FORMAL VERIFICATION OF LOW-LEVEL SOFTWARE
低レベルソフトウェアの形式的な証明

by

Nicolas Marti
マーティ ニコラ

A Dissertation

Submitted to
the Graduate School of
the University of Tokyo
on March 27, 2008

in Partial Fulfillment of the Requirements
for the Degree of Doctor of Information Science and
Technology in Computer Science

Thesis Supervisor: Yonezawa Akinori 米澤 明憲
Professor of Information Science

I would like to thank Prof. Akinori Yonezawa, to have given me the opportunity to come to study in Japan, and more specifically in his laboratory at the University of Tokyo. I want also to thank all the members of the Yonezawa Laboratory for there helps and interesting scientific discussions.

I would like to thank all the people from AIST-RCIS, and especially among them, Reynald Affeldt, who teach me nearly everythings I have learned during this Ph.D. (sans toi, la these je l'avais pas! En fait, je la commencait meme pas!).

Finally I would like to thank Ako, for her support during the long hours of work at home.

要旨

Formal verification of low-level software has became a trend in the formal methods community. Indeed, this software captures much of our attention, as it tends to be embedded in every day devices. However, the real stakes remain in the fact that much of these devices are critical for human lives: airplanes flying controls, vital medical devices or security probes in power plants. Beyond such stakes, the challenge that such software verification represents also motivates the research community.

In this work we investigate how to verify a memory property for a special case of low-level software: operating systems. More precisely, we are interest in proving the task isolation for the Topsy operating system. The task isolation is the property that asserts that no user application can access the operating system kernel memory. Topsy is an embedded operating system, dedicated to intelligent network devices, implemented in the Engineering School of Zurich (Switzerland).

Our approach makes use of two well-known formal methods: model-checking and interactive theorem proving. Using the SPIN model checker, we build an abstraction of the whole operating system, its underlying hardware, and some test user-applications. We verify formally on this model several properties, and among them, the task isolation. Through this model we are able to identify which parts of the code have a key role in the property. To formally prove the correctness of the source code, we use the Coq proof assistant. More precisely, we implement a library to mechanically prove Hoare-triples. We formally verified several parts of Topsy source code: context switching, the list library, and the memory allocator. For the later, our verification allowed to find non-trivial bugs.

The most important contribution of our work is a set of Coq libraries to mechanically prove Hoare-logic triples. These libraries has proven useful in the verification of source code taken from a realistic operating system. Our libraries implement the semantics of a subset of C and MIPS assembly, separation logic connectives, and proof systems for separation logic triples. For all of them we prove their completeness and soundness w.r.t. the formal semantics of separation logic triples (defined through the language semantics). Our libraries also include several tactics that discharge the users from several subgoals. Another important contribution is an original verification algorithm for separation-logic triples. We implemented it by reflection inside the Coq proof assistant, and prove mechanically its soundness. This is the first certified verifier, for separation logic, implemented by reflection in a prof assistant.

形式的検証の研究分野では、ハードウェアに近い低レベルソフトウェアの形式的検証が盛んに行われはじめている。これは、低レベルソフトウェア (自動車のブレーキシステムや飛行機の飛行制御、生命に関わる医療機器、発電所の保安調査など) が我々の生命に影響するからである。これに加え、低レベルソフトウェアの検証の困難さも形式的検証が盛んになっている理由の一つである。実際、低レベルソフトウェアは複雑で、しばしば複数のプログラミング言語で記述され、また難解な制御フローを実装しているため、検証は困難である。

本論文は、低レベルソフトウェアの一つである、オペレーティングシステムカーネルの形式的検証を扱う。なお、本論文ではネットワークデバイスのための組み込みオペレーティングシステムである Topsy を対象とする。オペレーティングシステムのようなソフトウェアは、ユーザアプリケーションを実行したり、それらにサービスを提供したり、外部とのインターフェイスを提供したりする。我々の主目的は、オペレーティングシステムカーネルとユーザアプリケーションの相互作用にある。

本論文は、まずシステムを SPIN モデル検査器においてハードウェア、カーネル、ユーザアプリケーションのモデルとして抽象化する。この抽象化を用いて、メッセージ通信サービスとカーネルメモリ領域の保護に関する性質を検証する。カーネルメモリ領域の保護に関する性質を扱うためには、メモリ確保とコンテキスト切替の機能を実現するコードが重要となる。このため、次にこれらの機能を実装するソースコードの検証を行う。この目的のために、C に似た言語と MIPS アーキテクチャ向けアセンブリ言語を Coq 定理証明支援系においてモデル化しライブラリとして実装する。また、分離論理と呼ばれるよく知られた Hoare 論理の拡張もライブラリとして実装する。これらのライブラリを用いて Topsy のメモリアロケータとコンテキスト切替のコードを検証する。更に、検証の自動化についても調査を行う。また、正しさが証明された分離論理の検証のための関数を Coq において実装し、この関数から正しさが保証された検証器を自動生成する。更にこの検証器を用いて、オペレーティングシステムに必須のデータ構造であるリスト操作の検証を行う。

Contents

List of Figures

Chapter 1

Introduction

Formal verification of low-level software has become a trend in the formal methods community. This is justified by the fact that our lives rely on this software. Indeed, nowadays cars brake systems rely on such software, as does the flying controls of our airplanes, the vital medical devices in our hospitals, and the probes that maintain the security in our power plants. Beyond these crucial stakes, the difficulty of such software verification also motivates the community. Indeed this software verification presents a challenge. Low-level software are complex because they directly manipulate their underlying hardware, because, for sake of code reusability, they are written in different languages and finally because, for most of them (like operating systems), they implement subtle control-flow.

1.1 Motivation

In this thesis, we study the formal verification of a particular case of low-level software: operating system kernels. The main goal of these kind of software is to provide an abstraction, of the underlying hardware, to applications. More precisely, the operating system kernels provide to applications a running environment: execution slots (abstraction of the processor), services (like synchronization and communication), and a normalized interface for inputs/outputs (abstraction of the peripherals). Therefore, applications are highly dependant on the operating system they are running on. This implies that the correctness of a program can only be assured if the operating system is also correct.

The recent progress of the hardware technologies (both in prices and features of processors and memories), and economics stakes (reusable software are more profitable than specific ones) fed the trend of using operating systems in more and more embedded solutions. However, the trade-off of this evolution is that certification of systems becomes more and more tedious.

In this thesis, we investigate how one can verify an operating system, using two of the most well-studied formal methods: model-checking and interactive theorem proving. Obviously the full verification of this kind of software is not yet tractable, still we provide a method that can be used for the verification of precise property. In our work we focus on a memory property: the *task isolation* [7]. This property can be stated informally as:

`A user application cannot access (i.e., read or write in) the kernel space memory`

This property is important, as it ensures that no user applications will corrupt the system through its data. Most of viruses and root-kits try to override their privilege in order to gain the control of the system. We illustrate our methods through the verification of an embedded operating system, for network devices, named Topsy [10].

Basically, our approach consists in building the abstraction of the whole operating system, as well as its underlying hardware, inside the SPIN model checker. Thanks to this model we verify some properties of the system, such as flag consistency for the message passing facility, property of the election of the next thread to be executed, and obviously the task isolation. Through this model we can find which part of the system are important for this later property. Our next move is to formally verify the corresponding source code. For this purpose, we implement, in the Coq proof assistant, a library to mechanically certified C-like and MIPS assembly source code. This libraries are used to formally verify the memory allocator (also known as the heap manager), and the context switching source code of Topsy. Our verification has proven useful, as it allowed us to find non-trivial bugs inside the original source code.

1.2 Background

In this section, we provide some basic background informations that should help the readers with the material presented in this thesis. First we provide a general description of what is an operating system, and an overview of the Topsy operating system. Then, we describe the two formal methods that we applied to the verification of Topsy: model checking and interactive theorem proving.

1.2.1 Operating Systems

In order to build an application for a computer, there is mainly two approaches: either directly write a program that will manage the underlying hardware (the processor and the peripherals), or to leave this role to some underlying software, that will provide an abstraction of the underlying hardware (namely, an operating system)[1]. The first approach main benefit is the possibility to build an application as optimized as possible, and main disadvantages are that they are more difficult to develop, and that they can only run on the same model of hardware. The second approach disadvantage is an overhead over the resource usage. The main benefits are that if the interface of the operating system is standard, then a given application should be able to run on different platform (as long as they are all supported by the operating system), and that the development costs are reduced.

Although it is customary to use the term operating system, it would be more correct to use the term operating system kernel. This term refers to the software that run on top of the hardware, and that provide the abstractions to

[1]A further abstraction consists in building an additional layer between the operating system and the applications (this layer is oftenly named a virtual machine). The most famous system which embraces such an approach is Java from SunMicrosystem.

the application, whereas operating system could refer to a whole set of software (obviously including the kernel, but also the user interface or the file system, if any).

The Topsy Operating System

We have chosen the Topsy operating system as a test-bed for the formal verification of task isolation. Topsy was initially created for educational use and has recently evolved into an embedded operating system for network cards [10]. It is well-suited for mechanical verification because it is small and simple, yet it is a realistic use-case because it includes most classical features of operating systems.

Topsy was originally developed for the MIPS architecture [40], yet it provides an abstraction layer for the underlying processor. This abstraction layer has proven useful to port Topsy to the Intel x86 architecture. Topsy implements a flat memory model, which means that the paging feature of the processors is not used. Topsy splits the memory of the underlying hardware into two segments: one is used for the kernel programs, and the other for the user programs.

Topsy is a micro-kernel, which means that all the high-level services are provided by kernel-thread servers, while the kernel only provides message passing. The interfaces between user applications and kernel-thread servers are build on top of the message passing interface.

Topsy does not include dynamic loading of code. This implies that the user applications (which may be composed of several threads) and the kernel code must be compiled and linked together. This specificity is not a limitation to Topsy usage, because it is an embedded operating system.

1.2.2 Formal Methods

The denomination formal methods refers to the set of approaches that allow to mathematically model a system and its properties. All these methods may be sorted by the notion of tradeoff between expressiveness (which condition the size and accuracy of models) and the effort cost for verification.

In our work we make use of two well-known formal methods: model-checking and interactive theorem proving. Basically, model checking provides automatic verification for state transition systems (graphs that model all the possible executions), whereas interactive theorem proving provides an environment to write by hand proofs for models build through functional programs and logical relations.

Model Checking

Model checking is a formal method which focuses on model: abstraction of some system (software, hardware or more complex system, such as a subway). The main principle of model checking consists in enumerating all the possible execution traces, and find either, if for all of them a desired property is satisfied, or if there is some state that satisfies an error property. The main challenge faced by model checker is the state explosion: the number of possible execution traces grows exponentially in function of the number of variables in the model.

The main advantage of model checking is that verification of some property for a given model is entirely automatic. However, in general, the state explosion problem implies that the verification may be untractable. Thus, most of the time, the use of model checker, such as SPIN, asks the user to manipulate a certain number of parameters which control optimizations for the state space exploration (like compression of states, partial order reduction, . . .). Obviously the fact that model checking is automatic also implies some trade-off over the expressiveness of the model description and the formula languages.

Interactive Theorem Proving

Interactive theorem proving is a formal method which objective is to mechanically formalize the pencil-and-paper proofs, such that their correctness can be automatically verified[2]. More precisely, one models the systems in terms of mathematical definitions and functions. Proof assistants also provide definition for the most common logical formulas, which are used to build lemmas. Finally, these tools provide interactive command language to build proofs.

Interactive proof assistant are convenient to implement complex systems in their full details. Indeed these tools languages have enough expressiveness to detail every aspects of any systems. This expressiveness also allows to express any properties, as precise as possible. The obvious trade-off is that the verification of complex properties over complex systems imply complex proofs. However, most well engineered proof assistant implements several decision procedure (like first order, or Presburger arithmetic), in order to discharge the user from tedious, yet not complex, proofs. Although such facility exists, building proofs in this tools remains hard, as it includes a non-negligible overhead compared to pencil-and-paper proofs.

1.3 Our Verification Approach

Our approach is to consider the system from different points of view. First, we build an abstract model of the whole system, inside the SPIN model checker [8]. This abstraction allows to consider each parts of the system, their interactions, and gives an accurate image of how the system executes. Through this model, we try to find which parts of the system are important for the task isolation property. Once we have identified them, we observe what are their desired behaviors and how they are related to the task isolation. Through these studies we deduce for each part what its specification is. More precisely, we capture through logical assertions their behaviors that, if not respected, may lead a user thread to be able to read or write inside the kernel memory.

Then we take a closer look at the source code of each identified parts. We input their concrete specifications, and semantics inside the Coq proof assistant [12], and formally build their proofs of correctness.

[2]The Goedel incompleteness asserts that the proofs cannot be generated automatically. However, the Curry-Howard isomorphism asserts that proofs can be checked automatically.

1.3.1 Design Verification

For the design verification, we use the SPIN model checker [8]. Thanks to its model language `Promela`, we can model the whole Topsy operating system, its underlying hardware, and an illustrating user application. For our model, we have chosen to implement a user application that makes use of all the services implemented in our model: a multi-threaded echo-server. This user application uses the thread creation and deletion services, the message passing facility, and the services provided by the network server.

We also model some desired properties for the system, and among them the task isolation [44]. The model benefits are to give a readable image of how the system works, and also to test an application running together with its environment. Another benefits is that we can test which parts of the system must be proved correct.

Thanks to the model, we identify several parts of the code for which correctness is a necessary condition for the task isolation property. Among them, we identified the memory allocator of Topsy (also known as the heap manager), which is used by the kernel and the servers to allocate fresh blocks of memory.

1.3.2 Source Code Verification

To verify the Topsy source code, we implement the separation logic inside of the Coq proof assistant [32]. On this base, we develop tactics to help for the verification of C-like source code. Thanks to this library, we verify all the functions of the Topsy memory allocator. An interesting output of our verification is that we have found bugs, one of them leading to the lost of allocable memory. This is an important problem, as it may freeze the system (no allocable memory means that the system cannot performs most of its actions).

Another part of the code that we must verify is the context switching. This code is written in assembly. As we previously have investigated the verification of programs in the MIPS architecture in [42], we have chosen to focus on the MIPS port of Topsy. We implement a library inside the Coq proof assistant to deal with the verification of MIPS assembly source code. Thanks to this implementation we verify the function that restores the threads context.

1.3.3 Verification Tools

We also have identified simple piece of code which correctness must be ensured. For instance, the function that creates the threads. Although their verification are important, such straightforward code may be cumbersome to prove inside our implemented library. For such piece of code, we investigate a decidable fragment of the separation logic, and provide an original verification algorithm [45]. We implement it inside the Coq proof assistant, and mechanically prove its soundness (which means that if the verifier says that the specification is correct, then it is indeed the case). Thanks to the extraction mechanism of Coq, we are able to extract a certified and stand-alone verifier in Ocaml.

So far, we have verified several piece of code in both C and assembly. Yet these specifications are not directly compatible, because they do not consider the same command language. Our next effort will focus on how to resolve such an important issue: the composition of specifications. Indeed, when one builds a runnable image of Topsy, one has to compile all the source code and to link it together. The problem is that we would like to reason about the specification of the resulting program, using, if possible, already verified specifications of its different components. We propose a translator from C-like to assembly language, implemented and certified inside the Coq proof assistant. Thanks to its correctness we show how one can reuse specifications, verified at both language levels, in order to build the specification of the results of the translation.

1.4 Related Work

The delta-core project [46] aims at verifying a micro-kernel written in a C-like language. Verification of properties of system calls have been specified and verified in the PowerEpsilon proof assistant after source code translation. The main difference from our work is that we focus on properties of memory management, and that we prove both C and assembly source code. Moreover, our verification includes a mechanical verification of the whole operating system through model checking.

The VFiasco project [47] aims at verifying memory properties of a micro-kernel. The approach is to automatically translate a subset of the C++ language into the PVS proof assistant where a model of x86 processors has been implemented. The translation process seems to be the current challenge this project is facing. As far as we know, no illustration of formal verification of source code inside a proof assistant has been yet released.

The FLINT group at Yale intends to build verification methods and tools for constructing large-scale system software [51]. This project is composed of a verification framework inside the Coq proof assistant, used for several verification purposes: a memory allocator [15], a self-modifying code [52], and a context switching code [41]. This project does not yet focus an automation of verification, or certification of compilation. Moreover, they do not focus especially on operating system, and hence does not propose approaches for their verification.

Singularity [53] aims to build a highly reliable OS based on a model of object oriented applications. All code outside the kernel execute in an encapsulation, called SIP: a closed object space. Singularity is developed using a type-safe language (C#) and assembly code. The code composing the kernel is either "verified" or "trusted". "Verified" means that the compiler checks the type and memory safety, while "trusted" refers to parts written in unsafe C#, assembly, or C++. There exists several differences with our work. First, their is no effort to consider the whole system, as we propose through our abstraction in the SPIN model checker. Then, the verification of Singularity does not really on mechanical proof, as we propose through our library for verifying source code, using the Coq proof assistant.

The project Verisoft [54] aims to achieve the correctness of critical system using computer-aided logical proofs. This project model the whole system, from the hardware to the user application, and the operating system kernel. For this purpose they used model-checker, theorem prover and proof assistant. For so long, the project is in a planning process, particularly emphasizing in the conception of the set of tools they will need to fulfill the verification of operating systems. Our approach is different, as we choose to use already existing tools (namely, the SPIN model-checker and the Coq proof assistant), and focus entirely on the verification of a realistic operating system.

1.5 Contributions

The main contributions of this thesis are libraries to build proof in separation logic for both a subset of C and the MIPS assembly, implemented in the Coq proof assistant. These libraries have proven useful to certify realistic code.

Another contribution of this thesis is an original algorithm for the automatic verification of a decidable subset of the separation logic. This algorithm can be experimentally shown to produce, in presence of pointer arithmetic, proofs smaller than previous approaches. We implemented our algorithm on top of our separation logic library in Coq, and proved its correctness. An interesting output is a stand-alone and certified verifier for separation logic.

A last contribution, regarding verification of source code, is a translator from a subset of C to a subset of an assembly language, that is proved to preserve the semantics. This translator and its preservation properties are implemented in Coq.

The model of Topsy inside the SPIN model-checker stands as another contribution of our work. This model represent a clear image of the whole system. It also allows to test the design of a user application, and to simulate its execution inside in its running environment. Finally, this model can be a good opportunity for a reimplementation of the operating system.

1.6 Dissertation Outline

This Ph.D. thesis is organized as follows. In Chap. 2, we introduce knowledges that should help the reader not familiar with formal verification and operating systems. First we present the main theory and the tool that we used to formally verify source code, namely the separation logic and the Coq proof assistant. Then, we present the model of Topsy inside the SPIN model checker. After describing the abstraction of the system, we present the formalization of some desired properties, and among them the task isolation. We then identify several parts of the code where correctness are necessary for this property. In Chap. 3, we present a Coq library whose purpose is to model source code in C, as well as the formal verification of the Topsy memory allocator. In Chap. 4, we present a variant of the previous library, which deals with verification of MIPS assembly source code. Using this implementation, we verify the source code for the context restoring function of Topsy. In Chap. 5, we present a certified verifier for a decidable fragment of separation logic, implemented and certified sound inside the Coq proof assistant. In Chap. 6, we present a translator from a C-like language to an assembly language, which has been proved preserving the programs semantics. Through this translator we emphasize how one can compose specifications of source code written in both C and assembly. Finally, in Chap. 7, we conclude this Ph.D. thesis, by summarizing our work.

Part I

Verification

Chapter 2

Background

In this chapter, we provide to the reader the minimal knowledge to be comfortable with the material presented in the next chapters. Our main motivation is to provide a self-contained document. However we obviously point to further materials, for readers interested in the underlying researches that are used as foundations for this work.

First, in Sect. 2.1, we describe the most important theoretical work on which this thesis is based: the separation logic [1]. This is an extension of the Hoare-logic with a native notion of heap and pointers. Then, in Sect. 2.2, is about the Coq proof assistant. This tool was used to formalize the separation logic framework and to build the proof of correctness for the source code (Chap. 3 and Chap. 4), as well as to develop a certified verifier (Chap. 5), and a certified translator (Chap. 6). Finally, in Sect. 2.3.1, we present the Topsy operating system, more precisely a model written in the SPIN model-checker. Through this abstraction, we test several properties, and among them the task isolation. These experiments allows us to identify which parts of the kernel play a key role in this property.

2.1 The Separation Logic

The separation logic is an extension of the Hoare-logic with a native notion of heap and pointers, introduced by John C. Reynolds in [1]. We start this section by presenting the command language syntax and semantics. Then, we present how the separation logic extends an assertion language through new logical connectives. Finally we present the Reynolds axioms: a set of rules providing a sound and complete proof system w.r.t. the semantics.

2.1.1 The Command Language

The command language of separation logic is an extension of the Hoare-logic command language. Separation logic extends (1) the state with a heap, and (2) the command language with new assignments to access the heap. These extensions necessitate to take into account an error case in the semantics, which may arise when a program tries to access a cell that is not present in the memory.

$$\begin{array}{lll@{\qquad}lll}
\mathit{nexpr} ::= & x,\ y,\ z,\ \ldots & (\texttt{var}) & \mathit{bexpr} ::= & \texttt{true},\ \texttt{false} & (\texttt{const}) \\
 & \ldots,\ -1,\ 0,\ 1,\ \ldots & (\texttt{const}) & & \mathit{nexpr} \texttt{==} \mathit{nexpr} & \\
 & \mathit{nexpr}\ +\ \mathit{nexpr} & & & \mathit{nexpr}\ \leq\ \mathit{nexpr} & \\
 & \mathit{nexpr}\ -\ \mathit{nexpr} & & & \mathit{nexpr}\ <\ \mathit{nexpr} & \\
 & \mathit{nexpr}\ *\ \mathit{nexpr} & & & \mathit{nexpr}\ \geq\ \mathit{nexpr} & \\
 & \mathit{nexpr}\ /\ \mathit{nexpr} & & & \mathit{nexpr}\ >\ \mathit{nexpr} & \\
 & & & & \mathit{bexpr} \vee \mathit{bexpr} & \\
 & & & & \mathit{bexpr} \wedge \mathit{bexpr} & \\
 & & & & \neg \mathit{bexpr} &
\end{array}$$

Figure 2.1: Numerical and boolean expression languages.

States

A state of the separation logic language is a couple of a store and a heap. A store is a function from *variables* to *values*. There exist an initial store where all the variables are initialized to some value. For x a variable and s a store we note $[\![x]\!]_s$ for the evaluation of x through s. We note $s_{[v/x]}$ the store s where the variable x is updated with the value v. A heap is a map from *locations* to values. This is a partial function, as some location may not be present in the heap domain. This implies some "option" value for the evaluation of the heap. More formally we note the evaluation of a location l through a heap h as:

$$[\![l]\!]_h = \begin{cases} v & \text{if } h \text{ maps } l \text{ to } v \\ \mathit{none} & \text{if } l \text{ is not in the domain of } h \end{cases}$$

We note $\emptyset_{heap}$ the heap which has an empty domain. We define disjointness of two heaps h_1 and h_2 by the facts that the intersection of their domains is empty, and we note it as: $h_1 \perp h_2$. We define the concatenation of two heaps h_1 and h_2, as the union of their definitions, and we note it as: $h_1 \cup h_2$. We note $h_{[v/l]}$ the heap h where the location l is updated with the value v. Note that an update (respectively a concatenation) is only valid if the location l is already include in the domain of h (resp. if both heap domains are disjoint).

The Language Syntax

The command language of separation logic is similar to the original command language for Hoare-logic. It contains two expression languages: the numerical expression language and the boolean expression language. We defined these languages formally in the figure 2.1.

$$\begin{array}{lll}
\mathit{cmd} ::= & \texttt{skip} & \\
 & \mathit{var} \leftarrow \mathit{nexpr} & (\texttt{assignment}) \\
 & \mathit{var} \leftarrow\!\!* \ \mathit{nexpr} & (\texttt{lookup}) \\
 & \mathit{nexpr}\ *\!\!\leftarrow \mathit{nexpr} & (\texttt{mutation}) \\
 & \mathit{cmd};\ \mathit{cmd} & (\texttt{sequence}) \\
 & \texttt{if}\ \mathit{bexpr}\ \texttt{then}\ \mathit{cmd}\ \texttt{else}\ \mathit{cmd} & (\texttt{conditional}) \\
 & \texttt{while}\ \mathit{bexpr}\ \texttt{do}\ \mathit{cmd} & (\texttt{loop})
\end{array}$$

Figure 2.2: Command language.

In figure 2.2, we formally define the separation logic command language. The commands `lookup` and `mutation` are the extensions to the classical Hoare-logic command language. These commands are respectively used to read and to write values inside a heap. We use a variant of the original notation for command proposed in [1].

The Language Semantics

The execution of the command language is described as a big-step operational semantics in 2.3. In these rules, we use two extensions to the notation for the evaluation function through stores: if e (respectively b) is a numerical expression (resp. a boolean expression), then $[\![e]\!]_s$ (resp. $[\![b]\!]_s$) stands for its evaluation through s. More precisely, this is the evaluation of an expression when its variables are evaluated through the store s.

Most of the semantics rules are well-known. The new rules concern the mutation and the lookup. For both commands, the first rule represents a valid execution, and the second rule represents an error in its execution, due to the fact that the addressed location is not in the domain of the initial heap.

$$\frac{}{(s,h) = \texttt{skip} \Longrightarrow (s,h)}\ \texttt{skip} \qquad \frac{}{(s,h) = (x \leftarrow e) \Longrightarrow (s_{[[\![e]\!]_s/x]}, h)}\ \texttt{assign}$$

$$\frac{[\![e]\!]_s = l \quad [\![l]\!]_h = v}{(s,h) = (x \leftarrow\!\ast\ e) \Longrightarrow (s_{[v/x]}, h)}\ \texttt{lookup} \qquad \frac{[\![e]\!]_s = l \quad [\![l]\!]_h = none}{(s,h) = (x \leftarrow\!\ast\ e) \Longrightarrow \texttt{Abort}}\ \texttt{lookup\_error}$$

$$\frac{[\![e_1]\!]_s = l \quad [\![l]\!]_h \neq none \quad [\![e_2]\!]_s = v}{(s,h) = (e_1 \ast\!\twoheadleftarrow e_2) \Longrightarrow (s, h_{[v/l]})}\ \texttt{mutation} \qquad \frac{[\![e_1]\!]_s = l \quad [\![l]\!]_h = none}{(s,h) = (e_1 \ast\!\twoheadleftarrow e_2) \Longrightarrow \texttt{Abort}}\ \texttt{mutation\_err}$$

$$\frac{}{\texttt{Abort} = c \Longrightarrow \texttt{Abort}}\ \texttt{abort}$$

$$\frac{[\![b]\!]_s = true \qquad (s,h) = c_1 \Longrightarrow (s',h')}{(s,h) = (\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2) \Longrightarrow (s',h')}\ \texttt{if\_true}$$

$$\frac{[\![b]\!]_s = false \qquad (s,h) = c_2 \Longrightarrow (s',h')}{(s,h) = (\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2) \Longrightarrow (s',h')}\ \texttt{if\_false}$$

$$\frac{[\![b]\!]_s = false}{(s,h) = (\texttt{while}\ b\ \texttt{do}\ c) \Longrightarrow (s,h)}\ \texttt{while\_false}$$

$$\frac{[\![b]\!]_s = true \qquad (s,h) = c \Longrightarrow (s',h') \qquad (s',h') = (\texttt{while}\ b\ \texttt{do}\ c) \Longrightarrow (s'',h'')}{(s,h) = (\texttt{while}\ b\ \texttt{do}\ c) \Longrightarrow (s'',h'')}\ \texttt{while\_true}$$

Figure 2.3: Big-step operational semantics of the command language

2.1.2 The Assertions Language

In this section we describe the connectives introduced by separation logic. In the classical Hoare-logic, the assertion language can be any logic that allows to assert properties over the values of the variables. Thus, for any predicate P of this logic, we define the satisfiability relation by $s \models P$, for some store s. Separation logic defines connectives to reason about the heaps. These connectives are used to extends the assertion language, such that the satisfiability relation extends over a couple of a store and a heap.

Basic Assertions

The first new predicate defined by separation logic is `emp`. This predicate only holds for an empty heap, more precisely for a heap which domain is the empty set. We can formally defined it as:

$$\emptyset_{heap} \models \texttt{emp} \wedge \forall h, h \neq \emptyset_{heap} \rightarrow h \not\models \texttt{emp}$$

Another predicate allows to describe a cell. We note $l \mapsto v$ the predicates that holds for a heap h which (1) domain is a singleton {l}, and (2) such that the image of the location l is v. More formally:

$$h \models (l \mapsto v) \iff [\![l]\!]_h = v \wedge \forall l', l \neq l' \rightarrow [\![l']\!]_h = none$$

We can overweight the connective $\mapsto$ such that its satisfiability relation extends to a state:

$$(s, h) \models (e_1 \mapsto e_2) \iff h \models ([\![e_1]\!]_s \mapsto [\![e_2]\!]_s)$$

We note the implication between separation logic predicates as $\Rightarrow$, and we defined it formally as:

$$\forall P, Q, P \Rightarrow Q \iff \forall s, h, (s, h) \models P \rightarrow (s, h) \models Q$$

Similarly we defined $\Leftrightarrow$ as:

$$\forall P, Q, P \Leftrightarrow Q \iff P \Rightarrow Q \wedge Q \Rightarrow P$$

The Separating Conjunction

Separation logic defines a special conjunction for which each hand-side assertions hold for disjoint parts of a heap. This connective allows to reason over non-overlapping parts of a heap. We note this conjunction, named *separating conjunction*, as $P \star Q$ (for any predicates P and Q). Formally, this predicate holds if a heap h can be split into two disjoint heaps h_1 and h_2 such that P holds for the first one and Q holds for the other. More formally:

$$h \models P \star Q \iff \exists h_1, \exists h_2, (h = h_1 \cup h_2) \wedge (h_1 \perp h_2) \wedge (h_1 \models P) \wedge (h_2 \models Q)$$

Once again, we can derive easily a satisfiability relation over a couple of a store and a heap. Here follows some examples:

$$(s, h) \models (x \mapsto x + 1 \star x + 1 \mapsto 0)$$
$$(s, h) \not\models (x \mapsto x \star x \mapsto y)$$

Here follows the lemmas asserting the commutativity and associativity of the separating conjunction:

$$\forall P, Q, P \star Q \Leftrightarrow Q \star P$$

$$\forall P, Q, R, (P \star Q) \star R \Leftrightarrow P \star (Q \star P)$$

The Separating Implication

Separation logic defines also a variant of the implication noted $-\!\!\star$, namely the *separating implication.* This connective allows to reason about concatenation of a heap with a fresh heap for which the l.h.s. assertion holds. More formally:

$$h \models P -\!\!\star Q \iff \forall h', (h \perp h' \wedge h' \models P) \rightarrow (h \cup h') \models Q$$

The separating implication, used with the separating conjunction, can be used to represent a destructive update.

An interesting feature of the separation logic connectives is that they have a modus-ponens-like rule:

$$\forall P, Q, (P \star (P -\!\!\star Q)) \Rightarrow Q$$

Data-structures Definitions

The separation logic has shown convenient to describe some of the most widely used data-structures. Here we just define some of the most well-known.

We define the singly-linked list element as a couple of two contiguous cells, where the first is used to store a data and a second is used as a link to the next element. The corresponding separation logic predicate takes two arguments: the first is the location of the initial element of the list, and the second is the location of the last pointed element. Formally put:

$$\begin{array}{lll}(s,h) \models \mathit{list}(e_1,e_2) & \iff & (\llbracket e_1 \rrbracket_s = \llbracket e_2 \rrbracket_s \wedge h \models \mathtt{emp}) \vee \\ & & (\llbracket e_1 \rrbracket_s \neq \llbracket e_2 \rrbracket_s \wedge \exists d, n, (s,h) \models (e_1 \mapsto d \star e_1{+}1 \mapsto n \star \mathit{list}(n,e_2)))\end{array}$$

The doubly-linked list is defined in a analogous way. This time each element is composed of three contiguous cells, because we now need to keep information about two links: one for the previous element and one for the next element. The predicate describing doubly-linked list takes four arguments: the first and the second are respectively the location of the first element and its previous link, whereas the third and forth are the next pointer of the last element and its location:

$$\begin{array}{lll}(s,h) \models \mathit{dlist}(e_1,e_2,e_3,e_4) & \iff & (\llbracket e_1 \rrbracket_s = \llbracket e_3 \rrbracket_s \wedge \llbracket e_2 \rrbracket_s = \llbracket e_4 \rrbracket_s \wedge h \models \mathtt{emp}) \vee \\ & & (\neg(\llbracket e_1 \rrbracket_s = \llbracket e_3 \rrbracket_s \wedge \llbracket e_2 \rrbracket_s = \llbracket e_4 \rrbracket_s) \wedge \\ & & \quad \exists d, n, (s,h) \models (e_1 \mapsto d \star e_1{+}1 \mapsto n \star e_1{+}2 \mapsto e_2 \star \mathit{dlist}(n,e_1,e_3,e_4)))\end{array}$$

The trees are also widely used data-structures. The separating conjunction is a convenient connective to define trees. Each element of the tree is defined by three contiguous cells: the first for the data, and the two others for the children. We define a value `end`, to capture that a link does not point to a child. The tree predicate takes an unique argument: the location of the root element.

$$\begin{array}{lll}(s,h) \models \mathit{tree}(e_1) & \iff & (\llbracket e_1 \rrbracket_s = \mathtt{end} \wedge h \models \mathtt{emp}) \vee \\ & & (\llbracket e_1 \rrbracket_s \neq \mathtt{end} \wedge \exists d, n_1, n_2, \\ & & \quad (s,h) \models (e_1 \mapsto d \star e_1{+}1 \mapsto n_1 \star e_1{+}2 \mapsto n_2 \star \mathit{tree}(n_1) \star \mathit{tree}(n_2)))\end{array}$$

2.1.3 The Separation Logic Triples

Similarly to Hoare-logic, the final purpose of separation logic is to abstract the operational semantics, using pre/post-conditions instead of initial/final-states. For this purpose the separation logic extends the Hoare proof systems with new rules, namely the Reynolds' axioms. They are represented by the rules `lookup` and `mutation` in Fig.2.4. Both rules implements a backward reasoning, using the separation logic connectives to model a destructive update in the pre-condition.

$$\frac{}{\{Q\}\ \texttt{skip}\ \{Q\}}\ \texttt{skip} \qquad \frac{}{\{Q[e/x]\}\ (x \leftarrow e)\ \{Q\}}\ \texttt{assign}$$

$$\frac{}{\{\exists v, (e \mapsto v) \star ((e \mapsto v) \mathrel{-\!\!\star} Q[v/x])\}\ (x \leftarrow\!\!* \ e)\ \{Q\}}\ \texttt{lookup}$$

$$\frac{}{\{\exists v, (e_1 \mapsto v) \star ((e_1 \mapsto e_2) \mathrel{-\!\!\star} Q)\}\ (e_1 \ast\!\!\leftarrow e_2)\ \{Q\}}\ \texttt{mutation}$$

$$\frac{\{P \wedge [\![b]\!] = \texttt{true}\}\ (\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\ \{Q\} \quad \{P \wedge [\![b]\!] = \texttt{false}\}\ (\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\ \{Q\}}{\{P\}\ (\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\ \{Q\}}\ \texttt{ifte}$$

$$\frac{\{I \wedge [\![b]\!] = \texttt{true}\}\ c\ \{I\}}{\{I\}\ (\texttt{while}\ b\ \texttt{do}\ c)\ \{I \wedge [\![b]\!] = \texttt{false}\}}\ \texttt{while}$$

$$\frac{P \Rightarrow P' \quad \{P'\}\ c\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ c\ \{Q\}}\ \texttt{impl}$$

Figure 2.4: Axiomatic semantics for separation logic.

All the rules in the proof system Fig. 2.4 can be proved sound and complete w.r.t. the operational semantics Fig. 2.3. Both properties are formally specified as:

Soundness: $\forall P, c, Q,$
$\{P\}\ c\ \{Q'\} \rightarrow$
  $\forall s, h, (s, h) \vdash P \rightarrow$
    $\forall s', h', (s, h) =\!= c \Longrightarrow (s', h') \rightarrow$
      $(s', h') \vdash Q$

Completeness: $\forall P, c, Q,$
$(\forall s, h, (s, h) \vdash P \rightarrow$
  $\forall s', h', (s, h) =\!= c \Longrightarrow (s', h') \rightarrow$
    $(s', h') \vdash Q) \rightarrow$
      $\{P\}\ c\ \{Q'\}$

2.2 The Coq Proof Assistant Through an Example: an Arithmetic Verification Procedure

In this section we provide to the reader an introduction to the Coq proof assistant. Rather than focusing on its underlying theory, we prefer to give a practical overview of the tool. The main motivation of this section is to help the reader to read the excerpts of Coq code that illustrate this Ph.D. thesis.

Coq is a proof assistant, a tool that allows to build mathematical models, to assert properties on them and to build their proofs. It uses the principle of the Curry-Howard isomorphism: a lemma is a type, while a proof is a lambda term of the corresponding type. For Coq, the underlying lambda calculus is named the Predictive Calculus of CoInductive Constructions (abbreviate as pCic). In this calculus all the terms have a type, even the types themselves. The pCic is a formalization of the type theory, including inductive definitions. One of the specificity of pCic is to define the types and the terms in the same syntactical structure.

The Coq proof assistant provides: a programming languages to build models and to assert lemmas, a type checker to mechanically check that all terms are well-type (which through the Curry-Howard isomorphism corresponds also to mechanically check the correctness of a proofs), and finally a tactic language to interactively build a term of a given type (i.e., to build a proof).

We introduce how one can use the Coq proof assistant for the implementation of a provably sound arithmetic verification procedure. This decision procedure implements variables elimination by application of the Fourier-Motzkin Lemma. This verification procedure has been proved useful in the different implementations described in this thesis.

