Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Gillian: Shared Memory Concurrency

Author:
Tiberiu Bucur

Supervisor:
Dr. Azalea Raad

Second Marker:
Dr. Robert Chatley

June 22, 2023

Abstract

This project extends Gillian, a multi-language program analysis tool parametric in the memory model of the target language, built by the Verified Software research group at Imperial College London, with support for shared memory concurrency. Used, among others, as a verification tool, Gillian was only able to verify purely sequential code prior to this project. Building on previous research in the theory of software verification, this project enriches Gillian with the ability to verify concurrent programs, by implementing the model of fractional permissions. The work presented here focuses on extending WISL, a while-like language used as a didactic tool in the Scalable Software Verification course at Imperial College London, with fractional permissions, in the shape of a new target language, called WISLFP. This constitutes a foundational work in the process of providing support for concurrency to Gillian, in an attempt to make it competitive alongside other academic or industrial verification tools.

Acknowledgements

This project was the crowning of 4 years of hard work and dedication as an MEng student in the Department of Computing here at Imperial. For this journey and everything it has brought me, for all the smiles and laughter, all the difficult moments, sleepless nights and proud successes altogether, and, most importantly, for the unbelievable, unique memories I will cherish for the rest of my life, I am truly grateful.

There are so many people that have contributed to this success, and perhaps this page is not big enough to include all of them. Firstly, I want to thank my supervisor, Dr. Azalea Raad, for bumping into me on the corridors of Huxley by chance, in October, and introducing me to this fascinating project, and for believing in me and supporting me ever since. I want to also thank Sacha Ayoun and Petar Maksimovič, for their relentless and patient guidance, answering all my questions and helping me during debugging.

Secondly, I want to thank my family for the big emotional and financial sacrifices they made so I can go fulfill my dream of studying here. It has been the experience of my life, and none of it would have been possible without them.

Thirdly, I want to thank my grandmother, who, ever since I was a little kid, has taught me the most important lesson in life: to never give up. Thank you grandma, I hope you can see me from up there and are proud of me.

I want to thank my amazing friends that I was once praying for and never thought I would have, who have been here for me through thick and thin and have made these the best years of my life. To Adi, Ştefi, Matei, Radu, Andrei, Daniel, Raluca, Daria, Alessandro, Sudo, Ghush, and many more, thank you! I love you all.

Special thanks go to perhaps my dearest friend on this planet, Jamie Willis, who showed me that when you think you can't do it anymore, you can still do a bit more. I couldn't have done it without you, man.

Finally, to the greatest thing that has happened to me in a time I least deserved it, to Christina, who destroyed all my pre-judgements and misconceptions, challenged me and motivated me to become a better person and, most importantly, showed me what true love is, thank you. I love you, darling.

Contents

Chapter 1

Introduction

1.1 Towards a world of verified software

In the context of the world we live in today, technological advancements have been at the centre of making people's lives easier. We have come a long way from the very first computer machines ever built, which Alan Turing famously used to decypher secret Nazi messages in the Second World War, to the omnipresent microchips we interact with on a daily basis. Every part of our life, from our weekly shopping, to our holiday plans and travelling, our health system, financial situation, forms of getting information, entertainment or completing routine activities like merely turning a lamp on or off, all of them have been slowly but steadily taken over by innovative use of software in an attempt to facilitate comfort. The amount of code we interact with on a daily basis is in the order of billions of lines.

It is becoming obvious that, as the modern world is getting increasingly dominated by software, ensuring reliability of said software is an imperious challenge. It has been shown before that human errors in the code written for mission critical software has incurred enormous financial damage [1, 2] and even cost lives [3]. Testing production code extensively is an attempt at preventing some of the bugs that would otherwise occur in untested software. However, tests are often not exhaustive of the entire set of possible behaviours a program could exhibit, and most of the times can carry the programmer's own biases about what they think the code they wrote should do. In the words of Edsger W. Dijkstra, *testing can prove the presence, not the absence of bugs.*

It seems that more complex software reliability techniques are needed to prevent tragedies similar to that of Therac-25 from happening again. Indeed, software verification tools based on symbolic execution [4, 5] are being developed and used on industry scale code. We mention VeriFast [6, 7], an academic tool for semi-automated verification of Java and C, used on verification of FreeRTOS and AWS code, and Infer [8, 9], which successfully verified millions of lines of Java and C/C++ production code as part of Meta's code review pipeline.

Gillian [10, 11, 12, 13] is another such semi-automatic verification tool, which brings together the exhaustive nature of symbolic testing with the power of separation logic (Section 2.2). Developed internally at Imperial College London by the Verified Software Group, Gillian has already been able to verify and find bugs in a deserialising SDK used within the core of the client-server interaction code in AWS. The advantage of Gillian over other symbolic execution tools is that it is parametric in the memory model. That is, one can provide a (concrete or symbolic) memory model of any target language, which would in turn be used as a compiler from the target language to GIL, a small goto language used by Gillian internally. Gillian has currently been instantiated to JavaScript, C and WISL (Section 2.5.1) and ongoing efforts are in place for a novel Rust instantiation.

1.2 The importance of concurrency

For decades, software performance has been primarily influenced by technological advancements in the design and manufacturing of processing units. Getting increasingly smaller and fitting more transistors per unit of space, CPUs guaranteed an approximate doubling of performance every 18 months, as observed empirically in Moore's Law with Dennard scaling. Unfortunately, physical limitations have been at the root of an observed rapid flattening in CPU performance over the last decade, despite the number of transistors still being on the rise (Figure 1.1). With this in mind, all CPU designers and manufacturers have decided to use the ever-growing number of transistors by separating a processing unit into different CPU cores, producing the multicore architectures of today. However, the implication this has for software developers is crucial: in order to extract the performance benefits of the latest processors, they have to make explicit use of parallelism in their code. This comes with obvious challenges such as preventing different threads of execution from accessing certain parts of the code at the same time if this would lead to corruption of the system state (data races), ensuring consistency across different memory models, ensuring freedom from starvation for any thread of execution in a program etc.

In fact, operating systems have been the heaviest users of concurrency paradigms since their incipient stages. Since an OS has to be able to run multiple processes simultaneously in order ensure a good overall user experience by giving the illusion of parallelism (between device drivers responsible for processing network packets or user input, user space applications like compilers, browsers etc.), ensuring correct concurrent execution has been at the core of OS kernel development even before multicore architectures.

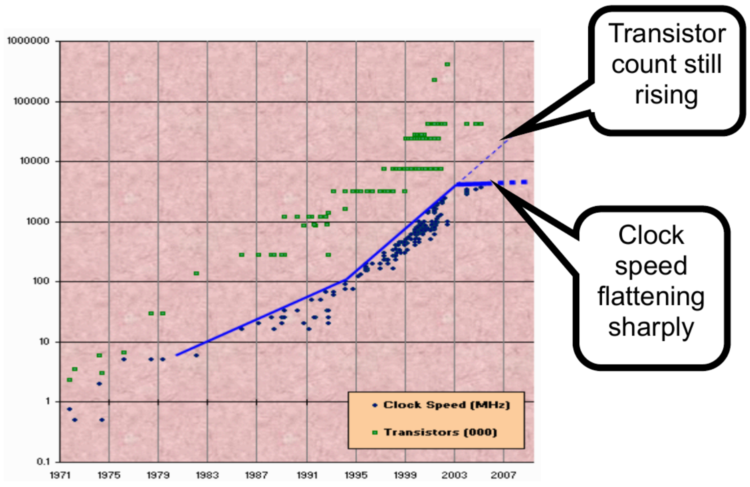


Figure 1.1: The end of Moore's Law with Dennard scaling. From [14].

1.3 Objectives

The C standard defines data races as undefined behaviour, meaning that, should such behaviour occur, no resulting outcome is excluded. This fact is very dangerous when unwrapping carefully, because it means the compiler can produce code that can affect the system in any way possible: it can crash, it can corrupt the memory, produce the wrong result or even execute as intended, but, crucially, **it will behave non-deterministically**, in this case meaning we might see the same code behaving differently across different executions. This makes debugging concurrent code an enormous challenge: how can we find out where the problem is if we cannot even reproduce the problem? However, we still want our software to behave correctly before anything, so as to avoid catastrophic outcomes. It should be obvious that ensuring correctness of essentially concurrent software is a crucial and far from mundane task.

So far, due to the way it has been essentially designed, Gillian has only been able to verify sequential code. This is an obvious limitation that this project attempts to rectify. Previous work has been done to come up with a complete semantics for verifying correctness of concurrent code (Section 2.3). Fractional permission concurrency has emerged as a clear, flexible and intuitive candidate for specifying concurrency.

The objective of this project is to enrich Gillian with support for shared memory concurrency, by extending Gillian's WISL instantiation with fractional permissions and successfully verifying useful concurrent algorithms, in the form of binary tree traversals and lambda term substitution. Although shared memory concurrency is not a new concept in the world of verification, as several other tools already support it [6, 15], we claim to have successfully added this support for Gillian and thus make the first step towards bridging the gap to the competition. This work is an initial effort in supporting concurrency paradigms for our tool and its purpose is to provide a proof of concept and lay the foundations for future development in concurrent program verification.

Chapter 2

Background

2.1 Hoare Logic

In 1969, Tony Hoare proposed *Hoare Logic* [16] as a way to specify imperative programs. Hoare Logic is based on so-called "Hoare triples" which encompass the following semantics: given a state that satisfies the pre-condition P, if the code fragment C terminates, it will produce a state satisfying the post-condition Q.

$$\vdash \{P\}\, C\, \{Q\}$$

P and Q are first-order logic assertions. Hoare logic also provides inference rules in order to ensure compositionality of derivations. That is, using certain axioms that we can prove as they are, we can derive facts about a command based on previously derived facts about that same command (or a subcommand, should the original one be a sequence of several different commands). For example, here is the assignment rule:

$$\frac{}{\vdash \{P\}\, x := E\, \{P[E/x]\}}$$

Using this axiom, we can create more complicated proofs, combining several applications of this (or other axioms). For example, using the rule of composition:

$$\frac{\{P\}\, C\, \{Q\}, \{Q\}\, C'\, \{T\}}{\vdash \{P\}\, C; C'\, \{T\}}$$

We can prove the following very simple specification. Starting from a state where both variables a and b have value 0, after performing the 2 assignments, we get to a state where a has value 1 and b has value 2.

$$\frac{\{a = 0, b = 0\}\, a := 1; b := 2\, \{a = 1, b = 0\}, \{a = 1, b = 0\}\, b := 2\, \{a = 1, b = 2\}}{\vdash \{a = 0, b = 0\}\, a := 1; b := 2\, \{a = 1, b = 2\}}$$

In this example, we have only looked at stack variables as possible targets for side-effecting commands. We can, however, model the heap as well, using the binary cell predicate $x \hookrightarrow a, z$, which describes a set of heaps containing only the value a at address x, and the value z at address x + 1. We use $x \hookrightarrow -$ as a shortcut for $\exists z.x \hookrightarrow z$ Using this, we can construct predicates that describe real data structures. Take, for example, the list predicate

$$\mathit{List}(x) \overset{\texttt{def}}{=} \exists z.x \hookrightarrow -, z \wedge \mathit{List}(z)$$

This predicate says that address x holds a number, the first element of the linked list, and address x + 1 holds a pointer to the next element of the list. Using this predicate, we might express pre-conditions like $\mathit{List}(x) \wedge \mathit{List}(y)$, which intuitively should tell us that there are 2 different (disjoint) lists, one at address x, and the other at address y [17]. However, first-order logic does not forbid x and y from being equal to each other or having common elements (one being a sublist of the other). In order to prevent this from happening, we would have to define a

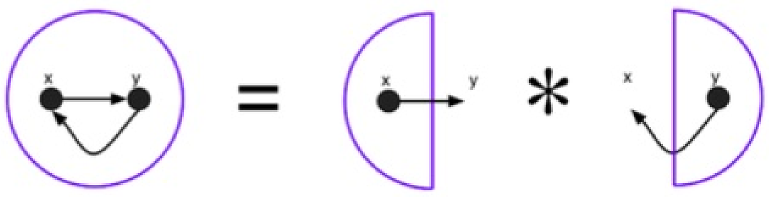


Figure 2.1: Illustration of the frame rule. From [20].

predicate like $reach(x,u) \overset{\texttt{def}}{=} x = u \lor (\exists z.x \hookrightarrow -, z \land reach(z,u))$, and then our predicate would look like following:

$$List(x) \land List(y) \land (\forall u.reach(x,u) \land reach(y,u) \implies u = null)$$

If we were to do the same for three lists instead of two, we would have to restrict each pair of lists to be disjoint. We can quickly see that this implies an exponential growth in the number of references to the reach predicate, so it clearly does not scale. In fact, it took researchers 30 years to find a fix to this problem, to come up with a scalable form of logic for reasoning about programs.

2.2 Separation Logic

Separation Logic was introduced in 2001 by Peter O'Hearn, John Reynolds and Hongseok Yang [18] as a solution to the limitations exposed by Hoare Logic. The assertion language works with partial instead of complete heaps, based on the work of O'Hearn and Pym on bunched implications [19]. Similar to the way we did before in Hoare Logic, we define the binary cell predicate $x \mapsto a, y$ to describe the singleton set containing the heap with *only* the binary cell with value a at address x and value y at address x + 1. Like before, we write $x \mapsto -, y$ to mean $\exists a.x \mapsto a, y$.

The breakthrough consists in the appearance of the separating conjunction operator, *, which combines two disjoint heaps, h1 with property P1 and h2 with property P2, to obtain a bigger heap h. We denote with e the logical environment containing logical variables (different from program variables, as they do not appear in the code) used inside assertions, the current state of the program with s, and the set of all heaps as Heap. We write $h1 \uplus h2$ to mean the disjoint union of heaps h1 and h2 [17].

$$e, s, h \vDash P1 * P2 \iff \exists h1, h2 \in Heap.h = h1 \uplus h2 \land e, s, h1 \vDash P1 \land e, s, h2 \vDash P2$$

We can therefore write $a \mapsto 1 {*} b \mapsto 2$ for the heap containing *exactly* two cells, the cell at address a containing 1, and the cell at address b containing 2. We write `emp` for the empty heap and put a dot symbol across any operator to denote "and the heap is empty" (eg. $a \doteq 1 \iff a = 1 \land \texttt{emp}$).

Just like previously in Hoare Logic, we have axioms that help us prove pre- and post-conditions about programs, making use of the same Hoare triple notation. Particularly of interest is the frame rule, which says that if I can prove a certain post-condition Q for program C starting from P, then I can also show Q * R starting from P * R, assuming the program does not interfere with any free variables in assertion R.

$$\frac{\{P\}\,C\,\{Q\}, mod(C) \cap pv(R) = \emptyset}{\{P * R\}\,C\,\{Q * R\}}\ \texttt{FRAME}$$

It is now obvious how we can prove the very simple example above using the frame rule.

$$\dfrac{\dfrac{\dfrac{}{\{a \doteq 0\}\,a := 1\,\{a \doteq 1\}}\ \texttt{ASSIGN}}{\{a \doteq 0 * b \doteq 0\}\,a := 1\,\{a \doteq 1 * b \doteq 0\}}\ \texttt{FRAME} \quad , \quad \dfrac{\dfrac{}{\{b \doteq 0\}\,b := 2\,\{b \doteq 2\}}\ \texttt{ASSIGN}}{\{a \doteq 1 * b \doteq 0\}\,b := 2\,\{a \doteq 1 * b \doteq 2\}}\ \texttt{FRAME}}{\{a \doteq 0 * b \doteq 0\}\,a := 1; b := 2\,\{a \doteq 1 * b \doteq 2\}}\ \texttt{SEQ}$$

Notice how we completely *framed off* the part of the heap we were not interested in for each one of the two assignments. In the case when dealing with heap addresses and not stack variables, as was the previous example, the semantics of the separating conjunction actually *forbids* us from duplicating a resource, since it operates only on disjoint heaps. This is where the breakthrough manifests: we can now write $List(x) * List(y)$ and it would mean that the two lists at addresses x and y respectively **are disjoint**, since any duplicate element between the 2 would break the assumption of the disjointness of the 2 partial heaps. We have effectively reduced the mundane task of ensuring the two lists do not overlap by writing exponentially many pair-wise assertions to a very simple and elegant use of the * operator.

