Thèse

Présentée à

L'École Normale Supérieure de Cachan

en vue de l'obtention du titre de

Docteur en Informatique

Heaps and Hops

Soutenue le 18 février 2011 par

Jules Villard

devant un jury composé de :

Peter O'Hearn	*rapporteurs*
Nobuko Yoshida	
Cristiano Calcagno	*examinateur*
Byron Cook	*président du jury*
Étienne Lozes	*directeurs*
Alain Finkel	

Abstract

This dissertation is about the specification and verification of concurrent message-passing programs that operate over a shared memory. In this setting, instead of copying messages over channels, processes may exchange pointers into a shared memory where the actual contents of messages lay. Channels are themselves objects in the memory (or heap) that can be communicated, thus achieving full mobility. This flexible and efficient programming paradigm must be used carefully: every pointer that is communicated becomes shared between its sender and its recipient, which may introduce races. To err on the side of caution, the sender process should not attempt to access the area of storage circumscribed by a message once it has been sent. Indeed, this right is now reserved to the recipient, who may already have modified it or even disposed of it. In other words, the ownership of pieces of heap hops from process to process following the flow of messages.

Copyless message passing combines two features of programs that make formal verification challenging: explicit memory management and concurrency. To tackle these difficulties, we base our approach on two recent developments. On the one hand, concurrent separation logic produces concise proofs of pointer-manipulating programs by keeping track only of those portions of storage owned by the program. We use such local reasoning techniques to analyse the fluxes of ownership in programs, and ensure in particular that no dangling pointer will be dereferenced or freed at runtime. On the other hand, channel contracts, a form of session types introduced by the Sing♯ programming language, provide an abstraction of the exchanges of messages that can be used to statically verify that programs never face unexpected message receptions and that all messages are delivered before a channel is closed.

The contributions contained in this dissertation fall into three categories. First, we give semantic definitions to copyless message-passing programs, to the ownership transfers they induce, and to channel contracts, and link the three together. In doing so, we provide the first formal model of a theoretically significant subset of the Sing♯ programming language. In particular, we show that some properties of channel contracts rub off on programs, which justifies their use as protocol specifications. Second, we introduce the first proof system for copyless message passing, based on separation logic and channel contracts. The proof system discharges parts of the verification of programs on the verification of their contracts. The marriage of these two techniques allows one to prove that programs are free from memory faults, race conditions, and message-passing errors such as unspecified receptions and undelivered messages. Moreover, we show how the logic and contracts cooperate to prove the absence of memory leaks. Third, we give an implementation of our analysis, Heap-Hop,

that takes annotated programs as input and automatically checks the given specifications and deduces which of the properties above are enjoyed by the program. Heap-Hop requires programs to be annotated with pre and postconditions for each function, loop invariants, and the channel contracts that communications should obey.

Contents

Remerciements

Ma gratitude va en premier lieu à Étienne Lozes, qui a encadré mes travaux avec enthousiasme, gentillesse et rigueur scientifique. Merci pour ta disponibilité sans faille. Je remercie également Cristiano Calcagno, qui a guidé mes premiers pas dans le domaine de la logique de séparation, et qui a donné le point de départ de cette thèse. Je remercie Daniel Hirschkoff pour avoir parrainé ma vie scientifique, et Alain Finkel pour avoir dirigé ma thèse.

Je remercie toutes les autres personnes avec qui j'ai eu l'occasion de parler de science pendant ma thèse, et qui m'ont encouragé à continuer : Benedikt Bollig, Steve Brookes, Luis Caìres, Byron Cook, Dino Distefano, Rob Dockins, Mike Dodds, Philippa Gardner, Maurice Herlihy, Aquinas Hobor, Florent Jacquemard, Peter O'Hearn, François Pottier, Hugo Torres Vieira, Ralf Treinen, Viktor Vafeiadis, Hongseok Yang, Nobuko Yoshida et bien d'autres.

Merci à tout le LSV (Renaudeau et Iris représentent !) pour m'avoir fourni un accueil chaleureux, des voyages scientifiques épanouissants, et un environnement de travail stimulant.

Big up aux Gawky : Flutch, Nicolas et Oanig, et aux geeks de ♯`sos` : bgoglin, Dain, GruiicK, iderrick, julio, Louloutte, mandos, mrpingouin, nanuq, Nimmy, proutbot, stilgar, TaXules, tonfa, Vinz, youpi. Merci surtout à ceux qui ont contribué à rendre moins aride la traversée de la thèse : Amélie M., Arnaud D., Arnaud S., Delphine L., Günther S., Juliette S., Miles A., Mouchy, Philippe P., Sylvain S., j'en passe et pas des pires.

Je remercie enfin Jade, mes parents, et ma famille, pour tout ce qu'ils m'ont apporté.

Introduction

Modelling and verification of message-passing programs

The formal verification of programs strives to ensure, with the certainty of a mathematical theorem, that programs do what they were conceived to do. Programs do not always behave as desired, as one may experience in its everyday interactions with computers: sometimes they crash, or produce unexpected results. What is a mere annoyance on a home computer can have dramatic consequences for more sensitive programs. For instance, in 1996, the Ariane 5 spatial rocket exploded in the atmosphere due to an integer overflow present in its code. Despite careful examination by engineers, the anomaly went undetected, whereas formal methods could have been applied to prevent it.

Formal methods allow one to go beyond naive testing of programs. Indeed, tracking errors down by means of testing involves making sure that a program will behave correctly, whatever its input. Unfortunately, the number of situations that a program can run into is most of the time virtually infinite. Thus, empirically testing the conformance of a program against a certain specification by running it against each possible configuration of the system cannot be done in any reasonable amount of time.

On the other hand, if one is given a mathematical model of what the program does and a mathematical model of its specification, one can try and *formally verify* that there is a match between the two. Thus, one can provide a mathematical proof of a program's correction, which amounts to having successfully tested this program against all possible inputs.

Amongst the programs that one may wish to formally verify, concurrent ones make for especially challenging problems, while coincidentally being the ones most in need of formal methods of verification. Concurrent programs are already numerous, and their number is bound to grow in the future. Indeed, due to a shift in the consequences of Moore's law (which states that the number of transistors that can be placed on an integrated circuit doubles approximately every year) recent years have seen a proliferation of computers containing more and more processors and cores capable of running in parallel, a characteristic that was previously the privilege of specialised, high performance computers. As a result, programs are also forced to become parallel to take advantage of the increase in available computing power, instead of merely relying on the frequency of single processors to increase.

Yet, concurrent programming is still, after decades of research on the subject, a complicated matter. Whenever several processes interact towards a common goal, the number of ways they have of misbehaving skyrockets. Worse, the errors they make may depend on subtle timing issues between the actions of each process. This makes them hard to detect and reproduce using traditional debugging tools, especially since merely trying to observe a problem may hide it away. Formal methods have thus an important role to play in verifying concurrent programs.

Finally, it is worth mentioning that designing the models that form the basis for formal verification is often a difficult task in itself. This is particularly true in the concurrent setting where the effect of interactions has to be carefully taken into account.

In this thesis, we are interested in the formalisation and formal verification of message-passing programs. Our goal is to define a realistic model of a particular class of such programs and to analyse some of their properties in a manner that is suitable to some level of automation. Amongst the many properties that one may wish to establish for message-passing programs, we have chosen to focus on those which ensure that a program *never goes wrong*, that is in *safety* properties.

Copyless message passing

Parallel programs, in all generality, need means to synchronise and communicate with each other. Message passing is a theoretically elegant tool to achieve this: if a process wishes to communicate a value, it sends it to another process; if two processes should wait for each other before advancing further, they may exchange messages and wait for corresponding acknowledgements.

The relative simplicity of the message passing paradigm, while helping in writing conceptually simpler code, usually comes with a cost in efficiency, as this programming paradigm is quite remote from the inner operations of a machine. This may explain why more basic primitives like spin-locks, semaphores, or even memory barriers, have often been favoured in programming languages, although they are arguably more error-prone than message passing for the programmer.

However, message passing can also be an efficient programming paradigm, as has been demonstrated by Singularity. Singularity [HL07] is a research project and an operating system written from the premise that one may replace hardware memory protection mechanisms by systematic static analysis of programs' source code on the one hand, and dynamic checking at runtime on the other hand which, together, ensure strict isolation between processes. Indeed, Singularity can safely run processes sharing a single address space without hardware memory protection. Executable programs are compiled from the Sing♯ programming language, derived from C♯, which supports an efficient form of message-passing primitives described below.

In Sing♯, the contents of messages reside in a portion of the memory that is shared by all the processes. Thanks to this, a process wishing to send a message can avoid copying its contents over the channel, and instead send a mere pointer to the location of the message's contents in memory. When a process receives such a pointer, it gains the ability to access the contents of the message. In effect, the sender and the recipient of a message located in memory are *sharing* this piece of memory and one has to be careful to avoid some of the drawbacks of shared-memory communications, such as race conditions. The communications of SING♯ are in fact an example of *copyless message passing*, which constitute the main interest of this thesis.

In this thesis, we will explore one means of ensuring statically (and automatically to some extent) that programs communicating via copyless message passing do not encounter memory faults by requiring that a process sending a message loses its access right over its contents: this right is transferred to the receiving process. This condition is sufficient to ensure the absence of *race conditions* between processes. Moreover, we will insist on communications being ruled by channel contracts. As we will see, this reduces the checking of certain communication errors in programs to an analysis of their channel contracts, which are much easier to reason about. Our programming language features asynchronous, bidirectional FIFO (first-in first-out) channels, each made of two endpoints; endpoints are themselves exchangeable over channels (in fact, they are allocated on the heap, like regular memory cells). Our verification technique will be a *program logic* that builds on the ideas of *local reasoning*.

Local reasoning

Since copyless message passing assumes a shared memory between processes, any verification technique aiming at proving the correctness of programs using this paradigm will have to be able to cope with the memory (also referred to as the *heap*). Reasoning about memory-manipulating programs has long stood up as a difficult topic in the area of the formal verification of programs, which is mainly attributable to *aliasing*: in a heap-manipulating program, the same memory location may be accessed through multiple program variables (all containing this location as value). Whenever the contents of the memory at this location changes, the update should be visible for all the processes that may access it through one of these variables. From a verification standpoint, this usually means that one has to keep track of *all* the ways a memory location could be aliased. Aliasing makes reasoning about memory-manipulating programs *global* in that sense, which results in tedious proofs at best, if one indeed dares to go and prove such programs in such a way [Bur72, Bor00].

Separation logic [IO01, Rey02], which is derived from the logic of Bunched Implications [OP99] and extends Floyd-Hoare logic [Hoa69, Flo67], was proposed as a way to reason about such programs while avoiding the pitfalls that result from the tedious bookkeeping of aliasing constraints. The reasoning induced by separation logic is rooted in the following *local reasoning* idiom, defined as such by O'Hearn *et al.* [ORY01]:

> To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged.

Separation logic takes advantage of this idiom by maintaining assertions only about the portion of the heap that a program *knows* it will be the only one to access at this point in the execution, what it *owns* in a sense, in contrast to what it can access. This form of reasoning supposes that, in O'Hearn's words, "processes mind their own business" and typically gives rise to more concise proofs than those obtained by keeping track of aliasing constraints.

Moreover, local reasoning proved to be an elegant tool for reasoning about concurrent programs as well, provided that a clean separation between what is owned by each process can be established at any point in time. This can benefit the analysis of copyless message-passing programs. Indeed, the ownership reading of the heap allows us to translate our discussion about the transfer of the "right" to access the memory cells corresponding to the contents of a message into the following argument: whenever a process sends a message, it *loses ownership* of that portion of memory corresponding to the contents of the message; conversely, when a process receives a message, it *gains* ownership of that portion of memory. This form of *ownership transfer* prevents the overlapping of areas of memory that are owned by different processes at a given time: ownership of pieces of heaps hop from one process to another following the flow of messages.

Channel contracts

The message passing paradigm is not without its own pitfalls: for instance, it can introduce *deadlocks* in programs, when two processes each wait for the other to send a message, or *communication errors*, when an unexpected message is sent on a channel or when a channel is closed with undelivered messages still in its buffers. Moreover, the programmer may wish to export the *protocol* that is used on a particular channel, for instance for others to use as an interface to its program. To address some of these issues, Singularity introduced *contracts*, which are communicating automata used to specify such protocols. Contracts also allow the SING♯ compiler to statically verify some interesting properties about communications (which sometimes have to be double-checked at runtime in their case), for instance the absence of deadlocks on a single channel, or the existence of a bound on the number of pending messages. Their authors make the following appealing claim of modularity [FAH+06]:

> The channel contracts provide a clean separation of concerns between interacting components and help in understanding the system architecture at a higher level.

Contracts are part of our framework as well: our logic tries to prove a given program correct with respect to certain contracts, while these contracts are analysed separately. We were able to formally establish that good and bad properties of contracts sometimes rub off on the programs that implement them. Moreover, including contracts gave us the opportunity to formalise the ideas implemented by the Singularity Operating System (such a formalisation was not provided by their authors). We believe we have achieved such a "clean separation of concerns" in our approach.

Automation

Reasoning about programs that are concurrent and have direct access to the heap is challenging. Combating the complexity of such reasoning with local reasoning and compositionality makes for simpler proofs of programs. Yet, as we are interested in verifying concurrent, memory-manipulating programs (without first abstracting their behaviour by casting them into models more amenable to other verification techniques like finite model-checking [CES86]), the vast majority of properties this thesis tries to establish about programs are *undecidable* ones. Nevertheless, being able to give (arguably) simple, human-understandable proofs of programs supports the hope that these proofs may be automated, at least to some extent. Indeed, one may circumvent this issue and write automatic tools nonetheless either by helping the tool (for instance with program annotations), or by means of abstractions (for instance, by using an abstract domain of formulas, as in the Space Invader tool [DOY06], over which abstract interpretation techniques [CC77] can be applied). In this thesis, we have taken the first approach and use annotated programs as input for our Heap-Hop tool.

Contributions

Let us now present the contributions of this thesis, which are threefold.

Firstly, we propose a modelling of a programming language which, without being full-fledged, contains all the message-passing primitives needed for copyless message passing, and is already sufficiently rich in features to expose the intricate verification issues which follow from this paradigm. This modelling itself has three components:

- An operational semantics for programs in which messages are pure values. This semantics does nothing to explicitly track the ownership of each memory cell, variable and channel endpoint. However, as our analysis strives to eliminate them, *racy* programs, which perform simultaneous accesses to the same variable, memory cell or endpoint without proper synchronisation (resulting in a *race*), are forced to fault.

- A formalisation of contracts in terms of communicating finite states machines, a well-known model of communications.

- A second operational semantics for programs where messages carry the pieces of states whose ownership they are meant to transfer (in addition to their values), and where channels are associated to contracts. Although its main purpose is to help us prove the soundness of our logic, this second semantics also highlights the impact that contracts have on the verification of programs. Indeed, we connect this second semantics both to the first operational semantics of programs and to the semantics of contracts.

Secondly, we provide a proof system for copyless message passing programs with contracts based on separation logic. The logic discharges parts of the verification of the program on the verification of its contracts, thus realising the claim made by Singularity that contracts should provide a "clean separation of concerns" between the different aspects of the verification of copyless message passing programs. Our analysis can prove in particular that a program causes no memory fault or leak, has no race condition, and that channel communications are safe from both unspecified receptions and undelivered messages.

Thirdly, we have developed a tool, Heap-Hop, that takes annotated programs as input and automatically answers whether its specification is correct or not using our logic, and whether it enjoys the properties mentioned above. The logical fragment manipulated by Heap-Hop makes the entailment problems between formulas decidable while being powerful enough to permit symbolic execution.

Our proof system was first presented in the APLAS conference [VLC09], with a different proof of soundness based on *abstract separation logic*, a framework for proof systems based on separation logic. Heap-Hop was first presented at the TACAS conference [VLC10]. The formalisation of contracts, the modelling of the programming language and the proof of soundness presented here are original to this thesis.

Related work

Program verification is too vast a domain of computer science to be faithfully surveyed in this introduction. Let us compare the work presented in this thesis with existing literature on domains related to each of the three contributions pointed out above: modelling of message-passing programs and systems, proof systems for concurrent program verification, and automated verification of message-passing programs.

The literature on the modelling of message-passing concurrency is rich to say the least, and has spawned a variety of models, from communicating automata and process algebras to game semantics. The behaviours of these systems have themselves been studied using other models, such as message sequence charts or Mazurkiewicz traces [Maz86]. Models that belong to the family of process algebras are often derivatives of either the calculus of communicating systems (CCS), the pi-calculus (both introduced by Milner [Mil80, Mil99]), or Hoare's communicating sequential processes [Hoa78] (CSP). Hoare and O'Hearn have defined a denotational semantics for a message-passing programming language (close to CSP) with dynamically allocated channels [HO08]. Yet, very few other models of message-passing concurrency, if any, include simultaneously a tractable and realistic model of the heap. None of them are able to model storable channels or contracts in a natural fashion.

Recent progress has been made in the verification of communicating automata, our model of contracts. Our analysis shares many concerns with the recent work of Stengel and Bultan, who have also studied the contracts of SING♯ in a formal setting [SB09], albeit not relating them to a general model of message-passing programs. A cousin of our analysis of communications is the type system of Takeushi *et al.* [THK94, HVK98] for a process algebra derived from the pi-calculus, which describes channel protocols using *session types*, and has been the starting point of a rich and growing literature. In particular, session types can be used to describe asynchronous communications between any number of participants [HYC08] (whereas this thesis focuses on the binary case) and to capture event-driven programming [HKP$^+$10]. Finally, our own analysis of contracts has many roots in the study of half-duplex communicating automata by Cece and Finkel [CF05]. None of these models apply to heap-based communications, and in particular to our model of storable channels.

In the domain of proof techniques for copyless message passing, the logic of Bell *et al.* [BAW10], that also extends separation logic with copyless message passing, is very close to our own. Their proof system is however only concerned with proving the safety of programs with respect to memory faults, and as such do not include a treatment of contracts. We discuss the relationship between our logic and theirs in more details on page 98. Among the numerous extensions of concurrent separation logic, two in particular resemble our own. Firstly RGSep [VP07, Vaf07], which applies ideas of rely/guarantee [Jon83] to separation logic in order to obtain compositional proofs of concurrent programs communicating via shared memory. Secondly, the logic for storable locks of Gotsman *et al.* [GBC$^+$07], whose aim is to handle dynamically allocatable locks, much like we extend separation logic to handle dynamically allocatable channels. A comparison between our work and these logics (and in their proofs of soundness) can be found in Chapter 6.

Finally, as far as automatic tools are concerned, the SING♯ compiler also operates in a setting with copyless message passing and contracts. In fact, the language we consider is directly inspired by SING♯. However, their compiler delegates parts of the verification to the runtime, whereas we strive to keep the verification entirely static. Moreover, no formal study of their analysis has yet been published. Another neighbouring tool is Chalice [LM09, LMS10], which is capable of verifying programs (written in a variant of C♯) communicating via asynchronous unidirectional channels. However, the Chalice tool lacks support for contracts and different message *tags*. As a result, each channel may only be used to transport *one* particular type of value, whereas channels in our model may carry different types according to the current protocol state of the channel. On the other hand, Chalice is capable of analysing communications so as to prevent *deadlocks* [LMS10], which Heap-Hop does not not currently support. Similarly, the SESSION JAVA language [HYH08] is an extension of JAVA which verifies channel communications specified using session-types. Programs are typed using an analysis that concentrates purely on ensuring the safety of communications. Finally, the Spin model-checker [Hol97] is capable of analysing programs communicating via asynchronous or synchronous channels. The last two tools have only naive models of the heap, which would surely impair the verification of copyless message-passing programs.

There are few other tools capable of analysing communicating systems. The TReX tool [ABS01] analyses timed automata that manipulate counters and communicate via *lossy* FIFO channels. Lossy channels may non-deterministically lose some of the messages that are communicated over them, and form a class of systems easier to analyse than perfect channels [CFI96]. McScM [HGS09] (model-checker for systems of communicating FIFO machines), on the other hand, is capable of analysing automata communicating via asynchronous and reliable FIFO channels. Both of these tools drift away from programming languages to concentrate on communicating automata, and are thus more comparable to the analysis that Heap-Hop performs on contracts. They are more involved in that respect, as they focus on semantic properties of systems whereas Heap-Hop merely perform syntactic checks on contracts from which one can deduce the semantic properties of their behaviour.

Outline

The outline of the rest of the manuscript is as follows.

Chapter 1 This chapter informally motivates the design choices of our message-passing programming language, and presents the main verification questions that this thesis will address.

Chapter 2 This chapter gives a formal syntax and operational semantics to our copyless message-passing programming language.

Chapter 3 In this chapter, the focus is on the semantics and properties of contracts. Since some aspects of the verification of programs boil down to checking properties of their associated contracts, as we show in the following chapters, the decidability of various contract properties is investigated.

Chapter 4 This background chapter presents existing concurrent separation logics.

Chapter 5 In this chapter, we introduce our extension of separation logic to copyless message passing and discuss how to prove a few key examples.

Chapter 6 This technical chapter presents a second, *open* semantics for our programming language and produces the soundness proof for our logic: proved programs are shown to respect ownership of resources, to abide by their contracts, and thus to communicate in a safe way if their contracts so guarantee.

Chapter 7 This chapter presents our automatic proof tool: Heap-Hop.

Table of notations

The notations used throughout the thesis are grouped in Table 0.1.

Symbol	Denotation
$\varnothing$	empty set
$\mathbb{N}$	the set of natural numbers
$\mathbb{Z}$	the set of integers
i, j, n, m	integers
w	word
α, β	buffer contents
␣	empty word or queue
$\texttt{x,y,e,f,}\ldots$	variables
E	program or logical expression
B	boolean program expression
c	command
g	guarded external choice
p	program
v	value
ε	endpoint address
σ	closed program state
s	stack
h	cell heap
k	closed endpoint heap
$\mathfrak{M}$	CFSM
$\mathfrak{S}$	system of CFSMs
$\mathfrak{C}$	contract
$\mathfrak{D}$	dialogue system
q	control state of a CFSM
C	configuration of a CFSM
$a, b, c, \mathsf{cell}, \mathsf{fin}, \ldots$	message identifiers
$\dot{\sigma}$	idealised local state
$\dot{k}$	idealised endpoint heap
ϕ,ψ	formulas
γ	invariant or footprint
Γ	footprint context
$\underline{\sigma}$	open state
$\underline{k}$	open endpoint heap

Table 0.1: Notation used in the typescript.

Chapter 1

Concurrency and Ownership

1.1 Concurrency and the heap

1.1.1 Concurrent programs

A concurrent program can be described by a collection of *processes* (or *threads*) executing simultaneously. Each process consists of a sequence of instructions, and simultaneity is meant in a loose way; it may occur for a number of reasons, including the following ones:

- the processes are executed on different machines;
- the processes are executed on different processors of the same machine;
- the processes are executed on different cores of the same processor;
- the processes are executed on a single core, together with a *scheduler* that decides of an interleaving of the instructions of each process;
- any combination of the above.

In general, processes execute independently (for instance, unrelated user applications on a desktop computer), meaning that all the data they manipulate, have manipulated in the past, or will manipulate in the future live in disjoint areas of the memory.[1] Processes may also need to share or exchange data, and this is of course when interesting things happen. Inter-process communications mostly happen via shared portions of states, message passing, or variations of one of these two paradigms (for instance remote procedure calls, akin to message passing). We call *resource* a shared variable or portion of memory.

The rest of this section is devoted mainly to a survey of shared-variable concurrency for heap-manipulating programs. This will be useful in the rest of the thesis to be able to compare our analysis for message passing with existing ones for shared-variable concurrency.

[1] In the low-level details of operating systems, there may be some sharing even in this case, for instance via dynamically linked shared libraries or system calls. When specifying and verifying user-level code, we assume that the underlying operating system is sufficiently well-behaved that we can safely elude these details.

1.1.2 Race-free programs

A race condition occurs whenever there are at least two simultaneous accesses to a shared resource that are not protected by any synchronisation primitive (such as a lock), and at least one of them is a write. A program whose executions may exhibit a race condition is called *racy*, and *race-free* otherwise. More precisely, two threads or processes are in a race condition if they simultaneously try to access the same resource, and if at least one of them is trying to write to the resource. This is most of the time considered as an error. Indeed, the outcome of races is not deterministic as it may not even correspond to one of the outcomes produced by interleaving the instructions of the two culprit processes. This makes them very hard to detect and reproduce using traditional debugging tools. This also complicates considerably the definition of the semantics of programs; in particular, the semantics of the parallel composition of two programs becomes unclear in the presence of races.

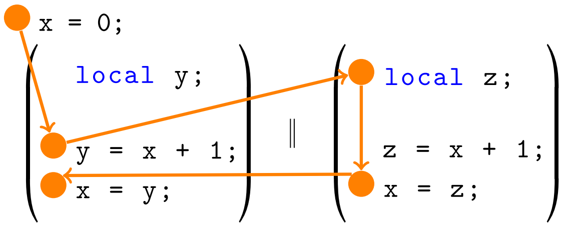
Racy programs Races often result in undesirable behaviours, for instance reading an incoherent value because another thread is writing to it, or eventually storing a scrambled value because of two simultaneous writes to a resource (this can happen when writes are not atomic, for instance when writing a 64-bit long word to a memory where the writing unit is 32 bit). Worse is the case of weak-memory models, in which the semantics of racy programs depend on the architecture: to determine the exact outcome of a race, it could be necessary to model the effect of all caches, store buffers, and other abstruse features of such architectures. Thus, races violate the atomicity of accesses to a resource: one cannot assume, in a racy execution, that accesses to a resource respect a certain order.

From a higher-level perspective, for instance in object-oriented programing where resources may be *objects* composed of several memory blocks and methods to operate on them, atomicity violations may put a resource in an incoherent internal state. Usually, each process assumes the shared resource to be in a certain state, to respect a certain *invariant*, and it could be the case that one process reads the contents of the resource while it is being manipulated by a second process that has temporarily put the resource into an inconsistent internal state.

Another case of erroneous racy execution occurs when the flow of a program is interrupted by another program in a piece of code that was meant to be executed atomically by the programmer, but was left unprotected. Consider the program of Figure 1.1, where two non-atomic increments of a shared variable `x` are performed in parallel and result in a race.

At the end of the execution of this program, one could expect the value of `x` to be 2, as it would be the case with two increments ordered in sequence. However, in a concurrent setting, the final result can also be 1, as exhibited by the interleaving drawn in Figure 1.1: the first thread is scheduled first, but is interrupted after the line `y = x + 1`. At this point, the second thread starts and is executed entirely. Now `x`, `y` and `z` all hold 1. The first thread is finally allowed to finish, and writes again 1 to `x`, hence the overall effect has been to set `x` to 1 instead of 2.

More generally, races make apparent the level of atomicity (or *granularity*) of programs and even of basic instructions; in their presence, one has to precisely state what is guaranteed to execute without interruption.



```
x = 0;
( local y;        )     ( local z;        )
(                 )  ∥  (                 )
( y = x + 1;      )     ( z = x + 1;      )
( x = y;          )     ( x = z;          )
```

Figure 1.1: Two unprotected increments in parallel and an interleaving which leads to the final state x = 1.

Of course, not all races are bugs: some programs are designed to be racy for efficiency purposes, and much care is given to ensure that these races will not affect the program badly. Yet, even in these cases, races are limited to instructions that are known to execute atomically, like compare-and-swap or test-and-set: these programs are closer to *non-blocking* or *fine-grained* concurrent programs than to truly racy ones. Programs of this sort are nonetheless very hard to get right, let alone formally verify. The book *The Art of Multiprocessor Programming* by Herlihy and Shavit [HS08] provides insights on the subject. Vafeiadis describes how to formally prove some of them in his thesis [Vaf07].

Race-free programs On the other hand, race-free programs (programs without race conditions) have several advantages that make them attractive for programmers and easier to verify. In particular, as Reynolds points out [Rey04], their semantics is *grainless*: one does not have to take the precise atomicity of primitive commands into account to give a semantics to race-free programs, and an interleaving semantics suffices to account for concurrency. Moreover, race-freedom allows one to elude the details of the memory model, as it restores sequential consistency (the fact that only those outcomes obtained by an interleaving of the executions of the processes are observed [Lam79]) on weak architectures.[2]

1.1.3 Synchronisation

If one wants to eliminate race conditions, one has to regulate access to resources. For instance, for the program of Figure 1.1, one has to protect the increment of x so as to prevent interferences from other threads. In other words, one has to ensure *mutual exclusion* between the accesses to x: no two threads may be accessing x at the same time when one of them is writing to it. To this end, one usually uses either locks or conditional critical regions.

Conditional critical regions Conditional critical regions (CCR), or monitors or Hoare monitors, are pieces of code that may be executed only when a specific *resource* is held. This ensures mutual exclusion between the CCRs guarded by the same resource. Resource

[2] This is a theorem of the folklore known as the data-race-free guarantee [AH90], which has been formally proved to hold for many existing architectures recently [Alg10, BP09, OSS09].

declarations and usage are syntactically scoped. Using resources, we can write a correct, race-free version of our example:

```
resource r in {
x = 0;
⎛local y;        ⎞     ⎛local z;        ⎞
⎜                ⎟     ⎜                ⎟
⎜with r {        ⎟     ⎜with r {        ⎟
⎜  y = x + 1;    ⎟  ‖  ⎜  z = x + 1;    ⎟
⎜  x = y;        ⎟     ⎜  x = z;        ⎟
⎝}               ⎠     ⎝}               ⎠
}
```

Or more concisely, writing `x++` for `x = x + 1`:

```
resource r;
x = 0;
with r { x++; } ‖ with r { x++; }
```

All the executions of the program above are race free and give a final value of 2 for `x`.

Deadlocks Although they may be used to prevent races, conditional critical regions can themselves introduce errors in the form of *deadlocks* (as in fact any means of synchronisation between programs). A deadlock arises when two or more processes are waiting for each other to take a step in a cyclic fashion (for instance, process A is waiting for process B to take a step, while B itself is waiting for A). As neither of them can make any progress, they are stuck forever. Deadlocks can occur as soon as one has the ability to use nested CCR. Consider the following example:

```
with r1 { with r2 { skip }} ‖ with r2 { with r1 { skip }}
```

Suppose now that the process on the left successfully enters the first critical region corresponding to `r1` before being interrupted. The second process can then enter its first critical region corresponding to `r2` which has not yet been claimed by the first process, but cannot enter the next one corresponding to `r1`. The first process is now stuck as well, and in fact the whole program is stuck in a deadlock.

The usual solution to avoid this kind of deadlocks is to assume an order that must be followed whenever a process acquires resources, for instance that one should never attempt to acquire `r2` before `r1`. This would forbid the problematic program above.

Locks Locks can simply be viewed as monitors where the beginning and end of critical regions are no longer statically determined, but are instead made operationally explicit through the operations `acquire(l)` and `release(l)`, where `l` is a *lock identifier*. In this sense, they are more general and allow more complex behaviours than monitors. Moreover, in most settings, one can dynamically allocate new locks on the heap, and hence have an unbounded number of them, which complicates the analysis. On the other hand, this allows fine-grained synchronisation, for instance by associating each node of a linked list to a different lock, and then manipulating the list concurrently with an hand-over-hand locking scheme [HS08]. Although seemingly more complex, some of these programs are also amenable to compositional reasoning with separation logic, for instance using the work of Gotsman *et al.* [GBC+07], or the marriage of rely-guarantee reasoning with separation logic of Vafeiadis and Parkinson [VP07].

In the parallel increment example, one could rewrite each side of the parallel composition using locks to protect the increments (assuming a previously declared lock `l`), although monitors are sufficient, as seen above:

```
acquire(l);
x++;
release(l);
```

Signals Another reading of locks is as *wait* and *signal* primitives: acquiring a lock is similar to waiting for this lock to be free, and releasing it is similar to signaling whomever is waiting for it that it is now free. Signals are similar to locks in this regard, but they may get lost when nobody is waiting for them.

Signals are used through two primitives: `signal` and `wait`. They take as argument an identifier, much as CCR and locks do. When calling `wait(c)`, a process is put to sleep until another process calls `signal(c)`. Signals that were issued while nobody had been waiting for them are discarded, so the moment at which signals are fired can be crucial.

Let us illustrate this definition with the *producer-consumer* problem, a classic problem in concurrency. The scenario is as follows: two processes share access to a finite buffer in memory. One of them, the *consumer*, pops elements from the buffer while the other, the *producer*, puts new elements into it. The consumer process should not attempt to receive from the buffer if it is empty, nor should the producer process try to enqueue something if the buffer is already full. To avoid these situations, they must synchronise.

A solution with two signals, `non_empty` and `non_full`, is presented in Figure 1.2. The example assumes that the buffer can be accessed through the `pushbuffer(x)` and `x = popbuffer()` operations, that respectively enqueues the value of `x` into the buffer and dequeues the first value in the buffer and stores it into `x`. The producer can choose a new value to enqueue using any function `produce_item` (provided that this function does not introduce bugs of its own, or attempts to wait or signal `non_empty` or `non_full`). The capacity of the buffer is `BUFFER_SIZE`, and the number of items currently in the buffer is `items` (initially 0).

This implementation may deadlock, as a result of the following race condition between the producer and the consumer. Suppose that the consumer process notices that the buffer is empty and gets ready to sleep; but before the `wait(non_empty)` command is issued, the producer process is scheduled and makes a whole pass: it enqueues an item, increments `items`, and, seeing that the buffer is empty, signals the consumer. However, since the consumer has not yet gone to sleep, the signal is lost forever; the consumer will then call `sleep`, never to be awaken by the producer. What can only happen then is that the producer fills up the entire buffer and puts itself to sleep. Both processes are now waiting for each other to issue a signal; they are deadlocked.

Using conditional critical regions instead of signals, one can write the simpler solution depicted Figure 1.3. This solution is race and deadlock free and even avoids the race not

Consumer code:

```
while (true) {
  if (items == 0)
    /* the buffer is empty, wait */
    wait(non_empty);

  x = pop_buffer();
  items--;

  if (items == BUFFER_SIZE - 1)
    /* the buffer was full but is not anymore */
    signal(non_full);

  /* do something with x */
}
```

Producer code:

```
while (true) {
  if (items == BUFFER_SIZE)
    /* the buffer is full, wait */
    wait(non_full);

  x = produce_item();
  push_buffer(x);
  items++;

  if (items == 1)
    /* the buffer was empty but is not anymore */
    signal(non_empty);
}
```

Figure 1.2: A solution to the producer-consumer problem using signals, with a potential deadlock.

mentioned above between the `items++` and `items--` instructions, that the astute reader may have noticed in Figure 1.2.

Readers-writers problem Before closing this section, let us present another classical problem, that embodies another kind of concurrency idiom: the readers-writers problem. This problem consists in orchestrating accesses to a shared portion of state that some processes (the readers) wish to read, while others (the writers) wish to write. To avoid races, only other readers may access the shared state when it is being read, and nobody can access it when it is being written.

A solution with two locks is presented Figure 1.4. Note that, contrarily to the producer-consumer problem, any number of processes executing the code for readers can run in parallel with any number of processes executing the code for writers. This solution is race and deadlock free, but exhibits a new kind of misbehaviour: it can be the case that readers

Consumer code:

```
while (true) {
  with buf when (items > 0) {
    x = pop_buffer();
    items--;
  }
  /* do something with x */
}
```

Producer code:

```
while (true) {
  with buf when (items < BUFFER_SIZE) {
    x = produce_item();
    push_buffer(x);
    items++;
  }
}
```

Figure 1.3: A solution to the producer-consumer problem using CCR.

Reader code:

```
while (true) {
  acquire(r);
  readers++;
  if (readers == 1)
    acquire(w);
  release(r);

  /* read here */

  acquire(r);
  readers--;
  if (readers == 0)
    release(w);
  release(r);
}
```

Writer code:

```
while (true) {
  acquire(w);

  /* write here */

  release(w);
}
```

Figure 1.4: A solution to the readers-writers problem using locks.

keep on interleaving their reads while never releasing the writing lock `w`, thus preventing the writers processes to ever execute. One should thus modify this program to ensure a *fairness* condition that forbids such executions. This *progress* property is one of *liveness* ("something good eventually happens"), as opposed to the problems described before, which were all about *safety* properties ("something bad should never happen"). This thesis will only be about safety properties.

1.1.4 Ownership

One way to distinguish between racy and race-free programs is to see racy programs as those violating the following *ownership hypothesis*, as formulated by O'Hearn in the paper "Resources, concurrency and local reasoning" [O'H07]:

> **Ownership Hypothesis** A code fragment can access only those portions of state that it owns.

In this hypothesis, a portion of state contains variables and memory locations and "own" means either that the program has shared access to it if it only ever reads from it, or exclusive access if it writes to it. It should not be confused with the portion of state reachable from the program variables: it can be a mere fraction of it, or even contain unreachable locations (which is usually a sign that the program has a memory leak: it is in charge of a part of the memory that it is unable to access). One can imagine each variable and memory location to be decorated with the identifier of the process (or processes) that owns it, and that a process accessing a cell that it does not own provokes a runtime error. If all processes abide by the ownership hypothesis, they will never run into such an error. Of course, this decoration should only be, to paraphrase O'Hearn, "in the eye of the asserter": in reality, it has no operational meaning. It is a useful proof artifact nonetheless. We shall give formal definitions of these concepts in Chapters 4 and 6.

If all the components of a program respect the ownership hypothesis, then the following *separation property* (from the same paper by O'Hearn) holds, where one can think of a grouping of mutual exclusion as a particular lock or critical region identifier:

> **Separation property** At any time, the state can be partitioned into that owned by each process and each grouping of mutual exclusion.

A crucial concept of the ownership reading of the heap is that the partition of the state described by the separation property is not *fixed*, but rather can dynamically change throughout the execution of the program. Theses changes occur either when a process allocates or deallocates portions of the state that it owns, or at synchronisation points. Thus, acquiring a lock or a resource results in taking ownership of the portion of state attached to it and adding it to the portion of state owned by the process. Symmetrically, releasing a lock or exiting a critical region corresponds to gouging a portion of the owned state of the process to give to the corresponding resource. As we will see in this thesis (and especially in Chapter 5), the same reading can be given for message passing: sending a message corresponds to a loss of ownership, and receiving a message corresponds to a gain.

The idea that one should be able to attribute each memory location to at most one process at a time can also be found in the very design of the Singularity OS [HL07]. Each Singularity process owns a portion of the state that is called *local* to the process and that can only be accessed by it or be given back to the operating system. In particular, local states cannot be shared between processes. Memory locations that are destined to navigate between processes are allocated in the *exchange heap*. Its functioning is described as such in the paper "Language support for fast and reliable message-based communication in Singularity OS" by Fähndrich *et al.* [FAH$^+$06], where Singularity channels are introduced:

> Although all processes can hold pointers into the exchange heap, every block of memory in the exchange heap is owned (accessible) by at most one process at any time during the execution of the system. Note that it is possible for processes to have dangling pointers into the exchange heap (pointers to blocks that the process does not own), but the static verification will prevent the process from accessing memory through dangling references.
>
> To make the static verification of the single owner property of blocks in the exchange heap tractable, we actually enforce a stronger property, namely that

> each block is owned by at most one thread at any time. The fact that each block in the exchange heap is accessible by a single thread at any time also provides a useful mutual exclusion guarantee.

This quote also highlights the idea that the separation has no operational meaning: it is enforced by the Sing♯ compiler so that runtime checks for separation (or *process isolation* in their own words) are not needed.

1.1.5 Memories

Explicit deallocation When considering memory-manipulating languages, two choices are possible with respect to how memory is deallocated. This task can either be performed explicitly by the programmer, or be delegated to a *garbage collector* that periodically frees unused memory. Correctly disposing of unused memory cell is important for the well-being of a system. Indeed, and although for simplicity we will model it as infinite in capacity, memory is a limited resource. Thus, constantly allocating new cells that are never freed can exhaust this resource. Failure to dispose of unused memory cells results in *memory leaks*.

In many programming languages memory leaks are not a concern because deallocation is handled *implicitly*, by a garbage collector. To give a few examples, this is the case for OCaml, Java, C♯, and most scripting languages like Perl and Python.

On the other hand, explicit deallocation is present in many so-called *low-level* programming languages, like C, C++ and assembly languages. It is a desirable feature if memory is scarce, for instance in embedded systems, or for memory-intensive applications wherein the usage of memory should be optimised, or finally for software that cannot bear the cost of having a garbage collector running in the background for performance reasons. Correct handling of explicit deallocation is also a crucial concern in cryptography, where sensitive data residing in clear text in memory cannot be handed back to the operating system by a mere deallocation. Otherwise, an attacker might be given the same memory cells by the memory manager and read the secrets. Instead, extra cleanup is required every time such a resource is to be deallocated, which forbids the use of a traditional garbage collector, at least for these resources.

In this work, we use a non-garbage-collected language, thus we are also interested in proving the absence of memory leaks. Furthermore, we claim that our techniques can be adapted to handle garbage-collected languages. This hope is backed by the fact that separation logic, on which our own analysis is based, has been successfully adapted to handle garbage collection in the past [CO01].

Ownership tracking The presence of the heap complicates the bookkeeping of what belongs to whom; for programs that only manipulate variables, it is easy to state that two programs operate on disjoint portions of the state: one simply has to check that they use different variable identifiers. This is not possible anymore for memory-manipulating programs, because of *aliasing*: two syntactically distinct variables may point to the same location in memory during the execution. This problem has crippled the verification of memory-manipulating programs for years before it found an elegant solution with separation logic:

most of previous efforts made the non-aliasing constraints explicit, resulting in a blow-up of the formulas that were used as specifications compared to their variable-only counterparts.

1.2 Message passing

When two processes wish to communicate via asynchronous message passing, they must share a *buffer*. One process may then send, or queue messages on the buffer, that the other one can receive, or dequeue. Our communication model is heavily inspired from the one of the Singularity OS [FAH+06]: communication buffers are grouped pairwise into bi-directional, asynchronous *channels*. Access to a channel is performed via two *endpoints* on which one may both send to a buffer and receive from the other, achieving bi-directional communication. A similar model of communication is the IPC (inter-process communication) socket model found in many operating systems. In our setting, the communications are moreover *copyless*: sending a message does not result in copying its contents over the channel, but rather in sending a mere pointer to the location of the message's data in memory. This assumes a shared memory between processes, which can be achieved for instance if they are run on the same machine.

We first introduce informally the communication primitives of our message-passing language. We then discuss the design choices that have been made, and the problems of interest related to the verification of message-passing programs.

1.2.1 Communication primitives

Let us describe informally the communication primitives we have adopted in this thesis. The primitives for opening a channel and sending and receiving are direct adaptations of the corresponding primitives in Singularity, while closing is performed differently in our setting.

Opening a channel Opening a channel allocates two endpoints that are each other's *peer* (or *mate*) and have initially empty buffers.

Sending and receiving messages Each send and receive instruction of our language specifies a message *identifier*, or *tag*, for the kind of message that is sent or that is expected on the channel. These tags can be thought of as representing the type of what is being transferred. Receiving is only possible via a `switch receive` construct. Each branch of this construct specifies an endpoint, a tag, and a continuation. The semantics of `switch receive` is to select one of the branches that are ready to execute given the current contents of the various endpoints' buffers, perform the corresponding receive and call its continuation. If none of the branches is ready to execute, it sleeps until a new message arrives on one of the empty buffers. Sending, on the other hand, is asynchronous and always succeeds.

When there is only one branch in a `switch receive`, for instance

```
switch receive { x = receive(a,e): p }
```

```
message a;

put_val(e,x) {
  send(a,e,x);
}

get_val(f) {
  y = receive(a,f);
  if (y == 42) {
    print("The answer is 42.");
  } else {
    print("What was the question again?");
  }
}
```

```
main() {
  local e,f;

  (e,f) = open();
  put_val(e,42) || get_val(f);
  close(e,f);
}
```

Figure 1.5: Value passing.

where p is a program, `x` and `e` are variables, and `a` a message *tag*, we simply write

```
x = receive(a,e); p
```

Closing a channel Finally, closing a channel takes as arguments two expressions pointing to the locations of both endpoints of a channel, and disposes of them.

1.2.2 Examples

In the examples that follow, we will use function calls, which are not part of the syntax of our programming language. However, they can be simulated by inlining the code at the call sites, provided that we are careful not to use mutually recursive functions. We use `e`, `f`, … for variables pointing to endpoints and `x`, `y`, … for variables containing integers or pointers to regular memory cells.

The program of Figure 1.5 exchanges an integer value between two threads `put_val` and `get_val` by passing a message tagged by `a` (declared in the preamble of the program – in Sing♯ `a` would be annotated with a type, for instance `int`), and then proceeds to close the channel. The variable `y` is global. Except for the printing operations, it is equivalent to the program below:

```
local e,f;

(e,f) = open();
send(a,e,42);
y = receive(a,f);
close(e,f);
```

It is also equivalent to simply setting `y` to 42:

```
y = 42;
```

```
put_cell(e,x) {                  main() {
  send(cell,e,x);                  local e,f;
}
                                   x = new();
get_cell(f) {                      (e,f) = open();
  y = receive(cell,f);             put_cell(e,x) || get_cell(f);
  dispose(y);                      close(e,f);
}                                }
```

Figure 1.6: Cell passing.

The value of a message can also refer to an allocated location in memory, in which case the ownership of it may be transferred from the sender process to the receiver process. The program of Figure 1.6 exchanges a memory cell between two threads `put_cell` and `get_cell` by passing a message `cell`, and then proceeds to close the channel. As `get_cell` gets ownership of the memory cell, it may modify it at will, in particular it may safely dispose of it, as in the example. If `put_cell` tries and dereferences `x` after the send, a race can occur: `get_cell` may have already freed it, which would result in a memory violation (for instance a segmentation fault or a bus error). Notice that the fact that ownership is transferred is not enforced by the syntax; we could very well replace `put_cell` by

```
send(cell,e,x);
dispose(x);
```

and `get_cell` by `y = receive(cell,f);` without changing the overall effect of the program or introducing a race condition. In other words, what is transferred operationally are mere values, and the meaning we attach to them is an artifact of the informal proof of correctness of the program that we have in mind.

Endpoints themselves can be sent over channels, in the sense that one may send the location of an endpoint on a channel, thus letting the receiving process know of the location of this endpoint, which makes it possible for this process to use it to send and receive over it. As we consider only point-to-point communications (see next section), concurrent accesses to the same endpoint are forbidden. This usually means that further access to a cell or endpoint location sent over a channel is forbidden to the sending process, as the receiving one may now interact with them. In other words, as already mentioned before, ownership of the contents of a message is lost upon sending (and granted upon receiving).

Using endpoint transfer, it is possible to transfer a value indirectly over a channel by encapsulating it inside another channel, as in Figure 1.7. In this example, the address of a memory cell is sent over a private channel, the receiving endpoint of which is sent to another process. This process retrieves the endpoint, and can then receive the desired address from it. This program has a memory leak: it fails to dispose the channel (`e_cell`, `f_cell`).

Another use for endpoint passing, mentioned earlier and pictured in Figure 1.8, is for channel closing: the example of Figure 1.6 can be extended by sending `e` over itself after sending `x`, so that `get_cell2` can close the channel (`e`, `f`). The same extension is possible

```
put_cell_indirect(e,x) {
  local e_cell,f_cell;

  (e_cell,f_cell) = open();
  send(cell,e_cell,x);
  send(ep,e,f_cell);
}

get_cell_indirect(f) {
  local y,ff;

  ff = receive(ep,f);
  y = receive(cell,ff);
  dispose(y);
}
```

```
main() {
  local x,e,f;

  x = new();
  (e,f) = open();
     put_cell_indirect(e,x)
  || get_cell_indirect(f);
  close(e,f);
}
```

Figure 1.7: Indirect cell passing through a secondary channel.

```
put_cell2(e,x) {
  send(cell,e,x);
  send(fin,e,e);
}

get_cell2(f) {
  local y,ee;

  y = receive(cell,f);
  ee = receive(fin,f);
  close(ee,f);
}
```

```
main() {
  local x,e,f;

   x=new();
   (e,f)=open();
   put_cell2(e,x) || get_cell2(f);
}
```

Figure 1.8: Cell and endpoint passing.

for the program of Figure 1.7 to avoid the memory leak spotted above.

Let us give a final version of the program that sends a whole linked list starting at x cell-by-cell through the channel in Figure 1.9. The receiving process (get_list) cannot know anymore when the fin message will arrive, so we use the switch receive construct between the two possible messages cell and fin, which will select the appropriate branch depending on which one is available in the buffer. The main function is as in Figure 1.8. We write [x] for the dereferencement of the pointer contained in x. In this example, we ignore the data that may be contained in the list, hence each node of the list simply points to its successor in the list.

A particularity of the copyless message passing setting is that sending x before dereferencing it in the example above (that is, moving send(cell,e,x); one line earlier) could result in a fault, as the cell pointed to by x is not owned by this thread after it has been sent,

```
put_list(e,x) {          get_list(f) {
  local t;                 local x, ee;
  while (x != 0) {
    t = [x]; /* tail */    ee = 0;
    send(cell,e,x);        while (ee == 0) {
    x = t; }                 switch receive {
  send(fin,e,e);             x = receive(cell,f): { dispose(x); }
}                            ee = receive(fin,f): {}
                           }}
                           close(ee,f);
                         }
```

Figure 1.9: Cell-by-cell list passing.

and hence may have been deallocated in between the two instructions.

The linked list can also be sent in one single message, as once a process knows where the head of the list is all the other cells become reachable. The resulting program would then be similar to the one of Figure 1.8. This is another example where the meaning attached to messages of a program depends on what the program does. The same message may indicate a transfer of ownership of a single memory cell or of a whole linked list.

1.2.3 Design choices

Asynchronous or synchronous Communications may be either asynchronous or synchronous; in the latter case, the sending of a message must wait until the other party is ready to receive, and this exchange acts as a *synchronisation* point between the two, whereas in the case of asynchronous message passing the sending operation can return immediately. In this case, the maximal size of what can be stored in the channel matters, because a possibly unbounded number of messages can be sent before they begin to be received. If this number exceeds the capacity of the channel, the send operation can either block until room is made for new incoming messages, or fault, depending on the implementation. Synchronicity can be retrieved from an asynchronous model by making each receive be followed by an acknowledgement message. In this work, we consider asynchronous communications with unbounded buffers, to simplify the model. We will provide in Chapter 3 a sufficient condition to ensure that buffers do not grow past a certain size.

Similarly, the receive operation blocks when the buffer is empty, whereas a more realistic setting should also include a non-blocking receive, or one with an optional timeout, that would return a special value should the buffer remain empty after some time has elapsed. We believe that the work presented in this thesis could easily be extended to include such behaviours: as the `switch receive` construct of our language allows a program to wait for several kinds of messages on several channel endpoints simultaneously, non-blocking receives might be modelled by waiting on an additional channel for a "timeout" message delivered periodically by an auxiliary process.

Finally, there also exists a more extreme case of asynchronicity, for instance in MPI (the message passing interface [3]), where sending returns even before the message has been fully sent. In this case, there is usually also a primitive to wait for the asynchronous send to finish. In our setting, sends are atomic in the sense that they return only once the message has been put in the receive buffer of its recipient.