2.2.1 The Arithmetic Formulas

The formulas for arithmetic assertions are composed of two languages. First, there is the numerical expression language, which terms are evaluated to a value through a store (a map from variables to values). This expression language is a component inside the boolean expression language, which terms are evaluated to boolean values through a store. In this section, we describe the definitions of both expression languages, their evaluations, and some of their properties.

The Numerical Expression Language

Before defining the design of the numerical language we need to choose how we model the variables. We have chosen to model the variables as natural number, through the alias `fm_var`. This allows to make a clear difference between the natural number and the variables identifiers. We define a store (alias `fm_store`) as a partial function from the variables to values (Here `Z`, the type for Coq integers).

```
Definition fm_var := nat.
(* part_funct A B, is an alias for the type
     A → option B *)
Definition fm_store := part_funct fm_var Z.
```

As Coq only provides total functions, we built a library that implements partial functions, through the Coq `option` type:

```
Inductive option (A: Type) : Type :
| Some: A → option A
| None: option A.
```

This type is defined with the `Inductive` keyword, that allows to define inductive types. The type `Type` is a base type of Coq (Together with `Set` and `Prop`, which are included in `Type`). The type `option` takes as argument a term `A` of type `Type` (and thus this may be a terms which type is `Set` or `Prop`), and returns a term of type `Type`. It contains two constructors: (1) `Some` (of type `A → option A`), that we use to model a value returned by a partial function, and (2) `None` of type `option A`, that models the fact that the argument is not in the domain of the partial function. The evaluation of the variable `x` through the partial function `f` is simply written in Coq as `(f x)`.

We define the numerical expression language as the `fm_nexpr` inductive type of type `Set` (which will allow us to extract it as a data-structure in Ocaml). This type constructors correspond in order to: a variable, a constant, the addition, the subtraction and multiplication of two numerical expressions. For these later constructors we defined Coq notations, which allow to make the terms more readable.

```
Inductive fm_nexpr : Set :=
| nfm_var : fm_var → fm_nexpr
| fm_ncons: Z → fm_nexpr
| fm_nplus: fm_nexpr → fm_nexpr → fm_nexpr
| fm_nminus: fm_nexpr → fm_nexpr → fm_nexpr
| fm_nmult: fm_nexpr → fm_nexpr → fm_nexpr.

Notation "e1 +e e2" := (fm_nplus e1 e2) (at level 78) : fm_scope.
Notation "e1 -e e2" := (fm_nminus e1 e2) (at level 78) : fm_scope.
Notation "e1 *e e2" := (fm_nmult e1 e2) (at level 78) : fm_scope.
```

We define the evaluation of numerical expressions through a store by the recursive function `fm_neval`. A specificity of Coq is that all its functions must finish (more precisely, the pCic is strongly normalizing). It implies that all recursive functions must be proved to finish. Coq provides several ways to assert this termination. In `fm_neval`, we use a structural recursion (`struct`) over the numerical expression `e`, for which Coq verifies that we recursively call the function on sub-terms (and thus automatically proves that the function will finish). The evaluation of the numerical expression `e` is based on a case analysis over the constructor. Please note that if a variable value is not defined by the store we give it a default value (here 0).

```
Fixpoint fm_neval (e: fm_nexpr) (s: fm_store) {struct e} : Z :=
  match e with
  | nfm_var v ⇒ match (s v) with
                 | None ⇒ 0
                 | Some z ⇒ z
                end
  | fm_ncons z ⇒ z
  | e1 +e e2 ⇒ (fm_neval e1 s) + (fm_neval e2 s)
  | e1 -e e2 ⇒ (fm_neval e1 s) - (fm_neval e2 s)
  | e1 *e e2 ⇒ (fm_neval e1 s) * (fm_neval e2 s)
end.
```

We define an alternative evaluation for the numerical expression, but without any store this time. This evaluation only returns a value if there is no variable in the expression. This behavior is modeled by the type `option Z`:

```
Fixpoint fm_nexpr_compute (e: fm_nexpr) {struct e} : option Z :=
  match e with
    nfm_var x ⇒ None
    | fm_ncons x ⇒ Some x
    | e1 +e e2 ⇒ match fm_nexpr_compute e1 with
                    None ⇒ None
                    | Some e1' ⇒
                      match fm_nexpr_compute e2 with
                        None ⇒ None
                        | Some e2' ⇒ Some (e1' + e2')
                      end
                  end
    | e1 -e e2 ⇒ match fm_nexpr_compute e1 with
                    None ⇒ None
                    | Some e1' ⇒
                      match fm_nexpr_compute e2 with
                        None ⇒ None
                        | Some e2' ⇒ Some (e1' - e2')
                      end
                  end
    | e1 *e e2 ⇒ match fm_nexpr_compute e1 with
                    None ⇒
                    match fm_nexpr_compute e2 with
                      None ⇒ None
```

```
                        | Some e2' ⇒ if Z_eq_dec e2' 0 then Some 0 else None
                      end
                      | Some e1' ⇒
                        if Z_eq_dec e1' 0 then Some 0 else
                          match fm_nexpr_compute e2 with
                            None ⇒ None
                            | Some e2' ⇒ Some (e1' * e2')
                          end
                    end
    end.
```

We can capture the specification of this function property through the lemma `fm_nexpr_compute_correct`. It asserts that if `fm_nexpr_compute` returns some value, then the evaluation through any store will return this value. Here follows the lemma:

```
  Lemma fm_nexpr_compute_correct: ∀ s e z,
    fm_nexpr_compute e = Some z →
    fm_neval e s = z.
```

Lets now take a look at how one can build a proof for this lemma in Coq. First, Coq presents an ongoing proof through a number of goals with their hypothesizes. At the beginning, there is no hypothesizes, and just one goal: the lemma to be proved.

```
    1 subgoal

  ============================
   forall (s : fm_store) (e : fm_nexpr) (z : Z),
   fm_nexpr_compute e = Some z -> fm_neval e s = z
```

This proof can be proved by induction over the universally quantified term `e`, of type `fm_nexpr`. A convenient feature of Coq is that it generates automatically an induction principle through an inductive type definition. To use this induction principle over `e`, we just have to call the `induction e` tactic, which generates the following proof subgoals:

```
    5 subgoals

  s : fm_store
  f : fm_var
  ============================
   forall z : Z,
   fm_nexpr_compute (nfm_var f) = Some z -> fm_neval (nfm_var f) s = z

subgoal 2 is:
 forall z0 : Z,
 fm_nexpr_compute (fm_ncons z) = Some z0 -> fm_neval (fm_ncons z) s = z0
subgoal 3 is:
 forall z : Z,
 fm_nexpr_compute (e1 +e e2) = Some z -> fm_neval (e1 +e e2) s = z
subgoal 4 is:
 forall z : Z,
 fm_nexpr_compute (e1 -e e2) = Some z -> fm_neval (e1 -e e2) s = z
subgoal 5 is:
 forall z : Z,
 fm_nexpr_compute (e1 *e e2) = Some z -> fm_neval (e1 *e e2) s = z
```

We remark that we have now five subgoals, one for each constructor of `fm_nexpr` (the shown hypothesizes are related to the first subgoal). For the first subgoal, we first introduce the hypothesizes by using the `intros` command, which transforms the subgoal into:

```
  s : fm_store
  f : fm_var
```

```
z : Z
H : fm_nexpr_compute (nfm_var f) = Some z
============================
 fm_neval (nfm_var f) s = z
```

A simple look at the definition of `fm_nexpr_compute` informs us that it should return `None` when its argument is a variable (which is the case in the hypothesis `H`). We can compute this value by taping the command `simpl fm_nexpr_compute in H`. The goal is now:

```
s : fm_store
f : fm_var
z : Z
H : None = Some z
============================
 fm_neval (nfm_var f) s = z
```

By the definition of the inductive type `option` the hypothesis `H` is false, because it asserts the equality of two different constructors. Coq can resolve such goal through the `discriminate` command. The second subgoal (for a constant) is trivial, we now focus on the subgoal for the `nplus` constructor:

```
s : fm_store
e1 : fm_nexpr
IHe1 : forall z : Z, fm_nexpr_compute e1 = Some z -> fm_neval e1 s = z
e2 : fm_nexpr
IHe2 : forall z : Z, fm_nexpr_compute e2 = Some z -> fm_neval e2 s = z
============================
 forall z : Z,
 fm_nexpr_compute (e1 +e e2) = Some z -> fm_neval (e1 +e e2) s = z
```

In this subgoal, additional hypothesizes have appeared (namely, `IHe1` and `IHe2`): the induction hypothesizes. They assert that the property we want to prove holds for both subterms of the `nplus` constructor. After simplifying the goal, and introducing the hypothesizes, we have the following goal:

```
s : fm_store
e1 : fm_nexpr
IHe1 : forall z : Z, fm_nexpr_compute e1 = Some z -> fm_neval e1 s = z
e2 : fm_nexpr
IHe2 : forall z : Z, fm_nexpr_compute e2 = Some z -> fm_neval e2 s = z
z : Z
H : match fm_nexpr_compute e1 with
    | Some e1' =>
        match fm_nexpr_compute e2 with
        | Some e2' => Some (e1' + e2')
        | None => None (A:=Z)
        end
    | None => None (A:=Z)
    end = Some z
============================
 fm_neval e1 s + fm_neval e2 s = z
```

Here, we will do a case analysis for each possible values of `fm_nexpr_compute e1` and `fm_nexpr_compute e2` (here either `None`, or `option x`, with x some fresh Coq variable). We can remark that on both the `None` cases, the hypothesis H is invalid. Coq allows to compose the proof commands with a semi-colon. The r.h.s command will be applied to each subgoals generated by the l.h.s command. Here we want all the possible cases for `fm_nexpr_compute e1` and `fm_nexpr_compute e2` and try to solve the subgoals for which `H` is invalid. Thus we enter the following command:

```
destruct (fm_nexpr_compute e1); destruct (fm_nexpr_compute e2); try discriminate.
```

The goal becomes:

```
s : fm_store
e1 : fm_nexpr
z0 : Z
IHe1 : forall z : Z, Some z0 = Some z -> fm_neval e1 s = z
e2 : fm_nexpr
z1 : Z
IHe2 : forall z : Z, Some z1 = Some z -> fm_neval e2 s = z
z : Z
H : Some (z0 + z1) = Some z
============================
 fm_neval e1 s + fm_neval e2 s = z
```

Now, we can rewrite the conclusion of `IHe1` and `IHe2` in the goal. For this we need to give an element of `Z`, as well as a proof for the equality. This is done by the following commands (where `refl_equal` is the only constructor for equality, which corresponds to the reflexivity of the equality):

```
  rewrite (IHe1 z0 (refl_equal (Some z0))).
  rewrite (IHe2 z1 (refl_equal (Some z1))).
```

The resulting goal is:

```
s : fm_store
e1 : fm_nexpr
z0 : Z
IHe1 : forall z : Z, Some z0 = Some z -> fm_neval e1 s = z
e2 : fm_nexpr
z1 : Z
IHe2 : forall z : Z, Some z1 = Some z -> fm_neval e2 s = z
z : Z
H : Some (z0 + z1) = Some z
============================
 z0 + z1 = z
```

We can deduce the goal from the hypothesis `H`. For this purpose, we use the command `injection`. Our goal is simply resolved by the following command:

```
  injection H; intro X; exact X.
```

The other subgoals of the induction are resolved in a similar way.

The function `fm_nexpr_compute` can be used to build a function that will simplify (i.e., reduce the number of constructor if possible) a numerical expression:

```
Fixpoint simpl_fm_nexpr (e: fm_nexpr) : fm_nexpr :=
  match e with
    | nfm_var v ⇒ nfm_var v
    | fm_ncons z ⇒ fm_ncons z
    | e1 +e e2 ⇒
      let e1' := simpl_fm_nexpr e1 in (
        let e2' := simpl_fm_nexpr e2 in (
          match (fm_nexpr_compute e1', fm_nexpr_compute e2') with
            | (Some z1, Some z2) ⇒ fm_ncons (z1 + z2)
            | (_ , Some z2) ⇒
              if Z_eq_dec z2 0 then e1' else e1' +e (fm_ncons z2)
            | (Some z1 , _) ⇒
              if Z_eq_dec z1 0 then e2' else (fm_ncons z1) +e e2'
            | _ ⇒ e1' +e e2'
          end
        )
      )
    | e1 -e e2 ⇒
      let e1' := simpl_fm_nexpr e1 in (
        let e2' := simpl_fm_nexpr e2 in (
          match (fm_nexpr_compute e1’, fm_nexpr_compute e2’) with
            | (Some z1, Some z2) ⇒ fm_ncons (z1 - z2)
            | (_ , Some z2) ⇒
              if Z_eq_dec z2 0 then e1’ else e1’ -e (fm_ncons z2)
            | (Some z1 , _) ⇒ (fm_ncons z1) -e e2’
            | _ ⇒ e1’ -e e2’
          end
        )
      )
    | e1 *e e2 ⇒
      let e1’ := simpl_fm_nexpr e1 in (
        let e2’ := simpl_fm_nexpr e2 in (
          match (fm_nexpr_compute e1’, fm_nexpr_compute e2’) with
            | (Some z1, Some z2) ⇒ fm_ncons (z1 * z2)
            | (_ , Some z2) ⇒
              if Z_eq_dec z2 0 then (fm_ncons 0)
                else (if Z_eq_dec z2 1 then
                  e1’ else e1’ *e (fm_ncons z2))
            | (Some z1 , _) ⇒
              if Z_eq_dec z1 0 then (fm_ncons 0)
                else (if Z_eq_dec z1 1 then
                  e2’ else (fm_ncons z1) *e e2’)

            | _ ⇒ e1’ *e e2’
          end
        )
      )
  end.
```

The correctness of the simplification is stated as the fact that for any store, an expression and its simplification have the same evaluation. This lemma, which is proved by induction over the numerical expressions, is defined below:

```
Lemma simpl_fm_nexpr_correct: ∀ s e,
  fm_neval e s = fm_neval (simpl_fm_nexpr e) s.
```

As previously stated, our arithmetic verification procedure is based on variables elimination. For this purpose we need to be able to factorize a variable in a numerical expression. We implement the function `fm_nexpr_fm_var_fact` that provides this feature.

```
Fixpoint fm_nexpr_fm_var_fact (e: fm_nexpr) (v: nat) struct e : (fm_nexpr * fm_nexpr) :=
 match e with
   | nfm_var x => if eq_nat_dec x v then (fm_ncons 1, fm_ncons 0) else (fm_ncons 0, nfm_var x)
   | fm_ncons z => (fm_ncons 0, fm_ncons z)
   | e1 +e e2 =>
     match (fm_nexpr_fm_var_fact e1 v, fm_nexpr_fm_var_fact e2 v) with
       | ((e11, e12 ), (e21, e22)) =>
         (e11 +e e21, e12 +e e22)
     end
   | e1 -e e2 =>
     match (fm_nexpr_fm_var_fact e1 v, fm_nexpr_fm_var_fact e2 v) with
       | ((e11, e12 ), (e21, e22)) =>
         (e11 -e e21, e12 -e e22)
     end
   | e1 *e e2 =>
     match (fm_nexpr_fm_var_fact e1 v, fm_nexpr_fm_var_fact e2 v) with
       | ((e11, e12 ), (e21, e22)) =>
         (((e11 *e e22) +e (e21 *e e12)) +e ((nfm_var v) *e (e11 *e e21)), e12 *e e22)
     end
end.
```

This function returns two numerical expressions: the first represents the factor for the variable `v`, and the second corresponds to the rest of the numerical expression (in which `v` does not appear). This specification is formally asserted in the `fm_nexpr_fm_var_fact_sem` lemma:

```
Lemma fm_nexpr_fm_var_fact_sem: forall s v e e1 e2,
  fm_nexpr_fm_var_fact e v = (e1, e2) ->
  fm_neval e s = fm_neval ((nfm_var v *e e1) +e e2) s.
```

We eventually build a function, named `fm_nexpr_simpl_fm_var_fact`, which will simplifies both numerical expressions returned by `fm_nexpr_fm_var_fact`.

```
Definition fm_nexpr_simpl_fm_var_fact (n: fm_nexpr) (v: nat) :=
  match (fm_nexpr_fm_var_fact n v) with
    | (e1, e2) => (simpl_fm_nexpr e1, simpl_fm_nexpr e2)
  end.

Lemma fm_nexpr_simpl_fm_var_fact_sem: forall s v e e1 e2,
  fm_nexpr_simpl_fm_var_fact e v = (e1, e2) ->
  fm_neval e s = fm_neval ((nfm_var v *e e1) +e e2) s.
```

The Boolean Expression Language

The boolean expression language is defined, like the numerical expression language, as an inductive type:

```
Inductive fm_bexpr : Set :=
| fm_beq : fm_nexpr → fm_nexpr → fm_bexpr
| fm_blt : fm_nexpr → fm_nexpr → fm_bexpr
| fm_ble : fm_nexpr → fm_nexpr → fm_bexpr
| fm_bgt : fm_nexpr → fm_nexpr → fm_bexpr
| fm_bge : fm_nexpr → fm_nexpr → fm_bexpr
| fm_bneg : fm_bexpr → fm_bexpr
| fm_band: fm_bexpr → fm_bexpr → fm_bexpr
| fm_bor: fm_bexpr → fm_bexpr → fm_bexpr.

Notation "e1 == e2" := (fm_beq e1 e2) (at level 78) : fm_scope.
Notation "e1 << e2" := (fm_blt e1 e2) (at level 78) : fm_scope.
Notation "e1 <<= e2" := (fm_ble e1 e2) (at level 78) : fm_scope.
Notation "e1 >> e2" := (fm_bgt e1 e2) (at level 78) : fm_scope.
Notation "e1 >>= e2" := (fm_bge e1 e2) (at level 78) : fm_scope.
Notation "! e" := (fm_bneg e) (at level 78) : fm_scope.
Notation "e1 //\\ e2" := (fm_band e1 e2) (at level 78) : fm_scope.
Notation "e1 \\// e2" := (fm_bor e1 e2) (at level 78) : fm_scope.
Notation "e1 ==> e2" := (fm_bor (fm_bneg e1) e2) (at level 78) : fm_scope.
```

The function `fm_beval`, evaluates a boolean formula, through a store, as a term of `Prop`: the type for logical propositions in Coq (˜ is the Coq notation for the negation).

```
Fixpoint fm_beval (b: fm_bexpr) (s: fm_store) struct b : Prop :=
  match b with
    | e1 == e2 ⇒ (fm_neval e1 s) = (fm_neval e2 s)
    | e1 << e2 ⇒ (fm_neval e1 s) < (fm_neval e2 s)
    | e1 <<= e2 ⇒ (fm_neval e1 s) <= (fm_neval e2 s)
    | e1 >> e2 ⇒ (fm_neval e1 s) > (fm_neval e2 s)
    | e1 >>= e2 ⇒ (fm_neval e1 s) >= (fm_neval e2 s)
    | ! b1 ⇒ ˜ (fm_beval b1 s)
    | b1 //\\ b2 ⇒ (fm_beval b1 s) ∧ (fm_beval b2 s)
    | b1 \\// b2 ⇒ (fm_beval b1 s) ∨ (fm_beval b2 s)
  end.
```

One of the main specificity of this function is that it is decidable for a given store. Coq provides a useful inductive type for this kind of assertion, `sumbool`:

```
Inductive sumbool (A : Prop) (B : Prop) : Set :=
| left : A → sumbool A B
| right : B → sumbool A B.

Notation "{ A } + { B }" := (sumbool A B).
```

This inductive type can be used to define decidability of some predicate P (by taking P for A and ~P for B). It has mainly two advantages. The first one occurs when building a proof. Indeed, if an hypothesis as the type {P} + {~P}, it can be destructed, creating two subgoals with as hypothesis a proof respectively of P and of ~P. The other advantage is that Coq can automatically extract a decision procedure from a decidability lemma (because the type of sumbool is Set). The decidability for the evaluation of a boolean expression through a store is defined as:

```
Lemma fm_beval_dec: forall s b,
  {fm_beval b s} + {~ fm_beval b s}.
```

2.2.2 The Verification Procedure

The verification procedure is split into several steps. The first one consists in propagating the negation as deep as possible (this kind of term is said to be in negation normal form). As our formulas are quantifier free, it means that we propagate negation until the atomic constructor of the boolean expressions. The second step consists in transforming the formula into its disjunctive normal form. A formula is said to be in disjunctive normal form if it is a disjunction of conjunctions.

Negation Normal Form

In this step, we have decided to keep the boolean expression language bexpr to represent the terms in negation normal form. Yet, all the terms of bexpr are not in this form, so we define an inductive predicate that characterizes which boolean expressions are in negation normal form (in the following the function (fm_bexpr_size e) computes the number of constructors of the boolean expression e):

```
Inductive is_neg_propagate : fm_bexpr → Prop :=

(* atomic boolean expression are valid formulas *)
| fm_beq_is_neg_propagate: ∀ e1 e2, is_neg_propagate (e1 == e2)
| fm_bge_is_neg_propagate: ∀ e1 e2, is_neg_propagate (e1 >>= e2)
| fm_bgt_is_neg_propagate: ∀ e1 e2, is_neg_propagate (e1 >> e2)
| fm_ble_is_neg_propagate: ∀ e1 e2, is_neg_propagate (e1 <<= e2)
| fm_blt_is_neg_propagate: ∀ e1 e2, is_neg_propagate (e1 << e2)

(* if a boolean expression is negative, its size
must be 1, which means that it is an atomic formula*)
| fm_bneg_is_neg_propagate: ∀ e, (fm_bexpr_size e = 1)%nat → is_neg_propagate (! e)

(* formulas on both size of connectives must be valid
formulas *)
| fm_band_is_neg_propagate: ∀ e1 e2,
(is_neg_propagate e1) →
(is_neg_propagate e2) →
(is_neg_propagate (e1 //\\ e2))

| fm_bor_is_neg_propagate: ∀ e1 e2,
(is_neg_propagate e1) →
(is_neg_propagate e2) →
(is_neg_propagate (e1 \\// e2)).
```

We define the function neg_propagate that propagates the negations until the atomic boolean expressions. We also prove a lemma that asserts the preservation of the evaluation, and a lemma asserting the correctness of the function: the fact that all returned formulas are in the negation normal form:

```
Function neg_propagate (b: fm_bexpr) (n: bool) struct b : fm_bexpr := ...

Lemma neg_propagate_preserve: ∀ b n,
   (∀ s, fm_beval (neg_propagate b n) s ↔ fm_beval (if n then (fm_bneg b) else b) s).

Lemma neg_propagate_correct: ∀ b n,
   is_neg_propagate (neg_propagate b n).
```

Disjunctive Normal Form

For the transformation of a boolean formula in disjunctive normal form, we have chosen to define a few new types:

```
Definition constraint := fm_nexpr.
Definition andlist  := list constraint.
Definition orlist := list andlist.
```

There is mainly two reasons for such a design choice. First, these types define by construction disjunctive normal form (a disjunction of conjunction), which avoid the need for a predicate as the one used to characterized boolean expression in negation normal form. Another reason is that the algorithms for variables elimination are simpler to implement through list traversals. Together with these new types definitions, we define their translation into boolean expression:

```
Definition constraint_semantic (c: constraint) : fm_bexpr := (fm_ncons 0) >>= c.

Fixpoint andlist_semantic (l: andlist) : fm_bexpr :=
  match l with
    nil ⇒ btrue
    | hd::tl ⇒ (constraint_semantic hd) //\\ (andlist_semantic tl)
  end.

Fixpoint orlist_semantic (l: orlist) : fm_bexpr :=
  match l with
    nil ⇒ ! btrue
    | hd::tl ⇒ (andlist_semantic hd) \\// (orlist_semantic tl)
  end.
```

Then, we define a function that transforms a boolean expression into its disjunctive normal form, and prove its evaluation preservation lemma:

```
Fixpoint disj_nf (b: fm_bexpr) : orlist := ...

Lemma disj_nf_preserve: ∀ b,
  is_neg_propagate b →
  (∀ s, fm_beval (orlist_semantic (disj_nf b)) s ↔ fm_beval b s).
```

Finally, we implement the function `fm_neval_orlist`, that tries to evaluate an `orlist`, such that its evaluation does not depend on the store. The function signature, and its correctness lemma are presented bellow:

```
Definition fm_neval_orlist (a: orlist) : option bool := ...

Lemma eval_orlist2orlist_semantic: ∀ a b,
  fm_neval_orlist a = Some b →
  (∀ s, fm_beval (if b then orlist_semantic a else ! (orlist_semantic a)) s).
```

Variables Elimination

The main function that allows the elimination of variable is `elim_fm_var_constraint`. This function tries to eliminate a variable by composing two constraints. The function takes four numerical expressions, corresponding to the factorization of the variable in both constraints. Then, it tests the sign of both factors of the variable to know if it can use the Fourrier-Motzkin lemma:

```
Lemma fourier_motzkin_for_integers: ∀ a1 b1 a2 b2 x,
  a1 < 0 →
  0 < a2 →
  0 >= x * a1 + b1 →
  0 >= x * a2 + b2 →
  a1 * b2 >= a2 * b1.
```

Here follows the function signature and its correctness lemma:

```
Definition elim_fm_var_constraint (e11 e12 e21 e22: nexpr): option constraint := ...

Lemma elim_fm_var_constraint_correct: ∀ e11 e12 e21 e22 v c2',
  elim_fm_var_constraint e11 e12 e21 e22 = Some c2' →
  ∀ s,
    fm_beval (constraint_semantic ((nfm_var v *e e11) +e e12)) s →
    fm_beval (constraint_semantic ((nfm_var v *e e21) +e e22)) s →
    fm_beval (constraint_semantic c2') s.
```

We can note that the lemma implies that we have eliminated the variable `v`, as the numerical expression in the goal does not contain occurrences of `v`. This function is important as it is used to eliminate the variables in a `andlist` (implemented in `elim_allfm_var_andlist`) and in a `orlist` (implemented in `elim_allfm_var_orlist`):

```
Definition elim_allfm_var_andlist (l: andlist) : andlist := ...

Lemma elim_allfm_var_andlist_correct: ∀ s l,
  fm_beval (andlist_semantic l) s →
  fm_beval (andlist_semantic (elim_allfm_var_andlist l)) s.

Fixpoint elim_allfm_var_orlist (l: orlist) struct l: orlist := ...

Lemma elim_allfm_var_orlist_correct: ∀ s l,
  fm_beval (orlist_semantic l) s →
  fm_beval (orlist_semantic (elim_allfm_var_orlist l)) s.
```

2.2.3 Put It All Together

Thanks to all the previously described functions we are able to implement the arithmetic verification procedure:

```
Definition fm_dp (b: fm_bexpr) : bool :=
  match fm_neval_orlist (elim_allfm_var_orlist (disj_nf (neg_propagate true b))) with
    | Some res ⇒ negb res
    | _ ⇒ false
  end.
```

This function computes the negation of a given boolean expression, computes its disjunctive normal form, tries to eliminate all the variables, and finally computes the evaluation over all possible stores. The validity of the original boolean expression is hence the negation of this evaluation. We provide a soundness lemma, asserting that if the verification procedure return true, then the boolean expression is valid for any store (this lemma is proved using all the preservation and correctness lemmas previously presented):

```
Lemma fm_dp_correct: ∀ b,
  fm_dp b = true →
  ∀ s,
    fm_beval b s.
```

2.2.4 The Coq Tactic

Our verification procedure only reason over the `fm_bexpr` boolean expression language. However, we would like to use it to solve Coq goals. Hence we need to build a set of tactics that will transform a Coq assertion into a boolean expression. More precisely, we build a boolean expression that is equivalent to the goal, and prove that its evaluation (it translation into the Coq language) implies the goal. Here follows the main tactic for proving a Coq arithmetic goal using our verification procedure:

```
Ltac fm_dp_decision :=
  match goal with
    | |- ?G =>
      let l := (Build_env G) in (
      let x := (To_bexpr G l) in (
      new_cut (fm_beval x (list2fm_store (rev l))); [
        eapply fm_dp_correct; vm_compute; apply refl_equal
        |
        simpl; intuition
      ]
    )
  )
end.
```

The tactics `Build_env` builds a map that matches the Coq variables to variables for numerical expression (more precisely, that matches the Coq variables to a natural). The `To_bexpr` tactic translates the Coq arithmetic goal into a boolean expression, using the previously constructed map. Then we assert that the evaluation of this boolean expression implies the Coq goals. This assertion produced two subgoals. One is for the logical implication, and the other is for a proof that the evaluation of the boolean expression is correct. To build this proof, we apply the soundness lemma of our verification procedure (`fm_dp_correct`), transforming the goal into the assertion that the function `fm_dp`, with for argument the boolean expression (build by `To_bexpr`) evaluates to `true`.

Example

We therefore present an illustrating example of how work our tactic. Let consider the following Coq goal:

```
============================
 forall x y z res : Z,
 res = x /\ x >= y /\ x >= z -> res >= x /\ res >= y /\ res >= z
```

We first introduce the variables as hypothesizes (using the command `do 4 intro`):

```
x : Z
y : Z
z : Z
```

```
    res : Z
    ============================
    res = x /\ x >= y /\ x >= z -> res >= x /\ res >= y /\ res >= z
```

We propose to apply a customized version of our tactic, where the subgoals that it generates are keep untouched (using the `idtac` command):

```
  match goal with
    | |- ?G =>
      let l := (Build_env G) in (
        let x := (To_expr_b G l) in (
          new_cut (fm_beval x (list2fm_store (rev l))); [
            idtac (* do nothing for this subgoal *)
            |
            idtac (* do nothing for this subgoal *)
          ]
        )
      )
  end.
```

Applying this tactics generates the following Coq proof obligations:

```
  x : Z
  y : Z
  z : Z
  res : Z
  ============================
   fm_beval
     (((nfm_var 1%nat == nfm_var 3%nat) //\\
       ((nfm_var 3%nat >>= nfm_var 2%nat) //\\
        (nfm_var 3%nat >>= nfm_var 0%nat))) ==>
      ((nfm_var 1%nat >>= nfm_var 3%nat) //\\
       ((nfm_var 1%nat >>= nfm_var 2%nat) //\\
        (nfm_var 1%nat >>= nfm_var 0%nat))))
     (list2fm_store (rev (z :: res :: y :: x :: nil)))

subgoal 2 is:
 fm_beval
   (((nfm_var 1%nat == nfm_var 3%nat) //\\
     ((nfm_var 3%nat >>= nfm_var 2%nat) //\\
      (nfm_var 3%nat >>= nfm_var 0%nat))) ==>
    ((nfm_var 1%nat >>= nfm_var 3%nat) //\\
     ((nfm_var 1%nat >>= nfm_var 2%nat) //\\
      (nfm_var 1%nat >>= nfm_var 0%nat))))
   (list2fm_store (rev (z :: res :: y :: x :: nil))) ->
 res = x /\ x >= y /\ x >= z -> res >= x /\ res >= y /\ res >= z
```

We therefore explain in details the resolutions of both subgoals.

First Subgoal If we apply the verification procedure soundness lemma we obtain the following goal:

```
    x : Z
  y : Z
  z : Z
  res : Z
  ============================
   fm_dp
     (((nfm_var 1%nat == nfm_var 3%nat) //\\
       ((nfm_var 3%nat >>= nfm_var 2%nat) //\\
        (nfm_var 3%nat >>= nfm_var 0%nat))) ==>
      ((nfm_var 1%nat >>= nfm_var 3%nat) //\\
       ((nfm_var 1%nat >>= nfm_var 2%nat) //\\
        (nfm_var 1%nat >>= nfm_var 0%nat)))) = true
```

Here we just have to compute the result of the `fm_dp` function. This is achieved by applying the `compute` command, which transforms the goal in the following tautology:

```
x : Z
y : Z
z : Z
res : Z
============================
 true = true
```

Second Subgoal We can compute the result of the `fm_eval` by using the `simpl` command:

```
x : Z
y : Z
z : Z
res : Z
============================
~ (res = x /\ x >= y /\ x >= z) \/ res >= x /\ res >= y /\ res >= z ->
res = x /\ x >= y /\ x >= z -> res >= x /\ res >= y /\ res >= z
```

This kind of first order goals is easily proved by the `intuition` tactic.

Related Work

We introduced the use of the Coq proof assistant through an illustrating example: the implementation of a provable sound arithmetic verification procedure. There exists several decision procedures for arithmetic in Coq. One of them, named MicroMega [60], provides also a tactic implemented by reflection. However this decision procedure does not deal with most of the propositional operators ($\wedge$, $\vee$ and ~). From version 8.1, Coq includes by default the tactic `romega` [61]. The originality of this work is that it uses an external prover. This program constructs a certificate, which is given back to a checker written and proved correct in Coq. Hence, this tactic is fast (much of the computation is external to Coq), yet generates small proof term (as the verification is the result of the execution of the checker). The only limitation of this approach is that one cannot extract a certified decision procedure in Ocaml, as it can be done with our implementation. Finally, Coq provides a tactic to resolve arithmetic subgoals over natural (named `omega`). Yet this tactics is not implemented by reflection, but is instantiated by a collection of Coq tactics. The main issue of this design choice is that the build proof terms are often huge. A different decision procedure, namely the Cooper algorithm, has been proved sound and complete by Amine Chaieb et al., inside the Isabelle proof assistant [30].

2.3 The Topsy Operating System Through its Model

Model-checking has proved effective to verify low-level properties on isolated parts of operating systems such as scheduling algorithms or implementations of inter-process communications (IPCs) [5, 4, 6]. In such situations, the relevant implementation is well-localized in the source code, and the modeling language usually lends itself very well to formal paraphrase.

However, there are high-level properties of operating systems that require modeling of various parts of the implementation. For example, *task isolation*, the property that user threads cannot access kernel memory [7], requires modeling of thread management, memory management, hardware protection mechanisms, etc. It is possible to break the verification of such high-level properties that span the whole source code into smaller verifications on well-localized parts of the source code. However, this approach naturally augments the number of specifications and it introduces the risk of making conflicting model assumptions. Ideally, one would prefer a single model, abstract enough to be refined at will, and that would lend itself easily to verification of several, possibly orthogonal properties.

In this section, we show how to build in the Spin model-checker [8] a model of the Topsy operating system [10] that covers most parts of the implementation, thus enabling verification of high-level properties such as task isolation. The main difficulty of building such a global model is the trade-off between exhaustivity and tractability: too fine-grained abstractions would irremediably lead to state-space explosion. In our model, we provide abstractions to deal with several aspects of operating systems, such as scheduling, IPCs, memory management, hardware interface, and user applications. We show experimentally that these abstractions enable verifications of several, both low-level and high-level properties, such as memory protection and kernel-data consistencies for scheduling and message passing.

This chapter is organized as follows. In Sect. 2.3.1, we describe the Topsy operating system together with its Spin model. In Sect. 2.3.2, we discuss the specification and verification of several properties, including task isolation. In Sect. 2.3.3 we show how one can use the model to find which part of the kernel are important for task isolation. Finally, in Sect. 2.3.4, we compare with related work.

2.3.1 Model

We explain in detail our model in a modular way: multi-threading in Sect. 2.3.1, scheduling in Sect. 2.3.1, hardware in Sect. 2.3.1, memory in Sect. 2.3.1, IPCs and system calls in Sect. 2.3.1. For each part, we first give a general description with possibly specificities for Topsy, then we describe their model in Spin.

Multi-threading

Definitions A *thread* is a control-flow implemented by a piece of code and described by a data structure called *thread descriptor*. A thread descriptor contains an id that identifies uniquely the thread, a set of levels of privileges (to control resource access and scheduling), a context (the state of the thread, i.e., the values of the local variables and the last instruction executed), and status information (for scheduling and communication).

A multi-threaded operating system is the control-flow resulting of the interleaved execution of threads. A special thread, called the *interrupt handler*, manages the scheduling and the interaction between threads; it is activated in-between the execution of any other two threads.

Spin Model We model threads by Spin processes, whose interleaved execution is provided natively by Spin. In this setting, each thread is given a unique Spin process id, and its context is modeled by the state of the Spin process. A thread descriptor is represented by a Spin data-structure composed of the Spin id (and therefore the associated context), the execution privilege, scheduling information and a message queue for communication:

```
typedef Thread_desc {
  byte contextPtr;
  bool privilege;
  SchedulerInfo schedInfo;
  MessageQueue msgQueue;
};
```

In order to control the interleaving, we use the special Spin keyword `provided`, that specifies a condition under which a process is executed or not. For example, the kernel service of Topsy in charge of IOs is modeled as follows:

```
proctype ioThread()
provided (_curr_ctxt == _pid)  ... ;
```

where `_curr_ctxt` is the Spin id of the currently running operating system thread, and `_pid` is the Spin id of the currently executing Spin process (this is a variable natively provided by Spin).

Although Spin has a native feature to spawn a process, it does not allow to end it dynamically: a process must reach its last statement to terminate. In order to model termination of user threads (kernel threads are never killed), we add a clause to the `provided` condition that allows for a killed process to execute to its end:

```
proctype uThread()
provided (_curr_ctxt==_pid || _killed[_pid])
 ... ;
```

Scheduling

Definitions Scheduling is the operation by which the interrupt handler chooses the next thread to be run. This decision is based on priorities associated with threads. There is a wide variety of algorithms for this purpose.

The Topsy kernel implements a priority-based round-robin scheduling, by which the highest-priority ready-thread is always chosen. The scheduler uses queues to store the threads according to their status (`RUNNING`, `READY`, or `BLOCKED`). In addition, threads in the `READY` queue are sorted by their priority: "kernel" > "user" > "idle" (for a special idle thread executed when no thread is ready).

Spin Model The algorithm implemented in the model obeys the same priority rules as in Topsy. For this purpose, we use the scheduling status and priority that are stored into the `schedInfo` field of thread descriptors. The scheduling decision is stored in a global variable `_curr_id` that indexes a thread descriptor, from which one can retrieve the corresponding context (a Spin id).

There is a small difference with the original algorithm: we use a traversal of thread descriptors instead of queues, so that the scheduling is not fair anymore. This is not a problem for the properties we verify in this paper, because they are not related to the scheduling policy. However, in order to verify fairness properties, one would need to re-implement the scheduler with queues.