Separation logic as presented in its current form is already powerful enough to express many complex predicates and fully specify interesting data structures, such as circular doubly-linked lists or binary search trees. In fact, the current version of Gillian (Section 2.5) implements this original form of separation logic, and has proved effective at finding real bugs in production software [11]. However, this only deals with purely sequential programs. In order to extend this reasoning to concurrent programs, which is the main focus of this project, we have to look further at several proposed extensions to separation logic which can handle parallel composition of programs operating on disjoint resources (Section 2.3) and account for access to shared resources between threads (Section 2.4.2).

2.3 Concurrent Separation Logic

In 2007, Peter O'Hearn showed how separation logic can be used in a concurrent context to reason about the usage of resources [21]. Resources in a system can be anything from memory to CPU time, network bandwitdth and so on, but henceforth we will refer to parts of program state as resources. O'Hearn speculated that the spatial separation offered by separation logic through the frame rule might suit itself well to reason modularly about parallel processes with access to shared variables. He started from the early work of C.A. Hoare, "Towards a Theory of Parallel Programming" [22], where formal proof rules for shared variable concurrency were achieved through syntactic enforcement of separation. He proceeded to add the separating conjunction into the extensions of these formal proof rules, proposed by Owicki and Gries [23]. However, the rules were proved to be unsound by John Reynolds, unless certain restrictions to the types of assertions we can formulate are met: namely, assertions have to be "precise", meaning they can be satisfied by precisely one subpart of the heap, and not multiple. Brookes showed that restricting the proof rules to using only precise assertions guarantees soundness [24].

2.3.1 Fundamentals

Before we dive deeper into proving specifications of concurrent programs, we quickly mention some fundamental terms and concepts that will be prevalent across the following sections.

We denote the parallel execution of two commands through the intuitive $\|$ symbol. Our goal is to be able to prove Hoare triples of the form $\{P\}\, C1 \parallel C2\, \{Q\}$.

As per usual, programs running in concurrent settings can race with each other. Formally, a program is defined to be *racy* if two concurrent processes attempt to access the same portion of the state at the same time. Otherwise, the program is *race-free*. An example of a racy program is the following, where not only the interleaving of the 2 threads, but also the granularity of operations can lead to the program producing different results. We can imagine an addition being split into multiple load operations followed by adding the operands, the atomicity of which is dependent on the memory model implemented by the underlying architecture.

$$x := x + y \parallel x := x * z$$

O'Hearn also goes on to differentiate between cautious and daring concurrency [21]:

> *A program is cautious if, whenever concurrent processes access the same piece of state, they do so only within commands from the same mutual exclusion group. Otherwise, the program is daring.*

By mutual exclusion group we mean a group of commands that must never overlap in execution. When programming with semaphores, a mutual exclusion group is determined by the calls to the P and V operations, whereas in [22], such a group is determined by a conditional critical region of the form `with` r `when` B `do` C, where r is the name of a shared resource.

Particularly of interest is that programs such as $x := x + 1; x := x + 1$ and $x := x + 2$ are no longer guaranteed to be equivalent under interleaving semantics. It is obvious that our inability to reliably verify such a program is rooted in its inherent raciness. Therefore, we aim to formalise the verification of programs which restrict access to shared mutable state through language enforced separation (critical sections).

Some crucial concepts, particularly for shared resources concurrency, are those of ownership and separation. O'Hearn defines the two in the following way [21]:

> *Ownership Hypothesis.* A code fragment can access only those portions of state that it owns.
>
> *Separation Property.* At any time, the state can be partitioned into that owned by each process and each grouping of mutual exclusion.

What these two properties combined mean is that different code fragments (perhaps processes or threads of execution) cannot interfere with each other's local portion of the state, but can transfer these portions between each other, for the purpose of collaboration (message passing, shared variables etc.). This fact is of great significance, as it properly enables us to reason about separate processes in isolation, and thus still preserve desirable equivalences like $x := x = 1; x := x + 1$ and $x := x + 2$ so long as they operate solely on local state.

2.3.2 Disjoint Concurrency

Before we talk about shared variable concurrency, it is worth exploring the simpler case of disjoint concurrency and its applications. The rule of disjoint concurrency is presented below.

$$\frac{\{P\}\,C\,\{Q\} \quad \{P'\}\,C'\,\{Q'\}}{\{P * P'\}\,C \parallel C'\,\{Q * Q'\}}$$

It is easy to see that the semantics of the separating conjunction forbid the access of two different threads of the same program to the same piece of state, and thus rule out any possible races. In particular, we cannot prove a program such as $\{10 \mapsto -\}\,[10] := 1 \parallel [10] := 2\,\{???\}$ (where $[x] := y$ means modifying the heap at address x to take value y), since that would need splitting the heap into two sub-heaps of the form $10 \mapsto -$, and so the initial precondition must be of the form $10 \mapsto - * 10 \mapsto -$, which is unsatisfiable. We can however prove potentially racy programs, as long as the race is discounted by the semantics of the separating conjunction. Take, for example, the proof below, where x and y are known to be different from the precondition.

$$\frac{\{x \mapsto 3\}\,[x] := 4\,\{x \mapsto 4\} \quad \{y \mapsto 3\}\,[y] := 5\,\{y \mapsto 5\}}{\{x \mapsto 3 * y \mapsto 3\}\,[x] := 4 \parallel [y] := 5\,\{x \mapsto 4 * y \mapsto 5\}}$$

One might think that the disjoint concurrency paradigm is not very expressive and, indeed, it does not allow us to specify any concurrency paradigms involving sharing of resources between threads, such as readers and writers or producers and consumers. However, it does appear to be particularly suitable to embarassingly parallelisable algorithms, such as the ones exploiting a divide and conquer strategy. Consider the following example of merge-sort, where the predicate $array(a, i, j)$ describes a contiguous block of memory in the heap, from address a + i to address a + j, and the predicate $sorted(a, i, j)$ means that segment is sorted.

```
{array(a, i, j)}
procedure ms(a, i, j)
newvar m := (i + j)/2;
if i < j then
    (ms(a, i, m) || ms(a, m + 1, j));
    merge(a, i, m + 1, j);
{sorted(a, i, j)}
```

Assuming the pre and post condition of the function as hypotheses, as per usual, we can prove the body of the if statement in the following way, by splitting the heap into two sub-heaps, each one containing the portion of the array to be passed to the recursive call of the function.

$$\begin{array}{c}
\{array(a,\ i,\ m) * array(a,\ m+1,\ j)\} \\
\begin{array}{c|c}
\{array(a,\ i,\ m)\} & \{array(a,\ m+1,\ j)\} \\
\texttt{ms(a, i, m)} & \texttt{ms(a, m + 1, j)} \\
\{sorted(a,\ i,\ m)\} & \{sorted(a,\ m+1,\ j)\}
\end{array} \\
\{sorted(a,\ i,\ m) * sorted(a,\ m+1,\ j)\}
\end{array}$$

2.3.3 Shared resources concurrency

We saw how concurrent separation logic enables us to reason in isolation about threads of execution that access disjoint portions of state. This subsection will showcase how CSL can be used to verify specifications of programs making use of shared resource concurrency.

For this part of our presentation, notions described in *Fundamentals* (Section 2.3.1), namely the Ownership and Separation Properties, will be of great importance. In particular, the Ownership hypothesis materialises in that shared resources can only be accessed by their resource owners at that specific moment in time. Resource owners can be threads, but also synchronisation primitives implementing mutual exclusion groups, such as semaphores (we will give an example that illustrates this in the current section).

To ensure that threads do not interfere with each other's state in a shared variable concurrency paradigm, O'Hearn enforces the following restrictions for program well-formedness [21]:

- a variable must belong to at most one resource
- if a variable x belongs to resource r, then it cannot appear in a parallel process except in a critical region for r.
- if a variable x is changed in one process, it cannot appear in another unless it belongs to a resource.

In this sense, all variables that are shared between threads must belong to a resource, access to which is protected by critical sections that are executed with mutual exclusion. In particular, racy programs like

$x := 3 \ \ \| \ \ x := x + 1$

and

$x := 3 \ \ \| \ \texttt{with}\ r\ \texttt{when}\ true\ \texttt{do}\ x := x + 1.$

are excluded by the first two restrictions, while the third condition ensures that threads cannot interfere with each other's local variables. Notice however, that the current restrictions do not prevent race conditions in programs with pointers, such as the following, in the case that x and y are equal:

$[x] := 3 \ \ \| \ \ [y] := 4$

or even more obvious data-races, such as:

$[10] := 3 \ \ \| \ \ [10] := 4$

O'Hearn however ensures proper avoidance of interference at the level of heap cells in the proof rules by use of the separating conjunction.

Figure 2.2 presents a very simplified example of a producer-consumer paradigm through message passing implemented using semaphores and intuitively explain the reasoning behind resource ownership transfer. Once again, the semaphores are resource owners, meaning portions of the state (in this case the heap cell at location 10) are attached to sempahores. Ownership can be transferred from the semaphore to the thread and viceversa, and only when the thread acquires ownership of the resource can it make use of it (through assignment or reading). We assume that the $free$ semaphore is initialised with 1 and the *busy* semaphore is initialised with 0, therefore $free$ is the sole ownership of the heap cell. In fact, both semaphores respect the following invariant: either they are set to 1 and they own the heap cell 10, or they are set to 0 and they own nothing. The left hand side thread executes first, decreasing the semaphore's internal value to 0, and the semaphore releases ownership of the cell into the code. From there, the thread can modify the cell and place the message m in it. Once it calls V on the *busy* semaphore, the thread transfers ownership of the cell to the semaphore, which increments its value to 1. The second threads then executes P on the *busy* semaphore and acquires ownership of cell 10 from the semaphore, which decrements its value to 0. The thread reads the message at cell 10 and finally calls V on the $free$ semaphore to transfer ownership back to it and reset for a potential new message transfer.

The proof rules as highlighted by O'Hearn in hus paper are presented below. In order to prove the specification of a program of the form

$init;$
$\texttt{resource}\ r_1(variable\ list), ..., r_m(variable\ list)$
$C_1 \| ... \| C_n$

$$\begin{array}{c||c}
\ldots & \ldots \\
\{\mathtt{emp}\} & \{\mathtt{emp}\} \\
P(free); & P(busy); \\
\{10 \mapsto -\} & \{10 \mapsto -\} \\
[10] := m; & n := [10]; \\
\{10 \mapsto -\} & \{10 \mapsto -\} \\
V(busy); & V(free); \\
\{\mathtt{emp}\} & \{\mathtt{emp}\} \\
\ldots & \ldots
\end{array}$$

Figure 2.2: Producer-consumer message passing

a resource invariant RI_{r_i} must be defined for each resource r_i that satisfies that any command $x = \ldots$ which modifies a variable x that is free in RI_{r_i} must occur within a critical region for r_i. Resource invariants are also required to be *precise*, as per Brooks [24]. The inference rule is presented below:

$$\frac{\{P\}\, init\, \{RI_{r_1} * \ldots * RI_{r_m} * P'\} \quad \{P'\}\, C_1 \parallel \ldots \parallel C_n\, \{Q\}}{\{P\} init; \mathtt{resource}\ \ r_1(variable\ \ list), \ldots, r_m(variable\ \ list) C_1 \parallel \ldots \parallel C_n \{RI_{r_1} * \ldots * RI_{r_m} * Q\}}$$

We observe here that the resource invariants are established by the initialisation sequence, together with a portion of state to be accessed locally by the processes, and then re-established by the conclusion. The inference rule for parallel composition is the same as the rule for disjoint concurrency, except the triples establishing the reasoning for sequential processes $\{P_i\}\, C_i\, \{Q_i\}$ use the resource invariants RI_{r_i} in the context.

$$\frac{\{P_1\}\, C_1\, \{Q_1\} \ \ldots\ \{P_n\}\, C_n\, \{Q_n\}}{\{P_1 * \ldots * P_n\}\, C_1 \parallel \ldots \parallel C_n\, \{Q_1 * \ldots * Q_n\}}$$

This additional contextual information becomes useful in the rule for conditional conditional critical regions:

$$\frac{\{(P * RI_r) \wedge B\}\, C\, \{Q * RI_r\}}{\{P\}\, \mathtt{with}\ \ r\ \ \mathtt{when}\ \ B\ \ \mathtt{do}\ \ C\ \ \mathtt{endwith}\, \{Q\}}$$

Below we show how we can implement semaphores in terms of conditional critical regions. Note that this implementation is just an aid for ease of reasoning: one would actually prefer implementing semaphores in terms of CCRs in practice.

```
    P($s$) = with $s$ when $s > 0$ do $s := s - 1$ endwith
    V($s$) = with $s$ when true do $s := s + 1$ endwith
```

Now we have reached a point where we can prove the specification of the producer-consumer paradigm using semaphores described above. First, we establish the resource invariants, which for both the *free* and *busy* semaphores are as follows:

$$RI_s = (s = 0 \wedge \mathtt{emp}) \vee (s = 1 \wedge 10 \mapsto -)$$

Then we can prove the triples for the semaphore operations using rules derived from the CCR proof rules:

$$\frac{\{(A * RI_s) \wedge s > 0\}\, s := s - 1\, \{A' * RI_s\}}{\{A\}\, P(s)\, \{A'\}}$$

$$\frac{\{A * RI_s\}\, s := s + 1\, \{A' * RI_s\}}{\{A\}\, V(s)\, \{A'\}}$$

Here is how we can obtain the triple $\{\mathtt{emp}\}\, V(free)\, \{10 \mapsto -\}$. Notice how after the effects of the command are propagated to the post condition, we have to re-establish the resource invariant.

$$\begin{array}{l}
\{10 \mapsto - * ((free = 0 \wedge \texttt{emp}) \vee (free = 1 \wedge 10 \mapsto -)\} \\
\{10 \mapsto - * (free = 0 \wedge \texttt{emp})\} \\
free := free + 1 \\
\{10 \mapsto - * (free = 1 \wedge \texttt{emp})\} \\
\{\texttt{emp} * (free = 1 \wedge 10 \mapsto -)\} \\
\{\texttt{emp} * ((free = 0 \wedge \texttt{emp}) \vee (free = 1 \wedge 10 \mapsto -))\}
\end{array}$$

and similarly we can prove it for the P operation. In order to combine the two sequential processes together using the concurrency rule, we have to wrap them around while loops, assuming there would always be messages to be generated by the left process (and to be consumed by the right process). Below is a proof sketch of the overall program:

$$\begin{array}{c}
\{10 \mapsto -\} \\
free := 1, busy := 0; \\
\{(free = 1 \wedge 10 \mapsto -) * (busy = 0 \wedge \texttt{emp})\} \\
\{RI_{free} * RI_{busy} * \texttt{emp} * \texttt{emp}\} \\
\texttt{resource}\ \ free(free), busy(busy); \\
\{\texttt{emp} * \texttt{emp}\} \\
\begin{array}{c||c}
\{\texttt{emp}\} & \{\texttt{emp}\} \\
\texttt{while true do} & \texttt{while true do} \\
\{\texttt{emp} \wedge true\} & \{\texttt{emp} \wedge true\} \\
\{\texttt{emp}\} & \{\texttt{emp}\} \\
produce\ m & \texttt{P}(busy); \\
\{\texttt{emp}\} & \{10 \mapsto -\} \\
\texttt{P}(free); & n := [10]; \\
\{10 \mapsto -\} & \{10 \mapsto -\} \\
[10] := m; & \texttt{V}(free); \\
\{10 \mapsto -\} & \{\texttt{emp}\} \\
\texttt{V}(busy); & \texttt{consume n} \\
\{\texttt{emp}\} & \{\texttt{emp}\} \\
\{\texttt{emp} \wedge true\} & \{\texttt{emp} \wedge \neg true\} \\
\{\texttt{false}\} & \{\texttt{false}\}
\end{array} \\
\{\texttt{false} * \texttt{false}\} \\
\{RI_{free} * RI_{busy} * \texttt{ false}\} \\
\{\texttt{false}\}
\end{array}$$

More complex examples of daring concurrent programs can be correctly verified using the mechanism we just described. O'Hearn goes on to show how we can apply the same reasoning to pointer-transferring buffers and memory managers, which are clear examples of daring concurrency, since the shared bits of state will be accessed from several different critical regions. Transferring complete ownership of a resource between resource owners and restricting processes to only accessing resources they own ensures safety properties ensured by the lack of data-races implied by the ownership and separation properties. However, a data-race is defined to be multiple unordered accesses to a shared state by two concurrent processes, *at least one of which is a write.* Clearly, multiple reads of the same resource should be allowed, since they do not constitute a data-race and therefore do not carry the risk of state corruption or wrong behaviour. There are many examples of useful concurrent programing paradigms where concurrent read-only access to a piece of state can achieve higher performance than complete mutual exclusion. One such example is the readers and writers paradigm, where multiple readers are allowed to access the same portion of state, provided nobody is trying to write to it at the same time, in which case only the writer has access to that resource. Our current reasoning model does not encapsulate this notion of *passivity.* We will see in the next section how introducing different types of permissions and splitting them between processes can correctly model this behaviour and thus enable verifications of programs such as readers and writers.

