \mathsf{Buffer} & \text{(global heaps)} \\
\mathsf{Buffer} & \triangleq & (\mathsf{MsgTag} \times \mathsf{Val}^* \times \mathsf{Heap})^* & \text{(buffers)}
\end{array}$$

where $\mathsf{Val} = \mathsf{Loc} + \mathsf{Endpoint} + \ldots$ and $\mathcal{E}^*$ denotes the set of finite sequences of elements of $\mathcal{E}$.

Figure 3.5: State Model

and its contracts' state. The content of the incoming buffer of an endpoint is not modeled in the local heap (which would create the aforementioned circularity), but in the global heap: $\boxed{h}$ associates to each endpoint a buffer buf, modeled as a finite sequence of messages. A message is then a tuple $(m, \vec{v}, h)$ of a message tag m, a tuple of values $\vec{v}$ of the same arity as the one of the message tag m, and a local heap h modeling the footprint of the message.

We adopt the expected composition $h \bullet h'$ of local heaps; this yields a composition of states $(s, h, \boxed{h}) \bullet (s', h', \boxed{h'})$ defined as $(s, h \bullet h', \boxed{h})$ when $h \bullet h'$ is defined, $s = s'$, and $\boxed{h} = \boxed{h'}$, undefined otherwise. We write $ftp(\boxed{h})$ for the set of local heaps that appear as footprints in the buffers of the global queues' state $\boxed{h}$. The semantics of formulas on logical states is the expected relation $s, h \vDash \varphi$; it is extended to states as $s, h, \boxed{h} \vDash \varphi$ if $s, h \vDash \varphi$. We assume a proof environment Γ that associates to each contract identifier a contract, to each contract's state identifier a contract's state, and to each message tag m a footprint formula φ_m.

Definition 3.2 (Consistent States) *A state $\sigma = (s, h, \boxed{h})$ is said* consistent *with respect to a proof environment Γ, $\Gamma \vdash \sigma$, if it satisfies the following conditions:*

Finiteness *the set of $\varepsilon \in \mathsf{Endpoint}$ such that $\boxed{h}(\varepsilon)$ is not the empty sequence is finite;*

Disjointness *the composition of local heaps $\mathit{flat}(\sigma) = h \bullet \mathop{\Large\bullet}\limits_{h' \in ftp(\boxed{h})} h'$ is defined;*

Duality *for all $\varepsilon_1, \varepsilon_2, \varepsilon_3$ such that $\boxed{h}(\varepsilon_1) = (\varepsilon_2, buf_1)$ and $\boxed{h}(\varepsilon_2) = (\varepsilon_3, buf_2)$, $\varepsilon_1 = \varepsilon_3$; if $\mathit{flat}(\sigma)(\varepsilon_1) = (\varepsilon_2', C, q)$, then $\varepsilon_2 = \varepsilon_2'$; if moreover $\mathit{flat}(\sigma)(\varepsilon_2) =$*

(ε_3, C', q'), *then* $\Gamma \vdash C' = \overline{C}$*;*

Footprints' correctness *for all* $\varepsilon \in \mathsf{Endpoint}$*, if*

$$\boxed{h}(\varepsilon) \quad = \quad (\ \varepsilon'\ ,\ (m_1, \vec{v_1}, h_1) \ldots (m_n, \vec{v_n}, h_n)\ ) \quad ,$$

then $s, h_i \vDash \varphi_{m_i}(\vec{v}_i)[\mathsf{dst}{\leftarrow}\varepsilon]$ *for all* $i = 1 \ldots n$*.*

Villard's operational semantics is a small-step semantics of a form similar to the one of Fig 1.2; it defines a binary relation:

$$\begin{array}{llll} p, \sigma \underset{\Gamma}{\leadsto} \mathit{result} \quad \text{where} \quad \mathit{result} & := & p', \sigma', & \text{(normal execution)} \\ & & \mathbf{OwnError}, & \text{(ownership violation)} \\ & & \mathbf{ProtoError} & \text{(contract violation)} \\ & & \mathbf{MsgError} & \text{(communication error)} \end{array}$$

Two things should be observed about this semantics:

- it is parametric in the environment Γ;
- it introduces two new error states **ProtoError** and **MsgError**.

The parametricity in the environment Γ can be explained as follows: when a message is sent, its footprint, as determined[10] by Γ, is transfered from the local heap to the global heap. As we already discussed it, it may be desirable to keep the ownership transfer "purely logical", and we may seek a semantics that is independent of the proof environment Γ. We later solve this issue. Figure 3.6 shows the formal definition of the operational semantics of send and receive in the case of a safe execution. The semantics of send and receive introduce an intermediate instruction $\text{skip}_{\varepsilon\lambda}$ whose role is to modify the contract's state of the endpoint ε, following a transition labeled with λ. The rest of the definition of the semantics of send and receive is in charge of the ownership transfer.

Let us now comment the two new error states: **ProtoError** is modeling a violation of a communication contract, and **MsgError** is modeling a communication error. Contracts violations occur either at send/receive when the message tag is not allowed by the contract, or at channel's closure when the two endpoints are not in a same final state, or in a switch construct if the listed cases are not exhaustive with respect to what is prescribed by the contract. Communication errors occur either in a switch construct when the tag of the first-out message is not listed (unspecified reception), or at a channel's closure if a message is still in one of the queues (message orphan). The rules generating errors are listed on Figure 3.7.

Theorem 3.1 (Soundness [59]) *Assume* $\{\varphi\}\ p\ \{\psi\}$ *has a proof. Then, for all* $\sigma \vDash \varphi$*, the following holds:*

[10] We assume that formulas of messages' footprints are *precise*, otherwise falling into Reynolds' paradox, see [59]; similarly, we assume that contracts are deterministic automata.

$$\frac{\boxed{h}(\varepsilon) = (\varepsilon', -) \qquad q_0 = \mathsf{init}(C) \qquad \begin{array}{c} \varepsilon, \varepsilon' \text{ not in } \mathit{dom}(\mathit{flat}(s, h, \boxed{h})) \\ s' = [s \mid \mathrm{e} \mapsto \varepsilon), \mathrm{f} \mapsto \varepsilon'] \end{array} \qquad h' = \varepsilon \mapsto (\varepsilon', C, q_0)), \varepsilon' \mapsto (\varepsilon, \overline{C}, q_0)}{(\mathrm{e}, \mathrm{f}) \ = \mathrm{open}(\mathrm{C}), (s, h, \boxed{h}) \underset{\Gamma}{\leadsto} \mathrm{skip}, ((s', h \bullet h', \boxed{h})}$$

$$\frac{h' = \{\varepsilon_1 \mapsto (\varepsilon_2, -), \varepsilon_2 \mapsto (\varepsilon_1, -)\} \qquad [\![E_i]\!]_s = \varepsilon_i}{\mathrm{close}\,(E_1,E_2), (s, h \bullet h', \boxed{h}) \underset{\Gamma}{\leadsto} \mathrm{skip}, (s, h, \boxed{h})}$$

$$\frac{\begin{array}{c} [\![E]\!]_s = \varepsilon \qquad [\![\vec{F}]\!]_s = \vec{v} \qquad \boxed{h}(\varepsilon) = (\varepsilon', -) \qquad \boxed{h}(\varepsilon') = (\varepsilon, \mathsf{buf}) \qquad h_m \vDash \varphi_m(\vec{F})[\mathsf{dst}{\leftarrow}\varepsilon'] \\ \mathrm{skip}_{\varepsilon!m}, (s, h, \boxed{h}) \underset{\Gamma}{\leadsto} \mathrm{skip}, (s, h' \bullet h_m, \boxed{h}) \qquad \boxed{h'} = [\boxed{h} \mid \mathit{buffer}(\varepsilon'){\leftarrow}\mathsf{buf}.(m, \vec{v}, h_m)] \end{array}}{\mathrm{send}(\mathrm{m},E,\vec{F}), (s, h, \boxed{h}) \underset{\Gamma}{\leadsto} \mathrm{skip}, (s, h', \boxed{h'})}$$

$$\frac{[\![E]\!]_s = \varepsilon \qquad \boxed{h}(\varepsilon) = (\varepsilon', (m, \vec{v}, h_m).\mathsf{buf}) \qquad s' = [s \mid \vec{x} \mapsto \vec{v}] \qquad \boxed{h'} = [\boxed{h} \mid \mathit{buffer}(\varepsilon){\leftarrow}\mathsf{buf}]}{\vec{x}{=}\mathrm{receive}(\mathrm{m},E), (s, h, \boxed{h}) \underset{\Gamma}{\leadsto} \mathrm{skip}_{\varepsilon?m}, (s', h \bullet h_m, \boxed{h'})}$$

$$\frac{h(\varepsilon) = (\varepsilon', C, q) \qquad q \xrightarrow{\lambda} q' \in \mathit{transitions}(C) \qquad h' = [h \mid \mathit{control}(\varepsilon){\leftarrow}q']}{\mathrm{skip}_{\varepsilon\lambda}, (s, h, \boxed{h}) \underset{\Gamma}{\leadsto} \mathrm{skip}, (s, h', \boxed{h})}$$

$$\frac{\vec{x_i}{=}\mathrm{receive}(m_i, E_i) \underset{\Gamma}{\leadsto} \mathrm{skip}, \sigma'}{\mathrm{switch}\{\ldots, \mathrm{case}\ \vec{x_i}{=}\mathrm{receive}(m_i, E_i) : \{p_i\}, \ldots\}, \sigma \underset{\Gamma}{\leadsto} p_i, \sigma'}$$

Figure 3.6: Operational semantics: safe executions.