Hardware

Definitions The hardware consists essentially of a processor that provides execution of a single thread and accesses to resources such as segments of memory. To control the accesses to resources, the processor provides privileges (usually, operating systems privileges map the hardware privileges). In particular, the execution privilege level of the currently running process is always kept as a part of the context (usually in a register).

To enable interleaving of executions, the processor provides *interrupts*: a mechanism that puts a value into a special register and triggers switching of threads. Usually, the hardware includes an internal clock that can be used by the operating system to switch a thread after a predetermined execution period.

Spin Model To provide exclusive execution, we model the program-counter register of the processor by the global variable `_curr_ctxt`, that contains the Spin id of the current thread. In order to keep track of the current privilege in our model, there is a global variable called `CPU_MODE`; it is set by the context switch to the execution privilege level of the running thread.

To switch from one process to another, we use the `provided` clause: the execution of a given thread is triggered by changing the value of `_curr_ctxt` to the appropriate Spin id.

To model interrupts, we use a special channel to store the nature of the interrupt before context switching. For example, any thread can raise a software interrupt by sending a software interrupt message on this special channel and switch the currently running process. In contrast, IOs interrupt are raised by external Spin process (representing some piece of hardware). In particular, the internal clock is model by a Spin process that non-deterministically sends a time interrupt and makes the kernel process to be executed.

Memory

Definitions For security reasons, processors allow to partition memory into independent regions associated with a memory access privilege level. When a thread attempts a memory access, the processor compares its execution privilege with the access privilege of the corresponding region. If the execution privilege is equal or higher, the access is granted, otherwise the processor stops the execution and raises an interrupt.

To use its memory efficiently, a kernel usually appeals to dynamic memory allocation. For example, dynamic memory allocation is used for dynamic creation and destruction of threads. Typically, such an allocator maintains a partition of free and allocated blocks inside the kernel data memory.

Since Topsy uses only one multi-threaded user application, the memory is split in two regions to separate the kernel from the user application.

Spin Model We model the memory access mechanism by (1) an array that associates each region with its privilege level and (2) a set of macros that models memory accesses. At each memory access, these macros check the current mode processor (`CPU_MODE`). If the access is allowed, the Spin assignments are executed, otherwise a memory-fault interrupt is raised and the context is switched.

In order to model a memory allocator that manipulates several types of data structures, we provide a macro that creates an instance of a specialized memory allocator for a given data-structure:

```
#define HL(type,size,data,used,hmlock,
hmInit,hmAlloc,hmFree)
type data[size];
bool used[size];
```

```
chan hmlock = lock;
inline hmInit(i)  ... ;
inline hmAlloc(return)  ... ;
inline hmFree(return)  ... ;
```

For illustration, the memory allocator of the kernel is instantiated by:

```
HL(Thread_desc, MEMBLOCK_N, mem, used, hmlock,
hmInit, hmAlloc, hmFree)
```

The effect of the macro expansion above is to build an array `mem` of thread descriptors, an array `used` to keep track of which thread descriptors are free or allocated, and a set of functions `hmInit`, `hmAlloc` and `hmFree`. The initialization function `hmInit` sets all the data-structures to `Free`. The allocation function `hmAlloc` tries to find a free data-structure, declares it `Allocated` and returns its index. The deallocation function `hmFree` sets a data-structure of a given index to `Free`.

Kernel Services

IPCs

Definitions IPCs are a message passing mechanism that allows threads to communicate with each other. To use this mechanism, a thread sets its register to appropriate values (id of the receiver/sender, address of a buffer where the body of the message is stored or have to be stored), and then raises a software interrupt. This interrupt switches the context of the thread with the context of the interrupt handler. The latter routes the message, makes a scheduling decision and finally restores the running thread by switching the context.

Spin Model A global channel is used to pass arguments from a thread to the interrupt handler, whereas the response is sent back to the thread through a private channel whose pointer is passed as the `reply` argument of the message receiving and sending functions:

```
inline recvmsg(from, smsg, reply)  ... ;
inline sndmsg(to, smsg, reply)  ... ;
```

These two functions make use of the `msgQueue` field of the thread descriptor to store messages and communication status information. Let us explain in more details their implementations.

When a thread wants to receive a message, the interrupt handler looks into its message queue. If an adequate message is present, it is dequeued and sent back to the thread, otherwise the thread is blocked and declared waiting for a message, and the IPC arguments are saved. Concretely, the interrupt handler sets the thread to a status called `WAITING` (`msgPendingStatus` field of `msgQueue`), saves the expected sender id (`threadIdPending` field) and the pointer of the channel where to send back the message (`msgPendingPtr` field).

When a thread wants to send a message, the interrupt handler tries to find the receiver. If it is not an existing thread, the IPC fails, otherwise the interrupt handler checks whether the receiving thread is waiting for this message. If this is the case, the message is sent directly to the thread which is unblocked, otherwise it is inserted into the thread message queue `msgQueue`.

System Calls

Definitions In Topsy, system calls (such as threads creation and destruction) are provided by kernel threads. More precisely, a thread makes a system call by sending a request to the appropriate kernel threads via an IPC message, whose body contains the name of the system call and its arguments. In the same way, the result is sent back into a message.

Spin Model The kernel threads are defined as Spin processes, and therefore are managed like the other threads. Yet, they belong to the kernel code and hence can manipulate directly its data. These threads are implemented as infinite loops which receive and parse a message, execute the system call code, and send back a response.

Our model implements the thread manager (responsible for creation and deletion of threads), the IO manager (that act as a directory for IO drivers), and the network manager (network IO driver) but not the memory management kernel thread (whose purpose is to manage pages of virtual memory, but Topsy v2 uses a flat memory model and there is no therefore no implementation yet).

2.3.2 Experiments

We present several verifications done on the Topsy model described in the previous section: "status correctness" is a low-level property that only deals with the scheduler, "status consistency" and 'reply consistency" are high-level properties that deal with both the scheduler and the IPCs, and "task isolation" is a high-level property that deals with execution privilege and memory management.

A Generic Test Program

Verifications are done using a test program. This test program is an echo server that is generic in the sense that it uses all the kernel services provided by the model of the Topsy thread manager and IO manager (see Sect. 2.3.1).

The main thread of the echo server repeatedly does the following: it tries to create a child-thread using the Topsy thread manager and waits for a message by which the child-thread indicates it is about to terminate. The child-thread does the following. When it starts, it tries to open a network connection by sending a request to the Topsy IO manager; the response it gets is the id of a kernel thread managing the network. Via this kernel thread, the child-thread eventually receives a network packet, sends it back (the echo service), and sends a closing message to the IO manager. Then, it sends a message to its father indicating it will make an exit system-call.

Status Correctness

The status correctness property states that the kernel always restores the thread that has been scheduled. Put formally, "whenever a thread is executing (`threadrun` assertion below), its scheduling status is `RUNNING` (`runningthreadcurr` assertion)":

```
#define threadrun
(_curr_ctxt == (mem[curr_id].contextPtr) &&
_syst_run)
```

```
#define runningthreadcurr (used[curr_id] &&
mem[curr_id].schedInfo.status == RUNNING)

[](threadrun -> runningthreadcurr)
```

Status Consistency

This property states that a thread waiting for a message can never be scheduled. Put formally, "if a thread is waiting for a message (`waitingthread` assertion below), its scheduling status must be `BLOCKED` (`blockedthread` assertion)":

```
#define waitingthread (used[ut_init_id] &&
mem[ut_init_id].msgQueue.msgPendingStatus ==
WAITING)

#define blockedthread (used[ut_init_id] &&
mem[ut_init_id].schedInfo.status == BLOCKED)

[]((waitingthread && threadrun) ->
blockedthread)
```

In this specification, `threadrun` ensures that this is not the interrupt handler that is executing. This is essential to distinguish this situation because the interrupt handler precisely may break this property when updating the thread status.

The Topsy thread id of the user thread is hard-wired (`ut_init_id` variable). This is not a limitation of our approach because in Spin it is always possible to construct by hand a never-claim with an implicit quantification over a range of thread ids. However, this is not directly expressible in LTL.

Reply Consistency

The reply consistency property states that the return channel for an IPC is not changed until the thread is unblocked and the expected message is sent. Put formally, "if a thread is waiting for a message, the value of its return channel (field `msgPendingPtr` below) does not change until the thread is `NOT_WAITING`":

```
#define notwaitingthread (used[ut_init_id] &&
mem[ut_init_id].msgQueue.msgPendingStatus ==
NOT_WAITING)

#define pendingPtrval (used[ut_init_id] &&
mem[ut_init_id].msgQueue.msgPendingPtr ==
_reply_chan[ut_init_id])

[](waitingthread ->
(pendingPtrval U notwaitingthread))
```

Task Isolation

The task isolation property is important for operating system verification because it implies that only the kernel can change its data. For a multi-threaded OS such as Topsy (only one user application), the task isolation means that whenever a user thread is running, it must not have a privilege level that grants him access to the memory of the kernel. Put formally, "whenever a user thread is running (`userthreadrun` assertion below), the current privilege does not allow access to kernel memory (`kernelaccess` assertion)":

```
#define kernelaccess
((CPU_MODE == segment[0]) || kernelmode)
```

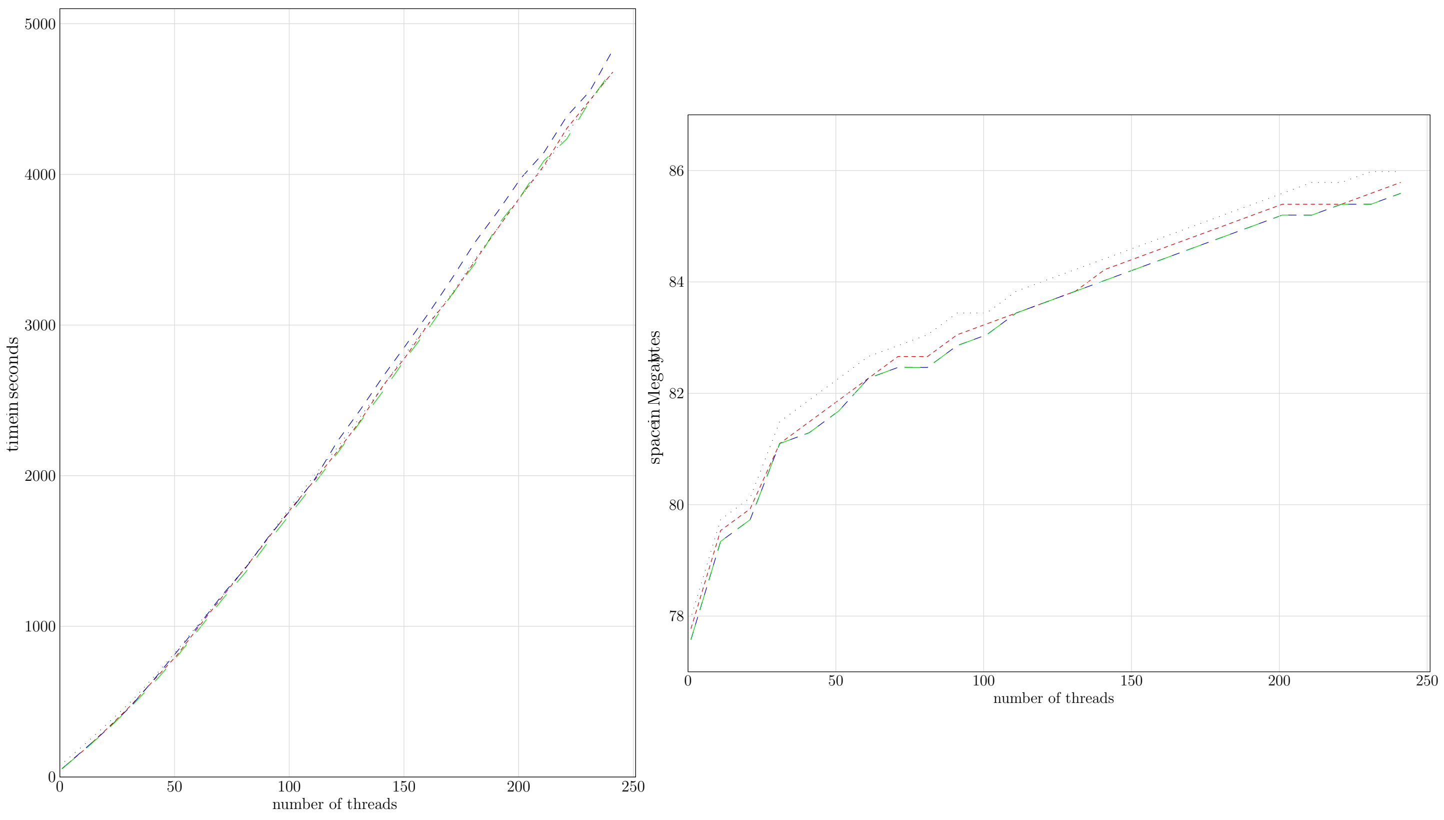


Figure 2.5: Time and Space consumption of SPIN verifications

```
#define userthreadrun
(_curr_ctxt == (mem[curr_id].contextPtr) &&
_user_thread[curr_id] && _syst_run)

[](userthreadrun -> !kernelaccess)
```

Results

Despite its completeness, the size of the model is reasonable: 770 lines of code for Topsy and 80 lines for the echo server (the whole Spin development is available online [11]).

We measure[1] resource consumption in function of the number of child-threads created by the echo server for the properties of the previous section. We observe that, for all properties, the time consumption (Fig. 2.5) is linear, but that the space consumption is logarithmic. This came from the fact that all the child threads share the same behavior, leading SPIN to optimize the memory usage.

[1]Experiments done on an Opteron (64-bit) 2.4GHz machine with 16GB of RAM.

2.3.3 Discussion

Model checking is a tool useful to find problems in system design. In our model, the fact that the task isolation property holds for the model does not prove that it holds for the concrete operating system. In this condition, one can obviously question the pertinence of this verification. We claim that this verification on the abstraction model is interesting because it indicates that, at least, the concrete code should have the same behavior as its abstraction. Although these behaviors are not formally written as specifications, we can intuitively states some of them, thanks to the model of Topsy that gives us a readable image of the system. For instance, we could say that the memory allocation function should return a fresh slot of memory.

However, the question that we can ask ourselves is: whatever if the memory allocator does not fulfill its specification? Our model allows us to modify the code of the memory allocator by injecting some bugs, for instance: a memory allocation function that returns a random slot of memory. On this updated model, we rerun the verification of the task isolation, which does not hold anymore (SPIN gives back a trace that illustrates the corruption of the property). Thus, we can conclude that the fact that memory allocation returns fresh memory is a necessary condition for the task isolation.

Finally, the model makes evident that the context switching may be a sensible part of the system. Indeed, if the privilege is wrongly saved or restored, the task isolation property does not hold anymore. Once again, a modification of the model allows to prove that the correctness of the context switching is necessary for the task isolation.

All this observations motivate the different verifications we have done in this thesis. First, we implement inside the Coq proof assistant a library to prove the specification of C-like source code. This implementation is used to prove the correctness of the memory allocator. As the thread creation is also written in C, we could also use this library to verify it. However, we can observe that this snippet of code is straightforward. This motivates the implementation of a certified verifier for such code. In order to verify the context switching, we have implemented a variant of our library for an assembly language. Finally, the fact that we verify both C-like and assembly code arises another question: how one can compose the verification for snippets in both languages when they are all linked together inside a program, as it is the case in operating systems. For this purpose, we implement a certified translator, that transforms C-like language programs into assembly programs, preserving the semantics. This translator compiles and links snippets into machine level programs, and allows to reason about their specifications.

2.3.4 Related Work

Contrary to our work, all Spin-based verification of operating systems focus on only one property. In consequence, the proposed models are specification-oriented and does not enable verification of high-level properties.

In [4], the inter-task communication facility of the RUBIS micro-kernel is modeled to build test programs. Verifications check properties such as consistency of flags or validity of the status, similarly to “status correctness” and “status consistency” we checked in Sect. 2.3.2.

In [5], the Fluke kernel is modeled by a set of macros and used in several test programs. These programs are decorated with assertions checking the return values of kernel functions. Although the model spans a wide part of the kernel, it does not include any modeling of the hardware.

In [6], the VFiasco IPCs are modeled to verify the communication mechanism of the kernel. The approach is to translate directly the source code in Spin. The modeled scenario is composed of two communicating threads, modeled as Spin processes. All the tests focus on the consistency of flags used by the communication mechanism.

Chapter 3

C-like Verification

Our goal is to verify that the implementation of the Topsy heap manager is "correct". By correct, we mean that the heap manager provides the intended service: the allocation function allocates large-enough memory blocks, these memory blocks are "fresh" (they do not overlap with previously allocated memory blocks), the deallocation function turns the status of blocks into free (except for the terminal block), and the allocation and deallocation functions do not behave in unexpected ways (in particular, they do not modify neither previously allocated memory blocks nor the rest of the memory). Guaranteeing the allocation of fresh memory blocks and the non-modification of previously allocated memory blocks is a necessary condition to ensure that the heap manager preserves exclusive usage of allocated blocks.

Our approach is to use separation logic to formally specify and mechanically verify the goal informally stated above. We choose separation logic for this purpose because it provides a native notion of pointer and memory separation that facilitates the specification of heap-lists. Another advantage of separation logic is that it is close enough to the C language to enable systematic translation from the original source code of Topsy.

The verification of the heap manager of an existing operating system is a difficult task because it is usually written in a low-level language that makes use of pointers, and it is usually not written with verification in mind. For these reasons, the verification of dynamic memory allocation is sometimes considered as a challenge for mechanical verification [23].

The chapter is organized as follows. In Sect. 3.1, we explain how we encode it in Coq. In Sect. 3.2.1 we present an overview of the Topsy heap manager. In Sect. 3.2.2, we formally specify and prove the properties of the underlying data structure used by the heap manager, and we formally specify and explain the verification of the functions of the heap manager. In Sect. 3.3, we discuss practical aspects of the verification such as automation and translation from the original C source code, as well as the output of our experiment: in particular, issues and bugs found in the original source code of the heap manager. In Sect. 3.4, we comment on related work.

3.1 Separation Logic in Coq

3.1.1 Programming Language

The programming language of separation logic is imperative. The current state of execution is represented by a pair of a store (that maps local variables to values) and a heap (a finite map from locations to values). We have an abstract type `var.v` for variables (ranged over by `x`, `y`), a type `loc` for locations (ranged over by `p`, `adr`), and a type `val` for values (ranged over by `v`, `w`) with the condition that all values can be seen as locations (so as to enable pointer arithmetic). Our implementation is essentially abstracted over the choice of types, yet, in our experiments, we have taken the native Coq types of naturals `nat` and relative integers `Z` for `loc` and `val` so as to benefit from better automation. Stores and heaps are implemented by two modules `store` and `heap` whose types are (excerpts):

```
Module Type STORE.
Parameter s : Set. (* the abstract type of stores *)
Parameter lookup : var.v → s → val.
Parameter update : var.v → val → s → s.
End STORE.

Module Type HEAP.
Parameter l : Set. (* locations *)
Parameter v : Set. (* values *)
Parameter h : Set. (* the abstract type of heaps *)
Parameter emp : h. (* the empty heap *)
Parameter singleton : l → v → h. (* singleton heaps *)
Parameter lookup : l → h → option v.
Parameter update : l → v → h → h.
Parameter union : h → h → h.       Notation "h1 ∪ h2" := (union h1 h2).
Parameter disjoint : h → h → Prop.  Notation "h1 ⊥ h2" := (disjoint h1 h2).
End HEAP.

Definition state := prod store.s heap.h.
```

To paraphrase the implementation, `(store.lookup x s)` is the value of the variable `x` in store `s`; `(store.update x v s)` is the store `s` in which the variable `x` has been updated with the value `v`; `(heap.singleton p v)` is a heap composed of a unique cell of location `p`, containing the value `v`; `(heap.lookup p h)` is the contents (if any) of location `p`; `(heap.update p v h)` is the heap `h` in which the location `p` has been mutated with the value `v`; `h ∪ h'` is the disjoint union of `h` and `h'`; and `h ⊥ h'` holds when `h` and `h'` have disjoint domains.

The programming language of separation logic manipulates arithmetic and boolean expressions that are evaluated w.r.t. the store. They are encoded by the inductive types `expr` and `expr_b` (the parts of the definitions which are not essential to the understanding of this paper are abbreviated with "...."):

```
Inductive expr : Set :=
var_e : var.v → expr
| int_e : val → expr
| add_e : expr → expr → expr        Notation "e1 '+e' e2" := (add_e e1 e2).
...
Definition null := int_e 0%Z.
Definition nat_e x := int_e (Z_of_nat x).
Definition field x f := var_e x +e int_e f.
Notation "x '-.>' f " := (field x f).
Inductive expr_b : Set :=
eq_b : expr → expr → expr_b       Notation "e == e'" := (eq_b e e').
| neq_b : expr → expr → expr_b      Notation "e =/= e'" := (neq_b e e').
| and_b : expr_b → expr_b → expr_b  Notation "e &&& e'" := (and_b e e').
| gt_b : expr → expr → expr_b       Notation "e >> e'" := (gt_b e e').
...
```

There is an evaluation function `eval` such that `(eval e s)` is the result of evaluating the expression `e` w.r.t. the store `s`.

The commands of the programming language of separation logic are also encoded by an inductive type:

```
Inductive cmd : Set :=
assign : var.v → expr → cmd       Notation "x <- e" := (assign x e).
| lookup : var.v → expr → cmd       Notation "x '<-*' e" := (lookup x e).
| mutation : expr → expr → cmd      Notation "e '*<-' f" := (mutation e f).
| seq : cmd → cmd → cmd             Notation "c ; d" := (seq c d).
| while : expr_b → cmd → cmd
| ifte : expr_b → cmd → cmd → cmd   Notation "'ifte' b 'thendo' c 'elsedo' c"
...                                   := (ifte b c d).
```

From this presentation, we omit the memory allocation and deallocation commands of separation logic, as they are not useful for our use-case precisely because we verify the implementation of a memory allocation facility. Indeed, such piece of software intend to implement allocation and deallocation not as primitives, but as a set of functions.

The operational semantics of the programming language of separation logic is defined by the following inductive type. An object of type `(exec s c s')` represents the execution of the command `c` from state `s` to state `s'`. Because heaps are finite maps, lookup and mutation may fail; to take this possibility into account, we use an option type.

```
Inductive exec : option state → cmd → option state → Prop :=
exec_assign : ∀ s h x e,
exec (Some (s, h)) (x <- e) (Some (store.update x (eval e s) s, h))
| exec_lookup : ∀ s h x e p v,
val2loc (eval e s) = p → heap.lookup p h = Some v →
exec (Some (s, h)) (x <-* e) (Some (store.update x v s, h))
| exec_lookup_err : ∀ s h x e p,
val2loc (eval e s) = p → heap.lookup p h = None →
exec (Some (s, h)) (x <-* e) None
| exec_mutation : ∀ s h e e' p v,
val2loc (eval e s) = p → heap.lookup p h = Some v →
exec (Some (s, h)) (e *<- e') (Some (s, heap.update p (eval e' s) h))
| exec_mutation_err : ∀ s h e e' p,
val2loc (eval e s) = p → heap.lookup p h = None →
exec (Some (s, h)) (e *<- e') None
...
```

3.1.2 Assertions and Reynolds' Axioms

Assertions of Hoare logic are predicate calculus formulas with the same expressions as the programming language. In consequence, the validity of an assertion depends on the current execution state of the program. There are mainly two ways to encode the semantics of such assertions in a proof assistant:

1. *Deep encoding*: define a syntax for assertions and a satisfaction relation between states and assertions.

2. *Shallow encoding*: identify formulas with functions from states to some "boolean type".

The advantage of shallow encoding over deep encoding is that deciding the validity of formulas becomes a function computation, for which the proof assistant provides native facilities (for example, tactics to prove tautologies).

We have developed a shallow encoding of separation logic in Coq. For this purpose, we identify assertions of separation logic with functions from states to `Prop`, the native type for predicate calculus formulas. For example, `True:Prop` represents truth and `∧:Prop → Prop → Prop` represents classical conjunction in Coq. This gives rise to the type `assert` below. By way of example, we also show the encoding of truth and conjunction in separation logic.

```
Definition assert := store.s → heap.h → Prop.
Definition TT : assert := fun s h => True.
Definition And (P Q:assert) : assert := fun s h => P s h ∧ Q s h.
```

Assertions of Separation Logic The assertion that holds for empty heaps is defined by testing whether the heap is empty:

```
Definition emp : assert := fun s h => h = heap.emp.
```

`e ↦ e'` is the formula that holds for a singleton heap whose only location is the result of evaluating `e` and this location has for contents the result of evaluating `e'`:

```
Definition mapsto e e' s h := ∃ p,
  val2loc (eval e s) = p ∧ h = heap.singleton p (eval e' s).
Notation "e1 ↦ e2" := (mapsto e1 e2).
```

For example, `(var_e x ↦ int_e 4)` asserts that the variable `x` points to a cell that contains the integer 4. The following derived definitions will prove useful later: `e ↦ _` asserts that the cell `e` has some undefined contents, and `e ⤇ l` asserts that there is a list `l` of contiguous cell contents starting from `e`.

The separating conjunction `P ** Q` holds for a heap that can be decomposed into two disjoint heaps for which `P` and `Q` respectively hold:

```
Definition con (P Q:assert) : assert := fun s h =>
  ∃ h1, ∃ h2, h1 ⊥ h2 ∧ h = h1 ∪ h2 ∧ P s h1 ∧ Q s h2.
Notation "P ** Q" := (con P Q).
```

For example, `(var_e x ↦ nat_e p) ** (nat_e p ↦ int_e 2)` is the formal version of the example given in the beginning of this section.

The separating implication `P -* Q` is less intuitive. It is used to represent logically mutations. In particular, the idiom `(e ↦ _ ** (e ↦ e' -* P))` holds for a heap such that the mutation of location `e` to contents `e'` leads to a heap that satisfies `P`. Section 3.2.2 gives a concrete example of such a formula together with its utilization. For the time being, we limit ourselves to the formal definition:

```
Definition imp (P Q:assert) : assert := fun s h =>
  ∀ h', h ⊥ h' ∧ P s h' → ∀ h'', h'' = h ∪ h' → Q s h''.
Notation "P -* Q" := (imp P Q).
```

Reynolds' Axioms The axioms of separation logic are defined by the following inductive type. An object of type `(semax P c Q)` represents the fact that, going from a state satisfying `P`, the execution of the command `c` leads to a state satisfying `Q`:

```
Inductive semax : assert → cmd → assert → Prop :=
    semax_assign : ∀ P x e,
     semax (update_store2 x e P) (x <- e) P
  | semax_lookup : ∀ P x e,
     semax (lookup2 x e P) (x <-* e) P
  | semax_mutation : ∀ P e e',
    semax (update_heap2 e e' P) (e *<- e') P
  | semax_seq : ∀ P Q R c d,
    semax P c Q → semax Q d R → semax P (c ; d) R
  ...
Notation "{{ P }} c {{ Q }}" := (semax P c Q).
```

where `update_store2`, etc. are predicate transformers, for example:

```
Definition update_store2 (x:var.v) (e:expr) (P:assert) : assert :=
  fun s h => P (store.update x (eval e s) s) h.
```

Using these definitions, we have implemented much of [1], including the proofs of soundness and completeness of the axioms of separation logic (w.r.t. the operational semantics), the "frame rule", various axioms for backward reasoning, etc. For example, let us just give the axiom for backward reasoning used in the example at the beginning of this section:

```
Lemma semax_mutation_backwards : ∀ P e e',
  {{ fun s h => ∃ e'', (e ↦ e'' ** (e ↦ e' -* P)) s h }} e *<- e' {{ P }}.
```

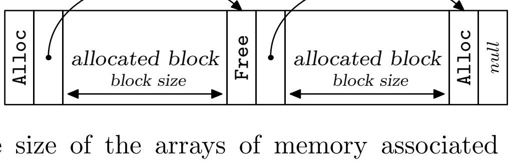
3.2 Verification of the Topsy Heap Manager

3.2.1 Heap Manager Overview

The heap manager of an operating system is the set of functions that provides dynamic memory allocation. In Topsy, these functions and related variables are defined in the files `Memory/MMHeapMemory.{h,c}`, with some macros in the file `Topsy/Configuration.h`. We are dealing here with the heap manager of Topsy version 2; a browsable source code is available online [9].

The *heap* is the area of memory reserved by Topsy for the heap manager. The latter divides the heap into allocated and free memory blocks: allocated blocks are memory blocks in use by programs, and free blocks form a pool of memory available for new allocations. In order to make an optimal use of the memory, allocated and free memory blocks form a partition of the heap. This is achieved by implementing memory blocks as a simply-linked list of contiguous blocks. In the following, we refer to this data structure as a *heap-list*.

In a heap-list, each block consists of a two-fields header and an array of memory. The first field of the header gives information on the status of the block (allocated or free, corresponding to the `Alloc` and `Free` flags); the second field is a pointer to the next block, which starts just after the current block. For example, here is a heap-list with one allocated block and one free block:



Observe that the size of the arrays of memory associated to blocks can be computed using the values of pointers. (In this paper, when we talk about the size of a block, we talk about its "effective" size, that is the size of the array of memory associated to it, this excludes the header.) The terminal block of the heap-list always consists of a sole header, marked as allocated, and pointing to null.

Initialization of the heap manager is provided by the following function:

```
  Error hmInit(Address addr) ...
```

Concretely, hmInit initializes the heap-list by building a heap-list with a single free block that spans the whole heap. The argument is the starting location of the heap. The size of the heap-list is defined by the macro KERNELHEAPSIZE. The function always returns HM_INITOK.

Allocation is provided by the following function:

```
Error hmAlloc(Address* addressPtr, unsigned long int size) ...
```

The role of hmAlloc is to insert new blocks marked as allocated into the heap-list. The first argument is a pointer provided by the user to get back the address of the allocated block, the second argument is the desired size. In case of successful allocation, the pointer contains the address of the newly allocated block and the value HM_ALLOCOK is returned, otherwise the value HM_ALLOCFAILED is returned. In order to limit fragmentation, hmAlloc performs compaction of contiguous free blocks and splitting of free blocks.

Deallocation is provided by the following function:

```
Error hmFree(Address address) ...
```

Concretely, hmFree turns allocated blocks into free ones. The argument corresponds to the address of the allocated block to free. The function returns HM_FREEOK if the block was successfully deallocated, or HM_FREEFAILED otherwise.

3.2.2 Heap Manager Specification and Verification

Heap-list

We define an assertion called Heap_List that holds for heaps that contain a well-formed heap-list. Separation logic is very convenient for this purpose. In particular, the property that blocks are disjoint can be expressed using the separating conjunction. The fact the blocks are contiguous relies on pointer arithmetic and this can also be expressed directly in separation logic.

Before defining the Heap_List assertion, we define an assertion to represent arrays of memory, i.e. sets of contiguous locations. Array p sz holds for a heap whose locations p, . . . , p+sz-1 have some contents:

```
Fixpoint Array (p:loc) (size:nat) struct size : assert :=
match size with
O => emp
| S n => (fun s h => ∃ y, (nat_e p ↦ int_e y) s h) ** Array (p+1) n
end.
```

We now come to the definition of heap-lists *without* terminal block (let us call them *pre-heap-lists* for convenience). Intuitively, (hl p l) represents the set of headers of a pre-heap-list whose first block starts at location p together with the set of free blocks (the allocated blocks are left outside); information about the blocks is captured by the parameter (l:list (nat*bool)): the list of sizes and flags of the blocks (:: is the list constructor and nil is the empty list):

```
Inductive hl : loc → list (nat*bool) → assert :=
| hl_last: ∀ s p h,
emp s h → hl p nil s h
| hl_Free: ∀ s h p h1 h2 size tl,
h1 ⊥ h2 → h = h1 ∪ h2 →
((nat_e p ⤇ Free::nat_e (p+2+size)::nil) ** (Array (p+2) size)) s h1 →
hl (p+2+size) tl s h2 →
hl p ((size,free)::tl) s h
```

```
| hl_Allocated: ∀ s h p h1 h2 size tl,
h1 ⊥ h2 → h = h1 ∪ h2 →
(nat_e p ⤇ Allocated::nat_e (p+2+size)::nil) s h1 →
hl (p+2+size) tl s h2 →
hl p ((size,alloc)::tl) s h.
```

where `free` and `alloc` are booleans. The first constructor specifies empty pre-heap-lists. The second constructor specifies pre-heap-lists that start with a free memory block (that is, a header marked as free and its associated block) followed by a pre-heap-list. The third constructor specifies pre-heap-lists that start with an allocated memory header (in this case, the associated block is left outside). Observe that the definition above uses pointer arithmetic to guarantee that there is no lost space between linked blocks.

Finally, we define heap-lists (*with* terminal block). This is simply the separating conjunction of a pre-heap-list with a terminal block (an allocated block pointing to null):

```
Definition Heap_List (l:list (nat*bool)) (p:nat) : assert :=
(hl p l) ** (nat_e (get_endl l p) ⤇ Allocated::null::nil).
```

where `(get_endl l p)` returns the location at the end of the list `l` starting from location `p`, i.e., the location of (the header of) the terminal block.

Properties of Heap-lists .

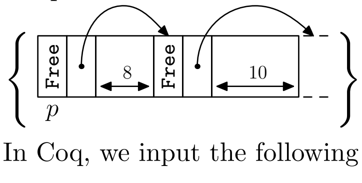
The heart of our verification of the Topsy heap manager consists of a few basic lemmas capturing the properties of operations such as compaction of blocks, splitting of a block, changing the status of blocks, etc. Since these operations rely on destructive updates, the properties in question are adequately expressed using the separating implication.

For example, the following lemma expresses compaction of two contiguous free blocks (`++` is the list append function of Coq):

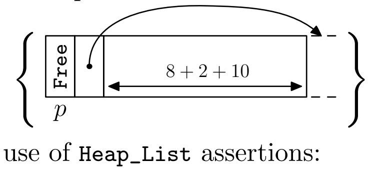
```
Lemma hl_compaction: ∀ l1 l2 size size' p s h,
Heap_List (l1 ++ (size,free)::(size',free)::nil ++ l2) p s h →
∃ y, (nat_e (get_endl l1 p + 1) ↦ y **
(nat_e (get_endl l1 p + 1) ↦ nat_e (get_endl l1 p + size + size' + 4) -*
Heap_List (l1 ++ (size+size'+2,free)::nil ++ l2) p)) s h.
```

The left-hand side of the (classical) implication states the existence of two contiguous free blocks `(size,free)` and `(size',free)`. The right-hand side represents the destructive update of the "next" field of the first block that is made to point to the block following the second block. As a result, the first block sees its size increased by the size of the second block. The function `get_endl` is used to compute the starting location of a block.

We can use this lemma to verify that a destructive update really performs compaction of blocks. Let us consider a concrete example:



$p + 1$ `*<-` $p + 22$



In Coq, we input the following goal, that makes use of `Heap_List` assertions:

```
Goal ∀ p,  Heap_List ((8,free)::(10,free)::nil) p
nat_e p +e int_e 1 *<- nat_e p +e int_e 22
 Heap_List ((20,free)::nil) p .
```

The application of the axiom for backward reasoning (seen in Sect. 3.1.2) leads to:

```
p : nat
s : store.s
h : heap.h
H : Heap_List ((8, free) :: (10, free) :: nil) p s h
============================
∃ e'' : expr,
((nat_e p +e int_e 1) ↦ e'' **
((nat_e p +e int_e 1) ↦ (nat_e p +e int_e 22) -*
Heap_List ((20, free) :: nil) p)) s h
```

This new goal is precisely the conclusion of the lemma we gave above. Application of this lemma terminates the proof.

HMInit

The initialization function `hmInit` transforms a given area of raw memory into an initial heap-list that consists of a single free block. In the source code, this area starts at location `hmStart` and has a fixed length `KERNELHEAPSIZE`. We formally verify `hmInit` for the general case of any starting location and any size greater than 4: the minimal space needed for two headers (the header of the free block and the header of the terminal block):

```
Definition hmInit_specif := ∀ p size, size ≥ 4 →
 Array p size  hmInit p size  Heap_List ((size-4,free)::nil) p .
```

The size of the array of memory corresponding to the free block is the size of the whole area of memory minus the size of the two headers. The verification of this triple is done almost automatically using a tactic provided by our Coq implementation. The non-automatic part is due to the translation of the assertions `Array` and `Heap_List` into the fragment of separation logic handled by this tactic. See Sect. 3.3.1 for more details.

Despite its apparent simplicity, this function turns out to be buggy, as we explain in Sect. 3.3.3.

HMAlloc

The allocation function `hmAlloc` searches for a large-enough free block in the heap-list, possibly performing compaction of free blocks if needed. If an adequate block is found, it is split into an allocated block (whose location is returned) and a free block (available for further allocations); otherwise, an error is returned.

We introduce new assertions to simplify specifications. Under the hypothesis that `(Heap_List lst p0)` holds, the assertion `(In_hl lst (p,size,flag) p0)` means that the block starting at location `p` has size `size` and flag `flag`. The assertion `(s |= b)` holds when `b` is true in the store `s`.

As informally stated, the specification of the allocation function consists in checking that (1) newly allocated blocks have at least the requested size, (2) they do not overlap with already allocated memory blocks (they are "fresh"), and (3) neither previously allocated memory blocks nor the rest of the memory is modified.