Recipients of messages When sending messages, one has to specify who the recipient of the message is. This can be done either explicitly by naming the recipient (in which case the name of the channel is implicit), for instance by its process identifier as it can be done in MPI, or implicitly by naming the channel instead. We support the latter form with our channel model: sending and receiving are performed on any of the two endpoints of a channel, irrespective of whom they belong to.

Message tags and branching receives When a channel may carry messages of different sorts, one is usually faced with having to take different actions depending on the kind of message that is present at the head of the buffer. One means of discrimination between such messages is to associate "tags" to each of them that unambiguously determine their type. A language construct may then be used to discriminate messages according to their tags and take different actions for different tags. In addition to our own setting, this solution has been chosen by Singularity, session types, and the ERLANG programming language, to name a few. We have adopted the `switch receive` construct from Singularity.

One-to-one, one-to-many, many-to-one In some cases, one may wish to broadcast a message to a set of recipients instead of sending the same message over several individual channels. Practically, this could mean being able to duplicate an endpoint between several participants, each allowed to perform only receives on it. We only support point-to-point communications in this work. Moreover, we ensure a linear usage of channels: sharing of endpoints is disallowed and only one process at a time may attempt to send or receive on a particular endpoint.

Life and death of channels In our setting, channels can be dynamically allocated, disposed of, and individual endpoints can be passed around through other channels, thus allowing the topology of the interconnection network between processes to change during the execution. Acknowledging the fact that endpoints are allocated on the heap allows for a flexible handling of channels, one benefit of which being that they may be stored inside arbitrarily complex mutable data structures.

Protocol specification It is often desirable to be able to describe what *protocol* a given channel follows, that is which sequence of messages can be expected on this channel. This can help in proving the safety of communications on any channel following this protocol for instance. For this purpose, we have chosen to borrow the notion of *contracts* from the

[3] `http://www.mpi-forum.org/`

Singularity project [FAH+06]. Session types [THK94] represent another proposal to structure communications, and have been implemented as an extension of JAVA called SESSION JAVA [HYH08].

Ordering of messages and reliability of channels When the two participants of a communication are physically apart, for instance in the case of two machines communicating over the Internet, implementation choices have to be made that may no longer ensure that messages arrive in the order they were sent or that they will arrive at all. The Internet Protocol (IP), for instance, ensures nothing of the sort, and packets may even arrive twice or be corrupted during transport. Most of the time, an additional layer, for instance the Transmission Control Protocol (TCP), controls such bad behaviours and restores the first-in first-out (FIFO) ordering of messages as well as their integrity.

As we are interested in communications that occur mostly within a single machine, for instance in low-level operating system code, we choose not to concern ourselves with what could happen on longer-distance communications and consider only reliable channels with a FIFO ordering of messages.

Closing a channel It is common in message-passing languages to close a channel by closing each endpoint separately. For instance in SING♯, closing an endpoint on its own results in a special message ChannelClosed being sent to its peer. The other endpoint can continue sending on the channel, even after receiving the closing message. Once both endpoints have sent the ChannelClosed message, the channel is effectively closed.

This approach can easily be simulated in our setting. In fact, in several of the examples we have given, for instance those of Figures 1.8 and 1.9, this is how the channels were closed: the program implementing one side of the communication sends its endpoint over itself at the end of the communication (for example, `send(fin,e,e)`, where `e` points to said endpoint, and fin is the label chosen to indicate the end of communications). The peer program can then receive this endpoint through its own endpoint, and thus close the channel since it now holds both of its endpoints.

1.2.4 Problems of interest

In Chapters 3 and 5 we will introduce mechanisms to prove certain safety properties about programs that are specific to message passing. Let us informally discuss them.

Boundedness For asynchronous communications, a buffer is used to hold pending messages (messages that have been sent and are waiting to be received). It is often interesting to know whether a *bound* can be put on the maximum number of pending messages of all the executions of a program, hence on the size of the buffer, in which case communications are said to be *bounded*. The communications of the list sending program (Figure 1.9) are not bounded, because arbitrarily many cells may be sent (depending on the length of the list) before the receiving process catches on. We can write another version of this program where the size of the buffers is bounded by 1 by adding acknowledgement messages after each send: `receive(ack,e);` is added after `send(cell,e,x)` in `put_list`, and the corresponding branch of the `switch receive` of `get_list` becomes:

```
x = receive(cell,f): { send(ack,f); dispose(x); }
```

In fact, adding acknowledgement messages is a general recipe to bound the size of all buffers by 1, and to mimic the synchronous setting.

The Singularity OS, for instance, insists on all communication buffers being bounded, by a bound that is known at the time the program is compiled [FAH⁺06]. Indeed, the operating system itself uses message passing for communications, and it must be able to operate even in an out-of-memory context (when all the available memory has been allocated, for instance because of a memory leak in one of the running programs; in this case, the faulty program is generally terminated by the kernel and its memory reclaimed). Were the buffers unbounded, they would have to be implemented as dynamic data structures, for instance as linked lists, and sending a message would need to allocate a new node in this list. This is not possible in an out-of-memory context. On the other hand, if a bound is known on the size of a buffer, it can be implemented by a fixed-size array allocated once and for all, and the channel will be able to operate without requiring any additional memory.

Deadlock-freedom Communications, like all means of synchronisation, can introduce deadlocks. A communication deadlock happens when two or more processes are blocked waiting for each other to send a message. We can also call deadlock a situation where a process is waiting for messages of some kind on an endpoint, but either another type of message arrives, or no message at all. For instance, assuming that $(\texttt{e}, \texttt{f})$ and $(\texttt{ee}, \texttt{ff})$ form two channels, the two program snippets below can and will deadlock if executed with empty initial buffers:

```
send(a,e);  ‖ switch receive { receive(b,f): {} receive(c,f): {} }
```

The program above deadlocks because the process on the right is waiting for b or c, but a is sent. The program below deadlocks because it waits for messages on the wrong channel (it may not deadlock if another process sends a message `a` on `ee` at some point):

```
send(a,e);  ‖ receive(a,ff);
```

The program below shows a more intricate example of deadlock:

$$\left\{\begin{array}{l}\texttt{receive(b,e);}\\ \texttt{send(a,e);}\end{array}\right\} \parallel \left\{\begin{array}{l}\texttt{receive(a,f);}\\ \texttt{send(b,f);}\end{array}\right\}$$

The deadlock disappears if the order of sends and receives is permuted on either side of the parallel composition (or on both sides):

$$\left\{\begin{array}{l}\texttt{send(a,e);}\\ \texttt{receive(b,e);}\end{array}\right\} \parallel \left\{\begin{array}{l}\texttt{receive(a,f);}\\ \texttt{send(b,f);}\end{array}\right\}$$

Fault-freedom One of the examples of the previous paragraph requires more attention, namely the first one, in which the message at the head of the buffer is not one of the messages expected by the switch receive. This can be seen as an erroneous communication: either the send should not be a a message, or the `switch receive` should include another case for this message. As it is, it corresponds to what Cécé and Finkel call an *unspecified reception* in the paper "Verification of programs with half-duplex communication" [CF05]:

> An unspecified reception configuration corresponds to a configuration in which a machine is definitively blocked by its inability to receive the head messages present in its input channels.

This kind of deadlock is also considered as disastrous by the session type community: a key result in this area is that well-typed programs do not exhibit this error [THK94].

Finally, in Sing♯, that inspired our own language, a `switch receive` would fault in this case [FAH^{+}06]:

> If the message at the head of the buffer is not the expected message or the channel is closed by the peer, the receive fails.

Interestingly enough, the solution first proposed by Singularity to eliminate these deadlocks was unsound, as shown by Stengel and Bultan [SB09], who corrected it independently from our own work (which came to the same conclusion and solution, presented in Chapter 3).

Leak-freedom When a communication channel is closed, nothing guarantees that its buffers are empty. As a result, the following program leaks a memory cell that could have been retrieved from the channel, had it not been closed abruptly:

```
(e,f) = open();
x = new();
send(cell,e,x);
x = 0;
close(e,f);
```

Without garbage collection, this kind of error should also be eliminated. In Singularity, where the memory is garbage collected, nothing is done to guard against these programs.

Copyless communications Finally, one can wonder whether or not communicating in a copyless fashion introduces races or memory faults, as we have already seen in the examples that were presented.

1.2.5 Encoding locks and message passing into one another

Shared memory and message passing are, in all generality, equally expressive: one can transform any program using one of the two paradigms into an equivalent program using the other one. Locks can of course encode message passing; this is how message passing would be implemented in practise. Let us discuss here the other way around.

One way to simulate locks using our programming language is by using one central locking authority per lock identifier, by whom the other threads will be granted locks. Each thread holds a different endpoint to each of the locks authorities. Lock identifiers are thus replaced by endpoints to the relevant authority. A thread can acquire a lock by sending an acq message to the central authority and then waiting for an ack message. The lock is released by a rel message. This gives the following pattern, assuming that e is an endpoint through which the thread can communicate with the central authority:

```
send(acq,e,l);
receive(ack,e);
// Critical section here
send(rel,e,l);
```

The authority for a given lock is given by the following code, assuming two threads that send locking requests on $\mathtt{e}_1$ and $\mathtt{e}_2$:

```
while (true) {
switch receive {
  receive(acq,e1): {
    send(ack,e1);
    receive(rel,e1);
  }
  receive(acq,e2): {
    send(ack,e2);
    receive(rel,e2);
  }
}}
```

More robust solutions that could handle an unbounded number of locks and threads can be programmed if we extend the `switch receive` construct to wait on a *linked list* of endpoints. This extension exists for instance in Singularity.

1.2.6 Further examples of message passing

To conclude this chapter, let us show how message passing can be used to solve the producer-consumer and the readers-writers problems.

Producer-consumer As our setting embeds the concept of a buffer in the synchronisation primitives themselves, the producer-consumer problem has a very easy solution, presented Figure 1.10. Contrast it with the same solution using CCR (Figure 1.3, page 17). The message passing solution assumes a channel $(\mathtt{e}, \mathtt{f})$ between the producer and the consumer processes, over which messages of type item may be exchanged. Note that the communication is uni-directional, from the producer to the consumer. This solution is race free, fault free and deadlock free. A leak-free full-fledged version would require an extra message to signify the end of production. The communication buffer is unbounded, but as usual could be bounded by adding acknowledgement messages.

Readers-writers While the producer-consumer problem is easily solved using message passing because its solution essentially simulates message passing, this is not the case for the readers-writers problem. In fact, to implement a solution of the readers-writers problem using message passing, one essentially re-implements locks. The solution proposed

Consumer code:

```
while (true) {
  x = receive(item,f);

  /* do something with x here */
}
```

Producer code:

```
while (true) {
  x = produce_item();
  send(item,e,x);
}
```

Figure 1.10: A solution to the producer-consumer problem using message passing.

Reader code:

```
while (true) {
  send(read_request,e);
  x = receive(value,e);

  /* do something with x here */
}
```

Writer code:

```
while (true) {
  y = produce_item();
  send(write_request,e,y);
  receive(write_ack,e);
}
```

Figure 1.11: A solution to the readers-writers problem using message passing.

Figure 1.11 assumes that the endpoint e used in each instance of readers and writers is different each time. All the peers of these endpoints are held by a central authority which takes care of granting reading and writing rights only when mutual exclusion is ensured. We omit the code for this process. Note that the code for just the readers and writers is still simpler than its locking counterpart, but only because the mutual exclusion mechanism happens outside of it.

CHAPTER 2

A Model of Copyless Message Passing

This chapter describes the syntax and semantics of our message-passing programming language, and formalises some of the ideas introduced in the previous chapter. The semantics of the language is independent of contracts and logical specifications.

2.1 Programming language

2.1.1 Heap-manipulating Imperative Programming Language

Grammar Let us define HIP, the heap-manipulating imperative programming language. We assume that the following sets are defined:

- the countably infinite set Var of program variables, ranged over by `e`, `f`, `x`, `y`, …
- the countably infinite set Cell of memory locations, ranged over by l, …
- the countably infinite set Val of values, ranged over by v, …

We suppose that $\mathsf{Var} \cap \mathsf{Cell} = \mathsf{Var} \cap \mathsf{Val} = \varnothing$, $0 \notin \mathsf{Cell}$ and $\mathsf{Cell} \cup \mathbb{N} \subseteq \mathsf{Val}$. The grammar of expressions, boolean expressions, atomic commands and programs is given in Figure 2.1.

The syntax is defined algebraically to ease the definition of the operational semantics and of the logic. However, using `assume` and non-deterministic choice, one can recreate the usual `if` and `while` constructs of programming languages: we write `if` (B) p_1 `else` p_2 for

$$(\texttt{assume}(B);\ p_1)\ +\ (\texttt{assume}(\texttt{not}\ B);\ p_2)\ .$$

Likewise, `while` (B) p is syntactic sugar for

$$(\texttt{assume}(B);\ p)^*;\ \texttt{assume}(\texttt{not}\ B)\ .$$

E	::=		expressions
		`x`	variable
		$\mid v$	value
		$\mid E + E \mid E - E \mid \cdots$	arithmetic operations
B	::=	$E = E \mid B$ `and` $B \mid$ `not` B	boolean expressions
c	::=		instructions
		`skip`	inaction
		$\mid$ `assume(`B`)`	assume condition
		$\mid$ `x = `E	variable assignment
		$\mid$ `x = new()`	memory allocation
		$\mid$ `x = [`E`]`	memory lookup
		$\mid$ `[`E`] = `E	memory mutation
		$\mid$ `dispose(`E`)`	memory deallocation
p	::=		programs
		c	atomic command
		$\mid p;\ p$	sequential composition
		$\mid p \parallel p$	parallel composition
		$\mid p + p$	non-deterministic choice
		$\mid p^*$	iteration
		$\mid$ `local x in` p	local variable declaration

Figure 2.1: Syntax of the Hip programming language (x $\in$ Var and $v \in$ Val).

Records Although the syntax lacks a record model, we can simulate one by taking Loc to be the set of strictly positive integers and adding a size argument to the allocation command `new`. This allows programs to allocate consecutive cells in memory, which can be accessed through pointer arithmetic to simulate field accesses. For instance, one would allocate an element of a doubly linked list with `x = new(3)` and access its datum at `x`, its forward link at address `x` + 1 and its backward pointer at address `x` + 2.

Expressions could be extended with richer arithmetic operators (*e.g.* × or ÷), or other basic data types like strings. However, the fragment presented above is sufficient for our exposition.

HipCCR. Hip does not provide any concurrency primitives to synchronise or communicate between programs (other than sharing access to variables without any protection mechanism such as locks). HipCCR is an extension of Hip with constructs for declaring resources and using them inside conditional critical regions (CCR). It extends the grammar of Hip's

c	::=		commands
		$\cdots$	see Figure 2.1
		| `(e,f) = open()`	channel creation
		| `close(`E`,`E`)`	channel destruction
		| `send(a,`E`,`E`)`	labelled message sending
p	::=		programs
		$\cdots$	see Figure 2.1
		| g	guarded external choice
g	::=		guarded external choice
		`x = receive(a,`E`)` : p	guarded program
		| $g_1 \,\square\, g_2$	external choice

Figure 2.2: Syntax of HIPMP (a ∈ MsgId and e, f ∈ Var).

programs in the following way:

p	::=		programs
		$\cdots$	see Figure 2.1
		| `resource r in` p	resource declaration
		| `with r when (`B`)` p	CCR

Once a resource has been declared, it can be used to create conditional critical regions which will only execute if no CCR using the same resource is currently executing and if some boolean condition is satisfied. The boolean condition can be omitted (B = true in the grammar above), in which case we can simply write `with r` p.

2.1.2 HIPMP

Let us extend HIP with message-passing primitives, and call the resulting language HIPMP. We assume two extra countably infinite sets MsgId and Endpoint of respectively message identifiers ranged over by a, … and endpoint locations ranged over by ε, … The two sets are disjoint, and disjoint from previously defined sets, with the exception of Val. Although MsgId ∩ Val = ∅, endpoint locations are added as possible values: Endpoint ⊆ Val. Moreover, as for regular memory cells, $0 \notin$ Endpoint. The syntax of HIPMP is given in Figure 2.2.

The `switch receive` construct described in the previous chapter is replaced with its more algebraic counterpart: the guarded external choice. One can encode the former using the latter. For instance,

```
switch receive {
x = receive(cell,f): { dispose(x); }
e = receive(fin,f): {}
}
```

becomes

```
(x = receive(cell,f): dispose(x)) □ (e = receive(fin,f): skip)
```

2.2 Semantics

2.2.1 States

States consist of a *stack*, a *cell heap* that contains regular cells and an *endpoint heap* mapping endpoints to their peers and their receive buffers. We write $\sigma = (s, h, k)$ for elements of State defined below.

$$\mathsf{State} \triangleq \mathsf{Stack} \times \mathsf{Heap} \times \mathsf{EHeap} \qquad \mathsf{Stack} \triangleq \mathsf{Var} \rightharpoonup_{\mathit{fin}} \mathsf{Val} \qquad \mathsf{Heap} \triangleq \mathsf{Cell} \rightharpoonup_{\mathit{fin}} \mathsf{Val}$$

$$\mathsf{EHeap} \triangleq \mathsf{Endpoint} \rightharpoonup_{\mathit{fin}} \mathsf{Endpoint} \times (\mathsf{MsgId} \times \mathsf{Val})^*$$

As is standard for models of separation logic, we model the heap as a *finite partial map* (written “$\rightharpoonup_{\mathit{fin}}$”) from locations to values. The domain of the heap corresponds to the allocated addresses and, for simplicity, is unbounded in size, although only a finite number of cells may be allocated at any given point in time. Alternatively, one may think of the domain of the heap as the set of memory cells that are *owned* in the current state.

The novelty compared to usual models of separation logic is the *endpoint heap* k. The separation of dynamically allocatable resources into two distinct heaps is a purely cosmetic design choice that simplifies the presentation of the semantics, because each type of object has a different type of image by the heap and is used in different contexts (for instance, an endpoint cannot be freed using the `dispose` command, and a cell cannot be used to send messages). Thus, a cell can point to any value, but an endpoint points to its peer and to its buffer. The latter represents the pending messages waiting to be received on this endpoint: it is a word of pairs (a, v) where a is a message identifier and v a value. Equivalently, there could be only one heap that associates one of two sorts to each allocated locations, similarly to the work of Gotsman *et al.* on storable locks [GBC$^+$07] where the different sorts that coexist in the same one heap are one for locks and one for regular memory cells.

The stack follows the same idiom: variables must be owned by the program to be accessible without a risk of a race condition.

If $\sigma = (s, h, k)$, $\varepsilon \in \mathit{dom}(k)$ and $k(\varepsilon) = (\varepsilon', \alpha)$, ε' is called the *peer* (or *mate*) of ε and is denoted by $\mathit{mate}(k, \varepsilon)$ or $\mathit{mate}(\sigma, \varepsilon)$, and α is called the *buffer* of ε and is denoted by $\mathit{buffer}(k, \varepsilon)$ or $\mathit{buffer}(\sigma, \varepsilon)$.

Each state (s, h, k) should be *well-formed*, in the sense that the following axioms hold:

$$\forall \varepsilon \in \mathit{dom}(k).\, \mathit{mate}(k, \varepsilon) \in \mathit{dom}(k) \qquad \text{(Channel)}$$

$$\forall \varepsilon \in \mathit{dom}(k).\, \mathit{mate}(k, \varepsilon) \neq \varepsilon \qquad \text{(Irreflexive)}$$

$$\forall \varepsilon \in \mathit{dom}(k).\, \mathit{mate}(k, \mathit{mate}(k, \varepsilon)) = \varepsilon \qquad \text{(Involutive)}$$

The first condition ensures that if the domain of the endpoint heap contains one endpoint of a channel then it also contains its peer. As we will see in Section 2.2, this is required by the operational semantics since to send on one endpoint one needs to modify its peer's buffer.

The other two conditions ensure that channels are well-formed: endpoints are paired into bi-directional channels composed of two buffers.

2.2.2 Notations

The operational semantics is given as a set of rules that describe the reduction relation $\Rightarrow$ between a *configuration* p, σ where p is a program and σ a state, and either another configuration p', σ' or an error, written respectively $p, \sigma \Rightarrow p', \sigma'$ and $p, \sigma \Rightarrow \mathbf{error}$. Because we are interested in race-free programs, we will force racy ones to fault. Similarly, we force programs in an unspecified reception configuration to fault. This gives rise to two kinds of errors:

$\mathbf{OwnError}$ indicates an *ownership error*: the program has tried to access a resource it does not currently own, be it a variable, a memory cell or an endpoint;

$\mathbf{MsgError}$ indicates a *message error* during a reception: there is an unexpected message at the head of a receive queue, so the receiving operation fails.

We write $\mathbf{error}$ for one of $\mathbf{OwnError}$ or $\mathbf{MsgError}$ in the rules that propagate an error through programming constructs. We write $\Rightarrow^*$ for the reflexive and transitive closure of $\Rightarrow$.

Explicit error detection is a rather unrealistic aspect of the semantics, because threads would normally not fault when merely trying to access the same memory location, and may or may not fault when receiving an unexpected message depending on the implementation. They could however end up in an incoherent state in the case of a race, and as discussed in the previous chapter defining an exact semantics for this kind of behaviours is challenging to say the least. In this thesis, our goal is to certify that no such error happens, thus we make them explicit even in the semantics. This makes it easier to identify unsafe starting states for a program p: they are the states σ such that $p, \sigma \Rightarrow^* \mathbf{OwnError}$ or $p, \sigma \Rightarrow^* \mathbf{MsgError}$. This is a standard practise in models of separation logic [Bro07, COY07], that reflects its fault-avoiding interpretation of Hoare triples: a proved program never runs into an error state.

If a program p does not reduce into an error from a given state σ, the reduction rules give the continuation p' of p and the resulting state σ':

$$p, \sigma \Rightarrow p', \sigma'$$

If p' is `skip`, the program has successfully reached a final state σ'.

An expression E is associated to a value $[\![E]\!]_s$ by an evaluation that looks up the values of the program variables mentioned in E (the set of which is written $var(E)$) into the stack s. If E mentions a variable not in the domain of the stack, then $[\![E]\!]_s$ is undefined (this will be the cause of an ownership error in many cases of the semantics). The value of $[\![E]\!]_s$ is computed by structural induction on the expression:

$$[\![v]\!]_s \triangleq v \qquad [\![\mathtt{x}]\!]_s \triangleq s(\mathtt{x}) \qquad [\![E_1 + E_2]\!]_s \triangleq [\![E_1]\!]_s + [\![E_2]\!]_s$$

$$[\![E_1 - E_2]\!]_s \triangleq [\![E_1]\!]_s - [\![E_2]\!]_s \qquad \dots$$

Some of these operations are undefined if one of the subexpressions is not an integer (for instance for locations, although one may consider that locations as integers to allow pointer arithmetic, as discussed page 32).

Finally, given a function f, we write $[f \mid x : v]$ for the function f' that has the same domain as f, possibly increased by x if x was not in the domain of f, and such that $f'(x) = v$ and $f'(y) = f(y)$ for all $y \in \mathit{dom}(f) \setminus \{x\}$. We write $f \setminus S$ for the function f' defined on $\mathit{dom}(f) \setminus S$ such that f' and f coincide on this new domain. We further abbreviate modifications of the buffer of an endpoint by writing $[k \mid \mathit{buffer}(\varepsilon) {\leftarrow} \alpha]$ for $[k \mid \varepsilon : (\mathit{mate}(k, \varepsilon), \alpha)]$. Although we will not use it in this chapter, we also write $\varnothing$ for a function with empty domain and $[x_1 : v_1, \dots, x_n : v_n]$ for $[\varnothing \mid x_1 : v_1, \dots, x_n : v_n]$.

For concision, whenever possible, we describe all the cases where executing a command will produce an ownership violation together with the reduction where the command executes normally. We do so by putting the premises that are necessary for the command not to fault in $\boxed{\text{boxes}}$. A boxed premise indicates a reduction to **OwnError** from a state where the premise is either false or undefined.

Consider for instance the rule for memory lookup:

$$\frac{\boxed{\mathtt{x} \in \mathit{dom}(s)} \qquad \boxed{[\![E]\!]_s} = l \qquad \boxed{h(l)} = v}{\mathtt{x = [}E\mathtt{]}, (s, h, k) \Rightarrow \mathtt{skip}, ([s \mid \mathtt{x} : v], h, k)}$$

It indicates that $\mathtt{x = [}E\mathtt{]}$ will fault whenever $\mathtt{x}$ or one of the variables in E is not present in the current stack, or when E does not evaluate to an address present in the heap. Thus, this rule stands for the following three rules:

$$\frac{\mathtt{x} \in \mathit{dom}(s) \qquad [\![E]\!]_s = l \qquad h(l) = v}{\mathtt{x = [}E\mathtt{]}, (s, h, k) \Rightarrow \mathtt{skip}, ([s \mid \mathtt{x} : v], h, k)} \qquad \frac{\mathit{var}(\mathtt{x}, E) \not\subseteq \mathit{dom}(s)}{\mathtt{x = [}E\mathtt{]}, (s, h, k) \Rightarrow \mathbf{OwnError}}$$

$$\frac{[\![E]\!]_s = l \qquad l \notin \mathit{dom}(h)}{\mathtt{x = [}E\mathtt{]}, (s, h, k) \Rightarrow \mathbf{OwnError}}$$

2.2.3 Operational semantics

The reduction rules of the operational semantics are given Figures 2.3 to 2.6. Let us review the effect each command can have on a given state (s, h, k).

Figure 2.3 groups the stack commands.

- `skip` does not alter the current state.
- `assume`(B) faults if B cannot be evaluated in the current stack. Otherwise, it will either block or do nothing depending on the truth value of $[\![B]\!]_s$.

$$\frac{}{\texttt{skip}, \sigma \Rightarrow \texttt{skip}, \sigma} \qquad \frac{\boxed{[\![B]\!]_s} = \mathsf{true}}{\texttt{assume}(B), (s,h,k) \Rightarrow \texttt{skip}, (s,h,k)}$$

$$\frac{\boxed{\texttt{x} \in dom(s)} \qquad \boxed{[\![E]\!]_s} = v}{\texttt{x = } E, (s,h,k) \Rightarrow \texttt{skip}, ([s \mid \texttt{x} : v], h, k)}$$

Figure 2.3: Operational semantics of stack commands.

$$\frac{\boxed{\texttt{x} \in dom(s)} \qquad l \in \mathsf{Cell} \setminus dom(h) \qquad v \in \mathsf{Val}}{\texttt{x = new}(), (s,h,k) \Rightarrow \texttt{skip}, ([s \mid \texttt{x} : l], [h \mid l : v], k)}$$

$$\frac{\boxed{[\![E]\!]_s} = l \qquad \boxed{l \in dom(h)}}{\texttt{dispose}(E), (s,h,k) \Rightarrow \texttt{skip}, (s, h \setminus \{l\}, k)}$$

$$\frac{\boxed{\texttt{x} \in dom(s)} \qquad \boxed{[\![E]\!]_s} = l \qquad \boxed{h(l)} = v}{\texttt{x = [}E\texttt{]}, (s,h,k) \Rightarrow \texttt{skip}, ([s \mid \texttt{x} : v], h, k)}$$

$$\frac{\boxed{[\![E_1]\!]_s} = l \qquad \boxed{[\![E_2]\!]_s} = v \qquad \boxed{l \in dom(h)}}{\texttt{[}E_1\texttt{] = } E_2, (s,h,k) \Rightarrow \texttt{skip}, (s, [h \mid l : v], k)}$$

Figure 2.4: Operational semantics of heap commands.

- `x = `E faults if E cannot be evaluated in the current stack or if `x` is not present in the stack; otherwise, the value of `x` is updated to $[\![E]\!]_s$.

Cell manipulation commands are grouped in Figure 2.4.

- `x = new()` picks an unallocated address l and assigns a non-deterministic value to it. The variable `x` is updated to contain l. If `x` was not in the stack, the command faults.

- `dispose(`E`)` faults if either E cannot be evaluated on the current stack, or if $[\![E]\!]_s = l$ is not allocated in h. Otherwise, it removes l from the domain of h.

- `x = [`E`]` faults if either E cannot be evaluated on the current stack, or if $[\![E]\!]_s = l$ is not allocated in h, or if `x` is not present in the stack. Otherwise, the value of `x` is updated to the value contained at address l.

$$\frac{\boxed{e, f \in \mathit{dom}(s)} \qquad \varepsilon, \varepsilon' \in \mathsf{Endpoint} \setminus \mathit{dom}(k)}{\begin{array}{c}\texttt{(e,f)} \ = \ \texttt{open}(), (s, h, k) \Rightarrow \\ \texttt{skip}, ([s \mid e : \varepsilon, f : \varepsilon'], h, [k \mid \varepsilon : (\varepsilon', \text{\textvisiblespace});\ \varepsilon' : (\varepsilon, \text{\textvisiblespace})])\end{array}}$$

$$\frac{\boxed{[\![E_1]\!]_s} = \varepsilon_1 \qquad \boxed{[\![E_2]\!]_s} = \varepsilon_2 \qquad \boxed{k(\varepsilon_1) = (\varepsilon_2, \alpha_1)} \qquad \boxed{k(\varepsilon_2) = (\varepsilon_1, \alpha_2)}}{\texttt{close}(E_1,E_2), (s, h, k) \Rightarrow \texttt{skip}, (s, h, k \setminus \{\varepsilon_1, \varepsilon_2\})}$$

$$\frac{\boxed{[\![E_1]\!]_s} = \varepsilon \qquad \boxed{[\![E_2]\!]_s} = v \qquad \boxed{\mathit{mate}(k, \varepsilon)} = \varepsilon'}{\texttt{send}(\texttt{a},E_1,E_2), (s, h, k) \Rightarrow \texttt{skip}, (s, h, [k \mid \varepsilon' : \mathit{buffer}(\varepsilon') \leftarrow \mathit{buffer}(k, \varepsilon') \cdot (a, v)])}$$

$$\frac{\begin{array}{c}\boxed{\texttt{x}_1, \ldots, \texttt{x}_n \in \mathit{dom}(s)} \qquad \boxed{[\![E_1]\!]_s \in \mathit{dom}(k)} \mathrel{\&} \ldots \mathrel{\&} \boxed{[\![E_n]\!]s \in \mathit{dom}(k)} \\ [\![E_i]\!]_s = \varepsilon_i \qquad \mathit{buffer}(k, \varepsilon_i) = (a_i, v) \cdot \alpha\end{array}}{\mathop{\Box}\limits_{j=1}^{n} \texttt{x}_j \ = \ \texttt{receive}(\texttt{a}_j,E_j) : p_j, (s, h, k) \Rightarrow p_i, ([s \mid \texttt{x}_i : v], h, [k \mid \mathit{buffer}(\varepsilon_i) \leftarrow \alpha])}$$

$$\frac{\begin{array}{c}[\![E_1]\!]_s = \varepsilon_1 \mathrel{\&} \ldots \mathrel{\&} [\![E_n]\!]_s = \varepsilon_n \\ \exists j.\, \exists a.\, \exists v.\, \exists \alpha.\, \mathit{buffer}(k, \varepsilon_j) = (a, v) \cdot \alpha \mathrel{\&} (\forall j'.\, \varepsilon_{j'} = \varepsilon_j \Rightarrow a_{j'} \neq a)\end{array}}{\mathop{\Box}\limits_{j=1}^{n} \texttt{x}_j \ = \ \texttt{receive}(\texttt{a}_j,E_j) : p_j, (s, h, k) \Rightarrow \mathbf{MsgError}}$$

Figure 2.5: Operational semantics of communications.

- For [E_1] = E_2 to succeed, the stack must be able to evaluate E_1 and E_2, and $[\![E_1]\!]_s$ must be allocated in h. The mutation of this cell then takes place.

Communication commands are grouped in Figure 2.5.

- (e,f) = open() allocates two fresh endpoint locations in the endpoint heap k and sets their value so that they point to each other and have initially empty buffers (the empty word is written $\text{\textvisiblespace}$). It faults if either e or f is not allocated in the stack.

- close(E_1,E_2) checks that the values of E_1 and E_2 point to endpoints which are each other's peer, hence which form a channel. If this is the case, it deallocates them from the endpoint heap. Note that any message remaining in the queues is lost, which is often undesirable. Part of our analysis will try to ensure that it is never the case that messages are left in the queues when a channel is closed. If any of the conditions mentioned above is not satisfied, the command faults.

- send(a,E_1,E_2) looks up the value ε of E_1 in the current stack which must be an allocated endpoint address of k, the value v of E_2, then fetches the peer ε' of ε from

k to enqueue the pair (a, v) in the buffer of ε', resulting in $\alpha \cdot (a, v)$ (where $\cdot$ is the concatenation operation over words). The command faults if any of the lookups fails. Note that sending never blocks and, provided that appropriate resources are held, always succeeds.

- The case of receiving is more involved, since it may only be performed inside an external choice block.[1] The reduction rule selects an available message in one of the endpoints' buffers that is awaited by one of the receives. In other words, it selects a message (a_i, v) such that $\mathtt{x}_i$ `=` `receive`$(\mathtt{a}_i, E_i)$ `:` p_i is one of the guarded processes of the external choice. If it finds such a pair, it pops it from the corresponding buffer, and stores the value of the message in $\mathtt{x}_i$. If not, there are two possibilities: either the buffers of all the endpoints mentioned are all empty, in which case the receive blocks, or the pair at the top of one of the endpoints' buffers is an unexpected one. The latter case produces a **MsgError** error, even when other branches could execute successfully.

  Finally, if any of the $\mathtt{x}_j$ is absent from the stack, or if the expressions cannot be evaluated in the stack, or if they do not all correspond to allocated endpoint locations, the command faults and results in **OwnError**.

The semantics of programming constructs is given Figure 2.6. We write $\mathit{freevar}(p)$ for the set of *free variables* of p, which is defined as usual by structural induction on p as the set of variables `x` that appear in p outside of the scope of a `local x in` p construct.

For simplicity, the `local` construct is treated as a statically scoped allocation and deallocation of a fresh variable on the stack. For this purpose, a `delete` statement is appended at the end of the program, that deallocates the local variable. Since it only occurs in such instances, the variable will always be allocated when `delete` is called, hence we need not specify it as a requirement in the semantics. The `delete` command is not part of the programming language (in particular, it cannot be used by the programmer): it is only an artifact of the semantics.

The semantics of a parallel composition $p_1 \parallel p_2$ is all the possible interleavings of the actions of p_1 and p_2, possibly resulting in a fault if there is a race between the two programs. The purpose of the predicate $\mathit{norace}(p_1, p_2, \sigma)$ is to detect such races: it checks if there is an immediate race between p_1 and p_2, that is if p_1 and p_2 are both about to access to the same resource, and one of them is trying to write to it. Here a resource may be a stack variable, a memory location or an endpoint of a channel.

Given a program p and a state σ, we define the sets $\mathit{iw}(p, \sigma)$ and $\mathit{ir}(p, \sigma)$ of resources that are respectively *immediately* written to and read by p in the state σ. Immediately means that said resources will be accessed by the program at the very next step of its execution according to the operational semantics. For instance, `x =` E`;` p immediately writes to `x` and to `x` only regardless of what p is, while any program of the form $p_1 + p_2$ does not

[1] The way it is performed here is unfortunately not compositional, in that we describe only the reduction of all of its branches at the same time. In particular, we lack a rule that would describe the behaviour of $g_1 \Box g_2$ solely in terms of the behaviours of g_1 and g_2 considered independently. A modular description is in fact also possible, but would not be simpler, so we use the non-modular one in this thesis.

$$\frac{}{p_1 + p_2, \sigma \Rightarrow p_1, \sigma} \qquad \frac{}{p_1 + p_2, \sigma \Rightarrow p_2, \sigma}$$

$$\frac{p_1, \sigma \Rightarrow p_1', \sigma'}{p_1;\ p_2, \sigma \Rightarrow p_1';\ p_2, \sigma'} \qquad \frac{}{\texttt{skip};\ p_2, \sigma \Rightarrow p_2, \sigma} \qquad \frac{p_1, \sigma \Rightarrow \mathbf{error}}{p_1;\ p_2, \sigma \Rightarrow \mathbf{error}}$$

$$\frac{\boxed{norace(p_1, p_2, \sigma)} \qquad p_1, \sigma \Rightarrow p_1', \sigma'}{p_1 \parallel p_2, \sigma \Rightarrow p_1' \parallel p_2, \sigma'} \qquad \frac{p_1, \sigma \Rightarrow \mathbf{error}}{p_1 \parallel p_2, \sigma \Rightarrow \mathbf{error}}$$

$$\frac{\boxed{norace(p_1, p_2, \sigma)} \qquad p_2, \sigma \Rightarrow p_2', \sigma'}{p_1 \parallel p_2, \sigma \Rightarrow p_1 \parallel p_2', \sigma'} \qquad \frac{p_2, \sigma \Rightarrow \mathbf{error}}{p_1 \parallel p_2, \sigma \Rightarrow \mathbf{error}}$$

$$\frac{}{\texttt{skip} \parallel \texttt{skip}, \sigma \Rightarrow \texttt{skip}, \sigma} \qquad \frac{}{p^*, \sigma \Rightarrow \texttt{skip} + (p;\ p^*), \sigma}$$

$$\frac{\mathtt{y} \in dom(s)}{\texttt{delete(y)}, (s, h, k) \Rightarrow \texttt{skip}, (s \setminus \{\mathtt{y}\}, h, k)}$$

$$\frac{v \in \mathsf{Val} \qquad \mathtt{y} \notin dom(s) \cup freevar(p)}{\texttt{local x in}\ p, (s, h, k) \Rightarrow p[\mathtt{x} \leftarrow \mathtt{y}];\ \texttt{delete(y)}, ([s \mid \mathtt{y} : v], h, k)}$$

Figure 2.6: Operational semantics of programming constructs.

immediately writes to or reads any resource (it must first choose whether to execute p_1 or p_2).

Table 2.1 gives the sets $iw(p, \sigma)$ and $ir(p, \sigma)$ for all programs p and states σ. Let us denote witch $ia(p, \sigma)$ the immediate *accesses* of p from σ:

$$ia(p, \sigma) \triangleq iw(p, \sigma) \cup ir(p, \sigma)$$

The predicate $norace(p_1, p_2, \sigma)$ is defined as:

$$norace(p_1, p_2, \sigma) \triangleq \begin{cases} \mathsf{true} & \text{if } iw(p_1, \sigma) \cap ia(p_2, \sigma) = iw(p_2, \sigma) \cap ia(p_1, \sigma) = \emptyset \\ \mathsf{false} & \text{otherwise} \end{cases}$$

The semantics of all the other programming constructs is standard.

2.2.4 Properties of the operational semantics

Well-formed states In this subsection, we prove some properties that act as touchstones for the operational semantics presented above. First, the operational semantics does not create ill-formed states from well-formed ones.

Program	Immediate writes	Immediate reads
$\mathtt{assume(}B\mathtt{);}\ p$	$\varnothing$	$\mathit{var}(B)$
$\mathtt{x\ =\ }E\mathtt{;}\ p$	$\{\mathtt{x}\}$	$\mathit{var}(E)$
$\mathtt{x\ =\ new();}\ p$	$\{\mathtt{x}\}$	$\varnothing$
$\mathtt{x\ =\ [}E\mathtt{];}\ p$	$\{\mathtt{x}\}$	$\mathit{var}(E) \cup \{[\![E]\!]_s\}$
$\mathtt{[}E_1\mathtt{]\ =\ }E_2\mathtt{;}\ p$	$\{[\![E_1]\!]_s\}$	$\mathit{var}(E_1, E_2)$
$\mathtt{dispose(}E\mathtt{);}\ p$	$\{[\![E]\!]_s\}$	$\mathit{var}(E)$
$\mathtt{(e,f)\ =\ open();}\ p$	$\{\mathtt{e}, \mathtt{f}\}$	$\varnothing$
$\mathtt{close(}E_1\mathtt{,}E_2\mathtt{);}\ p$	$\{[\![E_1]\!]_s, [\![E_2]\!]_s\}$	$\mathit{var}(E_1, E_2)$
$\mathtt{send(a,}E_1\mathtt{,}E_2\mathtt{);}\ p$	$\{[\![E_1]\!]_s\}$	$\mathit{var}(E_1, E_2)$
$\Box_{i=1}^{n}\ \mathtt{x}_i\ \mathtt{=\ receive(a}_i\mathtt{,}E_i\mathtt{):}\ p_i$	$\{\mathtt{x}_i \mid 1 \le i \le n\}$ $\cup \{[\![E_i]\!]_s \mid 1 \le i \le n\}$	$\bigcup_{i=1}^{n} \mathit{var}(E_i)$
otherwise	$\varnothing$	$\varnothing$

Table 2.1: Sets of immediate writes and immediate reads of programs executing from a state (s, h, k).

Lemma 2.1 *For each well-formed state σ and program p, if there are p', σ' such that $p, \sigma \Rightarrow p', \sigma'$ then σ' is well-formed.*

Proof Assume that $p, \sigma \Rightarrow p', \sigma'$, that $\sigma' = (s', h', k')$ and that $\sigma = (s, h, k)$ is well-formed. We need to prove that σ' satisfies (Channel), (Irreflexive) and (Involutive). Since they are solely properties of endpoints, all the cases that do not modify the endpoint heap trivially produce well-formed states from well-formed states. Moreover, since the contents of the queue are not important in the axioms, the cases of `send` and `receive` are straightforward as well. This leaves us with `open` and `close`.

In the case of `open`, since the new endpoints ε, ε' are allocated in the endpoint heap at the same time and (Channel) held on σ, it still holds on σ'.

Suppose that (Irreflexive) does not hold anymore, and let ε be such that $\mathit{mate}(k', \varepsilon) = \varepsilon$. The misbehaving endpoint cannot be one of the newly allocated ones since they clearly do not violate this axiom, nor can it be one of the other endpoints since σ is supposed to be well-formed. This leads to a contradiction.

Similarly, (Involutive) only needs to be verified on the newly allocated endpoints, on which it does hold.

The case of `close` is straightforward, as it can be shown that any substate of a well-formed state is itself well-formed. □

Semantic equivalence of programs Let us now show some basic properties of the programming constructs. For instance, one should be able to prove that ";" is associative and that "+", "□" and "‖" are associative and commutative. This calls for a notion of program equivalence; we take as a basis for such a notion a coarse denotational semantics based on the set of reachable states.

Definition 2.1 (Program equivalence) *A program p denotes a partial function*

$$[\![p]\!] : \begin{array}{rcl} \mathsf{State} & \rightharpoonup & \wp(\mathsf{State} \uplus \{\mathbf{OwnError}, \mathbf{MsgError}\}) \\ \sigma & \mapsto & \{\sigma' \mid p, \sigma \Rightarrow^* \texttt{skip}, \sigma'\} \uplus \{\mathbf{error} \mid p, \sigma \Rightarrow^* \mathbf{error}\} \end{array}$$

Two programs p and q are equivalent, *written $p \sim q$, if $[\![p]\!] = [\![q]\!]$.*

Note that we only take into account final configurations $\texttt{skip}, \sigma$ and errors in this definition. The following lemma groups together some typical equivalences between programs.

Lemma 2.2 *For all programs p, p_1, p_2 and p_3, the following equivalences hold:*

$$p_1 \parallel (p_2 \parallel p_3) \sim (p_1 \parallel p_2) \parallel p_3 \qquad p_1 \parallel p_2 \sim p_2 \parallel p_1 \qquad \texttt{skip} \parallel p \sim p$$

$$p_1 + (p_2 + p_3) \sim (p_1 + p_2) + p_3 \qquad p_1 + p_2 \sim p_2 + p_1 \qquad p + p \sim p$$

$$p_1;\ (p_2;\ p_3) \sim (p_1;\ p_2);\ p_3 \qquad \texttt{skip}; p \sim p$$

Proof Straightforward by induction on the structure of the programs involved. □

Chapter 3

Dialogue systems

Communicating finite-state machines (CFSM for short) are a widely studied model of communicating systems and protocols in general. Dialogue systems are communicating systems consisting of two CFSMs communicating over a bi-directional FIFO channel modelled by two buffers. They can be used to specify the protocols followed by the communication channels of a message-passing program as defined in the previous chapter. In all generality, the buffers may be bounded or unbounded, perfect or lossy. We confine our study to unbounded perfect buffers, for the reasons detailed in Section 1.2.3.

Channel contracts are a special case of dialogue systems. They have been used to specify all the inter-process communications of the Singularity OS project. They were introduced by Fähndrich *et al.* in the paper "Language support for fast and reliable message-based communication in Singularity OS" [FAH$^+$06], along with sufficient syntactic conditions to ensure some safety properties. However, the lack of formal semantics lead to some bugs being discovered later on by Stengel and Bultan [SB09] in published contracts of Singularity. Whenever we use the term "Singularity contracts," and unless specified otherwise, we refer to contracts which satisfy the constraints imposed by Singularity in their fixed version by Stengel and Bultan.

We show that, thanks to these restrictions, Singularity contracts are free from reception errors, leaks and deadlocks, and have bounded communication buffers. Moreover, we show that they have the *half-duplex* property: only one channel may be non-empty at any given time in the execution of contracts. Figure 3.1 depicts the relationships between dialogue systems, half-duplex dialogue systems, contracts, and Singularity contracts.

In this chapter, we give a formal semantics to these objects, and study the decidability of the safety properties above, which include the ones sought after by Singularity. These properties are decidable for half-duplex dialogue systems, a result that follows directly from the fact that such systems have a decidable reachability set [CF05]. We will show that these properties are *undecidable* for contracts in general, and that they can be proved to always hold for Singularity contracts.

We will later use contracts as helpers to prove message passing programs in Chapter 5, and we will formally connect the properties of contracts to those of the programs that abide by them in Chapter 6.

We begin by a general presentation of communicating finite-state machines.

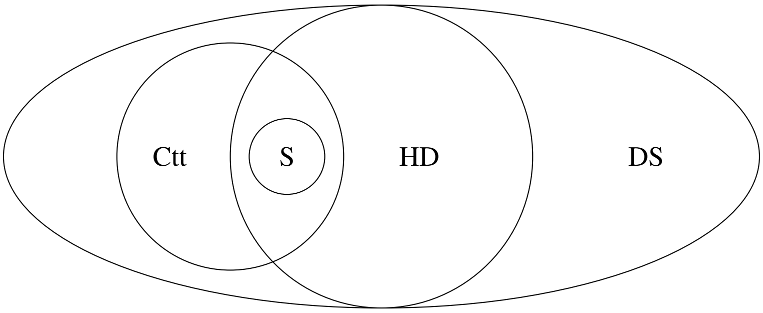


Figure 3.1: Relationships between dialogue systems (DS), half-duplex dialogue systems (HD), contracts (Ctt), and Singularity contracts (S).

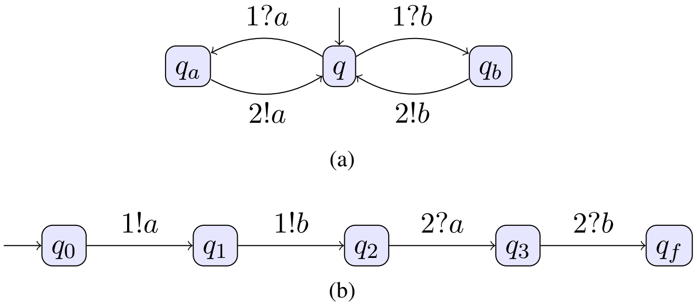


Figure 3.2: A pair of communicating machines.

3.1 Communicating systems

3.1.1 Definition

Figure 3.2 presents a communicating system made of two machines and two buffers (1 and 2). Figure 3.2a depicts a copycat machine that copies everything that is received on buffer 1 onto buffer 2; the CFSM of Figure 3.2b starts in state q_0, sends two messages a and b on buffer 1 and then receives them back on buffer 2. These two machines may interact; at the end of their interaction, the first one will be in state q again, the second one in q_f, and all the messages will have been consumed. As we will shortly see, (q, q_f) is a good candidate to be a *final state* of this system.

Definition 3.1 (Communicating finite-state machine) *A* communicating (finite-state) machine *(CFSM for short) is a 5-tuple* (Σ, B, Q, q_0, T) *where:*

- Σ *is a finite* alphabet *of messages;*

- B *is a finite set of* buffers*;*

- Q *is a finite set of* control states*;*
- $q_0 \in Q$ *is the* initial control state*;*
- $T \subseteq Q \times (B \times \{?, !\} \times \Sigma) \times Q$ *is a (finite) set of* transitions.

Given a CFSM $\mathfrak{M} = (\Sigma, B, Q, q_0, T)$*, we write* $init(\mathfrak{M})$ *for* q_0*.*

Notations 3.1 *In this chapter the set of the buffers of a machine will always be a finite subset of* $\mathbb{N}$*. We write* ‽ *for one of* $\{?, !\}$*. Elements of* $B \times \{?, !\} \times \Sigma$ *are called* actions *and are written e.g.* n‽a *instead of* $(n,$ ‽$, a)$*. We use* τ *to range over actions.*

For instance, the machine depicted on Figure 3.2a is formally represented by the tuple $\mathfrak{M}_0 = (\{a, b\}, \{1, 2\}, \{q, q_a, q_b\}, q, T_0)$ where the set of transitions T_0 is

$$T_0 = \{(q, 1?a, q_a), (q, 1?b, q_b), (q_a, 2!a, q), (q_b, 2!b, q)\}\ .$$

Similarly, the machine of Figure 3.2b is $\mathfrak{M}_1 = (\{a, b\}, \{q_0, q_1, q_2, q_3, q_f\}, q_0, T_1))$ where $T_1 = \{(q_0, 1!a, q_1), (q_1, 1!b, q_2), (q_2, 2?a, q_3), (q_3, 2?b, q_f)\}$.

Let us group communicating machines together to form *communicating systems*.

Definition 3.2 (Communicating system) *A (communicating) system* $\mathfrak{S}$ *is a tuple* $\mathfrak{S} = (F, M_1, \ldots, M_n)$ *of communicating machines* $M_i = (\Sigma, B, Q_i, q_{0_i}, T_i)$ *that share the same alphabet of messages and the same set of buffers, together with a set* $F \subseteq \prod_{i=1}^{n} Q_i$ *of* final states.

For instance, the system of Figure 3.2 is $\mathfrak{S}_0 = (\{(q, q_f)\}, \mathfrak{M}_0, \mathfrak{M}_1)$. The set of final states does not matter yet, as no semantics is attached to it. Intuitively, when a communicating system has reached a final state, its communications may be considered as finished, and the channels can be closed.

Let us define straight away some syntactic notions on systems relevant for the rest of this chapter.

Definition 3.3 (Determinism) *A CFSM* (Σ, B, Q, q_0, T) *is* deterministic[1] *if for all states* $q \in Q$ *and all actions* $\tau \in B \times \{?, !\} \times \Sigma$*,* $(q, \tau, q') \in T$ *and* $(q, \tau, q'') \in T$ *implies* $q' = q''$*.*

A system is deterministic if all its constituting machines are.