1. $p, \sigma \underset{\Gamma}{\not\leadsto}^* \textbf{OwnError}$

2. $p, \sigma \underset{\Gamma}{\not\leadsto}^* \textbf{ProtoError}$

3. *if* $p, \sigma \underset{\Gamma}{\leadsto}^* \mathrm{skip}, \sigma'$, *then* $\sigma' \vDash \psi$.

Remark Proving the soundness of the parallel rule naturally enforces to extend the operational semantics with a rule modeling interferences (see Villard [59]). For presentation purpose, we omit the rule of interferences for now, as it requires some theory on contracts. The above soundness result is still true without this rule – the soundness for the semantics with interferences implies the soundness for the semantics without interferences.

The reader might have expected that the proof system would rule out the **MsgError**. Nevertheless, this is not the case in general:

Example 7 *Consider the following program*

```
producer(e){
  local x;
  x:=new();
  send(cell,e,x);
  send(close_me,e,e)
}
```

$$\frac{\mathit{buffer}(\boxed{h})(\llbracket E_i \rrbracket_s) \text{ not empty for some } i \in \{1,2\}}{\mathsf{close}\,(E_1, E_2), (s, h, \boxed{h}) \underset{\Gamma}{\rightsquigarrow} \textbf{MsgError}}$$

$$\frac{h(\llbracket E_i \rrbracket_s) = (-, C, q_i) \qquad q_1 \neq q_2 \text{ or } q_1 \notin \mathit{finals}(C)}{\mathsf{close}\,(E_1, E_2), (s, h, \boxed{h}) \underset{\Gamma}{\rightsquigarrow} \textbf{ProtoError}}$$

$$\frac{\boxed{h}(\llbracket E_i \rrbracket_s) \text{ undef} \qquad \text{or} \qquad \boxed{h}(\llbracket E_i \rrbracket_s) \neq (\llbracket E_{2-i} \rrbracket_s, -, -,) \qquad \text{for some } i \in \{1,2\}}{\mathsf{close}\,(E_1, E_2), (s, h, \boxed{h}) \underset{\Gamma}{\rightsquigarrow} \textbf{OwnError}}$$

$$\frac{\boxed{h}(\varepsilon) = (\varepsilon', -) \qquad \begin{array}{c} \llbracket E \rrbracket_s = \varepsilon \\ \mathsf{skip}_{\varepsilon!m}, (s, h, \boxed{h}) \underset{\Gamma}{\rightsquigarrow} \mathsf{skip}, (s, h', \boxed{h}) \end{array} \qquad \forall h_m \preceq h', s, h_m \not\models \varphi_m(\vec{F})[\mathsf{dst} \leftarrow \varepsilon']}{\mathsf{send}(m, E, \vec{F}), (s, h, \boxed{h}) \underset{\Gamma}{\rightsquigarrow} \textbf{OwnError}}$$

$$\frac{h(\varepsilon) = (-, C, q) \qquad \nexists q'.\ q \xrightarrow{\lambda} q' \in \mathit{transitions}(C)}{\mathsf{skip}_{\varepsilon\lambda}, (s, h, \boxed{h}) \underset{\Gamma}{\rightsquigarrow} \textbf{ProtoError}} \qquad \frac{h(\varepsilon) \text{ undefined}}{\mathsf{skip}_{\varepsilon\lambda}, (s, h, \boxed{h}) \underset{\Gamma}{\rightsquigarrow} \textbf{OwnError}}$$

$$\frac{\boxed{h}(\llbracket E \rrbracket_s) = (-, (m, -, -).\mathsf{buf}) \qquad m \notin \{m_1, \ldots, m_n\}}{\mathsf{switch}\{\ldots, \vec{x_i} = \mathsf{receive}(m_i, E) : p_i, \ldots\}, \sigma \underset{\Gamma}{\rightsquigarrow} \textbf{MsgError}}$$

$$\frac{h(\llbracket E \rrbracket_s) \text{ undefined}}{\mathsf{switch}\{\ldots, \vec{x_i} = \mathsf{receive}(m_i, E) : p_i, \ldots\}, \sigma \underset{\Gamma}{\rightsquigarrow} \textbf{OwnError}}$$

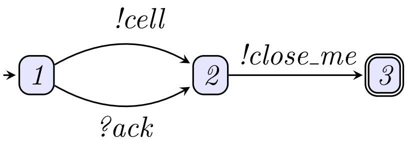
$$\frac{h(\llbracket E \rrbracket_s) = (-, C, q) \qquad \exists m \notin \{m_1, \ldots, m_n\}.\ \ q \xrightarrow{?m} q' \in \mathit{transitions}(C)}{\mathsf{switch}\{\ldots, \vec{x_i} = \mathsf{receive}(m_i, E_i) : p_i, \ldots\}, \sigma \underset{\Gamma}{\rightsquigarrow} \textbf{ProtoError}}$$

Figure 3.7: Operational semantics: errors.

```

consumer(f){
  local e;
  send(ack,f);
  switch {
    e:= receive(close_me,f) :
         close(e,f);
}}
```

This program has an unspecified reception error at runtime – the first cell message is not caught by the consumer. But this program has a proof! To see how it can be proved, consider the following contract:

```
contract C {
    initial state 1 {
        ! cell  →  2;
        ?ack  →  2;
    }
    state 2 {
        !close_me  →  3;
    };
    final state 3 {};
};
```



Note that this program also has a head-to-head deadlock (see Section 3.1.4). Although head-to-head deadlocks may not be considered as real ”deadlocks” depending on the implementation of channels, it must be considered as a real “error” in the contract’s approach. To better explain this point, we go now into the details of contracts.

3.3 A Theory of Sing♯ Contracts

The first thing we need is a proper semantics for contracts. It could be done in terms of a process algebra, like it is the tradition for session types, but we adopt here a more direct approach and give a semantics in terms of communicating finite state machines.

We assume a contract $C = (Q, \Sigma, \Delta, q_{init}, F)$. The set of *configurations* of C is defined as

$$Conf(C) \triangleq Q \times Q \times \Sigma^* \times \Sigma^*.$$

Intuitively, a configuration $\gamma = (q_0, q_1, w_0, w_1)$ corresponds to a situation where the endpoint ε_0 ruled by C is in contract’s state q_0, its peer ε_1 is in q_1, the incoming queue of ε_0 contains a sequence of message tags w_0, and the incoming queue of ε_1 contains a sequence of message tags w_1. Contracts’ configurations permit to introduce an operational semantics for contracts. We say that a configuration $\gamma = (q_0, q_1, w_0, w_1)$ may evolve to a configuration $\gamma' = (q'_0, q'_1, w'_0, w'_1)$, $\gamma{\rightarrow}\gamma'$, if γ results in γ' after one endpoint has either sent or received a message in a contract obedient way. This is expressed formally as follows (note that the contract C_0 of ε_0 is C, whereas the contract C_1 of ε_1 is $\overline{C}$): $\gamma{\rightarrow}\gamma'$ if and only if there is some $i \in \{0, 1\}$, and some $m \in \Sigma$ such that:

- either $q_i \xrightarrow{!m} q'_i \in \Delta_i$, and the following holds:
  - $w'_i = w_i$
  - $q'_{1-i} = q_{1-i}$
  - $w'_{1-i} = w_{1-i}.m$;
- or $q_i \xrightarrow{?m} q'_i \in \Delta_i$, and the following holds:

- $m.w'_i = w_i$
- $q'_{1-i} = q_{1-i}$
- $w'_{1-i} = w_{1-i}$.