The formal specification of `hmAlloc` follows. In the pre-condition, we isolate some already allocated block `(x,sizex,alloc)`. In the post-condition, we ensure that (1) the newly allocated block (y,size'',alloc) has an appropriate size (i.e., greater than the requested size), (2) this newly allocated block does not overlap with previously allocated blocks (more precisely, the newly allocated block is built out of free blocks since (Heap_List l adr ** Array (y+2) size''), and it cannot be the previously allocated block x since x ≠ y), and (3) previously allocated memory blocks and the rest of the memory are not modified (because these areas are left outside of the area described by the Heap_List assertion). The second disjunction in the post-condition applies when allocation fails.

```
Definition hmAlloc_specif := ∀ adr x sizex size, adr > 0 → size > 0 →
{{ fun s h => ∃ l, Heap_List l adr s h ∧ In_hl l (x,sizex,alloc) adr ∧
(s |= var_e hmStart == nat_e adr) }}
hmAlloc result size entry cptr fnd stts nptr sz
{{ fun s h => (∃ l, ∃ y, y > 0 ∧ (s |= var_e result == nat_e (y+2)) ∧
∃ size'', size'' ≥ size ∧ (Heap_List l adr ** Array (y+2) size'') s h ∧
In_hl l (x,sizex,alloc) adr ∧ In_hl l (y,size'',alloc) adr ∧ x ≠ y)
∨
(∃ l, (s |= var_e result == nat_e 0) ∧
Heap_List l adr s h ∧ In_hl l (x,sizex,alloc) adr) }}.
```

Other assertions are essentially technical. The equality about the variable hmStart and the location adr is necessary because the variable hmStart is actually global and written explicitly in the original C source code of the allocation function. The inequality about the location adr is necessary because the function implicitly assumes that there is no block starting at the null location. The inequality about the requested size is not necessary, it is just to emphasize that null-allocation is a special case (see Sect. 3.3.3 for a discussion).

The allocation function relies on three functions to do (heap-)list traversals, compaction of free blocks, and eventually splitting of free blocks. In the rest of this section, we briefly comment on the verification of these three functions.

Traversal The function findFree traverses the heap-list in order to find a large-enough free block. It takes as parameters the requested size and a return variable entry to be filled with the location of an appropriate block if any:

```
Definition findFree_specif := ∀ adr x sizex size, size > 0 → adr > 0 →
{{ fun s h => ∃ l, Heap_List l adr s h ∧ In_hl l (x,sizex,alloc) adr ∧
(s |= (var_e hmStart == nat_e adr) &&& (var_e result == null)) }}
findFree size entry fnd sz stts
{{ fun s h => ∃ l, Heap_List l adr s h ∧ In_hl l (x,sizex,alloc) adr ∧
(s |= (var_e hmStart == nat_e adr) &&& (var_e result == null)) ∧
((∃ y, ∃ size'', size'' ≥ size ∧ In_hl l (y,size'',free) adr ∧
(s |= (var_e entry == nat_e y) &&& (nat_e y >> null)))
∨
s |= var_e entry == null) }}.
```

The post-condition asserts that the search succeeds and the return value corresponds to the starting location of a large-enough free block, or the search fails and the return value is null. The whole formal verification of findfree is summarized in Fig. B.2.

Compaction The function compact is invoked when traversal fails. Its role is to merge all the contiguous free blocks of the heap-list, so that a new traversal can take place and hopefully succeeds:

```
Definition compact_specif:= ∀ adr size sizex x, size > 0 → adr > 0 →
{{ fun s h => ∃ l, Heap_List l adr s h ∧ In_hl l (x,sizex,alloc) adr ∧
(s |= (var_e hmStart == nat_e adr) &&& (var_e result == null) &&&
(var_e cptr == nat_e adr)) }}
compact cptr nptr stts
{{ fun s h => ∃ l, Heap_List l adr s h ∧ In_hl l (x,sizex,alloc) adr ∧
(s |= (var_e hmStart == nat_e adr) &&& (var_e result == null)) }}.
```

The formal specification of `compact` asserts that it preserves the heap-list structure. Its verification is technically involved because it features two nested loops and therefore large invariants. The heart of this verification is the application of the compaction lemma already given in Sect. 3.2.2. The whole formal verification of `compact` is summarized in Fig. B.3.

Splitting The function `split` splits the candidate free block into an allocated block of appropriate size and a new free block:

```
Definition split_specif := ∀ adr size sizex x, size > 0 → adr > 0 →
{{ fun s h =>  ∃ l, Heap_List l adr s h ∧ In_hl l (x,sizex,alloc) adr ∧
(s |= (var_e hmStart == nat_e adr) &&& (var_e result == null)) ∧
(∃ y, ∃ size'', size'' ≥ size ∧ In_hl l (y,size'',free) adr ∧
(s |= var_e entry == nat_e y) ∧ y > 0 ∧ y ≠ x) }}
split entry size cptr sz
{{ fun s h => ∃ l, In_hl l (x,sizex,alloc) adr ∧
(∃ y, y > 0 ∧ (s |= var_e entry == int_e y) ∧
(∃ size'', size'' ≥ size ∧
(Heap_List l adr ** Array (y+2) size'') s h ∧
In_hl l (y,size'',alloc) adr ∧ y ≠ x)) }}.
```

The pre-condition asserts that there is a free block of size greater than `size` starting at the location pointed by `entry` (this is the block found by the previous list traversal). The post-condition asserts the existence of an allocated block of size greater than `size` (that is in general smaller than the original free block used to be). The whole formal verification of `split` is summarized in Fig. B.4.

HMFree

The deallocation function `hmFree` does a list traversal; if it runs into the location passed to it, it frees the corresponding block, and fails otherwise. Besides the fact that an allocated block becomes free, we must also ensure that `hmFree` does not modify previously allocated blocks nor the rest of the memory; here again, this is taken into account by the definition of `Heap_List`:

```
Definition hmFree_specif := ∀ p x sizex y sizey statusy, p > 0 →
{{ fun s h => ∃ l, (Heap_List l p ** Array (x+2) sizex) s h ∧
In_hl l (x,sizex,alloc) p ∧ In_hl l (y,sizey,statusy) p ∧
x ≠ y ∧ s |= var_e hmStart == nat_e p }}
hmFree (x+2) entry cptr nptr result
{{ fun s h => ∃ l, Heap_List l p s h ∧
In_hl l (x,sizex,free) p ∧ In_hl l (y,sizey,statusy) p ∧
s |= var_e result == HM_FREEOK }}.
```

The main difficulty of this verification was to identify a bug that allows for deallocation of the terminal block, as we explain in Sect 3.3.3.

3.3 Discussion

3.3.1 About Automation

Since our specifications take into account many details of the actual implementation, a number of Coq tactics needed to be written to make them tractable.

Tactics to decide disjointness and equality for heaps turned out to be very important. In practice, proofs of disjointness and equality of heaps are ubiquitous, but tedious because one always needs to prove disjointness to make unions of heaps commute; this situation rapidly leads to intricate proofs. For example, the proof of the lemma `hl_compaction` given in Sect. 3.2.2 leads to the creation of 15 sub-heaps, and 14 hypotheses of equality and disjointness. With these hypotheses, we need to prove several goals of disjointness and equality. Fortunately, the tactic language of Coq provides us with a means to automate such reasoning.

We also developed a certified tactic to verify automatically programs whose specifications belong to a fragment of separation logic without the separating implication (to compare with related work, this is the fragment of [17] without inductively defined datatypes).

We used this tactic to verify the `hmInit` function, leading to a proof script three times smaller than the corresponding interactive proof we made (58 lines/167 lines). Although the code in this case is straight-line, the verification is not fully automatic because our tactic does not deal directly with assertions such as `Array` and `Heap_List`.

Let us briefly comment on the implementation of this tactic. The target fragment is defined by the inductive type `assrt`. The tactic relies on a weakest-precondition generator `wp_frag` whose outputs are captured by another inductive type `L_assrt`. Using this weakest-precondition generator, a Hoare triple whose pre/post-conditions fall into the type `assrt` is amenable to a goal of the form `assrt` → `L_assrt` → `Prop`. Given a proof system `LWP` for such entailments, one can use the following lemma to automatically verify Hoare triples:

```
Lemma LWP_use: ∀ c P Q R,
wp_frag (Some (L_elt Q)) c = Some R →
LWP P R →
{{ assrt_interp P }} c {{ assrt_interp Q }}.
```

—The function `assrt_interp` projects objects of type `assrt` (a deep encoding) into the type `assert` (the shallow encoding introduced in this paper).
Goals of the form `assrt` → `L_assrt` → `Prop` can in general be solved automatically because the weakest-precondition generator returns goals that are inside the range of Presburger arithmetic (pointers are rarely multiplied between each other) for which Coq provides a native tactic (namely, the Omega test).

3.3.2 Translation from C Source Code

The programming language of separation logic is close enough to the subset of the C language used in the Topsy heap manager to enable a translation that preserves a syntactic correspondence. Thanks to this correspondence, it is immediate to identify a bug found during verification with its origin in the C source code. Below, we explain the main ideas behind the translation in question. Though it is systematic enough to be automated, we defer its certified implementation to future work and do it by hand for the time being.

The main difficulty in translating the original C source code is the lack of function calls and labelled jumps (in particular, the `break` instruction) in separation logic. To deal with function calls, we add global variables to serve as local variables and to carry the return value. To deal with the `break` instruction,

```
static void compact(HmEntry at) {
HmEntry atNext;

while (at != NULL) {
atNext = at->next;


while ((at->status == HM_FREED) &&
(atNext != NULL)) {


if (atNext->status != HM_FREED)
break;

at->next = atNext->next;

atNext = atNext->next;
}
at = at->next;}
}
```

```
Definition compact (at
atNext
brk tmp cstts nstts:var.v) :=
while (var_e at =/= null) (
atNext <-* (at -.> next);
brk <- nat_e 1 ;
cstts <-* (at -.> status);
while ((var_e cstts == Free) &&&
(var_e atNext =/= null) &&&
(var_e brk == nat_e 1)) (
nstts <-* (atNext -.> status);
ifte (var_e nstts =/= Free) thendo (
brk <- nat_e 0
) elsedo (
tmp <-* atNext -.> next;
at -.> next *<- var_e tmp;
atNext <-* atNext -.> next
));
at <-* (at -.> next)
).
```

Figure 3.1: Code Translation from C to Coq—Example

we add a global variable and a conditional branching to force exiting where loops can break.

Another minor point is that we need to add temporary variables to make up for the restricted set of expressions and commands of separation logic. For example, the evaluation of an expression in separation logic never returns a location, only values, thus we need beforehand to load a location into variable to be able to use it in a boolean expression; also, there is no command to lookup *and* mutate memory at the same time. We overcome these restrictions by decomposing complex expressions and commands, and using temporary variables. These temporary variables correspond to the parameters written without vowels in our specifications.

By way of example, Fig. 4.1 displays side-by-side the original `compact` function and its Coq counterpart.

The tables below summarize the whole Coq implementation:

Script files	Contents (lines)
`util.v`	Non-standard lemmas about integers, lists, etc. (825)
`heap.v`	Modules for locations, values, and heaps (2388)
`bipl.v`	Separation logic connectives (with tactics) (1579)
`axiomatic.v`	Separation logic triples, frame rule (1080)
`vc.v`	Weakest-precondition generator (196)
`contrib.v`	Various lemmas (arrays, etc.) (1077)
`contrib_tactics.v`	Various tactics (Omega extensions, etc.) (324)
`examples.v`	Small examples (411)
`example_reverse_list.v`	Reverse-list example (383)
`frag.v`	Tactic for a fragment of separation logic (1972)
`frag_examples.v`	Examples for the tactic above (176)

total: 10411 lines

Script files	Contents (lines)
`topsy_hm.v`	Heap-list definition and properties (1015)
`topsy_hmInit.v`	Initialization code, specification, and verification (313)
`topsy_hmAlloc.v`	Allocation code, specifications, and verifications (2762)
`topsy_hmFree.v`	Deallocation code, specification, and verification (536)
`hmAlloc_example.v`	Example of Sect. 3.3.3 (130)

total: 4756 lines

3.3.3 Benefits of Formal Verification

The main output of our experiment is that we have found several issues and bugs in the original source code of the Topsy heap manager. Another output is the Coq implementation of separation logic, that is readily available for other experiments. In particular, the verification of the Topsy heap manager in itself can actually be used for other verifications.

Issues and Bugs found in the Original Source Code

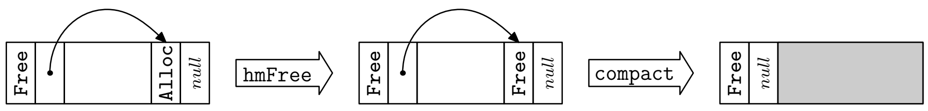
Out of Range Initialization When verifying the initialization function of the heap manager (Sect. 3.2.2), we found that the header of the terminal block was actually written outside of the memory area reserved for the heap manager. This illegal destructive update made the `Heap_List` assertion unprovable because the latter holds for a fixed area of memory. We corrected this bug by changing a single arithmetic operation, suggesting a programming miss. In all fairness, we must say that this bug was corrected in versions of Topsy posterior to version 2 (that we are using for verification).

Optimizations of Allocation When verifying the allocation function (Sect. 3.2.2), we found several useless operations that suggested immediate optimizations.

One such useless operation is the possibility to allocate a non-empty memory block (that is, a header and a non-empty array of memory) when performing a null-size allocation. Since null-size allocations are not filtered out, the alignment calculation is applied anyway, resulting in a non-empty allocation (in addition to the header). This was highlighted when writing assertions. We improved the implementation by forcing failure for null-size allocation.

Among other optimizations, there were useless assignments (to dead variables) and useless tests. For example, there were two identical variables assignments before calling and at the beginning of the `findFree` function; this was highlighted when writing the loop invariant in `findFree`. More interestingly, there was a useless test in the `compact` function. The second conjunct of the test of the inner loop (see Fig. 4.1) is useless because only the terminal block marked as allocated can point to `null`. Such an optimization cannot be done by an ordinary compilers, contrary to the former one.

Deallocation of the Terminal Block When verifying the deallocation function (Sect. 3.2.2), we found that it was possible to suppress allocable space without performing any allocation. This is because it is possible to deallocate the terminal block of the heap-list to trick compaction. The problem is better explained by the following scenario:



In this scenario, the terminal block is preceded by a free block. If we deallocate the terminal block and try to allocate a too-large block, this will trigger compaction and cause the leading free block to point to null. This problem is easily identified by the `Heap_List` assertion that enforces the terminal block to be marked as allocated. We fixed this problem by adding a test on the "next" field of the block to be deallocated in the deallocation function.

Using the Verification Result to Verify Other Code

Our verification of the Topsy heap manager provides us with new separation logic axioms that can be used for dynamic memory allocation without resorting to the native malloc/free commands of separation logic. In other words, we can use the specifications of `hmAlloc` and `hmFree` as triples to verify programs. For example, let us consider the following program:

```
Definition hmAlloc_example result entry cptr fnd stts nptr sz v :=
hmAlloc result 1 entry cptr fnd stts nptr sz;
ifte (var_e result =/= nat_e 0) thendo (
(var_e result *<- int_e v)
) elsedo ( skip ).
```

This program allocates a new block using `hmAlloc`, stores its location into the variable `result`, and stores some value `v` into this block. Using the specification of `hmAlloc` proved in Sect. 3.2.2, we can prove the following specification:

```
Definition hmAlloc_example_specif := ∀ v x e p, p > 0 →
{{ (nat_e x ↦ int_e e) **
    (fun s h => ∃ l, (s |= var_e hmStart == nat_e p) ∧
    Heap_List l p s h ∧ In_hl l (x,1,alloc) p) }}
hmAlloc_example result entry cptr fnd stts nptr sz v
{{ fun s h => s |= var_e result =/= nat_e 0 →
    ((nat_e x ↦ int_e e) ** (var_e result ↦ int_e v) ** TT **
    (fun s h => ∃ l, Heap_List l p s h ∧ In_hl l (x,1,alloc) p)) s h }}.
```

The post-condition asserts that, in case of successful allocation, the newly allocated block is separated from any previously allocated block.

3.4 Related Work

Our use case is reminiscent of work by Yu et al. that propose an assembly language for proof-carrying code and apply it to certification of dynamic storage allocation [15]. The main difference is that we deal with existing C code, whose verification is more involved because it has not been written with verification in mind. In particular, the heap-list data structure has been designed to optimize space usage; this leads to trickier manipulations (e.g., nested loop in `compact`), longer source code, and ultimately bugs, as we saw in Sect. 3.3.3. Also both allocators differ: the Topsy heap manager is a real allocation facility in the sense that the allocation function is self-contained (the allocator of Yu et al. relies on a pre-existing allocator) and that the deallocation function deallocated only valid blocks (the deallocator of Yu et al. can deallocate partial blocks).

The implementation of separation logic we did in the Coq proof assistant improves the work by Weber in the Isabelle proof assistant [16]. We think that our implementation is richer since it benefits from a substantial use case. In particular, we have developed several practical lemmas and tactics. Both implementations also differ in the way they implement heaps: we use an abstract data type implemented by means of modules for the heap whereas Weber uses partial functions.

Mehta and Nipkow developed modeling and reasoning methods for imperative programs with pointers in Isabelle [20]. The key idea of their approach is to model each heap-based data structure as a mapping from locations to values together with a relation, from which one derives required lemmas such as separation lemmas. The combination of this approach with Isabelle leads to compact proofs, as exemplified by the verification of the Schorr-Waite algorithm. In contrast, separation logic provides native notions of heap and separation, making it easier to model, for example, a heap containing different data structures (as it is the case for the `hmInit` function). The downside of separation logic is its special connectives that call for more implementation work regarding automation.

Tuch and Klein extended the ideas of Mehta and Nipkow to accommodate multiple datatypes in the heap by adding a mapping from locations to types [21, 58]. Thanks to this extension, the authors certified an abstraction of the virtual mapping mechanism in the L4 kernel from which they generate verified C code. Obviously, such a refinement strategy is not directly applicable to the verification of existing code such as the Topsy heap manager. More importantly, the authors point that the verification of the implementation of malloc/free primitives is not possible in their setting because they "break the abstraction barrier" (Sect. 6 of [21]).

Schirmer also developed a framework for Hoare logic-style verification inside Isabelle [19]. The encoded programming language is very rich, including in particular procedure calls, and heap-based data structures can be modeled using the same techniques as Mehta and Nipkow. Thanks to the encoding of procedure calls, it becomes easier to model existing source code (by avoiding, for example, the numerous variables we needed to add to translate the source code of the Topsy heap manager into our encoding of separation logic). However, it is not clear whether this richer encoding scales well for verification of non-trivial examples.

Caduceus [22] is a tool that takes a C program annotated with assertions and generates verification conditions that can be validated with various theorem provers and proof assistants. It has been used to verify several non-trivial C programs including the Schorr-Waite algorithm [23]. The verification of the Topsy heap manager could have been done equally well using a combination of Caduceus and Coq. However, Caduceus does support separation logic. Also, we needed a verification tool for assembly code in Topsy; for this purpose, a large part of our implementation for separation logic is readily reusable (this is actually work in progress). Last, we wanted to certify automation inside Coq instead of relying on a external verification condition generator.

Berdine, Calcagno and O'Hearn have developed Smallfoot, a tool for checking separation logic specifications [18]. It uses symbolic execution to produce verification conditions, and a decision procedure to prove them. Although Smallfoot is automatic (even for recursive and concurrent procedures), the assertions only describe the shape of data structures without pointer arithmetic. Such a limitation excludes its use for data structures such as heap-lists.

Chapter 4

MIPS Assembly Verification

In Chap. 3 we present a library that has proved to be efficient for the specification and the verification of C-like source code. However, low-level software, such as operating system, customary contains source code written in assembly code, for sake of optimization or for parts that are highly dependant on the architecture specificity. For instance, in operating systems, the code that manages the switching of context must be written in assembly. Indeed, this code loads and saves the value of the registers of the underlying processor for each threads. As emphasized in Chap. 2.3, such assembly code correctness is important for the task isolation.

In our effort of verification of the Topsy operating system, we have implemented a library to specify and verify source code written in MIPS assembly language [40]. Beyond the fact that the command language is different from the library presented in Chap. 3, this work also relies on deepest technical details. Whereas we have used unbound integers for our verification of the Topsy heap manager (because we modeled variables), the fact that we now manipulate registers binds us to use finite integers. Another specificity of assembly languages is that they do not provide structured control-flow (like loops and branches), but instead, jump instructions.

All these differences with our previous work justify the implementation of a new verification library. However, both implementations share some parts, for example the module that implements heaps.

This chapter is organized as follows. In Sect. 4.1 we describe our model of the MIPS architecture inside the Coq proof assistant, in Sect. 4.2 we present the verification of the function that loads the context of a thread, in Sect. 4.3 we discuss some facts about our verification, and finally, in Sect. 4.4 we present the related work.

4.1 MIPS Assembly Language in Coq

4.1.1 Machine-Integers Arithmetic

Formal verification of arithmetic functions is usually done w.r.t. high-level mathematical specifications. However, at the level of assembly code, many arithmetic properties of instructions depend on the finiteness of registers and on the physical representation of data. For example, the (signed) integer "-1" appears to be larger than any unsigned (and therefore positive) integer; some instructions trap on integer overflow while others do not, etc. Overlooking such problems often leads to security breaches, most famously integer-overflow bugs (see [38] for illustrations). It is therefore important to provide formal means to define the semantics of instructions together with lemmas that capture their properties in terms of mathematical (i.e., unbounded) integers.

Our approach to encode machine-integers arithmetic is to closely model the hardware circuitry using lists of bits (booleans) to represent the contents of registers and recursive functions to represent the operations on registers. We choose this approach because it is easy to extend with new, specialized instructions, compared to encoding machine integers with, say, sign-magnitude integers modulo.

Example: Hardware Arithmetic Operations We model the hardware addition as a recursive function that does bitwise comparisons and carry propagation:

```
Inductive bit : Set := o : bit | i : bit.
(* addition with LSB first *)
Fixpoint add_lst' (a b:list bit) (carry:bit) : list bit :=
match (a, b) with
(o :: a', o :: b') => carry :: add_lst' a' b' o
| (i :: a', i :: b') => carry :: add_lst' a' b' i
| (_ :: a', _ :: b') => match carry with
o => i :: add_lst' a' b' o | i => o :: add_lst' a' b' i end
| _ => nil
end.
(* addition with MSB first *)
Definition add_lst a b carry := rev (add_lst' (rev a) (rev b) carry).
```

Most computers distinguish between unsigned integers and signed integers in two's complement notation. The negation of a signed integer is defined using ones' complement and addition:

```
Definition cplt b := match b with i => o | o => i end.

Fixpoint cplt1 (lst:list bit) : list bit :=
match lst with nil => nil | hd :: tl => cplt hd :: cplt1 tl end.

Definition cplt2 lst :=
add_lst (cplt1 lst) (zero_extend_lst (length lst - 1) (i::nil)) o.
```

Using the addition, we further modeled the unsigned multiplication; using two's complement, we further modeled the signed multiplication, and so on.

Physical constraints and implementation choices make hardware arithmetic operations peculiar. Because of the finiteness of registers, they actually implement arithmetic modulo. A list of bits `(an::...::a0)` is interpreted as $(a_n \dots a_0)_2$, the encoding in base 2 of a mathematical integer; but depending on the context, this integer is unsigned, in which case its decimal value is $a_n 2^n + \dots + a_0$, or signed in two's complement notation, in which case its decimal value is $-a_n 2^n + a_{n-1}2^{n-1} + \ldots + a_0$. It is customary for assembly code to rely on properties of arithmetic modulo (e.g., to detect overflows) and to freely mix unsigned and signed integers (e.g., to access memory). Precise characterization of the properties of the hardware arithmetic operations w.r.t. their mathematical counterpart is therefore a must-have for formal verification of assembly code.

Example: Overflow Properties of Addition Let us note `[[lst]]u` (resp. `[[lst]]s`) the decimal value of the list of bits `lst` seen as an unsigned (resp. signed) integer. In Coq, these notations are implemented as recursive functions from lists of bits to mathematical integers. The hardware addition behaves like the mathematical addition only when non-overflow conditions are met:

```
Lemma add_lst_nat : ∀ n a b, length a = n → length b = n →
0 ≤ [[a]]u + [[b]]u < 2^^n → [[add_lst a b o]]u = [[a]]u + [[b]]u.

Lemma add_lst_Z : forall n a b, length a = S n → length b = S n →
-2^^n ≤ [[a]]s + [[b]]s < 2^^n → [[add_lst a b o]]s = [[a]]s + [[b]]s.
```

We proved further lemmas that capture the overflow properties of the hardware addition when overflow conditions are *not* met, the correctness of subtraction and multiplications, the relations between unsigned and signed integers, etc.

Because a processor usually manipulates machine integers of different sizes (e.g., to represent constants or contents of special registers such as accumulators), it is cumbersome to use directly lists of bits: the conditions about their lengths clutter formal verification. To simplify our development, we encapsulate all the functions modeling the hardware circuitry and the lemmas capturing their properties in a Coq module that provides an abstract type for machine integers. This abstract type is parameterized by the length of the underlying list of bits: `Parameter int : nat → Set`. This makes the relation between the lengths of the input and the output of operations explicit in the type of hardware operations.

Technically, this abstract type is implemented using dependent pairs: a machine integer of length `n` is a dependent pair whose first projection is a list of bits `lst` and whose second projection is the proof that its length is equal to `n`: `Inductive int (n:nat) : Set := mk_int : ∀ (lst:list bit), length lst = n → int n`. An excerpt of the interface of the resulting module is given below:

```
Parameter add : ∀ n, int n → int n → int n.
Notation "a ⊕ b" := (add a b).
Parameter u2Z : ∀ n, int n → Z.  (* lists of bits as unsigned *)
Parameter s2Z : ∀ n, int n → Z.  (* lists of bits as signed *)
Parameter add_u2Z : ∀ n (a b:int n), u2Z a + u2Z b < 2^^n →
u2Z (a ⊕ b) = u2Z a + u2Z b.
Parameter add_s2Z : ∀ n (a b:int (S n)), -2^^n ≤ s2Z a + s2Z b < 2^^n →
s2Z (a ⊕ b) = s2Z a + s2Z b.
Parameter Z2u : ∀ n, Z → int n.
Parameter Z2s : ∀ n, Z → int n.
```

`Z2u n z` (resp. `Z2s n z`) builds an unsigned (resp. a signed) machine integer of decimal value `z` and length `n` (if possible). These two constructors are used to defined constants, such as: `Definition four32 := Z2u 32 4`.

4.1.2 States

The state of a SmartMIPS processor is modeled as a tuple of a store of general-purpose registers, a store of control registers, an integer multiplier, and a heap (the mutable memory):

`Definition state := gpr.store * cp0.store * multiplier.m * heap.h.`

The module `gpr` is a store (in other words a map) from the type `gp_reg` of general-purpose registers to (32-bit) words, the module `cp0` is a map from the type `cp0_reg` of control registers, and `heap` is a map from natural numbers to words. We restrict ourselves to a word-addressable heap because it is all we need for arithmetic functions. The module for heap is implemented using a module for finite maps presented in Sect. 3.1.1. Let us comment more in detail on the implementation of the `multiplier` module, that makes an extensive use of our module for machine integers.

The SmartMIPS multiplier is a set of registers called ACX, HI, and LO that has been designed to enhance cryptographic computations. HI and LO are 32 bits long; ACX is only known to be at least 8 bits long. We implement the multiplier as an abstract data type `m` with three lookup functions `acx`, `hi`, and `lo` that return respectively a machine integer of length at least 8 bits and machine integers of length 32. Here follows the corresponding excerpt of the module interface:

```
Parameter acx_size : nat.
Parameter acx_size_min : 8 ≤ acx_size.
Parameter m : Set.
Parameter acx : m → int acx_size.
Parameter lo : m → int 32.
Parameter hi : m → int 32.
Parameter utoZ : m → Z. (* multiplier as an unsigned *)
```

The SmartMIPS instruction set features special instructions to take advantage of the SmartMIPS multiplier. For illustration, let us explain the encoding of the `mflhxu` instruction, that is often used in arithmetic functions: it performs a division of the multiplier by β=2^{32}, whose remainder is put in a general-purpose register and whose quotient is left in the multiplier. The corresponding hardware circuitry is essentially a shift: it puts the contents of LO into some general-purpose register, puts the contents of HI into LO, and zeroes ACX. Here is how we model this operation:

```
Definition mflhxu_op m := let (acx', hi') := (acx m, hi m) in
(Z2u acx_size 0, (zero_extend 24 acx', hi')).
```

What is important for verification is the properties of `mflhxu` w.r.t. the decimal value of the multiplier. Such properties can be derived as lemmas from the definition of `mflhxu_op`. For example, the decimal values of the multiplier before and after `mflhxu` are related as follows (`Zbeta n` stands for β^n=2^{32n}):

`Lemma mflhxu_utoZ : ∀ m, utoZ m = utoZ (mflhxu_op m) * Zbeta 1 + u2Z (lo m).`

4.1.3 The Mips Assembly Language

The syntax of MIPS is defined in the inductive type `cmd`. Except for control-flow commands (`seq`, `jr`,), the type constructors have the same names as their SmartMIPS counterparts. We define a `block` as a couple of a command and its starting memory address (the type `pc`). Formally in Coq:

```
Definition immediate := int 16.
Definition address := int 32.
Definition word := int 32.
```

```
Inductive cmd : Set :=
| lw : gp_reg -> immediate (*index*) -> gp_reg (*base*) -> cmd
| sw : gp_reg -> immediate (*index*) -> gp_reg (*base*) -> cmd
| la : gp_reg -> address -> cmd
| li : gp_reg -> word -> cmd
| mfhi : gp_reg -> cmd
| mflo : gp_reg -> cmd
| mtlo : gp_reg -> cmd
| mthi : gp_reg -> cmd
| mfc0 : gp_reg -> cp0_reg -> cmd
| mtc0 : gp_reg -> cp0_reg -> cmd
| move : gp_reg -> gp_reg -> cmd
| jr : gp_reg -> cmd
| nop : cmd
| seq: cmd -> cmd -> cmd.

Definition pc := int 32.
Definition block := pc * cmd.
```

We define the operational semantics in a manner similar to [2]. This approach allows us to describe an command language with jumps (here the command `jr`), with a big-step operational semantics. Details about this approach are discussed in details in Chap. 6. Formally in Coq, the semantics is encoded as an inductive type of signature:

```
Inductive cmd_semop: option (pc * state) →
                            (pc * cmd) → option (pc * state) → Prop := ...
```

The main difference with the operational semantics presented in Sect. 3.1.1, is that the initial and the final states are respectively coupled with a starting and an ending address. This is due to the fact that the assembly language is unstructured, more precisely that its instructions can be refereed by their addresses through jumps instructions.

4.1.4 Hoare-Triples

We use an instance of the separation logic assertion language for assembly (c.f., 2.1). Assertions are encoded as truth-functions from states to `Prop`, the type of predicates in Coq (this technique of encoding is called *shallow embedding*):
`Definition assert := gpr.store → cp0.store → multiplier.m → heap.h → Prop.`
Using this type, one can easily encode any first-order predicate. For example, the predicate that is true when variables `x` and `y` have the same contents is encoded as follows (`lookup` is a function provided by the interface of stores):

```
Definition x_EQ_y (x y : gp_reg) : assert :=
fun s _ _ _ => gpr.lookup x s = gpr.lookup y s.
```

To encode separating connectives, we use a module for heaps. In the following, we omit the definitions of heap-related functions: their names and notations are self-explanatory and details can be found in Chap. 3. First, we introduce a language for expressions used in separating connectives:

```
Inductive expr : Set :=
var_e : gp_reg → expr | int_e : int 32 → expr | add_e : expr → expr → expr | ...
```

In this language, variables are registers and constants are (32-bit) words. Given an expression `e` and a store `s`, the function `eval` returns the value of the expression `e` in store `s`.

The *mapsto* connective `e1 ↦ e2` holds in a state with a store `s` and a singleton heap with address `eval e1 s` and contents `eval e2 s`:

```
Definition mapsto (e e':expr) : assert := fun s _ _ h =>
∃ p, u2Z (eval e s) = 4 * p ∧ h = heap.singleton p (eval e' s).
Notation "e1 ↦ e2" := (mapsto e1 e2).
```

The *separating conjunction* `P ** Q` holds in a state whose heap can be divided into two disjoint heaps such that `P` and `Q` hold:

```
Definition sep_con (P Q:assert) : assert := fun s s' m h =>
∃ h1, ∃ h2, h1 ⊥ h2 ∧ h = h1 ∪ h2 ∧ P s s' m h1 ∧ Q s s' m h2.
Notation "P ⋆ Q" := (sep_con P Q).
```

In practice, the separating conjunction provides a concise way to express that two data structures reside in disjoint parts of the heap.

Using the separating conjunction, the mapsto connective can be generalized to arrays of words: `(e ⤇ a::b::...)` holds in a state whose heap contains a list of contiguous words `a`, `b`,... starting at address `eval e s`:

```
Fixpoint mapstos (e:expr) (lst:list (int 32)) : assert :=
match lst with
| nil => empty_heap
| hd::tl => (e ↦ int_e hd) ⋆ (mapstos (add_e e (int_e four32)) tl)
end.
Notation "e ⤇ lst" := (mapstos e lst).
```

Seaparation logic triples are encoded through their semantics. More formally, in Coq:

```
Definition semax_sem (P: assert) (C: pc * cmd) (Q: assert) : Prop :=
  let (lc, c) := C in (
    ∀ s1 s2 m h,
      P s1 s2 m h →
      ∃ s1',
        ∃ s2',
          ∃ m',
            ∃ h',
              cmd_semop (Some (lc, (s1, s2, m, h)))
                        (lc, c)
                        (Some (lc (+) Z2u _ (cmd_size c * 4), (s1', s2', m', h'))) /\
              Q s1' s2' m' h'
  ).

Notation "[[ P ]] c [[ Q ]]" := (semax_sem P c Q) (at level 82, no associativity).
```

Informally, the semantics of separation logic is: if we start the execution of the first instruction of the block `C` (which is the command `c` starting at the address `lc`) in a state for which the assertion `P` holds, then the execution will reach the address next to the last instruction of `c` in a state for which `Q` holds. These kind of triples are said to be of *total correctness*, because they assert that the execution will reach a final state.

For the verification of triples, we provide a set of lemmas, implementing the Reynolds axioms. This approach differs in the one chose in Sect. 3.1.2, where we build a set of rules, and proved there soundness and correctness w.r.t the triples semantics. For instance, the lemmas for the triples of the `la` instruction is:

```
Lemma la_semax: ∀ lc rt a P,
u2Z lc < 2 ^^ 32 - 4 →
[[ fun s s' m h => (P (gpr.update rt a s) s' m h) ]] (lc, la rt a) [[ P ]].
```

4.2 Verification of the Topsy Context Switching: `restoreContext`

In operating systems, the context switching code is responsible for saving and loading the context of a thread. This context corresponds to the set of the processor registers value (including the code pointer). Topsy uses an array, named `contexPtr`, to save the threads state . The function `saveContext` saves the current registers values inside this data-structure, while the function `restoreContext` restores the registers values from it.

Intuitively, the specifications of these functions make a comparison between the `contextPtr` and the values of the restored/saved registers. Thus, in order to build these specifications, we need two assertions: one models the `contextPr` data-structure which is resident in the memory (which is the `thread_context_mem` separation logic assertion bellow), and one that represents the processor register values (which is the `thread_context` separation logic assertion bellow):

```
Definition thread_context_mem (at_ v0 v1 a0 a1 a2 a3 t0 t1 t2 t3 t4 t5 t6 t7
t8 t9 s0 s1 s2 s3 s4 s5 s6 s7 gp sp fp ra hi lo status pc l: int 32) : assert :=
(add_e (int_e l) (int_e (sign_extend_16_32 RETURNVALUE0_OFFSET))↦int_e v0) **
(add_e (int_e l) (int_e (sign_extend_16_32 RETURNVALUE1_OFFSET))↦int_e v1) **
(add_e (int_e l) (int_e (sign_extend_16_32 ARGUMENT0_OFFSET))↦int_e a0) **
...

Definition thread_context (at_ v0 v1 a0 a1 a2 a3 t0 t1 t2 t3 t4 t5 t6 t7
t8 t9 s0 s1 s2 s3 s4 s5 s6 s7 gp sp fp ra hi lo status: int 32) : assert :=
fun s s' m h =>
emp  s s' m h /\
gpr.lookup gpr_at s = at_ /\
gpr.lookup gpr_v0 s = v0 /\
gpr.lookup gpr_v1 s = v1 /\
gpr.lookup gpr_a0 s = a0 /\
...
```

In Fig. 4.1, we present both the original MIPS source code and the Coq model of the function `restoreContext`. As we can see, the translation is rather straightforward, and both codes are close.

We now take a closer look at how this function is working. In the starting states, the register `a0` points to the `contextPtr` to be restored. The first instruction makes the register `k1` to point to this data-structure. Then, one by one, the function restores the registers from `a0` to `ra`. Finally, the function restores three specials registers: `hi` and `lo`, which are the registers of the multiplier, and `status`, which contains the privilege of the restored thread.