Definition 3.4 *A state* q *of a CFSM* (Σ, B, Q, q_0, T) *is a* sending state *(resp. a* receiving state*) if all the transitions leaving from it are sending (resp. receiving) ones:* $\forall (q, n$‽$a, q') \in T.$ ‽ $= !$ *(resp.* $\forall (q, n$‽$a, q') \in T.$ ‽ $= ?$*).*

A state that has at least one sending and one receiving transitions is called a mixed state.

[1] In the literature on communicating automata (see for instance the book by Bollig [Bol06]), determinism on sending actions is often seen as the following stronger property: for all states $q \in Q$, if $(q, n!a_1, q_1) \in T$ and $(q, n!a_2, q_2) \in T$ then $a_1 = a_2$ and $q_1 = q_2$. The definition of determinism for receive actions coincides with ours. We have chosen a weaker definition because it is the one assumed by Singularity contracts [FAH+06, SB09], and because it suffices, along with other restrictions detailed below, to ensure the good properties enjoyed by Singularity contracts.

Definition 3.5 (Positional) *A CFSM (Σ, B, Q, q_0, T) is* positional *if it has no mixed state. A system is positional if all its constituting machines are.*

The system $\mathfrak{S}_0$ is both deterministic and positional.

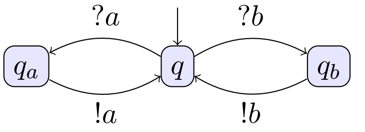
3.1.2 Dialogue systems and contracts

Let us now introduce dialogue systems and contracts. A dialogue system is a communicating system of only two machines and two buffers (1 and 2) where each buffer is used in only one direction, and a contract is a compact representation of a dialogue system whose CFSMs are each other's *dual*.

Definition 3.6 (Dialogue system) *A* dialogue machine *is a CFSM $\mathfrak{M} = (\Sigma, B, Q, q_0, T)$ where $B = \{1, 2\}$, and where the actions of T are either all of the form $1!a$ or $2?a$, or all of the form $2!a$ or $1?a$.*

A dialogue system *is a communicating system $\mathfrak{D} = (F, \mathfrak{M}_1, \mathfrak{M}_2)$ consisting of precisely two dialogue machines $\mathfrak{M}_i = (\Sigma, \{1, 2\}, Q_i, q_{0_i}, T_i)$, where $\mathfrak{M}_1$ sends on channel 1 and receives on channel 2, and vice-versa (hence for each machine $\mathfrak{M}_i$, the set T_i contains only actions of the form $i!a$ or $(3 - i)?a$).*

Since on each machine of a dialogue system the sending actions are all performed over the same buffer, and similarly for the receiving actions, we will often omit buffer numbers in their presentation. For instance, the example of Figure 3.2a can be depicted as:



The dual of a CFSM $\mathfrak{M}$ is $\mathfrak{M}$ where the actions $n?a$ are changed in $n!a$ and vice-versa.

Definition 3.7 (Dual actions, dual machines) *The dual of an action $n?a$ is $n!a$. The dual of an action $n!a$ is $n?a$. Given an action τ, we write $\bar{\tau}$ for its dual.*

The dual of a set of transitions T is $\bar{T} \triangleq \{\bar{\tau} \mid \tau \in T\}$. The dual of a CFSM $\mathfrak{M} = (\Sigma, B, Q, q_0, T)$ is $dual(\mathfrak{M}) \triangleq (\Sigma, B, Q, q_0, \bar{T})$.

Note that the dual operation is involutive: $\bar{\bar{\tau}} = \tau$, hence $dual(dual(\mathfrak{M})) = M$. The following definition formalises the channel contracts of Singularity [FAH$^+$06] (without restrictions yet, see Definition 3.25). Note that the set of final states is specified within the contract.

Definition 3.8 (Contract) *A* contract *is a tuple (Σ, Q, q_0, F, T) where $T \subseteq Q \times (\{?, !\} \times \Sigma) \times Q$.*

A contract is not a communicating system per se, but it is associated to a canonical one.

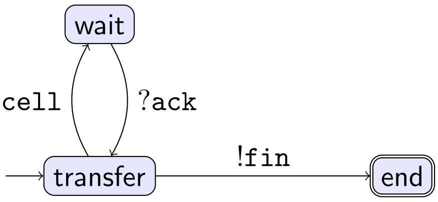
Definition 3.9 *Let $\mathfrak{C}$ be a contract (Σ, Q, q_0, F, T). The communicating machine associated to $\mathfrak{C}$ is*

$$\mathfrak{M}_{\mathfrak{C}} \triangleq (\Sigma, \{1,2\}, Q, q_0, \{(q, 1?a, q') \mid (q, ?a, q') \in T\} \cup \{(q, 2!a, q') \mid (q, !a, q') \in T\}) \ .$$

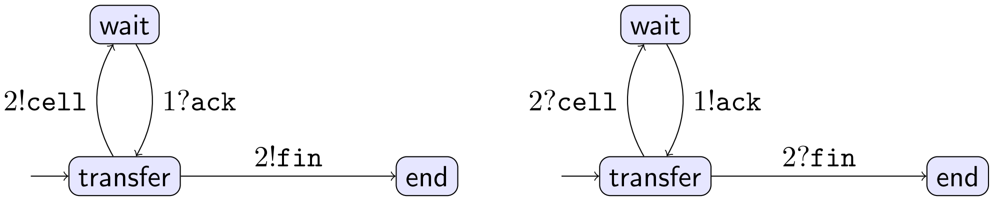
To the contract $\mathfrak{C}$, we associate the following dialogue system $\mathfrak{D}_{\mathfrak{C}}$:

$$\mathfrak{D}_{\mathfrak{C}} \triangleq (\{(q,q) \mid q \in F\}, \mathfrak{M}_{\mathfrak{C}}, \mathit{dual}(\mathfrak{M}_{\mathfrak{C}})) \ .$$

For instance, consider the contract $\mathfrak{C}$ with final state end (depicted by a double frame):



This contract denotes the following dialogue system, with final state $\{(\mathsf{end}, \mathsf{end})\}$:



The properties of dialogue systems are extended to contracts in the obvious manner. For instance, a contract is deterministic if the dialogue system that it denotes is. The dual $\mathit{dual}(\mathfrak{C})$ of a contract $\mathfrak{C}$ is also defined in a straightforward manner. Given a contract $\mathfrak{C} = (\Sigma, Q, q_0, F, T)$, we write $\mathit{init}(\mathfrak{C})$ for q_0 and $\mathit{finals}(\mathfrak{C})$ for F.

3.1.3 Semantics

In the following, and unless specified otherwise, $\mathfrak{S}$ is a communicating system $(F, \mathfrak{M}_1, \ldots, \mathfrak{M}_n)$ with $\mathfrak{M}_i = (\Sigma, B, Q_i, q_{0_i}, T_i)$ and $B = \{1, \ldots, p\}$. A configuration of a system (often written C) specifies the current state of each machine of the system and the contents of the buffers:

Definition 3.10 (System configuration) *Given a system $\mathfrak{S}$ where $|B| = p$, the set of configurations of the system is $(\prod_{i=1}^{n} Q_i) \times (\Sigma^*)^p$.*

We distinguish three particular kinds of configurations: the initial configuration, final ones, and stable ones, where all buffers are empty. We write $\llcorner\!\lrcorner$ for the empty word.

Definition 3.11 (Initial configuration) *The* initial configuration *of the system is $\langle q_{0_1}, \ldots, q_{0_n}, \llcorner\!\lrcorner, \ldots, \llcorner\!\lrcorner \rangle$.*

Definition 3.12 (Final configuration) *A configuration $\langle q_1, \dots, q_n, w_1, \dots, w_p \rangle$ is* final *if $(q_1, \dots, q_n) \in F$.*

Definition 3.13 (Stable configuration) *A configuration $\langle q_1, \dots, q_n, w_1, \dots, w_p \rangle$ is* stable *if $w_i = \sqcup$ for all i.*

A communicating system dictates how to go from one configuration to the other.

Definition 3.14 (System transition relation) *Given a communicating system $\mathfrak{S}$, the transition relation $\rightarrow_{\mathfrak{S}}$ of $\mathfrak{S}$ is defined as the union of the relations $\xrightarrow{\tau}_{\mathfrak{S}}$ for $\tau \in \bigcup_{i=1}^{n} T_i$, where $\langle q_1, \dots, q_n, w_1, \dots, w_m \rangle \xrightarrow{\tau}_{\mathfrak{S}} \langle q'_1, \dots, q'_n, w'_1, \dots, w'_m \rangle$ is defined as follows:*

- *if τ is a send action $j!a$ then there is $i \in \{1, \dots, n\}$ such that $(q_i, j!a, q'_i) \in T_i$, $w'_j = w_j \cdot a$, and $q'_k = q_k$ and $w'_l = w_l$ for all $k \neq i$ and $l \neq j$;*
- *if τ is a receive action $j?a$ then there is $i \in \{1, \dots, n\}$ such that $(q_i, j?a, q'_i) \in T_i$, $w_j = a \cdot w'_j$, and $q'_k = q_k$ and $w'_l = w_l$ for all $k \neq i$ and $l \neq j$;*

We write $\rightarrow$ and $\xrightarrow{\tau}$ when $\mathfrak{S}$ can be inferred from the context. A run of a system is a sequence of consecutive transitions from the initial state:

Definition 3.15 (Run of a system) *Given a system $\mathfrak{S}$, a* run *is a sequence of transitions $C_0 \xrightarrow{\tau_1} C'_0, \dots, C_k \xrightarrow{\tau_{k+1}} C'_k$ where $k \geq 0$, $C'_i = C_{i+1}$ and C_0 is initial. The run is* accepting *if C'_k is final. If $k = 0$, the run is simply C_0; it is accepting if C_0 is final.*

A configuration is reachable *if there is a run of the system that leads to it.*

We write $C_0 \xrightarrow{\tau_1} C_1 \xrightarrow{\tau_2} \dots \xrightarrow{\tau_k} C_{k+1}$ or $C_0 \xrightarrow{\tau_1 \dots \tau_k}{}^{k+1} C_{k+1}$ for a run $C_0 \xrightarrow{\tau_1} C_1, C_1 \xrightarrow{\tau_2} C_2, \dots, C_k \xrightarrow{\tau_k} C_{k+1}$, sometimes omitting some configurations and writing "$\cdot$" instead. We write $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$.

The accepting runs of our example $\mathfrak{S}_0$ of Figure 3.2 correspond to all the interleavings of the sequences of actions $1!a$ $1!b$ $2?a$ $2?b$ and $1?a$ $2!a$ $1?b$ $2!b$ that respect the precedence of a send over the corresponding receive, for instance

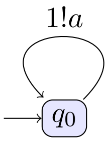
$$\langle q, q_0, \sqcup, \sqcup \rangle \xrightarrow{1!a} \cdot \xrightarrow{1!b} \langle q, q_2, ab, \sqcup \rangle \xrightarrow{1?a} \cdot \xrightarrow{2!a} \langle q, q_2, b, a \rangle \xrightarrow{1?b} \cdot \xrightarrow{2!b} \langle q, q_2, \sqcup, ab \rangle \xrightarrow{2?a}$$

$$\cdot \xrightarrow{2?b} \langle q, q_f, \sqcup, \sqcup \rangle$$

3.1.4 Safety properties

Let us describe the properties we are interested in for dialogue systems in general, and contracts in particular. As they are not specific to the class of dialogue systems, we define them for the more general class of communicating finite-state machines.

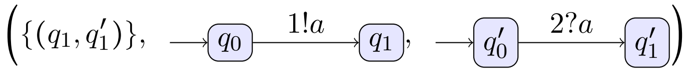
A *bounded system* is one in which there exists a bound on the size of the queues. For instance, the system of Figure 3.2 is bounded, but a system that contains the following machine is not, because this machine can enqueue arbitrarily many a messages in buffer 1:



Definition 3.16 (Bounded system) *A system is k-*bounded *if, for all reachable configurations $\langle q_1, \dots, q_n, w_1, \dots, w_p\rangle$, $|w_i| \leq k$ for all i.*

A system $\mathfrak{S}$ is bounded *if there exists a bound $k \in \mathbb{N}$ such that $\mathfrak{S}$ is k-bounded.*

A system is in a *deadlock state* if each participant is waiting for one of the other participants to send a message. In this case, they are all stuck, waiting for each other's messages. Note that the special case where all participants are in a final state is not a *proper* deadlock since the run of the system is finished. The system of Figure 3.2 is deadlock-free in that sense, but the one below is not ($\langle q_1, q_0', a, \sqcup\rangle$ is a reachable deadlock configuration: the second machine tries to receive from an empty channel).



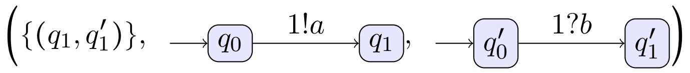
Definition 3.17 (Deadlock-free system) *We say that a configuration $\langle q_1, \dots, q_n, w_1, \dots, w_p\rangle$ of a system is a* deadlock configuration *if:*

1. *all q_i are receiving states*
2. *all the inspected queues are empty:*

$$\forall j. \, (\exists i. \, \exists a. \, \exists q'. \, (q_i, j?a, q') \in T_i) \Rightarrow w_j = \sqcup \, .$$

A configuration C is a proper deadlock *if C is deadlock and not final. A system is* deadlock-free *if no reachable configuration is a proper deadlock.*

A system is *faulty* if there is a possibility that, at some point in time, one participant is ready to receive certain messages but an unexpected one is at the head of one buffer. Again, the example depicted in Figure 3.2 is fault-free, whereas the one below can reach the faulty configuration $\langle q_1, q_0', a\rangle$ (the second machine is expecting a b message).



Definition 3.18 (Fault-free system) *We say that a configuration $\langle q_1, \dots, q_n, w_1, \dots, w_p\rangle$ of a system is an* unspecified reception configuration *if there are i, j and a such that*

- *$w_j = a \cdot w_j'$;*
- *$\{a' \mid (q_i, (j?a'), q') \in T_i\} \neq \varnothing$ and*
- *$a \notin \{a' \mid (q_i, (j?a'), q') \in T_i\}$.*

A system is fault-free *if none of its reachable configurations leads to an unspecified reception.*

Finally, the final configurations of a system should be reachable only when all the communications have been resolved and there is no message left in the queues (that is, final configurations should be *stable*). This is the case for the system of Figure 3.2 if we take as final state $\{(q, q_f)\}$, but not if we take for instance $\{(q, q_1)\}$, as the unstable configuration $\langle q, q_1, \sqcup, a\rangle$ would then be considered final.

Definition 3.19 (Leak-free system) *A system is* leak-free *if all accepting runs end in a stable configuration.*

Note that a system without accepting states is automatically leak-free.

3.2 Verification of communicating systems

3.2.1 Simulation of a Turing machine by a dialogue system

Unfortunately, given an arbitrary contract, one cannot decide whether it satisfies any of the safety properties listed in the previous section. The main reason behind this is that CFSMs can simulate Turing machines, which was first noticed by Brand and Zafiropulo [BZ83]; the main idea of the proof is that an unbounded buffer can act as the tape of a Turing machine. This result was later extended to dialogue systems of the form $(\mathfrak{M}, \mathfrak{M})$ (so-called identical communicating processes) by Finkel and McKenzie [FM97]. We extend it here to contracts, that is systems of the form $(\mathfrak{M}, \mathit{dual}(\mathfrak{M}))$. All the undecidability results presented in the next section are based on the encoding of Turing machines into dialogue systems presented below.

Definition 3.20 (Turing machine) *A Turing machine is a tuple* $(\Sigma, Q, T, \Box, q_0, q_f)$ *where:*

- Σ *is the tape alphabet;*
- Q *is the finite set of states;*
- $T \subseteq Q \times \Sigma \times Q \times \Sigma \times \{\triangleleft, \triangleright\}$ *is the transition relation;*
- $\Box \notin \Sigma \cup Q$ *is the blank symbol;*
- $q_0 \in Q$ *and* $q_f \in Q$ *are respectively the initial and final states.*

A configuration of a Turing machine $(\Sigma, Q, T, \Box, q_0, q_f)$ is a word from

$$(\Sigma \times \{\Box\})^*(\Sigma \times Q)(\Sigma \times \{\Box\})^*(\{(\Box, \Box)\})$$

that accounts for the current state of the machine, the current word written on the tape and the position of the head. We assume without loss of generality that every Turing machine starts on an empty tape, that the right end of every configuration is marked by $(\Box,\Box)$, that the machine never reaches the final control state q_f until the very end, and that when this happens the tape is again empty. The initial and final states are thus respectively $(\Box,q_0)(\Box,\Box)$ and $(\Box,q_f)(\Box,\Box)$.

The encoding of a Turing machine $\mathfrak{M}$ using a dialogue system intuitively uses the buffers to store the current configuration. One of the machines then simulates the transitions of $\mathfrak{M}$, while the other ones acts as a mere copycat. This allows the simulating machine to cycle freely through the configuration to find the position of the head at each transition, perform the transition locally, as this only requires the memory of the immediate surroundings of the head, and cycle again to repeat this process.

Let $\mathfrak{M} = (\Sigma_M, Q_M, T_M, \Box, q_0, q_f)$ be a Turing machine. The dialogue system that will model $\mathfrak{M}$ is $\mathfrak{S} = (F, \mathfrak{M}_{\mathsf{sim}}, \mathfrak{M}_{\mathsf{cat}})$ as defined below. Intuitively, using r_i to denote an element of $Q \cup \{\Box\}$, a configuration

$$C_{\mathfrak{M}} = (a_1, r_1)\cdots(a_k, r_k)\ (\Box,\Box)$$

will be coded by the system's global configuration $(q_{\mathsf{sim}}, q_{\mathsf{cat}}, w_1, w_2)$, where $w_1 \cdot w_2$ is a rotation of $C_{\mathfrak{M}}$, that is there exists a j such that

$$w_1 \cdot w_2 = (a_{j+1}, r_{j+1})\cdots(a_k, r_k)\ (\Box,\Box)\ (a_1, r_1)\cdots(a_j, r_j)\ .$$

The copying machine The copycat machine $\mathfrak{M}_{\mathsf{cat}}$ is defined as

$$((\Sigma \times (Q \cup \{\Box\})), \{1,2\}, \{i_{\mathsf{cat}}, q_{\mathsf{cat}}, \mathsf{end}_{\mathsf{cat}}\} \cup \{(a,r)\}_{(a,r)\in\Sigma\times(Q\cup\{\Box\})}, i_{\mathsf{cat}}, T_{\mathsf{cat}})\ .$$

Its structure, depicted in Figure 3.3, is simple. Following the notations of Bollig [Bol06], we write a for $(a,\Box)$ and $a{\leftarrow}q$ for (a,q). Its *modus operandi* is to copy everything from its input buffer to its output buffer, except for $\Box{\leftarrow}q_f$, which leads $\mathfrak{M}_{\mathsf{cat}}$ into the state $\mathsf{end}_{\mathsf{cat}}$ from which no transition emanates. In a run of the whole system, nothing will be left to receive when this message is received, and $\mathsf{end}_{\mathsf{cat}}$ will be the final state of $\mathfrak{M}_{\mathsf{cat}}$. This machine will be the first to make a transition in the whole system, filling the tape with the initial configuration of the Turing machine $(\Box,q_0)(\Box,\Box)$.

The transition machine Anticipating the needs of the proofs of the results of the next section, we will build a machine that is deterministic, positional and synchronising (as is $\mathfrak{M}_{\mathsf{cat}}$). Let us present how to encode $\mathfrak{M}$'s transitions.

Consider a transition $\tau = (q_i, a_i, q_{i'}, a_{i'}, \triangleleft) \in T$ wherein the head is moving to the left. This will be encoded by the part of the CFSM described in Figure 3.4. This fragment of CFSM can be decomposed into the left and right parts. Which one is triggered depends on where the head of the Turing machine is positioned on the tape: if it is on the first letter then the left branch is taken, otherwise the right one is taken and the letter that was read is remembered by the control state. This allows the CFSM to travel deterministically along the tape while searching for the head.

Let us describe the second case, the first one being similar. The tape is unrolled until the head is found: if when in the control state a_h the machine receives another headless letter

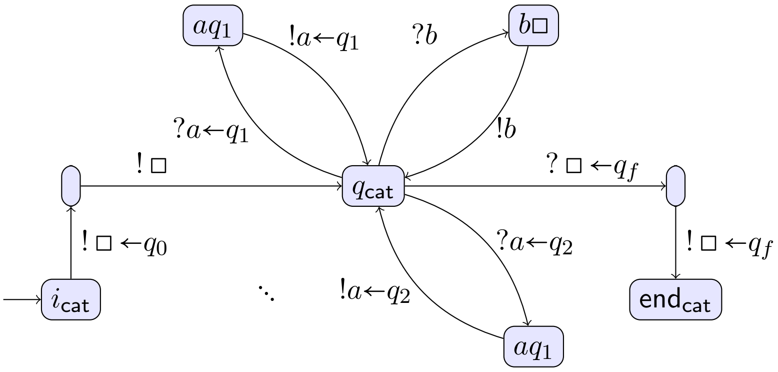


Figure 3.3: A few of the transitions of the copycat machine.

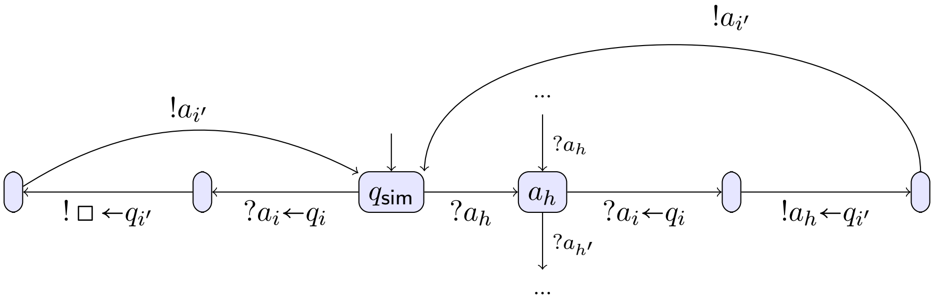


Figure 3.4: Encoding of a transition of $\mathfrak{M}$ moving the head to the left.

$(a_{h'}, \square)$, a_h is sent back (not pictured here) and the machine moves in state $a_{h'}$. The process is repeated until the head is found. When this is the case, that is when $a_i{\leftarrow}q_i$ is received, and assuming the previous letter was a_h, the machine sends back $a_h{\leftarrow}q_{i'}$, effectively moving the head to the left and updating the control state of the Turing machine. The final step is to send the updated letter under the previous location of the head $a_{i'}$.

Not pictured in the figures are transitions that reset the transition machine whenever we reach the end of the tape: from every control state a_h there is a transition back to q_{sim} labelled by $?\square$, and there is a loop from q_{sim} to q_{sim} consisting of two transitions: $q_{\mathsf{sim}} \xrightarrow{?\square} \cdot \xrightarrow{!\square} q_{\mathsf{sim}}$. There is also a final transition $q_{\mathsf{sim}} \xrightarrow{?\square\leftarrow q_f} \mathsf{end}_{\mathsf{sim}}$.

Consider now a transition $\tau = (q_i, a_i, q_{i'}, a_{i'}, \triangleright) \in T$ wherein the head is moving to the right. This will be encoded by the fragment of the CFSM described in Figure 3.5. The steps are similar to the left transition case, except that we have to handle the case where we have to extend the tape to the right in addition to the cases where the head is at the beginning of the tape. Consider for instance the loop going to the right and coming back to q_{sim} from below. This time, a_h is remembered only because the forwarding mechanism has to be embedded in every transition to obtain a deterministic machine. It is immediately

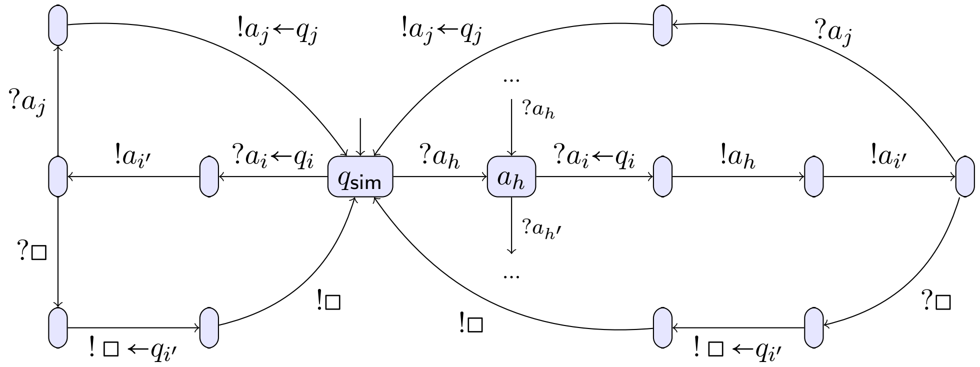


Figure 3.5: Encoding of a transition of $\mathfrak{M}$ moving the head to the right.

sent back unchanged when the head is found, followed by the updated letter $a_{i'}$. Then we try to read the next letter. In the case at hand, the end of the tape has been reached and we get □. The new head of the Turing machine is positioned on the final character (by sending back $(\square, q_{i'})$) and we end the sequence by sending the terminating symbol.

Theorem 3.1 *For every Turing machine $\mathfrak{M}$ there is a deadlock-free, positional and deterministic dialogue system that reaches its final state if and only if $\mathfrak{M}$ halts.*

Proof The dialogue system $(\{\mathsf{end}_{\mathsf{cat}}, \mathsf{end}_{\mathsf{sim}}\}, \mathfrak{M}_{\mathsf{cat}}, \mathfrak{M}_{\mathsf{sim}})$ has all the properties that the theorem asks for. □

3.2.2 A decidable class: half-duplex systems

A system is *half-duplex* (or has the *half-duplex property*) if there may never be more than one non-empty buffer at the same time in any of the executions. A classical physical example is the one of a group of persons communicating via walkie-talkies: there are never two persons speaking at the same time.

Definition 3.21 (Half-duplex system) *A system is* half-duplex *if, for all reachable configuration $\langle q_1, \dots, q_n, w_1, \dots, w_p \rangle$, there is at most one $i \in \{1, \dots, p\}$ such that $w_i \neq \sqcup$.*

Half-duplex systems are known to have a regular reachability set [CF05]; in particular, Cécé and Finkel have shown that all the safety problems we are interested in are decidable in polynomial time (on the size of the alphabet and the number of control states of each individual machine) on half-duplex dialogue systems, and that the half-duplex property itself is decidable in polynomial time.

Intuitively, every execution $C_0 \xrightarrow{\alpha} C_2$ of a half-duplex dialogue system can be turned into another execution $C_0 \xrightarrow{\alpha_1} C_1 \xrightarrow{\alpha_2} C_2$ that ends in the same configuration such that the run $C_0 \xrightarrow{\alpha_1} C_1$ is 1-bounded, C_1 is stable (its buffers are empty, see Definition 3.13), and $C_1 \xrightarrow{\alpha_2} C_2$ consists only of send actions by one of the two machines. This sequence of sending actions is performed by a finite-state machine, thus it can be described by a regular language. This shows that the language contained in the only non-empty queue of C_2 is in fact always regular. Let us remember this as the following lemma.

Lemma 3.1 ([CF05]) *Let C be a reachable configuration of a half-duplex system $\mathfrak{S}$ and α be an execution that reaches C from the initial configuration C_0 of $\mathfrak{S}$. There exist C', α_1 and α_2 such that:*

1. $C_0 \xrightarrow{\alpha_1} C' \xrightarrow{\alpha_2} C$,
2. *the execution $C_0 \xrightarrow{\alpha_1} C'$ is 1-bounded and C is stable, and*
3. *α_2 consists only of sending actions.*

To see why the class of half-duplex systems is recursive, Cécé and Finkel give the following lemma about non-half-duplex systems, which we will also use to characterise half-duplex contracts in the next section.

Lemma 3.2 ([CF05]) *A dialogue system $(F, \mathfrak{M}_1, \mathfrak{M}_2)$ is not half-duplex if and only if there exist two transitions $(q_1, 1!a, q_1') \in T_1$ and $(q_2, 2!b, q_2') \in T_2$ such that the state $\langle q_1, q_2, \sqcup, \sqcup \rangle$ is reachable by a 1-bounded execution.*

Intuitively, a system that is not half-duplex can reach a stable configuration from a 1-bounded execution from which both machines can perform a sending action. This condition is also obviously sufficient for the system not to be half-duplex, since if both machines perform their respective send, the two corresponding buffers will become non-empty.

Channel systems with errors In the case of a finite alphabet of messages, unreliable (or *lossy*) channel systems are known to have good decidability properties. Indeed, given a system where all the channels involved are lossy, one can decide whether a given configuration is reachable or not, from which the decidability of many safety problems follows [AJ93].

If messages arrive out-of-order, then CFSMs are equivalent to counter machines (because the queue is then a mere multiset of messages, and sending and receiving are equivalent to respectively incrementing and decrementing the counter associated to the corresponding message). In particular, if we cannot test the emptiness of the queue, that is if attempting to receive on an empty queue is blocking, then it is equivalent to a vector addition system with states (VASS). In this case, the reachability problem (along with many others) is also decidable [May81, ST77]. If there exists a non-blocking receive primitive, the system can encode counter machines, which are as powerful as Turing machines [Min67] (starting from 2 counters, hence two buffers).

3.3 Contracts

3.3.1 Syntactic sufficient conditions

Half-duplex contracts As we will see in the upcoming sections, contracts, restricted as they are by their symmetric form, still retain the power of Turing machines from dialogue systems (Theorem 3.2). However, simple syntactic restrictions on contracts are *sufficient* (but not necessary) to ensure fault, leak, and deadlock-freedom. These restrictions imply in particular that the system described by the contract must be half-duplex, which shows that we need not restrict ourselves to contracts to obtain decidability, where we could have used any half-duplex dialogue system. The restrictions include the fact that a contract should be deterministic and positional. On contracts, this is already enough to ensure the half-duplex property. The proof relies essentially on the following lemma.

Lemma 3.3 *Let $\mathfrak{C} = (\Sigma, Q, q_0, F, T)$ be a deterministic and positional contract. For all execution $\langle q_0, q_0, \sqcup, \sqcup \rangle \xrightarrow{\alpha}{}^{*} \langle q_1, q_2, \sqcup, \sqcup \rangle$ that ends in a stable state, $q_1 = q_2$.*

Proof Let us first remark that if α leads to a stable configuration by a 1-bounded run, then $|\alpha|$ is even, and α consists of consecutive sends $n!a$ and receives $n?a$ for some n and a. Moreover, by Lemma 3.1 and since the configuration resulting from the execution of α is stable, there exists a 1-bounded execution of the system associated to $\mathfrak{C}$ that produces the same state. We can thus assume without loss of generality that α is 1-bounded.

Let us show the lemma by an induction on the size of α. If $|\alpha| = 0$, then $q_1 = q_2 = q_0$. Suppose now that $n > 0$ and for all 1-bounded execution β of size strictly less than n, if $\langle q_0, q_0, \sqcup, \sqcup \rangle \xrightarrow{\beta}{}^{*} \langle q, q', \sqcup, \sqcup \rangle$ then $q = q'$. If $|\alpha| = n$, there is n and a such that $\alpha = \beta \cdot n!a\ n?a$, and by induction hypothesis,

$$\langle q_0, q_0, \sqcup, \sqcup \rangle \xrightarrow{\beta}{}^{*} \langle q, q, \sqcup, \sqcup \rangle \xrightarrow{n!a} \cdot \xrightarrow{n?a} \langle q_1, q_2, \sqcup, \sqcup \rangle \ .$$

In the last two steps, either the first machine is sending and the second receiving, or the opposite. Suppose we are in the first case, the other one being symmetric. Since $\mathfrak{C}$ is deterministic, there is only one outgoing transition from q with label $!a$, hence there is only one transition from q with label $?a$ in the dual machine. Hence, both machines made the step to the same next state and $q_1 = q_2$. □

Lemma 3.4 (Half-duplex contracts) *A contract that is both deterministic and positional is half-duplex.*

Proof Let $\mathfrak{C} = (\Sigma, Q, q_0, F, T)$ be a deterministic and positional contract. Suppose that $\mathfrak{C}$ is not half-duplex. By Lemma 3.2, there is a stable configuration $C = \langle q_1, q_2, \sqcup, \sqcup \rangle$ accessible by a 1-bounded run $\langle q_0, q_0, \sqcup, \sqcup \rangle \xrightarrow{\alpha}{}^{*} \langle q_1, q_2, \sqcup, \sqcup \rangle$ such that $(q_1, !a, q_1') \in T$ and $(q_2, !b, q_2') \in \bar{T}$, hence $(q_2, ?b, q_2') \in T$, for some a, b, q_1', q_2'.

By Lemma 3.3, $q_1 = q_2$, hence there is in fact both a sending and a receiving transition from q_1 in $\mathfrak{C}$, thus q_1 is not positional, which contradicts our hypothesis that $\mathfrak{C}$ is. □

The converse is not true in general: there are half-duplex contracts that are neither deterministic nor positional. However, it becomes true for deterministic *trimmed* contracts.

Definition 3.22 (Trimmed contract) *A contract $\mathfrak{C} = (\Sigma, Q, q_0, F, T)$ is* trimmed *if every control state is reachable from q_0 in the oriented graph $\mathfrak{G} = (Q, T')$ formed by the contract's states and transitions, where*

$$T' \triangleq \{(q, q') \mid \exists \tau. (q, \tau, q') \in T\} \ .$$

A non-trimmed contract is one in which there are control states that are “disconnected” from the rest of the control states. Any non-trimmed contract may be turned into a trimmed one with the same operational meaning simply by deleting those states.

Lemma 3.5 *For every deterministic trimmed contract $\mathfrak{C}$, $\mathfrak{C}$ is half-duplex if and only if it is positional.*

Proof The reverse implication is already proved by Lemma 3.4. Let us prove the direct implication.

Let us notice first that in the dialogue system associated to a trimmed contract $\mathfrak{C} = (\Sigma, Q, q_0, F, T)$, $\langle q, q, \sqcup, \sqcup\rangle$ is always reachable. Indeed $q \in Q$ is reachable in the oriented graph formed using Q as vertices and T as edges, so there is a sequence of transitions $(q_0, \tau_1, q_1), \ldots, (q_{n-1}, \tau_n, q)$ from q_0 to q. The following execution is thus valid, where $\tau' = \tau$ for sending actions and $\tau' = \bar{\tau}$ for receiving ones:

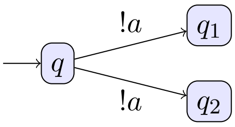
$$\langle q_0, q_0, \sqcup, \sqcup\rangle \xrightarrow{\tau_1'} \cdot \xrightarrow{\bar{\tau}_1'} \langle q_1, q_1, \sqcup, \sqcup\rangle \cdots \langle q_{n-1}, q_{n-1}, \sqcup, \sqcup\rangle \xrightarrow{\tau_n'} \cdot \xrightarrow{\bar{\tau}_n'} \langle q, q, \sqcup, \sqcup\rangle$$

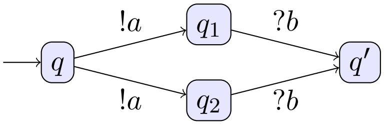
Suppose that $\mathfrak{C}$ is not positional. There is $q \in Q$ and a_1, a_2, q_1 and q_2 such that $(q, !a_1, q_1), (q, ?a_2, q_2) \in T$. Since $\langle q, q, \sqcup, \sqcup\rangle$ is reachable, the following execution is valid:

$$\langle q_0, q_0, \sqcup, \sqcup\rangle \to^* \langle q, q, \sqcup, \sqcup\rangle \xrightarrow{1!a_1} \langle q_1, q, a_1, \lambda\rangle \xrightarrow{2!a_2} \langle q_1, q_2, a_1, a_2\rangle$$

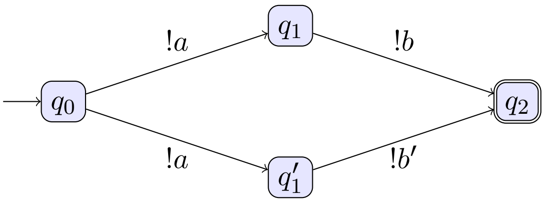
This execution is not half-duplex. □

Even trimmed half-duplex contracts are not necessarily deterministic. For instance, the following contracts are not deterministic but half-duplex nonetheless:

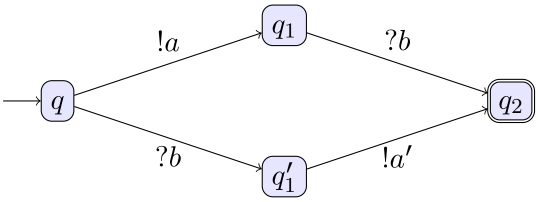




Fault-free contracts Deterministic and positional contracts have more properties than just being half-duplex: they are fault and deadlock-free. Before proving this claim, let us show that we cannot say the same of non-deterministic or non-positional contracts. For instance, the following non-deterministic contract is not fault-free:



The first machine can send a, then b, while the dual one receives a and then tries to receive b' whereas b is available. Non-positional contracts also embed a form of choice that can harm the safety properties, since in a non-positional state both machines may make dissonant moves.

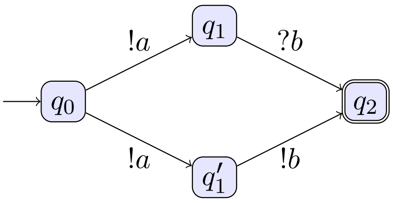


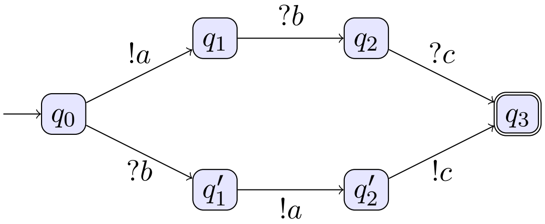
In the example above, the first machine may send a then wait for b, while the second one sends b and then wrongfully tries to receive a' whereas a is available.

Lemma 3.6 (Fault-freedom) *For every contract $\mathfrak{C}$, if $\mathfrak{C}$ is deterministic and positional then $\mathfrak{C}$ is fault-free.*

Proof If $\mathfrak{C}$ is deterministic and positional, it is half-duplex (Lemma 3.4). From Lemma 3.2, all executions are equivalent to a 1-bounded execution of the system followed by a sending-only phase. If a fault occurs, it must be during the first phase. During this phase, reachable states are of the form $\langle q, q, \sqcup, \sqcup \rangle$ (Lemma 3.3), $\langle q', q, a, \sqcup \rangle$ with $(q, !a, q') \in T$ or (symmetrically) $\langle q, q', \sqcup, a \rangle$ with $(q, ?a, q') \in T$. In the first case, queues are empty so there can be no fault. In the second and third case, the machine that did not perform the send is obviously about to receive a since its dual was able to send it from the same state, so there is no possible fault either. □

Deadlock-free contracts Similarly, the contracts depicted below are respectively non-deterministic and non-positional and both exhibit a deadlock.



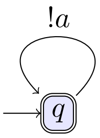


In the second example, the first machine can perform the sequence of actions $!a?b$ and its dual $!b?a$. Thus, they get both stuck waiting for c in the configuration $\langle q_2, q'_2, \sqcup, \sqcup \rangle$.

Lemma 3.7 (Deadlock-freedom) *For every contract $\mathfrak{C}$, if $\mathfrak{C}$ is deterministic and positional then $\mathfrak{C}$ is deadlock-free.*

Proof The proof follows the same reasoning as for Lemma 3.6. □

Leak-free contracts Being deterministic and positional is not sufficient for a contract to be leak-free, because there could be a loop consisting only of sends and involving a final state. In that case, the system could stop before the receiving machine has gone as many times around the loop as the sending, because both machines would be in a final state, like in this contract:



To enforce bounded communications (see Lemma 3.9), Sing♯ requires that all cycles in a given contract contain at least one send and one receive; we can require the same condition, restricted to those cycle containing a final state, to prevent leaks. We represent this by the notion of synchronising states of a dialogue machine. They are the states q such that each cycle going through q must contain at least one sending action and one receiving action. The name "synchronising" comes from the intuition that such a state q forces the two machines to synchronise between two consecutive stops at q. To simplify the presentation, we only define this notion for dialogue machines, where the criterion is easier to express, although it could be extended to arbitrary machines and systems.

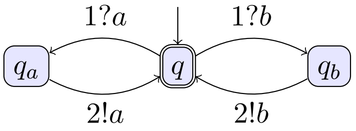
Definition 3.23 (Cycles) *Let $\mathfrak{M} = (\Sigma, B, Q, q_{init}, T)$ be a dialogue machine. A* cycle *in $\mathfrak{M}$ is a tuple of control states $(q_0, \dots, q_n)$ ($n \geq 1$) where $q_0 = q_n$ and $\forall 0 \leq k < n. \exists \tau \in B \times \{?, !\} \times \Sigma. (q_k, \tau, q_{k+1}) \in T$.*

A cycle is a sending (resp. receiving) cycle if all its transitions are sending (resp. receiving) ones.

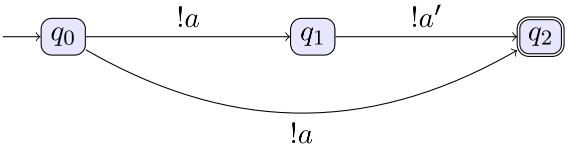
A cycle $(q_0, \dots, q_n)$ goes through q if $\exists 0 \leq i \leq n. q = q_i$.

Definition 3.24 (Synchronising states) *A state q of a dialogue machine $\mathfrak{M}$ is* synchronising *if no sending or receiving cycle of $\mathfrak{M}$ goes through it.*

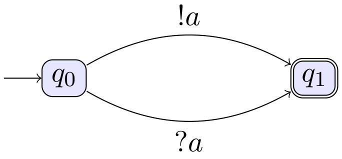
Note that in dialogue systems that are constructed from a contract, a state of one of the dialogue machines is synchronising if and only if the same state is synchronising in the other machine. We can thus talk freely about synchronising states of contracts, and in fact this property can be directly checked on the contract itself. For instance, all the states of this contract are synchronising:



Non-deterministic contracts may hide a leak as well. In the contract below, if each machines makes a different choice then they end in the same final state and can stop in the configuration $\langle q_2, q_2, a', \sqcup \rangle$ where there is still a' in one of the queues.



The same is true of non-positional contracts. In the following contract, each machine can take a different branch, which leaves one a message in each queue of the final configuration $\langle q_1, q_1, a, a\rangle$.



However, the three conditions together suffice to ensure leak-freedom.

Lemma 3.8 (Leak-freedom) *For every contract $\mathfrak{C}$, if $\mathfrak{C}$ is deterministic, positional and all its final states are synchronising, then $\mathfrak{C}$ is leak-free.*

Proof From Lemma 3.2, each execution of $\mathfrak{C}$ from its initial state is equivalent to a 1-bounded execution that ends in a stable state C, followed by a sending cycle in one of the machines. As there is no sending or receiving cycle surrounding a final state, if both machines are in the same control state then the final sending cycle is empty. The final configuration is thus stable, hence $\mathfrak{C}$ is leak-free. □

The SING♯ language additionally requires non-final states to be synchronising as well. As we are about to see, this ensures that communications are bounded.

Definition 3.25 (Singularity contracts) *We call* Singularity contract *any contract that is deterministic, positional, and whose control states are all synchronising.*

In a first version, the SING♯ compiler required all contracts to be deterministic and have only synchronising states, but not to be positional. Hence, contrarily to their claim, these contracts could lead to deadlocks, as remarked by Stengel and Bultan [SB09]. The definition above corresponds to the fixed version.

Bounded contracts As mentioned above, if we are more restrictive and assume that *all* the cycles of the contract contain at least one send and one receive, then the contract is bounded.

Lemma 3.9 (Boundedness) *For every contract $\mathfrak{C}$, if $\mathfrak{C}$ is deterministic, positional and all its states are synchronising, then $\mathfrak{C}$ is bounded.*

The bound is moreover computable in linear time in the number of transitions in the contract.

Proof Let us assume without loss of generality that $\mathfrak{C}$ is a trimmed contract, and show that the maximum size of the buffers in a run of the contract $\mathfrak{C}$ corresponds exactly to the maximum number of consecutive sending transitions of $\mathfrak{C}$, or to the maximum number of consecutive receiving transitions of $\mathfrak{C}$, whichever is the greatest. Since all the control states of $\mathfrak{C}$ are synchronising, both numbers are finite. Let us write N for their maximum.

- Let us show that this bound is reached: N corresponds to a sequence from a control state q of $\mathfrak{M}$. By Lemma 3.5, $\langle q, q, \sqcup, \sqcup\rangle$ is a reachable configuration of the system associated to $\mathfrak{C}$. If N corresponds to a sequence of sending transitions $q \xrightarrow{!a_1} \cdots \xrightarrow{!a_N} q'$, then $\langle q', q, a_1 \cdots a_N, \sqcup\rangle$ is reachable. Symmetrically, if N corresponds to a sequence of receiving transitions $q \xrightarrow{?a_1} \cdots \xrightarrow{?a_N} q'$, then $\langle q, q', \sqcup, a_1 \cdots a_N\rangle$ is reachable.

- Suppose now that a configuration $\langle q_1, q_2, w_1, w_2\rangle$ is reachable, with for instance $|w_1| = n_1$ and $|w_2| = n_2$. If $n_1 > 0$, then since $\mathfrak{C}$ is half-duplex by Lemma 3.4, $w_2 = \sqcup$. By Lemma 3.4 again and Lemma 3.3, the configuration can be reached from another configuration $\langle q, q, \sqcup, \sqcup\rangle$ by a sequence of sending transitions only. Thus, $n_1 \leq N$. If $n_1 = 0$ and $n_2 > 0$ the reasoning is similar. □

Let us summarise the results obtained in this section.

Corollary 3.1 *For every contract $\mathfrak{C}$, if $\mathfrak{C}$ is deterministic and positional then it is fault and deadlock-free. If moreover all its final states are synchronising then it is leak-free, and if all its states are synchronising it is bounded.*

Proof Direct consequence of Lemmas 3.6,3.7,3.8 and 3.9. □

In particular, Singularity contracts are bounded and free from faults, deadlocks and leaks.

3.3.2 Contract verification is undecidable

Theorem 3.2 *The following six problems are all undecidable:*

1. *Input: A synchronising and deterministic contract $\mathfrak{C}$.*
   Output: Whether or not $\mathfrak{C}$ is leak-free.

2. *Input: A synchronising and positional contract $\mathfrak{C}$.*
   Output: Whether or not $\mathfrak{C}$ is leak-free.

3. *Input: A synchronising and deterministic contract $\mathfrak{C}$.*
   Output: Whether or not $\mathfrak{C}$ is deadlock-free.

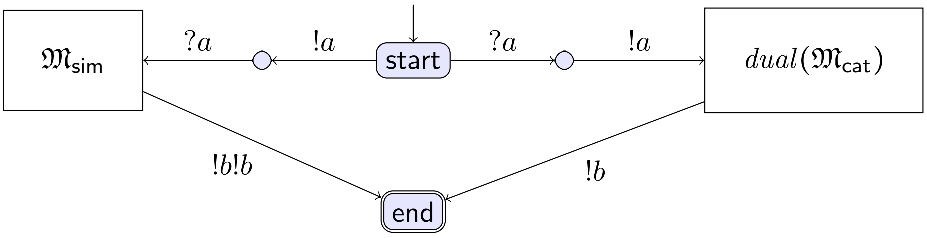
4. *Input: A synchronising and positional contract $\mathfrak{C}$.*
   Output: Whether or not $\mathfrak{C}$ is deadlock-free.

5. *Input: A synchronising and deterministic contract $\mathfrak{C}$.*
   Output: Whether or not $\mathfrak{C}$ is fault-free.

6. *Input: A synchronising and positional contract $\mathfrak{C}$.*
   Output: Whether or not $\mathfrak{C}$ is fault-free.

Proof We reduce each of these six problems to the halting problem of Turing machines using Theorem 3.1 as a black box. Let $\mathfrak{M} = (\Sigma, Q, T, \square, q_0, q_f)$ be a Turing machine, and $(F, \mathfrak{M}_{\mathsf{sim}}, \mathfrak{M}_{\mathsf{cat}})$ be the dialogue system equivalent to $\mathfrak{M}$ by Theorem 3.1.

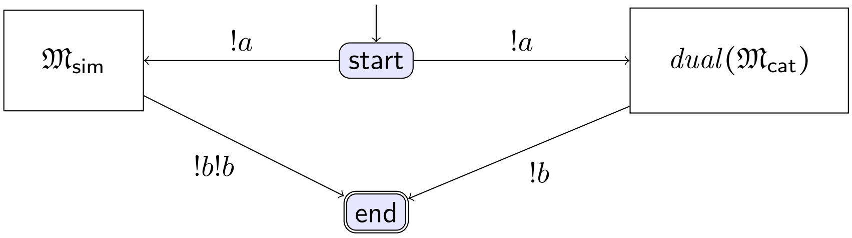
1. Consider the following contract $\mathfrak{C}_{\mathsf{leak}}$, where $a, b \notin \Sigma$, the incoming arrow into $\mathfrak{M}_{\mathsf{sim}}$ goes to i_{sim}, and the outgoing arrow comes from $\mathsf{end}_{\mathsf{sim}}$, and similarly for $dual(\mathfrak{M}_{\mathsf{cat}})$:


$\mathfrak{M}_{\mathsf{sim}}$
$?a$
$!a$
start
$?a$
$!a$
$dual(\mathfrak{M}_{\mathsf{cat}})$
$!b!b$
$!b$
end

This contract is synchronising and deterministic. Let us show that it exhibits a leak if and only if $\mathfrak{M}$ halts. Suppose that $\mathfrak{M}$ halts; the configuration $\langle \mathsf{end}_{\mathsf{sim}}, \mathsf{end}_{\mathsf{cat}}, \sqcup, \sqcup \rangle$ is then reachable. From this configuration, the first machine will send b twice, but only one b will be received by the second machine before the system comes to a halt in the final configuration $\langle \mathsf{end}, \mathsf{end}, \sqcup, b \rangle$, which is not stable.

Conversely, suppose that $\mathfrak{C}_{\mathsf{leak}}$ has a leak. A run of the dialogue system associated to $\mathfrak{C}_{\mathsf{leak}}$ corresponds either to both machines having taken the left branch and executing in $\mathfrak{M}_{\mathsf{sim}}$, or to both of them having taken the right branch, thus executing in the copycat machine, or, and this is the behaviour that simulates a run of the Turing machine, each one taking a separate way. In the first as well as the second case, it is straightforward to check that there is no leak, thanks to the construction of $\mathfrak{M}_{\mathsf{sim}}$ and $\mathfrak{M}_{\mathsf{cat}}$. So it must be that we are in the last case, hence that $\langle \mathsf{end}_{\mathsf{sim}}, \mathsf{end}_{\mathsf{cat}}, \sqcup, \sqcup \rangle$ is reachable, which is equivalent to saying that $\mathfrak{M}$ terminates by Theorem 3.1. This shows that checking whether a synchronising and deterministic contract is leak-free is undecidable.