2.4 Permissions

We have seen in the previous section how separation logic can describe ownership transfer of resources between processes. Heap cells are passed around between resource owners (in our examples before processes and semaphores) and the ownership and separation properties guarantee that each process will only access the resources it owns. This, however, is not enough to express several types of concurrent programs, in particular those that make use of different processes reading shared data concurrently. Consider this very simple example in a loosely specified pseudocode language with pointers, slightly different from our While language we have discussed before [25]:

```
a := new; b := new; c := new; d = a;
    *b += *a; || *c += *d;
        *a += *b + *c
```

In this program, we allocate three new heap cells and make d an alias of the first one. Then, the parallel actions read the same heap cell pointed to by both a and d, but write to separate locations. After the two forked processes are re-joined, the main process writes to the previously shared location a. Clearly, the approach of complete ownership for the heap cell pointed to by a and d is overly conservative, since the parallel processes only read that cell. However, we cannot compose these 2 processes using the concurrency rule. The way this has been dealt with before was to assign part of the state as immutable and the rest as mutable [26]. Only mutable state required separation, while immutable state (here the cell pointed to by a and d) could be shared across processes. However, state could move from the mutable to the immutable side, but not viceversa, and so we could not express a temporary read access to a shared state and invalidate it later. It seem that, since separation logic deals only with exclusivity, we have to find a new way to express what having access to a resource means in a non-exclusive way.

We introduce permissions as a new mechanism to express the right of a process to "access" a resource. The classic heap cell assertion from separation logic $N \mapsto E$ can be read both as a predicate asserting the heap cell contains only one cell at location N, containing E, and as a *permission* of the current process to read, write or dispose the cell. As the reader might expect, the interpretation of the empty heap predicate `emp` is that the current process has no permissions to access any cells.

If we see the concept of ownership described in the previous section as a total read/write/dispose permission, it becomes apparent that permissions can be split into an unbounded number of read-only permissions for any shared resource, thus ensuring passivity. Permissions can be split between multiple processes as many times as needed (since a read-only resource can be read by an unbounded number of processes). However, gathering the permissions back to reform the full ownership proves to be tricky, since either the original resource owner has to know how many read permissions its total permission has been split into, or the processes the permissions have been handed in to have to make sure they regather the exact number of permissions they gave out in the first place, before returning their full slice of the permission. This issue of *permission accounting* is discussed by Bornat et al. [27] in their paper, and two alternative strategies emerged from their analysis, both suitable to some but not all possible applications:

- Fractional permissions (Section 2.4.2) has emerged from the work of John Boyland in 2003 [28] who proposed a model where permissions could be represented as rational number z, with $0 < z \leq 1$. In his model, $z = 1$ would give exclusive ownership of the resource to the process, while any other value of z would guarantee read-only access. He correctly pointed out that separation logic could not support this at that moment, since the concurrency proof rule only dealt with exclusive access. However, he did suggest CSL could be extended with a predicate of the form $P \models \epsilon P * (1 - \epsilon)P$, to allow for separate reasoning about the shared portions of heap. Fractional permissions suit themselves well to divide and conquer algorithms and symmetric data structures like binary trees, since permissions can be easily split off at each level of the call stack/binary tree.

- Counting permissions (Section 2.4.1), proposed by Bornat et al. [27], is an alternative to the fractional permissions model that suits better applications that have a count of the number of permissions handed out, like the readers and writers problem. A source permission can only be split off by the "permission authority" and is annotated with the number of (read) permissions split off from it. The resulting read permissions cannot be split further, and only the original source with no split off permissions gives complete ownership of the resource.

We will take a look at both counting and fractional permissions in the upcoming subsections and will see that the two models are equivalent. However, their expressiveness makes them more useful for certain applications than others, as we discussed above.

2.4.1 Counting Permissions

In order to model counting permissions faithfully, we need to distinguish between the "source permission" and read permissions that have been split off from it. We also have to differentiate between the original, total permission, which guarantees complete ownership of the resource, and a permission which lacks some split off parts.

A total permission is denoted by $E \overset{0}{\mapsto} E'$. The 0 signifies that no permissions have been split off from it, making this the total permission. Intuitively, a source permission from which n read permissions have been split is denoted by $E \overset{n}{\mapsto} E'$. Conforming to our small algebra, a read permission would then be denoted by $E \xmapsto{-1} E'$, in order for the addition to model the re-gathering correctly. However, Bornat et al. [27] choose to use the notation $E \rightarrowtail E'$, to simplify the proof theory.

$$E \overset{n}{\mapsto} E' \Longrightarrow n \geq 0$$
$$E \overset{n}{\mapsto} E' \wedge n \geq 0 \Longleftrightarrow E \xmapsto{n+1} E' * E \rightarrowtail E'$$

Notice how the equivalence on the second line describes the splitting of permissions into a new read permission and the same source permission with an incremented number of split-offs. The axioms for allocation, disposal, reading and assignment are as follows:

$$\{\mathtt{emp}\}\, x := \mathit{new}(E)\, \{x \overset{0}{\mapsto} E\}$$
$$\{E' \overset{0}{\mapsto} -\}\, \mathit{dispose}\ E'\, \{\mathtt{emp}\}$$
$$\{E' \overset{0}{\mapsto} -\}\, [E'] := E\, \{E' \overset{0}{\mapsto} E\}$$
$$\{E' \rightarrowtail E\}\, x := [E']\, \{E' \rightarrowtail E \wedge x = E\}$$

We observe that only a total permission can write or dispose a cell, and creating a new cells gives, as expected, total ownership of that cell, while reading a cell can be performed with just a read permission.

We mentioned above the readers and writers program as a suitable example to illustrate counting permissions. Here is a pseudocode implementation of this paradigm, using conditional critical regions:

```
READERS                                              WRITER

with $read$ when true do
    if $count = 0$ then P($write$) else skip
    $count$ +:= 1                                    P($write$)

... reading happens here ...                         ... writing happens here ...

with $read$ when $count > 0$ do                      V($write$)
    $count$ -:= 1
    if $count = 0$ then V($write$) else skip
```

In order to prove this program, we first have to specify the resource invariants. Suppose the shared resource is the heap cell pointed to by y. Then, the 2 semaphores have the following resource invariants:

$$\mathit{write}\text{: if } \mathit{write} = 0 \text{ then } \mathtt{emp} \text{ else } y \overset{0}{\mapsto} -$$
$$\mathit{read}\text{: if } \mathit{count} = 0 \text{ then } \mathtt{emp} \text{ else } y \xmapsto{\mathit{count}} -$$

Using these resource invariants just established, we can prove the pre-conditions and post-conditions of the P and V operations.

```
$\{\texttt{emp}\}$
P$(write)$ :
    $\{(\texttt{emp} * \text{if } write = 0 \text{ then } \texttt{emp} \text{ else } y \overset{0}{\mapsto} - \wedge write = 1)\}$
    $\{(\texttt{emp} * y \overset{0}{\mapsto} -) \wedge write = 1\}$
    $write := 0$
    $\{y \overset{0}{\mapsto} - * (\texttt{emp} \wedge write = 0)\}$
    $\{y \overset{0}{\mapsto} - * (\text{if } write = 0 \text{ then } \texttt{emp} \text{ else } y \overset{0}{\mapsto} - \wedge write = 0)\}$
$\{y \overset{0}{\mapsto} -\}$

$\{y \overset{0}{\mapsto} -\}$
V$(write)$ :
    $\{y \overset{0}{\mapsto} - * \text{if } write = 0 \text{ then } \texttt{emp} \text{ else } y \overset{0}{\mapsto} -\}$
    $\{y \overset{0}{\mapsto} - * (\texttt{emp} \wedge write = 0\}$
    $write := 1$
    $\{\texttt{emp} - *(\texttt{emp} \wedge write = 1\}$
    $\{\texttt{emp} * (\text{if } write = 0 \text{ then } \texttt{emp} \text{ else } y \overset{0}{\mapsto} - \wedge write = 1)\}$
$\{\texttt{emp}\}$
```

Then the proof for that the readers prologue code releases a read permission inside the program is as follows. The epilogue performs the opposite action, absorbing the read permission from the code into the resource owner, with the additional requirement that count must be non-zero on entry.

```
$\{\texttt{emp}\}$
with $read$ when true do
    $\{\text{if } count = 0 \text{ then } \texttt{emp} \text{ else } y \xmapsto{count} - * \texttt{emp}\}$
    if $count = 0$ then
      $\texttt{emp}$ P$(write)\{y \overset{0}{\mapsto} -\}$
    else $\{y \xmapsto{count} -\}$ skip $\{y \xmapsto{count} -\}\{y \xmapsto{count} -\}$
    $count$ +:= 1
    $\{y \xmapsto{count-1} -\}$
    $\{y \xmapsto{count} - * z \rightarrowtail -\}$
$\{z \rightarrowtail N\}$

$\{z \rightarrowtail N\}$
with $read$ when $count > 0$ do
    $\{\text{if } count = 0 \text{ then } \texttt{emp} \text{ else } y \xmapsto{count} - * z \rightarrowtail N \wedge count > 0\}$
    $count$ -:= 1
    $\{\text{if } count + 1 = 0 \text{ then } \texttt{emp} \text{ else } y \xmapsto{count+1} - * z \rightarrowtail N \wedge count + 1 > 0\}$
    $\{y \xmapsto{count+1} - * z \rightarrowtail N \wedge count \geq 0\}$
    if $count = 0$ then
      $\{y \overset{0}{\mapsto}\}$V$(write)\{\texttt{emp}\}$
    else $\{y \xmapsto{count} -\}$ skip $\{y \xmapsto{count} -\}$
    $\{\text{if } count = 0 \text{ then } \texttt{emp} \text{ else } y \xmapsto{count} - * \texttt{emp}\}$
$\{\texttt{emp}\}$
```

2.4.2 Fractional Permissions

We detail below the model of fractional permissions as presented by Bornat et al. [27], which is in agreement with the numerical scheme presented by John Boyland [25]. We note that in his paper, Boyland mentions a number of alternative models and goes in detail in his presentation of more complex concepts like fraction multiplication and nested permissions. We see little benefit in covering these concepts here, as the scope of the project is restricted to the implementation of fractional permissions (and perhaps counting permissions) models, and any possible extensions would be decided at a later stage.

A permission is described by a rational number z, with $0 < z \leq 1$. $z = 1$ is a complete permission, allowing read, write and dispose operations to the holder, while any number $z < 1$ denotes a read only permission. As in the counting permissions model, we model the heap cell assertion with the level of permission it carries:

$$x \overset{z}{\mapsto} E \Longrightarrow 0 < z \leq 1$$

Heaps can only be combined using their separating conjunction if the heap cells with matching locations point to the same value and their permissions combine arithmetically (they yield a number z', with $0 < z' \leq 1$). The reverse implication states that permissions can always be split in 2 read-only permissions. Their exact arithmetic values do not aid in reasoning, since any fraction $z < 1$ represents a read-only permission. Note, however, that all permissions are restricted to positive numbers, to avoid fractions that arithmetically combine to legal values but which do not possess a meaningful semantics, such as $z = 2, z' = -1, z + z' = 1$.

$$x \overset{z}{\mapsto} E * x \overset{z'}{\longmapsto} E \Longleftrightarrow x \overset{z+z'}{\longmapsto} E \wedge z > 0 \wedge z' > 0$$

Below are the axioms for fractional permissions. Similarly to what we have already seen with counting permissions, new, dispose and assignment need full permissions, while reading can operate with any permission.

$$\begin{aligned}
&\{\texttt{emp}\}\, x := \mathit{new}()\, \{x \overset{1}{\mapsto} -\}\\
&\{E \overset{1}{\mapsto} -\}\, \mathit{dispose}\ E\, \{\texttt{emp}\}\\
&\{x \overset{1}{\mapsto} -\}\, [x] := E\, \{x \overset{1}{\mapsto} E\}\\
&\{E' \overset{z}{\mapsto} E\}\, x := [E']\, \{E' \overset{z}{\mapsto} E \wedge x = E\}
\end{aligned}$$

We can see below a very simple and intuitive example of how we can use fractional permissions in our reasoning for an example similar to the pointer aliasing problem described above:

$$\begin{array}{c}
\{\texttt{emp}\}\\
x := \mathit{new}();\\
\{x \overset{1}{\mapsto} -\}\\
[x] := 7\\
\{x \overset{1}{\mapsto} 7\}\\
\{x \overset{0.5}{\longmapsto} 7 * x \overset{0.5}{\longmapsto} 7\}\\
\begin{array}{c|c}
\{x \overset{0.5}{\longmapsto} 7\} & \{x \overset{0.5}{\longmapsto} 7\}\\
y := [x] - 1; & z := [x] + 1;\\
\{x \overset{0.5}{\longmapsto} 7 \wedge y = 6\} & \{x \overset{0.5}{\longmapsto} 7 \wedge z = 8\}
\end{array}\\
\{x \overset{0.5}{\longmapsto} 7 * x \overset{0.5}{\longmapsto} 7 \wedge y = 6 \wedge z = 8\}\\
\{x \overset{1}{\mapsto} 7 \wedge y = 6 \wedge z = 8\}\\
\mathit{dispose}\ x;\\
\{\texttt{emp} \wedge y = 6 \wedge z = 8\}
\end{array}$$