The formal specification of `rectoreContext`, in Coq, is:

```
Lemma restoreContextVerif:
  ∀ lc,
   u2Z lc + cmd_size restoreContext * 4 < 2 ^^ 32  ->
   [[
     fun s s' m h =>
     (thread_context_mem at_ v0 v1 a0 a1 a2 a3 t0 t1 t2 t3 t4 t5 t6 t7
           t8 t9 s0 s1 s2 s3 s4 s5 s6 s7 gp sp fp ra hi lo status pc (gpr.lookup gpr_a0 s)) s s' m h
   ]]
   (lc, restoreContext)
   [[
     fun s s' m h =>
     (thread_context_mem at_ v0 v1 a0 a1 a2 a3 t0 t1 t2 t3 t4 t5 t6
         t7 t8 t9 s0 s1 s2 s3 s4 s5 s6 s7 gp sp fp ra hi lo status pc (gpr.lookup gpr_k1 s)) s s' m h /\
     (thread_context  at_ v0 v1 a0 a1 a2 a3 t0 t1 t2 t3 t4 t5 t6 t7
         t8 t9 s0 s1 s2 s3 s4 s5 s6 s7 gp sp fp ra hi lo status)  s s' m heap.emp
   ]].
```

```
move k1, a0
lw   a0, ARGUMENT0_OFFSET(k1)
lw   a1, ARGUMENT1_OFFSET(k1)
lw   a2, ARGUMENT2_OFFSET(k1)
lw   a3, ARGUMENT3_OFFSET(k1)
lw   v0, RETURNVALUE0_OFFSET(k1)
lw   v1, RETURNVALUE1_OFFSET(k1)
lw   AT, AT_OFFSET(k1)
lw   t0, CALLERSAVED0_OFFSET(k1)
lw   t1, CALLERSAVED1_OFFSET(k1)
lw   t2, CALLERSAVED2_OFFSET(k1)
lw   t3, CALLERSAVED3_OFFSET(k1)
lw   t4, CALLERSAVED4_OFFSET(k1)
lw   t5, CALLERSAVED5_OFFSET(k1)
lw   t6, CALLERSAVED6_OFFSET(k1)
lw   t7, CALLERSAVED7_OFFSET(k1)
lw   t8, CALLERSAVED8_OFFSET(k1)
lw   t9, CALLERSAVED9_OFFSET(k1)
lw   s0, CALLEESAVED0_OFFSET(k1)
lw   s1, CALLEESAVED1_OFFSET(k1)
lw   s2, CALLEESAVED2_OFFSET(k1)
lw   s3, CALLEESAVED3_OFFSET(k1)
lw   s4, CALLEESAVED4_OFFSET(k1)
lw   s5, CALLEESAVED5_OFFSET(k1)
lw   s6, CALLEESAVED6_OFFSET(k1)
lw   s7, CALLEESAVED7_OFFSET(k1)
lw   gp, GLOBALPOINTER_OFFSET(k1)
lw   sp, STACKPOINTER_OFFSET(k1)
lw   fp, FRAMEPOINTER_OFFSET(k1)
lw   ra, RETURNADDRESS_OFFSET(k1)

lw   k0, HI_OFFSET(k1)
nop
mthi k0
lw   k0, LO_OFFSET(k1)
nop
mtlo k0
lw   k0, STATUSREGISTER_OFFSET(k1)
nop
mtc0 k0, c0_status
```

```
move gpr_k1 gpr_a0;
lw gpr_a0 ARGUMENT0_OFFSET gpr_k1;
lw gpr_a1 ARGUMENT1_OFFSET gpr_k1;
lw gpr_a2 ARGUMENT2_OFFSET gpr_k1;
lw gpr_a3 ARGUMENT3_OFFSET gpr_k1;
lw gpr_v0 RETURNVALUE0_OFFSET gpr_k1;
lw gpr_v1 RETURNVALUE1_OFFSET gpr_k1;
lw gpr_at AT_OFFSET gpr_k1;
lw gpr_t0 CALLERSAVED0_OFFSET gpr_k1;
lw gpr_t1 CALLERSAVED1_OFFSET gpr_k1;
lw gpr_t2 CALLERSAVED2_OFFSET gpr_k1;
lw gpr_t3 CALLERSAVED3_OFFSET gpr_k1;
lw gpr_t4 CALLERSAVED4_OFFSET gpr_k1;
lw gpr_t5 CALLERSAVED5_OFFSET gpr_k1;
lw gpr_t6 CALLERSAVED6_OFFSET gpr_k1;
lw gpr_t7 CALLERSAVED7_OFFSET gpr_k1;
lw gpr_t8 CALLERSAVED8_OFFSET gpr_k1;
lw gpr_t9 CALLERSAVED9_OFFSET gpr_k1;
lw gpr_s0 CALLEESAVED0_OFFSET gpr_k1;
lw gpr_s1 CALLEESAVED1_OFFSET gpr_k1;
lw gpr_s2 CALLEESAVED2_OFFSET gpr_k1;
lw gpr_s3 CALLEESAVED3_OFFSET gpr_k1;
lw gpr_s4 CALLEESAVED4_OFFSET gpr_k1;
lw gpr_s5 CALLEESAVED5_OFFSET gpr_k1;
lw gpr_s6 CALLEESAVED6_OFFSET gpr_k1;
lw gpr_s7 CALLEESAVED7_OFFSET gpr_k1;
lw gpr_gp GLOBALPOINTER_OFFSET gpr_k1;
lw gpr_sp GLOBALPOINTER_OFFSET gpr_k1;
lw gpr_fp FRAMEPOINTER_OFFSET gpr_k1;
lw gpr_ra RETURNADDRESS_OFFSET gpr_k1;

lw gpr_k0 HI_OFFSET gpr_k1;
mthi gpr_k0;

lw gpr_k0 LO_OFFSET gpr_k1;
mtlo gpr_k0;

lw gpr_k0 STATUSREGISTER_OFFSET gpr_k1;
mtc0 gpr_k0 cp0_status.
```

Figure 4.1: Code Translation from MIPS assembly to Coq

The precondition asserts that there is a `contextPtr` pointed by the `a0` register. The first part of the post-condition asserts that the `contextPtr` did not change, and that it is pointed by the register `k1`. The second part asserts that the registers contains the same values as the `contextPtr`. The most important, in regards of the task isolation property, is that the `status` register is restored with the proper value.

4.3 Discussion

In order to verify the function of Topsy which is responsible for the thread context loading, we implemented a set of lemmas directly using the semantics of the separation logic in total correctness. The main motivation is that such triples semantics are closed to the one presented in Sect. 6.1.3. We have also investigated the use of a Hoare-triples proof system similar to the one presented in [2]. We encoded its assertion language, proof system, and proved its soundness. However, it has shown to be difficult to use for the verification of the Topsy context switching code. Moreover, this proof system was originally designed for partial correctness triples (which is not compatible with our work in Chap. 6).

We met different issues in our verification, particularly we faced a huge resource consumption by Coq. This seemed to come from mainly three kind of subgoals: the equality and disjointness of heaps, and the finite integer arithmetic.

For this later, as we always face the same pattern of assertions, we build an ad-hoc decision procedure by reflection. If this code has proved useful compared to a pure Coq tactic (which sometimes, never finishes), it is not general enough to be used for other proof on finite integers arithmetic. An interesting future work should be the implementation of an efficient decision procedure for finite-integer arithmetic. We think that this decision procedure should be implemented on the same model as the Coq tactic `romega`: an external prover that gives a certificate to a checker implemented by reflection inside Coq. Such approach should allows to use elaborate algorithm (like the one presented in [38]) which may be too difficult to implement in Coq, and yet to generate small proof terms (due to the reflexive checker). For the issues about the heap (here also the tactics sometimes never finish), we advocate that it should be possible to implement a decision procedures entirely by reflexion (for both equality and disjointness), and that it could be a not so difficult, yet interesting, future work.

4.4 Related Work

Much work about formal encoding of assembly languages in proof assistants has been done with application to proof-carrying code (PCC) in mind [35, 36, 37]. Although the encoded semantics often allows for programs with arbitrary jumps, details such as machine integers are usually not treated. This makes it difficult to reuse existing implementations of PCC frameworks to formally verify arithmetic functions, whose algorithms require bit-level specifications.

There exist other encodings of machine integers in Coq. Leroy has encoded such a library for integers modulo 2^{32} as part of the development of a certified compiler [3]. His encoding uses the relative integers of Coq (the `Z` type) instead of lists of bits. We found it difficult to reuse directly his implementation because the length of integers (32) is hard-wired and we needed a similar library for several lengths. Chlipala has encoded a library similar to ours but based on dependent vectors [39]. We think that our implementation based on an abstract type is more flexible than dependent vectors because it separates the issues of formal proofs and dependent types.

Our use case is reminiscent of work by Ni et al. that proposed the verification for context switching code for x86 processors[41]. Beyond this underlying architecture difference, the code we verified was extracted from an existing operating system.

Part II

Verification Tools

Chapter 5

A Certified Verifier

There exist several implementations of verifiers for separation logic [18, 31, 33], but they all share a common weak point: they are not themselves verified.

It makes little doubt that a verifier for separation logic can be verified using, say, a proof assistant. The real question is: At which price? Indeed, such verifiers are non-trivial pieces of software. They require manipulation of concepts such as fresh variables, that are notoriously hard to get right in a proof assistant. They also rely on decision procedures for arithmetic that are not necessarily available in a suitable form. This means at least a non-negligible implementation work.

The basic design idea of our verifier is to turn separation logic triples into logical implications between assertions to be proved automatically. Similarly to related work [18, 33], this is implemented in three successive phases:
1. *Verification conditions generator*: The input triple is cut into a list of loop-free triples.
2. *Triple transformation*: Every loop-free triple is turned into logical implications between assertions.
3. *Entailment*: Every implication derived from the previous phase is proved valid.

Besides formal verification of these three phases, another originality of our work is the triple transformation phase in itself: we appeal to a new proof system that mixes backward and forward reasoning whereas related work [18, 33] essentially relies on forward reasoning (the advantages of our approach are discussed in detail in Sect. 5.6.1).

This chapter is organized as follows. First, we present the fragment of the separation logic our verifier deals with in Sect. 5.1. Then we explain for each phase of our verifier what it does and how we prove it correct: the entailment phase in Sect. 5.2, the triple transformation phase in Sect. 5.3, and the verification conditions generator in Sect. 5.4. The resulting verifier amounts to a simple combination of these three phases, as summarized in Sect. 5.5. In Sect. 5.6, we comment on practical aspects: the size of generated proof-terms, performance benchmarks for the derived stand-alone OCaml verifier, and examples of separation logic triples verifications. Sect. 5.7 is dedicated to comparison with related work.

5.1 Target Fragment of Separation Logic

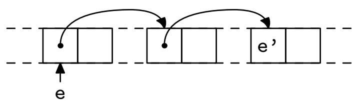
In this section, we present the fragment of the assertion language of separation logic that our verifier deals with. This is basically the same fragment as [17], where it was chosen as a good candidate for automation because entailment (classical implication of assertions) is decidable. We extend this fragment with Presburger arithmetic to handle pointer arithmetic. Since programs never multiply pointers between each other, we think that this extension suffices to enable most verifications; the same extension is done in [33]. The only datatype we deal with is singly-linked lists. We think that the ideas we develop in this paper for lists extend to other recursive datatypes such as trees, along the same lines as [18].

5.1.1 Syntax and Informal Semantics

Formulas of our fragment represent states symbolically. To represent a store symbolically, we use the language of boolean expressions `expr_b` introduced in Sect. 3.1.1. This gives us enough expressiveness to write pointer arithmetic formulas. To represent a heap symbolically, we use the following fragment `Sigma` of the assertion language of separation logic:

```
Inductive Sigma : Set :=
| emp : Sigma
| singl : expr → expr → Sigma
| cell : expr → Sigma
| star : Sigma → Sigma → Sigma
| lst : expr → expr → Sigma.
```

Simply put, this syntax represents the connectives defined in Sect. 3.1.2: `emp` represents the empty heap like the homonym connective defined by shallow encoding; `singl` is syntax for `mapsto`; `cell e` represents a singleton heap whose contents is unknown; `star h h'` is the syntactic separating conjunction (Coq notation: $h \star h'$; this is the same notation as the "semantic" separating conjunction in Sect. 3.1.2; in informal arguments, we will write $\star$ for the separating conjunction). Note that `Sigma` does not contain the separating implication of separation logic. Compared to the shallow encoding of Sect. 3.1.2, we add the formula `lst e e'` that represents a heap that contains a singly-linked list whose head has location `e` and whose last element points to `e'`, as illustrated below:



To summarize, the syntax of our assertion language `assrt` is defined as a product of `expr_b` and `Sigma`:

```
Definition Pi := expr_b.
Definition assrt := Pi * Sigma.
```

In informal arguments, we will write $\langle \pi, \sigma \rangle$ for assertions.

5.1.2 Formal Semantics

In the previous section, we have defined the syntax of formulas in Coq. Their semantics has already been defined in Sect. 3.1.2 by a shallow encoding. In this section, we make the relation between both with a satisfiability relation. This technique of encoding is called *deep encoding* and is typical of tactics by reflection. Indeed, the latter needs to "parse" the assertion language to prove the validity of formulas, which is difficult to do when the syntax is not an inductive type.

The formal semantics of Sigma formulas is a satisfiability relation between (syntactic) formulas and states. It is defined by a function Sigma_interp of type Sigma -> store.s -> heap.h -> Prop where store.s -> heap.h -> Prop is precisely the type assert of formulas in our shallow encoding:

```
Fixpoint Sigma_interp (a : Sigma) : assert :=
  match a with
    | emp => sep.emp
    | singl e1 e2 => fun s h =>
      (e1↦e2) s h ∧ eval e1 s ≠ 0
    | cell e => fun s h =>
      (∃ v, (e↦int_e v) s h) ∧ eval e s ≠ 0
    | s1 ** s2 =>
      Sigma_interp s1 ** Sigma_interp s2
    | lst e1 e2 => Lst e1 e2
  end.
```

(the formulas from Sect. 3.1.2 are encapsutaled in a module sep to avoid naming conflicts) where Lst is an inductive type of the appropriate type defining singly-linked lists:

```
Inductive Lst : expr → expr → assert :=
| Lst_end: ∀ e e' s h,
  eval e s = eval e' s → sep.emp s h →
  Lst e e' s h
| Lst_next: ∀ e e' e'' data s h h1 h2,
  h1 # h2 → h = h1 +++ h2 →
  eval e s ≠ eval e' s →
  eval e s ≠ 0 →
  eval (e +e nat_e 1) s ≠ 0 →
  (e↦e'' ** (e +e nat_e 1↦data)) s h1 →
  Lst e'' e' s h2 →
  Lst e e' s h.
```

The semantics of our fragment is finally defined as the conjunction of the satisfiability relations of its two components (expr_pi is syntactically equal to expr_b, and eval_pi is syntactically equal to eval_b):

```
Definition assrt_interp (a : assrt) : assert :=
match a with
| (pi, sigm) => fun s h =>
   eval_pi pi s = true ∧ Sigma_interp sigm s h
end.
```

5.1.3 Disjunctions of Assertions

In fact, we further need to extend our assertion language to represent disjunctions of assertions. Intuitively, this is because loop invariants are usually written as disjunctions. In informal arguments, we will write $\langle\pi_1, \sigma_1\rangle \vee \ldots \vee \langle\pi_n, \sigma_n\rangle$ for disjunctions of assertions. Adding this disjunction on top of the fragment allows to handle disjunction for the separation logic part, without multiplying the set of rules necessary to prove entailment and without loss of expressivity. Indeed, all formulas belonging to a fragment with disjunction inside Sigma have a counterpart in our fragment. For instance, $\langle\pi, \sigma_1 \star (\sigma_2 \vee \sigma_3) \star \sigma_4\rangle$ would have

$$\frac{\langle \pi_1, \sigma_2 \star \sigma_1 \rangle \vdash \langle \pi, \sigma \rangle}{\langle \pi_1, \sigma_1 \star \sigma_2 \rangle \vdash \langle \pi, \sigma \rangle}\ \texttt{coml} \qquad \frac{\langle \pi_1 \wedge e_1 \neq 0, \sigma_1 \star e_1 \mapsto e_2 \rangle \vdash \langle \pi_2, \sigma_2 \rangle}{\langle \pi_1, \sigma_1 \star e_1 \mapsto e_2 \rangle \vdash \langle \pi_2, \sigma_2 \rangle}\ \texttt{singl\_not\_null}$$

$$\frac{\pi_1 \rightarrow \pi_2}{\langle \pi_1, \texttt{emp} \rangle \vdash \langle \pi_2, \texttt{emp} \rangle}\ \texttt{tauto} \qquad \frac{\langle \pi_1 \wedge e_1 \neq e_3, \sigma_1 \star e_1 \mapsto e_2 \star e_3 \mapsto e_4 \rangle \vdash \langle \pi_2, \sigma_2 \rangle}{\langle \pi_1, \sigma_1 \star e_1 \mapsto e_2 \star e_3 \mapsto e_4 \rangle \vdash \langle \pi_2, \sigma_2 \rangle}\ \texttt{star\_neq}$$

$$\frac{\neg\, \pi_1}{\langle \pi_1, \texttt{emp} \rangle \vdash \langle \pi_2, \texttt{emp} \rangle}\ \texttt{incons} \qquad \frac{\pi_1 \rightarrow e_1 = e_3 \quad \pi_2 \rightarrow e_2 = e_4 \quad \langle \pi_1, \sigma_1 \rangle \vdash \langle \pi_2, \sigma_2 \rangle}{\langle \pi_1, \sigma_1 \star \texttt{lst}\, e_1\, e_2 \rangle \vdash \langle \pi_2, \sigma_2 \star \texttt{lst}\, e_3\, e4 \rangle}\ \texttt{lstsamelst}$$

$$\frac{\pi_1 \rightarrow e_1 = e_3 \quad \pi_1 \rightarrow e_2 = e_4 \quad \langle \pi_1, \sigma_1 \rangle \vdash \langle \pi_2, \sigma_2 \rangle}{\langle \pi_1, \sigma_1 \star e_1 \mapsto e_2 \rangle \vdash \langle \pi_2, \sigma_2 \star e_3 \mapsto e_4 \rangle}\ \texttt{star\_elim} \qquad \frac{\pi_1 \rightarrow e_1 = e_3 \quad \langle \pi_1, \sigma_1 \rangle \vdash \langle \pi_2, \sigma_2 \rangle}{\langle \pi_1, \sigma_1 \star \texttt{cell}\, e_1 \rangle \vdash \langle \pi_2, \sigma_2 \star \texttt{cell}\, e_3 \rangle}\ \texttt{star\_elim''}$$

$$\frac{\begin{array}{c}\pi_1 \rightarrow e_1 = e_3 \\ \langle \pi_1, \sigma_1 \star \texttt{cell}\, e_4 \rangle \vdash \langle \pi_2, \sigma_2 \star \texttt{lst}\, e_2\, e_4 \rangle\end{array}}{\langle \pi_1, \sigma_1 \star \texttt{cell}\, e_4 \star \texttt{lst}\, e_1\, e_2 \rangle \vdash \langle \pi_2, \sigma_2 \star \texttt{lst}\, e_3\, e_4 \rangle}\ \texttt{lstelim} \qquad \frac{\begin{array}{c}\pi_1 \rightarrow e_1 = e_3 \quad e_1 \neq e_4 \quad e_1 \neq 0 \\ \langle \pi_1, \sigma_1 \rangle \vdash \langle \pi_2, \sigma_2 \star \texttt{cell}\, e_1 + 1 \star \texttt{lst}\, e_2\, e_4 \rangle\end{array}}{\langle \pi_1, \sigma_1 \star e_1 \mapsto e_2 \rangle \vdash \langle \pi_2, \sigma_2 \star \texttt{lst}\, e_3\, e_4 \rangle}\ \texttt{lstelim'''}$$

Figure 5.1: Excerpt of the `entail` Proof System

the same semantics as $\langle \pi, \sigma_1 \star \sigma_2 \star \sigma_4 \rangle \vee \langle \pi, \sigma_1 \star \sigma_3 \star \sigma_4 \rangle$. We encode disjunctions of assertions by lists:

```
Definition Assrt := list assrt.
```

Like for `assrt`, the semantics of `Assrt` is defined as a satisfiability relation, that is simply the disjunction of the satisfiability relations of the `assrt` disjuncts (function `Assrt_interp`, of type `Assrt -> assert`).

5.2 Entailment

In this section, we present a proof system for entailments of assertions defined in the previous section. Using this proof system, we implement a Coq tactic and a function to prove validity for entailment between two formulas of type `assrt` (files `frag_list_entail.v` and `expr_b_dp.v` in [11]).

5.2.1 Entailment Proof System

Our proof system enables derivation of entailments of type `assrt -> assrt -> Prop` such that the left hand side (lhs) semantically implies the right hand side (rhs). In Coq, this proof system takes the form of an inductive predicate `entail`. An excerpt in informal notation is displayed in Fig. 5.1. Most rules are fairly intuitive. For example, we can take a look at the rule `coml`, that captures the fact that the separating conjunction is commutative on the left of implication.

We have implemented a tactic (`Entail`, which cannot be extract in Ocaml) that iteratively applies the rules of `entail` to solve entailments. Here follows an example of such a goal (see Fig. 5.2 for an informal account of the proof built underneath):

```
Goal entail
     (true_b, list e e' ** e'↦e'' ** cell (e'+1) ** list e'' 0)
     (true_b, list e 0).
  unfold e, e', e''; Entail.
Qed.
```

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\texttt{true\_b} \rightarrow \texttt{true\_b}}{\langle \texttt{true\_b}, \texttt{emp}\rangle \vdash \langle \texttt{true\_b}, \texttt{emp}\rangle}\ \texttt{tauto}}{\langle \texttt{true\_b}, \texttt{lst e'' 0}\rangle \vdash \langle \texttt{true\_b}, \texttt{lst e'' 0}\rangle}\ \texttt{lstsamelst}}{\langle \texttt{true\_b}, \texttt{cell e'+1} ** \texttt{lst e'' 0}\rangle \vdash \langle \texttt{true\_b}, \texttt{cell e'+1} ** \texttt{lst e'' 0}\rangle}\ \texttt{star\_elim''}}{\langle \texttt{true\_b}, \texttt{e'} \mapsto \texttt{e''} ** \texttt{cell e'+1} ** \texttt{lst e'' 0}\rangle \vdash \langle \texttt{true\_b}, \texttt{lst e' 0}\rangle}\ \texttt{lstelim'''}}{\langle \texttt{true\_b}, \texttt{lst e e'} ** \texttt{e'} \mapsto \texttt{e''} ** \texttt{cell e'+1} ** \texttt{lst e'' 0}\rangle \vdash \langle \texttt{true\_b}, \texttt{lst e 0}\rangle}\ \texttt{lstelim}$$

Figure 5.2: Example of Entailment: List Composition

We have proved formally that the `entail` proof system is correct, i.e., that only valid entailments can be derived:

```
Lemma entail_soundness : ∀ P Q, entail P Q →
  assrt_interp P ==> assrt_interp Q.
```

We think that the `entail` proof system is also complete because it contains the rules of the proof system of [17], which is complete. An important point for this proof system to be complete is that it makes explicit the arithmetic constraints that are deducible from the `Sigma` formulas. There are two kinds of such constraints: (1) by definition of `cell` and `singl`, all cells locations are strictly positive integers (e.g., rule `singl_not_null`), and (2) cells on both sides of `star` have pairwise different locations (e.g., rule `star_neq`).

5.2.2 Entailment Verification Procedure

In this section, we explain the Coq function `entail_fun` that proves entailments. Because we verify it, this function can be used as a tactic by reflection. It implements a reasoning similar to the `entail` proof system but this is no redundant work: we will actually use the `Entail` tactic to prove the correctness of `entail_fun`.

Implication Between Heaps

The first building block of the `entail_fun` function is a function `Sigma_impl` that proves the validity of implications between two abstract heaps. This function iteratively calls the function `elim_common_subheap` (Fig. 5.3), which tries to eliminate, subheap by subheap, the lhs `sig1 ** remainder` from the rhs `sig2`. This elimination is performed by the function `elim_common_cell` (Fig. 5.3), which tries to remove the subheap `sig` from both `sig ** remainder` and `sig'`. It is essentially a case-analysis on both heaps leading to the application of an `entail` rule. For example, Fig. 5.3 shows the case for which the rule `lstelim'''` of the `entail` proof system applies.

In fact, Fig. 5.2 also provides an illustration of what is achieved by the function `Sigma_impl`. The intermediate abstract heaps happen to be the successive results of elimination of common subheaps by `elim_common_cell`. For example, here is the result of the third call:

```
elim_common_cell true_b (cell e'+1)
  (lst e'' 0) (cell e'+1 ** lst e''0) =
  Some (lst e'' 0, lst e'' 0)
```

```
Fixpoint elim_common_subheap
  (pi : Pi) (sig1 sig2 remainder : Sigma)
    : option (Sigma * Sigma) :=
match sig1 with
  | sig11 ** sig12 =>
    match elim_common_subheap pi sig11 sig2
      (sig12 ** remainder) with
      | None => None
      | Some (sig11', sig12') =>
        Some (remove_empty_heap pi
          (sig11' ** sig12), sig12')
    end
  | _ => elim_common_cell pi sig1 remainder sig2
end.
```

```
 Fixpoint elim_common_cell
   (pi : Pi) (sig remainder sig' : Sigma)
     : option (Sigma * Sigma) :=
 match sig' with
   ...
   | _ => ...
   match (sig, sig') with
   ...
   (* this case corresponds to the application
   of the rule lstelim''' *)
   | (singl e1 e2, lst e3 e4) =>
   if andb (expr_b_dp (pi =b> (e1 == e3)))
     (andb (expr_b_dp (pi =b> (e1 =/= e4)))
     (expr_b_dp (pi =b> (e1 =/= nat_e 0))))
   then Some
     (emp, (cell (e1 +e nat_e 1)) ** (lst e2 e4))
  else None
  ...
  end
end.
```

Figure 5.3: Elimination of Common Subheaps

Entailments Between Assertions

Above, we explained a function `Sigma_impl` to prove the validity of the implication between two abstract heaps. Here, we explain how to use this function to verify entailments of assertions.

There are two ways of proving entailments between assertions (type `assrt`). The first way is to prove that the lhs is contradictory (i.e., it implies `False`); this corresponds to the application of the rule `incons` of the `entail` proof system. The second way is to prove the implication between the abstract heaps on both hand sides (using `Sigma_impl`) and to prove the implication between the abstract stores; this corresponds to the application of the rule `tauto` of the `entail` proof system. In order to prove the implication between abstract stores, we need a function to decide Presburger arithmetic; for this purpose, we have certified in Coq a decision procedure based on Fourier-Motzkin variable elimination (this is actually the function `expr_b_dp` that already appears in Fig. 5.3).

This reasoning is implemented by the function `assrt_entail_fun` that extends beforehand the lhs of the entailment with arithmetic constraints, as described at the end of Sect. 5.2.1.

Entailments Between Disjunctions

Above, we explained a function `assrt_entail_fun` to verify entailments of assertions (type `assrt`). Here, we explain how to use this function to verify entailments of disjunctions of assertions (type `Assrt`).

Elimination of Disjunctions in the Lhs To eliminate disjunctions in the lhs of the entailment we use the rule `elim_lhs_disj` (Fig. 5.4, function `Assrt_entail_Assrt_fun` in file `frag_list_entail.v`). Thanks to this rule, we can decompose an entailment between `Assrt` formulas into a list of entailments between an `assrt` formula (on the lhs) and an `Assrt` formula (on the rhs).

Elimination of Disjunctions in the Rhs The elimination of disjunctions in the rhs of the entailment is more subtle. It is possible to use the rule `elim_rhs_disj1` (Fig. 5.4, function `orassrt_impl_Assrt1` in file `frag_list_entail.v`).

$$\frac{\bigwedge_i (\langle \pi_i, \sigma_i \rangle \vdash A)}{(\bigvee_i \langle \pi_i, \sigma_i \rangle) \vdash A}\ \texttt{elim\_lhs\_disj}$$

$$\frac{\bigvee_i (\langle \pi, \sigma \rangle \vdash \langle \pi_i, \sigma_i \rangle)}{\langle \pi, \sigma \rangle \vdash (\bigvee_i \langle \pi_i, \sigma_i \rangle)}\ \texttt{elim\_rhs\_disj1}$$

$$\frac{\begin{array}{c}\pi \rightarrow (\bigvee_i \pi_i) \\ \bigwedge_i (\langle \pi \wedge \pi_i, \sigma \rangle \vdash \langle \texttt{true\_b}, \sigma_i \rangle)\end{array}}{\langle \pi, \sigma \rangle \vdash (\bigvee_i \langle \pi_i, \sigma_i \rangle)}\ \texttt{elim\_rhs\_disj2}$$

Figure 5.4: Entailment of Disjunctions of Assertions

But this rule is not sufficient, as illustrated by the following counter-example:

$$\langle \texttt{true\_b}, \sigma \rangle \vdash \langle \texttt{y} = \texttt{0}, \sigma \rangle \vee \langle \texttt{y} \neq \texttt{0}, \sigma \rangle$$

Such rhs are however important because they are typical of loop invariants. Indeed, a loop invariant usually consists of a disjunction of all possible outcomes of the loop condition, and each disjunct can only be proved under some hypothesis about this outcome. To handle these situations, we use the rule `elim_rhs_disj2` (Fig. 5.4, functions `orpi` and `orassrt_impl_Assrt2` in file `frag_list_entail.v`).

We are now equipped to explain the function `entail_fun`, that proves the validity of entailments. It takes as input an `assrt` and an `Assrt`, uses the rules from Fig. 5.4 to eliminate the disjunctions in the rhs, and finally calls `assrt_entail_fun`:

```
Definition entail_fun
  (a:assrt) (A:Assrt) (l:list (assrt * assrt))
    : result (list (assrt * assrt)) := ...
```

It returns an option type (constructor `Good` if everything is proved). The proof of correctness of `entail_fun` boils down to the following lemma:

```
Lemma entail_fun_correct: ∀ A a l,
  entail_fun a A l = Good →
  assrt_interp a ==> Assrt_interp A.
```

We do not think that the `entail_fun` function is a complete decision procedure because of the rules for entailments between disjunctions. However, it is already useful in practice, as illustrated by the various non-trivial examples in Sect. 5.6.

5.3 Triple Transformation

In the previous section, we saw how to solve entailments of assertions of separation logic. In this section, we explain how to transform a loop-free triple into such an entailment (file `frag_list_triple.v` in [11]).

5.3.1 Language for Weakest-preconditions

Before explaining the triple transformation, we need to introduce the type `wpAssrt`. This type represents the weakest precondition of a program with respect to its postcondition:

```
Inductive wpAssrt : Set :=
| wpElt: Assrt → wpAssrt
| wpSubst: list (var.v * expr)→ wpAssrt→ wpAssrt
| wpLookup: var.v → expr → wpAssrt → wpAssrt
| wpMutation: expr → expr → wpAssrt → wpAssrt
| wpIf: Pi → wpAssrt → wpAssrt → wpAssrt.
```

The constructor `wpElt` represents a postcondition with no program. The `wpSubst` constructor represents the weakest precondition of a sequence of assignments whose postcondition is itself some weakest precondition, etc.

The interpretation of this language is computed by a weakest precondition generator using backward separation logic axioms from [1]:

```
Fixpoint wpAssrt_interp (a: wpAssrt) : assert :=
  match a with
    | wpElt a1 => Assrt_interp a1
    | wpSubst l L =>
      subst_lst2update_store l (wpAssrt_interp L)
    | wpLookup x e L => (fun s h => ∃ e0,
      (e↦e0 ** (e↦e0 -*
      update_store2 x e 0 (wpAssrt_interp L))) s h)
    | wpMutation e1 e2 L => (fun s h => ∃ e0,
      (e1↦e0 ** (e1↦e2 -* wpAssrt_interp L)) s h)
    | wpIf b L1 L2 => (fun s h =>
      (eval_pi b s = true → wpAssrt_interp L1 s h) ∧
      (eval_b b s = false → wpAssrt_interp L2 s h))
  end.
```

5.3.2 Triple Transformation Proof System

Now that we have explained `wpAssrt`, we can explain the role of the `tritra` proof system. It has type `assrt -> wpAssrt -> Prop`. Intuitively, the two parameters form a triple of separation logic: the first parameter is an assertion of separation logic (a precondition) and the second parameter is a weakest precondition, or equivalently a program with a postcondition. The constructors of the `tritra` proof system represent elementary triple transformations. An excerpt in informal notation is displayed in Fig. 5.5.

The two rules `lookup` and `mutation` are intuitive because the lookup (resp. mutation) is the leading command of the program. When lookups and mutations are preceded by assignments, the transformation rules must take care of captures of variables, as exemplified by the rule `subst_lookup`. Despite these technical difficulties (in particular, the usage of fresh variables), we managed to prove the soundness of this proof system inside Coq:

```
Lemma tritra_soundness : ∀ P Q, tritra P Q →
  assrt_interp P ==> wpAssrt_interp Q.
```

5.3.3 Triple Transformation Procedure

Equipped with the `tritra` proof system, we can transform any valid triple $\{P\}c\{Q\}$ into a couple (P, Q') where Q' is a `wpAssrt` of the form `wpElt`. The implication $P \rightarrow Q'$ (or equivalently the entailment $P \vdash Q'$) can then be solved by `entail_fun`. This operation is implemented by the function `tritra_step` of type `Pi -> Sigma -> wpAssrt -> option (list ((Pi * Sigma) * wpAssrt))` that tries to apply `tritra` rules (at the price of some rewriting of the precondition) so as to return a list of subgoals.

$$\frac{\langle \pi_1, \sigma_1 \rangle \vdash \langle \pi_2[v_n/x_n]\cdots[v_1/x_1], \sigma_2[v_n/x_n]\cdots[v_1/x_1]\rangle}{\{\pi_1, \sigma_1\}x_1 \leftarrow v_1; \cdots; x_n \leftarrow v_n\{\pi_2, \sigma_2\}}\ \texttt{subst}$$

$$\frac{\pi_1 \rightarrow v_1 = e_1 \quad \{\pi_1, \sigma_1 \star e_1 \mapsto e_2\}x_1 \leftarrow e_2; c\{\pi_2, \sigma_2\}}{\{\pi_1, \sigma_1 \star e_1 \mapsto e_2\}x_1 \leftarrow\!* \ v_1; c\{\pi_2, \sigma_2\}}\ \texttt{lookup}$$

$$\frac{\begin{array}{c}\{\pi_1, \sigma_1 \star e_1 \mapsto e_2\}x' \leftarrow e_2; x_1 \leftarrow v_1; \cdots; x_n \leftarrow v_n; x \leftarrow x'; c\{\pi_2, \sigma_2\} \\ \pi_1 \rightarrow e_1 = e[v_n/x_n]\cdots[v_1/x_1] \quad \mathsf{fresh}\ x'\end{array}}{\{\pi_1, \sigma_1 \star e_1 \mapsto e_2\}x_1 \leftarrow v_1; \cdots; x_n \leftarrow v_n; x \leftarrow\!* \ e; c\{\pi_2, \sigma_2\}}\ \texttt{subst\_lookup}$$

$$\frac{\pi_1 \rightarrow v_1 = e_1 \quad \{\pi_1, \sigma_1 \star e_1 \mapsto v_2\}c\{\pi_2, \sigma_2\}}{\{\pi_1, \sigma_1 \star e_1 \mapsto e_2\}v_1 *\!\twoheadleftarrow v_2; c\{\pi_2, \sigma_2\}}\ \texttt{mutation}$$

$$\frac{\{\pi_1, \sigma_1\}e[v_n/x_n]\cdots[v_1/x_1] *\!\twoheadleftarrow e'[v_n/x_n]\cdots[v_1/x_1]; x_1 \leftarrow v_1; \cdots; x_n \leftarrow v_n; c\{\pi_2, \sigma_2\}}{\{\pi_1, \sigma_1\}x_1 \leftarrow v_1; \cdots; x_n \leftarrow v_n; e *\!\twoheadleftarrow e'; c\{\pi_2, \sigma_2\}}\ \texttt{subst\_mutation}$$

$$\frac{\{\pi_1 \wedge b, \sigma_1\}c_1\{\pi_2, \sigma_2\} \quad \{\pi_1 \wedge \neg b, \sigma_1\}c_2\{\pi_2, \sigma_2\}}{\{\pi_1, \sigma_1\}\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\{\pi_2, \sigma_2\}}\ \texttt{if}$$

$$\frac{\{\pi_1, \sigma_1\}\texttt{if}\ b[v_n/x_n]\cdots[v_1/x_1]\ \texttt{then}\ x_1 \leftarrow v_1; \cdots; x_n \leftarrow v_n; c_1\ \texttt{else}\ x_1 \leftarrow v_1; \cdots; x_n \leftarrow v_n; c_2\{\pi_2, \sigma_2\}}{\{\pi_1, \sigma_1\}x_1 \leftarrow v_1; \cdots; x_n \leftarrow v_n; \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\{\pi_2, \sigma_2\}}\ \texttt{subst\_if}$$

Figure 5.5: Excerpt of the `tritra` Proof System

The function that implements the whole triple transformation phase is `triple_transformation`: it recursively calls `tritra_step` and then `entail_fun` on resulting subgoals:

```
Fixpoint triple_transformation
  (P : Assrt) (Q : wpAssrt) { struct P }
    : option (list ((Pi * Sigma) * wpAssrt)) := ...

Lemma triple_transformation_correct: ∀ P Q,
  triple_transformation P Q = Some nil  ->
  Assrt_interp P ==> wpAssrt_interp Q.
```

The triple transformation is complete (in the sense that every valid triples can be transformed into an entailment) as long as the intermediate arithmetic goals it generates fall into Presburger arithmetic, which is likely in practice because pointers are never multiplied between each other. The fact that the triple transformation is complete simply comes from the fact the rules of the `tritra` proof system cover all possible programs.

5.4 Verification Conditions Generator

In the previous section, we explained how to prove loop-free separation logic triples. In this section, we explain how to turn a separation logic triple whose loops are annotated with invariants into a list of loop-free triples (file `frag_list_vcg.v` in [11]).

The generation of loop-free triples from a separation logic triple is the role of the verification conditions generator. The main idea of this operation can be explained as follows. Suppose we are given a triple $\{P\}c_1; \texttt{while}_I\ b\ \texttt{do}\ c; c_2\{Q\}$ where I is an invariant. To prove this triple, it is sufficient to prove the three triples $\{P\}c_1\{I\}$, $\{I \wedge b\}c\{I\}$, and $\{I \wedge \neg b\}c_2\{Q\}$. Applying this idea repeatedly turns a separation logic triple into a set of loop-free triples, as implemented by the following function:

```
Fixpoint vcg (c:cmd') (Q:wpAssrt) { struct c }
  : option (wpAssrt * (list (Assrt * wpAssrt))) := ...
```

In addition to a list of subgoals, `vcg` returns the weakest precondition of the program (this is the first projection of the return value in the type above).