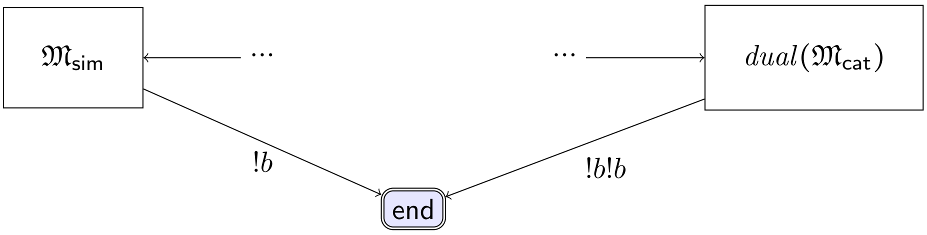
2. Following the same idea, one may build the following synchronising and positional contract:


$\mathfrak{M}_{\mathsf{sim}}$
$!a$
start
$!a$
$dual(\mathfrak{M}_{\mathsf{cat}})$
$!b!b$
$!b$
end

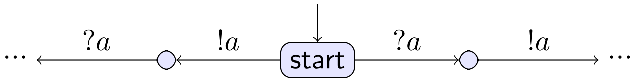
This contract has a leak if and only if $\mathfrak{M}$ halts, and it is synchronising and positional (but not deterministic).

3 & 4. One may prove in the same way that deadlock freedom is undecidable for synchronising contracts, even if they are either deterministic or positional (but not both, as

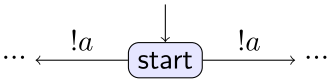
we will see next). It suffices to change the bottom part of the above contracts to the following:


$\mathfrak{M}_{\mathsf{sim}}$
$\cdots$
$\cdots$
$dual(\mathfrak{M}_{\mathsf{cat}})$
$!b$
$!b!b$
end

The initial part of the contract eluded above can be either

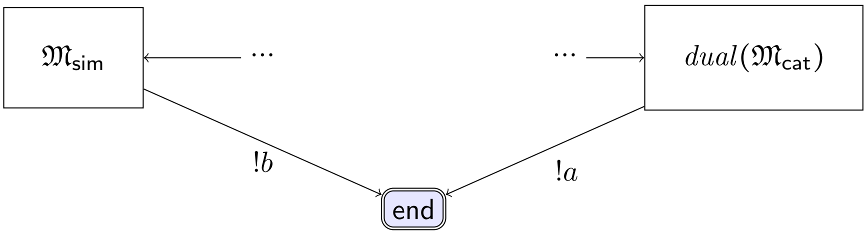


or



By taking the first template, we obtain a synchronising and deterministic (but not positional) contract that deadlocks if and only if the Turing machine $\mathfrak{M}$ halts, and by taking the second template we obtain a synchronising and positional (but not deterministic) contract that deadlocks if and only if $\mathfrak{M}$ halts.

5 & 6. With the same reasoning and initial templates, we prove undecidability of fault freedom for synchronising and deterministic or positional contracts with the following contract template:


$\mathfrak{M}_{\mathsf{sim}}$
$\cdots$
$\cdots$
$dual(\mathfrak{M}_{\mathsf{cat}})$
$!b$
$!a$
end

□

Conclusion

Table 3.1 sums up our decidability results for contracts and dialogue systems.

Restrictions	Bounded	Leak-free	Deadlock-free	Fault-free	Half-duplex
None	U	U	U	U	D
Half-duplex	D	D	D	D	Yes
Det. & Pos.	D	D	D	D	Yes
Contract	U	U	U	U	D
Det. & Synch. Contract	U	U	U	U	D
Pos. & Synch. Contract	U	U	U	U	D
Det. & Pos. & Synch. Contract	D	Yes	Yes	Yes	Yes

Table 3.1: Decidability results on dialogue systems. D: decidable U: undecidable, Yes: the property is always true, Det.: deterministic, Pos.: positional, Synch.: final states are synchronising. The first three lines consist of known results.

Contracts vs dialogue systems As this table shows, and perhaps surprisingly because of their syntactically restricted nature, contracts are as undecidable as arbitrary dialogue systems when it comes to the safety properties that we are interested in. Moreover, the syntactic restrictions imposed on contracts in Singularity make the contracts half-duplex. Since half-duplex dialogue machines already have all the required decidability properties, and the class of half-duplex dialogue machines is recursive, one might as well abandon contracts and use half-duplex systems instead.

However, this is not to say that contracts are useless as a class of communicating machines. Indeed one rarely needs more than contracts in practise. For instance, all the communication protocols of the Singularity project, which forms a fully functional operating system, have been expressed using contracts. Hence, our logic (defined in Chapter 5) and our tool (described in Chapter 7) use contracts instead of general dialogue systems.

Session types Session types [THK94] are a type system for a variant of the pi-calculus, where the types themselves may be seen as a kind of communicating system with primitives for sending and receiving messages of a certain type, branching according to a message tag, and recursion. In their structure, session types and contracts are very similar. One difference is that, while contracts only specify the sequence of message tags that occur on a channel, session types describe both the sequence of messages and the type of these messages. In a way, they combine into one object both the contracts and the footprints of messages that we will introduce in Chapter 5 and use to specify the type of our messages. Another difference is that the final state of a session types is always a state from which no communication can occur. Although this may seem mundane at first, this means that session types need not care about leak-freedom: every possible final state is automatically synchronising.

It is worth noting that session types have been extended to the *multiparty* case, with an arbitrary number of participants instead of two [HYC08], and even to parametrised interactions [YDBH10]. In particular, Deniélou and Yoshida recently provided an analysis of multiparty communications that can, among other things, compute a bound on the size of the channels of a pi-calculus process [DY10], whereas we analyse each bi-directional channel separately. Extending contracts to the multiparty case is certainly an interesting perspective, as it would enable us to take into account causality information given by all channels at once, and thus to perform a more accurate analysis.

Chapter 4

Concurrent Separation Logics

This chapter recalls the formalism of separation logic applied to the study of concurrent heap-manipulating programs.

4.1 Assertions

4.1.1 Models

The typical domain of application of separation logic is the proof of heap-manipulating programs [Rey02, O'H07, GBC$^+$07, Vaf07], and as such the models of its assertions are often taken to be a stack and a heap. Let us consider the following set of states State, and recall the definition of Stack and Heap from Chapter 2:[1]

$$\mathsf{State} \triangleq \mathsf{Stack} \times \mathsf{Heap} \qquad \mathsf{Stack} \triangleq \mathsf{Var} \rightharpoonup_{\mathit{fin}} \mathsf{Val} \qquad \mathsf{Heap} \triangleq \mathsf{Loc} \rightharpoonup_{\mathit{fin}} \mathsf{Val}$$

The composition of two states $\sigma_1 = (s_1, h_1)$ and $\sigma_2 = (s_2, h_2)$ is defined only when the domains of the two stacks and the domains of the two heaps are disjoint. When this is the case, we write $\sigma_1 \sharp \sigma_2$ and the composition $\sigma_1 \bullet \sigma_2$ is defined as

$$\sigma_1 \bullet \sigma_2 \triangleq (s_1 \uplus s_2, h_1 \uplus h_2)$$

where the disjoint union of two partial functions f and g with the same codomain S is

$$f \uplus g : \begin{array}{rcl} dom(f) \uplus dom(g) & \rightharpoonup & S \\ x & \mapsto & \begin{cases} f(x) & \text{if } x \in dom(f) \\ g(x) & \text{if } x \in dom(g) \end{cases} \end{array}$$

[1] In many models of separation logic, stacks are total functions assigning values to variables, and only the heap has a built-in notion of ownership by being a partial function from locations to values [Rey02]. This gives rise to rather unpleasant side-conditions in the proof rules, and there is a tendency to treat variables as resources more systematically, that is to define the stack as a partial map as well [BCY06]. We follow the latter approach.

$E ::=$	logical expressions
$\quad X$	logical variable
$\quad \mid \mathtt{x}$	program variable
$\quad \mid v$	value
$\quad \mid E + E \mid E - E \mid \cdots$	arithmetic operations
$\phi ::=$	formulas
$\quad E = E$	boolean expression
$\quad \mid \mathsf{emp}_s$	empty stack
$\quad \mid \mathsf{emp}_h$	empty heap
$\quad \mid \mathsf{own}(\mathtt{x})$	ownership of a single variable
$\quad \mid E \mapsto E$	singleton heap
$\quad \mid \phi * \phi$	separating conjunction
$\quad \mid \phi \mathrel{-\!\!*} \phi$	magicwand
$\quad \mid \phi \wedge \phi$	classical conjunction
$\quad \mid \neg\phi$	classical negation
$\quad \mid \forall X.\, \phi$	first-order quantification

Figure 4.1: Syntax of CSL assertions ($X \in \mathsf{LVar}$, $\mathtt{x} \in \mathsf{Var}$, $v \in \mathsf{Val}$).

4.1.2 Syntax

The assertions of separation logic, grouped in Figure 4.1, describe what is considered to be owned in a given state (that is the domains of the stack and heap of the state), as well as the values of program variables and the contents of memory cells. Logical expressions are identical to program expressions (we will use E to denote both), except for the fact that they may use logical variables as atoms in addition to program variables and values. The set of logical variables is denoted by LVar, and its elements by $X, Y, \ldots$

4.1.3 Semantics

The semantics of assertions is given by a *satisfaction* relation (or *forcing* relation) between a state σ, a map $i : \mathsf{LVar} \to \mathsf{Val}$ that assigns values to logical variables, and a formula ϕ of separation logic. The forcing relation is presented in Figure 4.2. Logical expressions are evaluated on both the program stack s and the logical "stack" i:

$$[\![v]\!]_{s,i} \triangleq v \qquad [\![\mathtt{x}]\!]_{s,i} \triangleq s(\mathtt{x}) \qquad [\![X]\!]_{s,i} \triangleq i(X) \qquad [\![E_1 + E_2]\!]_{s,i} \triangleq [\![E_1]\!]_{s,i} + [\![E_2]\!]_{s,i}$$

$$[\![E_1 - E_2]\!]_{s,i} \triangleq [\![E_1]\!]_{s,i} - [\![E_2]\!]_{s,i} \qquad\qquad \ldots$$

Let us go through the table of Figure 4.2. The comparison of two expressions $E_1 = E_2$ is true if they evaluate to the same value on s and i. It is false if they evaluate to different values or if E_1 or E_2 mentions variables not present in s. The assertions emp_s

$$\begin{array}{rll}
(s,h),i \models E_1 = E_2 & \text{iff} & var(E_1,E_2) \in dom(s) \mathbin{\&} \llbracket E_1 \rrbracket_{s,i} = \llbracket E_2 \rrbracket_{s,i} \\
(s,h),i \models \mathsf{emp}_s & \text{iff} & dom(s) = \varnothing \\
(s,h),i \models \mathsf{emp}_h & \text{iff} & dom(h) = \varnothing \\
(s,h),i \models \mathsf{own}(\mathtt{x}) & \text{iff} & dom(s) = \{\mathtt{x}\} \\
(s,h),i \models E_1 \mapsto E_2 & \text{iff} & var(E_1,E_2) \in dom(s) \\
 & & \mathbin{\&} dom(h) = \{\llbracket E_1 \rrbracket_{s,i}\} \mathbin{\&} h(\llbracket E_1 \rrbracket_{s,i}) = (\llbracket E_2 \rrbracket_{s,i}) \\
\sigma,i \models \phi_1 * \phi_2 & \text{iff} & \exists \sigma_1,\sigma_2.\, \sigma = \sigma_1 \bullet \sigma_2 \mathbin{\&} \sigma_1,i \models \phi_1 \mathbin{\&} \sigma_2,i \models \phi_2 \\
\sigma,i \models \phi_1 \mathrel{-\!\!*} \phi_2 & \text{iff} & \forall \sigma'.\, \text{if } \sigma \sharp \sigma' \mathbin{\&} \sigma',i \models \phi_1 \text{ then } \sigma \bullet \sigma',i \models \phi_2 \\
\sigma,i \models \phi_1 \wedge \phi_2 & \text{iff} & \sigma,i \models \phi_1 \mathbin{\&} \sigma,i \models \phi_2 \\
\sigma,i \models \neg\phi & \text{iff} & \sigma,i \not\models \phi \\
\sigma,i \models \forall X.\, \phi & \text{iff} & \forall v \in \mathsf{Val}.\, \sigma,[i \mid X : v] \models \phi
\end{array}$$

Figure 4.2: Semantics of concurrent separation logic's assertions.

and emp_h describe respectively the empty stack and the empty heap. The singleton stack is described by $\mathsf{own}(\mathtt{x})$ (where the value of $\mathtt{x}$ can be specified separately if need be, for instance $\mathsf{own}(\mathtt{x}) \wedge \mathtt{x} = 42$ is satisfied only by the stack $[\mathtt{x} : 42]$), and the singleton heap by $E_1 \mapsto E_2$ (contrarily to the case of the stack, this suffices to fully specify a singleton heap).

A crucial feature of separation logic is the ability to split a state into two *disjoint* parts: $\sigma, i \models \phi_1 * \phi_2$ is true if there is a splitting $\sigma_1 \bullet \sigma_2$ of σ such that each substate satisfies a subformula. As we will see in the proof rules, this allows one to easily eliminate the issues that normally arise due to potential aliasing of memory locations. The *magicwand* operator $\mathrel{-\!\!*}$ is the *adjunct* of $*$. Conjunction, negation and universal quantification are standard.

4.1.4 Derived formulas

We can derive all the usual first-order connectives from the language of Figure 4.1 and do so in Figure 4.3. Most of the formulas of Figure 4.3 should be self-explanatory. Let us detail the last two. The formula $E_1 \hookrightarrow E_2$ is true in a state where E_1 can be evaluated to an allocated location of the heap and attached to the value E_2. To negatively compare two expressions, some care is needed: one cannot simply define $E_1 \neq E_2$ as $\neg(E_1 = E_2)$, else it would also be true of states that do not hold enough resources to evaluate E_1 and E_2. This is why $E_1 = E_1$ and $E_2 = E_2$ are added to the definition. Although these formulas may seem tautological at first glance, they are not: they only hold if E_1 and E_2 can be evaluated on the current stack. With our definition, $E_1 \neq E_2$ holds if both expressions evaluate to different values on the current stack and with the current values of logical variables.

We use "–" as a wildcard; for instance, $E \mapsto -$ is read as $\exists X.\, E \mapsto X$. To lighten formulas, and following the variable as resources tradition [BCY06], we write $\mathtt{x}_1, \ldots, \mathtt{x}_n \Vdash \phi$ for $(\mathsf{own}(\mathtt{x}_1) * \cdots * \mathsf{own}(\mathtt{x}_n)) \wedge \phi$. Finally, the set $var(\phi)$ of the program variables of a formula ϕ is defined as usual by structural induction on the formula. Quantifications bind only logical variables, and there are no binders for program variables in the syntax of assertions. We sometimes write $\sigma \models \phi$ to mean that $\sigma, i \models \phi$ for all i.

$$\mathsf{true} \triangleq \forall X.\, X = X \qquad \mathsf{false} \triangleq \neg\mathsf{true} \qquad \phi_1 \vee \phi_2 \triangleq \neg(\neg\phi_1 \wedge \neg\phi_2)$$

$$\phi_1 \Rightarrow \phi_2 \triangleq (\neg\phi_1) \vee \phi_2 \qquad \phi_1 \Leftrightarrow \phi_2 \triangleq (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1) \qquad \exists X.\, \phi \triangleq \neg\forall X.\, \neg\phi$$

$$\mathsf{emp} \triangleq \mathsf{emp}_s \wedge \mathsf{emp}_h \qquad E_1 \hookrightarrow E_2 \triangleq E_1 \mapsto E_2 * \mathsf{true}$$

$$E_1 \neq E_2 \triangleq E_1 = E_1 \wedge E_2 = E_2 \wedge \neg E_1 = E_2$$

Figure 4.3: Shorthand formulas.

Examples Let us give a few examples of formulas and what they denote.

Formula	Denotation
$\mathtt{x} \mapsto 32 * \mathtt{y} \mapsto 52$	two-cell heap
$\mathtt{x} \mapsto 32 \wedge \mathtt{y} \mapsto 52$	unsatisfiable: the heap is made of one cell with conflicting values
$\mathtt{x} \mapsto - \wedge \mathtt{y} \mapsto -$	one-cell heap, $\mathtt{x} = \mathtt{y}$
$\mathtt{x} = 32 * \mathtt{x} = 32$	unsatisfiable: $\mathtt{x}$ cannot be present in the stacks on both sides of $*$

4.1.5 Inductive predicates

Finally, we need a way of describing inductive structures in the heap, for instance linked lists or binary trees. Following the tradition set by most of separation logic literature, we will not give a formal semantics to such predicates, only informal inductive definitions. Defining them more formally is of course possible, for instance using graph transformations [Dod09]. In this thesis, we will use the following singly-linked list segment and binary tree predicates:

$$\begin{aligned} \mathsf{lseg}(E, F) &\triangleq (E = F \wedge \mathsf{emp}_h) \vee (E \neq F \wedge \exists X.\, E \mapsto X * \mathsf{lseg}(X, F)) \\ \mathsf{tree}(E) &\triangleq (E = 0 \wedge \mathsf{emp}_h) \vee (\exists X, Y.\, E \mapsto l : X, r : Y * \mathsf{tree}(X) * \mathsf{tree}(Y)) \end{aligned}$$

When $F = 0$, the list segment $\mathsf{lseg}(E, F)$ is “terminated”: the list can be unfolded until there is no further successor. In this case, we may simply write $\mathsf{list}(E)$. Note that the separating conjunction between each node guarantees that the list segment is acyclic.

To define binary trees, we need a record model on the heap (because a parent needs one pointer to each of its children), which can be achieved with a slight extension of our model as explained in Chapter 2 (page 32). As for linked lists, the separating conjunctions in the definition ensure that all structures described by the formula are actual trees and not mere graphs or directed acyclic graphs.

Other, more complicated shapes, like doubly-linked lists, or lists of lists, or red-black trees of lists of trees, can be described using the same kind of definitions [Dod09]. There has been some work in the past on composite data structures [BCC$^+$07] and arrays [CDOY06] as well. We can also define predicates that describe the data along with the shape of a dynamic structure [BBL09, Rey02], for instance to prove total correctness of a sorting program. Finding more elegant and systematic ways to define inductive predicates, with or without data, is challenging, and a concern orthogonal to this thesis and more akin to the domain of *shape analysis*.

4.2 Proof system

4.2.1 An extension of Floyd-Hoare logic

In addition to the assertion language, separation logic is an extension of Floyd-Hoare logic (or simply Hoare logic), a logic used to specify and prove programs [Hoa69, Flo67]. The term "separation logic" usually refers both to the assertion language and to the proof system that uses it. Floyd-Hoare logic statements are called *Hoare triples*, and have the form $\{\phi\}\ p\ \{\psi\}$, where ϕ and ψ are formulas (in our case, separation logic formulas), and p a program. A Hoare triple $\{\phi\}\ p\ \{\psi\}$ has the following informal semantic reading in the *partial correctness* setting[2]:

> For any state σ satisfying ϕ, if $p, \sigma \Rightarrow^* \texttt{skip}, \sigma'$ then σ' satisfies ψ.

These triples are proved formally by constructing a derivation tree from a set of rules. There is typically one such rule per programming construct, and logical rules to rewrite the formulas of a triple.

Separation logic is tuned towards a *fault-avoiding* interpretation of triples. The goal is to obtain the following soundness statement:

> If $\{\phi\}\ p\ \{\psi\}$ is provable then, for any state σ satisfying the formula ϕ, $p, \sigma \not\Rightarrow^*$ **error** and if $p, \sigma \Rightarrow^* \texttt{skip}, \sigma'$ then σ' satisfies ψ.

4.2.2 Small axioms

One of the main features of separation logic is the soundness of the *frame rule*:

$$\textsc{Frame}\ \frac{\{\phi\}\ p\ \{\phi'\}}{\{\phi * \psi\}\ p\ \{\phi' * \psi\}}$$

This rule states that, whenever the execution of a program from a certain heap does not produce memory faults, it will not produce memory faults from a bigger heap either (a property called *safety monotonicity*), and the extra piece of heap will remain untouched by the program throughout its execution (we say that p is *local*). In traditional models of separation logic where stacks are total functions (whereas ours are partial), the soundness of the frame rule requires the following side-condition to be added:

$$\mathit{freevar}(\psi) \cap \mathit{modifs}(p) = \varnothing\ .$$

[2] In the full correctness setting, we would also require that p does not diverge on σ. The focus of this thesis is on partial, fault-avoiding correctness.

Remark 4.1 *Of course, the "classical" frame rule using $\wedge$ would be unsound:*

$$\textsc{And-frame-wrong}\quad \frac{\{\phi\}\ p\ \{\phi'\}}{\{\phi \wedge \psi\}\ p\ \{\phi' \wedge \psi\}}\ \mathit{freevar}(p) \cap \mathit{freevar}(\psi) = \varnothing$$

This is due to potential aliasing. If we augmented Floyd-Hoare logic with this erroneous frame rule, we could derive the following false specification of the mutation of a memory cell:

$$\frac{\{\mathtt{x} \mapsto 32\}\ \mathtt{[x]\ =\ 52}\ \{\mathtt{x} \mapsto 52\}}{\{\mathtt{x} \mapsto 32 \wedge \mathtt{y} \mapsto 32\}\ \mathtt{[x]\ =\ 52}\ \{\mathtt{x} \mapsto 52 \wedge \mathtt{y} \mapsto 32\}}$$

With the frame rule, one can restrict the specification of programs to only be about the cells they actually access (their *footprint* [RG08]). This also means that we can give the axioms for atomic commands in a very minimalistic way. Thus, the following triple, which states precisely and unambiguously the portion of state that is necessary for the command to successfully fire, can be used as a so-called *small* axiom for the mutate command:

$$\textsc{Mutate}\quad \frac{}{\{\mathit{var}(E_1, E_2) \Vdash E_1 \mapsto -\}\ \mathtt{[}E_1\mathtt{]\ =\ }E_2\ \{\mathit{var}(E_1, E_2) \Vdash E_1 \mapsto E_2\}}$$

4.2.3 Proof rules

The proof rules of concurrent separation logic, adapted to our setting with variables as resources but without permissions and to the programming language Hip, are split into the axioms for atomic commands of Figure 4.4, the rules for programming constructs of Figure 4.5 and the logical rules of Figure 4.6.

Let us describe each rule.

Skip The program has already reached its final state. Using this axiom and the frame rule, one can derive $\{\phi\}$ `skip` $\{\phi\}$ for any ϕ.

Assume If the test B is true, the program terminates. The stack must contain the variables of B.

Assign If the expression E can be read and the variable `x` can be written, the value of `x` is updated to be the same as E.

Lookup and Mutate Similar, except that E (resp. E_1) must point to an allocated location.

New and Dispose Change an empty heap into a single cell and vice-versa.

Sequence This is the classical Floyd-Hoare rule for composing programs sequentially: the post-condition of a program is taken as a precondition for its continuation.

$$\textsc{Skip}\quad \frac{}{\{\mathsf{emp}\}\ \mathtt{skip}\ \{\mathsf{emp}\}}$$

$$\textsc{Assume}\quad \frac{}{\{\mathit{var}(B) \Vdash \mathsf{emp}_h\}\ \mathtt{assume}(B)\ \{\mathit{var}(B) \Vdash B \wedge \mathsf{emp}_h\}}$$

$$\textsc{Assign}\quad \frac{}{\{\mathtt{x}, \mathit{var}(E) \Vdash E = X \wedge \mathsf{emp}_h\}\ \mathtt{x}\ \mathtt{=}\ E\ \{\mathtt{x}, \mathit{var}(E) \Vdash \mathtt{x} = X \wedge \mathsf{emp}_h\}}$$

$$\textsc{Lookup}\quad \frac{}{\{\mathtt{x}, \mathit{var}(E) \Vdash E = Y \wedge Y \mapsto X\}\ \mathtt{x}\ \mathtt{=}\ \mathtt{[}E\mathtt{]}\ \{\mathtt{x}, \mathit{var}(E) \Vdash Y \mapsto X \wedge \mathtt{x} = X\}}$$

$$\textsc{Mutate}\quad \frac{}{\{\mathit{var}(E_1, E_2) \Vdash E_1 \mapsto - \wedge E_2 = X\}\ \mathtt{[}E_1\mathtt{]}\ \mathtt{=}\ E_2\ \{\mathit{var}(E_1, E_2) \Vdash E_1 \mapsto X\}}$$

$$\textsc{New}\quad \frac{}{\{\mathtt{x} \Vdash \mathsf{emp}_h\}\ \mathtt{x}\ \mathtt{=}\ \mathtt{new()}\ \{\mathtt{x} \Vdash \mathtt{x} \mapsto -\}}$$

$$\textsc{Dispose}\quad \frac{}{\{\mathit{var}(E) \Vdash E \mapsto -\}\ \mathtt{dispose}(E)\ \{\mathit{var}(E) \Vdash \mathsf{emp}_h\}}$$

Figure 4.4: Proof rules for atomic commands.

$$\textsc{Sequence}\quad \frac{\{\phi\}\ p\ \{\phi'\} \qquad \{\phi'\}\ p'\ \{\psi\}}{\{\phi\}\ p; p'\ \{\psi\}} \qquad \textsc{Parallel}\quad \frac{\{\phi\}\ p\ \{\psi\} \qquad \{\phi'\}\ p'\ \{\psi'\}}{\{\phi * \phi'\}\ p \parallel p'\ \{\psi * \psi'\}}$$

$$\textsc{Choice}\quad \frac{\{\phi\}\ p\ \{\psi\} \qquad \{\phi\}\ p'\ \{\psi\}}{\{\phi\}\ p + p'\ \{\psi\}} \qquad \textsc{Star}\quad \frac{\{\phi\}\ p\ \{\phi\}}{\{\phi\}\ p^*\ \{\phi\}}$$

$$\textsc{Local}\quad \frac{\{\mathsf{own}(\mathtt{z}) * \phi\}\ p[\mathtt{x} \leftarrow \mathtt{z}]\ \{\mathsf{own}(\mathtt{z}) * \psi\}}{\{\phi\}\ \mathtt{local}\ \mathtt{x}\ \mathtt{in}\ p\ \{\psi\}}\ \mathtt{z} \notin \mathit{freevar}(\phi, p, \psi)$$

Figure 4.5: Proof rules for programming constructs.

$$\textsc{Frame}\ \frac{\{\phi\}\ p\ \{\phi'\}}{\{\phi * \psi\}\ p\ \{\phi' * \psi\}} \qquad \textsc{Weakening}\ \frac{\phi' \Rightarrow \phi \qquad \{\phi\}\ p\ \{\psi\} \qquad \psi \Rightarrow \psi'}{\{\phi'\}\ p\ \{\psi'\}}$$

$$\textsc{Conjunction}\ \frac{\{\phi_1\}\ p\ \{\psi_1\} \qquad \{\phi_2\}\ p\ \{\psi_2\}}{\{\phi_1 \wedge \phi_2\}\ p\ \{\psi_1 \wedge \psi_2\}} \qquad \textsc{Disjunction}\ \frac{\{\phi_1\}\ p\ \{\psi_1\} \qquad \{\phi_2\}\ p\ \{\psi_2\}}{\{\phi_1 \vee \phi_2\}\ p\ \{\psi_1 \vee \psi_2\}}$$

$$\textsc{Existential}\ \frac{\{\phi\}\ p\ \{\psi\}}{\{\exists X.\,\phi\}\ p\ \{\exists X.\,\psi\}} \qquad \textsc{Renaming}\ \frac{\{\phi\}\ p\ \{\psi\}}{\{\phi[\mathtt{x} \leftrightarrow \mathtt{y}]\}\ p[\mathtt{x} \leftrightarrow \mathtt{y}]\ \{\psi[\mathtt{x} \leftrightarrow \mathtt{y}]\}}$$

Figure 4.6: Logical proof rules.

Parallel The rule for parallel composition accounts for *disjoint* concurrency: one has to be able to partition the state into two disjoint substates that form valid respective preconditions for each of the two threads. The resulting post-conditions are glued together to form the post-condition of the parallel composition. Had we not considered stacks as partial functions, there would be numerous side-conditions attached to this rule [Bro07, BCY06].

Choice and Star Standard.

Local The rule allocates a new variable z which is used to prove p. The variable must be tracked throughout the proof of p and still be present at the end, when it is disposed.

Frame See earlier discussion.

Weakening This is a standard Floyd-Hoare rule, whose soundness follows directly from the definition of what a valid Hoare triple is.

Conjunction, Disjunction, Existential and Renaming Standard.

We give below the soundness of separation logic adapted to our setting. We will not prove it here, as it is not the focus of this thesis.

> For any program $p \in \textsc{Hip}$, if the triple $\{\phi\}\ p\ \{\psi\}$ is provable, then for any state σ and any interpretation i, if $\sigma, i \vDash \phi$ then $p, \sigma \not\Rightarrow^* \mathbf{error}$ and if $p, \sigma \Rightarrow^* \mathtt{skip}, \sigma'$ then $\sigma', i \vDash \psi$.

4.2.4 Conditional critical regions

As discussed in Chapter 2, there exist many programming language tools to allow the different components of a program to synchronise. Historically, the first mechanism that was

$$\textsc{Resource}\quad \dfrac{\Gamma, r : \gamma_r \vdash \{\phi\}\ p\ \{\psi\}}{\Gamma \vdash \{\phi * \gamma_r\}\ \texttt{resource r in}\ p\ \{\psi * \gamma_r\}}$$

$$\text{CCR}\quad \dfrac{\Gamma \vdash \{(\phi * \gamma_r) \wedge B\}\ p\ \{\phi' * \gamma_r\}}{\Gamma, r : \gamma_r \vdash \{\phi\}\ \texttt{with r when}\ B\ \ p\ \{\phi'\}}$$

Figure 4.7: Proof rules for conditional critical regions.

included in separation logic was conditional critical regions [O'H04] (CCR for short, see page 13). In this case, proofs keep track of the resources available to a program via a *context*. A context Γ is a list of pairs $r : \gamma_r$ that associates resource identifiers to separation logic formulas: their *invariants*. Resource invariants must hold when the resource is created, can be assumed every time a resource is successfully acquired, and have to be asserted when the resource is released. This behaviour is enforced by the rules of Figure 4.7 that are added to separation logic's proof system. The sequents are extended to support contexts, and are now of the form $\Gamma \vdash \{\phi\}\ p\ \{\psi\}$. Previous rules can be adapted to this new setting by adding Γ uniformly on the left-hand side of each turnstile. The resulting logic is called *concurrent separation logic* (CSL). Though it is but one concurrent version of separation logic, it has the primacy of being the first one.

Note that the CCR rule prevents the same resource from being acquired twice since it removes the resource from the proof environment when entering the critical section. This catches some of the possible deadlocks of programs.

Precision For the proof system to be sound, the resource invariants must be *precise* formulas. Precise formulas unambiguously identify a piece of state.

Definition 4.1 (Precise formulas) *A formula ϕ is* precise *if for all instantiations i of logical variables and all state σ, there is* at most *one substate $\sigma' \preceq \sigma$ such that $\sigma', i \vDash \phi$.*

In particular, true is not precise, and neither is $(\mathtt{x} \Vdash \mathsf{emp}_h) \vee (\mathtt{x} \Vdash \mathtt{x} \mapsto -)$. However, emp, $\mathtt{x} \Vdash \mathsf{emp}_h$ and $\mathtt{x} \Vdash \mathtt{x} \mapsto -$ are all precise. Without this requirement, soundness is lost, as first pointed out by Reynolds. Let us show here why, by rephrasing O'Hearn [O'H07, Section 11]. Let $\mathsf{one} \triangleq \mathsf{emp}_s \wedge 10 \mapsto -$ and suppose that the resource $\mathtt{r}$ has been assigned true as invariant, which is imprecise. Consider the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathsf{emp}\}\ \texttt{skip}\ \{\mathsf{emp}\}}\ \textsc{Skip}}{\{\mathsf{emp} * \mathsf{true}\}\ \texttt{skip}\ \{\mathsf{emp} * \mathsf{true}\}}\ \textsc{Frame}}{\{(\mathsf{emp} \vee \mathsf{one}) * \mathsf{true}\}\ \texttt{skip}\ \{\mathsf{emp} * \mathsf{true}\}}\ \textsc{Weakening}}{\{\mathsf{emp} \vee \mathsf{one}\}\ \texttt{with r \{ skip \}}\ \{\mathsf{emp}\}}\ \text{CCR}$$

From this derivation, one can prove the following two triples by either applying the proof above right away or by framing one first:

$$\dfrac{\{\mathsf{emp} \lor \mathsf{one}\}\ \mathtt{with}\ \mathtt{r}\ \{\ \mathtt{skip}\ \}\ \{\mathsf{emp}\}}{\{\mathsf{one}\}\ \mathtt{with}\ \mathtt{r}\ \{\ \mathtt{skip}\ \}\ \{\mathsf{emp}\}}\ \textsc{Weakening}$$

$$\dfrac{\dfrac{\dfrac{\{\mathsf{emp} \lor \mathsf{one}\}\ \mathtt{with}\ \mathtt{r}\ \{\ \mathtt{skip}\ \}\ \{\mathsf{emp}\}}{\{\mathsf{emp}\}\ \mathtt{with}\ \mathtt{r}\ \{\ \mathtt{skip}\ \}\ \{\mathsf{emp}\}}\ \textsc{Weakening}}{\{\mathsf{emp} * \mathsf{one}\}\ \mathtt{with}\ \mathtt{r}\ \{\ \mathtt{skip}\ \}\ \{\mathsf{emp} * \mathsf{one}\}}\ \textsc{Frame}}{\{\mathsf{one}\}\ \mathtt{with}\ \mathtt{r}\ \{\ \mathtt{skip}\ \}\ \{\mathsf{one}\}}\ \textsc{Weakening}$$

The conclusions of these two derivations are incompatible. The first says that ownership of the single cell is swallowed up by the resource, while the second says that it is kept by the program. Applying the conjunction rule with these two conclusions gives us as conclusion $\{\mathsf{one} \land \mathsf{one}\}\ \mathtt{with}\ \mathtt{r}\ \{\ \mathtt{skip}\ \}\ \{\mathsf{one} \land \mathsf{emp}\}$, and finally, using the rule of weakening:

$$\{\mathsf{one}\}\ \mathtt{with}\ \mathtt{r}\ \{\ \mathtt{skip}\ \}\ \{\mathsf{false}\}$$

This suggests that the above program loops, which is not the case. For a more in-depth account of this counter example and the possible remedies, see the paper by O'Hearn [O'H07, Section 11]. To summarise its conclusions, one can abandon the conjunction rule to restore soundness but, more often than not, resource invariants are required to be precise in concurrent separation logic and its variants and extensions [O'H07, Bro07, GBC$^+$07, Vaf07]. We adopt the latter restriction.

Soundness Brookes has proved [Bro07] that the soundness result stated earlier still holds for concurrent separation logic with conditional critical regions. A key intuition to the proof is again the separation property, exposed page 18. Let us show how it affects the soundness statement. Given a resource context $\Gamma = \mathtt{r}_1 : \gamma_1, \dots, \mathtt{r}_n : \gamma_n$ and following Brookes' notations, we write $\mathit{inv}(\Gamma)$ for the spatial conjunction of the invariants of Γ (we write $\circledast$ for the iterated version of $*$):

$$\mathit{inv}(\Gamma) \triangleq \bigcircledast_{i=1}^{n} \gamma_i$$

Adapted to our setting, Brookes' statement of soundness becomes

> If $\Gamma \vdash \{\phi\}\ p\ \{\psi\}$ is provable then for any state σ satisfying the formula $\phi * \mathit{inv}(\Gamma), p, \sigma \not\Rightarrow^* \mathbf{error}$ and if $p, \sigma \Rightarrow^* \mathtt{skip}, \sigma'$ then σ' satisfies $\psi * \mathit{inv}(\Gamma)$.

Example We can now prove that the following example is race-free. The global variable x is shared between two threads that both increment it within a critical region (this example was presented on page 14).

```
$\varnothing \vdash \{\mathsf{own}(\mathtt{x}) \wedge \mathsf{emp}_h\}$
   resource r;
   $\mathtt{r} : \mathsf{own}(\mathtt{x}) \vdash \{\mathsf{emp}\}$ with r { x++; } ∥ with r { x++; } $\{\mathsf{emp}\}$
   $\{\mathsf{own}(\mathtt{x}) \wedge \mathsf{emp}_h\}$
```

Each individual thread can be proved as follows:

```
$\mathtt{r} : \mathsf{own}(\mathtt{x}) \vdash \{\mathsf{emp}\}$
          with r {
            $\varnothing \vdash \{\mathsf{own}(\mathtt{x}) \wedge \mathsf{emp}_h\}$ x++; $\{\mathsf{own}(\mathtt{x}) \wedge \mathsf{emp}_h\}$
          }
          $\{\mathsf{emp}\}$
```

Their parallel composition is proved using the logical equivalence $\mathsf{emp} \Leftrightarrow \mathsf{emp} * \mathsf{emp}$.[3]

[3] Note that we did not prove that the final result of executing the whole program is to increment x by 2. One can use auxiliary variables (à la Owicki and Gries [OG76]) to achieve this.

CHAPTER 5

Proving Copyless Message Passing

In this chapter, we show how to apply the local reasoning ideas of the previous chapter to the copyless message-passing language with endpoints in the heap HIPMP (presented in Chapter 2). The resulting logic marries contracts (studied in Chapter 3) and separation logic within a proof system that discharges parts of the verification of a program to the verification of its contracts. The contracts are annotations in the source code and are used to abstract the contents of the communication buffers in the models of the logic, as well as to be able to assert further safety properties about communications.

Our analysis of a program p follows these steps:

1. Decorate p with contracts at each `open` instruction.

2. Prove a specification $\{\phi\}\ p\ \{\psi\}$, which ensures that p does not fault on memory accesses and obeys its contracts.

3. Prove that the contracts are fault-free and leak-free, in which case p does not fault on message receptions and does not close channels with undelivered messages.

5.1 Proof system

5.1.1 Session states

Let us introduce *session states*, which will be the models of our logic. The memory model of separation logic is augmented with an endpoint heap. Instead of attaching words of message tags and values to endpoints like in Chapter 2, we abstract away from the contents of the buffers and attach a contract and a control state to each endpoint. More precisely, a channel is associated to a contract, and each of its endpoints is associated to one of the two roles 1 or 2 in the dialogue system corresponding to the contract, and a current control state. This gives rise to concise predicates to describe the states of endpoints and, as we will see in the next chapter, allows one to prove that programs respect their protocols directly into the logic. The logical states, or *session states*, are elements $\dot{\sigma} = (s, h, \dot{k})$ of $\dot{\mathsf{State}}$:

$$\dot{\mathsf{State}} \triangleq \mathsf{Stack} \times \mathsf{Heap} \times \mathsf{Session} \qquad \mathsf{Stack} \triangleq \mathsf{Var} \rightharpoonup_{\mathit{fin}} \mathsf{Val} \qquad \mathsf{Heap} \triangleq \mathsf{Cell} \rightharpoonup_{\mathit{fin}} \mathsf{Val}$$

$$\mathsf{Session} \triangleq \mathsf{Endpoint} \rightharpoonup_{\mathit{fin}} \mathsf{Endpoint} \times \mathsf{Ctt}^{\mathsf{det}} \times \mathsf{Role} \times \mathsf{Control}$$

In the definition above, $\mathsf{Control}$ denotes the set of control states, $\mathsf{Role} \triangleq \{1, 2\}$ and $\mathsf{Ctt}^{\mathsf{det}}$ denotes the set of *deterministic* contracts. Forbidding non-deterministic contracts simplifies the presentation: non-deterministic contracts would require the logic to keep track of a set of possible control states for each endpoint instead of a single control state. Although we do not give here a full formal account of non-deterministic contracts, we will show how to extend our logic and proof system to support them in Section 5.3.1. These extensions are implemented in our tool Heap-Hop, which is not restricted to deterministic contracts.

As for closed states, we define the peer $mate(\dot{k}, \varepsilon)$ of an endpoint ε in the domain of a session endpoint heap $\dot{k}$: if $\dot{\sigma} = (s, h, \dot{k})$, $\varepsilon \in dom(\dot{k})$ and $\dot{k}(\varepsilon) = (\varepsilon', -, -, -)$, we write $mate(\dot{k}, \varepsilon)$ or $mate(\dot{\sigma}, \varepsilon)$ for ε'.

We only consider well-formed states, namely states $(s, h, \dot{k})$ which satisfy:

$$\forall \varepsilon \in dom(\dot{k}).\, mate(\dot{k}, \varepsilon) \neq \varepsilon \tag{SIrreflexive}$$

$$\begin{aligned}\forall \varepsilon \in dom(\dot{k}).\, mate(\dot{k}, \varepsilon) \in dom(\dot{k}) \Rightarrow \\ mate(\dot{k}, mate(\dot{k}, \varepsilon)) = \varepsilon\end{aligned} \tag{SInvolutive}$$

$$\forall \varepsilon_1, \varepsilon_2 \in dom(\dot{k}).\, mate(\dot{k}, \varepsilon_1) = mate(\dot{k}, \varepsilon_2) \Rightarrow \varepsilon_1 = \varepsilon_2 \tag{SInjective}$$

$$\forall \varepsilon, \varepsilon' \in dom(\dot{k}).\, \begin{cases} \dot{k}(\varepsilon) = (\varepsilon', \mathfrak{C}, r, q) \\ \dot{k}(\varepsilon') = (\varepsilon, \mathfrak{C}', r', q') \end{cases} \Rightarrow \begin{cases} \mathfrak{C}' = \mathfrak{C} \\ r' = 3 - r \end{cases} \tag{SDual}$$

$$\forall \varepsilon \in dom(\dot{k}).\, \dot{k}(\varepsilon) = (\varepsilon', (\Sigma, Q, q_0, F, T), r, q) \Rightarrow q \in Q \tag{SControl}$$

The first two rules are the session states counterparts of the (Irreflexive) and (Involutive) rules for closed states seen page 34. The third one, (SInjective), follows from (Channel) and (Involutive) in the closed model, but since we do not require the equivalent of (Channel) for session states we add it explicitly. The fourth one ensures that dual endpoints are assigned dual roles of the same contract, and the last one ensures that the control state an endpoint is said to be at is one of the control states of its contract.

The set $\dot{\mathsf{State}}$ is equipped with a composition law $\bullet$ defined as the disjoint union $\uplus$ of each of the components of the states, provided that the resulting state is well-formed:

$$(s, h, \dot{k}) \bullet (s', h', \dot{k}') \triangleq (s \uplus s', h \uplus h', \dot{k} \uplus \dot{k}')\ .$$

When $\dot{\sigma} \bullet \dot{\sigma}'$ is defined, we write $\dot{\sigma} \sharp \dot{\sigma}'$. Given two states $\dot{\sigma}$ and $\dot{\sigma}'$, $\dot{\sigma} \preceq \dot{\sigma}'$ and $\dot{\sigma}' - \dot{\sigma}$ are defined as usual (remark that if there exists $\dot{\sigma}''$ such that $\dot{\sigma} \bullet \dot{\sigma}'' = \dot{\sigma}'$ then $\dot{\sigma}''$ is unique). The empty state $(\varnothing, \varnothing, \varnothing)$ is denoted by $\dot{u}$. Session states form a partial commutative monoid. To prove this, we will need the following lemma.

Lemma 5.1 *If $\dot{\sigma} \preceq \dot{\sigma}'$ and $\dot{\sigma}'$ is well-formed then so is $\dot{\sigma}$.*

Proof Immediate by reductio ad absurdum and a case analysis on the well-formedness condition that $\dot{\sigma}$ fails to satisfy. □

Lemma 5.2 $(\dot{\mathsf{State}}, \bullet, \dot{u})$ *is a partial, commutative monoid with unit* $\dot{u}$*.*

Proof We have to show that $\dot{\mathsf{State}}$ is stable by $\bullet$ and that $\bullet$ is associative, commutative, and admits $\dot{u}$ as unit. The only delicate point is the proof of associativity.

Let us first remark that if we drop the well-formedness constraint, then associativity is immediate. This shows in particular that if $\dot{\sigma}_1 \bullet (\dot{\sigma}_2 \bullet \dot{\sigma}_3)$ and $(\dot{\sigma}_1 \bullet \dot{\sigma}_2) \bullet \dot{\sigma}_3$ are both defined then they are equal. What is left to prove is that if $\dot{\sigma}_2 \sharp \dot{\sigma}_3$ and $\dot{\sigma}_1 \sharp (\dot{\sigma}_2 \bullet \dot{\sigma}_3)$ then $\dot{\sigma}_1 \sharp \dot{\sigma}_2$ and $(\dot{\sigma}_1 \bullet \dot{\sigma}_2) \sharp \dot{\sigma}_3$; this is a consequence of the previous lemma. □

5.1.2 Assertions

Syntax We extend the assertion language of separation logic of Figure 4.1 with the following predicate to describe the endpoint heap.

$$\begin{array}{lll} \phi ::= & & \text{formulas} \\ \quad \cdots & & \text{separation logic formulas (see Figure 4.1)} \\ \quad \mid E_1 \mapsto (E_2, \mathfrak{C}, r, q) & & \text{singleton endpoint heap} \end{array}$$

Semantics The predicate above describes an endpoint heap where only the address corresponding to E_1 is allocated, and points to its peer E_2, follows contract $\mathfrak{C}$ with role r and is currently in the control state q of $\mathfrak{C}$. It moreover asserts the emptiness of the cell heap. The latter consequence serves no theoretical purpose, as we could leave the cell heap unspecified while describing the endpoint heap and vice-versa. However, both heaps are really just one heap cut in two for simplicity, which is more faithfully reflected by this design choice. Moreover, we find that it gives rise to more concise formulas in practise. Formally,

$$\begin{aligned} (s, h, \dot{k}), i \vDash E_1 \mapsto (E_2, \mathfrak{C}, r, q) \text{ iff } & \mathit{var}(E_1, E_2) \subseteq \mathit{dom}(s) \mathrel{\&} \mathit{dom}(h) = \varnothing \\ & \mathrel{\&} \mathit{dom}(\dot{k}) = \{[\![E_1]\!]_{s,i}\} \\ & \mathrel{\&} \dot{k}([\![E_1]\!]_{s,i}) = ([\![E_2]\!]_{s,i}, \mathfrak{C}, r, q) \end{aligned}$$

The semantics of the regular points-to predicate is changed to stress that the allocated location is not an endpoint, and the predicate emp_h now asserts the emptiness of both the cell and the endpoint heaps:

$$\begin{aligned} (s, h, \dot{k}), i \vDash E_1 \mapsto E_2 \text{ iff } & \mathit{var}(E_1, E_2) \subseteq \mathit{dom}(s) \mathrel{\&} \mathit{dom}(\dot{k}) = \varnothing \\ & \mathrel{\&} \mathit{dom}(h) = \{[\![E_1]\!]_{s,i}\} \mathrel{\&} h([\![E_1]\!]_{s,i}) = [\![E_2]\!]_{s,i} \\ (s, h, \dot{k}), i \vDash \mathsf{emp}_h \text{ iff } & \mathit{dom}(h) = \mathit{dom}(\dot{k}) = \varnothing \end{aligned}$$

In particular, the formula $X \mapsto Y \wedge X \mapsto (Y, \mathfrak{C}, r, q)$ is unsatisfiable: the same location cannot be both a regular cell and an endpoint. The shorthand $\mathsf{emp} \triangleq \mathsf{emp}_s \wedge \mathsf{emp}_h$ now asserts the emptiness of the stack, the cell heap, and the endpoint heap.

The semantics of all the other predicates and operators is straightforwardly adapted from Figure 4.1; their former formulation needs not be altered, but state composition and compatibility is now understood in terms of well-formed ideal states composition.

5.1.3 Rules for communication

Sending and receiving actions are a means of synchronisation between processes: even though sending is asynchronous, there is still a temporal constraint between the sending and the receiving of the same message. Thus, messages can be used to transfer ownership of pieces of state between the sender and the receiver of a message, similarly to how the effects of other forms of synchronisations are mirrored in separation logic.

From this moment on, we consider that all programs are annotated by deterministic contracts at each `open` instruction in the source code. Thus, each `(e,f) = open()` will henceforth be of the form `(e,f) = open(`$\mathfrak{C}$`)`. Intuitively, each channel that is opened by a program must declare the protocol that will take place on it. This places an implicit restriction that the contract has to be the same for each channel created at the same program point. This restriction also applies to Sing♯ and Session Java, where programs are annotated in a similar fashion.

Footprints Similarly to other synchronisation mechanisms as locks and resources, a message corresponds to a transfer of ownership. Thus, we attach to each message a formula of our separation logic, called its *footprint*[1]. Different messages can be associated with different footprints; we attach the same footprint to messages of the same tag. This is a natural choice, as the same tag is usually used for messages with the same meaning.

A footprint can mention four *parameters*, in the form of special free logical variables src, dst, rl and val.[2] When a message is sent, the proof system will check that the footprint holds for a particular instantiation of these variables: src is instantiated by the source endpoint, dst by its peer (the destination endpoint), rl by the role of the sending endpoint and val by the value of the message. When it is received, the variable that the value of the message is assigned to becomes equal to the val parameter, and dst and src are replaced according to the location of the receiving endpoint and its peer respectively. This form of reasoning is justified by the fact that the values denoted by these parameters are assured to remain constant between the time when a message is sent and the time when it is received.

For example, the footprint for a message containing the location of an allocated cell (like in the example of Figure 1.6 page 22) can be $\gamma_{\mathsf{cell}} \triangleq \mathsf{val} \mapsto -$. Similarly, sending an endpoint over itself (like in the example of Figure 1.8 page 23) can be expressed by the footprint $\gamma_{\mathsf{fin}} \triangleq \mathsf{val} \mapsto (-,-,-,-) \land \mathsf{val} = \mathsf{src}$ or $\gamma_{\mathsf{fin}} \triangleq \mathsf{val} \mapsto (\mathsf{dst},-,-,-)$. We can specify that only the first endpoints of channels following the contract $\mathfrak{C}$ will be attached to a message a, and that they will only be sent from an endpoint from the same role with the footprint $\gamma_{\mathsf{a}} \triangleq \mathsf{val} \mapsto (-,\mathfrak{C},1,-) \land \mathsf{rl} = 1$.

A *footprint context* is written Γ and is of the form $a_1 : \gamma_1, \ldots, a_n : \gamma_n$. If all the formulas γ_i are *precise* (see Definition 4.1), the context is *well-formed*. As for locks and resources,

[1] These formulas were called *invariants* in our original papers to emphasise their similarities to locks and CCRs. We felt the need to change the terminology because they do not represent the state of a shared resource that has to be maintained, but rather the ownership of a piece of state that is logically being transferred.

[2] Footprint parameters play a similar role as the lock parameters of the storable locks of Gotsman *et al.* [GBC+07].

imprecise footprints can lead to unsoundness in the proof system. We discuss this issue later in this chapter, in Section 5.3.2.

Since the set of message identifiers is fixed from the beginning, contrarily to resources identifiers that can be declared within a program, the context is actually fixed throughout the proof. We reflect this in our notations by making it a subscript of the turnstile: a triple provable under a context Γ is written $\vdash_\Gamma \{\phi\}\ p\ \{\psi\}$.