The example presented by Bornat et al [27] to illustrate the usefulness of the fractional permissions model is the one of lambda term substitution. We remind the reader of the syntax of lambda terms: $T ::= \texttt{Lam}\ v\ T \mid \texttt{App}\ T\ T \mid \texttt{Var}\ v$, and define substitution in the obvious way:

$$\begin{aligned}
&(\texttt{Lam}\ v'\ \beta)[\tau/v] = \begin{cases} \texttt{Lam}\ v'\ (\beta[\tau/v]) & v \neq v' \\ \texttt{Lam}\ v'\ \beta & v = v' \end{cases}\\
&(\texttt{App}\ \phi\ \alpha)[\tau/v] = \texttt{App}\ (\phi[\tau/v])\ (\alpha[\tau/v])\\
&(\texttt{Var}\ v')[\tau/v] = \begin{cases} \texttt{Var}\ v' & v \neq v' \\ \tau & v = v' \end{cases}
\end{aligned}$$

Following is a possible heap representation as an abstract syntax tree for a lambda term pointed to by x with access permission z. The choice of the 0, 1 and 2 integers to differentiate between types of nodes is arbitrary.

$$
\begin{aligned}
&AST\ x\ (\texttt{Lam}\ v\ \beta)\ z = \exists b.(x \overset{z}{\mapsto} 0, v, b * AST\ b\ \beta\ z)\\
&AST\ x\ (\texttt{App}\ \phi\ \alpha)\ z = \exists f, a.(x \overset{z}{\mapsto} 1, f, a * AST\ f\ \phi\ z * AST\ a\ \alpha\ z\\
&AST\ x\ (\texttt{Var}\ v)\ z = x \overset{z}{\mapsto} 2, v
\end{aligned}
$$

The algorithm is presented in the snippet below. The copy function is specified by the triple $\{AST\ y\ \tau\ z\}$ `x := copy y` $\{AST\ y\ \tau\ z\ *\ AST\ x\ \tau\ 1\}$, while the substitution function has the specification $\{AST\ x\ \tau\ 1\ *\ AST\ y\ \tau'\ z\}$ `z := subt x y v` $\{AST\ z\ (\tau[\tau'/v]\ 1\ *\ AST\ y\ \tau'\ z)\}$.

```
subst x y v =
    if [x] = 0 then
        if [x + 1] != v then
            [x + 2] := subst [x + 2] y v
        else skip
        x
    elsif [x] = 1 then
        ([x + 1] := subst [x + 1] y v || [x + 2] := subst [x + 2] y v);
        x
    elsif [x + 1] = v then
        dispose x; dispose(x + 1);
        new(2, copy y)
    else
        x
```

The part of the proof that highlights the use of fractional permissions is the application case ([x] = 1), for which we should be able to prove the following triple:

$$
\begin{aligned}
&\{AST\ x\ (\texttt{App}\ \phi\ \alpha)\ 1 * AST\ y\ \tau'\ z\}\\
&\quad \texttt{[x+1] := subst [x+1] y v ||}\\
&\quad \texttt{[x+2] := subst [x+2] y v}\\
&\{AST\ x\ (\texttt{App}\ (\phi[\tau'/v])\ (\alpha[\tau'/v]))\ 1 * AST\ y\ \tau'\ z\}
\end{aligned}
$$

The proof requires the permission on the substituting lambda term to be split, in order for the parallel composition to proceed. That is, we need an equivalence of the form $AST\ y\ z\ (z + z') \iff AST\ y\ \tau\ z\ *\ AST\ y\ \tau\ z'$. Using the rule of consequence from Hoare logic with this equivalence and after applying the frame rule once, we arrive at the following proof obligation:

$$
\begin{aligned}
&\{x \mapsto 1, f, a * AST\ f\ \phi\ 1 * AST\ y\ \tau'\ (z/2) * AST\ a\ \alpha\ 1 * AST\ y\ \tau'\ (z/2)\}\\
&\quad \texttt{[x+1] := subst [x+1] y v ||}\\
&\quad \texttt{[x+2] := subst [x+2] y v}\\
&\{x \mapsto 1, f', a' * AST\ f'\ (\phi[\tau'/v])\ 1 * AST\ y\ \tau'\ (z/2) * AST\ a'\ (\alpha[\tau'/v])\ 1 * AST\ y\ \tau'\ (z/2)\}
\end{aligned}
$$

The proof of this triple is straightforward, but the important point to be evidenced is that, because the proof uses fractions, we do not need to keep track of how many times each permission has been split. Indeed, the algorithm is perfectly symmetrical, and each permission can be split up further, as many times as needed. This showcases the clear advantages in elegance that the fractional permissions model possesses over the counting permissions one in this particular example, and illustrates how counting can suit some problems (eg. ones involving an authority keeping track of the number of permissions given), whilst fractions can suit others, such as divide and conquer algorithms and embarassingly parallelisable data structures.

2.5 Gillian

Gillian [10, 11, 12, 13] is a program symbolic execution analysis tool based on separation logic, developed at Imperial College London by the Verified Software Group. Implemented in OCaml, Gillian brings to life the goal of verifying full specification of industry scale production code, as was the case with the AWS SDK mentioned in Section 1.1. Unlike other similar tools, Gillian is parametric in the memory model of the target language, meaning in order to instantiate Gillian to a new language, all a developer has to do is:

- Provide a *concrete* and *symbolic* memory model for the target language of choice
- Implement a compiler from the target language to Gillian's intermediate language, GIL.

So far, as mentioned before, Gillian has been instantiated to three target languages: C, JavaScript, and WISL (While for Separation Logic), and ongoing efforts are in place for an instantiation to Rust. Section Section 2.5.1 restricts our attention to WISL, since it the choice of target language for our project.

There are three modes of operation that Gillian supports:

- whole-program symbolic testing, in which programmers write tests where they explicitly declare certain variables as symbolic and use first-order logic assertions that they want to be verified. Gillian then attempts to find program traces that invalidate those assertions and report such traces and their constraints to the programmer.
- verification, where programmers annotate their functions with *separation logic* assertions, loop invariants, pre-conditions and post-conditions. In turn, Gillian verifies if these assertions are satisfied by the body of the function.
- bi-abduction, in which the programmer provides a program with no annotations or unit tests, and Gillian infers separation-logic assertions that the program satisfies, up to a certain bound. If bugs are found in this process, Gillian generates a bug report for the programmers to inspect.

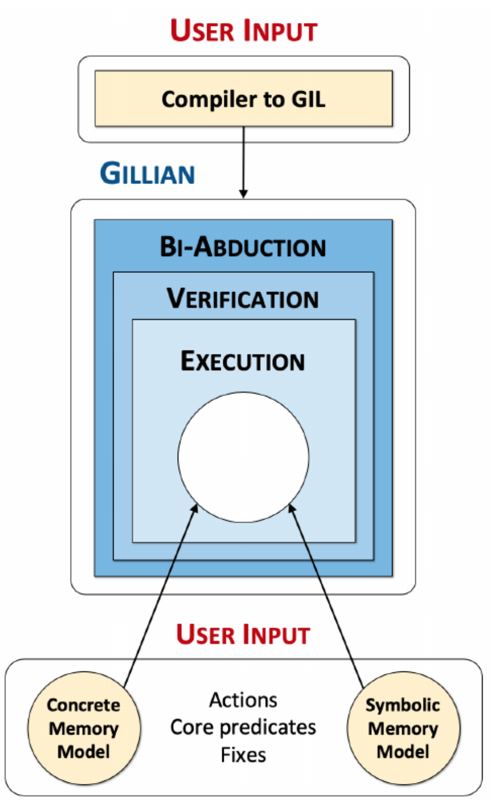


Figure 2.3: Gillian architecture. From [12].

The obvious limitation of Gillian that this project is trying to address is its ability to verify only sequential programs. As the prevalence of programs making heavy use of concurrency paradigms is increasing, the goal for Gillian is to become a more robust tool and provide a more complete verification support to programmers. We trust the choice to implement such support for WISL is apparent to the reader, given the innovative nature of the project, the simplicity of the target language and the available time for the Master's project.

2.5.1 WISL

Designed to be a simple while language for educational purposes, WISL is one of the three languages to which Gillian was instantiated. Its memory model consists of contiguous blocks, uniquely identified by a memory location, and contains two components: a list of consecutive cells mapping each offset from the source memory location to a value (the "contents of the block") and a natural number, identifying the bounds of the block.

We give an informal description of the WISL grammar below. Statements in can be one of:

- `x := E`, the assignment statement, which evaluates the expression E and assigns its result to the variable x in the variable store,
- `x := new(n)`, which allocates a new block of n contiguous cells, setting its bound to n,
- `delete(x)`, which deallocates the block of memory pointed to by x if and only if x actually points to the start of that memory block and the bounds are known,
- `x := [E]`, the reading of a heap location pointed to by the value of the expression E. The result of the reading operation is stored inside variable x,
- `[E1] := E2`, which writes the result of evaluating expression E2 to the heap cell pointed to by E1,
- `skip`, which is the no-op of WISL, reducing just to itself,
- `while (b) S`, which executes statement S as long as condition b is true (just like while loops in a regular iterative language)
- `if (b) S1 else S2`, the conditional statement, which executes statement S1 if and only if the condition b evaluates to true, otherwise executes statement S2.

To illustrate the capabilities of WISL, we include below some short code snippets of examples of WISL programs from inside the Gillian repository.

```
function llen(x) {
    if (x = null) {
        n := 0
    } else {
        t := [x+1];
        n := llen(t);
        n := n + 1
    };
    return n
}
```

Listing 2.1: List length in WISL

```
function tree_dispose(x) {
    if (x != null) {
      y := [x+1];
      z := [x+2];
      u := tree_dispose(y);
      u := tree_dispose(z);
      delete(x)
    } else {
      skip
    };
    return null
}
```

Listing 2.2: Binary tree dispose in WISL

Chapter 3

The Current State of Gillian

Adding concurrency to Gillian did not require reworking the entire codebase from the core. The onion-like structure of the codebase, due to the parameterisation in terms of the memory model, meant that most of the work was centered around the wisl instantiation and kept local: extending the language with a parallel composition operator, adding floating point arithmetics to operate with fractional permissions, rewriting the memory actions to make correct use of the permissions etc. However, the fact that concurrency is a completely new concept to the entirety of the Gillian infrastructure meant that the core part of Gillian (working on the GIL intermediate language) did not have support for concurrency, so new concepts had to be blended into the quite sizeable already existing legacy codebase. Henceworth, an overview of the architecture of the tool from a software engineering point of view should provide the reader a good understanding of the complexity of the task at hand, and should make the engineering decisions and approach taken during implementation apparent and natural. It was the complexity of the codebase, coupled with my unfamiliarity with the Ocaml language, in which Gillian is written, that made the initial part of the project particularly challenging, having spent around two weeks just scanning the code, gaining a reasonable level of understanding of its structure and the interactions happening, taking notes and making plans for how the new implementation would be written, what parts of the code had to be modified or extended and how we could employ the best software engineering practices, to make the code flexible for any future additions.

3.1 Folder Structure

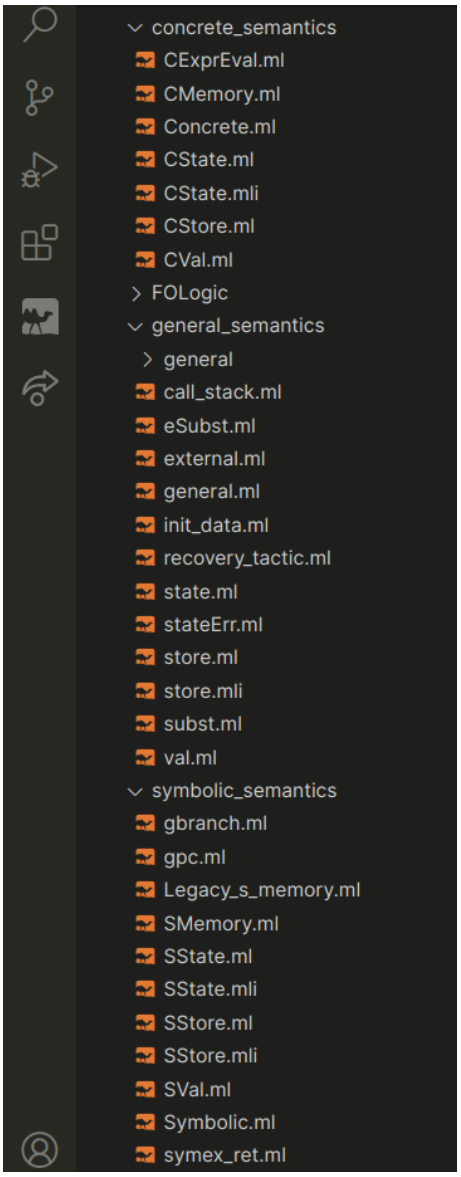
Gillian has quite a simple folder structure for a tool of its complexity, which lies rather in the level of abstraction at which it operates internally. Under the root folder, there is `/GillianCore`, which contains the core functionality of the tool, running at the level of GIL code; the three folders corresponding to the three instantiations of Gillian: `/Gillian-C`, `/Gillian-JS` and `/wisl`, the `/debugger-vscode-extension` folder which contains the code of the Gillian debugger, developed by Nat Karmios in a previous Master's project, and a number of other folders used for automated testing and CI/CD integration. We will restrict our attention to the `/Gillian-Core` and the `/wisl` directories.

3.2 Gillian-Core

`/Gillian-Core` contains the engine that powers the verification process of our tool. Whether it runs whole-program symbolic testing, verification or bi-abduction, the `/engine` sub-directory is responsible for analysing the program by keeping and updating the state through symbolic execution, consuming preconditions of the functions being verified and unifying against the expected post-conditions. In order for the tool to be parametric in the memory model of the target language, all of this work is performed on the GIL intermediate language, so a GIL parser is invoked to build an abstract syntax tree (henceforth AST) before symbolic execution begins. Adding concurrency capabilities to Gillian meant adding a language construct to GIL that can express the semantics of concurrent execution. Therefore, the parallel composition operator was first added to the syntax of the language (under `/GIL_Syntax`) and then interpreted appropriately by the engine.

3.2.1 General, Concrete and Symbolic Semantics

The engine is, perhaps unintuitively, a very high-level piece of code. Since we have multiple modes of execution, the semantics of the target language had to be generalised into a common interface that can be used by each one of them, depending on the needs. As a result, the `general_semantics` directory includes the common interfaces for the memory state and the variable store, and pieces of functionality that are common across all modes of execution (`call stack`, responsible for recording function calls, `substitution`, used for replacing variables inside logical assertions when unifying against the program state, `val`, which contains an interface of utility functions used for manipulation of GIL values, `g_interpreter`, under `general`, omitted here for brevity, which implements the abstract interpreter for the GIL language), or that is too tied to the engine itself to be extracted in a separate package (`recovery_tactic`, used in bi-abduction). The interfaces defined by the general semantics are then particularised by their counter-parts for concrete and symbolic semantics. However, there are two interfaces that are specific to the type of semantics and which are left unimplemented: the concrete and symbolic memories. In fact, these are used by their respective state implementations to execute memory actions. To instantiate Gillian to a target language, the developer would then have to implement both of these interfaces for the memory model of their target language of choice.



```
⌄ concrete_semantics
   CExprEval.ml
   CMemory.ml
   Concrete.ml
   CState.ml
   CState.mli
   CStore.ml
   CVal.ml
› FOLogic
⌄ general_semantics
   › general
   call_stack.ml
   eSubst.ml
   external.ml
   general.ml
   init_data.ml
   recovery_tactic.ml
   state.ml
   stateErr.ml
   store.ml
   store.mli
   subst.ml
   val.ml
⌄ symbolic_semantics
   gbranch.ml
   gpc.ml
   Legacy_s_memory.ml
   SMemory.ml
   SState.ml
   SState.mli
   SStore.ml
   SStore.mli
   SVal.ml
   Symbolic.ml
   symex_ret.ml
```

Figure 3.1: General, concrete and symbolic semantics file structure.