The verification of `vcg` amounts to check that, under the hypothesis that subgoals can be verified, the returned condition is indeed a weakest precondition. Recall from Sect. 3.1.2 that separation logic triples are noted `{{·}}·{{·}}`; `Assrt_interp` and `wpAssrt_interp` were defined respectively in Sections 5.1.3 and 5.3.1:

```
Lemma vcg_correct : ∀ c Q Q' l,
  vcg c Q = Some (Q', l) →
  (∀ A L, In (A, L) l →
    Assrt_interp A ==> wpAssrt_interp L) →
  {{ wpAssrt_interp Q' }}
       proj_cmd c
  {{ wpAssrt_interp Q }}.
```

The verification condition generator is complete, as it consists in applying the Reynolds axioms for sequence and loop, which have been proved complete formally inside of Coq (see Sect. 3.1.2).

5.5 Put It All Together

The resulting verification procedure is a Coq function that takes as input a command `c` (annotated with loop invariants), a precondition `P`, and a postcondition `Q`. First, it calls `vcg` to compute a set of sufficient subgoals. Then, it calls `triple_transformation` for all these subgoals. If all of them can be proved, it returns `Some nil`. Otherwise, it returns the list of unsolved subgoals for information:

```
Definition bigtoe_fun (c: cmd') (P Q: Assrt): option (list ((Pi * Sigma) * wpAssrt)) :=
match vcg c (wpElt Q) with
  | None => None
  | Some (Q', l) =>
    match triple_transformation P Q' with
      | Some l' =>
        match triple_transformations l with
          | Some l'' => Some (l' ++ l'')
          | None => None
        end
      | None => None
    end
end.
```

The correctness of this tactic amounts to prove that, if it returns `Some nil`, then the corresponding separation logic triple holds:

```
Lemma bigtoe_fun_correct: ∀ P Q c,
  bigtoe_fun c P Q = Some nil →
  {{ Assrt_interp P }}
       proj_cmd c
  {{ Assrt_interp Q }}.
```

Now, in our formal proofs of Hoare triples, we can apply this lemma to delegate the proof to the computation of the function `bigtoe_fun`.

$$
\begin{array}{cl}
\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vy}{*}{*}\mathtt{y}\mapsto\mathtt{vx}\rangle \vdash \langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vy}{*}{*}\mathtt{y}\mapsto\mathtt{vx}\rangle & \\ \hline
\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vy}{*}{*}\mathtt{y}\mapsto\mathtt{vx}\rangle\}\mathtt{t2'} \leftarrow \mathtt{vy}; \mathtt{t1} \leftarrow \mathtt{vx}; \mathtt{t2} \leftarrow \mathtt{t2'}\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vy}{*}{*}\mathtt{y}\mapsto\mathtt{vx}\rangle\} & \mathtt{subst\_elt} \\ \hline
\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vy}{*}{*}\mathtt{y}\mapsto\mathtt{vy}\rangle\}\underline{\mathtt{y} \mathbin{*\!\!\leftarrow} \mathtt{vx}}; \mathtt{t2'} \leftarrow \mathtt{vy}; \mathtt{t1} \leftarrow \mathtt{vx}; \mathtt{t2} \leftarrow \mathtt{t2'}\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vy}{*}{*}\mathtt{y}\mapsto\mathtt{vx}\rangle\} & \mathtt{mutation} \\ \hline
\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vy}{*}{*}\mathtt{y}\mapsto\mathtt{vy}\rangle\}\mathtt{t2'} \leftarrow \mathtt{vy}; \mathtt{t1} \leftarrow \mathtt{vx}; \mathtt{t2} \leftarrow \mathtt{t2'}; \underline{\mathtt{y} \mathbin{*\!\!\leftarrow} \mathtt{t1}}\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vy}{*}{*}\mathtt{y}\mapsto\mathtt{vx}\rangle\} & \mathtt{subst\_mutation} \\ \hline
\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vx}{*}{*}\mathtt{y}\mapsto\mathtt{vy}\rangle\}\underline{\mathtt{x} \mathbin{*\!\!\leftarrow} \mathtt{vy}}; \mathtt{t2'} \leftarrow \mathtt{vy}; \mathtt{t1} \leftarrow \mathtt{vx}; \mathtt{t2} \leftarrow \mathtt{t2'}; \mathtt{y} \mathbin{*\!\!\leftarrow} \mathtt{t1}\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vy}{*}{*}\mathtt{y}\mapsto\mathtt{vx}\rangle\} & \mathtt{mutation} \\ \hline
\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vx}{*}{*}\mathtt{y}\mapsto\mathtt{vy}\rangle\}\mathtt{t2'} \leftarrow \mathtt{vy}; \mathtt{t1} \leftarrow \mathtt{vx}; \mathtt{t2} \leftarrow \mathtt{t2'}; \underline{\mathtt{x} \mathbin{*\!\!\leftarrow} \mathtt{t2}}; \mathtt{y} \mathbin{*\!\!\leftarrow} \mathtt{t1}\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vy}{*}{*}\mathtt{y}\mapsto\mathtt{vx}\rangle\} & \mathtt{subst\_mutation} \\ \hline
\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vx}{*}{*}\mathtt{y}\mapsto\mathtt{vy}\rangle\}\mathtt{t1} \leftarrow \mathtt{vx}; \underline{\mathtt{t2} \mathbin{\leftarrow\!\!*} \mathtt{y}}; \mathtt{x} \mathbin{*\!\!\leftarrow} \mathtt{t2}; \mathtt{y} \mathbin{*\!\!\leftarrow} \mathtt{t1}\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vy}{*}{*}\mathtt{y}\mapsto\mathtt{vx}\rangle\} & \mathtt{subst\_lookup} \\ \hline
\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vx}{*}{*}\mathtt{y}\mapsto\mathtt{vy}\rangle\}\underline{\mathtt{t1} \mathbin{\leftarrow\!\!*} \mathtt{x}}; \mathtt{t2} \mathbin{\leftarrow\!\!*} \mathtt{y}; \mathtt{x} \mathbin{*\!\!\leftarrow} \mathtt{t2}; \mathtt{y} \mathbin{*\!\!\leftarrow} \mathtt{t1}\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vy}{*}{*}\mathtt{y}\mapsto\mathtt{vx}\rangle\} & \mathtt{lookup}
\end{array}
$$

Figure 5.6: Swap of Cells using our Proof System

$$
\begin{array}{c}
\langle \mathtt{t1} = \mathtt{vx} \wedge \mathtt{t2} = \mathtt{vy}, \mathtt{x}\mapsto\mathtt{t2}{*}{*}\mathtt{y}\mapsto\mathtt{t1}\rangle \vdash \langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vy}{*}{*}\mathtt{y}\mapsto\mathtt{vx}\rangle \\ \hline
\{\langle \mathtt{t1} = \mathtt{vx} \wedge \mathtt{t2} = \mathtt{vy}, \mathtt{x}\mapsto\mathtt{t2}{*}{*}\mathtt{y}\mapsto\mathtt{vy}\rangle\}\mathtt{y} \mathbin{*\!\!\leftarrow} \mathtt{t1}\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vy}{*}{*}\mathtt{y}\mapsto\mathtt{vx}\rangle\} \\ \hline
\{\langle \mathtt{t1} = \mathtt{vx} \wedge \mathtt{t2} = \mathtt{vy}, \mathtt{x}\mapsto\mathtt{vx}{*}{*}\mathtt{y}\mapsto\mathtt{vy}\rangle\}\mathtt{x} \mathbin{*\!\!\leftarrow} \mathtt{t2}; \mathtt{y} \mathbin{*\!\!\leftarrow} \mathtt{t1}\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vy}{*}{*}\mathtt{y}\mapsto\mathtt{vx}\rangle\} \\ \hline
\{\langle \mathtt{t1} = \mathtt{vx}, \mathtt{x}\mapsto\mathtt{vx}{*}{*}\mathtt{y}\mapsto\mathtt{vy}\rangle\}\mathtt{t2} \mathbin{\leftarrow\!\!*} \mathtt{y}; \mathtt{x} \mathbin{*\!\!\leftarrow} \mathtt{t2}; \mathtt{y} \mathbin{*\!\!\leftarrow} \mathtt{t1}\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vy}{*}{*}\mathtt{y}\mapsto\mathtt{vx}\rangle\} \\ \hline
\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vx}{*}{*}\mathtt{y}\mapsto\mathtt{vy}\rangle\}\mathtt{t1} \mathbin{\leftarrow\!\!*} \mathtt{x}; \mathtt{t2} \mathbin{\leftarrow\!\!*} \mathtt{y}; \mathtt{x} \mathbin{*\!\!\leftarrow} \mathtt{t2}; \mathtt{y} \mathbin{*\!\!\leftarrow} \mathtt{t1}\{\langle \mathtt{true\_b}, \mathtt{x}\mapsto\mathtt{vy}{*}{*}\mathtt{y}\mapsto\mathtt{vx}\rangle\}
\end{array}
$$

Figure 5.7: Swap of Cells using Forward Reasoning

5.6 Experimental Measurements

In this section, we present a comparison between our approach and backward/forward reasoning, as well as a benchmark for our verifier.

5.6.1 Comparison With Backward and Forward Reasoning

All previous work on automatic verification of separation logic triples use forward reasoning [18, 31, 33]. The main reason is that backward reasoning (using a standard weakest precondition generator for separation logic) produces postconditions with separating implications for which there exists no automatic prover (as pointed out in [18]). Although decidability results exist [27, 29, 28], the separating implication is actually seldom used in specifications of algorithms (one notable exception is [26]). However, forward reasoning has the disadvantage of adding, for each variable modification, a conjunctive clause with possibly a fresh variable, as it uses the Floyd rule for assignment. This is not desirable in practice because decision procedures for Presburger arithmetic have an exponential complexity w.r.t. the number of clauses and variables. Our approach based on the proof system `tritra` can be shown experimentally to produce less clauses.

In Fig. 5.6, we illustrate transformation steps for a program swapping the values of two cells, using our approach. The transformations produced by forward and backward reasoning are displayed in Figures 5.7 and 5.8. We can observe that `tritra` does not add new connectives or variables, contrary to both backward and forward reasoning. (For the latter, no fresh variables have been introduced, because the variables modified by the program do not appear in the precondition.)

In order to measure more precisely differences between our approach and forward reasoning, we have implemented, inside of Coq, a proof system similar to [18] extended with pointer arithmetic (file `LSF.v` in [11]). We proved inter-

$$\begin{array}{c}
\langle \texttt{true\_b}, \texttt{x} \mapsto \texttt{vx**y} \mapsto \texttt{vy}\rangle \vdash \exists \texttt{v4}, \texttt{x} \mapsto \texttt{v4**(x} \mapsto \texttt{v4} \mathrel{-\!\!*} (\exists \texttt{v3}, \texttt{y} \mapsto \texttt{v3**(y} \mapsto \texttt{v3} \mathrel{-\!\!*} (\exists \texttt{v2}, \ldots))))\\
\hline
\{\langle \texttt{true\_b}, \texttt{x} \mapsto \texttt{vx**y} \mapsto \texttt{vy}\rangle\}\texttt{t1} \mathrel{\texttt{<-*}} \texttt{x}\{\exists \texttt{v3}, \texttt{y} \mapsto \texttt{v3**(y} \mapsto \texttt{v3} \mathrel{-\!\!*} (\exists \texttt{v2}, \texttt{x} \mapsto \texttt{v2**(x} \mapsto \texttt{t1} \mathrel{-\!\!*} (\exists \texttt{v1}, \ldots)))))\}\\
\hline
\{\langle \texttt{true\_b}, \texttt{x} \mapsto \texttt{vx**y} \mapsto \texttt{vy}\rangle\}\texttt{t1} \mathrel{\texttt{<-*}} \texttt{x}; \texttt{t2} \mathrel{\texttt{<-*}} \texttt{y}\{\exists \texttt{v2}, \texttt{x} \mapsto \texttt{v2**(x} \mapsto \texttt{t2} \mathrel{-\!\!*} (\exists \texttt{v1}, \ldots))\}\\
\hline
\{\langle \texttt{true\_b}, \texttt{x} \mapsto \texttt{vx**y} \mapsto \texttt{vy}\rangle\}\texttt{t1} \mathrel{\texttt{<-*}} \texttt{x}; \texttt{t2} \mathrel{\texttt{<-*}} \texttt{y}; \texttt{x} \mathrel{\texttt{*<-}} \texttt{t2}\{\exists \texttt{v1}, \texttt{y} \mapsto \texttt{v1**(y} \mapsto \texttt{t1} \mathrel{-\!\!*} \langle \texttt{true\_b}, \texttt{x} \mapsto \texttt{vy**y} \mapsto \texttt{vx}\rangle)\}\\
\hline
\{\langle \texttt{true\_b}, \texttt{x} \mapsto \texttt{vx**y} \mapsto \texttt{vy}\rangle\}\texttt{t1} \mathrel{\texttt{<-*}} \texttt{x}; \texttt{t2} \mathrel{\texttt{<-*}} \texttt{y}; \texttt{x} \mathrel{\texttt{*<-}} \texttt{t2}; \texttt{y} \mathrel{\texttt{*<-}} \texttt{t1}\{\langle \texttt{true\_b}, \texttt{x} \mapsto \texttt{vy**y} \mapsto \texttt{vx}\rangle\}
\end{array}$$

Figure 5.8: Swap of Cells using Backward Reasoning

Program	`tritra`	forward reasoning
`swap`	16	20 (+19%)
`init (5)`	46	69 (+33%)
`init (10)`	138	225 (+38%)
`init (15)`	195	320 (+39%)
`max3`	9.0	7.9 (−14%)

Table 5.1: Size of Proof-terms files in kbytes

actively several separation logic triples, and compared the size of the compiled proofs terms produced by both approaches. This comparison was done on three different programs. `swap` is the separation logic triple whose transformation is illustrated in Fig. 5.6. The `init(n)` program is a loop that initializes a given field for `n` contiguous occurrences of a data-structure. This program makes use of pointer arithmetic, as the loop iteratively increments the value of the pointer to the current data-structure, while the data-structures locations are specified by a multiple of the data-structure's size in the pre/postconditions[1]. Finally, `max3` is a program that returns the maximum value of three variables. The results are presented in Table 5.1, where the percentages correspond to the overhead of forward reasoning. We can conclude that our approach produces smaller proof-terms, because the underlying arithmetic decision procedure (here, the Coq `omega`) applies less lemmas to prove the goals.

5.6.2 The Extracted OCaml Verifier

Thanks to the extraction facility of Coq, we can extract the verification function `bigtoe_fun` (and its underlying functions and data structures) in the OCaml language. The certified verifier is in file `extracted.ml` in [11]. We use OCaml-yacc to parse the input language (files `lexer.mll` and `grammar.mly`). The resulting verifier can handle three kind of goals: (1) arithmetic formulas (for which all variables are universally quantified), (2) entailments between assertions of `Assrt`, and (3) separation logic triples. As the verification functions return a list of unsolved subgoals, the verifier is able to print these subgoals to help for the debugging of program specifications.

We measure the performance of the OCaml verifier. The first version uses a decision procedure for arithmetic based on variable elimination using the Fourier-Motzkin theorems (FMVE). This is a decision procedure by reflection

[1]As there is no universal quantification in our assertion language, the behavior of `init(n)` is specified for only one arbitrary value, and the programs and pre/postconditions are computed by Coq functions.

Program	FMVE	Cooper
Reverse list	0.240 s	0.111 s
List traversal	0.160 s	0.085 s
List append	147.593 s	0.660 s
Insert head	0.009 s	0.108 s
Insert tail	unknown	2.580 s

Table 5.2: Execution Time

that we have implemented for our verifier (the `omega` tactic of Coq cannot be used because it is not implemented by reflection). Of course, this decision procedure has also been verified in Coq (file `expr_b_dp.v` in [11]). The second version uses a non-certified decision procedure based on the Cooper algorithm [34]. The reason why we provide this second version is that our decision procedure for arithmetic, though necessary for use inside of Coq, is not optimized enough to solve large arithmetic subgoals. A certified implementation of a more efficient decision procedure (such as the Cooper algorithm) is among our future work (Chaieb and Nipkow already did this work in the Isabelle proof assistant [30]). Table 5.2 summarizes the measurements (hardware: Pentium IV 2.4GHz with 1GB of RAM).

Here follows a brief description of the benchmark programs: `Reverse list` is an in-place reversal of a list as the one described in [1], `List traversal` is a program that iteratively explores each element of a list, `List append` appends two lists, and `Insert head` (resp. `Insert tail`) inserts an element at the head (resp. tail) of a list.

The extracted verifier using the Cooper algorithm is available for download and testing through a Web interface, see [11].

5.6.3 Code Verification

Beyond the programs verified for benchmarks, we also applied our verifier on more concrete examples. Mostly two experiences were run: an implementation and verification of several singly-linked list functions which mimics the Topsy list library functions, and a simplified version of the Topsy function that initializes the thread data-structure. We present these experiences in the following sections (please note that the illustrating triples are in ascii, and that they exactly correspond to the input for our verifier).

A List Library

Originally the list library of Topsy makes use of a doubly-linked list data-structure and memory allocations. However, our verifier currently deals with only a singly-linked list data-structure, and does not provide command for memory allocation. Thus, rather than a straightforward verification of Topsy list library, we present the verification for an adaptation of the library to our verifier. Yet, both codes share the same principles.

Topsy defines a data-structure, named `ListDesc`, that describes a list through three pointers: `first` points to the first element of the list, `current` points to some elements of the list on which the system is focusing, and finally `last` points to the last element of the list. As previously stated, the list managed by Topsy are doubly-linked, which means that elements have three fields: one for the data of the element, and two pointers for, respectively, the previous and the next elements. For our verifications, we will keep the notion of list descriptors, but we will considers only elements that point to their next element (because our verifier handles singly-linked lists).

Topsy provides several functions to remove or to add elements in a list, as well as a set a functions that allow to get the data of the first or the current elements (pointed by `current`). We propose to illustrate the use of bigtoe for some of these functions. In addition of the Topsy functions, we propose the verification of a search function to illustrate a specification containing a loop invariant.

listAddAtEnd The Topsy function, `listAddAtEnd`, originally inserts an element with a given data at the end of some list. This new element is created using the memory allocator of Topsy. For the verification of the function, we take as hypothesis (captured by the precondition), that there is a free element, or more precisely two contiguous cells, inside the memory (here pointed by `elem`). Bigtoe specification triple for the `listAddAtEnd` is:

```
{first <> 0 /\ current <> 0 /\ last <> 0 |
  (lst |-> first) ** (lst+1 |-> current) ** (lst+2 |-> last) **
  (list first current) ** (list current last) ** (last |-> 0) ** (last+1 |-> data) **
  (elem |->_) ** (elem+1 |->_)
}

if (lst == 0) then {
  skip
} else {
  tmp <-* lst;
  if (tmp == 0) then {
    first *<- elem;
    first *<- last;
    elem *<- 0
  } else {
    last *<- elem;
    (elem + 1) *<- data2;
    elem *<- 0;
    (lst + 2) *<- elem
  }
}

{first <> 0 /\ current <> 0 /\ last <> 0 |
  (lst |-> first) ** (lst+1 |-> current) ** (lst+2 |-> elem) **
  (list first current) ** (list current elem) ** (elem |-> 0) ** (elem+1 |-> data2)
}
```

Both pre/post-conditions are composed of two parts separated with the symbol `|`. The l.h.s. correspond to arithmetic constraints over the variables (more informally, an assertion over the store), and the r.h.s is a separation logic assertion over the heap. The precondition asserts that there is a list descriptor pointed by `lst`. The list starts from the location pointed by `first` and stops at a location pointed by `last`, passing through an element pointed by `current`. The last element of the list points to a null pointer and contains some data. A fresh list element, pointed by `elem`, is also present in the list. The arithmetic constraints assert that both lists are not empty. The `listAddAtEnd` function makes the last element to point to the fresh element, makes this fresh element to point to a null pointer and updates it with a given data (`data2`), and finally updates the list descriptor such that it last element field in up-to-date. Please remark that due to our precondition, the verification will only consider the second branch of both if-then-else control-flow. Bigtoe verifies this specification in around 9 seconds.

listGetNext This function makes use of the list descriptor to return the current element data and to update the current element field with the next element. Here follows the bigtoe specification:

```
{ hd <> null |
  lst -.> 0 |-> hd ** lst -.> 1 |-> current ** lst -.> 2 |-> lastcell **
  list hd current **
  current -.> 0 |-> nxt ** current -.> 1 |-> elt **
  list nxt 0
}

if (lst == 0) then {
  return <- LISTERROR
} else {
  tmp <-* (lst -.> 1);
  if (tmp == 0) then {
    itemPtr <- 0
  } else {
    itemPtr <-* (tmp -.> 1);
    tmp <-* (tmp -.> 0);
    (lst -.> 1) *<- tmp
  };
  return <- LISTOK
}

{return == LISTOK /\ itemPtr == elt |
  lst -.> 0 |-> hd ** lst -.> 1 |-> nxt ** lst -.> 2 |-> lastcell **
  list hd current **
  current -.> 0 |-> nxt ** current -.> 1 |-> elt **
  list nxt 0
  }
```

In the precondition we asserts that there exists a list in the memory, from `hd` to `current`, a list from `next` to a null pointer, and in-between them the list element pointed by `current`. This is the current element of the list. The post-condition assert that the variable `itemPtr` contains the data of this element, and that the current element is now `nxt`, pointed by the second field of the list descriptor. Bigtoe verifies this specification in around 2 seconds.

listSearch The original library of Topsy does not include the search of an element, yet most of storage libraries include one. We present the specification of such a search function for the list data-structure. This allows us to highlight how to write a specification with a loop invariant inside bigtoe. In the precondition, we asserts that there is a list descriptor, pointed by `lst`, and a list pointed by `hd`. The program tries to find the pointer of the first element which data is `elt`. The if-then-else command allows to takes in account the case of an empty list, and the while loops performs a list traversal that stops either if an element containing the data `elt` is found (in which case it is pointed by `found`), or that the end of the list is reached (in this case `found` is the null pointer). The post-condition asserts that: if an element has been found then its data is indeed `elt`, and if no element has been found then the memory did not change in shape. The bigtoe specification is:

```
{TT |
  lst -.> 0 |-> hd ** lst -.> 1 |-> cur ** lst -.> 2 |-> lastcell **
  list hd 0
}

lstElem <-* (lst -.> 0);
found <- 0;
if (lstElem == 0) then {
  tmp <- 0
} else {
  tmp <-* (lstElem -.> 1);
  nxt <-* (lstElem -.> 0)
};
while (lstElem <> 0 /\ tmp <> elt) [
  (lstElem <> 0 |
  lst -.> 0 |-> hd ** lst -.> 1 |-> cur ** lst -.> 2 |-> lastcell **
  list hd lstElem ** lstElem -.> 0 |-> nxt **
  lstElem -.> 1 |-> tmp ** list nxt 0) \/
  (lstElem == 0 /\ tmp == 0 |
  lst -.> 0 |-> hd ** lst -.> 1 |-> cur ** lst -.> 2 |-> lastcell **
  list hd lstElem)
]{
  lstElem <-* (lstElem -.> 0);
  if (lstElem == 0) then {
    tmp <- 0
  } else {
    tmp <-* (lstElem -.> 1);
    nxt <-* (lstElem -.> 0)
  }
};
found <- lstElem

{
  (found <> 0 |
  lst -.> 0 |-> hd ** lst -.> 1 |-> cur ** lst -.> 2 |-> lastcell **
  list hd found ** found -.> 0 |-> nxt ** found -.> 1 |-> elt ** list nxt 0) \/
  (found == 0 |
  lst -.> 0 |-> hd ** lst -.> 1 |-> cur ** lst -.> 2 |-> lastcell **
  list hd 0)
}
```

The interesting element of this specification is the loop invariant. It asserts that either we are investigating an element of the list, or that we have reached the end (in this case `lstElem` is null). Bigtoe takes around 14 seconds to verify such non-trivial triple.

Topsy Thread Creation

The function that initializes the field of the thread descriptor is named `threadBuild`. Although it initializes all the fields, our experiences over the SPIN informs us that we are mostly interested in one of the thread attributes: the thread privilege. In Topsy source code, this information is stored inside a data-structure that is part of the thread descriptor. We restrict the verification of `threadBuild` to the part of its code that is responsible for the privilege initialization. The bigtoe specification for this snippet is:

```
{TT |
  threadPtr -.> contextPtr |->_ **
  acontextPtr -.> status |->_
}

if (space == USER) then {
  mode <- USER
} else {
  mode <- KERNEL
};
(threadPtr -.> contextPtr) *<- acontextPtr;
k <-* threadPtr -.> contextPtr);
(k -.> status) *<- mode

{
  (space == USER |
  threadPtr -.> contextPtr |-> acontextPtr **
  acontextPtr -.> status |-> USER
  ) \/
  (space <> USER |
  threadPtr -.> contextPtr |-> acontextPtr **
  acontextPtr -.> status |-> KERNEL
  )
}
```

The function first tests the variable `space`, that is set to the value `USER` in case we initialize an user thread. Depending on this variable value, the `mode` variable is set either to `USER` or to `KERNEL`. Then, the function attaches to the thread descriptor a previously allocated `contextPtr` data-structure (which store all architecture dependant attributes of a thread) . Finally, the function initializes this data-structure, notably the field `status` that corresponds to the thread privilege. The bigtoe specification asserts that if we create a user thread, then its privilege is `USER`. Although this snippet of code is simple, its verification is important. Bigtoe allows to verify automatically such code, which verification by hand may be cumbersome, due to the fact that it requires all the intermediate assertions.

5.7 Related Work

Our main contribution w.r.t. related work is to provide a *certified* automatic verifier for separation logic triples.

Berdine et al. have developed Smallfoot, a tool for checking separation logic specifications [18]. It uses symbolic, forward execution to produce verification conditions, and a decision procedure to prove them. Although Smallfoot is automatic (even for recursive and concurrent procedures), the assertion language does not allow to deal with pointer arithmetic.

Calcagno et al. have proposed an extension of Smallfoot to verify automatically memory allocators [31]. More precisely, the assertion language is extended with: arithmetic, advanced data-structures (lists with variable-size arrays), and abstract interpretation, allowing to compute automatically loop invariants. A prototype has been developed and used on several examples, such as the Kernighan allocator.

A verifier for separation logic with user-defined data-structure has been proposed in [33]. This verifier uses folding/unfolding of data-structures definitions to prove entailments. A prototype has been developed and used for verification of several functions with advanced invariants.

We believe that the algorithms implemented in these last two work are so complex that verification in Coq would be an order of magnitude harder than the work presented in this paper.

Chapter 6

A Certified Translator

In Chap. 3 and Chap. 4 we have shown how to build verification for C and assembly source code. Now, we may ask ourselves how these verifications can be composed. This is an important issue when we deal with heterogeneous source code, like operating system kernel code. Indeed, through the process of compilation and linking, all the source code is lifted to the machine code level. In this chapter we investigate how one can reused the previously verified Hoare-triples and composed them into a Hoare-triples for the whole compiled code. Our approach is to lift the triples from C-like code to the level of the assembly code triples, in an automatic way. To achieve this goal, we build a translator from C-like to assembly and prove that the translation preserves the semantics of the source program, and by extension preserves the Hoare-triples.

This chapter is organized as follows. In Sect. 6.1 we present the translator. First, we present the languages it manipulates in Sect. 6.1.1. Then we describe the translation stages in Sect. 6.1.2, and finally how they are composed in Sect. 6.1.2. We present the preservation for Hoare-triples in Sect. 6.1.3. In Sect. 6.1.4, we present a method to compose the Hoare-triples of programs written with different languages. In Sect. 6.2, we discuss several technical points of the implementation of a certified translator inside the Coq proof assistant. Finally, we present the related work in Sect. 6.3

6.1 The Translator

In this section, we present a certified translator from a structured C-like language (the *source language*) to an assembly language (the *target language*). This translator is basically an implementation of the work describe in [2], extended with function calls. The main purpose of this translator is to allow the composition of Hoare-triples given by verification of the C-like and assembly source code. The different ways to compose the verification are captured by lemmas formally proved inside the Coq proof assistant.

A specificity of our implementation is the effort to provide a translator as generic as possible. Indeed the translator mainly expands the control-flow, translating a program written in a structured language (with loops and branches) into a program written in an unstructured language (with jumps). This implies that the translator only focus on the commands that implement control-flow.

Thus, the instructions that modify the states are irrelevant in the translation process. To emphasizes this fact we factorize the translator by the state-modifying instruction language, in such a way that one can specialize it for several instruction sets (for example the separation-logic, c.f. Sect. 2.1).

In a first step, we present the three languages that are used by the translator through there syntax, semantics, purposes and implementations. Then, we present the two translation steps that our translator implements. Finally, we present the main lemmas that captures the composition of Hoare-triples.

6.1.1 Translator Languages

Our translator manipulates three different languages: the source, the intermediate and the target languages. The source and target languages correspond respectively to the format of the input and output of the translator. The intermediate language is used to split the translation process into two stages, which allow to simplify the design of the translator.

As explained in Sect. 6.1, these languages are factorized by a state-modifying instruction language. This languages is shared by all the translator stages. More precisely, the translator is parameterized by the following sets: (1) `state` represents the data-structure that the programs use to store information, (2) `bexpr` is a language of boolean expression language that are evaluated, by the function `beval`, through a `state`, and (3) `insn` is the language of the state modifying instructions, which semantics is defined by `insn_semop`. This later is defined as a relation between a starting `state`, an element of `insn` and a final state (which may be an error state). In a more formal way, the semantics has the following Coq definition:

```
Variable insn_semop: state → insn → option state → Prop.
```

Informally, we note this semantics relation as $st \mathrel{—} i \mathrel{\rightarrowtail} st'$. We use the same notation for evaluation of boolean expressions through states as in Sect. 2.1 ($[\![\_]\!]_\_$).

Source Language

The source language is the input language of the translator. This is basically a Hoare-style command language. Its syntax is presented in a formal shape (more precisely in Coq) in Fig. 6.1. Please note that we extend the classical Hoare command language with a new command: the function call. This function call does not uses arguments and does not return a value. The callee rather directly uses the state to get its parameters and to return values.

The semantics of the source language is described informally (respectively formally in Coq) in Fig. 6.2 (resp. Fig. A.1). This semantics includes a function environment: a partial map from functions names to commands. This environment is essential to define a semantics for the `call` command. In Coq, this map in encoded as a list of couples of functions names and functions bodies (the type `fenv1`). The function `get_cmd1_f` is used to retrieve the body of a function which names is given as an argument.

```
Inductive cmd1: Set :=
| skip1: cmd1
| mutate1: insn → cmd1
| seq1: cmd1 → cmd1 → cmd1          (* Notation: c1; c2 *)
| ifte1: bexpr → cmd1 → cmd1 → cmd1 (* Notation: if b then c1 else c2 *)
| while1: bexpr → cmd1 → cmd1       (* Notation: while b do c *)
| call1: string → cmd1.             (* Notation: call s *)

Definition fenv1 := list (string * cmd1).

Fixpoint get_cmd1_f (f: fenv1) (s: string) struct f : option cmd1 :=
  match f with
    | nil ⇒ None
    | (hd1, hd2)::tl ⇒
      if (string_dec hd1 s) then (Some hd2) else (get_cmd1_f tl s)
  end.
```

Figure 6.1: Coq definition of source language commands and function environment.

Target Language

The target language corresponds to the output of our translator. It describes a set of raw and stand-alone programs (also known as statically linked programs). It has an unstructured control-flow (with jump commands) and address-oriented function calls: the callee is characterized by its starting address inside the whole program. We present the syntax of the target language formally in Coq in Fig. 6.3. The set of instructions is defined by the inductive type `insn3`, while `cmd3` defines a program of the target language as a list of instructions.

The target language is described by a small-step operational semantics: a relation that characterized the execution of one instruction, through an initial and a final state. The states of the target language are triples containing: (1) the current instruction (also referred as the *code pointer*), (2) a `state`, and (3) a stack that keeps track of the return address of function calls. A `call` instruction modifies the code pointer to the callee and pushes the return address (captured by the rule `call` in 6.4, and `call3_semop` in A.2), while a `ret` pops the top of the stack as the new code pointer.

In the Coq definition of the semantics, we use the function `get_insn3` to find an instruction inside a program. The function takes as parameters (in order): (1) a target language program, (2) its starting address, and (3) the address of the searched instruction. This function returns an option type (of instruction), to capture the fact that no instruction is located at the specified address.

Given this small-step operational semantics, we build a big-step operational semantics (as in the source language) through a reflexive and transitive closure, implemented by the Coq inductive type `cmd3_closure`. For sake of readability we present an informal version of the small-step operational semantics in Fig. 6.4. In this definition, we note the stack as a list, with `::` as the cons operator.

Intermediate Language

The intermediate language is an unstructured control-flow language (similarly to the target language), where functions are stored in a function environment (similarly to the source language). This in-between nature allows to split the

$$\frac{}{f \vdash \texttt{Abort} =\!= c =\!=\!\!\Rightarrow \texttt{Abort}}\ \texttt{abort} \qquad \frac{}{f \vdash st =\!= (\texttt{skip}) =\!=\!\!\Rightarrow st}\ \texttt{skip}$$

$$\frac{st - i \rightarrow st'}{f \vdash st =\!= i =\!=\!\!\Rightarrow st'}\ \texttt{mutate} \qquad \frac{st - i \rightarrow \texttt{Abort}}{f \vdash st =\!= i =\!=\!\!\Rightarrow \texttt{Abort}}\ \texttt{mutate\_error}$$

$$\frac{[\![b]\!]_{st} = true \qquad st =\!= c_1 =\!=\!\!\Rightarrow st'}{st =\!= (\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2) =\!=\!\!\Rightarrow st'}\ \texttt{if\_true}$$

$$\frac{[\![b]\!]_{st} = false \qquad st =\!= c_2 =\!=\!\!\Rightarrow st'}{st =\!= (\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2) =\!=\!\!\Rightarrow st'}\ \texttt{if\_false}$$

$$\frac{[\![b]\!]_{st} = false}{st =\!= (\texttt{while}\ b\ \texttt{do}\ c) =\!=\!\!\Rightarrow st'}\ \texttt{while\_false}$$

$$\frac{[\![b]\!]_{st} = true \qquad st =\!= c =\!=\!\!\Rightarrow st' \qquad st' =\!= (\texttt{while}\ b\ \texttt{do}\ c) =\!=\!\!\Rightarrow st''}{st =\!= (\texttt{while}\ b\ \texttt{do}\ c) =\!=\!\!\Rightarrow st''}\ \texttt{while\_true}$$

$$\frac{[\![fct]\!]_f = c \qquad f \vdash st =\!= c =\!=\!\!\Rightarrow st'}{f \vdash st =\!= (\texttt{call}\ \ fct) =\!=\!\!\Rightarrow st'}\ \texttt{call}$$

$$\frac{[\![fct]\!]_f = none}{f \vdash st =\!= (\texttt{call}\ \ fct) =\!=\!\!\Rightarrow \texttt{Abort}}\ \texttt{call\_err}$$

Figure 6.2: Informal source language semantics.

```
Inductive insn3: Set :=
| skip3: insn3
| mutate3: insn → insn3
| jmp3: Z → insn3
| ifnjmp3: bexpr → Z → insn3
| call3: Z → insn3
| ret3: insn3.

(** Definition of a target language command: a list of instruction*)

Definition cmd3 := (list insn3).
```

Figure 6.3: Informal target language definition.

translation process into two stages: first, we expand the loops and branches into unstructured control-flow through jumps, and secondly, we link all functions into a raw program. Such a design focuses on several desired properties: (1) an easy way to build an intermediate program through control-flow expansion of a source language program, (2) an easy way to link all the functions of an intermediate language program into a target language program, and (3) a semantics smart enough to ease the proof of both translation steps semantics preservation lemmas.

This last point motives us to choose a semantics similar to the one of `SGoto`, presented in [2]. Its particularity is to give a big-step operational semantics to an unstructured language. The main reason for this is that the proof of correctness of the second translation step of the translator consists in an induction over the intermediate language semantics. Technically, such proofs by induction over closure (needed when dealing with small-step operational semantics) present delicate issues. The most difficult point is to deal with the subgoal related to the transitive closure. In this case an intermediate state is created in the hypothesizes. Unfortunately, this state generally lacks hypothesis about its

$$\frac{}{(l, st, stk) =\!= (\texttt{skip}) =\!\!\Rightarrow (l, st, stk)}\ \texttt{skip}$$

$$\frac{st \text{ — } i \text{ -> } st'}{(l, st, stk) =\!= i =\!\!\Rightarrow (l+1, st', stk)}\ \texttt{mutate} \qquad \frac{st \text{ — } i \text{ -> } \texttt{Abort}}{(l, st, stk) =\!= i =\!\!\Rightarrow \texttt{Abort}}\ \texttt{mutate\_error}$$

$$\frac{}{(l, st, stk) =\!= (\texttt{jmp}\ l') =\!\!\Rightarrow (l', st, stk)}\ \texttt{jmp}$$

$$\frac{[\![b]\!]_{st} = true}{(l, st, stk) =\!= (\texttt{ifnjmp}\ b\ l') =\!\!\Rightarrow (l+1, st, stk)}\ \texttt{ifnjmp\_true}$$

$$\frac{[\![b]\!]_{st} = false}{(l, st, stk) =\!= (\texttt{ifnjmp}\ b\ l') =\!\!\Rightarrow (l', st, stk)}\ \texttt{ifnjmp\_false}$$

$$\frac{}{(l, st, stk) =\!= (\texttt{call}\ l') =\!\!\Rightarrow (l', st, (l+1)::stk)}\ \texttt{call}$$

$$\frac{}{(l, st, l'::stk) =\!= (\texttt{ret}) =\!\!\Rightarrow (l', st, stk)}\ \texttt{ret}$$

Figure 6.4: Informal target language semantics.

properties.

We present formally the syntax of the intermediate language in Fig. 6.5. Similarly to the target language, this language includes jump instructions. Yet, similarly to the source language, the syntax also includes a sequence instruction. Another similar feature is that functions bodies (here defined as `block2`) are stored inside a function environment (the `fenv2` type).