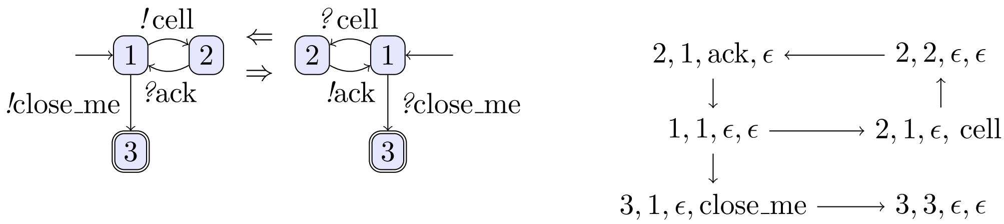
Definition 3.3 (Semantics of Contracts) *The set*

$$[\![C]\!] \triangleq \{\ \gamma \in Conf(C)\ \mid\ (q_{init}, q_{init}, \epsilon, \epsilon) \rightarrow^* \gamma\ \}$$

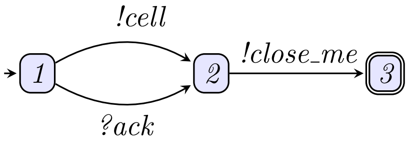
is called the set of reachable configurations *of* C*.*

Example 8 *The set of reachable configurations of the contract C of the program sending a list cell by cell (Example 3) is as follows:*

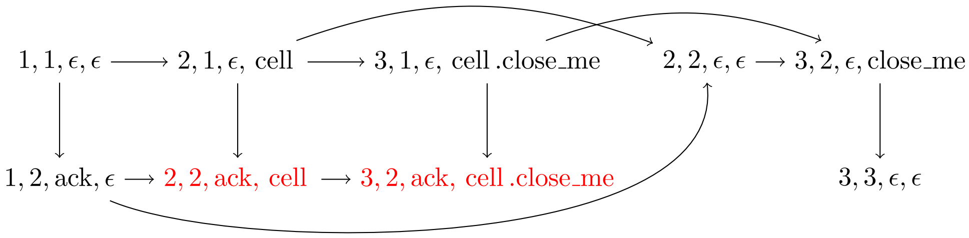
$$[\![C]\!] \ = \ \{ \ \begin{array}{lll} (1,1,\epsilon,\epsilon)\ , & (2,1,\epsilon,\text{ cell})\ , & (2,2,\epsilon,\epsilon)\ , \\ (2,1,\text{ack},\epsilon)\ , & (3,1,\epsilon,\text{close\_me})\ , & (3,3,\epsilon,\epsilon)\}. \end{array}$$



Example 9 *Similarly, the set of reachable configurations of the contract*



of Example 7 is depicted below:



The unspecified reception that occured in Example 7 can be directly observed on the semantics of its contract: it corresponds to the configurations $(2, 2, \text{ack}, \text{ cell})$ and $(3, 2, \text{ack}, \text{ cell .close\_me})$: in these configurations, the endpoint ruled by $\overline{C}$ is in state 2, and waits for the message `close_me`, but this one is blocked by the message `cell`. More generally, several properties of the programs can be directly expressed with the semantics of C.

Definition 3.4 (Contract's Safety) *A contract C is said* safe *if it satisfies the following conditions:*

Buffer's Boundedness $[\![C]\!]$ *is finite.*

Absence of Unspecified Receptions *For all* $\gamma = (q_0, q_1, w_0, w_1) \in [\![C]\!]$, *for all* $i \in \{0,1\}$, *if* $\gamma \not\rightarrow$ *and* $w_i = m.w'$, *then there is* $q_i \xrightarrow{?m} q'_i \in transitions(C_i)$ *for some* q'_i.

Absence of Message Orphans *For all* $\gamma = (q_0, q_1, w_0, w_1) \in [\![C]\!]$, *if* $q_0 = q_1 \in finals(C)$, *then* $w_0 = w_1 = \epsilon$.

Absence of Head-to-Head Deadlocks *For all* $\gamma = (q_0, q_1, w_0, w_1) \in [\![C]\!]$, *if* $q_0 = q_1 \in finals(C)$, *then* $w_i = \epsilon$ *for some* $i \in \{0,1\}$.

Contracts that are not safe should not be used for proving programs: it is always possible to reproduce an error that occurs in a contract inside a program typed by this contract; conversely, we will later see that, due to subject reduction (Proposition 3.2), the absence of a kind of error in a contract implies the absence of this kind of errors in the programs it types.

Then the question is: how do we recognize that a contract is safe? The notion of safe contract that has now[11] been adopted by Sing♯ compiler is somehow the following:

Definition 3.5 (Admissible Contracts) *A contract C is said admissible if it satisfies the following conditions:*

Determinism *if* $q \xrightarrow{\lambda} q_1$ *and* $q \xrightarrow{\lambda} q_2$, *then* $q_1 = q_2$.

Uniform Choice *if* $q \xrightarrow{\lambda_1} q_1$ *and* $q \xrightarrow{\lambda_2} q_2$, *then either* $\lambda_1 = !m_1$ *and* $\lambda_2 = !m_2$, *or* $\lambda_1 = ?m_1$ *and* $\lambda_2 = ?m_2$.

Boundedness *for all cyclic path* $q = q_0 \xrightarrow{\lambda_0} q_1 \xrightarrow{\lambda_1} \dots \xrightarrow{\lambda_n} q_n = q$, *there is* $i, j \in \{0, \dots n\}$ *such that* $\lambda_i = !m_i$ *and* $\lambda_j = ?m_j$.

It can be shown that this notion of admissible contract is sound, and even complete in our sense of safety. Let $det(C)$ denote the contract C' obtained from C by determinisation of the underlying finite-state automaton through the standard powerset construction.

Proposition 3.1 ([5]) *A contract C is safe if and only if $det(C)$ is admissible.*

[11] In a first version of Sing♯, the condition on uniform choice of Definition 3.5 was missing. The need of this condition was pointed out by Stengel and Bultan [64]).

The “only if” part of this result may be familiar to the communities of infinite-state model-checking and session types[12]. The reverse direction may be a bit surprising, and should be little commented. For it to be true, it is crucial that we restrict our attention to contracts without head-to-head deadlocks, aka *half-duplex*[13] contracts. The conditions taken in Definition 3.5 may also be justified by considering contracts as safe when they are either realizable [64], or synchronizable [68], but then only one implication of Proposition 3.1 can be proved.

To conclude, we should precise the relationship between contracts and our toy programming language. Two things should be explained: first, how the semantics of a contract over-approximates, in a certain sense, the one of the programs it types – this property is often called subject reduction in session types – and second, how the absence of errors in the contract's semantics ensures the absence of communication errors in the programs it types.

Consider a program's state $\sigma = (s, h, \boxed{h})$ and an endpoint ε such that $\mathit{flat}(s, h, \boxed{h})(\varepsilon)$ is defined, and assume $\mathit{flat}(s, h, \boxed{h})(\varepsilon) = (\varepsilon', C, q)$. We can construct the set $\mathsf{conf}(\sigma, \varepsilon)$ of configurations of C that reflect the contracts' states of ε and ε' and their queue's content according to σ: given a buffer $\mathsf{buf} = (m_1, \vec{v}_1, h_1) \ldots (m_n, \vec{v}_n, h_n)$ we write $w(\mathsf{buf})$ for the word $m_1 \ldots m_n$; then $\mathsf{conf}(\sigma, \varepsilon)$ is the set of configurations $(q, q', w(\mathsf{buf}), w(\mathsf{buf'}))$ such that

- buf and $\mathsf{buf'}$ are the incoming buffers of ε and ε', *i.e.* $\boxed{h}(\varepsilon) = (\varepsilon', \mathsf{buf})$ and $\boxed{h}(\varepsilon') = (\varepsilon, \mathsf{buf'})$;
- if $\mathit{flat}(s, h, \boxed{h})(\varepsilon')$ is defined, then it is equal to $(\varepsilon, \overline{C}, q')$, otherwise q' is arbitrary.

Note that there are only two forms of set $\mathsf{conf}(\sigma, \varepsilon)$: either it is a singleton, or it is a set of the form $\bigcup_{q' \in Q} \{(q, q', w, w'\}$ for some q, w and w'.

Definition 3.6 (Well-Typed State) *A consistent state σ is said* well-typed *if for all ε, C such that $\mathit{flat}(\sigma) \vDash \varepsilon \overset{ep}{\mapsto} (C\{-\}, -) * \top$, it holds that*

$$\mathsf{conf}(\sigma, \varepsilon) \cap [\![C]\!] \neq \emptyset$$

In other words, a program state σ is well-typed if the informations it gives on the contracts and contract's states of the endpoint on one side, and on the queue's content on the other side, can be seen as reachable configuration of the

[12] The analysis of dual communicating finite state machines goes back to the work of Gouda, Manning and Yue [65], and their results were unconsciously rediscovered – and extended – in the context of asynchronous session types quite recently [66].