The way in which the general semantics interfaces are implemented by their concrete and symbolic counterparts is comparable to how an interface is implemented by a concrete class in a traditional object-oriented language. OCaml uses modules and signatures to capture those concepts. A module type S is most often a signature consisting of function interfaces and nested signatures. A module can then have type S, which places an obligation on the programmer to implement the functions specified in signature S. Like any class we could see in Java, Kotlin or any other object-oriented language, a module can be instantiated by providing instances of its dependencies, in the same way as calling a constructor method of a class, passing the required arguments. As an example, Listing A.1 and Listing A.2 highlight the interface (signature) `State`, its extension for symbolic states (`SState.ml`) and the module implementing that interface, intuitively named `Make`. We observe the dependency on a module of type `SMemory.S`, meaning in order to build a symbolic state, we require a *symbolic memory model* (i.e. a module that implements the `SMemory.S` interface. This is used inside the `execute_action` function, which calls into the implementation of the symbolic memory model, passing along the symbolic heap, path constraint and arguments. In fact, the implementation of this idea is exactly what makes Gillian parametric in the memory model, as we shall see later.

There is one very important part of the engine that we have not yet mentioned, and can be found under the `Abstraction` directory. It contains infrastructure that is used particularly for *verification*, which is exactly the mode of execution this project is targeting. The verification engine uses a unifier, which is responsible for unifying a separation logic assertion against the program state. The predicate state (`PState`) is another implementation of the state signature, which, as opposed to the other two, is designed for predicate *abstraction* (handling of user defined predicates) *compositionality* (handling of core predicates such as a cell assertion and the unification process, through the unifier). These are two very important properties that differentiate verification from whole-program symbolic testing, since in the latter all function calls are inlined and executed symbolically without specification, with the purpose of invalidating assertions. In verification, we are concerned with checking a function's behaviour against its specification, written in terms of separation logic assertions. The program state is the place where custom functionality will be added for handling the parallel composition operator, by verifying the specifications of all the function calls inside the `par` block. This will very much mirror the `run_spec` function, whose signature is given in Listing A.43, which unifies the current state against the function's pre-condition, and produces the post-condition in the state framed off by the unifier. The challenge, however, will be to perform the unification for multiple function calls and maintain the concurrency semantics (see Section 4.3).

Finally, bi-abduction uses its own instance of the state signature interface, called `BiState`. We will not talk further about bi-abduction, as the details of its working mechanisms fall outside the scope of this project.

3.2.2 The GIL language

GIL is the intermediate language on which the core part of Gillian functions. Every target language program gets compiled to an equivalent version of that program in GIL. The GIL code then gets parsed and mapped to an AST structure, which is then interpreted by the symbolic execution engine.

GIL is meant to be a lower level language than any target language we might instantiate Gillian to, and for a very obvious reason. As developers of Gillian, we would want the tool to be capable of verifying programs in as many target languages as possible, all of them with different particularities and characteristics. For example, C has pointer arithmetics, which we cannot find in JavaScript. C is a statically typed language, whereas JavaScript is dynamically typed. WISL itself allows direct accesses on any heap memory location that has been allocated, similar to pointer dereference in C, but with no correspondent in JavaScript. In order to maintain modularity, we would like the core engine of Gillian to operate independently of what the target language is. As a solution, we compile each target language to a less complex, more expressive language, somewhat similar to assembly, capable of translating each describing each one of these particular idioms.

Another key requirement that GIL has to serve is the capability to run logic commands. As function specifications are written in terms of separation logic assertions, these have to be expressed at the level of the core language as well, in order to be unified against the state. In particular, complex predicates might need to be unfolded before they can be used directly to check the validity of executing a command in a particular context. For example, the list predicate below has two clauses, one where variable x is null, corresponding to the empty list case, and one where variable x holds a memory location pointing to the first element of the list and the pointer to the next, recursively applying the list predicate. In symbolic execution, such a predicate would have to be unfolded and both cases would be explored independently.

```
predicate list(+x, alpha) {
    (x == null) * (alpha == nil);
    (x -> #v, #z) * list(#z, #beta) * (alpha == #v::#beta)
}
```

Listing 3.1: List predicate in WISL

Conversely, the symbolic execution engine needs to be able to branch on logical formulae, exploring both the true and the false cases of a formula, in the same way that an *if-then-else* construct would be explored. Indeed, both of these cases are treated as `Logic` commands in GIL.

Finally, it is necessary to assume facts about the state of the program or the type of a variable, and to assert that a condition holds at any moment of time. Listing 3.2 shows the full definition of the logic command type. Folding and unfolding of predicates, loop invariants, lemmas and variable binding (`SepAssert`) are dealt with using an `SLCmd.t` node (described in Listing 3.3) inside the `SL` constructor. The `FreshSVar` command creates a new symbolic variable, and the `Macro` command is analogous to C-like macros, and are of no further interest to this project.

```
type t = TypeDef__.lcmd =
  | If of Expr.t * t list * t list  (** If-then-else *)
  | Branch of Formula.t  (** branching on a FO formula *)
  | Macro of string * Expr.t list  (** Macro *)
  | Assert of Formula.t  (** Assert *)
  | Assume of Formula.t  (** Assume *)
  | AssumeType of Expr.t * Type.t  (** Assume Type *)
  | FreshSVar of string  (** x := fresh_svar() *)
  | SL of SLCmd.t
```

Listing 3.2: GIL logic commands

```
type t = TypeDef__.slcmd =
  | Fold of string * Expr.t list * folding_info option  (** Fold *)
  | Unfold of string * Expr.t list * unfold_info option * bool (** Unfold *)
  | GUnfold of string  (** Global Unfold *)
  | ApplyLem of string * Expr.t list * string list  (** Apply lemma *)
  | SepAssert of Asrt.t * string list  (** Assert *)
  | Invariant of Asrt.t * string list  (** Invariant *)
  | SymbExec
```

Listing 3.3: GIL Separation Logic (SL) commands

Perhaps the most important type of command is the local action (`LAction`). It is a command that, when interpreted, calls the state to evaluate a certain action against it. What the action is depends on the target language, therefore the programmer can manipulate the way in which their target language is meant to be evaluated by defining types of local actions (i.e. define the memory model of the language). In WISL, as we shall see later, these actions refer to the manipulation of heap cells (get/set/remove), their allocation and disposal and their bounds.

Listing A.3 describes the type of GIL commands. This is the type we will need to extend with the parallel composition operator to be able to interpret concurrent programs.

3.2.3 The GIL interpreter

The GIL interpreter is the central part of the core engine, responsible for walking the AST and exploring all paths of the program through symbolic execution. Due to the requirement to support multiple modes of execution (whole-program symbolic testing, verification and bi-abduction), it is a heavily customised piece of code. The particularisation depending on the specific mode of execution is performed not through the interpreter, but through the state. Thus, we use the symbolic state for whole program symbolic testing, the predicate state for verification, and the bi-state for bi-abduction, as mentioned in Section 3.2.1.

The part of interest in the interpreter is the `Evaluate_cmd` module, with an evaluation function for each type of command. This is where we will add our `eval_par_call` method, which will handle the evaluation of the new parallel composition operator. Function `f` then pattern matches on each command based on its type and calls the appropriate handler. Below is a (simplified) code snippet, to illustrate the idea.

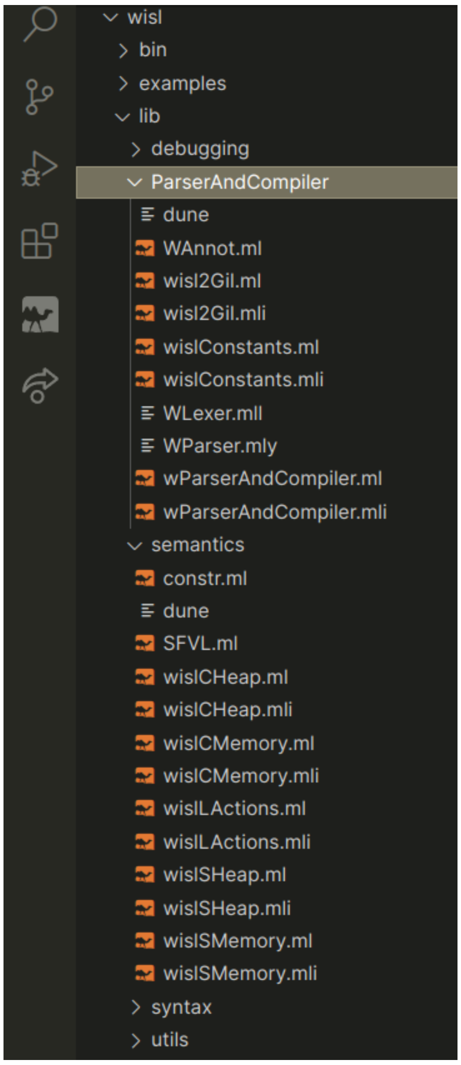
```
module Evaluate_cmd = struct
    let eval_assignment x e eval_state = ...
    let eval_laction ~branch_path x a es eval_state
    (* TODO: let eval_par_call functions eval_state = ... *)

    let f (cmd : int Cmd.t) eval_state =
    match cmd with
        | Assignment (x, e) -> eval_assignment x e eval_state
        | LAction (x, a, es) -> eval_laction ~branch_path x a es eval_state
        (* TODO: | Par (functions) -> eval_par_call functions eval_state *)
end
```

Listing 3.4: Function f, pattern matching on the given command

3.3 The WISL memory model

The WISL instantiation is found under its own folder called `wisl`, which contains the syntax of the language, a WISL parser (written using the Menhir parser generator library for OCaml) and a compiler to GIL, and the semantics of the language, implementing the `SMemory` signature interface mentioned in Section 3.2.1. This is where the majority of the added code will find itself. Although not reworked from scratch, a considerable percentage of the code here had to be highly adapted and extended, as WISL had a number of limitations that had to be addressed in order to support concurrency and the fractional permissions model:

- The local actions used for loading from and storing to memory were the same as the producers and consumers (see Section 3.3.1) used to manipulate logic assertions.
- The current model of heap cells did not support permissions (naturally, as there was no need for them in purely sequential code).
- A boolean flag specifying the expected behaviour during unification was not propagated through the execution of some actions, which caused problems when evaluating a parallel composition and checking against a post-condition (variables were identified to be equal as part of the path constraint, but this was not used when unifying the assertions, causing unsound results).
- There was no support for floating point arithmetics, which was crucial, as permissions were numbers between 0 and 1.



```
wisl
  bin
  examples
  lib
    debugging
    ParserAndCompiler
      dune
      WAnnot.ml
      wisl2Gil.ml
      wisl2Gil.mli
      wislConstants.ml
      wislConstants.mli
      WLexer.mll
      WParser.mly
      wParserAndCompiler.ml
      wParserAndCompiler.mli
    semantics
      constr.ml
      dune
      SFVL.ml
      wislCHeap.ml
      wislCHeap.mli
      wislCMemory.ml
      wislCMemory.mli
      wislLActions.ml
      wislLActions.mli
      wislSHeap.ml
      wislSHeap.mli
      wislSMemory.ml
      wislSMemory.mli
    syntax
    utils
```

Figure 3.2: The wisl instantiation file structure

Each one of these limitations was addressed in turn, with the goal in mind of keeping the code maintainable and flexible and applying good software engineering practices.

3.3.1 Producers and Consumers

As mentioned before, the essence of function verification in Gillian is symbolically executing the function body assuming its pre-condition and, under all explored paths, reaching a state satisfying the post-condition. This goal is achieved in Gillian through an infrastructure based on producers and consumers. A producer of a logical assertion is a local action that will update the abstract state in such a way that said assertion holds. A consumer removes (frames-off) the bits of the state that form the logical assertions. These producers and consumers are what constitutes WISL's memory model and powers the verification of WISL programs.

Each core predicate of a target language (including WISL) has a corresponding producer and consumer. Luckily, WISL has only two such core predicates:

- The cell predicate, corresponding directly to a cell assertion in separation logic. `cell (loc, ofs, val)` is equivalent to the separation logic assertion $(loc + ofs) \mapsto val$, meaning *at memory location loc there is a block, and inside that block, at offset ofs, is stored a value val.*
- The bound predicate, `bound (loc, b)`, which says that *the block at location loc has bound b (possibly None).*

These two core predicates are compiled to GIL as *GA* assertions (GilAssertions), which are the norm for representing core predicates in any target language. These assertions have *in* parameters, which are the parameters that are known when calling the predicate (marked with a + in WISL), and *out* parameters, which are the ones we learn from the ins by unfolding the predicate. In Listing 3.1, x is the in parameter and alpha is the out parameter we learn (it is the empty list if x is null, and a list with the head v, when x points to v). Listing A.5 shows how the cell and bound assertions are created. This code will need to be adapted to support permission argument for both types of assertion. We leave the question of whether this new parameter will be an in or an out unanswered for the time being.

Blocks and the SFVL

Blocks are represented by a very simple data type, allowing *freed* and *allocated* blocks, that are made of a *symbolic field-value list*, or **SFVL** (similar to a contiguous array in a language like C, except indexes and values are symbolic expressions) and an optional integer bound. The block type and value type of the SFVL will have to be extended with a field to model permissions of the bound assertion (in the case of the block) and of the cell assertion (for the SFVL).

```
  type t = Freed | Allocated of { data : SFVL.t; bound : int option }
```

Listing 3.5: The type of blocks, pre-project

```
(* SFVL.ml *)
type field_name = Expr.t
type field_value = Expr.t
```

Listing 3.6: The key and value types of SFVL, pre-project

The SFVL is a very important data structure, because it contains all the cells that have been allocated in that block, and their corresponding values. In order to access a cell in the heap, one must first retrieve the block at the specified location, then check that the offset is in bounds and index the SFVL structure at that offset to retrieve the symbolic value. Of course, during symbolic execution, we might learn information about our symbolic variables that might impact the result of insertion and retrieval operations into the SFVL. For this reason, the interface offers functions that take equality tests, in order to retrieve / add the value at a key that is *equal* (as far as the path constraint can determine) to the one passed by parameter (see Listing A.6).

Finally, during unification, it is possible that memory locations that were assumed to be different (and thus hold different blocks in the heap) have now been discovered to be equal. If this happens, the two blocks must be merged. In the current state of WISL, merging to SFVL blocks is equivalent to choosing one of them, as they are guaranteed to be identical if their locations coincide (see Listing A.7 for the implementation). However, this is no longer true when dealing with permissions, as we shall see later.

Local Actions

Throughout the project, a constant fact of life that we had to work around, as in any software engineering project, was working with legacy code. Adapting already existing code required deep understanding of its mechanism, and, while this was needed and achieved through a combination of mental analysis of the code and repeated trial and error, some particular details had to be learned and accepted, as changing them would have diverted considerably from the ultimate goal of the project. As a result, understanding the engineering aspects behind how the producers and consumers worked were not only the starting point of the project, but also the source of particular bugs we encountered along the way.