As stated previously, the intermediate language is described by a big-step operational semantics. We informally (respectively formally) present its definition in Fig. 6.6 (resp. in Fig. A.3). The rules for atomic instructions are similar to the ones for the target language. The specificity of this semantics are located in the rules for the sequence, and the reflective closure. These rules make use of a notion of size for command. This measure corresponds to the number of atomic instructions that compose a command.

Let us explain in more details the sequence rules. Through the starting state, we can find on which side of the sequence is the current instruction (this condition correspond to the first premise in both rules). So we just have to focus on the execution of this subcommand (which is given through to the second premise). Finally, it may be possible that through jump, the next instruction is located in the other side of the sequence, and thus we have to execute over the whole sequence as a continuation (captured by the third premise). The reflexive closure is used to close the semantics tree branch where the command does not contain the current instruction. This is mainly used for two purposes: (1) in case of a sequence of sequence, and (2) as an implicit return at the end of the execution of a function. We illustrate the first case in Fig. 6.7. The second case may be simply understood by watching the `call` rule. The second premise corresponds to the execution of the body c of the callee fct. When this function finishes, the label of the current instruction is "outside" of c, thus the `refl_closure` rule closes the semantics tree branch, finalizing the execution with the ending state of the function execution.

```
Inductive cmd2: Set :=
| skip2: cmd2
| mutate2: insn → cmd2
| jmp2: Z → cmd2
| ifnjmp2: bexpr → Z → cmd2
| seq2: cmd2 → cmd2 → cmd2      (* Notation: c1; c2 *)
| call2: string → cmd2.

(** #A block is a couple of a command and its starting label# *)

Definition block2 := prod Z cmd2.

Fixpoint cmd2_size (c: cmd2): Z :=
  match c with
    | skip2 ⇒ 1
    | mutate2 i ⇒ 1
    | jmp2 l ⇒ 1
    | ifnjmp2 b l ⇒ 1
    | c1; c2 ⇒ (cmd2_size c1) + (cmd2_size c2)
    | call2 l ⇒ 1
  end.

Definition fenv2 := list (prod string block2).

Fixpoint get_cmd2_f  (f: fenv2) (s: string) struct f : option block2 :=
  match f with
    | nil ⇒ None
    | (hd1, hd2)::tl ⇒ if (string_dec hd1 s) then (Some hd2) else (get_cmd2_f tl s)
  end.
```

Figure 6.5: Informal target language definition.

6.1.2 Translation Stages

We previously explained that the translation is split into two stages: (1) an expansion of the control-flow (from loops and branches to jump), and (2) a linking of all functions bodies. In this section we present these transformations, which go respectively from the source language to the intermediate language, and from the intermediate language to the target language.

From the Source Language to the Intermediate Language

The first translator step goal is to expand the control-flow from loops and branches to jumps. This transformation is illustrated informally in Fig. 6.8. The corresponding function is named `translate12` in the Coq source, and its code is shown in Fig. 6.9. An important argument for this function is the starting address (the parameter `l`) of the translated block. This information is used to compute the addresses for the jumping instructions. The function `translate12` returns a couple composed of the translated command (the second projection) and the address next to its last instruction (the first projection). This address is equal to the starting address plus the size of the translated command, as stated by the lemma `translate12_end`. The `translate12` function is used to translate all the functions body inside the function environment. This is implemented in the function `translate_fenv1_to_fenv2`.

The main result of this first translation stage is the semantics-preservation lemma `preservation12`, presented in Fig. 6.10. This lemma asserts that if a source language program (consisting of a command `c` and a function environment `f`) executes from the state `st` to the state `st'`, then its translation will

$$\frac{}{f \vdash (l, st) == (l, \texttt{skip}) \Longrightarrow (l+1, st)}\ \texttt{skip}$$

$$\frac{st \mathrel{-\!\!-} i \mathrel{-\!\!>} st'}{f \vdash (l, st) == (l, i) \Longrightarrow (l+1, st')}\ \texttt{mutate}$$

$$\frac{st \mathrel{-\!\!-} i \mathrel{-\!\!>} \texttt{Abort}}{f \vdash (l, st) == (l, i) \Longrightarrow \texttt{Abort}}\ \texttt{mutate\_error}$$

$$\frac{}{f \vdash (l, st) == (l, \texttt{jmp}\ \ l') \Longrightarrow (l', st)}\ \texttt{jmp}$$

$$\frac{[\![b]\!]_{st} = true}{f \vdash (l, st) == (l, \texttt{ifnjmp}\ \ b\ l') \Longrightarrow (l+1, st)}\ \texttt{ifnjmp\_true}$$

$$\frac{[\![b]\!]_{st} = false}{f \vdash (l, st) == (l, \texttt{ifnjmp}\ \ b\ l') \Longrightarrow (l', st)}\ \texttt{ifnjmp\_false}$$

$$\frac{[\![fct]\!]_f = (lc, c) \qquad f \vdash (lc, st) == (lc, c) \Longrightarrow (l', st')}{f \vdash (l, st) == (l, \texttt{call}\ \ fct) \Longrightarrow (l+1, st')}\ \texttt{call}$$

$$\frac{[\![fct]\!]_f = (lc, c) \qquad f \vdash (lc, st) == (lc, c) \Longrightarrow \texttt{Abort}}{f \vdash (l, st) == (l, \texttt{call}\ \ fct) \Longrightarrow \texttt{Abort}}\ \texttt{call\_error'}$$

$$\frac{[\![fct]\!]_f = none}{f \vdash (l, st) == (l, \texttt{call}\ \ fct) \Longrightarrow \texttt{Abort}}\ \texttt{call\_error}$$

$$\frac{lc \leq l < (lc + \texttt{size\_of}(c_1)) \qquad f \vdash (l, st) == (l, c_1) \Longrightarrow (l', st')}{f \vdash (l, st) == (lc, c_1; c_2) \Longrightarrow (l', st')}\ \texttt{seq\_left}$$

$$\frac{(lc + \texttt{size\_of}(c_1)) \leq l < (lc + \texttt{size\_of}(c_1; c_2)) \qquad f \vdash (l, st) == (l, c_2) \Longrightarrow (l', st')}{f \vdash (l, st) == (lc, c_1; c_2) \Longrightarrow (l', st')}\ \texttt{seq\_right}$$

$$\frac{l < lc \vee (lc + \texttt{size\_of}(c_1; c_2)) <= l}{f \vdash (l, st) == (lc, c) \Longrightarrow (l, st)}\ \texttt{refl\_closure}$$

Figure 6.6: Informal intermediate language semantics.

behave in a similar way, or more precisely, it will have the same initial and final states.

From the Intermediate Language to the Target Language

The second stage of our translator takes a main command and a function environment of the intermediate language and translates them into a target language program. This translation is split into several steps.

The first step consists in changing the addresses of the main commands and of the functions bodies such that their labels do not overlap. Indeed, the semantics of the intermediate language allows these commands to share the same address space. This came from the inductive premise of the `call` rule in Fig. 6.6, where only the function body is considered. In comparison, the `call` rule of the target language semantics (Fig. 6.4) only changes the current instruction address. As target language programs are lists of instructions, and thus the addresses of the functions does not overlapped. In Coq we implement a function, named `cmd2_chg_labels`, presented in Fig. 6.12. This function takes as arguments an intermediate language command `c`, and a relative number `delta`, and shift the command labels by this relative offset. This transformation preserves the semantics, changing only the starting address of the command

$$\dfrac{\dfrac{\dfrac{}{f \vdash (0, st) == (0, \texttt{jmp 3}) \Longrightarrow (3, st)} \quad \dfrac{3 \geq 0 + 2}{f \vdash (3, st) == (0, (\texttt{jmp 3}; \texttt{skip})) \Longrightarrow (3, st)}}{\vdash f == (0, st) \Longrightarrow (0, (\texttt{jmp 3}; \texttt{skip}))(3, st)} \quad \dfrac{\dfrac{3 \geq 2 + 1}{f \vdash (3, st) == (2, \texttt{skip}) \Longrightarrow (3, st)}}{\cdots}}{f \vdash (0, st) == (0, (\texttt{jmp 3}; \texttt{skip}); \texttt{skip}) \Longrightarrow (3, st)}$$

Figure 6.7: Illustration of the reflexive closure for a sequence of sequence.

$$\left\llbracket \begin{array}{ll} \texttt{while} & b \\ & c \end{array} \right\rrbracket_l = \begin{array}{l} \texttt{ifnjmp } b\ (l + 1 + \texttt{size\_of}(\llbracket c \rrbracket) + 1); \\ \llbracket c \rrbracket_{l+1}; \\ \texttt{jmp } l \end{array}$$

$$\left\llbracket \begin{array}{lll} \texttt{if} & b & \texttt{then} \\ & c_1 & \\ \texttt{else} & & \\ & c_2 & \end{array} \right\rrbracket_l = \begin{array}{l} \texttt{ifnjmp } b\ (l + 1 + \texttt{size\_of}(\llbracket c_1 \rrbracket) + 1); \\ \llbracket c_1 \rrbracket_{l+1}; \\ \texttt{jmp } (l + 1 + \texttt{size\_of}(\llbracket c_1 \rrbracket) + 1 + \texttt{size\_of}(\llbracket c_2 \rrbracket + 1)); \\ \llbracket c_2 \rrbracket_{l+1+\texttt{size\_of}(\llbracket c_1 \rrbracket)+1}; \end{array}$$

$$\left\llbracket \begin{array}{l} c_1; \\ c_2 \end{array} \right\rrbracket_l = \begin{array}{l} \llbracket c_1 \rrbracket_l; \\ \llbracket c_2 \rrbracket_{l+\texttt{size\_of}(\llbracket c_1 \rrbracket)}; \end{array}$$

Figure 6.8: Control-flow expansion.

and the addresses for both the initial and the final states. This is captured by the lemma `cmd2_chg_labels_preserv_exec`. Thanks to this function we can "align" all the functions bodies that are stored in an environment in such a way that they do not overlap. This is the role of the function `fenv2_chg_labels`, which takes as argument a function environment `f`, and a label `l`. The function changes the functions bodies starting labels as follows: all functions will be shifted such that their starting address is the final label of the previous function plus one (this implies a "hole" between functions, which will be used in a next step), and the first function is shifted such that it begins at label `l`.

The next step consists in translating an intermediate language command into a target language command. The main issue of this step concerns the translation for the `call` instructions. Indeed, whereas it takes a function names in the intermediate language, it takes an address in the target language. The approach is similar as the one used in classical compilers: we build a map from names to addresses. The function `build_fun_name_map` takes a function environment and build its names map, while the function `get_fun_label3` allows to consult the map for a given function name. Such a map is used in the function `compile_block`, which transforms an intermediate language command into a target language command, to translate to `call` instructions.

In the final step, all the functions translated by the previous step are merged into a single target language command. The issue for this step consists in concatenates a return instruction (`ret`) at the end of each of this functions. Indeed, a main difference between the intermediate and the target language is the way how the function returns are managed. Whereas in the intermediate language the function return is implicit (is it implemented through the `refl_closure` rule), in the target language the function return is explicit (implemented by the `ret` instruction and its semantics rule). In the source language, the functions execute until the ends, which by the preservation lemma `preservation12`

```
Fixpoint translate12 (l: Z) (c: cmd1) struct c : Z * cmd2 :=
  match c with
    | skip1 ⇒ (l + 1, skip2)
    | mutate1 i ⇒ (l + 1, mutate2 i)
    | seq1 c1 c2 ⇒
      match (translate12 l c1) with
        | (l1, c1') ⇒
          match (translate12 l1 c2) with
            | (l2, c2') ⇒ (l2, c1'; c2')
          end
      end
    | ifte1 b c1 c2 ⇒
      match (translate12 (l + 1) c1) with
        | (l1, c1') ⇒
          match (translate12 (l1 + 1) c2) with
            | (l2, c2') ⇒
              (l2, ifnjmp2 b (l1 + 1); c1'; jmp2 l2; c2')
          end
      end
    | while1 b c1 ⇒
      match (translate12 (l + 1) c1) with
        | (l1, c1') ⇒
          (l1 + 1,  ifnjmp2 b (l1 + 1); c1'; jmp2 l)
      end
    | call1 s ⇒
      (l + 1, call2 s)
  end.

Lemma translate12_end: ∀ c l l' c',
  translate12 l c = (l', c') →
  l' = l + cmd2_size c'.

Fixpoint translate_fenv1_to_fenv2 (f: fenv1) struct f : fenv2 :=
  match f with
    | nil ⇒ nil
    | (s,c)::tl ⇒
      match (translate12 0 c) with
        | (l, hd') ⇒
          (s,(0, hd')) :: (translate_fenv1_to_fenv2 tl)
      end
  end.
```

Figure 6.9: The translation functions, and some lemmas in Coq.

means that their intermediate language translation will execute until the label next to the final instruction. Thus, by appending a `ret` instruction at the end of each command, we makes all the function to return after their executions. This append is implemented by the function `add_ret3`. This instruction will be placed in the hole that we have introduced in the first step. Finally, we append all the translated functions bodies as well as the main command into the final target command. The function `translate_fenv2_to_cmd3` translates and appends the functions bodies, while the function `translate23` implements the whole translation and appending. All the translation steps are illustrated in the Fig. 6.13.

The main result for the translation second stage is the semantics preservation lemma `preservation23`, presented in Fig. 6.14. This lemma asserts that the translated target language command behave in the same way as the original intermediate language command. More formally, both executions have the same initial and final labels and states. The main issue of this proof is that the lemma is not general enough to be directly proved by induction over the

```
Lemma preservation12: ∀ c f st st',
  cmd1_semop f (Some st) c (Some st') →
  ∀ l l' c',
    translate12 l c = (l', c') →
    ∀ f',
      translate_fenv1_to_fenv2 f = f' →
      cmd2_semop f' (Some (l,st)) (l, c') (Some (l',st')).
```

Figure 6.10: The semantics preservation lemma in Coq.

```
Fixpoint cmd2_chg_labels (delta: Z) (c: cmd2) struct c : cmd2 :=
  match c with
    | skip2 ⇒ skip2
    | mutate2 i ⇒ mutate2 i
    | jmp2 l ⇒ jmp2 (l + delta)
    | ifnjmp2 b l ⇒ ifnjmp2 b (l + delta)
    | seq2 c1 c2 ⇒ seq2 (cmd2_chg_labels delta c1) (cmd2_chg_labels delta c2)
    | call2 l ⇒ call2 l
  end.

Lemma cmd2_chg_labels_preserv_exec: ∀ f ST C ST',
  cmd2_semop f ST C ST' →
  ∀ l c,
    C = (l, c) →
    ∀ li st,
      ST = Some (li, st) →
      ∀ lf st',
        ST' = Some (lf, st') →
        ∀ delta,
          cmd2_semop f
            (Some (li + delta, st))
            (l + delta, cmd2_chg_labels delta c)
            (Some (lf + delta, st')).

Fixpoint fenv2_chg_labels (f: fenv2) (l: Z) struct f : fenv2 :=
  match f with
    | nil ⇒ nil
    | (s, (l', hd))::tl ⇒
      (s, (l, (cmd2_chg_labels (l - l') hd)))::(fenv2_chg_labels tl (l + cmd2_size hd + 1))
  end.
```

Figure 6.11: Coq function to modify the address space of an intermediate language command.

intermediate language semantics. To achieve the proof, we propose the more generalized lemma `preservation23'`. This generalization embedded the fact that the translation of the original command is a subcommand of the translated command. This generalization is due to the fact that the translation merges all the functions and the main command into a target language command. More technically, this generalization is needed for the subgoal that deals with the function `call` case.

Put it all together

We have presented the two steps of our translator. We now present how they are composed and how to derive the main semantics preservation lemma of the translator (shown in Fig. 6.15). The translator is implemented by the Coq function `translate13`, which is the composition of all the steps previously presented. The semantics preservation lemma `preservation13` asserts that the

```
Fixpoint build_fun_name_map (f: fenv2) : list (string * Z) :=
  match f with
    | nil ⇒ nil
    | (s,(l,c))::tl ⇒ (s,l)::(build_fun_name_map tl)
  end.

Fixpoint get_fun_label3 (s: string) (map: list (string * Z)) struct map : Z :=
  match map with
    | nil ⇒ 0 (* default *)
    | (s', l)::tl ⇒
      if (string_dec s s') then l else (get_fun_label3 s tl)
  end.

Fixpoint compile_block (c: cmd2) (map: list (string * Z)) struct c : cmd3 :=
  match c with
    | skip2 ⇒ skip3::nil
    | mutate2 i ⇒ (mutate3 i)::nil
    | jmp2 l ⇒ (jmp3 l)::nil
    | ifnjmp2 b l ⇒ (ifnjmp3 b l)::nil
    | seq2 c1 c2 ⇒ (compile_block c1 map) ++ (compile_block c2 map)
    | call2 s ⇒ (call3 (get_fun_label3 s map))::nil
  end.

Definition add_ret3 (c: cmd3) := c ++ ret3::nil.

Fixpoint translate_fenv2_to_cmd3 (f: fenv2) (map: list (string * Z)) struct f : cmd3 :=
  match f with
    | nil ⇒ nil
    | (s, (l, c))::tl ⇒
      (add_ret3 (compile_block c map)) ++ (translate_fenv2_to_cmd3 tl map)
  end.

Definition translate23 (f: fenv2) (mc: cmd2) : cmd3 :=
  let mp := (build_fun_name_map f) in (
    (add_ret3 (compile_block mc mp)) ++ (translate_fenv2_to_cmd3 f mp)
).
```

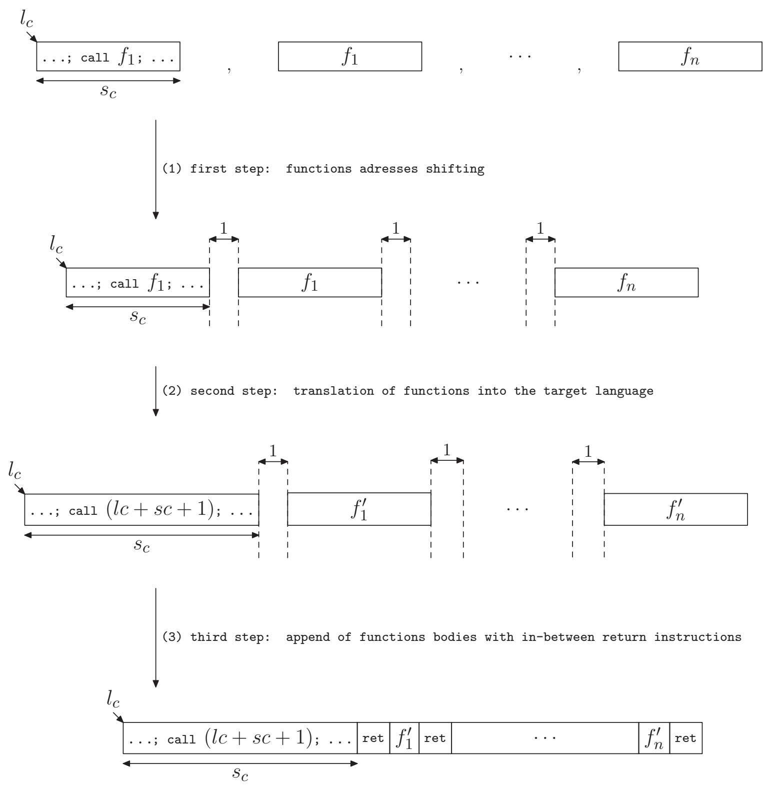
Figure 6.12: Coq function to modify the address space of an intermediate language command.

execution of the original command and of the translated command behave the same way. More precisely, the starting and ending states are the same in both executions, the target language command will execute entirely, and the initial and final stack are identical (which mean that all called function will return).

6.1.3 Hoare-triples Preservation

Until now, we have presented the languages that are manipulated by the translator (through their syntax and semantics), as well as the translator stages and their semantics preservation lemmas. Yet, our main concern still remains the composition of Hoare-logic triples. In this section we first present the Hoare-triples semantics for the source language, as well as a provably sound proof system. Then we present the Hoare-triples semantics for the intermediate language, and the preservation lemma for the first step of the translator. We then present the Hoare-triples semantics for the target language as well as the preservation lemma for the second step of the translator. Finally, by transitivity, we show the Hoare-triples preservation lemma from the source language to the target language.

As we previously stated, for sake of generality our translator is parameterized



l_c
...; call f_1; ...
s_c
f_1
...
f_n
(1) first step: functions adresses shifting
1
...; call f_1; ...
(2) second step: translation of functions into the target language
...; call $(lc + sc + 1)$; ...
f_1'
f_n'
(3) third step: append of functions bodies with in-between return instructions
ret

Figure 6.13: Illustration of the translation from the intermediate language to the target language.

by a set of state-modifying instructions (`insn`). For the Hoare-triples issue we add several parameters. The first one, `assert` defines the assertions as the Coq functions of type: `state` → `Prop`. Through this definition, we define the Hoare semantics for the state-modifying instruction language as:

```
Definition insn_tot_triple_sem (P: assert) (i: insn) (Q: assert) :=
  ∀ (st: state),
    (P st ->
     ∃ st',
       insn_semop st i (Some st') ∧ Q st'
    ).
```

We finally add as parameters `insn_tot_semax`, a Hoare-logic proof system which type is `assert` → `insn` → `assert` → `Prop`, and its soundness proof w.r.t. `insn_tot_triple_sem`:

```
Variable insn_tot_semax_sound: ∀ P i Q,
  insn_tot_semax P i Q ->
  insn_tot_triple_sem P i Q.
```

Hoare-triples for the Target Language

We define the semantics for Hoare-triples of the source language as:

```
Lemma preservation23': ∀ f ST ST' C,
  cmd2_semop f ST C ST' →
    ∀ l c,
    C = (l, c) →
      ∀ ls st,
        ST = Some (ls, st) →
        ∀ lf st',
          ST' = Some (lf, st') →
          ∀ lc mc f',
            fenv2_chg_labels f (lc + cmd2_size mc + 1) = f' →
            ∀ c',
              translate23 f' mc = c' →
              ∀ delta,
                subcmd3 c' lc (l + delta) (cmd2_size c) =
                Some (
                  compile_block (cmd2_chg_labels delta c) (build_fun_name_map f')
                ) →
                ∀ stk,
                  cmd3_closure
                    (Some (ls + delta, st, stk))
                    (lc, c')
                    (Some (lf + delta, st', stk)).


Lemma preservation23: ∀ f l st l' st' lc mc,
  cmd2_semop f (Some (l, st)) (lc, mc) (Some (l', st')) →
  ∀ f',
    fenv2_chg_labels f (lc + cmd2_size mc + 1) = f' →
    ∀ c',
      translate23 f' mc = c' →
      ∀ stk,
        cmd3_closure (Some (l, st, stk)) (lc, c') (Some (l', st', stk)).
```

Figure 6.14: Preservation lemmas of the translation from the intermediate language to the target language.

```
Definition triple1_sem (f: fenv1) (P: assert) (c: cmd1) (Q: assert)  :=
  ∀ (st: state),
  (P st →
   ∃ st',
     cmd1_semop f (Some st) c (Some st') ∧
     Q st'
  ).
```

This kind of semantics is known as total-correctness (in comparison to partial correctness presented in both lemmas of 2.1.3), which is characterized by the assertion of the existence of a final state `st'`. More informally, it states that: if the assertion `P` holds for the initial state `st` of the execution of the command `c`, then there exists a final state `st'` for which the assertion `Q` holds.

A provably sound proof system w.r.t this semantics is illustrated in Fig. 6.16. The main design difference with the proof system presented in Fig. 2.4 appears in the rule for the loop. In the present case, the premise that asserts that `I` is an invariant also include the fact that there exists a well-founded relation `P` over the sequence of initial and final states of the execution of the loop body. If such relation exists, then it implies that the loop always finishes. This new rule is the main issue of the proof of soundness. For this subgoal we need a nested well-formed induction over the natural number that is related by P to the initial state (`n`).

```
Definition translate13 (l: Z) (f: fenv1) (c: cmd1) : (Z * cmd3) :=
  match (translate12 l c) with
    | (l', c') ⇒
      (l', translate23 (fenv2_chg_labels (translate_fenv1_to_fenv2 f) (l' + 1)) c')
  end.

Lemma preservation13: ∀ f st c st',
  cmd1_semop f (Some st) c (Some st') →
  ∀ l l' c',
    translate13 l f c = (l', c') →
    ∀ stk,
      cmd3_closure (Some (l, st, stk)) (l, c') (Some (l', st', stk)).
```

Figure 6.15: The translator code and semantics preservation lemma.

Translator First Stage Preservation Lemma

The Hoare-triples semantics for the intermediate language is stated as:

```
Definition triple2_sem (f: fenv2) (P: assert) (ls: Z) (lc: Z) (c: cmd2) (lf: Z) (Q: assert)  :=
  ∀ (st: state),
  (P st →
   ∃ st',
     cmd2_semop f (Some (ls,st)) (lc, c) (Some (lf, st')) ∧ Q st'
  ).
```

Informally this definition states that if the precondition `P` holds for the initial state `st` of the execution of the block `(lc, c)` starting at the address `ls`, then there exists some final state `st'` after the execution until address `lf`, such that the post-condition `Q` holds. The preservation lemma for the translation between the source and intermediate state is:

```
Lemma triple_preservation12: ∀ c lc c' lc',
  translate12 lc c = (lc', c') →
  ∀ f f',
    translate_fenv1_to_fenv2 f = f' →
    ∀ P Q,
      triple1_sem f P c Q →
      triple2_sem f' P lc lc c' (lc + cmd2_size c') Q.
```

This lemma asserts that a source language and its translation in the intermediate language have the same precondition `P` and post-condition `Q`. The starting address (the first instruction of the command), and the final address (the next after the last instruction) of the execution are logically deduced from the operational semantics preservation lemma `preservation12` (Fig. 6.10).

Translator Second Stage Preservation Lemma

The Hoare-triples semantics for the target language is stated as:

```
Definition triple3_sem (P: assert) (ls: Z) (lc: Z) (c: cmd3) (Q: assert) (lf: Z) :=
  ∀ (st: state),
  (P st →
   ∃ st',
     ∀ stk,
       cmd3_closure (Some (ls, st, stk)) (lc, c) (Some (lf, st', stk)) ∧
       Q st'
  ).
```

This definition states that if a target language program executes from the address `ls`, in a state `st` for which the precondition `P` holds, then it will end

```
Inductive cmd1_tot_semax: fenv1 → assert → cmd1 → assert → Prop :=
| skip1_tot_semax: ∀ f P,
  cmd1_tot_semax f P skip1 P
| pre_str_tot_semax: ∀ f P P' c Q ,
  entail P P' →
  cmd1_tot_semax f P' c Q →
  cmd1_tot_semax f P c Q
| post_weak_tot_semax: ∀ f P c Q Q',
  entail Q' Q →
  cmd1_tot_semax f P c Q' →
  cmd1_tot_semax f P c Q
| mutate1_tot_semax: ∀ f P Q i,
  insn_tot_semax P i Q →
  cmd1_tot_semax f P (mutate1 i) Q
| seq1_tot_semax: ∀ f P Q R c1 c2,
  cmd1_tot_semax f P c1 R →
  cmd1_tot_semax f R c2 Q →
  cmd1_tot_semax f P (seq1 c1 c2) Q
| ifte1_tot_semax: ∀ f P Q b c1 c2,
  cmd1_tot_semax f (fun st ⇒ P st ∧ beval b st) c1 Q →
  cmd1_tot_semax f (fun st ⇒ P st ∧ ~ beval b st) c2 Q →
  cmd1_tot_semax f P (ifte1 b c1 c2) Q
| while1_tot_semax: ∀ f I b c (P: state → nat),
  (∀ n, cmd1_tot_semax f (fun st ⇒ I  st ∧ P st = n) c (fun st ⇒ I st ∧ (P st < n)%nat)) →
  cmd1_tot_semax f I (while1 b c) (fun st ⇒ I st ∧ ~ beval b st)
| call1_tot_semax: ∀ f P Q s c,
  get_cmd1_f f s = Some c →
  cmd1_tot_semax f P c Q →
  cmd1_tot_semax f P (call1 s) Q.

Lemma cmd11_tot_semax_sound: ∀ f P c Q,
  cmd1_tot_semax f P c Q →
  triple1_sem f P c Q.
```

Figure 6.16: Hoare-logic proof system for the target language and its soundness lemma.

at the address `lf` in some state `st'` for which the post-condition `Q` will hold. The Hoare-triples preservation lemma between the intermediate and the target language is defined as:

```
Lemma preservation_triple23: ∀ f P Q mc lc,
  triple2_sem f P lc lc mc (lc + cmd2_size mc) Q →
  ∀ f',
    fenv2_chg_labels f (lc + cmd2_size mc + 1) = f' →
    ∀ c',
      translate23 (fenv2_chg_labels f (lc + cmd2_size mc + 1)) mc = c' →
      triple3_sem P lc lc c' Q (lc + cmd2_size mc).
```

This lemma asserts that an intermediate language and its translation into the target language have the same pre/post-conditions. It also states that it will execute the whole translated block `(lc, c)`.

Put it all Together

Through the two previously shown lemmas, we can derived by transitivity of the implication an important result for the translator: the Hoare-triples preservation lemma.

```
Lemma preservation_triple13: ∀ f P Q c ,
  triple1_sem f P c Q →
  ∀ l l' c',
    translate13 l f c = (l', c') →
    triple3_sem P l l c' Q l'.
```

This lemma states that a source language program and its translation into the target language shares the same pre/post-conditions.

6.1.4 Hoare-triples Composition

The triples composition can be used in two different cases. The first one is when a C-like language program call an assembly routine. This case is often met in operating system kernels. Indeed, the parts of the source code that manipulate the hardware (such as context switching or memory paging) are written directly in assembly. These snippets of code can be called in the main loop of the operating system, which is written in C. In this case, the main issue is how to build a Hoare-triple for such a main loop which call assembly subroutines. The second case is met when a program uses a library written in a C-like language. For sake of reusability, one would like to use such library functions inside assembly code, instead of rewriting a corresponding library directly in assembly. In this case, the issue is how to build a specification for the assembly code that call C-like subroutines. Our approach is to lift the code to the same level: the assembly language. Thanks to our translator preservation lemmas, we know that the translation preserves the pre/post-conditions. Thus, the composition of triples is reduced to the composition of assembly triples by the sequence rule. The only technical issue is the manipulation of the function environment. Indeed, our approach considers that the subroutines are called, and thus their bodies are stored inside the function environment. In the rest of the section we explain in more technical details the lemma for triples composition for both cases.

C-like Program Calling Assembly Subroutines

In this case, we consider a snippet of the source language which corresponds to a sequence, of three commands: `c1; call s; c2`. In the case where the body of the function `s` is written in the source language, we can used the proof system introduced in Sect. 6.1.3 to prove the whole command Hoare-triple. However, in our case the body of the function `s` is a command of the intermediate language, for which this Hoare-triple proof system cannot be used anymore (the main problem is that we cannot apply the `call1_tot_semax`). For the verification of such template of source code we provide the following lemma:

```
Lemma heterogeneous_proof1:
  forall f P1 Q1 c1 P2 Q2 c2 f' P Q c lc s  c' l l'
    (c1_triple: triple1_sem f P1 c1 Q1)
    (c2_triple: triple1_sem f P2 c2 Q2)
    (f_trans:  translate_fenv1_to_fenv2 f = f')
    (no_s_in_f: forall c, ~ In (s, c) f)
    (c_triple: triple2_sem ((s, (lc, c))::f') P lc lc c (lc + cmd2_size c) Q)
    (Assert_H1: forall st, Q1 st -> P st)
    (Assert_H2: forall st, Q st -> P2 st)
    (prog_trans: translate12 l (c1; call1 s; c2) = (l', c'))
    (f_no_call: forall (s' : string) (c0 : cmd1),
          In (s', c0) f -> cmd1_no_call s c0)
    (c1'_no_call: cmd1_no_call s c1)
    (c2'_no_call: cmd1_no_call s c2),
      triple2_sem
           ((s, (lc, c))::f')
           P1 l
           l c'
           (l + cmd2_size c') Q2.
```

Lets examine more closely this lemma. In a first time, we have proved some Hoare-triple for c1 (respectively c2), with for precondition the assertion P1 (resp. P2), and the assertion Q1 (resp. Q2) for post-condition, both using the function environment f. These facts are captured by the hypothesizes c1_triple and c2_triple. The hypothesis f_trans captures that f' is the translation of the function environment f into the intermediate language. Next, we have proved the Hoare-triple for the intermediate language subroutine, namely c. This is captured by the hypothesis c_triple, which asserts that if the assertion P holds at the first instruction c, then after the execution of the last instruction, the assertion Q will holds. This triple is valid with for function environment f' to which we append the function s. This allows our subroutine to call itself, as well as a target language function. To be sure that this append will not interfere with the original function environment, the hypothesis no_s_is_f asserts that the name s is fresh in f (and thus by extension fresh in f'). We must also ensure that there is no functions in the environment f that call the function s. This property is captured by the hypothesis f_no_call, which uses the inductive predicate cmd1_no_call (for which a decidability lemma as been proved, meaning that Coq can build a sound and complete decision procedure). The goal of the lemma asserts that the translation of the whole command into the intermediate language (hypothesis prog_trans), have P1 for precondition and Q2 for post-condition, with for function environment the appending of the body c of function s. To achieve this goal, we need the fact that we can apply the precondition strengthening between (1) the c1 post-condition and the c precondition (hypothesis Assert_H1), and (2) the c post-condition and the c2 precondition (hypothesis Assert_H2).

Assembly Program Calling C-like Subroutines

In this case, an intermediate language command calls a target language. Similarly to the previous case, we decompose this command into three parts: a first intermediate command c1, followed by a call instruction to the target language subroutine (named c), and ending by an intermediate command c2. We use the same principle as previously: we translate the target language subroutine into an intermediate command, and we add it to the function environment (so that the call will execute it). This is captured by the following lemma:

```
Lemma heterogeneous_proof2:
  forall f P1 lc c1 Q1 P2 c2 Q2 P c Q s l l' c'
    (c1_triple: triple2_sem f P1 lc lc c1 (lc + cmd2_size c1) Q1)
    (c2_triple: triple2_sem f P2 (lc + cmd2_size c1 + 1)
         (lc + cmd2_size c1 + 1) c2 (lc + cmd2_size c1 + 1 + cmd2_size c2) Q2)
    (c_triple: triple1_sem nil P c Q)
    (no_s_in_f: forall c, ~ In (s, c) f)
    (Assert_H1: forall st, Q1 st -> P st)
    (Assert_H2: forall st, Q st -> P2 st)
    (prog_trans: translate12 l c = (l', c'))
    (f_no_call: forall (s' : string) (l0: Z) (c0 : cmd2),
         In (s', (l0, c0)) f -> cmd2_no_call s c0)
    (c1_no_call: cmd2_no_call s c1)
    (c2_no_call: cmd2_no_call s c2)
    (c_no_call: cmd1_no_call s c),
      triple2_sem
           ((s, (l, c'))::f)
           P1 lc
           lc (c1; call2 s; c2)
           (lc + cmd2_size c1 + 1 + cmd2_size c2) Q2.
```

6.2 Discussion

We have presented a generalize translator from a C-like language to an assembly language. This work is an extension of [2] with function calls. A main difference with this later work is that we only focus on the semantics preservation of the translation, whereas Saabas also provide proof for reflection. Such an extension should be an interesting future work.

In further implementations, we experimented other design choices and extensions. Yet these implementations are less advanced in features (for instance no triples composition lemmas). In a first variant we tried to avoid the sequence command in the intermediate language by using a list of instructions, similarly to the target language. This choice was motivated by the consideration of optimization steps for the intermediate language. Indeed any optimization implies the parsing of the language. It is well-known that parsing of tree-structured commands (i.e. with a binary sequence operator), is more tedious than a list traversal. The main issue for this design is located in the definition of the semantics for the sequence. Indeed in the case of a list of instructions, the left rule focuses on the first instruction of the command (i.e. the head of the list), and the right rule focuses on the remaining list (i.e. the tail of the list). Unfortunately, The lemmas that generalize the rules to version more similar to the original ones (considering not only the head and the tail, but any partition of the list), are difficult to prove by induction.

Another extension was to consider a particular instance of instructions for the target language. This instructions set includes a numerical and a boolean expressions. We intended to provide their translation into another instruction language without expressions. An illustrating example for such a translation is:

$$\llbracket\ x \leftarrow (y + (3 * z))\ \rrbracket_l \quad = \quad \begin{array}{l} x1 \leftarrow \texttt{mul}\ 3\ z; \\ x \leftarrow \texttt{add}\ y\ x1 \end{array}$$

Our approach was to translate every operator of the numerical and the boolean languages into instructions. We implemented this translator in Coq, and proved the preservation of the semantics. Although it is an interesting work (especially the generalization of lemmas for proof by induction), we found that the translation of the boolean expressions was not pertinent. Indeed, real compiler translate the boolean expression using the control flow of the underlying assembly, and not boolean instructions (like `and`, `or`). After this observation, we tried to implement the same mechanism using the intermediate language. For this purpose we started to inspect a way to extends the intermediate language with explicit labels (which map names to address in a command).

Another future work should be the implementation and certification of a registers allocation step, that replaces the source language variables, with registers for the target language.

6.3 Related Work

The verification of code translation is a popular trend in the community of proof assistants. Indeed, such tools allow to verify directly the implementation through clearly defined semantics.

The state of the art work is a compiler implemented and verified inside the Coq proof assistant [3]. It compiles a realistic subset of C into PPC assembly. The translation is split into several steps, and includes lightly optimizations. The only part of the compiler that is not proved correct is the register allocation.

Several previous work on certified compilation focus on Java-like languages. In [62], G.Klein and T.Nipkow propose Jinja, a Java-like programming language with formal semantics and type system. They propose also a virtual machine (JVM), and a compiler that is proved to preserve semantics and well-typeness. A similar work is presented in [63], with a compiler which is proved to preserve well-typeness.