Finally, when $\mathfrak{C} = (\Sigma, Q, q_0, F, T)$, we write $succ(\mathfrak{C}, r, q, \tau)$ for the state q' such that $q \xrightarrow{\tau} q'$ is a valid transition of the contract $\mathfrak{C}$ in the role r, that is either $r = 1$ and $(q, \tau, q') \in T$ or $r = 2$ and $(q, \tau, q') \in \bar{T}$. There is at most one such q' since all the contracts we consider are deterministic.

Proof system Let us present our proof system for the HipMP language. The rules of separation logic presented in the previous chapter (Figures 4.4, 4.5 and 4.6) are modified by uniformly adding $\vdash_\Gamma$ to the left of each Hoare triple. The rules for channel creation and destruction and for communications are given in Figure 5.1. We write $\gamma(E_1, E_2, E_3, E_4)$ for $\gamma[\mathsf{src}{\leftarrow}E_1][\mathsf{dst}{\leftarrow}E_2][\mathsf{rl}{\leftarrow}E_3][\mathsf{val}{\leftarrow}E_4]$.

Unfortunately, the variables as resource model without permissions forces us to perform some squirming around to read the same variable in two different parts of a formula separated by a separating conjunction: each time this situation arises, we have to first store the value of the incriminated variable into a logical variable and use the latter to refer to it. This is for the greater good, as it will lighten the notations of our upcoming proof of soundness. For instance, the formula $\mathtt{x} \mapsto \mathtt{y} * \mathtt{y} \mapsto \mathtt{x}$ is inconsistent (a variable may not appear in both the domains of disjoint stacks), and one should rather write $\mathtt{x} = X \wedge \mathtt{y} = Y \wedge (X \mapsto Y * Y \mapsto X)$ to say that x and y point to each other.

Let us review the five new rules, and how they ensure that both ownership and the contracts are respected.

Open The rule is similar to New: starting from an empty heap, it allocates two endpoint locations. They are initialised according to the contract that decorates the `open` command, and to point to each other.

Close The rule verifies that the two expressions given as arguments point to two endpoints that form a channel (they point to each other) and that are both in the same final state of the contract, hence that the whole channel is in a final state according to the definition of dialogue systems associated to contracts. If it is the case, the channel is deallocated, and the heap is considered empty.

Send Let us consider a simplified version of this rule first, in the case where the footprint does not mention the sending endpoint:

Send1

$$\frac{succ(\mathfrak{C}, r, q, !a) = q'}{\begin{array}{ll} & \{E_1 = X \wedge E_2 = Y \wedge X \mapsto (X', \mathfrak{C}, r, q) * \gamma_a(X, X', r, Y)\} \\ \vdash_\Gamma & \mathtt{send(a,}E_1\mathtt{,}E_2\mathtt{)} \\ & \{E_1 \mapsto (X', \mathfrak{C}, r, q')\} \end{array}}$$

OPEN

$$\frac{q_0 = \mathit{init}(\mathfrak{C})}{\begin{array}{ll} & \{\mathtt{e},\mathtt{f} \Vdash \mathsf{emp}_h\} \\ \vdash_\Gamma & (\mathtt{e},\mathtt{f})\ =\ \mathtt{open}(\mathfrak{C}) \\ & \{\mathtt{e},\mathtt{f} \Vdash \mathtt{e} = X \land \mathtt{f} = Y \land (X \mapsto (Y,\mathfrak{C},1,q_0) * Y \mapsto (X,\mathfrak{C},2,q_0))\} \end{array}}$$

CLOSE

$$\frac{q_f \in \mathit{finals}(\mathfrak{C})}{\begin{array}{ll} & \left\{ \mathit{var}(E_1,E_2) \Vdash \begin{array}{c} E_1 = X_1 \land E_2 = X_2 \\ \land (X_1 \mapsto (X_2,\mathfrak{C},1,q_f) * X_2 \mapsto (X_1,\mathfrak{C},2,q_f)) \end{array} \right\} \\ \vdash_\Gamma & \mathtt{close}(E_1,E_2) \\ & \{\mathit{var}(E_1,E_2) \Vdash \mathsf{emp}_h\} \end{array}}$$

SEND

$$\frac{\mathit{succ}(\mathfrak{C},r,q,!a) = q'}{\begin{array}{ll} & \left\{ \begin{array}{l} \quad E_1 = X \land E_2 = Y \\ \land (X \mapsto (X',\mathfrak{C},r,q) * (X \mapsto (X',\mathfrak{C},r,q') \mathrel{-\!\!*} \gamma_a(X,X',r,Y) * \phi)) \end{array} \right\} \\ \vdash_\Gamma & \mathtt{send}(\mathtt{a},E_1,E_2) \\ & \{\phi\} \end{array}}$$

CHANNELDISPATCH

$$\frac{\begin{array}{c} \phi \Rightarrow \bigwedge_{i=1}^{n} (E_i = Y \land \mathtt{x}_i = \mathrm{x}_i) * \mathsf{true} \\ \{a_i \mid i \in \{1,\ldots,n\}\} = \{a \mid \exists q'.\, \mathit{succ}(\mathfrak{C},r,q,?a) = q'\} \\ \forall i.\, \exists q'. \left\{ \begin{array}{l} \quad \mathit{succ}(\mathfrak{C},r,q,?a_i) = q' \\ \&\ \vdash_\Gamma \{\mathtt{x}_i = Z \land (Y \mapsto (Y',\mathfrak{C},r,q') * \phi * \gamma_{a_i}(Y',Y,3-r,Z))\}\ p_i\ \{\psi\} \end{array} \right. \end{array}}{\vdash_\Gamma \{Y \mapsto (Y',\mathfrak{C},r,q) * \phi\} \mathop{\Box}_{i=1}^{n} \mathtt{x}_i\ =\ \mathtt{receive}(\mathtt{a}_i,E_i) : p_i\ \{\psi\}} \ \dagger$$

$$^\dagger Z \notin \mathit{freevar}(\phi)$$

EXTCHOICE

$$\frac{\vdash_\Gamma \{\phi\}\ p\ \{\psi\} \qquad \vdash_\Gamma \{\phi\}\ p'\ \{\psi\}}{\vdash_\Gamma \{\phi\}\ p \,\Box\, p'\ \{\psi\}}$$

Figure 5.1: Proof rules for copyless message passing.

This is an instance of the SEND rule where $\phi = E_1 \mapsto (X', \mathfrak{C}, r, q')$. In this version, the precondition requires the sending endpoint and the message footprint to be present in the current state, and the message to be authorised by the contract in the current control state of the endpoint. The footprint of the message has been instantiated with the correct parameters: X for the source endpoint src, X' for the destination endpoint dst, r for the role rl and Y for the value val. As expected, ownership of the message's footprint is lost after sending, and the control state of the endpoint is updated according to the contract. Thus, a proved program cannot access the pieces of state attached to a message after it has been sent; only the recipient of the message will be able to further access it, once it has received the message. The following program is in particular not provable with $\Gamma = \text{cell} : \mathsf{val} \mapsto -$ and $succ(\mathfrak{C}, r, q, !\mathsf{cell}) = q'$.

```
{x,y,e ⊩ x ↦ − ∗ e ↦ (−,ℭ,r,q)}
send(cell,e,x);
{x,y,e ⊩ e ↦ (−,ℭ,r,q′)}
y = [x];
```

Swapping the two lines removes any possibility for a race and allows the proof to go through:

```
{x,y,e ⊩ x ↦ − ∗ e ↦ (−,ℭ,r,q)}
y = [x];
{x,y,e ⊩ x ↦ y ∗ e ↦ (−,ℭ,r,q)}
send(cell,e,x);
{x,y,e ⊩ e ↦ (−,ℭ,r,q′)}
```

If the endpoint itself is part of the message footprint, then its control state has to be updated *before* we try and verify that $\gamma_a(X, X', r, Y)$ is part of the state. Indeed, the receive rule will simply add $\gamma_a(X, X', r, Y)$ to the current state, hence it must be up-to-date with regard to the endpoint's control state at the moment it was sent. Updating the control state of the sending endpoint is the purpose of the interplay of $*$ and $-\!\!*$ in the precondition. In this case, the following rule can be derived (taking $\phi = \mathsf{emp}_h$):

SEND2

$$\frac{succ(\mathfrak{C}, r, q, !a) = q'}{\begin{array}{ll} & \left\{ \begin{array}{l} \quad E_1 = X \wedge E_2 = Y \\ \wedge (X \mapsto (X', \mathfrak{C}, r, q) * (X \mapsto (X', \mathfrak{C}, r, q') -\!\!* \gamma_a(X, X', r, Y))) \end{array} \right\} \\ \vdash_\Gamma & \mathtt{send(a,}E_1\mathtt{,}E_2\mathtt{)} \\ & \{\mathsf{emp}_h\} \end{array}}$$

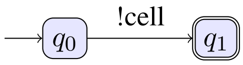
The rule SEND of Figure 5.1 is a more general rule that accounts for both cases.

CHANNELDISPATCH and EXTCHOICE These two rules govern external choice and reception. They have been made into two rules to simplify notations by treating separately each endpoint of an external choice composition. This supposes that the components of a guarded program are reordered for the needs of the proof, so as to group receives that target the same endpoint together.

In the case of receive, abiding by the contract is more intricate than in the case of send, where the action merely has to be one of the available transitions of the contract. Indeed, for a particular endpoint at control state q of a contract $\mathfrak{C}$, for instance with role 1, each receive performed on this endpoint has to be one of the available receiving actions going out from q but, conversely (and because the choice of which message is in the queue is not in the hands of the receiving endpoint), all possible receive actions should also be accounted for in the external choice composition. This is what is required by the second premise of the rule:

$$\{a_i \mid i \in \{1, \ldots, n\}\} = \{a \mid \exists q'.\, succ(\mathfrak{C}, r, q, ?a) = q'\} \ .$$

The third and last premise of the rule requires each possible continuation to be provable, with a precondition that differs from the precondition of the composition in three aspects: the control state of the receiving endpoint is updated, the piece of state attached to the message is tacked on the current state (using the spatial composition), and `x` is given a new value Z that corresponds to what the formula describing the footprint of the message states about the value of the message.

To give more intuitions on this rule, let us consider an example with a single branch. We let $\mathfrak{C}$ =  and Γ = cell : val ↦ −. The following proof is derivable:

```
{x, e ⊩ e ↦ (−, 𝔆, 1, q₀)}
x = receive(cell,e): {
   {x, e ⊩ x = Z ∧ (e ↦ (−, 𝔆, 1, q₁) ∗ Z ↦ −)}
   {x, e ⊩ e ↦ (−, 𝔆, 1, q₁) ∗ x ↦ −}
   dispose(x)
   {x, e ⊩ e ↦ (−, 𝔆, 1, q₁)}
   }
{x, e ⊩ e ↦ (−, 𝔆, 1, q₁)}
```

Soundness Although we will not prove the soundness of our proof system before the next chapter, let us give an intuition about what is proved by our logic. The originality of our soundness result is that some of the properties that a successful proof of the program entails depend on the properties of the protocols that the program follows, which can be studied separately.

We do not give here the formal statement of soundness, as it is non-trivial to express. In particular, it requires to make a connection between a closed state and a session state. In the following argument, we suppose that there is such a connection that allows us to say that a closed state satisfies a given formula of our logic.

Informally, if a triple $\vdash_\Gamma \{\phi\}\ p\ \{\psi\}$ is provable and Γ is well-formed then for all state σ and interpretation i, if there is a formula γ_0 corresponding to the concatenation of the footprints of the messages in transit in σ such that $\sigma, i \vDash \phi \ast \gamma_0$ then

1. p commits no ownership error (in particular, there is no race);

2. p follows its specified protocols;

3. if p terminates with the final state σ', then there is γ_f corresponding to the concatenation of the footprints of the messages in transit in σ' and there is ψ' corresponding to some leaked state such that $\sigma', i \vDash \psi * \gamma_f * \psi'$;

4. if all the contracts of p are fault-free then p commits no message error;

5. if all the contracts of p are leak-free and if additionally Γ is admissible (see Section 5.3.4), then there is in fact no leak and $\psi' = \mathsf{emp}$.

Our formal soundness statement for closed programs is given by Theorem 6.2 page 119, and leak freedom is discussed in Section 6.4.4.

5.2 Examples

Let us show how to prove a few typical examples. Following the notations of Heap-Hop, we give the footprint formula associated to a message identifier in brackets next to its declaration (for instance: `message cell [`$\mathsf{emp}_s \wedge \mathsf{val} \mapsto$`]`). For clarity, we allow the declaration of non-mutually recursive functions in the syntax, as they can simply be inlined at their call sites to obtain a program that conforms to the formal syntax. A function declaration will also mention its expected pre- and post-condition. The following code snippet indicates that $\{\phi\}$ `f` $\{\psi\}$ is provable:

```
f(x) [ϕ] {
  // body of f
} [ψ]
```

We also indicate intermediary steps of the derivation informally in the code comments (lines beginning with //). We use Heap-Hop's notations for contracts.

Let us begin by proving some of the examples described in Chapter 1.

5.2.1 Cell and endpoint passing

Cell passing We give a proof of the cell passing example of Figure 1.6 in Figure 5.2. As noted in Chapter 1, the message cell carries the ownership of the cell whose address is the value of the message. This is reflected in the footprint $\mathsf{emp}_s \wedge \mathsf{val} \mapsto -$. Notice that the transfer of ownership is visible in the pre and post-conditions of `put` and of `get`.[3]

As discussed when this example was introduced (see page 22), one could imagine a variation of this example where `put_cell` retains ownership of the cell pointed to by `x` and disposes it while the `get_cell` does nothing with the value it receives. This behaviour is also compatible with our proof rules of course: by changing the footprint associated to cell to emp, we reflect the fact that no ownership transfer happens and the proof goes through for the new version. The proof of `put_cell`, for instance, would become:

[3] In the case of a single cell, and if we ignore what is happening inside the endpoint heap, `send` and `receive` are similar to `dispose` and `new`.

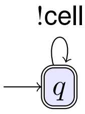
```
message cell [emp_s ∧ val ↦ −]

contract C {
  initial state q { !cell -> q'; }
  final state q' {}
}

put_cell(e,x) [e,x ⊩ e ↦ (Y, ℭ, 1, q) ∗ x ↦] {
  send(cell,e,x);
} [e,x ⊩ e ↦ (Y, ℭ, 1, q′)]

get_cell(f) [f ⊩ f ↦ (X, ℭ, 2, q)] {
  local y;

  y = receive(cell,f);
  // y,f ⊩ f ↦ (X, ℭ, 2, q′) ∗ y ↦ −
  dispose(y);
} [f ⊩ f ↦ (X, ℭ, 2, q′)]

main() [emp] {
  local x,e,f;

   x = new();
   // x,e,f ⊩ x ↦ −
   (e,f) = open(C);
   // x,e,f ⊩ x ↦ − ∗ e = X ∗ f = Y ∗ (X ↦ (Y, ℭ, 1, q) ∗ Y ↦ (X, ℭ, 2, q))
   put_cell(e,x) || get_cell(f);
   // x,e,f ⊩ e = X ∗ f = Y ∗ (X ↦ (Y, ℭ, 1, q′) ∗ Y ↦ (X, ℭ, 2, q′))
   close(e,f);
} [emp]
```

Figure 5.2: Proof of the cell transfer.

```
{e,x ⊩ e ↦ (Y, ℭ, 1, q) ∗ x ↦}
send(cell,e,x);
{e,x ⊩ e ↦ (Y, ℭ, 1, q′) ∗ x ↦}
dispose(x);
{e,x ⊩ e ↦ (Y, ℭ, 1, q′)}
```

With the original footprint $\mathsf{emp}_s \wedge \mathsf{val} \mapsto -$, this proof would not succeed as dispose(x) requires that x points to an allocated memory cell in its precondition.

As far as the communication protocol is concerned, we could have used any other specification that allows to send and receive the message cell and lead to a final configuration, for instance the following contract:

$\mathfrak{C}'_{\mathsf{cell}} \triangleq$ 

Although the program obeys $\mathfrak{C}'_{\mathsf{cell}}$ as well as $\mathfrak{C}_{\mathsf{cell}}$, the former choice is less sensible than

```
message cell  [$\mathsf{emp}_s \land \mathsf{val} \mapsto -$]
message fin   [$\mathsf{emp}_s \land \mathsf{val} \mapsto (-, \mathfrak{C}, 1, q'') \land \mathsf{val} = \mathsf{src}$]

contract C {
  initial state q { !cell -> q'; }
  state q'        { !fin -> q''; }
  final state q'' {}
}

put_cell(e,x) [$\mathsf{e},\mathsf{x} \Vdash \mathsf{e} \mapsto (-, \mathfrak{C}, 1, q) * \mathsf{x} \mapsto$] {
  send(cell,e,x);
  send(fin,e,e);
} [$\mathsf{e},\mathsf{x} \Vdash \mathsf{emp}_h$]

get_cell(f) [$\mathsf{f} \Vdash \mathsf{f} \mapsto (-, \mathfrak{C}, 2, q)$] {
  local y,ee;

  y = receive(cell,f);
  // $\mathsf{y},\mathsf{ee},\mathsf{f} \Vdash \mathsf{f} \mapsto (-, \mathfrak{C}, 2, q') * \mathsf{y} \mapsto -$
  dispose(y);
  ee = receive(fin,f);
  // $\mathsf{y},\mathsf{ee},\mathsf{f} \Vdash \mathsf{f} \mapsto (X, \mathfrak{C}, 2, q'') * X \mapsto (-, \mathfrak{C}, 1, q'') \land X = \mathsf{ee}$
  close(ee,f);
} [$\mathsf{f} \Vdash \mathsf{emp}_h$]

main() [$\mathsf{emp}$] {
  local x,e,f;

  x = new();
  // $\mathsf{x},\mathsf{e},\mathsf{f} \Vdash \mathsf{x} \mapsto -$
  (e,f) = open(C);
  // $\mathsf{x},\mathsf{e},\mathsf{f} \Vdash \mathsf{x} \mapsto - * \mathsf{e} = X * \mathsf{f} = Y * X \mapsto (Y, \mathfrak{C}, 1, q) * Y \mapsto (X, \mathfrak{C}, 2, q)$
  put_cell(e,x) || get_cell(f);
  // $\mathsf{x},\mathsf{e},\mathsf{f} \Vdash \mathsf{emp}_h * \mathsf{emp}_h$
} [$\mathsf{emp}$]
```

Figure 5.3: Proof of the cell and endpoint transfer.

the latter, as $\mathfrak{C}'_{\mathsf{cell}}$ provides less intuition about the communication protocol followed by the program. Moreover, $\mathfrak{C}'_{\mathsf{cell}}$ is not leak-free. As we have informally seen at the end of the previous section, this means that we will not be able to prove that this program is leak-free using our logic with $\mathfrak{C}'_{\mathsf{cell}}$ as contract.

According to our informal soundness statement, the proof of Figure 5.2 shows that the cell-passing program is race-free, fault-free and leak-free. Let us now show that the same is true of the cell and endpoint passing program, first seen in Figure 1.8.

Cell and endpoint passing The proof sketch of the cell and endpoint passing program is presented Figure 5.3. As it is very similar to the previous one, let us only discuss the footprint of the fin message. This footprint uses the parameter src to specify that the endpoint in transit is the one from which the message was sent. This is a crucial element of the proof: had we chosen $\gamma'_{\mathsf{fin}} \triangleq \mathsf{emp}_s \wedge \mathsf{val} \mapsto (-, \mathfrak{C}, 1, q'')$ as a footprint, the proof of `put_cell` would still go through (since γ_{fin} implies γ'_{fin}) but the last two lines of the proof of `get_cell` would give the following failed proof attempt:

```
// $\mathsf{y,f} \Vdash \mathsf{f} \mapsto (-, \mathfrak{C}, 2, q')$
ee = receive(fin,f);
// $\mathsf{y,f,ee} \Vdash \mathsf{f} \mapsto (-, \mathfrak{C}, 2, q'') * \mathsf{ee} \mapsto (-, \mathfrak{C}, 1, q'')$
close(ee,f);
```

The rule Close cannot be applied because at this point $(\mathsf{ee}, \mathsf{f})$ is not guaranteed to form a channel: γ'_{fin} says nothing about the origin of the message, so it could point to another endpoint with the same characteristics but from a different channel not related to `f`.

Alternatively, one could have given as footprint $\mathsf{emp}_s \wedge \mathsf{val} \mapsto (\mathsf{dst}, \mathfrak{C}, 1, q'')$ to specify that the transmitted endpoint is the other end of the channel and the proof would go through. Indeed, thanks to the (LInjective) axiom,

$$\mathsf{val} \mapsto (\mathsf{dst}, \mathfrak{C}, 1, q'') \Leftrightarrow (\mathsf{val} \mapsto (-, \mathfrak{C}, 1, q'') \wedge \mathsf{val} = \mathsf{src}) \; .$$

Sending a list cell by cell The proof of the copyless cell-by-cell list sending program sketched in Figure 5.4 uses the same message footprints but a protocol based on a slightly different contract that accounts for the a priori unbounded number of cells that can be sent over the channel. The proof is quite straightforward. We use brackets in `while` constructs to specify their loops invariants.

A new kind of possible error that this program could have is to omit one of the branches of the switch receive, for instance the branch corresponding to the final message fin. This would cause the program to block (or fail depending on the semantics one gives to unspecified receptions) when `put_list` sends its endpoint at the last step, unable to receive the fin message present in the buffer. The proof system would detect this problem because the contract states that two messages may be received from state q (cell and fin) hence the program should not attempt to receive only one of them.

5.2.2 Automatic teller machine

Figure 5.5 presents the proof of a simplistic automatic teller machine (ATM) that exhibits more complex communication patterns. This example is taken from one of the first papers on session types: “Language primitives and type discipline for structured communication-based programming” by Honda *et al.* [THK94]. The functioning of the ATM is stripped down to its interactions with users and the bank, hence the lack of code for checking the PIN of the user for instance.

This program does not fit our model stricto sensu: communications between the ATM and the bank do not happen over a shared memory but rather over a communication network. However, we argue that verifying this program in the copyless setting still makes sense if we look at it from an ownership point of view. Indeed, receiving a new endpoint is equivalent

```
message cell  [$\mathsf{emp}_s \land \mathsf{val} \mapsto -$]
message fin   [$\mathsf{emp}_s \land \mathsf{val} \mapsto (-, \mathfrak{C}, 1, q') \land \mathsf{val} = \mathsf{src}$]

contract C {
  initial state q { !cell -> q; !fin -> q'; }
  final state q' {}
}

put_cell(e,x)  [$\mathtt{e}, \mathtt{x} \Vdash \mathtt{e} \mapsto (-, \mathfrak{C}, 1, q) * \mathtt{x} \mapsto$] {
  local t;

  while(x != 0)  [$\mathtt{e}, \mathtt{x}, \mathtt{t} \Vdash \mathtt{e} \mapsto (-, \mathfrak{C}, 1, q) * \mathit{list}(\mathtt{x})$] {
    // $\mathtt{e}, \mathtt{x}, \mathtt{t} \Vdash \mathtt{e} \mapsto (-, \mathfrak{C}, 1, q) * \mathtt{x} \mapsto T * \mathit{list}(T)$
    t = x->tl;
    // $\mathtt{e}, \mathtt{x}, \mathtt{t} \Vdash \mathtt{e} \mapsto (-, \mathfrak{C}, 1, q) * \mathtt{x} \mapsto T * \mathit{list}(T) * \mathtt{t} = T$
    send(cell,e,x);
    // $\mathtt{e}, \mathtt{x}, \mathtt{t} \Vdash \mathtt{e} \mapsto (-, \mathfrak{C}, 1, q) * \mathit{list}(\mathtt{t})$
    x = t;
    // $\mathtt{e}, \mathtt{x}, \mathtt{t} \Vdash \mathtt{e} \mapsto (-, \mathfrak{C}, 1, q) * \mathit{list}(\mathtt{x})$
  }
  // $\mathtt{e}, \mathtt{x}, \mathtt{t} \Vdash \mathtt{e} \mapsto (-, \mathfrak{C}, 1, q)$
  send(fin,e,e);
}  [$\mathtt{e}, \mathtt{x} \Vdash \mathsf{emp}_h$]

get_cell(f)  [$\mathtt{f} \Vdash \mathtt{f} \mapsto (X, \mathfrak{C}, 2, q)$] {
  local y, ee;

  ee = 0;
  while(ee == 0)  [$\mathtt{f}, \mathtt{y}, \mathtt{ee} \Vdash (\mathtt{ee} = 0 \land \mathtt{f} \mapsto (-, \mathfrak{C}, 2, q))$
                      $\lor(\mathtt{ee} = X \land X \mapsto (-, \mathfrak{C}, 1, q') * \mathtt{f} \mapsto (X, \mathfrak{C}, 2, q'))$] {
    switch receive {
    y = receive(cell,f): { dispose(y); }
    ee = receive(fin,f): {}
  }}
  // $\mathtt{f}, \mathtt{y}, \mathtt{ee} \Vdash \mathtt{ee} = X \land \mathtt{f} = Y \land X \mapsto (Y, \mathfrak{C}, 1, q') * Y \mapsto (X, \mathfrak{C}, 2, q')$
  close(ee,f);
}  [$\mathtt{f} \Vdash \mathsf{emp}$]

main(x)  [$\mathtt{x} \Vdash \mathsf{list}(\mathtt{x})$] {
  local e,f;

   (e,f) = open(C);
   // $\mathtt{x}, \mathtt{e}, \mathtt{f} \Vdash \mathsf{list}(\mathtt{x}) * \mathtt{e} = X * \mathtt{f} = Y * X \mapsto (Y, \mathfrak{C}, 1, q) * Y \mapsto (X, \mathfrak{C}, 2, q)$
   put_list(e,x) || get_list(f);
}  [$\mathtt{x} \Vdash \mathsf{emp}_h$]
```

Figure 5.4: Proof of the cell-by-cell list transfer.

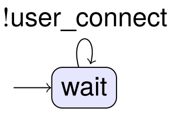
```
atm(a,b) [$\mathsf{a},\mathsf{b} \Vdash \mathsf{a} \mapsto (-, \mathfrak{C}_{\mathsf{user}}, 2, \mathsf{wait}) * \mathsf{b} \mapsto (-, \mathfrak{C}_{\mathsf{bank}}, 2, \mathsf{wait})$] {
  local k,h,id,amt;

  while (true) [$\mathsf{a},\mathsf{b},\mathsf{k},\mathsf{h},\mathsf{id},\mathsf{amt} \Vdash \mathsf{a} \mapsto (-, \mathfrak{C}_{\mathsf{user}}, 2, \mathsf{wait}) * \mathsf{b} \mapsto (-, \mathfrak{C}_{\mathsf{bank}}, 2, \mathsf{wait})$] {
    k = receive(user_connect,a);
    id = receive(pin,k);

    send(bank_init,b,id);
    h = receive(bank_connect,b);

    // $\mathsf{a},\mathsf{b},\mathsf{k},\mathsf{h},\mathsf{id},\mathsf{amt} \Vdash \mathsf{a} \mapsto (-, \mathfrak{C}_{\mathsf{user}}, 2, \mathsf{wait}) * \mathsf{b} \mapsto (-, \mathfrak{C}_{\mathsf{bank}}, 2, \mathsf{wait})$
    //                  $* \mathsf{k} \mapsto (-, \mathfrak{C}_{\mathsf{user\_session}}, 2, \mathsf{i}) * \mathsf{h} \mapsto (-, \mathfrak{C}_{\mathsf{bank\_session}}, 2, \mathsf{i})$
    switch receive {
      amt = receive(deposit_u,k): {
        send(deposit,h,id,amt);
      }
      amt = receive(withdraw_u,k): {
        send(withdraw,h,id,amt);
        switch receive {
          receive(success,h): { send(dispense,k,amt); }
          receive(failure,h): { send(overdraft,k); }
        }
      }
      receive(balance_u,k): {
        send(balance,h,id);
        amt = receive(receipt,h);
        send(receipt,k,amt);
      }
    }

    send(fin_u,k,k);
    send(fin_b,h,h);
  }
}
```

Figure 5.5: Proof of a simplistic ATM.

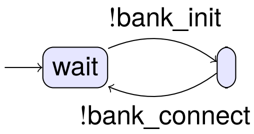
to receiving the right to use it, and sending an endpoint using the fin message is equivalent to relinquishing one's rights over it. Moreover, aside from the channels, only values are exchanged; the cell heap plays no role in this example.

The protocols of this program can once again be expressed with contracts. The contracts used to initiate the communications with the user and the bank simply wait for connection requests. The mechanisms for connecting to the user and to the bank are different: in one case, the user initiates the connection while in the other the ATM does.

$\mathfrak{C}_{\mathsf{user}} \triangleq$

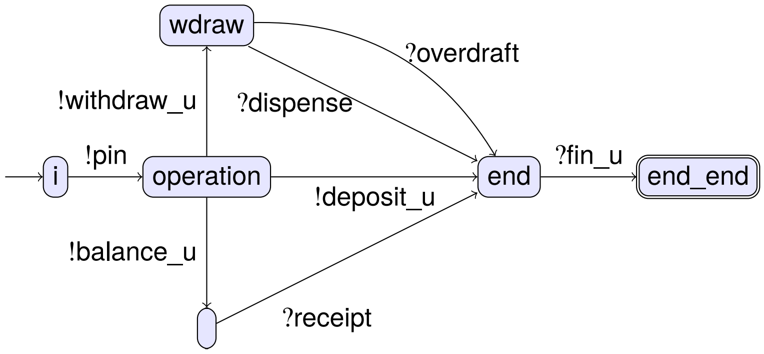


$\mathfrak{C}_{\mathsf{bank}} \triangleq$



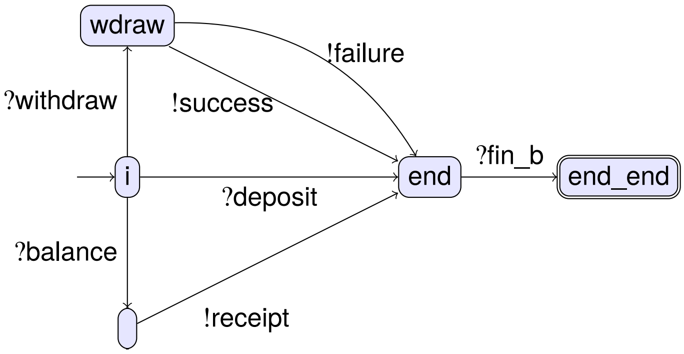
The contract between the user and the ATM follows the structure of the program:

$\mathfrak{C}_{\mathsf{user\_session}} \triangleq$



This contract points out the fact that this ATM could be improved (as noted by Honda *et al.*) to allow several operations in a row until the user quits. The protocol between the ATM and the bank abides by the following contract:

$\mathfrak{C}_{\mathsf{bank\_session}} \triangleq$



Both contracts are strikingly similar, which reflects the fact that the ATM acts as a relay between the user and the bank.

The footprints associated to the messages that open and close communication channels in this example are as follows:

$$\begin{aligned}
\gamma_{\mathsf{user\_connect}} &\triangleq \mathsf{val} \mapsto (-, \mathfrak{C}_{\mathsf{user\_session}}, 2, \mathsf{i}) \\
\gamma_{\mathsf{bank\_connect}} &\triangleq \mathsf{val} \mapsto (-, \mathfrak{C}_{\mathsf{bank\_session}}, 2, \mathsf{i}) \\
\gamma_{\mathsf{fin\_u}} &\triangleq \mathsf{val} \mapsto (-, \mathfrak{C}_{\mathsf{user\_session}}, 2, \mathsf{end\_end}) \wedge \mathsf{val} = \mathsf{src} \\
\gamma_{\mathsf{fin\_b}} &\triangleq \mathsf{val} \mapsto (-, \mathfrak{C}_{\mathsf{bank\_session}}, 2, \mathsf{end\_end}) \wedge \mathsf{val} = \mathsf{src}
\end{aligned}$$

Since all the other messages convey mere values, their associated footprint is emp.

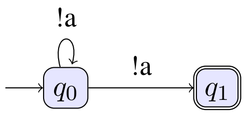
All the contracts of this example are deterministic, positional and synchronising. By Corollary 3.1, they are fault and leak-free hence as we will see in the next chapter the program itself is race, fault and leak-free. One can give a session type for this program, as did Honda *et al.* in the aforementioned paper [THK94], which would prove that it is fault-free.

5.3 Restrictions of the proof system

5.3.1 Deterministic contracts

In this chapter, as we have already pointed out, the protocols have to be expressible using deterministic contracts. Were we to allow non-deterministic contracts in our analysis, we would need the logic to keep track of the *set* of control states in which an endpoint might be in order to soundly account for non-determinism, instead of a single control state. For instance, an endpoint pointed to by x with mate X', starting in state q_0 of the contract

$\mathfrak{C} \triangleq$



could be in either one of the states q_0 or q_1 after sending a, hence would be best described by $\mathtt{x} \mapsto (X', \mathfrak{C}, 1, \{q_0, q_1\})$. As they have been described, the proof rules would allow one to prove, for a program p that sends a over x, the triple

$$\{\mathtt{x} \mapsto (X', \mathfrak{C}, 1, q_0)\}\; p\; \{\mathtt{x} \mapsto (X', \mathfrak{C}, 1, q_1)\}$$

whereas the semantics of p (defined in the next chapter) could choose to place x in the control state q_0 which, even without a formal soundness statement in mind, contradicts this post-condition.

Modifying the proof rules of communications to handle sets of control states instead of singletons is quite straightforward: one only has to extend *succ* to operate on sets of control states. The definition of this new operation $succ^{\wp}$ becomes

$$succ^{\wp}(\mathfrak{C}, r, Q, \tau) \triangleq Q' \text{ iff } \forall q' \in Q'.\, \exists q \in Q.\, succ(\mathfrak{C}, r, q, \tau) = q'\ .$$

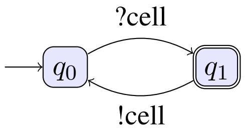
For instance, in the contract above,

$$succ^{\wp}(\mathfrak{C}, 1, \{q_0\}, !a) = \{q_0, q_1\} = succ^{\wp}(\mathfrak{C}, 1, \{q_0, q_1\}, !a)\ .$$

The proof rules for open and close are modified in the following way: Open creates endpoints in the singleton set of states $\{init(\mathfrak{C})\}$ and Close requires that, if the two endpoints are in the sets of possible control states Q_1 and Q_2 respectively and they follow contract $\mathfrak{C}$ (with appropriate roles), then $Q_1 \cap Q_2 \cap finals(\mathfrak{C})$ is not empty.

5.3.2 Precision of message footprints

If footprints are not precise formulas, a variation of Reynolds' original counter-example (described page 73) can be replayed. We have to devise a program p such that the triple $\{\mathsf{one} \vee \mathsf{emp}\}\ p\ \{\mathsf{emp}\}$ is provable. Let us take $\Gamma = a : \mathsf{true}$ and $\mathfrak{C}$ be the following contract:



The following proof sketch can be formally derived using our proof system (recall that $\mathsf{one} \triangleq \mathsf{emp}_s \wedge 10 \mapsto -$):

```
$\{(\mathsf{e} \Vdash \mathsf{e} \mapsto (-, \mathfrak{C}, 1, q_0)) * (\mathsf{one} \vee \mathsf{emp})\}$
receive(a,e);
$\{(\mathsf{e} \Vdash \mathsf{e} \mapsto (-, \mathfrak{C}, 1, q_1)) * (\mathsf{one} \vee \mathsf{emp}) * \mathsf{true}\}$
$\{(\mathsf{e} \Vdash \mathsf{e} \mapsto (-, \mathfrak{C}, 1, q_1)) * \mathsf{true}\}$
send(a,e);
$\{\mathsf{e} \Vdash \mathsf{e} \mapsto (-, \mathfrak{C}, 1, q_0)\}$
```

This can obviously be used in lieu of the previous CCR example to produce a proof of

$$\{(\mathsf{e} \Vdash \mathsf{e} \mapsto (-, \mathfrak{C}, 1, q_0)) * (\mathsf{one} \vee \mathsf{emp})\}\ \texttt{receive(a,e); send(a,e);}\ \{\mathsf{false}\}$$

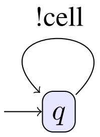
This is inconsistent, because the program above does not diverge if a is present in e's queue at the beginning of the execution. Thus, we require message footprints to be *precise*, which forbids this counter-example. As for concurrent separation logic, we used the conjunction rule to derive unsoundness; one might be able to have soundness with imprecise footprints if one gives up the conjunction rule.

The precision assumption comes into play in the proof of the locality of the command send (see Lemma 6.5 page 114).

5.3.3 Channel closure leaks

As it stands, the rule of close can hide a leak if messages were still present in the queues when the channel was closed. Consider for instance the following contract and proof sketch for a program that sends a cell over a channel and closes it (weakening steps are omitted):

$\mathfrak{C} \triangleq$



```
$\{\mathsf{e},\mathsf{f},\mathsf{x} \Vdash (\mathsf{e} = X \wedge \mathsf{f} = Y) \wedge (X \mapsto (Y, \mathfrak{C}, 1, q) * Y \mapsto (X, \mathfrak{C}, 2, q) * \mathsf{x} \mapsto -)\}$
send(cell,e,x);
x = 0;
$\{\mathsf{e},\mathsf{f},\mathsf{x} \Vdash (\mathsf{e} = X \wedge \mathsf{f} = Y) \wedge (X \mapsto (Y, \mathfrak{C}, 1, q) * Y \mapsto (X, \mathfrak{C}, 2, q))\}$
close(e,f);
$\{\mathsf{e},\mathsf{f},\mathsf{x} \Vdash \mathsf{emp}_h\}$
```

At the end of the execution of this program, x is still allocated, whereas the postcondition of the Hoare triple states that the heap is empty. Hence, the proof system alone cannot guarantee the absence of such leaks. We will see in the next chapter that requiring the contracts to be leak-free (Definition 3.19) is enough to prevent them. However, another kind of leaks is still possible, that does not depend on properties of the communicating system but rather on those of the message footprints, as we are just about to see.

5.3.4 Cycles of ownership leaks

Our proof system has a defect: some memory leaks can go unnoticed by the rules when a message footprint holds permission on the recipient endpoint, or more generally when the queues of a set of endpoints can contain a set of messages whose footprints, when put together, hold permission on all the peers of the endpoints of this set (in other words, when the footprints of these messages all contain the endpoints on which they have to be received). This makes it impossible, according to the logic, for a program to access these endpoints and receive theses messages. In other words, the right to receive these messages is held by the messages themselves, leading to an inextricable situation. This problem is very similar in nature to the one encountered by Gotsman *et al.* with their proof system for storable locks [GBC^{+}07].

Let us give some intuitions on the source of the issue. For instance, if $\gamma_{\mathsf{ep}} \triangleq \mathsf{emp}_s \wedge \mathsf{val} \mapsto (-,-,-,-)$, then the following program proof can be derived, given a system $\mathfrak{C}$ that allows the sending of ep:

```
$\{\mathtt{e},\mathtt{f} \Vdash \mathsf{emp}_h\}$
(e,f) = open($\mathfrak{C}$);
$\{\mathtt{e},\mathtt{f} \Vdash \mathtt{e} = X \wedge \mathtt{f} = Y \wedge (X \mapsto (Y,\mathfrak{C},r,q) * Y \mapsto (X,\mathfrak{C},r',q'))\}$
send(ep,e,f);
$\{\mathtt{e},\mathtt{f} \Vdash \mathtt{e} \mapsto (\mathtt{f},\mathfrak{C},r,q'')\}$
```

In this example, there is no ownership-abiding way of closing the channel $(\mathtt{e},\mathtt{f})$, since the ownership of the endpoint pointed to by f has been leaked away. This can be a means of detecting such pathological situations, since most of the times channels will be closed at some point in the program. However, more extreme situations are possible where the whole channel “disappears.” For instance, with a footprint describing a whole channel $\gamma_{\mathsf{chan}} \triangleq \mathsf{emp}_s \wedge \mathsf{val} \mapsto (Y,-,-,-) * Y \mapsto (\mathsf{val},-,-,-)$, the following program proof can be derived, given a system $\mathfrak{C}$ that allows the sending of chan:

```
$\{\mathtt{e},\mathtt{f} \Vdash \mathsf{emp}_h\}$
(e,f) = open($\mathfrak{C}$);
$\{\mathtt{e},\mathtt{f} \Vdash \mathtt{e} = X \wedge \mathtt{f} = Y \wedge (X \mapsto (Y,\mathfrak{C},r,q) * Y \mapsto (X,\mathfrak{C},r',q')\}$
send(chan,e,e);
$\{\mathtt{e},\mathtt{f} \Vdash \mathsf{emp}_h\}$
```

More complicated examples can be crafted if we try and forbid the previous situation by taking $\gamma'_{\mathrm{ep}} \triangleq \mathsf{emp}_s \wedge \mathsf{val} \mapsto (X,-,-,-) \wedge X \neq \mathsf{dst}$, which forbids sending the ownership of the recipient endpoint within the message. For instance, using two channels that lock each other:

```
$\{\mathtt{e},\mathtt{f},\mathtt{ee},\mathtt{ff} \Vdash \mathsf{emp}_h\}$
```

```
(e,f) = open(ℭ);
(ee,ff) = open(ℭ);
```

$$\left\{ \texttt{e},\texttt{f} \Vdash \begin{array}{l} \texttt{e} = X \wedge \texttt{f} = Y \wedge (X \mapsto (Y, \mathfrak{C}, r, q) * Y \mapsto (X, \mathfrak{C}, r', q')) \\ * \texttt{ee} = X' \wedge \texttt{ff} = Y' \wedge (X' \mapsto (Y', \mathfrak{C}, r, q) * Y' \mapsto (X', \mathfrak{C}, r', q')) \end{array} \right\}$$

```
send(ep,e,ff);
send(ep,ee,f);
```

$\{\texttt{e},\texttt{f} \Vdash \texttt{e} \mapsto (\texttt{f}, \mathfrak{C}, r, q'') * \texttt{ee} \mapsto (\texttt{ff}, \mathfrak{C}, r, q'')\}$

In this example, the endpoint pointed to by `f` is in the queue of the one pointed to by `ff` and vice-versa, making it impossible for the program to retrieve any of these two endpoints without cheating the ownership assumption. In the logic, both endpoints have vanished from the assertions, giving the false impression that someone else may own them, which could hide some leaks.

We will show in Section 6.4.4 that one may avoid these pathological cycles of ownership by considering only *admissible footprints*, for which they are impossible. For the formal definition, we follow the terminology Gotsman *et al.*.

Definition 5.1 (Admissible footprint context) *A footprint context* $\Gamma = \mathsf{a}_1 : \gamma_1, \ldots, \mathsf{a}_n : \gamma_n$ *is* admissible *if it is well-formed and if for all interpretations* i*, there do not exist*

- *a non-empty finite set* $A \subseteq \{a_1, \ldots, a_n\} \times \mathsf{Endpoint} \times \mathsf{Val}$*,* $A = \{(\mathsf{b}_i, \varepsilon_i, v_i) \mid i \in I\}$ *for some finite* $I \subseteq \mathbb{N}$*,*
- *endpoints* ε'_i *and roles* r_i *for each* ε_i*,*
- *and a state* $(s, h, \dot{k})$ *such that* $(s, h, \dot{k}) \vDash \circledast_{i \in I} \gamma_{\mathsf{b}_i}(\varepsilon_i, \varepsilon'_i, r_i, v_i)$

such that for all $i \in I$*,* $\varepsilon'_i \in dom(\dot{k})$ *and* $\dot{k}(\varepsilon'_i) = (\varepsilon_i, -, 3 - r_i, -)$*.*

Informally, a footprint context is admissible if it is impossible to send a certain number of messages that form a set A of tuples $(\mathsf{a}, \varepsilon, v)$ (where a is a tag in the domain of Γ, ε the source endpoint and v the value of the message) whose footprints hold permissions on all the buffers wherein they will be stored (the queues of the peers of all the source endpoints of A). Indeed, if the footprints hold such permissions then the program cannot rightfully receive any of the messages of A, hence their contents are leaked from an ownership point of view (and possibly unreachable, for instance if one of the queues also contains a location that is not referenced from anywhere else, which makes them a "real" leak in the sense that the program has no means–right or wrong–to access them).

The footprint contexts $\mathsf{ep} : \gamma_{\mathsf{ep}}$ and $\mathsf{ep}' : \gamma'_{\mathsf{ep}}$ above are not admissible. Let us show this for γ'_{ep}. We take $A = \{(\mathsf{ep}, \varepsilon_1, \varepsilon'_2), (\mathsf{ep}, \varepsilon'_1, \varepsilon_2)\}$. Consider the following state $\dot{\sigma}$:

$$(\varnothing, \varnothing, [\varepsilon_2 : (\varepsilon_1, \mathfrak{C}, 1, q), \varepsilon'_2 : (\varepsilon'_1, \mathfrak{C}, 1, q)])$$

This state satisfies the combined footprints of the two messages:

$$\dot{\sigma} \vDash \gamma'_{\mathsf{ep}}(\varepsilon_1, \varepsilon_2, 0, \varepsilon'_2) * \gamma'_{\mathsf{ep}}(\varepsilon'_1, \varepsilon'_2, 0, \varepsilon_2)$$

As $\dot{\sigma}$ owns the recipient endpoints ε_2 and ε'_2 of the two messages, the footprint of ep is not admissible.

Sufficient syntactic conditions. Fortunately, there are admissible footprint contexts. For instance, every footprint that cannot allow ownership of one or several endpoints is trivially admissible, as is any set of such footprints. However, a context with the footprint of the fin message that we have used in our examples so far is not, as it can be used to send both sides of a channel on themselves:

```
$\{\mathtt{e},\mathtt{f} \Vdash \mathtt{e} = X \wedge \mathtt{f} = Y \wedge (X \mapsto (Y, \mathfrak{C}, r, q) * Y \mapsto (X, \mathfrak{C}, r', q')\}$
send(fin,e,e);
send(fin,f,f);
$\{\mathtt{e},\mathtt{f} \Vdash \mathsf{emp}_h\}$
```

Fortunately, there is an easy fix: one can specify which side of the channel is to be sent within the footprint. For instance, allowing only the first endpoint to be sent can be achieved by taking $\gamma'_{\mathsf{fin}} \triangleq \mathsf{val} \mapsto (\mathsf{dst}, \mathfrak{C}, 1, q)$ (or $\gamma'_{\mathsf{fin}} \triangleq \mathsf{val} \mapsto (-, \mathfrak{C}, 1, q) \wedge \mathsf{val} = \mathsf{src}$).

A broader solution consists in allowing to send endpoints of only a certain role, which must match the role of the endpoint sending them. The following lemma shows that this is indeed sufficient to ensure admissibility. It also implies admissibility of footprints that contain no endpoints.

Lemma 5.3 (Sufficient admissibility condition) *Any footprint context* Γ *such that there is* $r \in \{0, 1\}$ *such that for all* $\gamma \in \Gamma$*,*

$$\vDash (\gamma \wedge (X \mapsto (-,-,-,-) * \mathsf{true})) \Rightarrow (\gamma \wedge (X \mapsto (-,-,r,-) * \mathsf{true} \wedge \mathsf{rl} = r))$$

is admissible.

Proof Suppose that Γ is not admissible and let us reuse the notations of Definition 5.1. There is $\dot{\sigma} = (s, h, \dot{k})$ such that $(s, h, \dot{k}) \vDash \circledast_{i \in I} \gamma_{\mathsf{b}_i}(\varepsilon_i, \varepsilon'_i, r_i, v_i)$ and for all $i \in I$, $\varepsilon'_i \in dom(\dot{k})$ and $\dot{k}(\varepsilon'_i) = (\varepsilon_i, -, 3 - r_i, -)$. Suppose that $I = \{1, \ldots, n\}$. In particular, there are $\dot{\sigma}_1 = (s_1, h_1, \dot{k}_1) \vDash \gamma_{\mathsf{b}_1}(\varepsilon_1, \varepsilon'_1, r_1, v_n)$, …, $\dot{\sigma}_n = (s_n, h_n, \dot{k}_n) \vDash \gamma_{\mathsf{b}_n}(\varepsilon_n, \varepsilon'_n, r_n, v_n)$. We can assume without loss of generality that none of the endpoint heaps $\dot{k}_1$, …, $\dot{k}_n$ is empty: if one of them $\dot{k}_j$ is, then $\dot{\sigma} - \dot{\sigma}_j$ and $\circledast_{i \in I \setminus \{j\}} \gamma_{\mathsf{b}_i}(\varepsilon_i, \varepsilon'_i, r_i, v_i)$ also form a witness that Γ is not admissible.

Let us show that the role of all the sending endpoints of the messages whose footprints form the proof of non-admissibility of Γ are of the same role r. Because all the endpoint heaps $\dot{k}_i$ are non-empty, it is the case that, for all $i \in I$,

$$\dot{\sigma}_i \vDash \gamma_{\mathsf{b}_i}(\varepsilon_i, \varepsilon'_i, r_i, v_i) \wedge (X \mapsto (-,-,-,-) * \mathsf{true}) \ .$$

Hence, by application of the hypothesis of the lemma,

$$\dot{\sigma}_i \vDash \gamma_{\mathsf{b}_i}(\varepsilon_i, \varepsilon'_i, r_i, v_i) \wedge (X \mapsto (-,-,r,-) * \mathsf{true} \wedge r_i = r) \ .$$

Let us now show that supposing that Γ is not admissible is absurd. Consider any of the recipient endpoints, for instance the first one, ε'_1. Because $\varepsilon'_1 \in dom(\dot{k})$, ε'_1 is also in the domain of $\dot{\sigma}_i$ for some $i \in I$, and

$$\dot{\sigma}_i \vDash \gamma_{\mathsf{b}_i}(\varepsilon_i, \varepsilon'_i, r_i, v_i) \wedge \left(\varepsilon'_1 \mapsto (-,-,-,-) * \mathsf{true}\right) \ .$$

By hypothesis of the lemma, it follows that

$$\dot{\sigma}_i \vDash \gamma_{\mathsf{b}_i}(\varepsilon_i, \varepsilon_i', r_i, v_i) \wedge \left(\varepsilon_1' \mapsto (-,-,r_i,-) * \mathsf{true}\right) \ .$$

Thus, $\dot{k}(\varepsilon_1') = \dot{k}_i(\varepsilon_1') = (-,-,r_i,-)$. Since $r_i = r = r_1$, $\dot{k}(\varepsilon_1') = (-,-,r_1,-)$. Since moreover $\dot{k}(\varepsilon_1') = (-,-,3-r_1,-)$, we obtain a contradiction: $r_1 = 3 - r_1$. □

We found that this criterion is sufficient in practise, as we have never needed more sophisticated ones in our case studies. It is easier to check, be it automatically (on certain fragments) or by hand, than the general criterion.

It may be worth noting that the Sing♯ language also imposes an admissibility condition on the endpoints that may be sent over channels: they may only be endpoints that are currently in a sending state of their contract. Although this constraint arises in their setting because of garbage collection considerations, we think that such a condition (translated into a constraint over footprints) may also be sufficient (but not necessary) to ensure admissibility.