The WISL state consumer is split between two different local actions: get, which retrieves the portion of the heap required by the assertion, and rem, which removes it from the state. As mentioned, there is a consumer for both core predicates, so we have four local actions in total: `GetCell`, `RemCell`, `GetBound` and `RemBound`. `GetCell` finds the block at the specified memory location, failing with a `MissingResource` error, if the block does not exist, or with a `UseAfterFree` error, if the block was previously freed. If the block exists, it checks for a possibly out of bounds access, which happens if the offset is larger than the bound, and in that case returns an `OutOfBounds` error. Note that this is not a validity, but a satisfiability check that is delegated to Gillian's SAT solver, Z3. Since Gillian is over-approximating, we must fail if the access *may* be out of bounds (if there is at least one symbolic execution trace leading to such an access). If all these checks succeed, `GetCell` accesses the block and returns the location, offset and value read from the heap, or a `MissingResource` error, if the cell is not there. `RemCell` also checks for missing-resource and use-after-free errors, and if they do not happen, it removes the cell at the specified offset and returns unit (note there is no need to check for an out-of-bounds access, as the check was already performed by the getter). Prior to the project, `GetCell` was also used to handle loading from the heap, as the two were functionally equivalent. Listing A.8 and Listing A.9 show the handlers for these local actions before the project. These are part of the `wislSHeap.ml` file, and they are called from the appropriate handlers inside the symbolic memory (`wislSMemory.ml`), for better modularity and isolation. One particularity of the mechanisms in which `GetCell` and `RemCell` are used that caused unfortunate bugs was that the values returned from `GetCell` **must** be the exact parameters that `RemCell` expects, in that exact order, as they are passed directly from one to the other. This invariant had to be kept along the development of the project. `GetBound` and `RemBound` behave analogously, except they do not need to checks for out-of-bounds errors, since they retrieve/remove the bounds from the block.

The producers for WISL core predicates are called `SetCell` and `SetBound`. They behave in the exact same way as their corresponding get consumers, except instead of returning the value in that specific cell (or, respectively, the bound of the specific block), they replace them with the value passed as a parameter and return unit. Similarly to `GetCell` explained above, before the project, `SetCell` was also used to handle stores to the heap, due to their equivalence with the cell assertion producers. The code for `SetCell` prior to the project can be found in Listing A.10.

The core part of the project was extending these consumers and producers with fractional permissions by first coming up with a correct semantics that would describe them in this context and then implementing these semantics in Gillian. Separate local actions needed to be written from scratch for load and store operations, as their semantics in the context of permissions differ those of from state consumers and producers (see Section 4.1).

3.3.2 WISL to GIL compiler

The compiler from WISL to the intermediate language GIL is found in the `wisl2gil.ml` file. It is merely a collection of functions compiling each type of WISL AST node in turn (expressions, logical expressions, statements, loop invariants etc.). Special attention had to be given to the compilation of statements: firstly, when reworking the local actions performed for loading and storing and for adding permissions support for the "points to" assertion, including floating point arithmetics, and secondly, when adding the parallel composition operator. These changes were mostly straightforward, as the AST nodes would most often have direct correspondents in GIL (for par and floating point operators), and the heavy lifting of permissions semantics was done inside the symbolic memory. Short simplifications of the the logic that had to be extended for compiling binary operators and statements can be found in Listing A.11 and Listing A.12.

3.4 Summary

This section has provided a quick overview of the current state of Gillian's architecture. We learned that Gillian is made of a core engine, responsible for running three modes of execution (whole-program symbolic testing, verification and bi-abduction), and each mode of execution uses a specific implementation of the `State` interface. We saw how the Gillian interpreter handles each type of GIL command and calls into the state passed to its module to perform side-effects, which in turn calls into the implementation of the memory model (in our case `wislSMemory`). Furthermore, we learned that the predicate state (`PState`) is used for abstraction (handling of user defined predicates) and compositionality (handling of core predicates), which is what verification of a function's pre- and post-condition is concerned with, and will therefore be the place to extend with the implementation of the parallel composition operator. Finally, we understood how blocks of data are represented, became familiar with producers and consumers of core assertions and shortly explored the compilation of WISL nodes to GIL commands. The next chapter will use the expertise summarised here and will fill in the gaps left in our current presentation, by implementing support for shared memory concurrency in the form of fractional permissions.

Chapter 4

Implementing Fractional Permissions

This chapter will explore the bulk of the work undergone during this Master's project, which lead to the achievement of building a verification tool capable of supporting shared memory concurrency. We will explore all the steps we took throughout the time we had available, all the implementation decisions that were made and their consequences, and bugs we had to overcome. We shall discuss the relevant parts of the architecture that were modified, first from a theoretical point of view, in order to understand the semantics of the new, extended WISL language, WISLFP and its memory model, and then the technical details of implementing the theory of fractional permissions within the tool.

Before we dive deep into the details of how WISLFP works, it is worth mentioning that the new memory model was built entirely on the skeleton of the old one. In fact, under the root directory of the Gillian codebase, there is now a folder called `/wisflp`, which was initially just a copy of the original `/wisl` folder. As a result, a lot of code will be duplicated between the two packages. The intent was for the new WISLFP implementation to replace the old WISL, but, for administratory reasons, the Gillian development team decided to postpone this replacement and have both variants of the language available in parallel.

4.1 Producers and Consumers

Section 3.3.1 introduced producers and consumers of core assertions in WISL and explained their working mechanism. Now it is time to explore their new semantics in the context of fractional permissions.

4.1.1 Theory

From a theoretical point of view, we can encompass the semantics of the new producers and consumers with fractional permissions using Hoare triples. We will respect the notation of separation logic cell assertions for their producers and consumers, with the intention that the reader understands the absence of the offset variable in the `GetCell`, `RemCell` and `SetCell` as a way to simplify the notation. We will, however, be loose with the expression of bound assertions, since they have no correspondent in separation logic. We will also, perhaphs unintuitively, provide derivation rules in erroneous cases. This is to highlight the type of error that the memory model would throw in those specific circumstances. We consider the benefit of completeness and preciseness of the theoretical presentation before showcasing the implementation to outweigh the cost of a less standard approach of representing errors as if they were valid heap states. We trust the reader understands that, when such an error occurs in practice, there can be no talking of a valid heap state, and execution would instead stop immediately.

As consumers in Gillian are split between *get* and *rem* operations, we treat each one individually. `Getcell` takes as parameters the location in the heap we want to access (in practice, accompanied by an offset) and the permission associated with the predicate it consumes (we call this the *outer permission*). It has actually no effect on the state of the heap in the successful case. It exists merely as a way to check for `MissingResource` (in the case the cell does not exist), `UseAfterFree` and `OutOfBounds` errors, and to verify that the outer permission is always at most equal to the permission inside the heap (henceforth we will call this the *inner permission*). We will delay the presentation of the first three erroneous cases for Section 4.1.2, as they are more reliant on the internal way of representing cells in Gillian, and there is no real value in expressing them using Hoare triples. The successful case is actually just a logical equivalence between two heap states, while the failure case throws a `MissingResource` error, with the missing permission as an argument. It is worth mentioning that, in all future derivation rules, $l \overset{0}{\mapsto} v$ is equivalent to `emp`.

$$\frac{q1 \geq 0}{\{l \xmapsto{q1+q2} v\}\, \texttt{getcell(l, q2)}\, \{l \xmapsto{q1} v * l \xmapsto{q2} v\}} \quad \text{(could also be written as an equivalence)}$$

$$\frac{q2 > q1}{\{l \xmapsto{q1} v\}\, \texttt{getcell(l, q2)}\, \{\texttt{Error(MissingResource(q2 - q1))}\}}$$

The second part of the consumer is `RemCell`. Semantically, its job is to remove the outer permission of the cell from the heap, leaving the remaining permission there, if any (as before, we treat a cell with 0 permission as the empty heap, and we will see the implementation does, too). Note the way in which the heap in the pre-condition has been split, as per the equivalence in the successful case of `GetCell`.

$$\frac{q1 \geq 0}{\{l \xmapsto{q1} v * l \xmapsto{q2} v\}\, \texttt{remcell(l, q2)}\, \{l \xmapsto{q1} v\}}$$

The producer for the cell assertion, `SetCell`, works by adding the cell with its value and the outer permission in the heap, if the cell was not already there, or if it was, add the outer and the inner permissions together. In addition, we *learn* that the value in the heap and the one passed to the producer are equal, and that the two permissions sum to a number at most equal to 1. The use of the word learn will find its importance once we discuss the practical implementation.

$$\{l \xmapsto{q1} v\} \quad \texttt{setcell(l, w, q2)} \quad \{l \xmapsto{q1+q2} v * (v \doteq w) * (q1 + q2 \leq 1 \wedge \texttt{emp})\}$$
$$\{\texttt{emp}\} \quad \texttt{setcell(l, w, q)} \quad \{l \xmapsto{q} w\}$$

The consumers and producers for the bound core assertions behave in the exact same way, at the theory level. Below are the derivation rules for them. Analogous to cell assertions, bound assertions with a permission of 0 are equivalent to `emp`.

$$\frac{q1 \geq 0}{\{bound(l, b, q1 + q2)\}\, \texttt{getbound(l, q2)}\, \{bound(l, b, q1) * bound(l, b, q2)\}}$$

$$\frac{q2 \geq q1}{\{bound(l, b, q1)\}\, \texttt{getbound(l, q2)}\, \{\texttt{Error(MissingResource(q2 - q1))}\}}$$

$$\frac{q1 \geq 0}{\{bound(l, b, q1) * bound(l, b, q2)\}\, \texttt{rembound(l, q2)}\, \{bound(l, b, q1)\}}$$

$$\frac{}{\{bound(l, b1, q1)\}\, \texttt{setbound(l, b2, q2)}\, \{bound(l, b1, q1 + q2) * (b1 \doteq b2) * (q1 + q2 \leq 1 \wedge \texttt{emp})\}}$$

$$\frac{}{\{emp\}\, \texttt{setbound(l, b, q)}\, \{bound(l, b, q)\}}$$

Allocating and disposing of cells is performed in the intuitive way: newly allocated cells will have full permission, whereas disposing of a block requires full permission on that block in the heap and the bound to be known and have full permission as well (otherwise, some other thread might be reading from a cell that we disposed of, resulting in a `UseAfterFree` error. We will use a loose notation for a block of n cells, for the sake of brevity.

$$\{\texttt{emp}\}\, \texttt{x := new(n)}\, \{(x + i) \overset{1}{\mapsto} null(i \in \overline{0, n-1}) * bound(x, n, 1)\}$$

$$\{(x + i) \overset{1}{\mapsto} m_i(i \in \overline{0, n-1}) * bound(x, n, 1)\}\, \texttt{dispose(x)}\, \{\texttt{emp}\}$$

Finally, since loading and storing is no longer handled by consumers and producers of cell assertions, we need separate local actions for both of them. The semantics are showcased below. Unlike producers and consumers, loading and storing do not have an outer permission parameter, as there is no logical assertion being produced or consumed that would contain such a permission. Rather, these local actions are accessing the heap and reading from / writing to the location specified. As a result, it is the inner permission which determines whether a load / store action is valid, in the way one expects: we require full permission to write to a heap location, whereas reading can be performed with a fractional one. This also means that loading can never fail due to insufficient permissions, but only in the "traditional" way (i.e. with `MissingResource`, `UseAfterFree` or `OutOfBounds` errors, which we are not interested in from a theoretical point of view). Of course, a permission q of 0 is, as mentioned before, equivalent to the absence of the cell, so we treat that case as an instance of the `MissingResource` error in the context of a missing cell. The store, in addition to all the erroneous cases shared with load, can also fail due to insufficient permission, if the permission of the heap cell assertion is less than 1. This corresponds to the case where, presumably, another thread has acquired partial ownership of this resource and is reading from it, so writing to that same location would induce a data race and trigger undefined behaviour. This is the clear way in which concurrent separation logic with fractional permissions ensures correctness of the program, by preventing data races from ever occurring.

$$\frac{q > 0}{\{l \overset{q}{\mapsto} v\}\,\texttt{n := load(l)}\,\{l \overset{q}{\mapsto} v * (n \doteq v)\}}$$

$$\frac{}{\{l \overset{1}{\mapsto} v\}\,\texttt{store(x, w)}\,\{l \overset{1}{\mapsto} w\}}$$

$$\frac{q < 1}{\{l \overset{q}{\mapsto} v\}\,\texttt{store(x, w)}\,\{\texttt{Error(MissingResource(1 - q))}\}}$$

4.1.2 Implementation

The new WISL memory model (WISLFP) was built on the foundation of the old model, making adaptations when necessary or writing new functionality from scratch, as was the case for the load and store local actions. We will restrict our attention for now only to the implementation of memory blocks and local actions, spread across different files in the `semantics` directory inside `wislfp`, the most notable being `wislSMemory.ml` and `wislSHeap.ml`.

Blocks and the SFVL

Firstly, we needed a new way to model memory blocks, since now the bound assertion of the block and every cell the block contains have a permission associated with them. As a result, our new type for representing blocks has an optional pair of an integer parameter (the bound) and an expression parameter (the permission on the bound assertion).

```
type t =
    | Freed
    | Allocated of { data : SFVL.t; bound : (int * Expr.t) option }
```

Listing 4.1: The type of blocks, after the project

The SFVL now has to keep a value and a permission in each cell, so we changed the `field_value` type accordingly.

```
type field_name = Expr.t
type field_value = { value : Expr.t; permission : Expr.t }
```

Listing 4.2: The key and value types of SFVL, after the project

The union of two blocks corresponding to the same location now has a different semantics. In the context of permissions, when the symbolic execution engine discovers that two memory locations are equal and has to unify the memory blocks associated with them, the cells in those blocks might each hold a fraction of the full permission. In order to recreate the original heap, we cannot choose either of the two blocks anymore. Instead, we have to *add* the permissions of the cells in the two blocks and produce a block with cells holding the original value (which is still the same in both blocks) and the sum of the permissions. The algorithm can be found in Listing A.13.