[13] Half-duplex communications have been shown to be much more tractable to analysis than full-duplex ones; in particular, $[\![C]\!]$ can be coded as a regular language, and all properties listed above can be checked in polynomial-time, even if the two machines are not dual one of each other [67], and even under some exotic semantics of buffers that are not FIFO [5].

contract. Another possible definition, that would avoid the CFSM semantics, would be that a state σ is well-typed if it can be reached by a contract-obedient program.

Proposition 3.2 (Subject Reduction) *For all proof environment Γ, for all programs p, p' and for all states σ, σ', if the following holds:*

1. $p, \sigma \underset{\Gamma}{\not\rightsquigarrow}^{*} \mathbf{ProtoError}$*, and*
2. $p, \sigma \underset{\Gamma}{\rightsquigarrow} p', \sigma'$*, and*
3. σ *is well-typed,*

then σ' is well-typed.

We can now present the interference rule of the operational semantics we did not present with the other rules – recall that this rule is needed in the proof of soundness of the parallel rule. Interferences should model changes that occur in the global heap[14] due to other threads' steps. We may over-approximate these interferences as follows:

$$\frac{s, h, \boxed{h'} \text{ is consistent and well-typed}}{p, (s, h, \boxed{h}) \underset{\Gamma}{\rightsquigarrow} p, (s, h, \boxed{h'})}$$

The reader maybe observed that subject reductions holds even if the contracts are not admissible. Contract safety shows up instead with this soundness result:

Proposition 3.3 (Soundness of Contracts) *For all proof environment Γ, for all programs p, p' and for all states σ, σ', if the following holds:*

1. $p, \sigma \underset{\Gamma}{\not\rightsquigarrow}^{*} \mathbf{ProtoError}$*,*
2. σ *is well-typed, and*
3. Γ *maps all contract identifiers to admissible contracts,*

then $p, \sigma \underset{\Gamma}{\not\rightsquigarrow}^{*} \mathbf{MsgError}$*.*

[14] Interferences over global variables could be considered as well, but they must be defined with some care to keep the soundness of the rule of auxiliary variables. Since we may prove the soundness of the parallel rule without considering interferences on global variables, we simply skip them.

3.4 Memory Leaks and Communication Errors

We informally argued that our proof system would forbbid not only memory violations and races, but also memory leaks, some forms of deadlocks (namely unspecified receptions, head-to-head deadlocks, and buffer overflows[15]), and message orphans.

We investigate how to formalize this claim. Three things must be defined:

- a "realistic" operational semantics that do not depend on the proof environment Γ (see discussion in Section 3.1.2),
- a formal notion of program safety,
- a formal notion of a proof of safety.

First, we adopt for our "realistic" operational semantics almost the same as the one we presented, except for two things: we do not want to keep the information on contracts and ownership at runtime, and we do not want interferences.[16] We define Γ_0 as the proof environment that maps all message tags m to the footprint formula $\varphi_m \triangleq \mathsf{emp}$, and all contract's identifiers to the universal contract:

```
contract universal {
  initial final state s {
   !m₁→s; ?m₁→s; ...; !mₙ→s; ?mₙ→s;
  }
}
```

Then we define our realistic semantics as the one parametrized by Γ_0, and we simply write $\rightsquigarrow$ when we mean that $\underset{\Gamma_0}{\rightsquigarrow}$ is derivable without using the interference rule. Note that $p, \sigma \rightsquigarrow$ **ProtoError** is impossible by definition of the universal contract.

Second, we define program safety as the absence of errors *and* an extra condition capturing the absence of memory leaks for terminating[17] programs. We call *support* of a global heap $\boxed{h}$ the set of endpoints such that $\boxed{h}(\varepsilon) = (-, \mathsf{buf})$ for some non-empty sequence buf. We say that a state $\sigma = (s, h, \boxed{h})$ is *unallocated* if $dom(h) = support(\boxed{h}) = \emptyset$.

Definition 3.7 (Program Safety) *A program p is said* safe *if for all unallocated state σ:*

[15] To ease the presentation, we sticked to Villard's semantics and did not model head-to-head deadlocks and buffer overflows in the operational semantics. We let the reader convince himself that modeling them as **MsgError** would preserve Proposition 3.3 and would let us formally show our claim.

[16] Just because interferences may cause memory leaks.

[17] It would be nice to formalize the absence of memory leaks for non-terminating programs as well, but we expect that it would require some extra machinery.

1. *$p, \sigma \not\rightsquigarrow^* \textbf{OwnError}$;*

2. *$p, \sigma \not\rightsquigarrow^* \textbf{MsgError}$;*

3. *if $p, \sigma \rightsquigarrow^* \mathsf{skip}, \sigma'$, then σ' is unallocated.*

The third point is the most subtle one: we may think that a program p is provable safe if there is a proof of $\{\mathsf{emp}\}\ p\ \{\mathsf{emp}\}$ in a proof environment Γ mapping contracts identifiers to admissible contracts. But this is not the case in general.

Example 10 *The program* `mem_leak_generator` *of Figure 3.3 can be proved as follows:*

```
message cell(x)    $\{x \mapsto -\}$
message endpoint(f) $\{\exists e. e \overset{ep}{\mapsto} (C\{3\}, \mathsf{dst}) * f \overset{ep}{\mapsto} (\overline{C}\{2\}, e)\}$
contract C {
  initial state 1 {!cell->2;};
  state 2 {!endpoint->3;};
  final state 3{}
}
mem_leak_generator(){
  $\{\mathsf{emp}\}$
  local x,e,f;
  x:=new();
  (e,f) = open(C);
  $\{(x \mapsto -) * e \overset{ep}{\mapsto} (C\{1\}, f) * f \overset{ep}{\mapsto} (\overline{C}\{1\}, e)\}$
  send(cell,e,x);
  send(endpoint,e,f);
  $\{\mathsf{emp}\}$
}
```

Remember how Sing♯ associates to each cell and endpoint a process that owns it. Villard's operational semantics slightly differs from this approach, in that cells and endpoints are owned either by a process, or *a message* – in other words, our proof system is not compatible with the transitivity rule we introduced in Section 3.1.2. The part of the memory hold by messages is likely to be invisible in our proof system when it forms a cycle of ownership – a cycle of ownership happens in a situation like "a message $m_1(\vec{v}_1, h_1)$ owns an endpoint ε_2 that owns a message $m_2(\vec{v}_2, h_2)$ that owns ... that owns the endpoint ε_1 that owns m_1". To prove the absence of memory leaks, we thus need to forbbid such cycles of ownership. More precisely, we should only forbid the cycles that will be invisible – some of them are visible, namely the ones for which at least one peer of one of the endpoints involved in the cycle is not owned by the cycle.

Definition 3.8 (Admissible proof environment) *We say that a proof environment Γ is admissible if*

- *Γ maps all contract identifiers to admissible contracts*

- Γ *maps all message tags to precise footprint formulas*
- Γ *does not allow invisible leaks,* i.e *there is no state* $\sigma = (s, h, \boxed{h})$ *such that:*
  1. σ *is consistent and well-typed with respect to* Γ
  2. $dom(h) = \emptyset$
  3. σ *is allocated,* i.e *support*$(\boxed{h}) \neq \emptyset$.
  4. *for all* $\varepsilon \in support(\boxed{h})$, *flat*$(\sigma)(\varepsilon)$ *and flat*$(\sigma)(\varepsilon')$ *are defined, where* ε' *is such that flat*$(\sigma)(\varepsilon) = (\varepsilon', -, -)$.

Several other ways of formalizing the absence of invisible leaks have been proposed, either in logical terms, following the approach of Gotsman & al for locks in the heap [61] (see also Villard's phD thesis [59]), or in terms of well-foundedness, an approach followed by Bono & al for a session-typed process algebraic model of Sing♯ [69].

Theorem 3.2 (Safety) *For all program p, for all admissible proof environment* Γ, *if* $\{\mathsf{emp}\}$ p $\{\mathsf{emp}\}$ *is provable then p is safe.*

In practice, Definition 3.8 is too abstract to be used directly. We now formalize the fact that restricting to shallow ownership transfers guarantees the absence of ownership cycles.

Definition 3.9 (Shallow Ownership Transfer) *We say that a proof environment* Γ *ensures (half-duplex) shallow ownership transfers if* Γ *maps contract identifiers to admissible contracts, and, for all footprint formula* φ_m *associated to a message identifier m, for all state* $(s, h, \boxed{h})$, *for all* ε, *if*

- $s, h \vDash \varphi_m$, *and*
- $h(\varepsilon) = (\varepsilon', C, q)$

then q is a send state in C, i.e. *there are no* m', q' *such that* $q \xrightarrow{?m'} q' \in transitions(C)$.