5.3.5 A first soundness result

Abstract separation logic There are numerous variants of separation logics. The goal of abstract separation logic, developed by Calcagno, O'Hearn and Yang [COY07], is to isolate the crucial properties that make all these variants valid models of separation logic, that is, models in which the Frame and Parallel rules (see Figures 4.6 and 4.5) are sound. There are fundamentally two ingredients to it: the underlying state model must be a *separation algebra*, and the atomic commands must behave as *local functions* over these states.

The model and proof system presented in this section fit the framework of abstract separation logic if one extends it to handle local variables and external choice. Many other models can be used as a basis for a separation logic: one can do away with the stack and reason only on the heap. or attach *permissions* to variables and/or memory locations to permit read-only sharing between programs.

The main theorem of abstract separation logic is a generic soundness result for all these logics and models. This result can be instantiated to recover Brookes' soundness result (see Section 4.2.3). In fact, the soundness of our original message-passing logic was first proved using abstract separation logic [VLC09]. To achieve this, we had to refine the proof system and the underlying model of abstract separation logic to account for synchronisation and communication issues. Indeed, as we will see below, inferring soundness from the rules presented above would lead to a semantics for the programming language which would be a very coarse approximation of the one presented in Chapter 2. The approach exposed in this thesis is different: we prove the soundness of our logic directly from a refined version of the operational semantics of Chapter 2 (see Chapter 6) instead of the denotational semantics provided by the abstract separation logic framework. Thus, we will not state the generic soundness result of abstract separation logic in this thesis, as it would require more definitions, and instead refer the interested reader to the original publication on abstract separation logic [COY07]. For our exposition, it suffices to know that such a result exists.

A coarse soundness result Since our proof system can benefit from the generic abstract separation logic soundness result, let us give examples of the issues that a semantics derived solely from our proof system would meet, a more detailed account of which can be found in the original paper [VLC09]. Let us give three example programs on which the semantics directly derived from the proof rules via abstract separation logic would be too coarse, unlike the semantics we will develop in the next section.

1. Receives may happen before the corresponding sending operation has happened. Indeed, nothing in the proof rules prevents it, nor is it prevented by the interleaving of traces, and these are the only things taken into account to build the semantics of programs in abstract separation logic. For instance, given a freshly opened channel $(\mathtt{e}, \mathtt{f})$ over which one may communicate the message a, the following program would not block:

   ```
   receive(a,f);
   send(a,e);
   ```

   Indeed, the rule for receive says nothing about the contents of the queues of `f` (in fact, queues are not even present in our model), so a semantics based on it may decide to receive whenever the contract allows it.

2. What is sent does not necessarily match what is received. In fact, following the rules Send and ChannelDispatch, the semantics of `send` corresponds to disposing of the message's contents, and the one of `receive` to a non-deterministic allocation of a portion of storage satisfying the message's footprint. In particular, the program

   ```
   (e,f) = open(ℭ);
   send(a,e,10);
   x = receive(a,f);
   close(e,f);
   ```

   assigns 10 to `x` only if we choose ϕ_m such that it implies $\mathsf{val} = 10$, whereas it should always be equivalent to

   ```
   x = 10;
   ```

   This highlights another issue of this semantics: it depends on the footprint context, which is supposed to be only an artifact of the proof.

3. The semantics would hide both forms of leaks described earlier, since the logic is not aware of them: as far as the rules are concerned, the contents of messages disappear when sent, and closing a channel is only about the endpoint heap, not the corresponding buffers.

The next chapter is devoted to defining a satisfactory semantics and to establish the soundness of our proof system with respects to this semantics.

Related work

To the best of this author's knowledge, this is the first extension of separation logic to message passing that supports copyless message passing. Not long after our own, another extension to copyless message passing has been published by Bell *et al.* [BAW10]. Their logic bears remarkable similarities to ours, except on two points regarding communications: firstly, they support multiple producers and consumers communicating through a single channel, that is they allow endpoints to be shared between processes; secondly, they do not consider safety of communications as an issue, and thus do not consider contracts (hence they model the contents of channel buffers explicitly in the logic). In our setting, sharing endpoints, for instance by assigning permissions to them, would prove troublesome in the logic: if a process shares an endpoint with its environment, then the control state associated to this endpoint may change at any time as a result of sends or receives performed by the environment on this endpoint. Hence, the logic may no longer keep track of the current state of this endpoint reliably. The proof of soundness of Bell *et al.* is closely related to our first published proof of soundness [VLC09], where we overcome the difficulties that come with local reasoning in abstract separation logic by assigning *histories* to endpoints, as they also do.

Other extensions of concurrent separation logic have been created in the past. We have already mentioned how Gotsman *et al.* have extended it to support storable locks, and how their approach and ours were faced with similar problems in some respects. They are however concerned with objects different in nature. Vafeiadis, on the other hand, has extended separation logic to support *rely/guarantee* reasoning. However, his work is about communications via shared variables, and the connection to the message-passing case is unclear, short of implementing buffers and contracts using lower-level primitives.

Chapter 6

Open Behaviours and Soundness

In this chapter, we define an *open semantics* for programs, with regards to which we prove the soundness of the logic of Chapter 5. This open semantics is a triple point between the proof rules, the semantics of Chapter 2, and the contracts of Chapter 3. Indeed, it gives an operational meaning to the ownership transfers suggested by the proof rules, while being connected both to the natural semantics of programs of Chapter 2 (henceforth the *closed semantics*) and to the semantics of the contracts of the program. This allows us to use the open semantics as an intermediate step in the proof of the soundness of our logic with respect to the closed semantics: we first prove the soundness with respect to the open semantics, and then connect the open semantics to the closed one. Similarly, we show that proved programs inherit the absence of unexpected receptions and undelivered messages from their contracts.

The open semantics may be seen as an instrumentation of the closed semantics that makes apparent the ownership transfers that occur when communications are performed on the one hand, and the obedience to the channel contracts on the other hand. More precisely, the open semantics of sending and receiving commands will be changed so that sending a message removes its footprint from the local view, and receiving it adds the corresponding footprint. So that what is received matches exactly what has been sent, the transient footprint is stored in the receiving endpoint's buffer along with the message identifier and value. Contrarily to the closed one, the open semantics depends on a footprint context and a contract decoration of the program.

The "open" character of this new semantics is also reflected in that it makes interferences from the environment *explicit*. Indeed, interferences will be able to simulate the sending and receiving of messages on those endpoints not in the program's possession, in order to account for the actions of the other programs that may be concurrently executing aside from the current one (its *environment*). This is necessary to study the parallel composition of two programs in a sound way: as each program is analysed independently from the other (as stated by the Parallel rule of Figure 4.5 page 71), one should be able to characterise the behaviour of one of them whatever the other one (its environment) does, hence to account for the behaviour of its environment.

The main results of this chapter are the soundness of the logic with respect to the open semantics (Theorem 6.1), the connection between the validity for the open semantics with fault-free contracts and the runtime safety of the closed semantics (Theorem 6.2), and the leak freedom of programs provable under an admissible footprint context and leak-free contracts (Theorem 6.3). Technical lemmas are kept in subsections 6.2.2, 6.3.1, 6.4.1 and 6.4.3.

Many proofs of other variants of separation logic exist. The original proof for concurrent separation logic is due to Brookes [Bro07] and is also based on an intermediate "open" semantics, although a denotational model based on complete traces is used instead of an operational one as in this thesis. This result was later made agnostic in the programming language and the underlying state model by the framework of abstract separation logic [COY07], as discussed in the previous chapter. Proofs for other extensions of separation logic include several ones for RGSep [VP07, Vaf07] and for CSL with storable locks and threads, either based on operational models [GBC+07], or on an oracle semantics [HAN08]. Some of these proofs might be applicable to our logic and prove its consistency by somehow encoding message passing into their framework. Yet, none of them include message passing or contracts as primitives, and thus the links to the semantics of copyless message passing and the semantics of contracts would be lost.

6.1 Open states

6.1.1 Well-formed open states

Definition From now on, the states defined in Section 2.2.1 are called *closed states* to avoid ambiguities. The states defined in this chapter are called *open states*; they account for the *local view* that the program has of the global state, as well as the buffers of the endpoints of all channels, with footprints (represented by session states of the previous chapter) added to each pending message. Open states extend both closed states and session states. They are elements $\underline{\sigma} = ((s, h, \dot{k}), \underline{k})$ of $\underline{\mathsf{State}}$, with

$$\underline{\mathsf{State}} \triangleq \dot{\mathsf{State}} \times \underline{\mathsf{EHeap}} \qquad\qquad \underline{\mathsf{EHeap}} \triangleq \mathsf{Endpoint} \rightharpoonup_{\mathit{fin}} (\mathsf{MsgId} \times \mathsf{Val} \times \dot{\mathsf{State}})^*$$

An open state $\underline{\sigma} = (\dot{\sigma}, \underline{k}) \in \underline{\mathsf{State}}$ can be decomposed as such, if one thinks of it as the local state of a program:

- $\dot{\sigma}$ is called the *local* part of the state, and is a session state that represents what is currently owned by the program, that is what it may safely access and act upon;

- $\underline{k}$ is an *open* endpoint heap that associates open buffers to endpoints. An open buffer is one where pieces of local state are attached to each pending message in the buffers. The domain of $\underline{k}$ must be sufficiently large to associate a buffer to all the endpoints that are allocated according to $\underline{\sigma}$ and to their peers (in other words, to each channel). These endpoints can come from two sources: either they belong to the local state $\dot{\sigma}$, or they belong to one of the footprints stored in $\underline{k}$ itself. This gives rise to a rather complicated well-formedness condition which is the subject of the rest of this section.

Flattening of a state As will become clear with the open operational semantics, the footprints stored in the open buffers correspond to actual pieces of the global state that have been previously sent away. Thus, one should be able to reconstruct the global state by tacking all these footprints onto the local state, an operation we call *flattening*. Flattening is useful both to obtain a global picture of the current state that includes all the information stored in the pieces of state hidden in the buffers (for instance, to know who the peer of an endpoint is, or what contract is associated to it) and to establish a correspondence between the open and the closed semantics (as we do in Section 6.4). For the flattening of an open state to be defined, all the corresponding session states should be pairwise disjoint and compatible as session states. When this is the case, the open state is deemed *flattable*.

Definition 6.1 (Flattable state) *An open state* $\underline{\sigma} = (\dot{\sigma}, \underline{k})$ *is* flattable *if* $\dot{\sigma}$ *and all the states* $\dot{\sigma}_a$ *for all* $(a, v, \dot{\sigma}_a) \in \underline{k}(\varepsilon)$ *and* $\varepsilon \in dom(\underline{k})$ *are well-formed and pairwise compatible as session states.*

Flattable buffers and open endpoint heaps are defined similarly. The following flattening function is defined on flattable open buffers and associates to an open buffer $\underline{\alpha}$ the session state formed by the concatenation of the footprints in $\underline{\alpha}$ (where $\bigodot$ is the iterated version of the composition $\bullet$ of session states defined in the previous chapter):

$$\mathit{flat}(\underline{\alpha}) \triangleq \bigodot_{(a,v,\dot{\sigma}_a)\in\underline{\alpha}} \dot{\sigma}_a$$

This represents the total amount of state that is stored inside a buffer. We overload this function to concatenate the flattenings of all the buffers of a flattable open endpoint heap $\underline{k}$:

$$\mathit{flat}(\underline{k}) \triangleq \bigodot_{\varepsilon\in dom(\underline{k})} \mathit{flat}(\underline{k}(\varepsilon))$$

Finally, let us define the flattening of a flattable open state (one whose open endpoint heap is flattable and compatible with its local state). Flattening applies to an open state by tacking the flattenings of its buffers onto its local state:

Definition 6.2 (Flattening of an open state) *The flattening of a flattable open state* $\underline{\sigma} = (\dot{\sigma}, \underline{k})$ *is defined as*

$$\mathit{flat}((\dot{\sigma}, \underline{k})) \triangleq \dot{\sigma} \bullet \mathit{flat}(\underline{k})\ .$$

We can now define the set $\underline{\mathsf{State}}^{\mathsf{wf}}$ of well-formed open states.

Definition 6.3 ($\underline{\mathsf{State}}^{\mathsf{wf}}$**)** *An open state* $\underline{\sigma} = (\dot{\sigma}, \underline{k})$ *is* well-formed *if it is flattable,* $\mathit{flat}(\underline{\sigma}) = (s_f, h_f, \dot{k}_f)$ *and the following conditions are satisfied:*

$$\begin{aligned} dom(\dot{k}_f) &\subseteq dom(\underline{k}) && \text{(OBuffers)} \\ \forall \varepsilon \in dom(\dot{k}_f).\ mate(\dot{k}_f, \varepsilon) &\in dom(\underline{k}) && \text{(OChannel)} \end{aligned}$$

The conditions (OBuffers) and (OChannel) ensure that every endpoint allocated in any of the session states that appear in the open state $\underline{\sigma}$ is associated with a buffer in $\underline{k}$, and so is its peer. The other well-formedness conditions imposed on closed and session states are already ensured by the fact that $\mathit{flat}(\underline{\sigma})$ is flattable (see Lemma 6.8 for more details).

Open states composition The composition of two open states $\underline{\sigma}_1 = (\dot{\sigma}_1, \underline{k}_1)$ and $\underline{\sigma}_2 = (\dot{\sigma}_2, \underline{k}_2)$ is defined provided that

- their local parts are disjoint and compatible as session states;
- $\underline{k}_1$ and $\underline{k}_2$ agree on the buffers of the endpoints in the intersection of their domains;
- the resulting open state, as defined below, is well-formed.

When this is the case, we write $\underline{\sigma}_1 \sharp \underline{\sigma}_2$ and the composition is defined as

$$(\dot{\sigma}_1, \underline{k}_1) \bullet (\dot{\sigma}_2, \underline{k}_2) \triangleq (\dot{\sigma}_1 \bullet \dot{\sigma}_2, \underline{k}_1 \cup \underline{k}_2)$$

where the union of two partial functions f and g with the same codomain S is defined, provided that they agree on the intersection of their respective domains of definition, as

$$f \cup g : \begin{array}{rcl} dom(f) \cup dom(g) & \rightharpoonup & S \\ x & \mapsto & \begin{cases} f(x) & \text{if } x \in dom(f) \\ g(x) & \text{if } x \in dom(g) \end{cases} \end{array}$$

6.2 Open operational semantics

From now on, let us consider as fixed a well-formed footprint context Γ.

6.2.1 Open semantics of programs

Notations As for the closed semantics, we define shorthands to describe updates to the current open state, and more precisely to its local part. If $\dot{k}(\varepsilon) = (\varepsilon', \mathfrak{C}, r, q)$, we write $control(\dot{k}, \varepsilon)$ for q, $role(\dot{k}, \varepsilon)$ for r, and $[\dot{k} \mid control(\varepsilon) \leftarrow q']$ for $[\dot{k} \mid \varepsilon : (\varepsilon', \mathfrak{C}, r, q')]$.

Given two session states $\dot{\sigma}_1$ and $\dot{\sigma}_2$, recall that $\dot{\sigma}_1$ is a substate of $\dot{\sigma}_2$, written $\dot{\sigma}_1 \preceq \dot{\sigma}_2$, if there is $\dot{\sigma}$ such that $\dot{\sigma}_1 \bullet \dot{\sigma} = \dot{\sigma}_2$. In this case, $\dot{\sigma}$ is unique, thus we can write $\dot{\sigma}_2 - \dot{\sigma}_1$ for the only such $\dot{\sigma}$.

Let us give the reduction rules of the open semantics. The rules implicitly depend on Γ and are of the form $p, \underline{\sigma} \to p', \underline{\sigma}'$ or $p, \underline{\sigma} \to \mathbf{error}$ with

$$\mathbf{error} \in \{\mathbf{LeakError}, \mathbf{MsgError}, \mathbf{OwnError}, \mathbf{ProtoError}\} .$$

The errors $\mathbf{MsgError}$ and $\mathbf{OwnError}$ have the same intuitive meanings as in Section 2.2.2: they represent respectively reception and ownership runtime errors. A program raises $\mathbf{LeakError}$ if it closes a channel with undelivered messages (thus creating a potential leak), and $\mathbf{ProtoError}$ if it does not act according to one of its contracts, either by performing an unauthorised action on a channel, or because of a `switch receive` that is not exhaustive with respect to one of the contracts involved.

Stack and heap commands For the stack and heap commands, the open semantics is derived straightforwardly from the closed one, taking the local portion of the state as the main stack and heap. The rules are shown in Figures 6.1 and 6.2. The semantics of `new` is the only one that requires special attention, namely to avoid reusing a location that may be hidden in the contents of a queue (which could result in an ill-formed state).

$$\frac{}{\mathtt{skip}, \underline{\sigma} \to \mathtt{skip}, \underline{\sigma}} \qquad \frac{\boxed{[\![B]\!]_s} = \mathsf{true}}{\mathtt{assume}(B), ((s,h,\dot{k}),\underline{k}) \to \mathtt{skip}, ((s,h,\dot{k}),\underline{k})}$$

$$\frac{\boxed{\mathtt{x} \in \mathit{dom}(s)} \qquad \boxed{[\![E]\!]_s} = v}{\mathtt{x}\ \mathtt{=}\ E, ((s,h,\dot{k}),\underline{k}) \to \mathtt{skip}, (([s \mid \mathtt{x} : v], h, \dot{k}),\underline{k})}$$

Figure 6.1: Operational semantics of stack commands.

$$\frac{\begin{array}{c} \mathit{flat}(((s,h,\dot{k}),\underline{k})) = (s_f, h_f, \dot{k}_f) \\ \boxed{\mathtt{x} \in \mathit{dom}(s)} \qquad l \in \mathsf{Cell} \setminus \mathit{dom}(h_f) \qquad v \in \mathsf{Val} \end{array}}{\mathtt{x}\ \mathtt{=}\ \mathtt{new}(), ((s,h,\dot{k}),\underline{k}) \to \mathtt{skip}, (([s \mid \mathtt{x} : l], [h \mid l : v], \dot{k}),\underline{k})}$$

$$\frac{\boxed{[\![E]\!]_s} = l \qquad \boxed{l \in \mathit{dom}(h)}}{\mathtt{dispose}(E), ((s,h,\dot{k}),\underline{k}) \to \mathtt{skip}, ((s, h \setminus l, \dot{k}),\underline{k})}$$

$$\frac{\boxed{\mathtt{x} \in \mathit{dom}(s)} \qquad \boxed{[\![E]\!]_s} = l \qquad \boxed{h(l)} = v}{\mathtt{x}\ \mathtt{=}\ \mathtt{[}E\mathtt{]}, ((s,h,\dot{k}),\underline{k}) \to \mathtt{skip}, (([s \mid \mathtt{x} : v], h, \dot{k}),\underline{k})}$$

$$\frac{\boxed{[\![E_1]\!]_s} = l \qquad \boxed{[\![E_2]\!]_s} = v \qquad \boxed{l \in \mathit{dom}(h)}}{\mathtt{[}E_1\mathtt{]}\ \mathtt{=}\ E_2, ((s,h,\dot{k}),\underline{k}) \to \mathtt{skip}, ((s, [h \mid l : v], \dot{k}),\underline{k})}$$

Figure 6.2: Operational semantics of heap commands.

Opening and closing a channel The semantics of `open` and `close`, shown in Figure 6.3, now takes the protocol of the channel into account: `open` creates a channel in the initial state of the contract and `close` deletes it if both endpoints are in the same final state of their contract and their buffers are empty. If this last condition is not satisfied, `close` raises **LeakError**. If the channel is closed in a non-final state of the contract, it raises a protocol error, and if the endpoints given as arguments do not form a channel, or are given in the wrong order (this last condition could easily be dropped), an ownership error is raised. Like `new`, `open` takes care not to reuse a location already present in one of the buffers, so as not to create an ill-formed state. Because of the (OBuffers) constraint on well-formed states, picking the new endpoints outside of the domain of $\underline{k}$ is enough.

$$\dfrac{\boxed{\mathtt{e},\mathtt{f} \in \mathit{dom}(s)} \qquad \varepsilon,\varepsilon' \in \mathsf{Endpoint} \setminus \mathit{dom}(\underline{k}) \qquad q_0 = \mathit{init}(\mathfrak{C})}{\begin{array}{l}(\mathtt{e},\mathtt{f})\ \mathtt{=}\ \mathtt{open}(\mathfrak{C}),((s,h,\dot{k}),\underline{k}) \to \\ \mathtt{skip},\left(\begin{array}{c}([s \mid \mathtt{e}:\varepsilon,\mathtt{f}:\varepsilon'],h,[\dot{k} \mid \varepsilon:(\varepsilon',\mathfrak{C},1,q_0),\varepsilon':(\varepsilon,\mathfrak{C},2,q_0)]),\\ [\underline{k} \mid \varepsilon:\sqcup,\varepsilon':\sqcup]\end{array}\right)\end{array}}$$

$$\dfrac{\begin{array}{c}\boxed{\llbracket E_1 \rrbracket_s} = \varepsilon_1 \qquad \boxed{\llbracket E_2 \rrbracket_s} = \varepsilon_2 \qquad \boxed{\dot{k}(\varepsilon_1)} = (\varepsilon_2,\mathfrak{C},1,q_f)\\ \boxed{\dot{k}(\varepsilon_2)} = (\varepsilon_1,\mathfrak{C},2,q_f) \qquad q_f \in \mathit{finals}(\mathfrak{C}) \qquad \underline{k}(\varepsilon_1)\cdot\underline{k}(\varepsilon_2) = \sqcup\end{array}}{\mathtt{close}(E_1\mathtt{,}E_2),((s,h,\dot{k}),\underline{k}) \to \mathtt{skip},((s,h,\dot{k} \setminus \{\varepsilon_1,\varepsilon_2\}),\underline{k} \setminus \{\varepsilon_1,\varepsilon_2\})}$$

$$\dfrac{\begin{array}{c}\llbracket E_1 \rrbracket_s = \varepsilon_1 \qquad \llbracket E_2 \rrbracket_s = \varepsilon_2 \qquad \dot{k}(\varepsilon_1) = (\varepsilon_2,\mathfrak{C},1,q_f)\\ \dot{k}(\varepsilon_2) = (\varepsilon_1,\mathfrak{C},2,q_f) \qquad q_f \in \mathit{finals}(\mathfrak{C}) \qquad \underline{k}(\varepsilon_1)\cdot\underline{k}(\varepsilon_2) \neq \sqcup\end{array}}{\mathtt{close}(E_1\mathtt{,}E_2),((s,h,\dot{k}),\underline{k}) \to \mathbf{LeakError}}$$

$$\dfrac{\begin{array}{c}\llbracket E_1 \rrbracket_s = \varepsilon_1 \qquad \llbracket E_2 \rrbracket_s = \varepsilon_2\\ \dot{k}(\varepsilon_1) = (\varepsilon_2,\mathfrak{C},1,q_1) \qquad \dot{k}(\varepsilon_2) = (\varepsilon_1,\mathfrak{C},2,q_2) \qquad q_1 \neq q_2 \text{ or } q_1 \notin \mathit{finals}(\mathfrak{C})\end{array}}{\mathtt{close}(E_1\mathtt{,}E_2),((s,h,\dot{k}),\underline{k}) \to \mathbf{ProtoError}}$$

$$\dfrac{\llbracket E_1 \rrbracket_s = \varepsilon_1 \qquad \llbracket E_2 \rrbracket_s = \varepsilon_2 \qquad \dot{k}(\varepsilon_1) = (\varepsilon_2',\mathfrak{C},r,q) \qquad \varepsilon_2' \neq \varepsilon_2 \text{ or } r \neq 1}{\mathtt{close}(E_1\mathtt{,}E_2),((s,h,\dot{k}),\underline{k}) \to \mathbf{OwnError}}$$

Figure 6.3: Open operational semantics of channel creation and destruction.

Sending a message The semantics of `send` is shown in Figure 6.4 and is split into three cases. Let us begin with the successful one: the endpoint and the value are accessible, the send is authorised by the contract, and enough state is held to find a substate that satisfies the message's footprint. Note that, as discussed in the description of the Send rule in the previous chapter, the control state of the sending endpoint is updated before the message is sent, so the footprint is checked against the updated state. When the footprint cannot be found in this state, an ownership error is issued. Similarly, if the send is not permitted by the current state of the contract, a protocol error is raised.

External choice and receiving a message The reduction rules that correspond to receiving a message via a guarded external choice are shown in Figure 6.5. There are four cases: it can either succeed and proceed with one of its branches, or fail, either because the receive is prohibited by the protocol, or because an unexpected message is present at the head of one of the inspected queues, or because, although no unexpected message is necessarily present, the contract stipulates that a message that is not expected by the program could be available.

$$\frac{\boxed{[\![E_1]\!]_s} = \varepsilon \qquad \boxed{[\![E_2]\!]_s} = v \qquad \boxed{\dot{k}(\varepsilon)} = (\varepsilon', \mathfrak{C}, r, q) \\ \mathit{succ}(\mathfrak{C}, r, q, !a) = q' \qquad \dot{\sigma}_a \preceq (s, h, [\dot{k} \mid \mathit{control}(\varepsilon) \leftarrow q']) \qquad \dot{\sigma}_a \vDash \gamma_a(\varepsilon, \varepsilon', r, v)}{\begin{array}{l}\texttt{send}(a, E_1, E_2), ((s, h, \dot{k}), \underline{k}) \to \\ \texttt{skip}, ((s, h, [\dot{k} \mid \mathit{control}(\varepsilon) \leftarrow q']) - \dot{\sigma}_a, [\underline{k} \mid \varepsilon' : \underline{k}(\varepsilon') \cdot (a, v, \dot{\sigma}_a)])\end{array}}$$

$$\frac{[\![E_1]\!]_s = \varepsilon \qquad [\![E_2]\!]_s = v \qquad \dot{k}(\varepsilon) = (\varepsilon', \mathfrak{C}, r, q) \qquad \mathit{succ}(\mathfrak{C}, r, q, !a) = q' \\ \neg\exists \dot{\sigma}_a .\, \dot{\sigma}_a \preceq (s, h, [\dot{k} \mid \mathit{control}(\varepsilon) \leftarrow q']) \;\&\; \dot{\sigma}_a \vDash \gamma_a(\varepsilon, \varepsilon', r, v)}{\texttt{send}(a, E_1, E_2), ((s, h, \dot{k}), \underline{k}) \to \mathbf{OwnError}}$$

$$\frac{[\![E_1]\!]_s = \varepsilon \qquad \dot{k}(\varepsilon) = (\varepsilon', \mathfrak{C}, r, q) \qquad \neg\exists q' .\, \mathit{succ}(\mathfrak{C}, r, q, !a) = q'}{\texttt{send}(a, E_1, E_2), ((s, h, \dot{k}), \underline{k}) \to \mathbf{ProtoError}}$$

Figure 6.4: Open operational semantics of send.

In the successful case, a message identifier is present at the head of an endpoint's buffer and the corresponding branch is selected. Conversely to the case of `send`, the footprint corresponding to the message is seized and added to the current local portion of state. This operation is possible because we always assume the states we consider to be well-formed (a property we will see is preserved by the reduction rules in Lemma 6.2), hence flattable, so $\dot{\sigma}_a$ is guaranteed to be compatible with $(s, h, \dot{k})$. The message is then removed from the queue, the value stored into the appropriate variable $\mathtt{x}_i$ and the control state of the endpoint is updated.

If no such successor exists, $\mathbf{ProtoError}$ is raised. If an unexpected reception occurs $\mathbf{MsgError}$ is raised. Finally, and similarly to the logical treatment of guarded external choice, not taking into account all the possible messages that can be received according to its contract is a fault, and the program reduces to $\mathbf{ProtoError}$.

Programming constructs The semantics of the remaining programming constructs is given in Figure 6.6. The rules are identical to the closed semantics ones, with two exceptions. First, the detection of races uses a different, finer-tuned (as will see in Lemma 6.10) function, based on the local part of the state, which is closer to what is required by the Parallel rule of the proof system: the predicate $\underline{\mathit{norace}}(p_1, p_2, \underline{\sigma})$ is false if it is impossible to partition $\underline{\sigma}$ into two substates with disjoint local parts on which each program can safely make a step. Formally, $\underline{\mathit{norace}}(p_1, p_2, \underline{\sigma})$ holds if and only if there exist well-formed open states $\underline{\sigma}_1$ and $\underline{\sigma}_2$ such that $\underline{\sigma}_1 \bullet \underline{\sigma}_2 = \underline{\sigma}$ and

$$\begin{array}{rl} & p_1, \underline{\sigma}_1 \not\to \mathbf{OwnError} \\ \& & p_2, \underline{\sigma}_2 \not\to \mathbf{OwnError}\,. \end{array}$$

The second difference is in the treatment of the local variable construct. As in the case

$$\frac{\begin{array}{c}\boxed{\mathtt{x}_1,\ldots,\mathtt{x}_n \in dom(s)}\\ \boxed{\llbracket E_1\rrbracket_s \in dom(\dot{k})}\ \&\ \ldots\ \&\ \boxed{\llbracket E_n\rrbracket_s \in dom(\dot{k})} \qquad \llbracket E_i\rrbracket_s = \varepsilon_i \\ \dot{k}(\varepsilon_i) = (\varepsilon', \mathfrak{C}, r, q) \qquad \underline{k}(\varepsilon_i) = (a_i, v, \dot{\sigma}_i)\cdot\underline{\alpha} \qquad succ(\mathfrak{C}, r, q, !a_i) = q'\end{array}}{\begin{array}{l}\mathop{\Box}\limits_{j=1}^{n} \mathtt{x}_j\ \mathtt{=}\ \mathtt{receive}(a_j, E_j) : p_j, ((s,h,\dot{k}),\underline{k}) \to \\ p_i, (([s \mid \mathtt{x}_i : v], h, [\dot{k} \mid control(\varepsilon_i) \leftarrow q']) \bullet \dot{\sigma}_i, [\underline{k} \mid \varepsilon_i : \underline{\alpha}])\end{array}}$$

$$\frac{\llbracket E_i\rrbracket_s = \varepsilon_i \qquad \dot{k}(\varepsilon_i) = (\varepsilon', \mathfrak{C}, r, q) \qquad \neg\exists q'.\, succ(\mathfrak{C}, r, q, ?a_i) = q'}{\mathop{\Box}\limits_{j=1}^{n} \mathtt{x}_j\ \mathtt{=}\ \mathtt{receive}(a_j, E_j) : p_j, ((s,h,\dot{k}),\underline{k}) \to \mathbf{ProtoError}}$$

$$\frac{\llbracket E_1\rrbracket_s = \varepsilon_1\ \&\ \ldots\ \&\ \llbracket E_n\rrbracket_s = \varepsilon_n \qquad \underline{k}(\varepsilon_i) = (a, v, \dot{\sigma}_a)\cdot\underline{\alpha} \qquad \forall j.\, \varepsilon_j = \varepsilon_i \Rightarrow a_j \neq a}{\mathop{\Box}\limits_{j=1}^{n} \mathtt{x}_j\ \mathtt{=}\ \mathtt{receive}(a_j, E_j) : p_j, ((s,h,\dot{k}),\underline{k}) \to \mathbf{MsgError}}$$

$$\frac{\begin{array}{c}\llbracket E_1\rrbracket_s = \varepsilon_1\ \&\ \ldots\ \&\ \llbracket E_n\rrbracket_s = \varepsilon_n\\ \dot{k}(\varepsilon_i) = (\varepsilon', \mathfrak{C}, r, q) \qquad \exists a.\, \exists q'.\, succ(\mathfrak{C}, r, q, ?a) = q'\ \&\ \forall j.\, \varepsilon_j = \varepsilon_i \Rightarrow a_j \neq a\end{array}}{\mathop{\Box}\limits_{j=1}^{n} \mathtt{x}_j\ \mathtt{=}\ \mathtt{receive}(a_j, E_j) : p_j, ((s,h,\dot{k}),\underline{k}) \to \mathbf{ProtoError}}$$

Figure 6.5: Open operational semantics of external choice and receive.

of `new`, we now have to take into account not only the domain of the surface stack, but also all the variables that are allocated in the footprints of the messages of all buffers in order to avoid creating ill-formed states.

6.2.2 Subject reduction

Let us describe two properties of open states that are preserved by program reductions: well-formedness with respect to Γ, which supersedes well-formedness, and being ceremonious, that is having buffers consistent with the contracts and current control states of the endpoints.

Well-formedness with respect to a footprint context Given a footprint context Γ, it is natural to define the notion of well-formedness *w.r.t.* Γ: for each element $(a, v, \dot{\sigma}_a)$ of the buffer of an endpoint ε whose peer is ε' and has role r', it should be the case that $\dot{\sigma}_a \vDash \gamma_a(\varepsilon', \varepsilon, r', v)$. The set of all such states is denoted by $\mathsf{State}^{\mathsf{wf}}_{\Gamma}$. This will crucially help us in proving the soundness of the ChannelDispatch rule: receiving a message a

$$\frac{}{p_1 + p_2, \underline{\sigma} \to p_1, \underline{\sigma}} \qquad \frac{}{p_1 + p_2, \underline{\sigma} \to p_2, \underline{\sigma}}$$

$$\frac{p_1, \underline{\sigma} \to p_1', \underline{\sigma}'}{p_1;\ p_2, \underline{\sigma} \to p_1';\ p_2, \underline{\sigma}'} \qquad \frac{}{\texttt{skip};\ p_2, \underline{\sigma} \to p_2, \underline{\sigma}} \qquad \frac{p_1, \underline{\sigma} \to \mathbf{error}}{p_1;\ p_2, \underline{\sigma} \to \mathbf{error}}$$

$$\frac{\boxed{\underline{norace}(p_1, p_2, \underline{\sigma})} \qquad p_1, \underline{\sigma} \to p_1', \underline{\sigma}'}{p_1 \parallel p_2, \underline{\sigma} \to p_1' \parallel p_2, \underline{\sigma}'} \qquad \frac{p_1, \underline{\sigma} \to \mathbf{error}}{p_1 \parallel p_2, \underline{\sigma} \to \mathbf{error}}$$

$$\frac{\boxed{\underline{norace}(p_1, p_2, \underline{\sigma})} \qquad p_2, \underline{\sigma} \to p_2', \underline{\sigma}'}{p_1 \parallel p_2, \underline{\sigma} \to p_1 \parallel p_2', \underline{\sigma}'} \qquad \frac{p_2, \underline{\sigma} \to \mathbf{error}}{p_1 \parallel p_2, \underline{\sigma} \to \mathbf{error}}$$

$$\frac{}{p^*, \underline{\sigma} \to \texttt{skip} + (p;\ p^*), \underline{\sigma}} \qquad \frac{\mathtt{y} \in dom(s)}{\texttt{delete(y)}, ((s, h, \dot{k}), \underline{k}) \to \texttt{skip}, ((s \setminus \{\mathtt{y}\}, h, \dot{k}), \underline{k})}$$

$$\frac{\mathit{flat}(((s, h, \dot{k}), \underline{k})) = (s_f, h_f, \dot{k}_f) \qquad v \in \mathsf{Val} \qquad \mathtt{y} \notin dom(s_f) \cup \mathit{freevar}(p)}{\texttt{local x in } p, ((s, h, \dot{k}), \underline{k}) \to p[\mathtt{x} \leftarrow \mathtt{y}];\ \texttt{delete(y)}, (([s \mid \mathtt{y} : v], h, \dot{k}), \underline{k})}$$

Figure 6.6: Operational semantics of programming constructs.

from a state that is well-formed with respect to Γ adds a piece of state verifying γ_a to the local state.

Definition 6.4 ($\underline{\mathsf{State}}_\Gamma^{\mathsf{wf}}$) *An open state $\underline{\sigma} = (\dot{\sigma}, \underline{k})$, with $\mathit{flat}(\underline{\sigma}) = (s_f, h_f, \dot{k}_f)$, is well-formed w.r.t. the footprint context Γ if*

- *$\underline{\sigma}$ is a well-formed open state;*
- *for all endpoint $\varepsilon \in dom(\dot{k}_f)$, if $\dot{k}_f(\varepsilon) = (\varepsilon', \mathfrak{C}, r, q)$ then for all $(a, v, \dot{\sigma}_a) \in \underline{k}(\varepsilon)$,*

$$\dot{\sigma}_a \vDash \gamma_a(\varepsilon', \varepsilon, r, v)\ .$$

Note that if $\underline{\sigma}_1$ and $\underline{\sigma}_2$ are both well-formed with respect to Γ, then $\underline{\sigma}_1 \bullet \underline{\sigma}_2$ is also well-formed with respect to Γ if and only if it is well-formed. Hence, we do not need an additional notation for compatibility in that sense.

Let us detail what well-formed open states with respect to a footprint context are in two special cases: the context where no constraint is put on the footprints (notice that in this case they are not precise, hence the footprint context is ill-formed), and the one where all footprints are empty.

- If Γ_{true} is the footprint context where all messages are associated to the footprint true, then $\underline{\sigma}$ is well-formed with respect to Γ_{true} if and only if it is well-formed.

- If Γ_{emp} is the footprint context where all messages are associated to the footprint emp, then $(\dot{\sigma}, \underline{k})$ is well-formed with respect to Γ_{emp} if and only if it is well-formed and $\mathit{flat}(\underline{k}) = \dot{u}$.

Ceremonious states Ceremonious states emerge from the following intuition: given a channel $(\varepsilon, \varepsilon')$ in an open state $\underline{\sigma} = (\dot{\sigma}, \underline{k})$, the contents of the buffers of the two endpoints in terms of message identifiers should be consistent with the control states of the contract that they are in according to $\underline{\sigma}$. This notion is crucial to translate properties of contracts into properties of the program, and to define what the environment is allowed to do (in particular, it will only be able to produce ceremonious states).

We first define the contract configurations associated to a channel in a given open state. Let $\underline{\sigma} = (\dot{\sigma}, \underline{k})$ and $\mathit{flat}(\underline{\sigma}) = (s_f, h_f, \dot{k}_f)$. The set of channels of $\underline{\sigma}$ is the set of pairs $(\varepsilon, \varepsilon')$ such that at least one of ε, ε' belongs to $\mathit{dom}(\dot{k}_f)$:

$$\mathit{chans}(\underline{\sigma}) \triangleq \{(\varepsilon, \varepsilon') \mid \mathit{mate}(\dot{k}_f, \varepsilon) = \varepsilon'\} \ .$$

Note that, because at least one of the two endpoints of a channel of $\underline{\sigma}$ must appear in the domain of $\dot{k}_f$, the channels of $\underline{\sigma}$ are all associated to a contract, the roles of the endpoints are known, and so is the control state of at least one of the two endpoints.

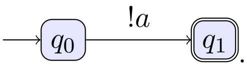
Notations Given an open state $\underline{\sigma}$ such that $\mathit{flat}(\underline{\sigma}) = (s_f, h_f, \dot{k}_f)$, if $\dot{k}(\varepsilon) = (\varepsilon', \mathfrak{C}, r, q)$ is defined then we write $\mathit{contract}(\underline{\sigma}, \varepsilon)$ for $\mathfrak{C}$, $\mathit{role}(\underline{\sigma}, \varepsilon)$ for r and $\mathit{control}(\underline{\sigma}, \varepsilon)$ for q. Given an open buffer $\underline{\alpha} = (a_1, v_1, \dot{\sigma}_1) \cdots (a_n, v_n, \dot{\sigma}_n)$, we write $\mathit{word}(\underline{\alpha})$ for $a_1 \cdots a_n$.

Definition 6.5 (Contract configuration of a channel) *Let $\underline{\sigma}$ be a well-formed open state such that $\mathit{flat}(\underline{\sigma}) = (s_f, h_f, \dot{k}_f)$ and $(\varepsilon, \varepsilon') \in \mathit{chans}(\underline{\sigma})$. The set of contract configurations $\mathit{cconf}(\underline{\sigma}, \varepsilon)$ associated to ε (or to $(\varepsilon, \varepsilon')$) is*

$$\left\{ \langle q, q', \mathit{word}(\underline{k}(\varepsilon)), \mathit{word}(\underline{k}(\varepsilon')) \rangle \,\middle|\, \left\{ \begin{array}{l} \varepsilon \in \mathit{dom}(\dot{k}_f) \Rightarrow q = \mathit{control}(\dot{k}_f, \varepsilon) \\ \varepsilon' \in \mathit{dom}(\dot{k}_f) \Rightarrow q' = \mathit{control}(\dot{k}_f, \varepsilon') \end{array} \right. \right\} \ .$$

A ceremonious state is one in which in every set of contract configuration of every channel there is a configuration reachable by the contract attached to the channel from its initial state. Let us remark that this is not the case in general for arbitrary well-formed open states. Take for instance the following state:

$$((s, h, [\varepsilon : (\varepsilon', \mathfrak{C}, 1, q_0), \varepsilon' : (\varepsilon, \mathfrak{C}, 2, q_0)]), \dot{u}, [\varepsilon : (a, v, \dot{u}), \varepsilon' : \sqcup])$$

where $\mathfrak{C}$ is the contract 

. The contents of the buffer of ε is not coherent with the fact that ε and ε' are both in the initial state of $\mathfrak{C}$. The only acceptable queues for ε and ε' in this local state would be the empty ones. In contrast, the following states are ceremonious:

$$\begin{array}{l} ((s, h, [\varepsilon : (\varepsilon', \mathfrak{C}, 1, q_0), \varepsilon' : (\varepsilon, \mathfrak{C}, 2, q_0)]), \dot{u}, [\varepsilon : \sqcup, \varepsilon' : \sqcup]) \\ ((s, h, [\varepsilon : (\varepsilon', \mathfrak{C}, 1, q_1), \varepsilon' : (\varepsilon, \mathfrak{C}, 2, q_0)]), \dot{u}, [\varepsilon : \sqcup, \varepsilon' : (a, v, \dot{u})]) \\ ((s, h, [\varepsilon : (\varepsilon', \mathfrak{C}, 1, q_1), \varepsilon' : (\varepsilon, \mathfrak{C}, 2, q_1)]), \dot{u}, [\varepsilon : \sqcup, \varepsilon' : \sqcup]) \end{array}$$

Let us formalise this intuition.

Definition 6.6 (Ceremonious state) *Given a well-formed open state $\underline{\sigma}$ and an endpoint ε such that $cconf(\underline{\sigma},\varepsilon)$ is defined and $contract(\underline{\sigma},\varepsilon) = \mathfrak{C}$, we say that $\underline{\sigma}$ is* ceremonious for ε *if*

$$\exists C \in cconf(\underline{\sigma},\varepsilon).\, \langle init(\mathfrak{C}), init(\mathfrak{C}), \sqcup, \sqcup\rangle \rightarrow_{\mathfrak{C}}^{*} C\ .$$

A state $\underline{\sigma}$ is ceremonious *if it is well-formed and ceremonious for each endpoint ε such that $cconf(\underline{\sigma},\varepsilon)$ is defined.*

The set of ceremonious states is denoted by $\underline{\mathsf{State}}^{\mathsf{ctt}}$, and the set of ceremonious states that are well-formed with respect to Γ by $\underline{\mathsf{State}}^{\mathsf{ctt}}_{\Gamma}$. Note that $\underline{\mathsf{State}}^{\mathsf{ctt}} \subset \underline{\mathsf{State}}^{\mathsf{wf}}$ and $\underline{\mathsf{State}}^{\mathsf{ctt}}_{\Gamma} \subseteq \underline{\mathsf{State}}^{\mathsf{wf}}_{\Gamma}$.

Let us show that ceremonious states with respect to a footprint context are stable by transformations by ceremonious programs (those that do not reduce into a protocol error), with the help of the following lemma. For two sets of configurations S_1 and S_2, we say that there is a transition from S_1 to S_2 by the contract $\mathfrak{C}$ and write $S_1 \rightarrow_{\mathfrak{C}} S_2$ if

$$\forall C' \in S'.\, \exists C \in S.\, C \rightarrow_{\mathfrak{C}} C'\ .$$

Lemma 6.1 *For all $p, p', \underline{\sigma}, \underline{\sigma}', \varepsilon, \varepsilon'$, if*

- $p, \underline{\sigma} \rightarrow p', \underline{\sigma}'$
- $(\varepsilon, \varepsilon') \in chans(\underline{\sigma}')$
- $contract(\underline{\sigma}', \varepsilon) = \mathfrak{C}$

then

1. *either $contract(\underline{\sigma},\varepsilon)$ is undefined and $cconf(\underline{\sigma}',\varepsilon) = \{\langle init(\mathfrak{C}), init(\mathfrak{C}), \sqcup, \sqcup\rangle\}$*
2. *or $contract(\underline{\sigma},\varepsilon) = \mathfrak{C}$, $(\varepsilon,\varepsilon') \in chans(\underline{\sigma},\varepsilon)$ and*

$$\begin{cases} cconf(\underline{\sigma},\varepsilon) \rightarrow_{\mathfrak{C}} cconf(\underline{\sigma}',\varepsilon) \text{ or} \\ cconf(\underline{\sigma},\varepsilon) = cconf(\underline{\sigma}',\varepsilon) \end{cases}$$

Proof Let $\underline{\sigma} = (\dot{\sigma}, \underline{k})$ and $\underline{\sigma}' = (\dot{\sigma}', \underline{k}')$. If the transition $p, \underline{\sigma} \rightarrow p', \underline{\sigma}'$ is not a channel operation then $\underline{k} = \underline{k}'$ and the result is immediate. Let us analyse the remaining cases. Let us fix some ε satisfying all the above hypothesis.

Let us assume first that ε is not one of the two endpoints of the channel over which the instruction applies. It can be observed that $control(\cdot,\varepsilon)$ is either undefined in both $\underline{\sigma}$ and $\underline{\sigma}'$, or defined in both $\underline{\sigma}$ and $\underline{\sigma}'$, in which case $control(\underline{\sigma},\varepsilon) = control(\underline{\sigma}',\varepsilon)$ (recall that $control(\underline{\sigma},\cdot)$ is defined according to the flattening of $\underline{\sigma}$). Thus, for an endpoint that is not one of the endpoints of the channel over which some action is performed, the constraint is the same in $\underline{\sigma}$ and $\underline{\sigma}'$, and $cconf(\underline{\sigma},\varepsilon) = cconf(\underline{\sigma}',\varepsilon)$, which shows that case 2 above holds.

Let us assume now that ε is one of the two endpoints of the channel over which the instruction applies, and reason by case analysis on the instruction.

- `close`: Since $\mathsf{contract}(\underline{\sigma}', \varepsilon) = \mathfrak{C}$ is defined by hypothesis, ε is not the closed channel. This case is thus impossible.

- `open`: Case 1 above holds.

- `send`: Let us show that case 2 above holds, and in particular that $\mathit{cconf}(\underline{\sigma}, \varepsilon) \to_{\mathfrak{C}} \mathit{cconf}(\underline{\sigma}', \varepsilon)$. By symmetry, we may assume that ε is the endpoint used for sending. By the rule of `send`, $\mathit{control}(\underline{\sigma}, \varepsilon)$ and $\mathit{control}(\underline{\sigma}', \varepsilon)$ must be defined and equal to some control states q_1 and q_2. Moreover, the rule also ensure that there is a transition $q_1 \xrightarrow{!a} q_2$ in $\mathfrak{C}$. Let $C_2 = \langle q_2, q_2', w_2, w_2' \rangle$ be a configuration in $\mathit{cconf}(\underline{\sigma}', \varepsilon)$. Then $w_2' = w_1' \cdot a$, and $C_1 = \langle q_1, q_2', w_2, w_1' \rangle \to C_2$. Moreover, $C_1 \in \mathit{cconf}(\underline{\sigma}', \varepsilon)$ since q_2' is constrained in the same way in $\underline{\sigma}$ and $\underline{\sigma}'$.

- `receive` and external choice: Same arguments as for `send`. □

Lemma 6.2 (Subject reduction) *If* $\underline{\sigma} \in \underline{\mathsf{State}}^{\mathsf{ctt}}_{\Gamma}$ *and* $p, \underline{\sigma} \to p', \underline{\sigma}'$*, then* $\underline{\sigma}' \in \underline{\mathsf{State}}^{\mathsf{ctt}}_{\Gamma}$.

Proof The fact that well-formedness with respect to a footprint context is preserved by the reductions of a program is immediate from the reduction rules presented in the previous subsection. Let us focus on the ceremonious property.

Let us check each point of the definition of ceremonious states for $\underline{\sigma}'$. Firstly, it is flattable. Secondly, let $(\varepsilon, \varepsilon') \in \mathit{chans}(\underline{\sigma}')$ be such that $\mathit{contract}(\underline{\sigma}', (\varepsilon, \varepsilon'))$ is defined and equal to $\mathfrak{C}$, with $\mathit{init}(\mathfrak{C}) = q_0$. We need to show that there is $C' \in \mathit{cconf}(\underline{\sigma}', \varepsilon)$ such that

$$\langle q_0, q_0, \sqcup, \sqcup \rangle \to^*_{\mathfrak{C}} C' .$$

We can apply the previous lemma. If we are in the first case, the result is immediate. If we are in the second case, then either $\forall C' \in \mathit{cconf}(\underline{\sigma}', \varepsilon). \exists C \in \mathit{cconf}(\underline{\sigma}, \varepsilon). C \to_{\mathfrak{C}} C'$ or $\mathit{cconf}(\underline{\sigma}, \varepsilon) = \mathit{cconf}(\underline{\sigma}', \varepsilon)$. From the hypothesis that $\underline{\sigma}$ is ceremonious, there is $C \in \mathit{cconf}(\underline{\sigma}, \varepsilon)$ such that $\langle q_0, q_0, \sqcup, \sqcup \rangle \to^*_{\mathfrak{C}} C$. If $\mathit{cconf}(\underline{\sigma}, \varepsilon) = \mathit{cconf}(\underline{\sigma}', \varepsilon)$, then $C \in \mathit{cconf}(\underline{\sigma}', \varepsilon)$. Otherwise since $p, \underline{\sigma} \to p', \underline{\sigma}'$, $p, \underline{\sigma} \not\to \mathbf{ProtoError}$, hence $\mathit{cconf}(\underline{\sigma}', \varepsilon) \neq \emptyset$. Since moreover $\mathit{cconf}(\underline{\sigma}, \varepsilon) \to_{\mathfrak{C}} \mathit{cconf}(\underline{\sigma}', \varepsilon)$, there are finally $C \in \mathit{cconf}(\underline{\sigma}, \varepsilon)$ and $C' \in \mathit{cconf}(\underline{\sigma}', \varepsilon)$ such that $\langle q_0, q_0, \sqcup, \sqcup \rangle \to^*_{\mathfrak{C}} C \to_{\mathfrak{C}} C'$, which ends the proof. □

6.3 Soundness

6.3.1 Locality up to interferences

We prove here crucial technical lemmas about the properties of the open operational semantics that will allow us to derive the soundness of our proof system in the next subsection. The structure of the lemmas is similar in essence to previous work, for instance by Brookes [Bro07] or Vafeiadis [Vaf07]. There is however one important difference: in the present setting, interferences are needed to express the behaviour of an open program (one that may receive messages from the environment or send to it, because it possesses only one end of a particular channel), which has non-trivial consequences on the formulation of the locality and parallel decomposition lemmas (Lemmas 6.5 and 6.6).

$$\frac{(\dot{\sigma}, \underline{k}') \in \underline{\mathsf{State}}^{\mathsf{ctt}}_{\Gamma}}{(\dot{\sigma}, \underline{k}) \dashrightarrow (\dot{\sigma}, \underline{k}')} \qquad \frac{\underline{\sigma} \dashrightarrow \underline{\sigma}'}{p, \underline{\sigma} \rightsquigarrow p, \underline{\sigma}'} \qquad \frac{p, \underline{\sigma} \rightarrow p', \underline{\sigma}'}{p, \underline{\sigma} \rightsquigarrow p', \underline{\sigma}'}$$

Figure 6.7: Open operational semantics of interferences.