To imagine this scenario happening, take function `f` written below that consumes half of the cell at a specific memory location. The notation `#x -> (0.5: #v)` means *#x points to a cell containing the value #v with a permission of 0.5.* (note that variables with names starting with a # character are logical, whereas all others are program variables). When the function call is interpreted, the consumer of the function's pre-condition will remove half the permission on that cell. As a result, the cell in the heap is left with a permission of 0.5. The function will not leak resources and thus include the same cell with the 0.5 permission in the post-condition as well. The producer of the post-condition will produce the cell assertion with the 0.5 permission, hopefully also finding the partial ownership of that cell that was left in the heap and adding them together, to re-establish the full permission.

```
{ (x == #x) * (#x -> (0.5: #v) }
function f(x) {
...
}
{ (#x -> (0.5: #v) }
```

Listing 4.3: Function `f`, using partial ownership of a resource

However, in Gillian, the unification process against the pre- or post-condition of a function is performed according to the *unification plan* of that pre- or post-condition. Unification plans are created before symbolic execution starts, so are fixed during execution. Since the unification plans for the pre- and post-conditions of the same function are independent, Gillian does not use the same logical variables to describe the same resources across them. Instead, it will create new logical variables and rely on the Unifier to apply the correct substitution to the state, once it discovers their equality. For this exact reason, `#x` will be assigned two different logical variables in the internal representation of the state, one for the pre-condition, and one for the post-condition. Observe the log below, which assigns `{{ _$l_1, 0i }}` (the object containing a pointer to memory location `_$l_1`, offset 0, also equivalent to `{{ #wisl__0, #wisl__1 }}`, as per the substitution) to `#x` in the pre-condition (hence why it is part of the original substitution), while the unification plan of the post-condition produces another `#x`, holding an object defined by 2 new logical variables: `#wisl__2` and `#wisl__3`. Notice how `_$l_1` still holds half a full permission in the heap.

```
Produce assertion: types(#wisl__2 : Obj, #wisl__3 : Int) *
                   (#v == null) *
                   <cell>(#wisl__2, #wisl__3, #p; #v) *
                   (#x == {{ #wisl__2, #wisl__3 }})
-------------------------
Produce simple assertion: types(#wisl__2 : Obj)
With subst: [ (p: 0.5),
              (x: {{ _$l_1, 0i }}),
              (#p: 0.5),
              (#v: null),
              (#wisl__0: _$l_1),
              (#wisl__1: 0i),
              (#x: {{ _$l_1, 0i }})]
           -------------------------
STATE: SPEC VARS:
STORE:
  (p: Lit 0.5)
  (x: {{ ALoc _$l_1, Lit 0i }})
```

```
MEMORY:
  _$l_1 -> BOUND: Some (1, 1.)
           {0i: null [0.5]}
```

Listing 4.4: Logs of producing post-condition of function `f`

The consequence is that a new cell holding the value `#v` with permission 0.5 will be produced at a different location, `_$l_3`. The unifier will then use the path constraint that holds the equality between the two different locations and *merge* the corresponding blocks.

```
STATE BEFORE SIMPLIFICATIONS:
SPEC VARS:
STORE:
  (gvar0: {{ ALoc _$l_1, Lit 0i }})
  (u: LVar _lvar_3)
  (x: {{ ALoc _$l_1, Lit 0i }})
MEMORY:
  _$l_3 -> BOUND: None
           {_lvar_5: null [0.5]}
  _$l_1 -> BOUND: Some (1, 1.)
           {0i: null [0.5]}
PURE FORMULAE:
  (_lvar_3 == 0i)
  (_$l_3 == _lvar_4)
  ((_$l_1 == _lvar_4) /\
  (0i == _lvar_5))
TYPING ENVIRONMENT:
  (_lvar_3: Int)
  (_lvar_4: Obj)
  (_lvar_5: Int)
  -----------------------------------
  Filtered and fixed subst, to be applied to memory:
[ (_lvar_3: 0i),
                                                    (_lvar_4: _$l_1),
                                                    (_lvar_5: 0i),
                                                    (_$l_3: _$l_1) ]
WARNING: SFVL.union: merging with field in both lists (0i: null [0.5] and null [0.5]),
adding permissions.
```

Listing 4.5: Producing cells at two different locations and merging the blocks

Once merging is performed, we end up with the state holding our cell with full permission, just as before the function call was executed.

```
STATE BEFORE SIMPLIFICATIONS:
SPEC VARS:
STORE:
  (u: LVar _lvar_3)
  (x: {{ ALoc _$l_1, Lit 0i }})
MEMORY:
  _$l_1 -> BOUND: Some (1, 1.)
           {0i: null [1.]}
PURE FORMULAE:
  (_lvar_3 == 0i)
  (_$l_1 == _lvar_4)
```

Listing 4.6: Final state

In and out parameters

In Section 3.3.1 we mentioned that we would have to add a new parameter to the cell and bound assertions constructors to represent the permission. We did not, however, specify whether this would be an in or an out parameter.

The reader might (and rightfully so) believe that the permission should be an out parameter. The thought process might be: *using the location and the offset, we retrieve the cell at that location and find both the value it holds and its permission.* Unfortunately, things are not that simple.

In the theoretical presentation of our derivation rules for producers and consumers (Section 4.1.1) we made a distinction between the permission of the resource held by the heap (the inner permission) and the one coming from a local action like `GetCell` (the outer permission). It is true that in some circumstances it would be necessary to infer the inner permission of a cell, such as when including a cell assertion as part of a predicate, which itself might be a part of a more complex specification of a function that should work with any permission parameter, and so does not require one passed as an argument. Nevertheless, producing a cell assertion requires us to know the outer permission parameter beforehand, so that it can be added to the inner permission or, if the cell does not already exist in the heap, a new cell can be created with the outer permission. Similarly, consuming a cell assertion demands that we check the owner does not "ask for too much" (i.e. the outer permission parameter is not greater than the inner permission). These use cases clearly indicate a need for the permission to act as an in-parameter.

It appears as if adding support for permissions would require us to define a new type of parameter, that can act both as an in and an out. However, this approach would have involved significant research without the guarantee of a viable solution and, in the fortunate case, deep reworking of the Gillian architecture, which was not sustainable in the time available for this project. So, how did we solve this issue?

The answer came as the need for this dual behaviour showed up, quite late in the project. Clearly, the permission needed to work as an in parameter to get even the most basic programs to verify correctly, and this was, indeed, the path we chose to follow for both cell and bound assertions (see Listing A.14). However, having permissions as out parameters only showed up during the evaluation process, when writing more complex predicates and functions that, as mentioned above, were meant to verify correctly without passing a permission explicitly. The way to do it was through *specification targets*, a feature of the core of Gillian that had almost no use case prior to this project.

When functions are compiled to the GIL intermediate language, their specifications are also compiled to their corresponding GIL syntax. However, specifications can be parameterised too. Consider the specification of the function below, which calculates the height of a binary tree.

```
spec height(t)
  <height_spec: #p>
  [[  (t == #t) * binary_tree(#t, #tree, #p)  ]]
  [[  tree_height(#t, ret, #p)  ]]
  normal
proc height(t) {
...
    lh := "height"(lt) use_subst [height_spec - #p: #p]
...
};
```

Listing 4.7: Parameterised specification, taking a parameter `#p`

This function's specification describes that logical variable `#t` is a binary tree and that the return value holds the height of that tree (assuming a correct implementation of the `binary_tree` and `tree_height` predicates). Since permissions need to work as in parameters for cell assertions, they also have to be in parameters of any predicate that uses them. But we still want to be able to call the function with any permission, since finding the height of a binary tree requires read-only access, so any non-zero permission would be suitable. Normally, we would achieve this by adding another parameter to the procedure that would hold the value of the logical variable `#p`. However, the specification target `<height_spec:  #p>` says that this specification takes `#p` as a parameter, so we can use this instead and pass this parameter to our predicates, without needing to know its true value or taking on an extra argument to the function. We get a cleaner, more elegant function from a programming point of view, and simpler specifications that manipulate fewer variables. The function calls itself recursively, to showcase the syntax of instantiating its specification.

However, we are still not saved from dealing with what is, after all, purely logical boilerplate for the verification tool, which the programmer might have no willingness or time to learn about. The long term goal is for Gillian to be able to *infer* the correct value for `#p` during the compilation process, so that all calls to the `height` procedure can be made with the right amount of permission to instantiate its specification (which will always be either full permission, `#p = 1.0` or anything less than that, `0.0 < #p < 1.0`). We leave this as opportunity for future development, as the time constraints of the project did not allow for the research process needed to explore this idea. What we provide, however, is an extension to the WISL language that compiles exactly to the GIL code above. In our new WISLFP, we can write the height function with the specification parameter needed for it to be used more flexibly, without needing separate predicates that specify a permission parameter suitable for read-only access.

```
[ spec height_spec: #p ]
{ (t == #t) *  binary_tree(#t, #tree, #p) }
function height(t) {
...
}
{ tree_height(#t, ret, #p) }
```

Listing 4.8: Parameterised specification in WISLFP

Local Actions

The implementation of the local actions closely follows the rules described in the Section 4.1.1. For the sake of brevity, we will only explicitly include listings where important behaviour not deducible from the theoretical description is present.

We will start with the consumers. These have been smartly engineered and separated into multiple functions that perform functionality common across different local actions. For example, `GetCell` simply creates the permission check for the cell as a separate function which, given a parameter q (which would be the inner permission), checks that the expression `q < out_perm` is unsatisfiable. If this is not the case, it returns the missing permission required for the action to succeed. The `.` after the operators signifies that we are using the floating point equivalents of these operators. This uncovered another previously unknown bug, in that some of the comparison operators (such as `<=.`) were being compiled to the their integer GIL counterparts, instead of the float ones. After these checks are done, `GetCell` just calls into a separate higher-order function, called `access_cell`, which is shared with the `Load` local action, passing in the permission check as a parameter (see Listing A.15 for the implementation of `Load`).

```
let get_cell ~unification ~pfs ~gamma heap loc ofs out_perm =
  let fl q = Formula.Infix.(q#<.out_perm) in
  let permission_check q =
    if Solver.check_satisfiability ~unification (fl q :: PFS.to_list pfs) gamma
    then Some Expr.Infix.(out_perm -. q)
    else None
  in access_cell ~unification ~pfs ~gamma heap loc ofs permission_check
```

Listing 4.9: Implementation of `GetCell`, after the project

The `access_cell` function is the one that performs all the heavy-lifting common for the `GetCell` and `Load` actions. It firsts finds the block pointed to by the location passed as a parameter. If no such block is found, we return a `MissingResource` error with the missing cell. If the block exists, but has been freed, we have encountered a `UseAfterFree` error. Otherwise, the block must be allocated, so we have a data SFVL and a bound. We then perform the bounds check: if the bound is non-existent, we cannot have an `OutOfBounds` error, but if the bound exists, we check that the offset is not greater than the bound, returning the error in case this is satisfiable (remember, all sanity checks have to be satisfiability checks, since we are operating in an over-approximating setting). If everything worked until now, we index the SFVL data structure and retrieve the corresponding cell, through the `check_sfvl` function, described in Listing A.16. Here we will only define what we shall do in case the retrieval returns a `None` or a `Some`. If we get a `None`, it means the cell is not present, so again, we return a `MissingResource` error. Otherwise, the `check_sfvl` function will call the success case passing in the offset (which might be a different variable from our given offset, but should always be equal to it, due to unification), the value read from the cell and the inner permission. The success case only takes the inner permission and performs the permission check that it has been passed as a parameter, returning a `MissingResource` error with the missing permission, in case of failure, as per the theoretical semantics, or the location, offset and value (mind the order in which they are returned!), in case of success, which is equivalent to splitting the cell in two, but not framing it off yet, as described in Section 4.1.1. We can easily see how this function can handle both the `GetCell` local action, when the permission check is a meaningful one, and the `Load`, when the permission check will be a dummy function always returning `None` (and the load will then discard the location and offset received from `access_cell` and just return the value).

```
let access_cell ~unification ~pfs ~gamma heap loc ofs permission_check =
  match Hashtbl.find_opt heap loc with
  | None -> Error (MissingResource (Cell, loc, Some ofs))
  | Some Block.Freed -> Error (UseAfterFree loc)
  | Some (Allocated { data; bound }) ->
    let open Syntaxes.Result in let* () = match bound with
      | None -> Ok ()
      | Some (n, _) -> let expr_n = Expr.int n in
          let open Formula.Infix in
          if Solver.sat ~unification ~pfs ~gamma expr_n #<= ofs then
            Error (OutOfBounds (Some n, loc, ofs))
          else Ok ()
    in let none_case () = Error (MissingResource (Cell, loc, Some ofs)) in
    let success_case ofs value permission = match permission_check permission with
      | Some missing_perm -> Error (MissingResource (Cell, loc, Some missing_perm))
      | None -> Ok (loc, ofs, value)
    in check_sfvl ~pfs ~gamma ofs data none_case success_case
```

Listing 4.10: Implementation of `RemCell`, after the project

In case of `Load`, this is the end of the story. The value is returned and handled by the assignment in the GIL interpreter. However, as it stands, due to legacy implementation, the consumer has a second half, called `RemCell`, which takes the parameters **in the exact order** they are returned from `GetCell` (this was the source of painful debugging which led to the discovery that the two producers did not agree on the order in which the parameters were passed). `RemCell` performs the same sanity checks as `GetCell` (as it has before the project) and, in the case the block is allocated, we first retrieve the cell from the block (note, there is no need to check that the cell exists anymore or that we have enough permission, since this was done by `GetCell`), and then calculate the new permission by subtracting the outer permission from the inner. If the resulting permission is 0, we completely remove the cell from the heap, otherwise we add the cell with the new permission back into the block. Finally, we return an `Ok()` to the interpreter, signalling the action was successful and there is no interesting information to return (since its whole purpose was to remove the cell from the heap). The implementations of `GetBound` and `SetBound` are analogous to their cell counterparts, and thus we leave them out of discussion for brevity. The interested reader can find their implementations in Listing A.17 and Listing A.18.

```
let rem_cell ~pfs ~gamma heap loc offset out_perm =
  match Hashtbl.find_opt heap loc with
  | None -> Error (MissingResource (Cell, loc, Some offset))
  | Some Block.Freed -> Error (UseAfterFree loc)
  | Some (Allocated { data; bound }) -> (
    match SFVL.get offset data with
    | None -> (* Should not happen if get_cell is correct *)
        failwith "Called rem_cell with an offset that is not in the data SFVL!"
    | Some SFVL.{ value; permission } ->
      let data = SFVL.remove offset data in
      let new_perm = Expr.Infix.(permission -. out_perm) in
      let data =
        if Solver.is_equal ~pfs ~gamma new_perm (Expr.num 0.0) then data
        else SFVL.add offset SFVL.{ value; permission = new_perm } data
      in let () = Hashtbl.replace heap loc (Allocated { data; bound }) in Ok ())
```

Listing 4.11: Implementation of `RemCell`, after the project

We move on to the producers. Similarly to what we did with consumers, we extracted common functionality of `SetCell` and `Store` into a separate higher-order function, called `overwrite_cell`, which the reader can find in Listing A.19. What it does is it first tries to find the block at the specified location. In case it fails, we call a `missing_block` procedure, which will be specialised by the two usecases: in the case of `SetCell`, we have to create this block from scratch, so we create the SFVL with the given value and outer permission and add them to the heap at the specified location, whereas for `Store`, we have to return a `MissingResource` error. If the block is freed, we return a `UseAfterFree` error, as per usual. If we find an allocated block, we perform a bounds check (in case the bound exists) and execute an `in_bounds` procedure passed to us as a parameter, which should define the way the function behaves if the bound check succeeds. Again, this is particularised by the two usecases. `SetCell`, given below, checks the data block and, in case the cell is not found, it extends the block to contain this new cell (the details of the `extend_block` function can be found in Listing A.20). In the case it is found, we have to *learn* the equality of the value inside the heap and the one passed as a parameter and add the outer and the inner permissions, also learning that their sum is less than the full permission, so we return the list of learned formulas as part of the `Ok` result. In the case of storing (see Listing A.21), the none case will lead to a `MissingResource` error (since a missing cell translates to an illegal access), while the permission check has to verify that the inner permission of that cell is not less than the full permission, in order for the store to be valid.

```
let set_cell ~unification ~pfs ~gamma heap loc_name ofs v out_perm =
  let block_missing () =
    let data =
      SFVL.add ofs SFVL.{ value = v; permission = out_perm } SFVL.empty in
    let bound = None in
    let () = Hashtbl.replace heap loc_name (Block.Allocated { data; bound })
    in Ok []
  in let in_bounds data bound =
    let full_perm = Expr.num 1.0 in
    let none_case () =
        extend_block ~pfs ~gamma heap loc_name ofs v data bound out_perm
    in let some_case _ value permission =
      let eq = Formula.Infix.(value #== v) in
      let new_perm = Expr.Infix.(permission +. out_perm) in
      let fl = Formula.Infix.(new_perm#<=.full_perm) in
      let () = Hashtbl.replace heap loc_name (Block.Allocated { data; bound })
      in Ok [ eq; fl ]
    in check_sfvl ~pfs ~gamma ofs data none_case some_case
  in overwrite_cell ~unification ~pfs ~gamma heap loc_name ofs
    block_missing in_bounds
```

Listing 4.12: Implementation of `SetCell`, after the project

Allocating and disposing of blocks work very intuitively. Alloc sequentially creates the cells containing the null values and a full permission and, in the end, creates the block holding the resulting

SFVL and the given bound, again with full permission (see Listing A.22). During development, we discovered a a bug that had been present in the codebase for 4 years without the knowledge of the development team. When calling the alloc function from the handler in `wislSMemory.ml`, we were creating a literal made of the given location, instead of an abstract location type, causing the unifier to fail in comparing memory locations that should be equal, and thus failing in merging the SFVL blocks, as described above (see Listing A.23 and Listing A.24 for a description of the bug). Dispose performs the required sanity checks (checking that the block exists, is not freed and has a known bound with permission 1), and returns the appropriate types of errors if any of those checks fail. If these checks succeed, dispose checks the permission of each individual cell in the block and, if any of them can be less than 1, returns the appropriate **MissingResource** error. Otherwise, we set the entire block to freed and return an `Ok()` result, signalling the success of the operation (see Listing A.25).