Another previous work, from Okuma et al., [64] proposes a certified compiler from a scheme-like language into Jaba bytecode. This compiler implements an illustrating optimization: elimination of dead code. The main difference with our work is the source language. Indeed, whereas we focus on the compilation of a structured and imperative language, this related work implement the compilation of a functional language.

Chapter 7

Conclusion

In this Ph.D. thesis, we have investigated how one can verifies low-level software, such as operating system. For our experiments we have chosen the Topsy operating system as our test-bed [10]. We claim that Topsy was a pertinent choice, for several reasons:

- It directly controls its underlying hardware, and thus is obviously an instance of low-level software
- As an operating system, Topsy tries to build an abstraction of the underlying for hardware for the user applications. This adds another level of complexity.
- Even if Topsy is an embedded operating system, it implements several features also present in other operating systems: memory allocation, multi-programming, multi-threading, message passing, etc. This implies that our method and implementations should be reusable as a starting point for the verification of more complex operating system kernels, such as Linux.
- Finally, as Topsy started as a student project, it is simple and clear to read, making its analysis tractable.

Our approach is based on a top-down analysis of the system. Starting with its abstraction, we have focused on the verification of non-trivial properties, such as the task isolation, which informally can be stated as: the user threads cannot access the kernel memory. This abstraction, implemented as a model in SPIN, provides a more readable image of the system and allows to pinpoint the parts of the code that play a key role in the task isolation property. In order to verify that their correctness is a necessary condition for the task isolation, we injected bugs in their abstractions. Through this corrupted abstraction, SPIN has found execution traces for which task isolation does not hold any more.

Through several similar experiments we were able to find several parts of the code which correctness are a necessary condition for the task isolation. The interesting thing is that they all present different characteristics: some of them are non-trivial code in C, some are more straightforward, and some are written in assembly. To deal with the source code verification, we have implemented libraries inside the Coq proof assistant. Through these libraries, we have formally verified the correctness of some C and assembly source code. On top of these libraries, we have implemented and certified an original verification procedure for a decidable fragment of an extension of the Hoare-logic. We also have implemented and certified a translator from a C-like to an assembly language.

7.1 Summary of the Dissertation

In this section, we present a detail summary of the work presented in this Ph.D. thesis.

In Chap. 2.3, we presented a model for the Topsy operating system inside the SPIN model checker. This model abstracts the underlying hardware, the Topsy kernel as well as the network and thread servers, and an echo-server user application, which use all implemented operating system services. Through this model we verified several properties of the system:

- Status Correctness: the property that the kernel always restore the thread that has been elected by the scheduling algorithm
- Status Consistency: the property that a thread waiting for a message can never be scheduled
- Reply consistency: this property states that the receiver pointer for the message passing answer is not changed until the thread is unblocked and the expected message is sent
- Task Isolation: our main desired property, which states that the user threads of the system cannot access the kernel memory space

This model highlight the interactions between the system components and the underlying hardware. It helps to identify several parts of the code which correctness are necessary for this later property. We verify the importance of this system parts by injection of errors and verifying that the task isolation property does not hold anymore.

In Chap. 2, we presented the methods by which we verified the source code: (1) the separation logic, and (2) the Coq proof assistant. Separation logic is a extension of the Hoare-logic with a native notion of heap and pointers. We presented the semantics of the command language, the separation logic connectives that extends the Hoare-logic assertion language, and a provably sound and complete proof system. The Coq proof assistant is a tool which allows one to build mathematical models, and to mechanically construct and verify proof of their properties. We highlighted how Coq can be use to implement a certified arithmetic decision procedure by reflection.

In Chap. 3, we presented a library implementing the separation logic inside the Coq proof assistant. We detailed the implementation choices for the syntax and the semantics of the command language, and presented the provably sound and complete proof system for separation logic triples. Then we presented a use-case verification: the Topsy memory allocator (also known as the heap manager). This verification output is that we have found bugs, and through our verification we were able to patch the original source code.

In Chap. 4, we presented a variant of the previous library, that deals with verification of MIPS assembly source code. Using this implementation, we verified the source code for the context restoring function of Topsy.

In Chap. 5, we presented a decidable fragment of separation logic, as well as an original decision procedure. We implemented and proved the soundness of this procedure, on top of our separation logic library in Coq. The output is a tactics by reflection. Thanks to Coq extraction system, we provided a stand-alone, certified verifier in Ocaml.

In Chap. 6, we presented a translator from a C-like language to an assembly language, which has been proved to preserve the programs semantics. Through this translator, we have shown how one can compose specification of source code written in both C and assembly, to specify a statically linked program.

7.2 Contributions

7.3 Citations and Awards

A preliminary overview of this Ph.D. work was detailed in [43]. Our presentation was rewarded with the Takahashi Award from the Japanese Society for Software Science and Technology (JSSST) [48]. Our work about the automated verification of separation logic triples [45], presented in Chap. 5, was rewarded with the PPL2007 Best Paper Award.

Our work has been reused in a few research work: Andrew Appel extends our separation logic library in Coq with tactics for forward reasoning in [55], and Chunxiao Lin reuses our implementation ideas in Coq in [57]. Our work has also been cited in conference papers: the verification of the memory management in the L4 micro-kernel [58], and the design of a separation logic framework for the C minor programming language in Coq [59]. Finally, our separation logic library and the automated verification of separation logic triples was cited by Edmund Clarke in a survey of the separation logic [56].

Finally, we participated to an external work which aimed at formalize the verification of cryptographic protocols inside the Coq proof assistant [50].

Appendix A

Translator Languages Semantics in Coq

```
Inductive cmd1_semop:
fenv1 → option state →
cmd1 → option state → Prop :=

| err1_semop: ∀ c f,
cmd1_semop f None c None
| skip1_semop: ∀ f st,
cmd1_semop f (Some st) skip1 (Some st)
| mutate1_semop: ∀ st i st' f,
insn_semop st i st' →
cmd1_semop f (Some st) (mutate1 i) st'
| seq1_semop: ∀ st st' st'' c1 c2 f,
cmd1_semop f (Some st) c1 st' →
cmd1_semop f st' c2 st'' →
cmd1_semop f (Some st) (seq1 c1 c2) st''
| ifte1_true_semop: ∀ st b c1 c2 st' f,
beval b st  →
cmd1_semop f (Some st) c1 st' →
cmd1_semop f (Some st) (ifte1 b c1 c2) st'
| ifte1_false_semop: ∀ st b c1 c2 st' f,
~ beval b st →
cmd1_semop f (Some st) c2 st' →
cmd1_semop f (Some st) (ifte1 b c1 c2) st'
| while1_false_semop: ∀ st b c f,
~ beval b st →
cmd1_semop f (Some st) (while1 b c) (Some st)
| while1_true_semop: ∀ st b c st' st'' f,
beval b st →
cmd1_semop f (Some st) c st' →
cmd1_semop f st' (while1 b c) st'' →
cmd1_semop f (Some st) (while1 b c) st''
| call1_err_semop: ∀ st s f,
get_cmd1_f f s = None →
cmd1_semop f (Some st) (call1 s) None
| call1_semop: ∀ st s f st' c,
get_cmd1_f f s = Some c →
cmd1_semop f (Some st) c st' →
cmd1_semop f (Some st) (call1 s) st'.
```

Figure A.1: Coq source language semantics.

```
Inductive cmd3_semop:
option (Z * state * stack3) → (Z * cmd3) →
option (Z * state * stack3) → Prop :=
| err2_cmd3_semop: ∀ s l st lc c,
get_insn3 c lc l = None →
cmd3_semop (Some (l, st, s)) (lc, c) None
| err_cmd3_semop: ∀ c,
cmd3_semop None c None
| skip3_semop: ∀ l st lc c s,
get_insn3 c lc l = Some (skip3) →
cmd3_semop (Some (l, st ,s)) (lc, c) (Some (l + 1, st, s))
| mutate3_semop: ∀ l st st' i s c lc,
get_insn3 c lc l = Some (mutate3 i) →
insn_semop st i (Some st') →
cmd3_semop (Some (l, st ,s)) (lc, c) (Some (l + 1, st', s))
| mutate3_err_semop: ∀ l st i s c lc,
get_insn3 c lc l = Some (mutate3 i) →
insn_semop st i None →
cmd3_semop (Some (l, st, s)) (lc, c) None
| jmp3_semop: ∀ l st l' s lc c,
get_insn3 c lc l = Some (jmp3 l') →
cmd3_semop (Some (l, st, s)) (lc, c) (Some (l', st, s))
| ifnjmp3_true_semop: ∀ b l st l' s lc c,
get_insn3 c lc l = Some (ifnjmp3 b l') →
beval b st →
cmd3_semop (Some (l, st, s)) (lc, c) (Some (l + 1, st, s))
| ifnjmp3_false_semop: ∀ b l st l' s lc c,
get_insn3 c lc l = Some (ifnjmp3 b l') →
~ beval b st →
cmd3_semop (Some (l, st, s)) (lc, c) (Some (l', st, s))
| call3_semop: ∀ lc c l st l' s,
get_insn3 c lc l = Some (call3 l') →
cmd3_semop (Some (l, st, s)) (lc, c) (Some (l', st, (l + 1)::s))
| ret3_semop: ∀ lc c l st hd tl,
get_insn3 c lc l = Some (ret3) →
cmd3_semop (Some (l, st, hd::tl)) (lc, c) (Some (hd, st, tl)).

Inductive cmd3_closure:
option (Z * state * stack3) → (Z * cmd3) →
option (Z * state * stack3) → Prop :=
| None_closure: ∀ lc c,
cmd3_closure None (lc, c) None
| one_step_closure: ∀ lc c l st st' s st'',
cmd3_semop (Some (l, st, s)) (lc, c) st' →
cmd3_closure st' (lc, c) st'' →
cmd3_closure (Some (l, st, s)) (lc, c) st''
| refl_closure: ∀ l st lc c stk,
cmd3_closure (Some (l, st, stk)) (lc, c) (Some (l, st, stk)).
```

Figure A.2: Coq target language semantics.

```
Inductive cmd2_semop:
fenv2 → option (Z * state) →
block2 → option (Z * state) → Prop :=

| errl_cmd2_semop: ∀ l st lc c f,
lc + (cmd2_size c) <= l ∨ l < lc →
cmd2_semop f (Some (l, st)) (lc, c) (Some (l, st))
| errs_cmd2_semop: ∀ c f,
cmd2_semop f None c None
| skip2_semop: ∀ l st f,
cmd2_semop f (Some (l, st)) (l, skip2) (Some (l + 1, st))
| mutate2_cmd2: ∀ l st i st' f,
insn_semop st i (Some st') →
cmd2_semop f (Some (l, st)) (l, mutate2 i) (Some (l + 1, st'))
| mutate2_err_cmd2: ∀ l st i f,
insn_semop st i None →
cmd2_semop f (Some (l, st)) (l, mutate2 i) None
| jmp2_semop: ∀ l st l' f,
(*l <> l' → *)
cmd2_semop f (Some (l, st)) (l, jmp2 l') (Some (l', st))
| ifnjmp2_true_semop: ∀ l st b l' f,
beval b st →
cmd2_semop f (Some (l, st)) (l, ifnjmp2 b l') (Some (l + 1, st))
| ifnjmp2_false_semop: ∀ l st b l' f,
(*l <> l' → *)
~ beval b st  →
cmd2_semop f (Some (l, st)) (l, ifnjmp2 b l') (Some (l', st))
| seq2_left_semop: ∀ l st lc c1 c2 st' st'' f,
lc <= l < lc + cmd2_size c1  →
cmd2_semop f (Some (l, st)) (lc, c1) st' →
cmd2_semop f st' (lc, seq2 c1 c2) st'' →
cmd2_semop f (Some (l, st)) (lc, seq2 c1 c2) st''
| seq2_right_semop: ∀ l st lc c1 c2 st' st'' f,
lc + cmd2_size c1 <= l < lc + cmd2_size (seq2 c1 c2) →
cmd2_semop f (Some (l, st)) (cmd2_size c1 + lc, c2) st' →
cmd2_semop f st' (lc, seq2 c1 c2) st'' →
cmd2_semop f (Some (l, st)) (lc, seq2 c1 c2) st''
| call2_semop: ∀ st l l' f st' s  c',
get_cmd2_f f s = Some (l', c') →
cmd2_semop f (Some (l', st)) (l', c') (Some (l' + cmd2_size c', st')) →
cmd2_semop f (Some (l, st)) (l, call2 s) (Some (l + 1, st'))
| call2'_semop: ∀ st l l' f s c',
get_cmd2_f f s = Some (l', c') →
cmd2_semop f (Some (l', st)) (l', c') None →
cmd2_semop f (Some (l, st)) (l, call2 s) None
| call2_err_semop: ∀ st l f s,
get_cmd2_f f s = None →
cmd2_semop f (Some (l, st)) (l, call2 s) None.
```

Figure A.3: Coq intermediate language semantics.

Appendix B

Hoare-triples of Topsy Heap Manager Functions

	`Definition hmAlloc result size entry cptr fnd stts nptr sz :=`	$\{\ \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x)\ \}$
1	`result <- null;`	$\{\ \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \texttt{result=}0\ \}$
2	`findFree size entry fnd sz stts;`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \texttt{result=}0 \wedge \\ \left(\begin{array}{c} \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Free}) \in l \wedge \texttt{entry=}y \\ \vee \\ \texttt{entry=}0 \end{array} \right) \end{array} \right\}$
3	`ifte (entry == null) thendo (`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \texttt{result=}0 \wedge \\ \texttt{entry=}0 \end{array} \right\}$
4	`  cptr <- hmStart;`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \texttt{result=}0 \wedge \\ \texttt{entry=}0 \wedge \texttt{cptr=hmStart} \end{array} \right\}$
5	`  compact cptr nptr stts;`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \texttt{result=}0 \wedge \\ \texttt{entry=}0 \end{array} \right\}$
6	`  findFree size entry fnd sz stts;`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \texttt{result=}0 \wedge \\ \left(\begin{array}{c} \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Free}) \in l \wedge \texttt{entry=}y \\ \vee \\ \texttt{entry=}0 \end{array} \right) \end{array} \right\}$
7	`) elsedo (`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \texttt{result=}0 \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Free}) \in l \wedge \texttt{entry=}y \end{array} \right\}$
8	`  skip;`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \texttt{result=}0 \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Free}) \in l \wedge \texttt{entry=}y \end{array} \right\}$
9	`)`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \texttt{result=}0 \wedge \\ \left(\begin{array}{c} \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Free}) \in l \wedge \texttt{entry=}y \\ \vee \\ \texttt{entry=}0 \end{array} \right) \end{array} \right\}$
10	`ifte (entry == null) thendo (`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \texttt{result=}0 \wedge \texttt{entry=}0 \end{array} \right\}$
	`  (* HM_ALLOCFAILED is equal to 0 *)`	
11	`  result <- HM_ALLOCFAILED;`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \texttt{result=}0 \wedge \texttt{entry=}0 \end{array} \right\}$
12	`) elsedo (`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \texttt{result=}0 \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Free}) \in l \wedge \texttt{entry=}y \wedge x \neq y \end{array} \right\}$
13	`  split entry size cptr sz;`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \texttt{result=}0 \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Alloc}) \in l \wedge \texttt{entry=}y \wedge x \neq y \end{array} \right\}$
14	`  result <- entry + 2;`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Alloc}) \in l \wedge \\ \texttt{entry=}y \wedge \texttt{result=entry}{+}2 \wedge x \neq y \end{array} \right\}$
15	`).`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \left(\begin{array}{c} \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Alloc}) \in l \wedge \\ \texttt{entry=}y \wedge \texttt{result=entry}{+}2 \wedge x \neq y \\ \vee \\ \texttt{result=}0 \end{array} \right) \end{array} \right\}$

$\mathsf{Hp}(l) \overset{def}{=} \mathsf{Heap\text{-}list}\ l\ \texttt{hmStart}\ 0$

Figure B.1: Sketch of `hmAlloc` proof (the proofs for grayed instructions appear in Fig. B.2, Fig. B.3, and Fig. B.4)

`Definition findFree size entry fnd sz stts :=`	$\{\ \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x)\ \}$
`entry <- hmStart;`	$\{\ \ldots\ \}$
`stts <-* (entry -.> status);`	$\{\ \ldots\ \}$
`fnd <- 0;`	$\left\{\begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \texttt{entry=hmStart} \wedge \\ \exists status.(status\texttt{=Alloc} \vee status\texttt{=Free}) \wedge \texttt{stts=}status \wedge \\ \exists size'.(\texttt{hmStart}, size', status) \in l \wedge \texttt{fnd=}0 \end{array}\right\}$
`while ((entry =/= null) &&& (fnd =/= 1)) (`	$\boxed{\left\{\begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \\ \wedge \\ \left(\begin{array}{c} \left(\begin{array}{c} \exists bloc\_adr.\texttt{entry=}bloc\_adr \wedge bloc\_adr > 0 \wedge \\ \texttt{fnd=}1 \wedge \\ \exists size'.size' \geq \texttt{size} \wedge (bloc\_adr, size', \texttt{Free}) \in l \\ \vee \\ \texttt{fnd=}0 \wedge \\ \exists status.(status\texttt{=Alloc} \vee status\texttt{=Free}) \wedge \\ \texttt{stts=}status \wedge \exists size'.(bloc\_adr, size', \texttt{Free}) \in l \\ \vee \\ \texttt{fnd=}0 \wedge \\ (\mathsf{Heap\text{-}list}\ l\ \texttt{hmStart}\ bloc\_adr\ \texttt{**} \\ \mathsf{Heap\text{-}list}\ nil\ bloc\_adr\ 0) \end{array}\right) \\ \vee \\ \texttt{entry=}0 \end{array}\right) \end{array}\right\}}$
`  stts <-* (entry -.> status);`	$\{\ \ldots\ \}$
`  ENTRYSIZE entry sz;`	$\{\ \ldots\ \}$
`  ifte ((stts == Free) &&& (sz >>= size)) thendo`	$\{\ \ldots\ \}$
`    fnd <- 1`	$\left\{\begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \exists bloc\_adr.\texttt{entry=}bloc\_adr \wedge bloc\_adr > 0 \wedge \\ \texttt{fnd=}1 \wedge \\ \exists size'.size' \geq \texttt{size} \wedge (bloc\_adr, size', \texttt{Free}) \in l \end{array}\right\}$
`  elsedo`	$\{\ \ldots\ \}$
`    entry <-* (entry -.> next)`	$\{\ \ldots\ \}$
`).`	$\left\{\begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \left(\begin{array}{c} \exists y.\exists size_y.\ size_y \geq \texttt{size} \wedge \\ (y, size_y, \texttt{Free}) \in l\ \wedge\ \texttt{entry=}y \\ \vee \\ \texttt{entry} = 0 \end{array}\right) \end{array}\right\}$

$\mathsf{Hp}(l) \overset{def}{=} \mathsf{Heap\text{-}list}\ l\ \texttt{hmStart}\ 0$, only relevant assertions are displayed, the loop invariant is boxed

Figure B.2: Sketch of `findFree` proof (partial proof of `hmAlloc` in Fig. B.1)

	`Definition compact cptr nptr stts :=`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \texttt{cptr=hmStart} \land \\ \text{Init-array } (x{+}2)\ (list_x) \end{array} \right\}$
1	`(* cptr points to the current block *)` `while (cptr =/= null) (`	$\boxed{\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \\ \text{Init-array } (x{+}2)\ (list_x) \land \exists y.\texttt{cptr}{=}y \\ \land \\ \left(\begin{array}{c} \exists sz.\exists st.\exists l_1.\exists l_2. \\ l{=}(l_1 {+\!+} ((y, sz, st) :: nil) {+\!+} l_2) \\ \lor \\ (\text{Heap-list } l \texttt{ hmStart } y {*\!*} \text{Heap-list } nil\ y\ 0) \\ \lor \\ y = 0 \end{array} \right) \end{array} \right\}}$
2	`stts <-* (cptr -.> status);`	$\{ \ldots \}$
3	`ifte (stts == Free) thendo (`	$\{ \ldots \}$
4	`nptr <-* (cptr -.> next);`	$\{ \ldots \}$
5	`(* nptr points to the block next to cptr *)` `while (nptr =/= null) (`	$\boxed{\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \\ \text{Init-array } (x{+}2)\ (list_x) \land \exists y.\texttt{cptr}{=}y \\ \land \\ \left(\begin{array}{c} \exists sz.\exists l_1.\exists l_2. \\ l{=}(l_1 {+\!+} ((y, sz, \texttt{Free}) :: nil) {+\!+} l_2) \land \\ \texttt{nptr}{=}y{+}2{+}sz \\ \lor \\ \exists sz.\exists sz'.\exists l_1.\exists l_2. \\ l{=}(l_1 {+\!+} ((y, sz, \texttt{Free}) :: \\ (y{+}2{+}sz, sz', \texttt{Alloc}) :: nil) {+\!+} l_2) \land \\ \texttt{nptr}{=}y{+}2{+}sz \\ \lor \\ \exists sz.\exists l_1. \\ l{=}(l_1 {+\!+} ((y, sz, \texttt{Free}) :: nil)) \land \\ \texttt{nptr}{=}0 \end{array} \right) \end{array} \right\}}$
6	`stts <-* (nptr -.> status);`	$\{ \ldots \}$
7	`ifte (stts == Free) thendo (`	$\{ \ldots \}$
8	`stts <-* (nptr -.> next);`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \\ \text{Init-array } (x{+}2)\ (list_x) \land \exists y.\texttt{cptr}{=}y \\ \land \\ \exists sz.\exists sz'.\exists l_1.\exists l_2. \\ l{=}(l_1 {+\!+} ((y, sz, \texttt{Free}) :: (y{+}2{+}sz, sz', \texttt{Free}) :: nil) {+\!+} l_2) \land \\ \texttt{nptr}{=}y{+}2{+}sz \land \texttt{stts}{=}y{+}4{+}sz{+}sz' \end{array} \right\}$
	COMPACTION	
9	`(cptr -.> next) *<- stts;`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \\ \text{Init-array } (x{+}2)\ (list_x) \land \exists y.\texttt{cptr}{=}y \\ \land \\ \exists sz.\exists sz'.\exists l_1.\exists l_2. \\ l{=}(l_1 {+\!+} ((y, sz + 2 + sz', \texttt{Free}) :: nil) {+\!+} l_2) \land \\ \texttt{nptr}{=}y{+}2{+}sz \land \texttt{stts}{=}y{+}4{+}sz{+}sz' \end{array} \right\}$
10	`nptr <- stts`	$\{ \ldots \}$
11	`) elsedo (`	$\{ \ldots \}$
12	`nptr <- null`	$\{ \ldots \}$
13	`)`	$\{ \ldots \}$
14	`)`	$\{ \ldots \}$
15	`) elsedo (`	$\{ \ldots \}$
16	`skip`	$\{ \ldots \}$
17	`)`	$\{ \ldots \}$
18	`cptr <-* (cptr -.> next)`	$\{ \ldots \}$
19	`).`	$\{ \exists l.\mathsf{Hp}(l) \land (x, size, \texttt{Alloc}) \in l \land \text{Init-array } (x{+}2)\ (list_x) \}$

$\mathsf{Hp}(l) \overset{def}{=}$ Heap-list l `hmStart` 0, only relevant assertions are displayed, loop invariants are boxed, the grayed area corresponds to a heap-list lemma application

Figure B.3: Sketch of `compact` proof (partial proof of `hmAlloc` in Fig. B.1)

	`Definition split entry size cptr sz :=`	$\left\{ \begin{array}{c} \exists l.\ \mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size}\ \wedge\ (y, size_y, \texttt{Free}) \in l \wedge \\ \texttt{entry}{=}y \wedge x \neq y \end{array} \right\}$
1	`ENTRYSIZE entry sz;`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size}\ \wedge\ (y, size_y, \texttt{Free}) \in l \wedge \\ \texttt{entry}{=}y \wedge x \neq y \wedge \texttt{sz}{=}size_y \end{array} \right\}$
2	`ifte (sz >>= (size + LEFTOVER + 2) thendo (`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Free}) \in l \wedge \\ \texttt{entry}{=}y \wedge x \neq y \wedge \texttt{sz}{=}size_y \end{array} \right\}$
3	`  cptr <- (entry + 2 + size);`	$\{\ \ldots\ \}$
4	`  sz <-* (entry -.> next);`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Free}) \in l \wedge \\ \texttt{cptr}{=}\texttt{entry}{+}2{+}\texttt{size} \wedge \texttt{sz}{=}y{+}2{+}size_y \wedge \\ \texttt{entry} = y \wedge x \neq y \end{array} \right\}$
	Splitting	
5	`  (cptr -.> next) *<- sz;`	$\{\ \ldots\ \}$
6	`  (cptr -.> status) *<- Free;`	$\{\ \ldots\ \}$
7	`  (entry -.> next) *<- cptr`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Free}) \in l \wedge \\ \texttt{entry}{=}y \wedge x \neq y \end{array} \right\}$
8	`) elsedo (`	$\{\ \ldots\ \}$
9	`  skip`	$\{\ \ldots\ \}$
10	`);`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Free}) \in l \wedge \\ \texttt{entry}{=}y \wedge x \neq y \end{array} \right\}$
	Change-status	
11	`(entry -.> status) *<- Allocated.`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Alloc}) \in l \wedge \\ \texttt{entry}{=}y \wedge x \neq y \end{array} \right\}$

$\mathsf{Hp}(l) \overset{def}{=} \mathsf{Heap\text{-}list}\ l\ \texttt{hmStart}\ 0$, only relevant assertions are displayed, grayed areas correspond to a heap-list lemma application

Figure B.4: Sketch of `split` proof (partial proof of `hmAlloc` in Fig. B.1)

Bibliography

[1] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, p. 55–74.

[2] Ando Saabas and Tarmo Uustalu. A compositional natural semantics and Hoare logic for low-level languages. In *Theor. Comput. Sci.*, v. 373, n. 3, pp. 273-302, 2007.

[3] Xavier Leroy Programming a Compiler with a Proof Assistant. In *33rd symposium Principles of Programming Language*, ACM PRESS, pp. 42-54, 2006.

[4] Gregory Duval and Jacques Julliand. Modeling and Verification of the RUBIS μ-kernel with SPIN. In *1st SPIN Workshop (SPIN 1995).*

[5] Patrick Tullmann, Jeff Turner, John McCorquodale, Jay Lepreau, Ajay Chitturi, and Godmar Back. Formal Methods: A Practical Tool for OS Implementors. In *6th Workshop on Hot Topics in Operating Systems (HotOS-VI 1997).*

[6] Endrawaty. Verification of the Fiasco IPC Implementation. Master Thesis. Desden University of Technology. 2005.

[7] William R. Bevier. A Verified Operating System Kernel. Ph. D. Thesis. University of Texas at Austin. 1987.

[8] Gerard J. Holzmann. The Spin Model Checker. Addison Wesley. 2003.

[9] Lukas Ruf and various contributors. TOPSY – A Teachable Operating System. `http://www.topsy.net/`.

[10] Lucas Ruf, Claudio Jeker, Boris Lutz, and Bernhard Plattner. Topsy v3: A NodeOS For Network Processors. In *2nd International Workshop on Active Network Technologies and Applications (ANTA 2003).*

[11] Reynald Affeldt and Nicolas Marti. Spin model of Topsy available at: `http://staff.aist.go.jp/reynald.affeldt/seplog/`.

[12] Various contributors. The Coq Proof assistant. `http://coq.inria.fr`.

[13] Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Towards Formal Verification of Memory Properties using Separation Logic. In *22nd Workshop of the Japan Society for Software Science and Technology (JSSST 2005).*

[14] Reynald Affeldt and Nicolas Marti. Towards Formal Verification of Memory Properties using Separation Logic. `http://savannah.nongnu.org/projects/seplog`. Online CVS.

[15] Dachuan Yu, Nadeem Abdul Hamid, and Zhong Shao. Building Certified Libraries for PCC: Dynamic Storage Allocation. *Science of Computer Programming*, 50(1-3):101–127. Elsevier, Mar. 2004.

[16] Tjark Weber. Towards Mechanized Program Verification with Separation Logic. In *13th Conference on Computer Science Logic (CSL 2004)*, volume 3210 of *LNCS*, p. 250–264. Springer.

[17] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A Decidable Fragment of Separation Logic. In *24th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2004)*, volume 3328 of *LNCS*, p. 97–109. Springer.

[18] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic Execution with Separation Logic. In *3rd Asian Symposium on Programming Languages and Systems (APLAS 2005)*, volume 3780 of *LNCS*, p. 52–68. Springer.

[19] Norbert Schirmer. A Verification Environment for Sequential Imperative Programs in Isabelle/HOL. In *11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2004)*, volume 3452 of *LNCS*, p. 398–414. Springer.

[20] Farhad Mehta and Tobias Nipkow. Proving Pointer Programs in Higher-Order Logic. In *Information and Computation*, 199:200–227. Elsevier, 2005.

[21] Harvey Tuch and Gerwin Klein. A Unified Memory Model for Pointers. In *12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005)*, volume 3835 of *LNCS*, p. 474–488. Springer.

[22] Jean-Christophe Filliâtre. Multi-Prover Verification of C Programs. In *6th International Conference on Formal Engineering Methods (ICFEM 2004).* volume 3308 of *LNCS*, p. 15–29. Springer.

[23] Thierry Hubert and Claude Marché. A case study of C source code verification: the Schorr-Waite algorithm. In *3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005).*

[24] Thomas Melham. A mechanized theory of the pi-calculus in HOL. Nordic journal of computing, 1(1):50-76, 1995.

[25] Samuel Boutin. Using Reflection to Build Efficient and Certified Decision Procedures. In *3rd Int. Symp. on Theoretical Aspects of Computer Software (TACS 1997)*, LNCS vol. 1281, p. 515–529, Springer.

[26] Hongseok Yang. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In *1st Work. on Semantics, Program Analysis, and Computing Environments For Memory Management (SPACE 2001).*

[27] Cristiano Calcagno, Hongseok Yang and Peter O'Hearn. Computability and Complexity Results for a Spatial Assertion Language for Data Structures. In *21st Int. Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2001)*, LNCS vol. 2001, p. 108–119, Springer.

[28] Didier Galmiche and Daniel Méry. Characterizing Provability in BI's Pointer Logic through Resource Graphs. In *12th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005)*, LNCS vol. 3835, p. 459–473, Springer.

[29] Cristiano Calcagno, Philippa Gardner and Matthew Hague. From Separation Logic to First-Order Logic. In *8th Int. Conf. on Foundations of Software Science and Computational Structures (FOSSACS 2005)*, LNCS vol. 3441, p. 395–409, Springer.

[30] Amine Chaieb and Tobias Nipkow. Verifying and Reflecting Quantifier Elimination for Presburger Arithmetic. In *12th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005)*, LNCS vol. 3835, p. 367–380, Springer.

[31] Cristiano Calcagno, Dino Distefano, Peter O'Hearn and Hongseok Yang. Beyond Reachability: Shape Abstraction in the Presence of Pointer Arithmetic. In *13th Int. Symp. on Static Analysis (SAS 2006)*, LNCS vol. 4134, p. 182–203, Springer.

[32] Nicolas Marti, Reynald Affeldt and Akinori Yonezawa. Formal Verification of the Heap Manager of an Operating System using Separation Logic. In *8th Int. Conf. on Formal Engineering Methods (ICFEM 2006)*, LNCS vol. 4260, p. 400–419, Springer.

[33] Huu Hai Nguyen, Christina David, Shengchao Qin and Wei-Ngan Chin. Automated Verification of Shape and Size Properties via Separation Logic. In *8th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI 2007)*, LNCS vol. 4349, Springer.

[34] John Harrison. Cooper's Algorithm for Presburger Arithmetic. `http://www.cl.cam.ac.uk/~jrh13/atp`.

[35] Nadeem Abdul Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A Syntactic Approach to Foundational Proof-Carrying Code. In *7th IEEE Symposium on Logic In Computer Science (LICS 2002)*, p. 89–100.

[36] Dachuan Yu and Nadeem Abdul Hamid and Zhong Shao. Building Certified Libraries for PCC: Dynamic Storage Allocation. In *12th European Symposium on Programming (ESOP 2003)*, volume 2618 of *LNCS*, p. 363–379. Springer.

[37] Nadeem Abdul Hamid and Zhong Shao. Interfacing Hoare Logic and Type Systems for Foundational Proof-Carrying Code. In *17th Conference on Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *LNCS*, p. 118–135. Springer.

[38] Domagoj Babić and Madanlal Musuvathi. Modular Arithmetic Decision Procedure. Microsoft Research Technical Report. MSR-TR-2005-114.

[39] Adam J. Chlipala. Modular development of certified program verifiers with a proof assistant. In *11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006)*, p 160–171.

[40] MIPS Technologies. MIPS32 4KS Processor Core Family Software User's Manual MIPS Technologies, Inc., 1225 Charleston Road, Mountain View, CA 94043-1353.

[41] Zhaozhong Ni and Dachuan Yu and Zhong Shao. Using XCAP to Certify Realistic Systems Code: Machine Context Management. In *20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'07)*, Kaisrslautern, Germany, September 2007. Lecture Notes in Computer Science Vol. 4732, p. 189–206. Springer.

[42] Reynald Affeldt and Nicolas Marti An Approach to Formal Verification of Arithmetic Functions in Assembly. In *11th Annual Asian Computing Science Conference (ASIAN 2006), Focusing on Secure Software and Related Issues*, Dec 2006. To appear in Lecture Notes in Computer Science, Springer.

[43] Nicolas Marti and Reynald Affeldt Towards Formal Verification of Memory Properties Using Separation Logic. In *22nd Meeting of the Japan Society for Software Science and Technology, Tohoku University, Sendai, Japan, September 13–15, 2005.* Japan Society for Software Science and Technology, Sep. 2005. Electronic proceeding (ISSN 1348-0901). Not referred. 6 pages.

[44] Nicolas Marti and Reynald Affeldt Model-checking of a Multi-threaded Operating System. In *23rd Workshop of the Japan Society for Software Science and Technology, University of Tokyo, Tokyo, Japan, September 13–15, 2006.*

[45] Nicolas Marti and Reynald Affeldt A Certified Verifier for a Fragment of Separation Logic. In *In 9th JSSST Workshop on Programming and Programming Languages (PPL 2007)*, Mars, 2007.

[46] Ming-Yuan Zhu, Lei Luo, and Guang-Zhe Xiong. A Provably Correct Operating System: delta-Core. *Operating Systems Review*, 35(1):17–33, January 2001.

[47] Michael Hohmuth and Hendrik Tews. The VFiasco Approach for a Verified Operating System. In *2nd ECOOP Workshop on Program Languages and Operating Systems (ECOOP-PLOS 2005).*

[48] Takahashi Award JSSST 2005 `http://www.jssst.or.jp/admin/awards.html`.

[49] PPL 2007 Best Paper Award `http://www.sato.kuis.kyoto-u.ac.jp/ppl2007/`.

[50] Reynald Affeldt and Miki Tanaka and Nicolas Marti Formal Proof of Provable Security by Game-playing in a Proof Assistant. In *1st International Conference on Provable Security (Provsec 2007).*

[51] FLINT group `http://flint.cs.yale.edu/`.

[52] Hongxu Cai and Zhong Shao and Alexander Vaynberg. Certified Self-Modifying Code. In *Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, CA, pages 66-77, June 2007.

[53] Galen C. Hunt and James R. Larus and Martin Abadi and Mark Aiken and Paul Barham and Manuel Fahndrich and Chris Hawblitzel and Orion Hodson and Steven Levi and Nick Murphy and Bjarne Steensgaard and David Tarditi and Ted Wobber and Brian Zill. An Overview of the Singularity Project. Microsoft Research Technical Report. MSR-TR-2005-135.

[54] M. Gargano and M. Hillbrand and D. Leinenbach and W. Paul. On the correctness of operating system kernels. In *Proc. 18th International Conference on Theorem Proving in High-Order Logics (TPHOLs'05)*, pages 2–16. Springer-Verlag, 2005.

[55] Andrew Appel Tactics for Separation Logics. `http;//www.cs.princeton.edu/~appel/papers/septacs.pdf`.

[56] Edmund Clarke, Ming-Hsien Tsai, Yih-Kuen Tsay, and Bow-Yaw Wang. Separation Logic: A Tutorial Survey. `http://www.icast.org.tw/project/project-305-cmu-programs-software-security-evaluation-systems/project-305-cmu/projectmanager.2006-10-31.3682464242/publication/seplog-survey-mht.pdf/view`.

[57] Chunxiao Lin, Yiyun Chen, Long Li, and Bei Hua. Garbage Collector Verification for Proof-Carrying Code. In *JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY*, 2007, VOL 22; NUMBER 3, pages 426-437, Springer

[58] G. Klein, H. Tuch, and M. Norrish. Types, Bytes, and Separation Logic. In *34th symposium Principles of Programming Language*, ACM PRESS, pp. 97-108, 2007.

[59] Andrew Appel and Sandrine Blazy. Separation logic for small-step Cminor. In *20th Int. Conference on Theorem Proving in Higher Order Logics (TPHOLs 2007)*, Kaiserslautern, Germany 10-13 September 2007. LNCS 4732, pp.5-21.

[60] Frederic Besson. Micromega: a reflexive tactic for a fragment of integer arithmetics. `http://coq.inria.fr/contribs/Micromega.html`

[61] Frederic Besson. Fast Reflexive Arithmetic Tactics: the linear case and beyond. In *Types for Proofs and Programs (Types'06)*, LNCS 4502, pages 48–62, Springer 2007.

[62] G. Klein and T. Nipkow A Machine-checked Model for a Java-like Language, Virtual Machine and Compiler. Technical Report 0400001T.1, National ICT Australia, Sydney, March 2004.

[63] M. Strecker Formal Verification of a Java Compiler in Isabelle. In *Proc. Conference on Automated Deduction (CADE02)*, 2002, Vol. 2932 of LNCS, pages 63–77, Springer.

[64] K. Okuma and Y. Minamide Executing Verified Compiler Specification. In *Proc. of the First Asian Symposium on Programming Languages and Systems (APLAS03)*, 2003, Vol. 2895 of LNCS, pages 178–194, Springer.