Environment interferences Interferences from the environment are modelled by a single rule, given in Figure 6.7. The rule transforms an open state into one with the same local state, but where the contents of the buffers may have changed. The resulting state must remain ceremonious with respect to the footprint context. The changes include the possibility for the environment to perform sends and receives on endpoints that are not directly controlled by the program, in accordance with their contracts, and to open and close channels not visible to the program. Interferences are denoted using a dashed arrow "$\dashrightarrow$," and transitions that are either a program step or an interference are written with a squiggly arrow "$\rightsquigarrow$."

This modelling is an over-approximation of what a ceremonious environment (that also respects the ownership hypothesis) might really do: with this definition, the environment can modify the buffers of endpoints owned by the program, and even of endpoints in the local state of the program, provided that it leaves the buffers in a state coherent with the protocols of these endpoints. An environment respecting the ownership hypothesis would not modify the buffers of endpoints owned by the program. We claim that giving a more precise account of interferences is possible, but that it would needlessly complicate the presentation since this over-approximation is enough to prove the soundness of our logic.

Let us first make an observation about the behaviour of interferences on the substates of an open state, which follows from the following lemma on ceremonious states.

Lemma 6.3 *For all $\dot{\sigma}$, $\dot{\sigma}' \in \dot{\mathsf{State}}$ and $\underline{k} \in \underline{\mathsf{EHeap}}$, if $(\dot{\sigma} \bullet \dot{\sigma}', \underline{k}) \in \underline{\mathsf{State}}^{\mathsf{ctt}}_{\Gamma}$ then $(\dot{\sigma}, \underline{k}) \in \underline{\mathsf{State}}^{\mathsf{ctt}}_{\Gamma}$.*

Proof Removing pieces of the local state merely removes constraints on the resulting open state. □

Lemma 6.4 *For all pairs of compatible open states $\underline{\sigma}_1$, $\underline{\sigma}_2$ such that $\underline{\sigma}_1 \bullet \underline{\sigma}_2 \in \underline{\mathsf{State}}^{\mathsf{ctt}}_{\Gamma}$, if*

$$\underline{\sigma}_1 \bullet \underline{\sigma}_2 \dashrightarrow \underline{\sigma}'$$

then there are $\underline{\sigma}'_1$, $\underline{\sigma}'_2$ such that $\underline{\sigma}'_1 \bullet \underline{\sigma}'_2 = \underline{\sigma}'$ and

$$\underline{\sigma}_1 \dashrightarrow \underline{\sigma}'_1 \qquad\qquad \underline{\sigma}_2 \dashrightarrow \underline{\sigma}'_2$$

Proof Let $\underline{\sigma}_1 = (\dot{\sigma}_1, \underline{k}_1)$ and $\underline{\sigma}_2 = (\dot{\sigma}_2, \underline{k}_2)$. According to the hypotheses, there is $\underline{k}'$ such that

$$(\dot{\sigma}_1 \bullet \dot{\sigma}_2, \underline{k}_1 \cup \underline{k}_2) \dashrightarrow (\dot{\sigma}_1 \bullet \dot{\sigma}_2, \underline{k}') \ .$$

By definition of interferences, $(\dot{\sigma}_1 \bullet \dot{\sigma}_2, \underline{k}') \in \underline{\mathsf{State}}^{\mathsf{ctt}}_{\Gamma}$, hence by Lemma 6.3, $(\dot{\sigma}_1, \underline{k}') \in \underline{\mathsf{State}}^{\mathsf{ctt}}_{\Gamma}$, so $\underline{\sigma}_1 \dashrightarrow (\dot{\sigma}_1, \underline{k}')$, and similarly for $\underline{\sigma}_2$, which concludes the proof. □

We can now state the locality and parallel decomposition lemmas, which are crucial ingredients of the proof of soundness of the next subsection.

Lemma 6.5 (Locality) *For all program p and open states $\underline{\sigma}_1$ and $\underline{\sigma}_2$ such that $\underline{\sigma}_1 \sharp \underline{\sigma}_2$ and $\underline{\sigma}_1 \bullet \underline{\sigma}_2 \in \underline{\mathsf{State}}^{\mathsf{ctt}}_{\Gamma}$,*

1. *if $p, \underline{\sigma}_1 \bullet \underline{\sigma}_2 \rightsquigarrow^* \mathbf{error}$ then $p, \underline{\sigma}_1 \rightsquigarrow^* \mathbf{error}$ or $p, \underline{\sigma}_1 \rightsquigarrow^* \mathbf{OwnError}$;*
2. *if $p, \underline{\sigma}_1 \bullet \underline{\sigma}_2 \rightsquigarrow^* p', \underline{\sigma}'$ then either $p, \underline{\sigma}_1 \rightsquigarrow^* \mathbf{error}$ or there exists $\underline{\sigma}'_1$, $\underline{\sigma}'_2$ such that*
   - $\underline{\sigma}' = \underline{\sigma}'_1 \bullet \underline{\sigma}'_2$;
   - $p, \underline{\sigma}_1 \rightsquigarrow^* p', \underline{\sigma}'_1$;
   - $\underline{\sigma}_2 \dashrightarrow \underline{\sigma}'_2$.

Proof The two parts of the lemma can be proved independently. The first part of the lemma follows from a straightforward induction on the length of the derivation and a case analysis on p. Let us focus on the second property.

For all commands except for communications and interferences, this property actually has a stronger formulation, identical to the usual locality principle of separation logic models, where interferences are not needed and $\underline{\sigma}_2$ remains untouched. For all these commands, and more generally for all programs p' made only out of these commands, if $p', \underline{\sigma}_1 \bullet \underline{\sigma}_2 \rightarrow^* p', \underline{\sigma}'$ then either $p', \underline{\sigma}_1 \rightarrow^* \mathbf{error}$ or there exists $\underline{\sigma}'_1 \sharp \underline{\sigma}_2$ such that

- $\underline{\sigma}' = \underline{\sigma}'_1 \bullet \underline{\sigma}_2$;
- $p, \underline{\sigma}_1 \rightarrow^* p', \underline{\sigma}'_1$.

This is a standard result, straightforward to prove by a case analysis on p' [Bro07, Vaf07].

Let us now examine the case of `open`, `close`, `send`, external choice, `receive`, and interferences. Let $\underline{\sigma}_1 = ((s_1, h_1, \dot{k}_1), \underline{k}_1)$ and $\underline{\sigma}_2 = (\dot{\sigma}_2, \underline{k}_2)$.

- $p = \texttt{(e,f) = open}(\mathfrak{C})$: Straightforward: $\varepsilon, \varepsilon' \in \mathsf{Endpoint} \setminus dom(\underline{k}_1 \cup \underline{k}_2)$ implies $\varepsilon, \varepsilon' \in \mathsf{Endpoint} \setminus dom(\underline{k}_1)$. No interference is needed on $\underline{\sigma}_2$.
- $p = \texttt{close}(E_1, E_2)$: A step of interference can erase the endpoints involved if they are also present in $\underline{k}_2$ and p has deallocated them from $\underline{k}_1$.
- Suppose that $p = \texttt{send}(\texttt{a}, E_1, E_2)$, that all the hypotheses of the successful execution of p hold on $\underline{\sigma}_1$ (otherwise the command faults, which also satisfies the lemma) and that the execution results in $\underline{\sigma}'_1$. Because of the precision assumption on message footprints, the session state $\dot{\sigma}_a$ chosen in $\underline{\sigma}_1$ must be the same as the one from the execution on $\underline{\sigma}$. It suffices to take $\underline{\sigma}'_2 = (\dot{\sigma}_2, [\underline{k} \mid \varepsilon' : \underline{k}_1(\varepsilon') \cdot (a, v, \dot{\sigma}_a)])$ to have $\underline{\sigma}'_1 \bullet \underline{\sigma}'_2 = \underline{\sigma}'$.

  Note that this reasoning would be invalidated by an imprecise footprint γ_a for a: in that case, if we pick $\underline{\sigma}_2$ to be the piece of state whose local part is the session state $\dot{\sigma}_a$

chosen by the reduction of p on $\underline{\sigma}$ to be sent away, and if $\underline{\sigma}_1$ contains another possible candidate $\dot{\sigma}'_a$ satisfying γ_a, then the semantics could chose to send $\dot{\sigma}'_a$ starting from $\underline{\sigma}_1$, and the resulting state could not be connected back to $\underline{\sigma}'$ (in particular, it would contain $\dot{\sigma}_a$ in its local part, whereas $\underline{\sigma}'$ does not).

- The case of `receive` is similar. The requirement of having precise footprints does not impact the reasoning for this command, since the piece of state to be received is already determined by the contents of the buffer.
- If the step was an interference then the property to prove corresponds to Lemma 6.4. □

Lemma 6.6 (Parallel decomposition) *For all pair of programs* p_1, p_2, *for all state* $\underline{\sigma}$ *and for all* $\underline{\sigma}_1$, $\underline{\sigma}_2$ *such that* $\underline{\sigma}_1 \bullet \underline{\sigma}_2 = \underline{\sigma}$,

1. *if* $p_1 \parallel p_2, \underline{\sigma} \rightsquigarrow^* \mathbf{error}$ *then* $p_1, \underline{\sigma}_1 \rightsquigarrow^* \mathbf{error}$ *or* $p_2, \underline{\sigma}_2 \rightsquigarrow^* \mathbf{error}$*;*
2. *if* $p_1 \parallel p_2, \underline{\sigma} \rightsquigarrow^* p'_1 \parallel p'_2, \underline{\sigma}'$ *then* $p_1, \underline{\sigma}_1 \rightsquigarrow^* \mathbf{error}$ *or* $p_2, \underline{\sigma}_2 \rightsquigarrow^* \mathbf{error}$ *or there are disjoint states* $\underline{\sigma}'_1$, $\underline{\sigma}'_2$ *such that* $\underline{\sigma}' = \underline{\sigma}'_1 \bullet \underline{\sigma}'_2$ *and*
   - $p_1, \underline{\sigma}_1 \rightsquigarrow^* p'_1, \underline{\sigma}'_1$ *and*
   - $p_2, \underline{\sigma}_2 \rightsquigarrow^* p'_2, \underline{\sigma}'_2$.

Proof The first point is obvious from the semantics of $\parallel$ and $\underline{norace}$ by using Lemma 6.5.1. For the second point, let us proceed by induction on the length n of the computation of $p_1 \parallel p_2$, the base case of a computation of length 0 being trivial.

Suppose that the proposition holds for n computation steps, and that $p_1 \parallel p_2, \underline{\sigma} \rightsquigarrow^{n+1} p'_1 \parallel p'_2, \underline{\sigma}'$. This execution can be decomposed as

$$p_1 \parallel p_2, \underline{\sigma} \rightsquigarrow p''_1 \parallel p''_2, \underline{\sigma}'' \rightsquigarrow^n p'_1 \parallel p'_2, \underline{\sigma}'$$

By induction hypothesis, for all $\underline{\sigma}''_1, \underline{\sigma}''_2$ such that $\underline{\sigma}''_1 \bullet \underline{\sigma}''_2 = \underline{\sigma}''$, either p''_1 or p''_2 faults respectively on $\underline{\sigma}''_1$ and $\underline{\sigma}''_2$, or there are disjoint states $\underline{\sigma}'_1$, $\underline{\sigma}'_2$ such that $\underline{\sigma}' = \underline{\sigma}'_1 \bullet \underline{\sigma}'_2$ and

- $p''_1, \underline{\sigma}''_1 \rightsquigarrow^* p'_1, \underline{\sigma}'_1$ and
- $p''_2, \underline{\sigma}''_2 \rightsquigarrow^* p'_2, \underline{\sigma}'_2$.

We have to prove that either p_1 or p_2 faults respectively on $\underline{\sigma}_1$ and $\underline{\sigma}_2$, or that there exist p''_1, p''_2, $\underline{\sigma}''_1$ and $\underline{\sigma}''_2$ such that $p_1, \underline{\sigma}_1 \rightsquigarrow^* p''_1, \underline{\sigma}''_1$, $p_2, \underline{\sigma}_2 \rightsquigarrow^* p''_2, \underline{\sigma}''_2$ and $\underline{\sigma}''_1 \bullet \underline{\sigma}''_2 = \underline{\sigma}''$. For this purpose, we perform a case analysis on the first step of the computation of $p_1 \parallel p_2$:

- If this was an environment step, then $p''_1 = p_1$, $p''_2 = p_2$ and Lemma 6.4 gives us $\underline{\sigma}''_1$ and $\underline{\sigma}''_2$ such that $\underline{\sigma}''_1 \bullet \underline{\sigma}''_2 = \underline{\sigma}''$.
- If p_1 did a step: $p_1, \underline{\sigma} \rightarrow p''_1, \underline{\sigma}''$, then by Lemma 6.5, either $p_1, \underline{\sigma}_1 \rightarrow \mathbf{error}$ or there are $\underline{\sigma}''_1$, $\underline{\sigma}''_2$ such that $p_1, \underline{\sigma}_1 \rightsquigarrow p''_1, \underline{\sigma}''_1$, $\underline{\sigma}_2 \dashrightarrow \underline{\sigma}''_2$ and $\underline{\sigma}''_1 \bullet \underline{\sigma}''_2 = \underline{\sigma}''$. By taking $p''_2 = p_2$, we finish the proof.
- If p_2 did a step the argument is symmetric. □

6.3.2 Soundness

Before defining what it means semantically for a program to satisfy a specification given as a Hoare triple, we need to define what it means for an open state to satisfy a formula of our logic.

Definition 6.7 (Formula satisfaction for open states) *An open state $\underline{\sigma}$ satisfies a formula ϕ, written $\underline{\sigma} \vDash \phi$, if its local part does:*

$$(\dot{k}, \underline{k}), i \vDash \phi \text{ iff } \dot{k}, i \vDash \phi \ .$$

Let us remark that formula satisfaction is stable under interferences from the environment.

Lemma 6.7 *For all $\underline{\sigma}, \underline{\sigma}' \in \underline{\mathsf{State}}^{\mathsf{ctt}}_{\Gamma}$, all formula ϕ and all valuation i, if $\underline{\sigma}, i \vDash \phi$ and $\underline{\sigma} \dashrightarrow \underline{\sigma}'$ then $\underline{\sigma}', i \vDash \phi$.*

Proof Immediate using the definitions of satisfaction and interferences: the latter cannot change the local state of $\underline{\sigma}$, which is the only part of the state that is used to determine whether or not $\underline{\sigma}, i \vDash \phi$ holds. □

Definition 6.8 (Validity) *A triple is* valid with respect to a footprint context Γ, *written $\vDash_{\Gamma} \{\phi\}\ p\ \{\psi\}$, if, for all logical stack i and all state $\underline{\sigma} \in \underline{\mathsf{State}}^{\mathsf{ctt}}_{\Gamma}$, if $\underline{\sigma}, i \vDash \phi$, then the following properties hold:*

1. *$p, \underline{\sigma} \not\rightsquigarrow^{*}$ **OwnError**;*
2. *$p, \underline{\sigma} \not\rightsquigarrow^{*}$ **ProtoError**;*
3. *if $p, \underline{\sigma} \rightsquigarrow^{*}$ `skip`$, \underline{\sigma}'$, then $\underline{\sigma}', i \vDash \psi$.*

Theorem 6.1 (Soundness) *If $\vdash_{\Gamma} \{\phi\}\ p\ \{\psi\}$ then $\vDash_{\Gamma} \{\phi\}\ p\ \{\psi\}$.*

Proof We prove this theorem by induction on the structure of the proof of $\{\phi\}\ p\ \{\psi\}$. Suppose first that it is one of the axioms of the proof system (see Figures 4.4 and 5.1). We proceed by case analysis. Let $\underline{\sigma} = ((s, h, \dot{k}), \underline{k})$.

- Assume: Suppose that $\underline{\sigma}, i \vDash var(B) \Vdash \mathsf{emp}_h$. If follows that $dom(s) = var(B)$, hence $[\![B]\!]_s$ is defined. If it evaluates to true, then $\underline{\sigma}, i \vDash var(B) \Vdash B \wedge \mathsf{emp}_h$, which is the postcondition. If not, then the program diverges, hence the property is also satisfied.

- Assign, New, Dispose, Lookup, Mutate, Open, Close: similarly straightforward.

- Send: Suppose that

$$\underline{\sigma}, i \vDash \begin{array}{ll} & E_1 = X \wedge E_2 = Y \\ \wedge & (X \mapsto (X', \mathfrak{C}, r, q) * (X \mapsto (X', \mathfrak{C}, r, q') \twoheadrightarrow \gamma_a(X, X', r, Y) * \phi)) \end{array}$$

In particular, the local part of $\underline{\sigma}$ holds ownership of $var(E_1, E_2)$ and of the endpoint pointed to by E_1. If $i(X) = \varepsilon$, $i(X') = \varepsilon'$ and $i(Y) = v$, then

$$\underline{\sigma}, i \vDash X \mapsto (X', \mathfrak{C}, r, q) * (X \mapsto (X', \mathfrak{C}, r, q') \mathrel{-\!\!*} \gamma_a(X, X', r, Y) * \phi)$$

Thus, $\dot{\sigma} = (s, h, [\dot{k} \mid control(\varepsilon){\leftarrow}q']), i \vDash \gamma_a(X, X', r, Y) * \phi$. Hence, there are $\dot{\sigma}_a$ and $\dot{\sigma}'$ such that $\dot{\sigma}_a \bullet \dot{\sigma}' = \dot{\sigma}$, $\dot{\sigma}_a, i \vDash \gamma_a(X, X', r, Y)$ and $\dot{\sigma}', i \vDash \phi$. This forbids the program to reduce to **OwnError** by the second reduction rule of `send`. The premise of the Send rule moreover forbids the program to reduce to **ProtoError**, hence only the first (and successful) reduction rule of `send` applies, and the resulting state is $(\dot{\sigma}', [\underline{k} \mid \varepsilon' : \underline{k}(\varepsilon') \cdot (a, v, \dot{\sigma}_a)])$. As previously stated $\dot{\sigma}', i \vDash \phi$, hence the result.

Let us now assume that $\vdash_\Gamma \{\phi\}\ p\ \{\psi\}$ is not an axiom. We discriminate on the last inference rule of the proof tree (see Figures 5.1, 4.5 and 4.6). Let again $\underline{\sigma} = ((s, h, \dot{k}), \underline{k})$.

- ChannelDispatch: Suppose that $\underline{\sigma}, i \vDash (Y', \mathfrak{C}, r, q) * \phi$. The first precondition of the inference rule,

  $$\phi \Rightarrow \bigwedge_{i=1}^{n} (E_i = Y \wedge \mathtt{x}_i = X_i) * \mathsf{true}$$

  ensures that the program has enough resources not to reduce to **OwnError** in one step. The second one,

  $$\{a_i \mid i \in \{1, \ldots, n\}\} = \{a \mid \exists q'.\, succ(\mathfrak{C}, r, q, ?a) = q'\}$$

  prevents both the second and the last of the four reduction rules of `receive` to apply, hence $p, \underline{\sigma} \not\to$ **ProtoError**. Let us assume that $p, \underline{\sigma} \not\to$ **MsgError**. Then, there is i such that

  $$p, \underline{\sigma} \to p_i, (([s \mid \mathtt{x}_i : v], h, [\dot{k} \mid control(\varepsilon_i){\leftarrow}q']) \bullet \dot{\sigma}_i, [\underline{k} \mid \varepsilon_i : \underline{\alpha}])$$

  Let us write $\underline{\sigma}'$ for the resulting open state. Since $\underline{\sigma} \in \underline{\mathsf{State}}_\Gamma^{\mathsf{ctt}}$, $\dot{\sigma}_i, i \vDash \gamma_{a_i}(Y', Y, 3 - r, Z)$ for some Z such that $i(Z) = v$. Hence,

  $$\underline{\sigma}', i \vDash \mathtt{x}_i = Z \wedge (Y \mapsto (Y', \mathfrak{C}, r, q') * \phi * \gamma_{a_i}(Y', Y, 3 - r, Z))$$

  By induction hypothesis, and since we know from the premises of ChannelDispatch that

  $$\vdash_\Gamma \{\mathtt{x}_i = Z \wedge (Y \mapsto (Y', \mathfrak{C}, r, q') * \phi * \gamma_{a_i}(Y', Y, 3 - r, Z))\}\ p_i\ \{\psi\}$$

  we can deduce that $p_i, \underline{\sigma}' \not\to^*$ **OwnError**, $p_i, \underline{\sigma}' \not\to^*$ **ProtoError**, and if $p_i, \underline{\sigma}' \leadsto^*$ `skip`, $\underline{\sigma}''$, then $\underline{\sigma}'', i \vDash \psi$. Hence, $p, \underline{\sigma} \not\to^*$ **OwnError**, $p, \underline{\sigma} \not\to^*$ **ProtoError**, and if $p, \underline{\sigma} \leadsto^*$ `skip`, $\underline{\sigma}''$, then $\underline{\sigma}'', i \vDash \psi$.

- ExtChoice: This rule is sound by a straightforward generalisation of the previous argument to multiple endpoints.

- Sequence: Straightforward.
- Parallel: Straightforward from Lemma 6.6.
- Choice, Star, Local: Straightforward.
- Frame: Recall that for all program p, $p \sim p \parallel \mathtt{skip}$ (Lemma 2.2). This is also true in the open semantics: $p, \underline{\sigma} \rightsquigarrow^* \mathtt{skip}, \underline{\sigma}'$ if and only if $p \parallel \mathtt{skip}, \underline{\sigma} \rightsquigarrow^* \mathtt{skip}, \underline{\sigma}'$, and $p, \underline{\sigma} \rightsquigarrow^* \mathbf{error}$ if and only if $p \parallel \mathtt{skip}, \underline{\sigma} \rightsquigarrow^* \mathbf{error}$. By applying the parallel decomposition lemma to $p \parallel \mathtt{skip}$, one obtains the soundness of the Frame rule.
- Weakening, Conjunction, Disjunction, Existential: Straightforward.
- Renaming: We can prove by structural induction on the formulas that for all $\dot{\sigma}$, i and ϕ, and for all variables $\mathtt{x}$ and $\mathtt{y}$, $\dot{\sigma}, i \vDash \phi$ if and only if $\dot{\sigma}[\mathtt{x} \leftrightarrow \mathtt{y}], i \vDash \phi[\mathtt{x} \leftrightarrow \mathtt{y}]$. It can also be proved by a straightforward structural induction on the program that for all p, $p, \underline{\sigma} \rightsquigarrow^* \mathbf{error}$ if and only if $p[\mathtt{x} \leftrightarrow \mathtt{y}], \underline{\sigma}[\mathtt{x} \leftrightarrow \mathtt{y}] \rightsquigarrow^* \mathbf{error}$ and that $p, \underline{\sigma} \rightsquigarrow^* \underline{\sigma}'$ if and only if $p[\mathtt{x} \leftrightarrow \mathtt{y}], \underline{\sigma}[\mathtt{x} \leftrightarrow \mathtt{y}] \rightsquigarrow^* \underline{\sigma}'[\mathtt{x} \leftrightarrow \mathtt{y}]$. These facts entail the soundness of the Renaming rule.

6.4 Properties of proved programs

This section is dedicated to the consequences of the soundness theorem on proved programs: we show that proved programs whose contracts are fault-free enjoy "runtime safety" (Theorem 6.2), that is they do not reduce into any error in the closed semantics, and that programs proved under an admissible footprint context and with leak-free contracts enjoy leak freedom (Theorem 6.3). Let us begin by defining how to obtain a closed state from an open one.

6.4.1 From open states to closed states

To transform an open state into a closed one, one first needs to transform its open buffers into closed ones. Given an open buffer $\underline{\alpha} = (a_1, v_1, \dot{k}_1)\cdots(a_n, v_n, \dot{k}_n)$, we write $\eta(\underline{\alpha})$ for its canonical projection onto a closed buffer in $(\mathsf{MsgId} \times \mathsf{Val})^*$:

$$\eta(\underline{\alpha}) \triangleq (a_1, v_1)\cdots(a_n, v_n)\ .$$

This notion is easily extended to project an open endpoint heap $\underline{k}$ onto a closed one $\eta(\underline{k})$ (with the obvious lifting of η), and open states onto closed ones (overloading η once more):

$$\eta : \begin{array}{c} \underline{\mathsf{State}} \to \mathsf{State} \\ ((s, h, \dot{k}), \underline{k}) \mapsto (s, h, \eta(\underline{k})) \end{array}$$

This projection is unfortunately too coarse for our needs, as much of the state may be buried inside the open buffers. Instead, we define the closed state $close(\underline{\sigma})$ corresponding to $\underline{\sigma}$ by the projection above applied to the *flattening* of $\underline{\sigma}$.

Definition 6.9 (Closed state of an open state) *The closed state $close(\underline{\sigma})$ of an opened state $\underline{\sigma} = (\dot{\sigma}, \underline{k})$ such that $flat(\underline{\sigma}) = (s_f, h_f, \dot{k}_f)$ is defined as*

$$close(\underline{\sigma}) \triangleq (s_f, h_f, \eta(\underline{k}))\ .$$

The closed state associated to a well-formed open state is also well-formed.

Lemma 6.8 *For all $\underline{\sigma} \in \underline{\mathsf{State}}^{\mathsf{wf}}$, $close(\underline{\sigma})$ is well-formed as a closed state.*

Proof The constraint (Channel) follows from the (OChannel) condition on $\underline{\sigma}$, and (Irreflexive), (Involutive) and (Injective) from the well-formedness of the local part of $flat(\underline{\sigma})$. □

6.4.2 Runtime validity

Let us show that programs proved under fault-free contracts do not reduce into an error in the closed semantics. More precisely, let us show that such programs are *runtime valid*.

Definition 6.10 (Runtime validity) *We say that a triple $\{\phi\}\ p\ \{\psi\}$ is* runtime valid *and write $\Vdash_\Gamma \{\phi\}\ p\ \{\psi\}$ if for all i and σ, there exists $\underline{\sigma}$ such that $\underline{\sigma}, i \vDash \phi$, $close(\underline{\sigma}) = \sigma$ and*

- $p, \sigma \not\Rightarrow^* \mathbf{OwnError}$ *and*
- $p, \sigma \not\Rightarrow^* \mathbf{MsgError}$ *and*
- *If $p, \sigma \Rightarrow^* \texttt{skip}, \sigma'$ then there exists $\underline{\sigma}'$ such that $\underline{\sigma}', i \vDash \psi$ and $close(\underline{\sigma}') = \sigma'$.*

To show that programs proved under fault-free contracts enjoy the above property, we will show that valid programs (in the sense of the open semantics, hence of Definition 6.8) are runtime valid when the contracts involved are fault-free. More precisely, we say that $\{\phi\}\ p\ \{\psi\}$ is valid under fault-free contracts if the contracts appearing in ϕ, ψ and p are all fault-free.

Lemma 6.9 *If $\vDash_\Gamma \{\phi\}\ p\ \{\psi\}$ under fault-free contracts then $\Vdash_\Gamma \{\phi\}\ p\ \{\psi\}$.*

If we manage to establish such a connection, then the result we seek will follow immediately from the soundness result of the previous section. The triple $\{\phi\}\ p\ \{\psi\}$ is provable with fault-free contracts if one of its proofs mentions only fault-free contracts (including the ones decorating p).

Theorem 6.2 *If $\vdash_\Gamma \{\phi\}\ p\ \{\psi\}$ with fault-free contracts then $\Vdash_\Gamma \{\phi\}\ p\ \{\psi\}$.*

Proved programs are race free Let us begin by proving that a valid program is race free, that is it does not reduce into $\mathbf{OwnError}$ in the closed semantics. In particular, we must prove that a race of $p_1 \parallel p_2$ on $close(\underline{\sigma})$ for the closed notion of race entails a race of the same program on $\underline{\sigma}$ for the open notion of race.

Lemma 6.10 *For all well-formed open state $\underline{\sigma}$, if $close(\underline{\sigma}) = \sigma$ and $p, close(\underline{\sigma}) \Rightarrow \mathbf{OwnError}$, then $p, \underline{\sigma} \rightarrow \mathbf{OwnError}$.*

Proof Let $\underline{\sigma} = ((s, h, \dot{k}), \underline{k})$ and $\sigma = close(\underline{\sigma}) = (s_c, h_c, k_c)$. We proceed by structural induction on p.

For all atomic commands, the lemma follows straightforwardly, either because the open semantics exhibits an extra faulting condition, or from the fact that $dom(s) \subseteq dom(s_c)$ and $dom(h) \subseteq dom(h_c)$.

The result is also straightforward to prove for all of the programming constructs reductions, except for the case where p exhibits a race.

Suppose now that $p = p_1 \| p_2$ and that $norace(p_1, p_2, \sigma)$ does not hold, that is $iw(p_1, \sigma) \cap ia(p_2, \sigma) \neq \emptyset$ or $iw(p_2, \sigma) \cap ia(p_1, \sigma) \neq \emptyset$. We need to show that, given any splitting $\underline{\sigma}_1 \bullet \underline{\sigma}_2$ of $\underline{\sigma}$, either $p_1, \underline{\sigma}_1 \rightarrow \mathbf{OwnError}$ or $p_2, \underline{\sigma}_2 \rightarrow \mathbf{OwnError}$.

Observe that in order for a program p not to reduce into an error in the open semantics, the local state must contain in its domain all the variables and locations that it immediately reads or writes to, that is the domains of the stack and heaps must contain $ia(p, \sigma)$. Thus, the fact that these sets are not disjoint for p_1 and p_2 implies that there is no way to split $\underline{\sigma}$ into two *disjoint* substates over which p_1 *and* p_2 would not fault. Indeed, in order for p_1 and p_2 not to fault respectively from $\underline{\sigma}_1$ and $\underline{\sigma}_2$, the domains of the stacks or the heaps of $\underline{\sigma}_1$ and $\underline{\sigma}_2$ must overlap. Hence, $\underline{norace}(p_1, p_2, \underline{\sigma})$ does not hold and $p_1 \| p_2, \underline{\sigma} \rightarrow \mathbf{OwnError}$. □

Proved programs with fault-free contracts are message fault-free We now turn to the second point of the runtime validity property that we wish to establish: we show that valid programs with fault-free contracts do not reduce into $\mathbf{MsgError}$ in the closed semantics. This is done in two steps: we first show that valid programs (in the sense of the open semantics, that is programs which do not reduce into an ownership error or a protocol error) with fault-free contracts do not fault on message receptions in the open semantics, and then that message reception errors in the closed semantics correspond to errors in the open one. Put together, these two facts show that valid programs with fault-free contracts have no unexpected receptions in the closed semantics.

Lemma 6.11 (Fault correspondence) *Let $\underline{\sigma}$ be a well-formed state with respect to Γ. If $p, \underline{\sigma} \not\rightarrow \mathbf{OwnError}$, $p, \underline{\sigma} \not\rightarrow \mathbf{ProtoError}$ and $p, \underline{\sigma} \rightarrow \mathbf{MsgError}$ then there is ε such that $contract(\underline{\sigma}, \varepsilon)$ is defined and $cconf(\underline{\sigma}, \varepsilon)$ contains an unspecified reception configuration for $contract(\underline{\sigma}, \varepsilon)$.*

Proof Assume the hypotheses. Since $p, \underline{\sigma} \rightarrow \mathbf{MsgError}$, there is a guarded external choice for which an unexpected message is in one of the considered buffers (say in the buffer of ε). Since $p, \underline{\sigma} \not\rightarrow \mathbf{OwnError}$, ε is in the domain of the local part of $\underline{\sigma}$, hence $contract(\underline{\sigma}, \varepsilon)$ and $cconf(\underline{\sigma}, \varepsilon)$ are defined. Moreover, since $p, \underline{\sigma} \not\rightarrow \mathbf{ProtoError}$, the messages expected by the guarded external choice composition match exactly the ones of $contract(\underline{\sigma}, \varepsilon)$. Thus, the unexpected message is also unexpected in any configuration of $cconf(\underline{\sigma}, \varepsilon)$ with respect to $contract(\underline{\sigma}, \varepsilon)$, hence $cconf(\underline{\sigma}, \varepsilon)$ contains an unspecified reception configuration for $contract(\underline{\sigma}, \varepsilon)$. □

In particular, if a program p reduces to $\mathbf{MsgError}$ from a ceremonious state $\underline{\sigma}$, but not to $\mathbf{OwnError}$ or $\mathbf{ProtoError}$, then either p is decorated by a faulty contract (one that is not fault-free), or $\underline{\sigma}$ contains one. Indeed, the contract configurations of channels are all reachable in $\underline{\sigma}$ and in the states resulting from the execution of p on $\underline{\sigma}$, hence one of the contracts can reach an unspecified reception configuration.

Lemma 6.12 *For all well-formed open state $\underline{\sigma}$, if $p, close(\underline{\sigma}) \Rightarrow \mathbf{MsgError}$ then either $p, \underline{\sigma} \rightarrow \mathbf{MsgError}$ or $p, \underline{\sigma} \rightarrow \mathbf{OwnError}$.*

Proof Straightforward. □

Proved programs follow their specifications Finally, let us show that the last part of the runtime validity definition holds for valid programs.

Lemma 6.13 *For all well-formed open state $\underline{\sigma}$, if $p, \sigma \Rightarrow^* p', \sigma'$ then either $p, \underline{\sigma} \rightarrow^*$ $\mathbf{error}$ with $\mathbf{error} \in \{\mathbf{LeakError}, \mathbf{OwnError}, \mathbf{ProtoError}\}$ or there exists $\underline{\sigma}' \in \underline{\mathsf{State}}^{\mathsf{wf}}_{\Gamma}$ such that $close(\underline{\sigma}') = \sigma'$ and $p, \underline{\sigma} \rightarrow^* p', \underline{\sigma}'$.*

Proof We proceed by case analysis on p. Let $\underline{\sigma} = (\dot{\sigma}, \underline{k})$, $\mathit{flat}(\underline{\sigma}) = \dot{\sigma}_f$, $\sigma = (s_c, h_c, k)$ and $\sigma' = (s'_c, h'_c, k')$. Suppose that $p, \sigma \Rightarrow^* p', \sigma'$ and $p, \underline{\sigma} \not\rightarrow^* \mathbf{error}$. We have to show that there exists $\underline{\sigma}'$ such that $close(\underline{\sigma}') = \sigma'$ and $p, \underline{\sigma} \rightarrow^* p', \underline{\sigma}'$.

- stack and heap commands: straightforward. In the case of `new`, observe that we can pick the same location as in σ' because $dom(h_f) = dom(h_c)$.
- `open`: straightforward.
- `close`: the flattening of the resulting state $\mathit{flat}(\underline{\sigma}')$ is equal to $\mathit{flat}(\underline{\sigma})$ where the endpoints have been deallocated from the endpoint heaps. From this observation, it follows that $close(\underline{\sigma}') = \sigma'$.
- `send`: If $p, \underline{\sigma} \rightarrow^* p', \underline{\sigma}'$ then it is easy to see that, since in the reduction rule $\dot{\sigma}_a$ is moved from the local portion of the state to the receiving queue of the peer ε' of the origin endpoint ε, $\dot{\sigma}'_f = \dot{\sigma}_f$ (using similar notations for $\underline{\sigma}'$ as for $\underline{\sigma}$). Moreover, if a is the message identifier and v its value, then $\underline{k}'(\varepsilon) = \underline{k}(\varepsilon)$ and $\underline{k}'(\varepsilon') = \underline{k}(\varepsilon') \cdot (a, v, \dot{\sigma}_a)$, hence $\eta(\underline{k}') = [\eta(\underline{k}) \mid \mathit{buffer}(\varepsilon') {\leftarrow} \eta(\underline{k}(\varepsilon')) \cdot (a, v)] = [k \mid \mathit{buffer}(\varepsilon') {\leftarrow} \mathit{buffer}(k, \varepsilon') \cdot (a, v)]$, which shows that $close(\underline{\sigma}') = \sigma'$.
- `receive`: If $p, \underline{\sigma} \rightarrow^* p', \underline{\sigma}'$ then it is easy to see that, since in the reduction rule $\dot{\sigma}_a$ is moved from the receiving queue of the endpoint to the local portion of the state, $\dot{\sigma}_f$ and $\dot{\sigma}'_f$ only differ in the value of $\mathtt{x}$ in the stack, which now contains v. Moreover, if ε is the receiving endpoint, a the message identifier and v its value and $\eta(\underline{k}(\varepsilon)) = (a_i, v) \cdot \alpha = \mathit{buffer}(k, \varepsilon)$, then $\eta(\underline{k}'(\varepsilon)) = \alpha = \mathit{buffer}(k', \varepsilon)$, hence $close(\underline{\sigma}') = \sigma'$.
- The other programming constructs are straightforward cases. □

By putting the three lemmas above together, we get a proof of Lemma 6.9, hence of Theorem 6.2.

6.4.3 Owned portion of a state

Our next property of interest is leak freedom. However, to define what constitutes a leak in the open semantics, we first neeed to introduce the notion of the *(reachable) owned portion*, or reachable local state, of a well-formed open state $\underline{\sigma} = (\dot{\sigma}, \underline{k})$. The owned portion of a state $\underline{\sigma}$ corresponds to every piece of state that can be reached in $\underline{\sigma}$ by retrieving the footprints of the messages in the queues of the endpoints present in $\dot{\sigma}$, adding them to the local state, and then repeating the process until a fixpoint is reached.[1] The fixpoint is reached when the queues accessible from the last local state all have empty footprints.

In the definition below, $emp(\underline{k})$ sets the footprints of the buffers of $\underline{k}$ to $\dot{u}$; it is the lifting to open endpoint heaps of the function

$$\mathsf{MsgId} \times \mathsf{Val} \times \dot{\mathsf{State}} \to \mathsf{MsgId} \times \mathsf{Val} \times \dot{\mathsf{State}}$$
$$(a, v, \dot{\sigma}) \mapsto (a, v, \dot{u})$$

In particular, $\mathit{flat}(emp(\underline{k})) = \dot{u}$ for all $\underline{k}$.

Definition 6.11 (Owned portion of a state) *Let $\underline{\sigma} = ((s, h, \dot{k}), \underline{k})$ be a well-formed open state. The* reachable owned portion *(or simply the owned portion) $owned(\underline{\sigma})$ of the state $\underline{\sigma}$ is defined inductively by:*

- *if $\mathit{flat}(\underline{k} \downharpoonright dom(\dot{k})) = \dot{u}$ then $owned(\underline{\sigma}) \triangleq (s, h, \dot{k})$;*
- *otherwise, let $\dot{\sigma}_{\underline{k}} = \mathit{flat}(\underline{k} \downharpoonright dom(\dot{k}))$ and $\underline{k}' = emp(\underline{k} \downharpoonright dom(\dot{k})) \uplus \underline{k} \downharpoonright (dom(\underline{k}) \setminus dom(\dot{k}))$. Then $owned(\underline{\sigma}) \triangleq owned((s, h, \dot{k}) \bullet \dot{\sigma}_{\underline{k}}, \underline{k}')$.*

This definition is well-founded since the number of buffers with non-empty footprints decreases strictly at each step. The resulting session state is well-formed.

Reachable owned states allow us to circumscribe the part of the state that the program owns, either directly (in its local state), or virtually, because the corresponding locations can become accessible after a certain number of receives.

Flattening gives an over-approximation of the owned portion of a state.

Lemma 6.14 *For all open well-formed state $\underline{\sigma}$, if $\mathit{flat}(\underline{\sigma}) = (\dot{\sigma}_f, \underline{k}_f)$, then*

$$owned(\underline{\sigma}) \preceq \dot{\sigma}_f \ .$$

Proof The proof is by induction on the number of steps in the computation of $owned(\underline{\sigma})$.

In the base case, we have $owned((\dot{\sigma}, \underline{k})) = \dot{\sigma} \preceq \dot{\sigma}_f$. Suppose that the lemma is true for states whose owned portion is computed in up to n steps. Let $\underline{\sigma} = (\dot{\sigma}, \underline{k})$ be a state whose owned portion can be computed in $n + 1$ steps, and let $\dot{\sigma} = (s, h, \dot{k})$ and $\dot{\sigma}_{\underline{k}} = \mathit{flat}(\underline{k} \downharpoonright dom(\dot{k}))$ and $\underline{k}' = emp(\underline{k} \downharpoonright dom(\dot{k})) \uplus \underline{k} \downharpoonright (dom(\underline{k}) \setminus dom(\dot{k}))$ as in the

[1] Gotsman *et al.* use a similar notion that they call "closure" to get around the issue of cycles of ownership leaks. [GBC+07]

definition. Let us moreover write $\pi_1((\dot\sigma, \underline{k}))$ for the local part $\dot\sigma$ of an open state $(\dot\sigma, \underline{k})$. We have:

$$\begin{aligned} \mathit{owned}(\underline{\sigma}) &= \mathit{owned}((\dot\sigma \bullet \dot\sigma_{\underline{k}}, \underline{k}')) && \text{(by definition)} \\ &\leq \pi_1(\mathit{flat}((\dot\sigma \bullet \dot\sigma_{\underline{k}}, \underline{k}'))) && \text{(induction hypothesis)} \\ &= \pi_1(\mathit{flat}((\dot\sigma, \underline{k}))) && \text{(by definition of } \dot\sigma_{\underline{k}} \text{ and } \underline{k}'\text{)} \\ &= \pi_1(\mathit{flat}(\underline{\sigma})) \end{aligned}$$

This concludes the proof. □

The other inclusion does not always hold: some of the queues may be unreachable from the local portion of an open state. Let us call the difference between $\mathit{flat}(\underline{\sigma})$ and $\mathit{owned}(\underline{\sigma})$ the *unreachable* portion of $\underline{\sigma}$. This unreachable part may come from two sources:

- either because some endpoints of the open endpoint belong to the environment (in the sense that, if $\mathit{owned}(\underline{\sigma}) = (s_o, h_o, \dot{k}_o)$, they are not in the domain of the endpoint heap $\dot{k}_o$). In this case, these endpoints and every footprint of every message appearing in their buffers will appear in $\mathit{flat}(\underline{\sigma})$ but not in $\mathit{owned}(\underline{\sigma})$;

- or because some endpoints form a "cycle of ownership" as described in the previous chapter (page 94); this is the case we want to eliminate.

6.4.4 Leak freedom

Let us study more precisely the formation and evolution of the *unreachable* portion of an open state $\underline{\sigma}$ (the difference between $\mathit{flat}(\underline{\sigma})$ and $\mathit{owned}(\underline{\sigma})$) over the execution of a program, which we mentioned above, as it may be a symptom of a leak.

Complete states We call a state *complete* if the domains of its open endpoint heap and its owned endpoint heap coincide. In other words, the environment controls no endpoint whose peer is owned by the state: the ownership of both endpoints of every channel present in this state can be retrieved by performing enough receives.

Definition 6.12 (Complete state) *An open state* $\underline{\sigma} = (\dot\sigma, \underline{k}) \in \underline{\mathsf{State}}^{\mathsf{wf}}_{\Gamma}$ *such that* $\mathit{owned}(\underline{\sigma}) = (s_o, h_o, \dot{k}_o)$ *is called* complete *if* $\mathit{dom}(\dot{k}_o) = \mathit{dom}(\underline{k})$.

In particular, if $\underline{\sigma} \vDash \mathsf{emp}$ and $\underline{\sigma}$ is complete then $\underline{\sigma} = (\dot{u}, \varnothing)$ and $\mathit{close}(\underline{\sigma}) = u$ (where u is the empty closed state $(\varnothing, \varnothing, \varnothing)$). Complete states have an empty unreachable portion: their flattening coincides with their owned portion.

Lemma 6.15 *If* $\underline{\sigma} = (\dot\sigma, \underline{k})$ *is complete then* $\mathit{flat}(\underline{\sigma}) = \mathit{owned}(\underline{\sigma})$.

Proof Let $\dot\sigma = (s, h, \dot{k})$. The proof is done by induction on the size of $\mathit{dom}(\underline{k}) \setminus \mathit{dom}(\dot{k})$. If it is 0 then $\mathit{owned}(\underline{\sigma}) = \dot\sigma$ and $\underline{k} = \mathit{emp}(\underline{k})$, thus $\mathit{flat}(\underline{\sigma}) = \dot\sigma = (\mathit{owned}(\underline{\sigma}), \mathit{emp}(\underline{k}))$.

If $\mathit{dom}(\underline{k}) \setminus \mathit{dom}(\dot{k}) = E \neq \varnothing$, let $\dot\sigma_{\underline{k}} = \mathit{flat}(\underline{k} \upharpoonright \mathit{dom}(\dot{k}))$ and $\underline{k}' = \mathit{emp}(\underline{k} \upharpoonright \mathit{dom}(\dot{k})) \uplus \underline{k} \upharpoonright (\mathit{dom}(\underline{k}) \setminus \mathit{dom}(\dot{k}))$. We have $\mathit{owned}(\underline{\sigma}) \triangleq \mathit{owned}((s, h, \dot{k}) \bullet \dot\sigma_{\underline{k}}, \underline{k}')$ by definition. Suppose that the lemma holds for all strict subsets of E. In particular, it is true for

$owned((s,h,\dot{k}) \bullet \dot{\sigma}_{\underline{k}}, \underline{k}'))$: either $\dot{\sigma}_{\underline{k}}$ contains some endpoints and the induction hypothesis applies, or it does not and then $dom(\dot{k}) = dom(\underline{k})$ which contradicts $E \neq \varnothing$. Thus, $flat(\underline{\sigma}) = owned((s,h,\dot{k}) \bullet \dot{\sigma}_{\underline{k}}, \underline{k}') = owned(\underline{\sigma})$. □

Leak freedom for complete programs If the footprint context is admissible and the contracts are all leak-free, completeness of states is preserved by reduction. This is not the case if Γ is not admissible or if the contracts are not leak-free. In the first case, cycles of ownership may render unreachable the corresponding endpoints and the footprints contained in their queues. In the second case, closing a channel may leak the footprints of the messages that were left in its buffers and not consumed. This is captured by the following definition and theorem.

Definition 6.13 (Leak-free program) *A program p is* leak free *from a complete state $\underline{\sigma}$ if*

- $p, \underline{\sigma} \not\rightarrow^* \mathbf{LeakError}$ *and*
- *if $p, \underline{\sigma} \rightarrow^* p', \underline{\sigma}'$ then $\underline{\sigma}'$ is complete.*

Theorem 6.3 *If Γ is admissible and $\vDash_\Gamma \{\phi\}\ p\ \{\psi\}$ under leak-free contracts, then p is leak-free from any complete state $\underline{\sigma}$ such that $\underline{\sigma} \vDash \phi$.*

In particular, if Γ is admissible and $\vdash_\Gamma \{\mathsf{emp}\}\ p\ \{\mathsf{emp}\}$ under leak-free contracts, then using the above theorem and Lemma 6.13 one can show that

$$p, u \Rightarrow^* \texttt{skip}, \sigma' \text{ implies } \sigma' = u\ .$$

Note that Theorem 6.3 is true of "complete" programs which execute from complete states. One would need another theorem for "open" programs, for which this is not necessarily the case, for instance by keeping track of the difference between $flat(\underline{\sigma})$ and $owned(\underline{\sigma})$. This is left to future work.

Let us decompose the proof of Theorem 6.3 into two lemmas, corresponding to the two conditions that characterise leak-free programs.

Lemma 6.16 (Leak correspondence) *Let $\underline{\sigma} = (\dot{\sigma}, \underline{k})$ be a ceremonious state well-formed with respect to Γ and containing only leak-free contracts, and p a program decorated only with leak-free contracts such that $p \not\rightarrow^* \mathbf{ProtoError}$. Then $p, (\dot{\sigma}, \underline{k}) \not\rightarrow \mathbf{LeakError}$.*

Proof Assume the hypotheses of the lemma and suppose that $p, (\dot{\sigma}, \underline{k}) \rightarrow \mathbf{LeakError}$ by the leaking reduction rule of `close` for the channel $(\varepsilon, \varepsilon')$. Because of the premises of this reduction rule, $\varepsilon, \varepsilon' \in dom(\dot{k})$ (where $\dot{\sigma} = (s,h,\dot{k})$) and there are a final control state q_f and words α and α' such that $cconf(\underline{\sigma}, (\varepsilon, \varepsilon')) = \{\langle q_f, q_f, \alpha, \alpha' \rangle\}$ and either $\alpha \neq \sqcup$ or $\alpha' \neq \sqcup$. Thus, $\langle q_f, q_f, \alpha, \alpha' \rangle$ is not a stable configuration, which contradicts the fact that $contract(\underline{\sigma}, (\varepsilon, \varepsilon'))$ is leak-free. □

Lemma 6.17 (Completeness preservation) *Let $\underline{\sigma} \in \underline{\mathsf{State}}^{\mathsf{ctt}}_\Gamma$ and suppose that all the contracts appearing in p and $\underline{\sigma}$ are leak-free and that Γ is admissible. If $\underline{\sigma}$ is complete, then for all p' and $\underline{\sigma}'$ such that*

$$p, \underline{\sigma} \rightarrow p', \underline{\sigma}'$$

the open state $\underline{\sigma}'$ is complete.

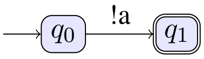
Proof The proof is done by case analysis on the reduction rule. The case of `close` is proved with the help of Lemma 6.16, and the case of `send` by the fact that Γ is admissible.

Let us detail the second case. Let $\underline{\sigma} = ((s, h, \dot{k}), \underline{k})$ and $\underline{\sigma}' = ((s', h', \dot{k}'), \underline{k}') = ((s, h, [\dot{k} \mid \mathit{control}(\varepsilon) {\leftarrow} q']) - \dot{\sigma}_a, [\underline{k} \mid \varepsilon' : \underline{k}(\varepsilon') \cdot (a, v, \dot{\sigma}_a)])$. Assume that $\underline{\sigma}'$ is not complete. The difference between $\mathit{dom}(\dot{k}')$ and $\mathit{dom}(\dot{k})$ can only come from endpoints in $\dot{\sigma}_a$, which means that $\dot{\sigma}_a \npreceq \mathit{owned}(\underline{\sigma}')$. By definition of owned, one can deduce that $\varepsilon' \notin \mathit{dom}(\dot{k}')$, otherwise the contents of its buffer, in particular $\dot{\sigma}_a$, would also appear in $\mathit{owned}(\underline{\sigma}')$. Let $\dot{\sigma}_a = (s_a, h_a, \dot{k}_a)$ and $\dot{\sigma}_{\mathit{cycle}} = (s_c, h_c, \dot{k}_c) = (s'_f, h'_f, \dot{k}'_f) - \mathit{owned}(\underline{\sigma}')$ where $\mathit{flat}(\underline{\sigma}') = (s_f, h_f, \dot{k}_f)$. We have $\dot{\sigma}_a \preceq \dot{\sigma}_{\mathit{cycle}}$.