Proposition 3.4 *If a proof environment* Γ *ensures shallow ownership transfer, then it does not allow invisible leaks.*

3.5 Determinism and Beyond

Provable programs might have more properties of interest than the ones we considered so far. In this section, we investigate the "confluence" of computations. It can be observed that our form of race-free message-passing seems to induce more than race-freedom: since at most one thread has the right to receive a given sent message, it is difficult to figure out how two threads could compete for the access to a resource, like it is the case with other synchronisation primitives. This form of concurrency has been called either "deterministic" [70] or "pipe-lined" [57] concurrency, but with the exception of Sassone & al [71], nobody really tried to formalize how this notion relates to selfishness. These works raise several interesting questions: are provable programs deterministic in some sense? Or can they encode locks and other synchronization primitives? And if our framework does not allow to encode standard synchronisation primitives, how can it be relaxed?

Before we formalize the confluence property, let us observe that our treatment of allocation makes our programs intrinsically non-deterministic. For instance, the internal choice $p + q$ can be encoded as:

```
choice(p,q){
  local x,y;
  x:=new(); free(x);
  y:=new(); free(y);
  if x=y then p else q;
}
```

There are several ways of answering this issue[18], we will choose a rather simple one and just assume that the programs we consider do not use allocation's instructions `new` and `open`.

Proposition 3.5 (Strong confluence) *Let p, σ be such that p is allocation-free and $p, \sigma \not\rightsquigarrow$* **OwnError**. *If $p, \sigma \rightsquigarrow p_1, \sigma_1$ and $p, \sigma \rightsquigarrow p_2, \sigma_2$, then there are p', σ' such that $p_1, \sigma_1 \rightsquigarrow p', \sigma'$ and $p_2, \sigma_2 \rightsquigarrow p', \sigma'$.*

$$\begin{tikzcd}
 & p, \sigma \arrow[dl, rightsquigarrow] \arrow[dr, rightsquigarrow] & \\
p_1, \sigma_1 \arrow[dr, rightsquigarrow, dotted] & & p_2, \sigma_2 \arrow[dl, rightsquigarrow, dotted] \\
 & p', \sigma' &
\end{tikzcd}$$

It is pretty clear that strong confluent programs, when they terminate for one scheduling, terminates for all scheduling, and always in the same final state. To prove Proposition 3.5, we would have liked to rely on abstract separation logic [12]. We indeed observed that it is possible to restrict the framework of local functions so as to ensure that provable programs are confluent. Assume that every atomic command c has a denotational semantics of the form $[\![c]\!]$: $\mathsf{State} \rightarrow P(\mathsf{State})^{\top}$, where State is a separation algebra, and $[\![c]\!]$ is such that:

(1) $\forall \sigma_1, \sigma_2 \in \Sigma, \quad f(\sigma_1 \bullet \sigma_2) \sqsubseteq f(\sigma_1) * \{\sigma_2\}$ (locality [12])

(2) $\forall \sigma, \sigma', \sigma_F \Sigma, \quad \sigma' \in f(\sigma)$ and $\sigma \perp \sigma_F$ implies $\sigma' \perp \sigma_F$ (non-expansion)

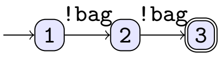
[18] One answer would be to consider each allocated cell as "fresh", and to consider states as nominal sets, which is the approach adopted for instance by Yang and Birkedal [72] (following Benton and Leperchey [73]).

Then any selfish program is strongly confluent. Note that the semantics of `x:=new()` in abstract separation logic is a local, but expanding function.

Despite this interesting framework, the only proof we have of Proposition 3.5 is a direct proof. The first semantics we introduced [60] and that was based on abstract separation logic is based on local but expanding function, so we cannot really use this framework. Villard's semantics seems also difficult to cast into the framework of abstract separation logic[19]. Maybe we did not find the right approach, or maybe there is some intrinsic difficulty to fit in this framework.

Proposition 3.5 shows the impossibility to encode locks or non-deterministic choice by provable functions using channels. Let us now explain what we should add to our proof system to handle them. Consider the following protocol implementing a non-deterministic internal choice between Paul (p_1) and Peggy (p_2): Paul and Peggy are in a dark room sitting around a table. On the table, there is a bag that initially contains a token. Paul scrabbles for the bag, and once he found it, he opens it. If the token is there, Paul takes it out, put the bag back on the table, and starts executing. If the token is not in the bag anymore, Paul has to stop. Peggy does exactly the same on her side. In our toy programming language, this encoding of the internal choice $p+q$ goes like the following program:

```
contract C {
```



```
}

choice($p_1$,$p_2$){
  (e,f):=open(C);
  send(bag,e,1);
  grop($p_1$) || grop($p_2$)
}
grop($p$){
  local i;
  i:=receive(bag,f);
  if (i=1) then {
    send(bag,e,0);
    $p$
  }
  else close(e,f);
}
```

The problem for proving this program is that the rule for receptions forces both Paul and Peggy to own f, which contradicts the parallel rule. One solution[20] could be to use fractional permissions for sharing endpoints, and consider that Paul (resp. Peggy) has half of the ownership of f. Then, the rule for send and receive actions on shared endpoints should assert that these actions are permitted even if only a fraction of the endpoint is owned, but then it should not change the contract's state (an action that changes the contract's state is like a "write" on a shared cell, it must be forbidden as it breaks the consistency of the local views of the contract's state of the endpoint). But even with such a support for permissions, it would not be possible to prove the above program,

[19] In the case of a concurrent receive on ε and send on ε's peer, it is necessary to split the incoming queue of ε, which complicates the definition of the separation algebra.

[20] A quite similar solution exists in session types, called*unrestricted qualified* types [74]. To the best of our understanding, unrestricted qualified types could be represented by fractionnal permissions, but not the other way around (because when a channel runs in unrestricted mode, it remains unrestricted forever).

because the actions on endpoints e and f precisely change the contract's state.

A different approach would be to consider that e and f are "owned by the bag" in the above example: when the bag is first put on the table, the ownership of e and f would be lost, so when they start neither Paul nor Peggy owns f. Later, when Paul (resp. Peggy) catches the bag, he also gets the right to open it, *i.e.* f. In other words, we may admit that the ownership of an endpoint ε could be granted *a posteriori* by the success of a reception on this endpoint ε. Formalizing this form of reasoning gives us the following rule:

$$\frac{q \xrightarrow{!m} q' \ \in \ \mathit{transitions}(C)}{\{\mathsf{emp}\}\ \vec{x} := \mathrm{receive}(m, E)\ \{E \overset{ep}{\mapsto} (C\{q'\}, F) * \big(E \overset{ep}{\mapsto} (C\{q\}, F) \mathrel{-\!\!*} \varphi_m(\vec{x})[\mathsf{dst}{\leftarrow}F]\big)\}}$$

which is a symmetric of the send rule used for handling the `close_me` messages.

Example 11 *Let us apply this new to derive a proof of the encoding of internal choice.*

```
1  message bag(i) {$e \overset{ep}{\mapsto} (C\{3-i\}, f) * f \overset{ep}{\mapsto} (\overline{C}\{2-i\}, e)$}
2  choice($p_1$,$p_2$){
3    {emp}
4    (e,f):=open(C);
5    {$e \overset{ep}{\mapsto} (C\{1\}, f) * f \overset{ep}{\mapsto} (\overline{C}\{1\}, e)$}
6    send(bag,e,1);
7    {emp}
8    grop($p_1$) || grop($p_2$)
9    {emp}
10 }
11 grop($p$){
12   {emp}
13   local i;
14   i:=receive(bag,f);
15   {$e \overset{ep}{\mapsto} (C\{3-i\}, f) * f \overset{ep}{\mapsto} (\overline{C}\{3-i\}, e)$}
16   if (i=1) then {
17     send(bag,e,0);
18     {emp}
19     $p$
20   }
21   else close(e,f);
22   {emp}
23 }
```

The rule is applied on line 14: initially, nothing is owned (not even the endpoint f, see line 12), but after the reception of the message bag, the endpoint f is owned.

Similarly, it is possible to use this extended rule for receive to provide encodings of locks in the heap, synchronization barriers, broadcast, and others [75]. The reader may find the extended rule for receives a bit disturbing, because it somehow goes against the natural intuition: one may consider that *attempting* to receive on an endpoint should require to own it first. We give two different answers to this criticism.