This shows that either ε' appears in $\mathit{dom}(\dot{k}_a)$ or that there is ε'' such that ε' is owned by the buffer of ε'' and $\varepsilon'' \in \mathit{dom}(\dot{k}_a)$, or more generally that there are $\varepsilon_1, \ldots, \varepsilon_n$ such that $\varepsilon_i \in \mathit{dom}(\dot{k}_c)$ for all i and $\dot{\sigma}_{\mathit{cycle}} = \mathit{flat}(\underline{k}'(\varepsilon_1)) \bullet \cdots \bullet \mathit{flat}(\underline{k}'(\varepsilon_n))$. Thus, $\dot{\sigma}_{\mathit{cycle}} \vDash \circledast_{i \in I} \gamma_{\mathsf{b}_i}(\varepsilon'_i, \varepsilon_i, r_i, v_i)$ where I the set of messages in the buffers of the various ε_i and $\dot{k}_c(\varepsilon_i) = (\varepsilon'_i, -, 3 - r_i, -)$ for all $i \in \{1, \ldots, n\}$. This contradicts the fact that Γ is admissible by Definition 5.1. □

6.4.5 Boundedness of communications and deadlock-freedom

Lemmas 6.11 and 6.16 connect the fault and leak-freedom properties of contracts to equivalent properties on programs. One could show, similarly to the proof of these lemmas, that a program with n-bounded contracts only needs communication buffers of size n.

Unfortunately, a program with deadlock-free contracts is not necessarily deadlock-free, even when the program only manipulates one channel. For instance, using the deadlock-free contract 

one can prove the following deadlocking program:

```
(e,f)=open(C)
receive(a,f);
send(a,e);
close(e,f);
```

This is because the logical rules do not take account causality between different endpoints, but only causality within the same endpoint. Similarly, contracts fail to account for interactions between different channels, which can also cause deadlocks, as in the program below, whose channels abide by the same deadlock-free contract as above.

```
{ receive(a,f1); }      { receive(a,f2); }
{ send(a,e2);    }  ∥   { send(a,e1);    }
```

Chapter 7

Heap-Hop

This chapter describes the tool that has been developed during the course of the thesis, named Heap-Hop. It takes programs annotated with logical formulas and contracts as input, and outputs either “`Valid.`” if all the functions have been successfully proved to comply with their specifications and the contracts are leak and fault free, or the part of the proof that has failed. A failed proof means that the specifications given by the user do not hold, hence one cannot know if the program has a genuine bug or if the specifications could be refined to a point where they are true. A proved program on the other hand is guaranteed to have all the good properties corresponding to the ones described in the last section of the previous chapter.

We first present Heap-Hop’s general modus operandi and then its public distribution, which includes the case studies Heap-Hop has been applied to.

7.1 Input

Heap-Hop takes as input programs annotated with pre and post-conditions for functions, loop invariants, message footprints and contracts. More precisely, Heap-Hop’s inputs are plain text files that can contain declarations for message identifiers, contracts and functions. The annotations are formulas of a fragment of our logic placed between brackets and written in plain text.

The contents of such a file containing the declarations necessary to prove the cell passing example of Figures 1.6 and 5.2 is reproduced Figure 7.1. This file is part of the Heap-Hop distribution (see Section 7.4) as `examples/cell.hop`. The notations will be explained shortly.

7.1.1 Programming language

The programming language used by Heap-Hop is a more practical version of the HipMP language named Hop. Its syntax is shown in Figure 7.2.

Hop supports records and field accesses, “if-then-else,” “while” and “switch receive” constructs, function calls and messages with multiple parameters. Moreover, although this

```
message cell [val|-> ]

contract Cell {
  initial state start { !cell -> end; }
  final state end { }
}

put_get() [emp] {
  local e,f,x;

  x = new();
  (e,f) = open(Cell);

  put(e,x) || get(f);

  close(e,f);
} [emp]

put(e,x) [e|-> Cell{start} * x|-> ] {
  send(cell,e,x);
} [e|->Cell{start}]

get(f) [f|-> ~Cell{start},pr:_ee] {
  local y;

  y = receive(cell,f);
  dispose(y);
} [f|-> ~Cell{start},pr:_ee]
```

Figure 7.1: Heap-Hop input file corresponding to the cell transfer program.

does not appear in the syntax, receive commands can be used as syntactic sugar for single-branch switch receives. The parallel composition is syntactically restricted to a parallel call of two functions. Because function declarations are annotated with pre and post-conditions, this syntactic restriction allows Heap-Hop to know what the splitting of resources between the two sides of a given parallel composition will be, instead of having to enumerate and try them all.

At the beginning of the file are message tags declarations. When declaring a new message tag with the `message a [`ϕ`]` statement, the user provides the number of parameters that messages with this tag will have, called the arity of the tag or the arity of the message. The arity of tag a is written $ar(a)$, and the syntax of a message declaration that also specifies an arity n is

message a(n) [ϕ]

A tag z with an arity of 0 has no parameters, is sent using `send(z,`E`)` and is received in branches of the form `receive(z,`E`):` p. Heap-Hop checks that arities are used consistently by the program. If some arities are not explicitly declared in the file, Heap-Hop infers

E	::=		expressions
		$\mathtt{x}$	variable
		$\mid v$	value
		$\mid E + E \mid E \text{ xor } E$	arithmetic operations
B	::=	$E = E \mid E \neq E$	boolean expressions
c	::=		commands
		$\texttt{x = } E$	variable assignment
		$\mid \texttt{x = new()}$	memory allocation
		$\mid \texttt{x = } E \rightarrow \texttt{t}$	field lookup
		$\mid E \rightarrow \texttt{t = } E$	field mutation
		$\mid \texttt{dispose(}E\texttt{)}$	memory deallocation
		$\mid \texttt{(e,f) = open(C)}$	channel creation
		$\mid \texttt{close(}E\texttt{,}E\texttt{)}$	channel destruction
		$\mid \texttt{send(}a\texttt{,}E\texttt{,}E_1\texttt{,}\ldots\texttt{,}E_{ar(a)}\texttt{)}$	send
r	::=	$(\mathtt{x}_1,\ldots,\mathtt{x}_{ar(a)}) = \texttt{receive(}a\texttt{,}E\texttt{):}\ p$	receive branch
p	::=		programs
		c	atomic command
		$\mid \texttt{switch receive } \{r_1\ \ldots\ r_n\}$	switch receive
		$\mid p;\ p$	sequential composition
		$\mid \texttt{if } (B)\ p \texttt{ else p}$	conditional
		$\mid \texttt{while } (B) \texttt{ p}$	loop
		$\mid f(\vec{\mathtt{x}}; \vec{E})$	function call
		$\mid f(\vec{\mathtt{x}}; \vec{E}) \parallel f(\vec{\mathtt{x}}; \vec{E})$	parallel composition

Figure 7.2: Syntax of the Hop programming language.

them from the program.

7.1.2 Contracts

The protocols of a program are described using contracts. In the future, Heap-Hop may support more general dialogue systems to specify protocols. Contracts are required to have at least one initial state, and need not be deterministic, positional or synchronising. However, these three properties are checked by Heap-Hop, which will issue appropriate warnings if they are not satisfied.

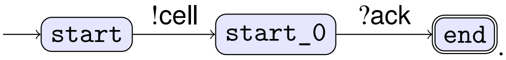
From a syntactic point of view, each state of a contract is introduced separately with the `state` keyword, which may be preceded by `initial` or `final`. A state declaration

E	$::=$	logical expressions
	`_x`	logical variable
	$\mid$ `x`	program variable
	$\mid n$	integer
	$\mid E + E \mid E \;\mathsf{xor}\; E$	arithmetic operations
	$\mid \{q_1, \ldots, q_n\}$	set of control states
ρ	$::= \mathtt{t}_1 : E_1, \ldots, \mathtt{t}_n : E_n$	record expressions
ϕ	$::=$	pure formulas
	$E = E$	boolean expression
	$\mid \mathsf{true}$	true
	$\mid \mathsf{false}$	false
	$\mid \mathsf{emp}$	empty heap
	$\mid E \mapsto \rho$	allocated location
	$\mid \mathsf{list}(E_1, E_2)$	linked list segment
	$\mid \mathsf{tree}(E)$	binary tree
	$\mid \phi * \phi$	separating conjunction
	$\mid \mathsf{if}\; B \;\mathsf{then}\; \phi \;\mathsf{else}\; \psi$	precise disjunction

Figure 7.3: Syntax of Heap-Hop assertions.

consists of a set of transitions starting from this state. Similarly to Singularity, one can declare a sequence of transitions in one go, and Heap-Hop will create appropriately named implicit states. For instance, the contract declared with

```
contract CellAck {
  initial state start { !cell -> ?ack -> end; }
  final state end { }
}
```

will be understood by Heap-Hop as .

7.1.3 Logical fragment

The formulas manipulated by Heap-Hop are all within a decidable fragment of separation logic. This fragment is described Figure 7.3. It is essentially the one used by Smallfoot [BCO05a] extended with assertions for endpoints.

This logic differs from the one defined in Chapter 5 on several points. It is a restricted version of it in most respects but also goes beyond it in some others. Below is a list of the discrepancies between the theory and the practise of our logic at the assertion level:

- non-deterministic contracts are supported, hence the logic keeps track of sets of control states instead of just one control state, as mentioned in Section 5.3.1;

- the tool logic does not treat variables as resources, hence the absence of own statements;
- the syntax of the logic is restricted so that implication between two formulas of this fragment is decidable (see Section 7.2), which is not the case for the logic of Chapter 5. Moreover, disjunctions are restricted to case analyses so that formulas of this fragment are always precise;
- the contents of cells and endpoints is addressed through a record notation;
- endpoints do not use a separate predicate to distinguish them from regular cells. Instead, a special read-only field `ep` (set to 1 for endpoints and 0 for regular cells) is used by Heap-Hop to distinguish between the two possible types. Endpoint-related information (the peer, contract, role and set of control states of an endpoint) is recorded in the fields `pr`, `cid`, `rl` and `st` corresponding respectively to the peer, contract identifier, role and control state of an endpoint. These fields cannot be modified directly by programs, only via the communication primitives of Hop.

7.2 Internals

7.2.1 Verification condition

The first phase of the processing of functions consists in chopping each of them, together with their specifications, into several *verification conditions*. Following the terminology of Berdine *et al.* in their paper describing Smallfoot [BCO05a], a verification condition is a triple $[\phi]\ si\ [\psi]$ where si is a *symbolic instruction*, that is a loop-free sequential program described by the following grammar:

$$si ::= c' \mid [\phi]\ \mathsf{jsr}_{\vec{\mathtt{x}}}\ [\psi] \mid si;\ si \mid \mathsf{if}\ B\ \mathsf{then}\ si\ \mathsf{else}\ si$$

where c' is the following set of atomic commands:

$$\begin{aligned} c' ::= {} & \mathit{assign}(\mathtt{x}, E) \mid \mathit{new}(\mathtt{x}) \mid \mathit{lookup}(\mathtt{x}, E, \mathtt{t}) \mid \mathit{mutate}(E, \mathtt{t}, E) \mid \mathit{free}(E) \\ & \mid \mathit{open}(\mathtt{e}, \mathtt{f}, \mathtt{C}) \mid \mathit{close}(E, E) \mid \mathit{send}(a, E, \mathcal{L}) \\ & \mid \mathit{extchoice}(\mathcal{LM}) \mid \mathit{receive}(\ell, a, E) \end{aligned}$$

In the above grammar, ℓ is used to denote a list of variables, $\mathcal{L}$ a list of expressions, and $\mathcal{LM}$ a list of pairs $(E, \mathcal{M})$ where E is an expression and $\mathcal{M}$ is itself a list of message identifiers (hence if Expr is the set of expressions $\mathcal{LM} \in (\mathsf{Expr} \times (\mathsf{MsgId}^*))^*$). Because of the initial parsing phase all commands are now presented in terms of constructors instead of their syntax-friendly versions. For instance, `x = new()` is translated into $\mathit{new}(\mathtt{x})$.

Function calls and parallel compositions are translated into jsr instructions, inherited from Smallfoot and described as such by their authors [BCO05a]:

> Semantically, $[\phi]\ \mathsf{jsr}_{\vec{\mathtt{x}}}\ [\psi]$ is a “generic command” [...]. It is the greatest relation satisfying the pre- and post-condition, and subject to the constraint that only the variables in $\vec{\mathtt{x}}$ are modified.

Switch receives are translated into guarded non-deterministic choices between each of their branches. The guard is an *extchoice* command that is used during symbolic execution to check that each endpoint is ready to receive all the messages specified by the contract from the endpoints' current states.

To see how the translation into verification conditions operates, and in particular the role of jsr instructions, let us consider the following example taken from the Heap-Hop distribution (`examples/send_list.hop`) and corresponding to the receiving end of the list-transfer program presented Figure 1.9 (page 24):

```
get(f) [f|->~C{transfer}] {
  local x, e;

  e = NULL;
  while(e == NULL) [if e==NULL
                    then f|->~C{transfer}
                    else e|->C{end} * f|->~C{end},pr:e] {
    switch receive {
    x = receive(cell,f): { dispose(x); send(ack,f); }
    e = receive(fin,f): {}
    }
  }
  close(e,f);
} [emp]
```

This program is transformed into the following two verification conditions (which can be observed by passing the `-verbose` option to Heap-Hop):

```
[f ↦ ~C{transfer}]
 assign(e, 0);
 [if (0 = e) then f ↦ ~C{transfer} else f ↦ ~C{end} ∗ e ↦ C{end}]
 jsr_{e,x}
 [(if (0 = e) then f ↦ ~C{transfer} else f ↦ ~C{end} ∗ e ↦ C{end}) ∗ 0 ≠ e]
 close(e, f);
[emp]
```

and

```
[(if (0 = e) then f ↦ ~C{transfer} else f ↦ ~C{end} ∗ e ↦ C{end}) ∗ 0 = e]
 extchoice([(f, [cell, fin])]);
 if nondet0 ≠ 42 then
   receive([x], f, cell);
   dispose(x);
   send(f, ack, []);
 else receive([e], f, fin);
[if (0 = e) then f ↦ ~C{transfer} else f ↦ ~C{end} ∗ e ↦ C{end}]
```

In this example, the while loop gives rise to two verification conditions: one in which it is bypassed and replaced by a generic jsr instruction that sums up its effect, and one that checks that the body of the loop preserves its invariant. The switch receive construct is replaced by a conditional that simulates a non-deterministic choice between its branches (a fresh variable `nondet0` is created to form the test of the conditional). It is guarded by an *extchoice* command as described above, which in this case states that `f` is ready to receive either cell or fin.

We will not give a formal description of the translation, as it bears no other important details. It should be straightforward to extrapolate from the explanation above. One can show that the verification conditions associated to an annotated function are provable Hoare triples if and only if the specification of the function is also provable.

Because variables are treated as in conventional Hoare logic, Heap-Hop (like Smallfoot) has to ensure the absence of race conditions on variables without being able to rely on proof rules to do so. This is achieved by tracking which variable is read or written by each function and ensuring that no pair of functions that both write to the same global variable are ever put in parallel. This syntactic check is performed during the verification condition generation phase.

7.2.2 Symbolic execution

The second step of the automatic verification consist in symbolically executing the body of the verification conditions. Starting from the first command, the precondition is transformed according to systematic rules that mimic the effect the command has on the state at the formula level. The new formula is then used as the precondition of the remaining program. Thanks to the fact that verification condition bodies are sequential and loop-free, this process can be repeated until no command remains. The resulting formula is then checked against the post-condition in the last step of the verification (described below). More precisely, a verification condition $[\phi]\ p\ [\psi]$ is transformed into several entailment checks $\phi' \Rightarrow \psi$ that hold only if the verification condition is a provable Hoare triple.

Symbolic execution follows a finite set of rules that may be seen as rewrite rules between verification conditions, one for each possible command at the head of the current program, and additional ones to rewrite the current formula into a more canonical form in-between the application of rules for commands. For instance, the rule for memory look-up can be presented as the following inference rule:

$$\frac{\{(\phi * E\,\texttt{|->}\rho)[\texttt{x}\leftarrow\texttt{x}'] * \texttt{x} = E_2[\texttt{x}\leftarrow\texttt{x}']\}\ p\ \{\phi'\}}{\{\phi * E\,\texttt{|->f}:E',\rho\}\ \texttt{x = }E\texttt{->f}\ \{\phi'\}}\ \texttt{x}'\text{fresh}$$

The interested reader may refer to the article of Berdine *et al.* [BCO05b] for details on the rest of the rules (in a setting without endpoints) and how they are applied. We will soon show how symbolic execution is performed in our example, but let us first introduce the *frame inference* problem.

Frame inference problem Along the way other entailments have to be checked. In fact, each jsr call requires an instance of the *frame inference* problem to be solved, as described below.

Definition 7.1 (Frame inference problem) *The frame inference problem consists in finding, given two formulas ϕ and ϕ', a third formula ψ such that the following implication is valid:*

$$\phi \Rightarrow (\phi' * \psi)$$

The name "frame inference" comes from its relation with the Frame rule: if one knows that $\{\phi_1\}\ p\ \{\phi_2\}$ is provable and is given ϕ as a precondition, one has to find a "frame" ψ such that $\phi \Rightarrow (\phi_1 * \psi)$ in order to apply the frame rule in the following way (after a weakening step):

$$\frac{\phi \Rightarrow (\phi_1 * \psi) \qquad \{\phi_1\}\ p\ \{\phi_2\}}{\{\phi\}\ p\ \{\phi_2 * \psi\}}\ \textsc{Weakening} + \textsc{Frame}$$

When the symbolic execution reaches a $[\phi_1]\ \mathsf{jsr}_\ell\ [\phi_2]$ instruction with ϕ as current formula, it tries and solve the following instance of the frame inference problem: is there a formula ψ such that

$$\phi \Rightarrow (\phi_1 * \psi)\ ?$$

If there is, the symbolic execution resumes with the rest of the program and $\phi_2 * \psi$ as current formula.

Berdine *et al.* explain how the frame inference problem is solved in Smallfoot in their paper on symbolic execution [BCO05b]. Adding endpoints to the heap model do not alter their algorithm in any but minor details. As the technique relies mainly on the way entailments are checked, we delay the discussion to after the exposition of the entailment-checking algorithm, in Section 7.2.3. For now, let us assume that such an algorithm is given.

Symbolic execution on an example Let us provide an informal description of the symbolic execution procedure by applying it to the second verification condition obtained earlier. The first command is $\mathit{extchoice}([(\mathtt{f}, [\text{cell}, \text{fin}])])$. To expose the contract associated to `f`, the precondition is first rewritten by substituting `e` by 0 and removing the corresponding equality, and then by performing a case-split on the value of the test $0 = 0$. As one of the formulas obtained presents a contradiction ($0 \neq 0$) it is discarded (more precisely, verification conditions of the form $[\mathsf{false}]\ p\ [\psi]$ are always true). The remaining verification condition has

$$\mathtt{f} \mapsto \mathtt{\sim C}\{\text{transfer}\}$$

as precondition. As the contract associated to `f` in the precondition gives only cell and fin as potential messages in the receive buffer, which matches the list of messages for which `f` is ready, the check is successful and the precondition is passed as-is to the conditional that follows. Each branch of the if-then-else construct is analysed separately. In the first branch, the receive transforms its precondition by updating the control state of `f` to the implicit control state transfer_0 and by adding to it the invariant of the message cell (correctly instantiated). The result is the formula

$$\mathtt{f} \mapsto \mathtt{\sim C}\{\text{transfer\_0}\} * \mathtt{x} \mapsto - \,.$$

The *dispose*(x) command that follows looks for a cell pointed to by x in its precondition and simply removes it, which gives

$$\mathtt{f} \mapsto \mathtt{\sim C}\{\text{transfer\_0}\}\ .$$

Finally, sending ack puts f back in the transfer control state and then tries and find a subformula corresponding to the footprint of ack. In this case, the footprint is emp so the whole process is trivial; let us detail it anyway. After the control state of the corresponding endpoint has been updated, the *send* command is replaced by a jsr call that corresponds to consuming the message's footprint, instantiated by the right parameters (something that could not have been done before this phase, as one needs to retrieve, for instance, the peer of the sending endpoint, which can only be done by looking at the formula at that point of the symbolic execution). More precisely, a call $\mathit{send}(a, E, E_1 :: \cdots :: [E_n])$ starting from formula ϕ is replaced with

$$[\gamma_a(E, E', r, E_1, \ldots, E_n)]\ \mathsf{jsr}_\varnothing\ [\mathsf{emp}]\ .$$

where $\phi \Rightarrow (E \mapsto \mathrm{pr} : E', \mathrm{rl} : r * \mathsf{true})$. In the case at hand, $\mathit{send}(\mathsf{ack}, \mathtt{f}, [])$ is replaced with $[\mathsf{emp}]\ \mathsf{jsr}_\varnothing\ [\mathsf{emp}]$. As described earlier, this call results in the entire formula being framed away (because $\phi \Rightarrow (\mathsf{emp} * \phi)$ for any ϕ), and the symbolic execution reaches the end of the branch with this final current formula:

$$\mathtt{f} \mapsto \mathtt{\sim C}\{\text{transfer}\}\ .$$

Similarly, the second branch of the conditional produces the formula

$$\mathtt{f} \mapsto \mathtt{\sim C}\{\text{end}\} * \mathtt{e} \mapsto \mathtt{C}\{\text{end}\}\ .$$

A formal description of the symbolic execution algorithm used by Smallfoot is given in the paper "Symbolic execution with separation logic" by Berdine *et al.* [BCO05b]. The procedure used by Heap-Hop extends it to handle communication commands. For instance, the effect of a *send* can be described by the following inference rule:

$$\frac{\begin{array}{l}\{\phi * E \mapsto \mathtt{cs}{:}Q', \mathtt{ctt}{:}C, \mathtt{pr}{:}E', \mathtt{rl}{:}r, \rho\}\\ [\gamma_m(E, E', r, E_1 :: \cdots :: [E_n])]\ \mathsf{jsr}_\varnothing\ [\mathsf{emp}];\ p\\ \{\phi'\}\end{array}}{\{\phi * E \mapsto \mathtt{cs}{:}Q, \mathtt{ctt}{:}C, \mathtt{pr}{:}E', \mathtt{rl}{:}r, \rho\}\ \mathit{send}(\mathtt{a}, E, E_1 :: \cdots :: [E_n]);\ p\ \{\phi'\}}\ \dagger$$

$$^\dagger \mathit{succ}^\wp(C, r, Q, !a) = Q'$$

Similarly, the rule for close takes this form:

$$\frac{\{\phi\}\ p\ \{\phi'\}}{\begin{array}{l}\{\phi * E \mapsto \mathtt{cs}{:}Q, \mathtt{ctt}{:}C, \mathtt{pr}{:}E', \mathtt{r}{:}1, \rho * E' \mapsto \mathtt{cs}{:}Q', \mathtt{ctt}{:}C, \mathtt{pr}{:}E, \mathtt{r}{:}2, \rho\}\\ \mathit{close}(E, E');\ p\\ \{\phi'\}\end{array}}\ \dagger$$

$$^\dagger Q \cap Q' \cap \mathit{finals}(C) \neq \varnothing$$

7.2.3 Entailment checking and frame inference

During the symbolic execution phase, one has to prove several entailments $\phi \Rightarrow \psi$ and solve frame inference problems. The decision procedure used by Heap-Hop for formulas follows straightforwardly from the one of Smallfoot, also described in Berdine *et al.* paper [BCO05b].

Intuitively, in the case of an entailment $\phi \Rightarrow \psi$, every predicate appearing in the right-hand side formula has to be matched by the left-hand side formula. If this is the case, it is removed from both sides and the entailment checking procedure continues with what is left on both sides. The goal is to reach an entailment of the form $(\phi' * \mathsf{emp}) \Rightarrow \mathsf{emp}$ where ϕ' is made solely of a conjunction of equalities between expressions (what Berdine *et al.* call *pure* formulas). In this case, the entailment is true. In fact, the decision procedure used by Smallfoot and Heap-Hop is complete, as shown by Berdine *et al.* [BCO05b] with little adaptation, hence the entailment is false if it cannot be reduced to one of the form $(\phi' * \mathsf{emp}) \Rightarrow \mathsf{emp}$ where ϕ' is pure.

Frame inference as performed by Smallfoot and Heap-Hop is closely connected to the procedure we have just described. If we have to find a frame ψ such that $\phi_1 \Rightarrow (\phi_2 * \psi)$, we try to prove $\phi_1 \Rightarrow \phi_2$ using the procedure above. If it succeeds, the frame emp suffices. If it fails however, it will be because of one of the following two situations: either a predicate of the right-hand side cannot be matched by a portion of the left-hand side formula, even after normalisation; in this case, Heap-Hop gives up on finding a frame. The second case is that the procedure reaches an implication $\phi' \Rightarrow \mathsf{emp}$ where ϕ' is impure ($\phi' \not\Rightarrow \mathsf{emp}$). In this case, one can show that ϕ' is an admissible frame solution: $\phi_1 \Rightarrow (\phi_2 * \phi')$.

7.2.4 Contract verification

Heap-Hop checks sufficient syntactic conditions to ensure that all contracts are fault-free and leak-free. The conditions are the one dictated by Corollary 3.1 (page 60: contracts must be deterministic, positional and synchronising). For this purpose, Heap-Hop stores each contract as an identifier, a set of initial and final states, and a list of states. Each state contains the list of outgoing transitions from this state.

To check determinism, Heap-Hop searches each of these lists of transitions for duplicate actions. The fact that a contract is positional is as easy to establish: Heap-Hop makes sure that each list of outgoing transitions only contains actions in one direction (either only sends or only receives). Finally, for each final state of the contract, Heap-Hop checks if there is a cycle consisting of either only sending or only receiving actions from that state (by a naive reachability analysis from that state). If this is not the case, the contract is synchronising because all its final states are.

As we have discussed in Chapter 3, and more precisely in Corollary 3.1, contracts that satisfy all three of these conditions are fault and leak-free. As we have seen in Theorems 6.2 and 6.3 of the previous chapter, programs that can be successfully proved using such contracts are themselves reception error free and will not close channels with undelivered messages in them. The conformance of the program to the declared contracts is checked at symbolic execution time using the logical rules described above and derived from those of

Chapter 5. Heap-Hop uses the criterion of Lemma 5.3 to check that the footprint context is admissible, hence that leaks are completely avoided if the contracts are leak-free.

7.3 Output

7.3.1 Properties of the contracts

If a contract fails one of the checks mentioned above, a warning is issued to the user appropriately: if a contract is either non-deterministic or non-positional, then Heap-Hop warns that it cannot prove that programs are either leak-free or fault-free; if a contract is not synchronising, Heap-Hop warns that programs cannot be proved leak free.

The incriminated contract and control states are indicated to the user as part of the warning. All warnings and errors in Heap-Hop are output using a standard format to indicate the line number where the abnormality is. For instance, in the case of a non-synchronising contract, Heap-Hop's output would be of the following form:

```
File "non-synch-contract.hop", line 13, characters 0-125:
WARNING: contract C is not synchronizing:
         there is a send cycle starting at state start
WARNING: program cannot be guaranteed leak free because of contract C
```

This formatting is recognised by the `emacs` text editor for instance, which allows the user to click on the warning to be directed to the relevant file and line.

Moreover, contracts are also output as `dot` files,[1] so that the user can have a graphical representation of them.

7.3.2 Hoare-triple verification

Once the contracts have been checked, Heap-Hop tries to prove that the specifications of the functions are true, using the process described in Section 7.2. If it fails to prove that a certain specification or loop invariant holds, it can be for one of two reasons, which both result in an error pointing to the offensive line of code being issued:

- either the symbolic execution is stuck at a particular command because the current precondition does not match the one of the command, which means that the command cannot safely execute (that is, there is a possible memory fault or race at that program point);

- or the result of the symbolic execution fails to entail the required post-condition, which means that the specification or loop invariant given by the user does not hold.

In either case, the error is reported to the user, with the current formula obtained by the symbolic execution and the location in the program where Heap-Hop was stuck. If *extchoice* detects a missing branch in a switch receive, a mere warning is issued given the incompleteness of relying on contracts to detect communication errors. In practise however, only one

[1] `http://www.graphviz.org/Documentation.php`

of our case studies exposes such a warning despite being correct (one of our solutions to the load balancing tree disposal problem exposed in Section 7.4.2 below, which can be found in the file `examples/spawning_threads_tree_disposal.hop`).

7.4 Distribution

7.4.1 Public distribution

License Heap-Hop is released under the "Q Public License version 1.0" which can be found on Trolltech's website.[2] In particular, its source code is publicly available and may be changed by anyone provided that the changes are made distinct from the original source and are released publicly and under the same license. Heap-Hop inherits this license from Smallfoot.

Contents of the distribution Heap-Hop's release contains

- the source code of the tool, written in OCaml, with a "Makefile" to compile it automatically;
- a battery of examples;
- a succinct user's manual;
- an `emacs` major mode to help editing examples and running Heap-Hop on them.

The `emacs` major mode is an extension to the `emacs` text editor that includes syntax highlighting for the Hop language.

Although still under active development, Heap-Hop is considered as stable by its authors and a public official version numbered `1.0` has been released. Unstable versions with more features are also available publicly for testing purposes.

Website Heap-Hop's official website, at the time of this writing, is located at `http://www.lsv.ens-cachan.fr/Software/heaph-hop/`. This contains the tarball distribution, an online documentation, a summary of the case studies and other resources such as scientific papers associated to Heap-Hop.

Heap-Hop is also hosted on the bitbucket website [3] to benefit from a dedicated infrastructure that provides a mailing-list, a bug tracker, and an online Mercurial (a free distributed source control management tool [4]) repository which allows anyone to download the latest version and submit patches to the code.

[2] `http://doc.trolltech.com/3.0/license.html`

[3] `http://bitbucket.org/jvillard/heap-hop/`

[4] `http://mercurial.selenic.com/`

Figures Let us now present some executive-compliant key figures about Heap-Hop's code base. All figures include comment lines. Heap-Hop, in its current form, represents 5018 lines of OCaml code and ships with 1273 lines of original examples. The Smallfoot code base, while retaining its essence, has undergone much changes: omitting examples, 925 lines were removed and 2316 lines were added (including lines that were modified).

7.4.2 Case studies

Examples presented so far Most of the examples presented in this manuscript (including for instance the cell-by-cell list sending program and the modelling of an ATM) are included in the `examples/` subdirectory of the Heap-Hop distribution. Let us present another example that ships with Heap-Hop that introduces what we believe to be an interesting problem: the load balancing problem for binary trees. This problem is only partially solved by Heap-Hop.

Load balancing What we call the load-balancing problem for two threads is this: given a binary tree whose nodes represent some tasks, which we assume to be of equal cost in terms of computing power, one tries to distribute these tasks to two threads running in parallel, such that both threads end up treating an approximately equal amount of tasks. The tree is not supposed to be balanced and its shape is not known in advance, so the algorithm has to distribute the tasks while exploring the tree. More precisely, the threads have to collaborate and dynamically balance the load of their tasks.

The solution we propose to this hypothetical problem follows. We suppose that the treatment of each task consists solely of disposing of the corresponding node, but in theory this could be an arbitrarily complex operation. One of the threads, called the "left thread," will dispose of the left children of all of the internal nodes of the tree, and the other thread, the "right thread," will dispose of all the right children. Both threads hold one endpoint of a shared channel.

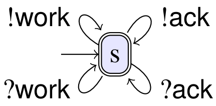
Initially, if the tree is not empty, the left thread starts with the left subtree (one step down). If this subtree is not empty, it disposes of its root, sends its right subtree (two steps down) over the channel, and proceeds with the left subtree. The right thread behaves symmetrically. If the tree it is working on becomes empty, the thread sends an acknowledgement and waits for a message. It may either receive a new tree to be disposed or an acknowledgement. The thread maintains a counter holding the number of tasks that it has sent and that have not yet been processed and acknowledged by the other thread. When this counter reaches zero, the other thread is done, hence the communication can be stopped.

The code for the left thread is presented in Figure 7.4. The code for the right thread is symmetric (it sends the left subtree and keeps the right one when descending into the current tree). Its annotations have been checked by Heap-Hop, as has been the full program (included in the Heap-Hop distribution[5]). The contract `C` that the program is proved to follow is the one that allows any sequence of messages work and ack on the channel:

[5] Since Heap-Hop does not yet support arithmetic, the code in the distribution includes additional variables and code to handle the values of the counter `i` by predicate abstraction.

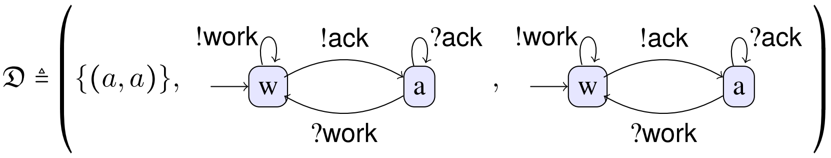
```
message work [tree(val)]
message ack  [emp]

dispose_left(e) [e|->C{s},pr:_f] {
  local l,r,t,i,is_waiting;

  i=1;
  is_waiting = 0;
  t = 0;
  while (i != 0)
  [e|->C{s},pr:_f * (if (i == 0)
                     then emp
                     else (if (is_waiting == 0)
                           then tree(t)
                           else emp))] {
    if (is_waiting == 0) {
      while (t != NULL) [e|->C{s},pr:_f * tree(t)] {
        l = t->l;
        r = t->r;
        send(work,e,r);
        i = i +1;
        dispose(t);
        t = l;
      }
      send(ack,e);
      is_waiting = 1;
    } else switch receive {
        t = receive(work,e) : { is_waiting = 0; }
        receive(ack,e) :      { i = i -1; }
      }
  }
} [e|->C{s},pr:_f]
```

Figure 7.4: Left thread for the load-balancing binary tree disposal.



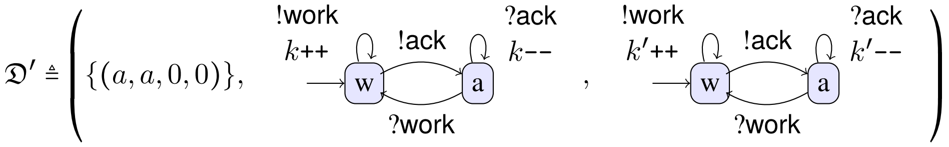
Interesting questions about this piece of code include: whether it terminates, whether it deadlocks, and whether it is leak-free. However, Heap-Hop cannot establish any of these properties. It only manages to prove that the program is safe with respect to memory management.

Although Heap-Hop cannot prove termination, it could prove leak-freedom if contracts were richer. For instance, one could use arbitrary dialogue machines instead of contracts for specifying the communication protocols. In this case, the protocol followed by the program can be (much more accurately) described by the following system $\mathfrak{D}$:



Note that both endpoint of the channel follow the same protocol, and not dual ones. In particular, the communications are not half-duplex. Although this dialogue system is not leak-free, it is deadlock-free. Using a similar method as with contracts, one could prove that the load-balancing program obeys this dialogue system, and that as a result the program will not get stuck waiting for a communication.

Yet, the program is also leak-free, and moving to general dialogue machines alone does not help in proving this. To achieve this, one would have to augment dialogue systems with counters. Informally, one would specify the communications using the following extended dialogue system $\mathfrak{D}'$ where transitions can be guarded by testing a counter k or k', or incrementing it (with k++) or decrementing it (with k−−):



The final state now also specifies that $k = k' = 0$. One can extend the definitions of Chapter 3 and show that $\mathfrak{D}'$ is indeed leak-free: whenever the final configuration $(a, a, 0, 0)$ is reached, the buffers are empty. This would show that if the program terminates then the whole tree has been properly disposed of. If we replace disposal with some other operation on nodes, this shows that the load-balancing program treats all the nodes. The proof system would not ensure that it terminates, or that the tasks are balanced between the two threads.

Conclusion

Contributions Let us begin this conclusion by a summary of the contributions contained in this manuscript. Firstly, a formal semantics has been given to a message-passing programming language that is heavily inspired by the Sing♯ language of the Singularity project. This formalisation is an attempt both at giving a sound bedding to the ideas introduced in Singularity and at studying copyless message passing in all generality. This effort is pursued on the protocol specifications of Singularity, the contracts, for which we also provide a formal semantics, in terms of the existing model of communicating automata. Various decidability questions are studied, from which one may draw the conclusion that the constrained syntactic form of contracts gives them no theoretical advantage over the more general class of half-duplex dialogue systems. Nevertheless, their practical usefulness, as demonstrated by the Singularity project, promotes them as the language of protocol specifications for our own logic.

Indeed, we define an original extension of separation logic to copyless message passing which validates programs with respect to their contracts. To prove its soundness, we define a second semantics for our programming language where the logical reasoning attached to copyless message passing is given an operational meaning. This second semantics is related to the first in a way that is independent of the properties of the program, and lets us explore the links between a program and the contracts that it abides by: some of the crucial properties of contracts, good or bad, rub off on programs. Thus, the verification of a message-passing program amounts to its provability in our proof system as much as it amounts to verifying its contracts.

Testing and validating our approach was helped by the Heap-Hop tool, developed during this thesis, that implements our technique to automatically verify annotated message-passing programs.

Richer protocol specifications Starting from this work, several directions are amenable to further research in verifying message-passing programs. First of all, contracts are but one means of specifying protocols, and several other options could have been considered. As discussed at the end of the last chapter, one possibility would be to consider more general dialogue machines extended with counters. We believe that our logic can easily cover such an extension, as do the semantic models we used. What would become problematic however, and where we believe interesting theoretical challenges lie, is the verification of such protocols. Indeed, our syntactic restrictions would no longer guarantee the desired properties on protocols. Furthermore, half-duplex machines with counters do not have many good decidability properties, as counter machines with at least two counters can simulate Turing machines. New techniques would thus have to be developed.

Another possible extension to the expressivity of the verification is the addition of *parameters* to footprints, in the following sense. Suppose that one wishes to transfer a linked list over a channel but that, instead of disposing of its nodes one by one as in the example shown previously, one wishes to reconstruct the list cell by cell. As it is, we could not find a proof of such a program with our proof system because the footprints of messages have no means of mentioning the value that will carry the next message, or the value that was carried by the previous one. One possible solution for this example would be to add a parameter X to the footprint of the cell message, which becomes allowed to mention both X and the next value X' of X: $\gamma_{\mathsf{cell}} \triangleq \mathsf{emp}_s \land \mathsf{Val} \mapsto X' \land \mathsf{Val} = X$. The contract remains unchanged, but if there are other messages between two cell messages, they may carry the previous value of X onward. For instance, if each cell message is followed by an ack message, the footprint of the acknowledgement message could be $\gamma_{\mathsf{ack}} \triangleq \mathsf{emp} \land X' = X$. The proof system would be in charge of handling these parameters.

One could also go in another direction and try to find a more general model of concurrency or of message passing that would encompass our own. It is a bit disappointing indeed that abstract separation logic was unable to give a natural meaning to message-passing constructs, and one may wish for such an abstract framework that would apply to more models of concurrency.

Finally, instead of limiting channels and protocols to dialogues, one could imagine *multiparty conversations* occurring on channels and specified by *multiparty contracts*. These contracts could take the form of arbitrary communicating systems with any number of participants for instance. One could follow what has been accomplished by the session types community in this direction [HYC08].

Progress Studying the relationship between progress properties of a program and the progress properties that can be established on the protocols it follows could allow one to prove that a program will never get stuck on a communication. To this end, one may be able to borrow some ideas from the Chalice tool [LMS10] or from multiparty session types [BCD+08], both of which can prove progress for programs using multiple communication channels.

Full-featured programming language Besides augmenting the power of logical specifications, the programming language itself may be extended, which would certainly prompt the need for extensions in our logic as well. As far as communications are concerned, several extensions are conceivable to try and embrace the myriad of existing message-passing paradigms.[6] Let us just mention a few of them. In our current setting, it is possible to spec-

[6] The MPI library by itself proposes no less than 77 functions (128 in its most recent standardisation) to configure and perform message passing.

ify a certain number of endpoints on which to wait for a new message to arrive. A natural extension of this would be to provide the ability to wait on a *linked list* of endpoints. This is useful for programs that have to manage an unbounded number of channels, for instance a name server as found in the Singularity Operating System, whose task implies maintaining channels open towards every running process.

One could also write a program with multiple producers and consumers over a same channel, effectively sharing endpoints between processes. This has already been discussed on page 98 and we believe that it would be relatively painless to add to our programming language but that it would be met with more resistance on the side of the proof system. Indeed, our abstraction of the buffer contents by contracts would no longer be sensible, as the control states associated to endpoints would change freely when one of its sharers acts on it.

Another possible way of sharing channels would be to handle a *broadcast* mechanism on endpoints: a channel would for instance have one endpoint on the sending side, but several on the receiving one. This poses interesting theoretical challenges as well; for instance, when a message is simultaneously received by all its recipients, how the corresponding footprint would be distributed amongst them and how to include this mechanism in a proof system is not obvious.

One can also wonder whether the concepts presented in this thesis apply to other existing message-passing paradigms, for instance to Posix pipes or to the Erlang programming language, or even simply to programs that implement a form a message passing simply by sharing buffers between threads, without higher-level language constructs.

Independently from message passing concerns, the language itself could be more full-fledged and support realistic programming constructs such as function calls (which Heap-Hop already support), arrays, or more general data structures. In particular, adding a support for arrays in Heap-Hop would enable us to try and verify several message-passing examples from the domain of intensive computing, where arrays and matrices are often passed around between processes.

Automation Trying and automating the verification of message passing programs to the extent of minimal to non-existent user interaction is a natural extension of our work on Heap-Hop. We believe that we have given to our message passing primitives a logical status close enough to the ones of other synchronisations in separation logic to claim that automating the verification of programs in concurrent separation logic would likely result in progress in the automation of verification in our own setting. There is however room for an additional sort of automation: instead of specifying the protocols followed by the program, one could wish to automate this process and infer contracts from a given program. This process is likely to bear similarities to the inference of session types given a pi-calculus process. From the point of view of the asserter, who has to verify existing programs that may not have been written with certification in mind, it is crucial that formal tools only require a minimal amount of work from his part (for instance, annotating the parts of a program that a tool fails to verify).

Towards certified concurrent programming Writing correct concurrent programs has always contained a part of challenge for the programmer, who must avoid the many pits of race conditions, deadlocks, and more generally must see to it that programs cooperate good-manneredly instead of stepping on each other's toes. One could hope to see programming languages rise that make this task easier by providing a strong formal bedding to the programming constructs, such that formal verification becomes possible even to non-experts in the field. Tools towards this end may include contracts for channel communications or explicit ownership tracking of resources by the language, which we hope this dissertation has shown to be useful concepts in the context of message-passing program verification.

Bibliography

[ABS01] Aurore Annichini, Ahmed Bouajjani, and Mihaela Sighireanu. TReX: A tool for reachability analysis of complex systems. In *CAV*, volume 2102 of *LNCS*, pages 368–372, 2001. Cited on page 7.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *ISCA*, pages 2–14, 1990. Cited on page 13.

[AJ93] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. In *LICS*, pages 160–170, 1993. Cited on page 54.

[Alg10] Jade Alglave. *A Shared Memory Poetics*. PhD thesis, Université Paris 7 and INRIA, 2010. Cited on page 13.

[BAW10] 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 7, 98.

[BBL09] Kshitij Bansal, Rémi Brochenin, and Étienne Lozes. Beyond shapes: Lists with ordered data. In *FOSSACS*, volume 5504 of *LNCS*, pages 425–439, 2009. Cited on page 69.

[BCC$^+$07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'-Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *CAV*, volume 4590 of *LNCS*, pages 178–192, 2007. Cited on page 68.

[BCD$^+$08] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433, 2008. Cited on page 144.

[BCO05a] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, volume 4111 of *LNCS*, pages 115–137, 2005. Cited on pages 130, 131.

[BCO05b] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, volume 3780 of *LNCS*, pages 52–68, 2005. Cited on pages 133, 134, 135, 136.

[BCY06] Richard Bornat, Cristiano Calcagno, and Hongseok Yang. Variables as resource in separation logic. *Electr. Notes Theor. Comput. Sci.*, 155:247–276, 2006. Cited on pages 65, 67, 72.

[Bol06] Benedikt Bollig. *Formal Models of Communicating Systems - Languages, Automata, and Monadic Second-Order Logic*. Springer, 2006. Cited on pages 45, 51.

[Bor00] Richard Bornat. Proving pointer programs in hoare logic. In *MPC*, volume 1837 of *Lecture Notes in Computer Science*, pages 102–126, 2000. Cited on page 3.

[BP09] Gérard Boudol and Gustavo Petri. Relaxed memory models: an operational approach. In *POPL*, pages 392–403, 2009. Cited on page 13.

[Bro07] Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007. Cited on pages 35, 72, 74, 102, 112, 114.

[Bur72] Rodney M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7(1):23–50, 1972. Cited on page 3.

[BZ83] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983. Cited on page 50.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977. Cited on page 5.

[CDOY06] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS*, volume 4134 of *LNCS*, pages 182–203, 2006. Cited on page 68.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986. Cited on page 5.

[CF05] Gérard Cécé and Alain Finkel. Verification of programs with half-duplex communication. *Inf. Comput.*, 202(2):166–190, 2005. Cited on pages 7, 28, 43, 53, 54.

[CFI96] Gérard Cécé, Alain Finkel, and S. Purushothaman Iyer. Unreliable channels are easier to verify than perfect channels. *Inf. Comput.*, 124(1):20–31, 1996. Cited on page 7.

[CO01] Cristiano Calcagno and Peter W. O'Hearn. On garbage and program logic. In *FoSSaCS*, volume 2030 of *LNCS*, pages 137–151, 2001. Cited on page 19.

[COY07] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS*, pages 366–378, 2007. Cited on pages 35, 97, 102.

[Dod09] Mike Dodds. *Graph transformation and pointer structures*. PhD thesis, 2009. Cited on page 68.

[DOY06] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS*, volume 3920 of *LNCS*, pages 287–302, 2006. Cited on page 5.

[DY10] Pierre-Malo Deniélou and Nobuko Yoshida. Buffered communication analysis in distributed multiparty sessions. In *CONCUR*, volume 6269 of *LNCS*, pages 343–357, 2010. Cited on page 64.

[FAH+06] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*, pages 177–190, 2006. Cited on pages 4, 18, 20, 26, 27, 28, 43, 45, 46.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Mathematics*, volume 19, pages 19–32, 1967. Cited on pages 3, 69.

[FM97] Alain Finkel and Pierre McKenzie. Verifying identical communicating processes is undecidable. *Theor. Comput. Sci.*, 174(1-2):217–230, 1997. Cited on page 50.

[GBC+07] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *APLAS*, LNCS, pages 19–37, 2007. Cited on pages 7, 15, 34, 65, 74, 80, 94, 102, 122.

[HAN08] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, volume 4960 of *LNCS*, pages 353–367, 2008. Cited on page 102.

[HGS09] Alexander Heußner, Tristan Le Gall, and Grégoire Sutre. Extrapolation-based path invariants for abstraction refinement of FIFO systems. In *SPIN*, volume 5578 of *LNCS*, pages 107–124, 2009. Cited on page 7.

[HKP+10] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in Java. In *ECOOP*, volume 6183 of *LNCS*, pages 329–353, 2010. Cited on page 7.

[HL07] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *Operating Systems Review*, 41(2):37–49, 2007. Cited on pages 2, 18.

[HO08] Tony Hoare and Peter W. O'Hearn. Separation logic semantics for communicating processes. *Electr. Notes Theor. Comput. Sci.*, 212:3–25, 2008. Cited on page 6.

[Hoa69] Tony Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. Cited on pages 3, 69.

[Hoa78] Tony Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. Cited on page 6.

[Hol97] Gerard J. Holzmann. The model checker Spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997. Cited on page 7.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Pub, 2008. Cited on pages 13, 14.

[HVK98] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138, 1998. Cited on page 6.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284, 2008. Cited on pages 6, 63, 144.

[HYH08] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *ECOOP*, volume 5142 of *LNCS*, pages 516–541, 2008. Cited on pages 7, 26.

[IO01] 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 page 3.

[Jon83] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983. Cited on page 7.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979. Cited on page 13.

[LM09] K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393, 2009. Cited on page 7.

[LMS10] 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 pages 7, 144.

[May81] Ernst W. Mayr. An algorithm for the general petri net reachability problem. In *STOC*, pages 238–246, 1981. Cited on page 54.

[Maz86] Antoni W. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, volume 255 of *LNCS*, pages 279–324, 1986. Cited on page 6.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980. Cited on page 6.

[Mil99] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge Univ Pr, 1999. Cited on page 6.

[Min67] Marvin L. Minsky. *Computation: finite and infinite machines*. 1967. Cited on page 54.

[OG76] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976. Cited on page 75.

[O'H04] Peter W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR*, volume 3170 of *LNCS*, pages 49–67, 2004. Cited on page 73.

[O'H07] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007. Cited on pages 17, 65, 73, 74.

[OP99] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999. Cited on page 3.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19, 2001. Cited on page 3.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *TPHOLs*, volume 5674 of *LNCS*, pages 391–407, 2009. Cited on page 13.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002. Cited on pages 3, 65, 69.

[Rey04] John C. Reynolds. Toward a grainless semantics for shared-variable concurrency. In *FSTTCS*, volume 3328 of *LNCS*, pages 35–48, 2004. Cited on page 13.

[RG08] Mohammad Raza and Philippa Gardner. Footprints in local reasoning. In *FoSSaCS*, volume 4962 of *LNCS*, pages 201–215, 2008. Cited on page 70.

[SB09] Zachary Stengel and Tevfik Bultan. Analyzing Singularity channel contracts. In *ISSTA*, pages 13–24, 2009. Cited on pages 6, 28, 43, 45, 59.

[ST77] George S. Sacerdote and Richard L. Tenney. The decidability of the reachability problem for vector addition systems (preliminary version). In *STOC*, pages 61–76, 1977. Cited on page 54.

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE*, volume 817 of *LNCS*, pages 398–413, 1994. Cited on pages 6, 26, 28, 63, 88, 92.

[Vaf07] Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007. Cited on pages 7, 13, 65, 74, 102, 112, 114.

[VLC09] Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving copyless message passing. In *APLAS*, volume 5904 of *LNCS*, pages 194–209, 2009. Cited on pages 6, 97, 98, 99.

[VLC10] Jules Villard, Étienne Lozes, and Cristiano Calcagno. Tracking heaps that hop with Heap-Hop. In *TACAS*, volume 6015 of *LNCS*, pages 275–279, 2010. Cited on page 6.

[VP07] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, volume 4703 of *LNCS*, pages 256–271, 2007. Cited on pages 7, 15, 102.

[YDBH10] Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. In *FOSSACS*, volume 6014 of *LNCS*, pages 128–145, 2010. Cited on page 64.