The first answer tries to formalize the intuition that "something must be owned for attempting to receive". Assume we decompose the endpoint's predicate in two smaller ones as follows:

$$e \overset{ep}{\mapsto} (C\{q\}, f) \qquad\qquad \Leftrightarrow \qquad\qquad e \overset{ep}{\mapsto} C\{q\} \ \ * \ \ e \overset{ep}{\mapsto} f.$$

The first predicate provides the ownership of the control's state, and the second one the ownership of the endpoint's capability. Then an alternative proof of the above example could be to let Paul and Peggy own a fraction of the capability to use e and f (hence the precondition of `grope`(p) would be $e \overset{ep}{\mapsto}_{.5} f * f \overset{ep}{\mapsto}_{.5} e$), and consider that owning a fractional permission on the capability allows to ‘attempt” receiving. Note however that it still requires to consider that the ownership of the endpoint's state is given by the bag message, so we cannot avoid to extend the receive rule. However, such a reasoning would have an advantage on the proof presented in Example 11, because the proof Example 11 requires a proof environment that is not admissible as it allows memory leaks.

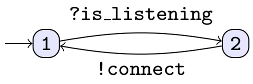
The second answer is that the extended receive rule makes sense because one should authorize to attempt to receive on a channel without owning it, as it may be a natural way of modeling certain protocols. Consider that the IP address “192.168.12.1:23” is a valid endpoint location. Then it may make sense to consider that clients using this IP address do not “own” this address: they just know it. In the code of Figure 3.8, we illustrate this idea a bit more in detail: we assume a `bind(ip)` primitive returning a channel (ip, f), and show how a negociated connection to a service offered at this IP address can be modeled and proved using the extended rule for receive.

Non-determinism can thus be introduced in two ways: either adopting permissions, or adopting a more symmetric view of send and receive. Example 11 suggest that permissions alone may sometimes be limited; conversely, it is not clear how far the second approach can be pushed without adopting permissions.

Client and Server

```
server(){
  local f,e',ip;
  {emp}
  ip:=<<192.168.12.1:23>>;
  f:= bind(ip);
  $\{ip \overset{ep}{\mapsto} (C\{1\}, f) * f \overset{ep}{\mapsto} (\overline{C}\{1\}, ip)\}$
  listen(f);
  $\{f \overset{ep}{\mapsto} (\overline{C}\{2\}, ip)$
  while(true) {
    e':=accept(f);
    $\{f \overset{ep}{\mapsto} (\overline{C}\{2\}, ip) * e' \overset{ep}{\mapsto} (C'\{q_0\}, -)\}$
    spawn service(e');
    $\{f \overset{ep}{\mapsto} (\overline{C}\{2\}, ip)\}$
  }
}
```

```
client(){
  local f';
  {emp}
  f':=connect(192.168.12.1:23);
  $\{f' \overset{ep}{\mapsto} (\overline{C'}\{q_0\}, -)\}$
  run_service(f');
}
// Messages
message is_listening $\{\mathsf{dst} \overset{ep}{\mapsto} (C\{1\}, -)\}$
message connect(ep)
  $\{\mathsf{dst} \overset{ep}{\mapsto} (C\{2\}, -) * ep \overset{ep}{\mapsto} (C'\{q_0\}, -)\}$
// Connection's Contract
contract C {
```



```
}
// Service's Contract
contract C' {initial state $q_0$ ...}
```

"System calls"

```
listen(f){
  local e';
  $\{ip \overset{ep}{\mapsto} (C\{1\}, f) * f \overset{ep}{\mapsto} (\overline{C}\{1\}, ip)\}$
  send(is_listening,f);
  $\{f \overset{ep}{\mapsto} (\overline{C}\{2\}, \})$
}

accept(f){
  local e';
  $\{f \overset{ep}{\mapsto} (\overline{C}\{2\}, -)\}$
  e':=receive(connect,f);
  $\{f \overset{ep}{\mapsto} (\overline{C}\{2\}, ip) \; * \; ip \overset{ep}{\mapsto} (C\{2\}, f)$
      $* \; e' \overset{ep}{\mapsto} (C'\{q_0\}, -)\}$
  send(is_listening,f);
  return e';
  $\{f \overset{ep}{\mapsto} (\overline{C}\{2\}, -) \; * e' \overset{ep}{\mapsto} (C'\{q_0\}, -)\}$
}
```

```
connect(ip){
  local e',f';
  {emp}
  receive(is_listening,ip);
  $\{ip \overset{ep}{\mapsto} (C\{2\}, -)\}$
  (e',f') := open(C');
  $\{ip \overset{ep}{\mapsto} (C\{2\}, -) \; * \; e' \overset{ep}{\mapsto} (C'\{q_0\}, f')$
        $* \; f' \overset{ep}{\mapsto} (\overline{C}\{q_0\}, e')\}$
  send(connect,ip,e');
  $\{f' \overset{ep}{\mapsto} (\overline{C}\{q_0\}, -)\}$
  return f';
}
```

Figure 3.8: A model of a negociated connection to a server

Chapter 4

Perspectives

Completeness and Conciseness of Separation Logic

The completeness result we presented (see Theorem 2.6) is very unsatisfactory, as it does not say anything about programs with function calls and concurrent programs, which are precisely the kind of features that motivates local reasoning. Local reasoning requires to adopt a lot of proof rules (the frame rule, the AVE rule, the parallel rule, the higher-order frame rule, the anti-frame rule, etc). These new proof rules "complete" the proof theory, as they allow to show more example programs, but the completeness of a given proof system with respect to a certain class of programs has at best been timidly raised. We are particularly interested in understanding which rules would yield a complete proof system for the subclass of race-free programs we called "selfish" programs.

The completeness results we stated show that some proof rules (namely the frame rule and the AVE rule) can be eliminated from the proof system in the case of sequential programs. This is also a quite unsatisfactory result. Even if some proof rules can be eliminated, it would be interesting to better formalize how they make the proofs of programs more concise. Among others, the importance of ghost variables and ghost code in proofs is extremely clear in practice, but very obscure from a theoretical point of view. If ghost code turned to be a fundamental feature for completeness or conciseness, this would probably raise difficult problems in automatic verification (how a ghost code can be efficiently checked to be "ghost"? how a ghost code can be automatically guessed, like other annotations are guessed? etc).

We hope to further extend our study of completeness issues to give a more precise answer to these questions.

Enriching Contracts

An active field of research consists in extending contracts, or more precisely session types, with facilities for describing multi-party sessions, roles, full-duplex communications, correlations among values of messages, etc. On the one hand, Sing♯ contracts can be considered as already expressive enough – weren't they successfully used for specifying a whole operating system? On the other hand, while playing with some example programs, we experienced several limitations of Sing♯ contracts (one example needs the contract to express that consecutive messages contain consecutive cells of a list, another needs to model counters in a protocol that is not even half-duplex). We expect to develop and integrate richer forms of contracts in our proof system and in tool implementations.

Proving Copying Message-Passing

We showed that local reasoning and the ownership concept were useful to analyse copyless message-passing. This form of message-passing is not, however, the most popular one – although the development of multi-cores may be currently leading program designers to pay more attention about it. Perhaps the most popular framework for message-passing programming is the message-passing interface (MPI). While reading through some tutorials on MPI, we discovered that it could be very useful to run a ownership analysis on MPI programs.

Indeed, MPI programs do not share their address space, but from time to time they delegate the ownership of a memory region to the MPI library. For instance, after a call to a non-blocking asynchronous send or receive, a memory block is scheduled to be read or written by the MPI library at some undefined point in the future. The memory block should thus not be incorrectly accessed afterwards, and, to express that, such primitives could be seen as transferring the ownership of the memory block from the process to the library. These calls to asynchronous non-blocking communications are usually followed at some point later in the program code by a call to a `wait` primitive, which returns only when the MPI library has finished to read or fill the memory block. Similarly, this wait could be considered as a transfer of the ownership of the memory block from the MPI library to the program.

It would be interesting to develop automatic techniques for local reasoning on MPI code. Impressive progresses on local reasoning for C code may suggest that it may be not such a complicated task, but on the other hand, the semantics of MPI seems almost as cumbersome as the one of C, and a serious modeling of MPI should probably be conducted with a lot of care.

Local Reasoning in Security

Local reasoning has just started to emerge in the area of security [76], and it is difficult to predict how it will play its role there. One question that might be of interest is how Shannon's information can be tracked as a "resource", or in other words, how the separation of information, *i.e.* the probabilistic independence, can be addressed by local reasoning. This question becomes interesting for security if information ownership entails a form of privacy[1]. A natural model

[1] This is not entirely clear: Separation Logic is a logic for *collaborative* concurrent programs, and the form of ownership it usually deals with corresponds to a form of agreement be-

of information as resource arises from the set of finite ponderations of concrete states of the form $p_1.\sigma_1 + \cdots + p_n.\sigma_n$: when they are equiped with the following composition law[2]

$$(p_1.\sigma_1 + \cdots + p_n.\sigma_n) \bullet (p'_1.\sigma'_1 + \cdots + p'_m\sigma'_m) \quad \triangleq \quad \sum_{i,j.\sigma_i \perp \sigma'_j} p_i.p'_j.\sigma_i \bullet \sigma'_j$$

ponderations of states form a separation algebra. The random instruction can be seen as “allocating information” in this resource interpretation: if $rand(x)$ denotes a variable containing a random value, then the random instruction admits the small axiom $\{\mathsf{emp}\}\ x := \text{random}()\ \{rand(x)\}$, and, by application of the frame rule, the independence of the random value of x with respect to any other random value is ensured. The difficulty faced by this approach is the representation of information sharing. It seems possible to represent sharing with permissions, and to prove for instance a simple concurrent version of an oblivious transfer [77], but it is not completely clear that it is the right way to do it, nor which resource model could support it. One may look for another resource model, possibly inspired by other models of information and knowledge, like the frames of the applied π-calculus [78] of Abadi and Fournet, where sharing is allowed by syntactically unbound names occuring in terms, or the shadow semantics [79] introduced by Morgan, where the randomness of shared variables is distinguished from the one of unshared variables. We already defined a form of separation over applied-pi frames [3, 80, 4] that could be useful to give some foundations to an assertion language for information-processing programs, but to be honest, this is far from giving us more than very vague intuitions on how information could be modeled as a resource.

tween all trusted parties involved in the process, whereas cryptography has to deal with *competitive* concurrent programs.

[2] To avoid renormalization, and to preserve the cancelative axiom of separation algebras, one should take $(p_1.\sigma_1 + \cdots + p_n.\sigma_n) \ \perp\ (p'_1.\sigma'_1 + \cdots + p'_m\sigma'_m)$ if for all i, j, either $\sigma_i \perp \sigma'_j$ or $p_i.p'_j = 0$

Bibliography

[1] Rémi Brochenin, Stéphane Demri, and Étienne Lozes. Reasoning about sequences of memory states. *Annals of Pure and Applied Logics*, 161(3):305–323, December 2009. Cited on page 5.

[2] Sébastien Bardin, Alain Finkel, Étienne Lozes, and Arnaud Sangnier. From pointer systems to counter systems using shape analysis. In Ramesh Bharadwaj, editor, *Proceedings of the 5th International Workshop on Automated Verification of Infinite-State Systems (AVIS'06)*, Vienna, Austria, April 2006. Cited on pages 6, 20.

[3] Étienne Lozes and Jules Villard. A spatial equational logic for the applied π-calculus. In Franck van Breugel and Marsha Chechik, editors, *Proceedings of the 19th International Conference on Concurrency Theory (CONCUR'08)*, volume 5201 of *Lecture Notes in Computer Science*, pages 387–401, Toronto, Canada, August 2008. Springer. Cited on pages 6, 69.

[4] Florent Jacquemard, Étienne Lozes, Ralf Treinen, and Jules Villard. Multiple congruence relations, first-order theories on terms, and the frames of the applied pi-calculus. In Sebastian Mödersheim and Catuscia Palamidessi, editors, *TOSCA*, volume 6993 of *Lecture Notes in Computer Science*, pages 166–185. Springer, 2011. Cited on pages 6, 69.

[5] Étienne Lozes and Jules Villard. Reliable contracts for unreliable half-duplex communications. In Marco Carbone and Jean-Marc Petit, editors, *WS-FM*, volume 7176 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 2011. Cited on pages 6, 55, 56.

[6] Kshitij Bansal, Rémi Brochenin, and Étienne Lozes. Beyond shapes: Lists with ordered data. In Luca de Alfaro, editor, *Proceedings of the 12th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'09)*, volume 5504 of *Lecture Notes in Computer Science*, pages 425–439, York, UK, March 2009. Springer. Cited on pages 6, 33, 34.

[7] Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *POPL*, pages 14–26, 2001. Cited on pages 9, 11.

[8] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001. Cited on page 9.

[9] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002. Cited on pages 9, 11.

[10] John C. Reynolds. An overview of separation logic. In Bertrand Meyer and Jim Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 460–469. Springer, 2005. Cited on page 9.

[11] Peter W. O'Hearn. Tutorial on separation logic (invited tutorial). In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 19–21. Springer, 2008. Cited on page 9.

[12] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE Computer Society, 2007. Cited on pages 14, 16, 30, 48, 61.

[13] Peter W. O'Hearn. Resources, concurrency and local reasoning. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 49–67. Springer, 2004. Cited on page 14.

[14] Stephen D. Brookes. A semantics for concurrent separation logic. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 16–34. Springer, 2004. Cited on page 16.

[15] Viktor Vafeiadis. Concurrent separation logic and operational semantics. *Electr. Notes Theor. Comput. Sci.*, 276:335–351, 2011. Cited on page 16.

[16] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16-18, 2004, Proceedings*, volume 3328 of *Lecture Notes in Computer Science*, pages 97–109. Springer, 2004. Cited on page 17.

[17] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2005. Cited on page 17.

[18] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005. Cited on page 17.

[19] Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *FSTTCS*, volume 2245 of *Lecture Notes in Computer Science*, pages 108–119. Springer, 2001. Cited on pages 17, 22.

[20] Nikos Gorogiannis, Max I. Kanovich, and Peter W. O'Hearn. The complexity of abduction for separated heap abstractions. In Eran Yahav, editor, *SAS*, volume 6887 of *Lecture Notes in Computer Science*, pages 25–42. Springer, 2011. Cited on pages 17, 20.

[21] Byron Cook, Christoph Haase, Joël Ouaknine, Matthew J. Parkinson, and James Worrell. Tractable reasoning in a fragment of separation logic. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR*, volume 6901 of *Lecture Notes in Computer Science*, pages 235–249. Springer, 2011. Cited on page 18.

[22] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2006. Cited on page 20.

[23] Arnaud Sangnier. *Vérification de systèmes avec compteurs et pointeurs*. Thèse de doctorat, Laboratoire Spécification et Vérification, ENS Cachan, France, November 2008. Cited on page 20.

[24] Sébastien Bardin, Alain Finkel, and David Nowak. Toward symbolic verification of programs handling pointers. In Ramesh Bharadwaj, editor, *Proceedings of the 3rd International Workshop on Automated Verification of Infinite-State Systems (AVIS'04)*, Barcelona, Spain, April 2004. Cited on page 20.

[25] Alain Finkel, Étienne Lozes, and Arnaud Sangnier. Towards model checking pointer systems. In Margaret Archibald, Vasco Brattka, Valentin Goranko, and Benedikt Löwe, editors, *Revised Selected Papers of the International Conference on Infinity in Logic & Computation (ILC'07)*, volume 5489 of *Lecture Notes in Artificial Intelligence*, pages 56–82, Cape Town, South Africa, October 2009. Springer-Verlag. Cited on page 20.

[26] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomás Vojnar. Programs with lists are counter automata. In

Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 517–531. Springer, 2006. Cited on page 20.

[27] Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Variance analyses from invariance analyses. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 211–224. ACM, 2007. Cited on page 20.

[28] Alexey Gotsman, Byron Cook, Matthew J. Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In Shao and Pierce [81], pages 16–28. Cited on page 20.

[29] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 211–222, New York, NY, USA, 2010. ACM. Cited on page 20.

[30] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In Shao and Pierce [81], pages 289–300. Cited on page 20.

[31] Peter W. O'Hearn. Abductive, inductive and deductive reasoning about resources. In Anuj Dawar and Helmut Veith, editors, *CSL*, volume 6247 of *Lecture Notes in Computer Science*, pages 49–50. Springer, 2010. Cited on page 20.

[32] Cristiano Calcagno, Dino Distefano, and Viktor Vafeiadis. Bi-abductive resource invariant synthesis. In Zhenjiang Hu, editor, *APLAS*, volume 5904 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2009. Cited on page 20.

[33] Étienne Lozes. Separation logic preserves the expressive power of classical logic. In *Proceedings of the 2nd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE'04)*, Venice, Italy, January 2004. Cited on page 22.

[34] Étienne Lozes. Adjuncts elimination in the static ambient logic. In Flavio Corradini and Uwe Nestmann, editors, *Proceedings of the 10th International Workshop on Expressiveness in Concurrency (EXPRESS'03)*, volume 96 of *Electronic Notes in Theoretical Computer Science*, pages 51–72, Marseilles, France, June 2004. Elsevier Science Publishers. Cited on page 22.

[35] Étienne Lozes. Elimination of spatial connectives in static spatial logics. *Theoretical Computer Science*, 330(3):475–499, February 2005. Cited on page 22.

[36] Anuj Dawar, Philippa Gardner, and Giorgio Ghelli. Adjunct elimination through games in static ambient logic. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS*, volume 3328 of *Lecture Notes in Computer Science*, pages 211–223. Springer, 2004. Cited on page 22.

[37] Cristiano Calcagno, Philippa Gardner, and Matthew Hague. From separation logic to first-order logic. In Vladimiro Sassone, editor, *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 395–409. Springer, 2005. Cited on page 22.

[38] Cristiano Calcagno, Matthew J. Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In Hanne Riis Nielson and Gilberto Filé, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2007. Cited on page 22.

[39] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context logic as modal logic: completeness and parametric inexpressivity. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 123–134. ACM, 2007. Cited on page 25.

[40] James Brotherston and Max I. Kanovich. Undecidability of propositional separation logic and its neighbours. In *LICS*, pages 130–139. IEEE Computer Society, 2010. Cited on page 25.

[41] Marius Bozga, Radu Iosif, and Swann Perarnau. Quantitative separation logic and programs with lists. *J. Autom. Reasoning*, 45(2):131–156, 2010. Cited on page 25.

[42] Anuj Dawar, Philippa Gardner, and Giorgio Ghelli. Expressiveness and complexity of graph logic. *Inf. Comput.*, 205(3):263–310, 2007. Cited on pages 25, 26, 27.

[43] Rémi Brochenin, Stéphane Demri, and Étienne Lozes. On the almighty wand. In Michael Kaminski and Simone Martini, editors, *Proceedings of the 17th Annual EACSL Conference on Computer Science Logic (CSL'08)*, volume 5213 of *Lecture Notes in Computer Science*, pages 323–338, Bertinoro, Italy, September 2008. Springer. Cited on pages 26, 28.

[44] M. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 41:1–35, 1969. Cited on page 26.

[45] L. Stockmeyer. *The complexity of decision problems in automata theory and logic*. PhD thesis, Department of Electrical Engineering, MIT, 1974. Cited on page 26.

[46] Jerzy Marcinkowski. On the expressive power of graph logic. In Zoltán Ésik, editor, *CSL*, volume 4207 of *Lecture Notes in Computer Science*, pages 486–500. Springer, 2006. Cited on page 27.

[47] Timos Antonopoulos and Anuj Dawar. Separating graph logic from mso. In Luca de Alfaro, editor, *FOSSACS*, volume 5504 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2009. Cited on page 27.

[48] Viktor Kuncak and Martin C. Rinard. On spatial conjunction as second-order logic. *CoRR*, cs.LO/0410073, 2004. Cited on page 28.

[49] Rémi Brochenin, Stéphane Demri, and Étienne Lozes. On the almighty wand. *Inf. Comput.*, 211:106–137, 2012. Cited on page 29.

[50] Mikolaj Bojanczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4):27, 2011. Cited on page 33.

[51] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 177–190, New York, NY, USA, 2006. ACM. Cited on page 35.

[52] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *ESOP*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. Cited on page 40.

[53] The erlang programming language. `http://www.erlang.org`. Cited on page 41.

[54] Sarvani S. Vakkalanka, Anh Vo, Ganesh Gopalakrishnan, and Robert M. Kirby. Precise dynamic analysis for slack elasticity: Adding buffering without adding bugs. In Rainer Keller, Edgar Gabriel, Michael M. Resch, and Jack Dongarra, editors, *EuroMPI*, volume 6305 of *Lecture Notes in Computer Science*, pages 152–159. Springer, 2010. Cited on page 41.

[55] Massimo Merro. *Locality in the π-calculus and applications to distributed objects*. PhD thesis, École des Mines de Paris, 2000. Cited on page 43.

[56] Philippa Gardner, Cosimo Laneve, and Lucian Wischik. Linear forwarders. *Inf. Comput.*, 205(10):1526–1550, 2007. Cited on page 43.

[57] Christian J. Bell, Andrew W. Appel, and David Walker. Concurrent separation logic for pipelined parallelization. In *SAS*, volume 6337 of *LNCS*, pages 151–166, 2010. Cited on pages 44, 61.

[58] K. Rustan M. Leino, Peter Müller, and Jan Smans. Deadlock-free channels and locks. In *ESOP*, volume 6012 of *LNCS*, pages 407–426, 2010. Cited on page 46.

[59] Jules Villard. *Heaps and Hops.* Thèse de doctorat, Laboratoire Spécification et Vérification, École Normale Suprieure de Cachan, France, February 2011. Cited on pages 46, 48, 50, 51, 60.

[60] Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving copyless message passing. In Zhenjiang Hu, editor, *Proceedings of the 7th Asian Symposium on Programming Languages and Systems (APLAS'09)*, volume 5904 of *Lecture Notes in Computer Science*, pages 194–209, Seoul, Korea, December 2009. Springer. Cited on pages 48, 62.

[61] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In Zhong Shao, editor, *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 19–37. Springer, 2007. Cited on pages 48, 60.

[62] Aquinas Hobor, Andrew Appel, and Francesco Nardelli. Oracle semantics for concurrent separation logic. In Sophia Drossopoulou, editor, *Programming Languages and Systems*, volume 4960 of *Lecture Notes in Computer Science*, pages 353–367. Springer Berlin / Heidelberg, 2008. Cited on page 48.

[63] Aquinas Hobor, Robert Dockins, and Andrew W. Appel. A theory of indirection via approximation. In Manuel V. Hermenegildo and Jens Palsberg, editors, *POPL*, pages 171–184. ACM, 2010. Cited on page 48.

[64] Zachary Stengel and Tevfik Bultan. Analyzing singularity channel contracts. In Gregg Rothermel and Laura K. Dillon, editors, *ISSTA*, pages 13–24. ACM, 2009. Cited on pages 55, 56.

[65] Mohamed G. Gouda, Eric G. Manning, and Yao-Tin Yu. On the progress of communications between two finite state machines. *Information and Control*, 63(3):200–216, 1984. Cited on page 56.

[66] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *POPL*, pages 273–284. ACM, 2008. Cited on page 56.

[67] Gérard Cécé and Alain Finkel. Verification of programs with half-duplex communication. *Information and Computation*, 202(2):166–190, November 2005. Cited on page 56.

[68] Samik Basu and Tevfik Bultan. Choreography conformance via synchronizability. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *WWW*, pages 795–804. ACM, 2011. Cited on page 56.

[69] Viviana Bono, Chiara Messa, and Luca Padovani. Typing copyless message passing. In Gilles Barthe, editor, *ESOP*, volume 6602 of *Lecture Notes in Computer Science*, pages 57–76. Springer, 2011. Cited on page 60.

[70] Mike Dodds, Suresh Jagannathan, and Matthew J. Parkinson. Modular reasoning for deterministic parallelism. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 259–270. ACM, 2011. Cited on page 61.

[71] Adrian Francalanza, Julian Rathke, and Vladimiro Sassone. Permission-based separation logic for message-passing concurrency. *CoRR*, abs/1106.5128, 2011. Cited on page 61.

[72] Lars Birkedal and Hongseok Yang. Relational parametricity and separation logic. In Helmut Seidl, editor, *FoSSaCS*, volume 4423 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2007. Cited on page 61.

[73] Nick Benton and Benjamin Leperchey. Relational reasoning in a nominal semantics for storage. In Pawel Urzyczyn, editor, *TLCA*, volume 3461 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2005. Cited on page 61.

[74] Marco Giunti and Vasco Thudichum Vasconcelos. A linear account of session types in the pi calculus. In Paul Gastin and François Laroussinie, editors, *CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 432–446. Springer, 2010. Cited on page 62.

[75] Étienne Lozes and Jules Villard. Sharing contract-obedient endpoints. Research Report LSV-11-23, Laboratoire Spécification et Vérification, ENS Cachan, France, December 2011. 42 pages. Cited on page 63.

[76] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 91–102. ACM, 2006. Cited on page 68.

[77] Etienne Lozes. Towards information as resource in separation logic. Draft presented at FCS-PrivMod, Edinburgh, 2010. Cited on page 69.

[78] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *POPL*, pages 104–115, 2001. Cited on page 69.

[79] Carroll Morgan. The shadow knows: Refinement and security in sequential programs. *Sci. Comput. Program.*, 74(8):629–653, 2009. Cited on page 69.

[80] Étienne Lozes and Jules Villard. A spatial equational logic for the applied π-calculus. *Distributed Computing*, 23(1):61–83, September 2010. Cited on page 69.

[81] Zhong Shao and Benjamin C. Pierce, editors. *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. ACM, 2009. Cited on page 74.