4.2 Parsing, Compilation and Floating Point Arithmetic

As always, every interesting project has some rather mundane but crucial tasks that need to be done. In our case, adding concurrency and fractional permissions to WISL required a way of expressing these constructs in the language. We have created our own, new language, capable of running multiple threads in parallel and writing function specifications that attach permissions to cells. Working with fractional permissions meant that we needed to model floating point numbers in WISL and perhaps provide some basic arithmetic for them, in order to manipulate these parameters in our specifications. As mentioned in Section 3.3, these features were not supported by the language, so we had to come up with a syntax to express them in WISL, then write parsers that would understand that syntax and correctly map it to AST nodes, and finally, extend the WISL to GIL compiler to treat the newly created types of nodes and translate them into GIL AST nodes.

The parser was written using Menhir [29], a parser generator library for OCaml. The lexer used in combination with Menhir was Ocamllex. They proved to be very easy to use and intuitive, which was very useful for developing features on the fly, as the need arose (such as for specification targets). Parsing of floating point numbers can be done as easily as using regular expression matching, while new keywords were created by giving a name and a string representation that would be matched on from the current buffer (see Listing A.26).

Parsing rules were just as easy to create. Fully compositional, Menhir allowed for easy manipulation of simpler constructs in the parsing rules of more complex ones, provided great out-of-the-box support for commonly required structures (lists, optionals) and the integration with the AST nodes written in OCaml was seamless (see Listing A.27).

We chose to keep the intuitive syntax of cell assertions and include the permission parameter next to each pointed to value. Before the project, an assertion like `x -> v, w` would read as "x points to v and x + 1 points to w". Now, in WISLFP, we can write assertions of the form `x -> (p1: v), (p2: w)`, meaning "x points to v with permission p1, and x + 1 points to w with permission p2". Compiling the points-to assertion was as easy as adjusting the cell and bound constructors (see Listing A.28 and Listing A.29 for the compilation before and after the project).

Adding the parallel composition operator was a challenging piece of work and a crucial part of the entire project. The parts that are concerned with the underlying support in the core engine are discussed in Section 4.3. Here, we will focus on the WISL syntax and the compilation to GIL.

A par block contains any non zero number of function calls, with their return values assigned to local variables. Those variables can be used after assignment outside of the par block, as per normal. We do not allow nested par blocks, since a par block does not constitute a function call, so there can be no par block inside another one. Listing A.30 contains an example of a function that reads two disjoint memory locations in parallel and then swaps them around.

We enforced the restriction to function calls inside a par block as a syntactic, rather than semantic rule (the parser would return an error if anything other than a function call was found inside a par block). The AST nodes do not convey this information. We could have extracted the function call as a different type of node and specialise the new Par node to take a list of function calls, but we decided that would complicate the structure of the statement AST node too much, so we opted for simplicity (aside, this is where object oriented features would have proved helpful, in conjunction with the functional paradigm, like Scala provides). A par node takes a list of statements that are pattern matched inside the WISL to GIL compiler. If anything other than a function call is found in the list of nodes passed inside par, the compiler will fail with a suggestive message. However, when it came down to adding the new par AST node for GIL, we decided having a clean structure was a priority. Since the representation of GIL is part of the core engine, making a poor engineering choice would impact future scalability of the code, should support for parallel computations be added to other Gillian instantiations. Listing A.31 contains the `WStmt.ml` node extended with the par operator, and its compilation to GIL can be found in Listing A.32.

In terms of floating point arithmetic, there were 2 options: overloading the already existing operators, or writing specific ones from scratch. We chose the latter, much because of the added complexity of overloading the integer arithmetic, which would have required heavier reworking of the WISL type system, interference with internal functions written by hand in GIL, and more complicated compilation when choosing between the underlying integer or floating point GIL operators (eg. do both operands of an addition have the same type? If not, which operator do we choose?). By restricting floats to their own set of arithmetic operators, we saved important development (and possibly painful debugging) time, and made the best use of the underlying support of the Gillian core engine.

Even though manipulating permissions in WISL would probably only require a division operator (eg. when splitting a permission in half, as we shall see in the evaluation section), adding just one operator seemed unintuitive. Instead, we chose to implement a full floating point arithmetic and let future users of WISL enjoy the benefits of this bi-product. Now we can perform all types of operations with floats, and we can even store them inside heap cells. All operators are the same as their integer correspondents, except they are preceded by an 'f' (see Listing A.33).

Finally, thanks to the already existing support for floats in GIL, originating from the JavaScript instantiation, compilation was an easy process (see Listing A.34).

4.3 Parallel Composition

Implementing the parallel composition operator was perhaps the most challenging part of the project. It required deep understanding not only of theoretical semantics, but also of the internals of the core engine of Gillian, as well as careful and ingenious software engineering design. It took almost an entire week to reach a working implementation, and the many hours spent on debugging proved to be a very enriching and fascinating experience.

Before discussing the implementation of parallel composition, we will look at its theoretical semantics and get a solid understanding of what verifying functions inside a par block entails.

4.3.1 Theory

Performing a function call during verification is a simple process: the pre-condition of the function has to be consumed from the state, and its post-condition is produced. But how do we deal with multiple function calls operating in parallel? The first idea is to generalise the same process to n functions: we consume the pre-condition of the first function and produce its post-condition, and repeat for the next one. However, doing this is incorrect from a semantic point of view: it is equivalent to performing sequential function calls. A sequence of the form `a := f(x); b := g(y)` would be verified in this exact manner: the pre-condition of f would be consumed and its post-condition would be produced, and only then, the same thing would happen for g.

So how can we express the concept of concurrent function calls semantically in our verifier? Intuitively, we want to create the illusion of incomplete execution of one function at the same time as executing the other. If we considered concurrent execution outside of a verification setting, we could theoretically interrupt the process and see an intermediate state, where both function calls would have started executing but neither would have finished. This is precisely the idea that we will use in verification too.

Evaluating a parallel composition of two (or any number of) functions is performed by first consuming the pre-conditions of *all* functions in the par block. This ensures that the state right before the par block is valid for a concurrent execution of all of these functions: they can all operate in parallel and consume bits of the same state, without interfering with each other. The second step is to produce the post-conditions of *all* function calls, equivalent to waiting until all threads have reached completion to resume the execution of the main thread. The specifications can be consumed or produced in any order, as the functions are completely independent from each other. Even though they might operate on shared resources, they do so by acquiring partial permissions on those resources, effectively consuming disjoint portions of state. To make the idea more clear, consider that functions f and g in the previous example both consumed a permission of 0.3 of the same cell. The order in which the pre-conditions are consumed has no implications on the outcome: the same cell will lose 0.3 of its permission twice, giving them off to the functions, and leaving a permission of 0.4 in the heap. Whether the first 0.3 belongs to f or g is irrelevant. In this sense, the portions of the heap consumed by the functions are disjoint, because they are expressed as subdivisions of the full permission.

Below is the derivation rule for the parallel composition operator, for 2 function calls. Γ is the function specification context, and γ is the function domain, while $f(\vec{x})$ signifies the application of function f with all its required parameters. The rule states that if functions f and g have the required specifications in the specification context, then we can execute them in parallel, provided we start from a state from which both pre-conditions P and P' can be consumed. If this is the case, we end up producing the post-conditions Q and Q', with the assignment variable (a and b, respectively) in place of the return variable.

$$\frac{\{P\}\, f(\vec{x})\, \{Q\} \in \Gamma,\ \{P'\}\, g(\vec{x})\, \{Q'\} \in \Gamma}{(\gamma, \Gamma) \vdash \{P * P'\}\, \texttt{par \{a := f(}\vec{\texttt{x}}\texttt{); b := g(}\vec{\texttt{x}}\texttt{)\}}\, \{Q[ret/a] * Q'[ret/b]\}}$$

The rule generalises very easily to any number of function calls, as long as the function specifications are included in the specification context and that all the pre-conditions can be consumed from the original state (see below, where $\circledast_{1 \le i \le n}$ is the aggregation of n separating conjunctions).

$$\frac{\{P_1\}\, f_1(\vec{x})\, \{Q_1\} \in \Gamma, \dots\ \{P_n\}\, f_n(\vec{x})\, \{Q_n\} \in \Gamma}{(\gamma, \Gamma) \vdash \{\circledast_{1 \le i \le n} P_i\}\, \texttt{par \{a1 := f1(}\vec{\texttt{x}}\texttt{); ... an := fn(}\vec{\texttt{x}}\texttt{)\}}\, \{\circledast_{1 \le i \le n} Q_i[ret/a_i]\}}$$

4.3.2 Implementation

The implementation of the parallel composition operator was very much a process of generalising the evaluation of function calls. Following the already existing code in the `g_interpreter.ml` and the `PState.ml` files was an easy process. However, generalising the implementation to n function calls proved to be difficult, as previous assumptions hardcoded in the single function call evaluation no longer served our needs.

The following presentation will guide the reader through the engineering process of implementing parallel composition, discussing all the relevant procedures in comparison to their single function call counterparts, starting with the match on the type of GIL command (Listing A.35).

The `eval_par_call` function (Listing A.36) extracts the assignment variable, procedure identifier, list of arguments and logical bindings from the program state, for each function in the block, in the same manner `eval_call` (Listing A.37) does for a single function.

Evaluating the procedure call references a specific function in the `Eval_proc_call` module, written in isolation, but still as part of `g_interpreter.ml`. Its purpose, in the case of a single function call (see Listing A.38), is to extract the specification of the function, if one exists, and execute it accordingly (with specification, i.e. consume the pre-condition and produce the post-condition, or without specification, i.e. symbolically execute the procedure), also depending on the mode of operation used (special processing of the call stack is necessary for bi-abduction). In the case of parallel composition, having a function without a specification is illegal, and there is no support for bi-abduction, which simplifies the implementation (see Listing A.39).

Executing the parallel call with specifications is effectively just a mapping of its single function call counterpart over the list of functions. The bindings are evaluated and converted to a final substitution that will be used by the `PState` when running the specification. The results are counted and then processed accordingly (inside `process_ret`), by updating the state, call stack, and yielding execution back to the interpreter in case of success, or reporting failure. Listing A.40 shows the implementation for par, while the single function version can be found in Listing A.41.

Finally, the most challenging part of the implementation is the `run_par_spec` function (see Listing A.42), inside `PState.ml`. This procedure is, once again, a generalisation of its single function call version, `run_spec` (see Listing A.43), excluding the additional processing for exact verification, as parallel composition is supported only for the classic, over-approximating verification mode. The reader will be able to observe the structural similarities of the two functions, as `run_par_spec` applies mostly the same operations as `run_spec`: initialising the new store by copying in the actual parameters, combining the existential (i.e logical variable) bindings and the store bindings in a full substitution, unifying the state with the pre-condition and returning a list of possible states after producing the post-condition. The only major difference is `run_par_spec` recurses into the tail of the list of functions passed as parameters, whereas `run_spec` naturally only deals with one function (hence why its arguments were separated instead of combined in a tuple, like in the case of `run_par_spec`).

Understanding the mechanism of this recursive algorithm was not an easy task. The fact that a successful unification might return multiple states was the first obstacle, as the post-processing code had to be mapped over the entire list of states (here performed through the `let++` operator, equivalent to `concat_map`). For a single function call, states can be returned in any order and treated independently by the interpreter. For a par block, states returned by producing the post-condition of a function have to be re-used and extended when producing the post-condition of the next. This is due to the principle of framing: the first function call will consume its pre-condition, leaving behind a frame state from which the next function call will consume its pre-condition. The final frame is the heap state after all function calls have consumed their pre-conditions. From there, the post-conditions produced by the functions are sequentially added to the frame state, constructing the final state at the exit of the par block. When there are multiple such frame states, the growth is exponential, as all states produced by the post-condition of the first function have to be crossed with all the states produced by the previous one.

Furthermore, these states will contain the variable stores with all the local variables of the respective function. The local variables of a function *must not* be kept in the state when producing the post-condition of the next, as aliasing might occur due to name clashes and the state can be corrupted. Since the unifier copies the store of the caller after producing the post-condition, we had to ensure that every function call would consume its pre-condition from a state with the store of the main (caller) thread, even though the heap has to be the frame of the previous unification. This resulted in the copying of the old store into the frame state before consuming the pre-condition, and the new store back into the frame state before consuming the post-condition.

This approach worked very well to prevent name aliasing and it would have been a good (albeit complicated) solution. There is, however, one caveat: the return variable, which is the only variable in the local store of a function that has to be persisted into the caller store. There would be as many such return variables as there are function calls. The problem with the algorithm above is that the return variable of the function call processed previously was always lost, and, as a result, only the first function call would have its return variable preserved, due to the reverse order processing of the functions (similar to a call stack). The solution to this problem can be considered "hacky", but it is actually quite elegant: we extract the return variable from the local store after producing the post-condition, then copy the old store (the one of the caller) back into the state and extend it with the return variable. The only caveat is that par blocks containing function calls that assign to the same variable will only observe the effects of the first function call. We considered this limitation to be acceptable, as any useful application of parallel composition that needs the return values of the functions outside the par block would assign them to separate variables.

4.4 Summary

This chapter provided a complete presentation of the process of implementing the fractional permission memory model in a new instantiation of Gillian, WISLFP. We built on the foundation on Chapter 3 and filled in the gaps to complete the goal of the project: supporting verification of shared memory concurrency in Gillian. We took a look at both the theory and the implementation behind the producers and consumers of core assertions, exploring the representation of blocks through the SFVL data structure and how unification of memory locations is performed, and we saw the engine in action consuming and producing fractional bits of heap cells. We also implemented specialised support for the load and store operations, and came up with a modular implementation using common software engineering practices. We discussed parsing with the Menhir parser generator library and the choices made to support floating point arithmetics and specification parameters, and we touched upon the small tweaks needed in the WISL to GIL compiler. Finally, we went in detail through the theory and implementation of the parallel composition operator and the changes made in the internals of the Gillian engine, namely the GIL interpreter and the predicate state.

Chapter 5

Evaluation

The project's merits are considerable with respect to the lifetime and evolution of the Gillian tool. We have successfully extended the tool with support for verification of programs making use of shared memory concurrency, implementing the model of fractional permissions. In this chapter, we will show the full power of the new WISLFP language and prove some idiomatic concurrent algorithms, from both the category of cautious and daring concurrency, with little to no performance overhead. The fact that Gillian can successfully verify such programs is a clear advancement towards its goal of becoming a robust and complete verification tool that anyone can use. Even though the project focused on extending the WISL toy programming language, all of the concepts implemented by the WISFLP memory model can be applied to real world languages, like C and JavaScript. Moreover, the support for parallel composition of function calls has been implemented in the core of the Gillian engine, thus future efforts towards adding capabilities of concurrency verification in any other instantiation of Gillian can make use of this common trunk.

An important fact to mention is that we do not claim that this work is the first instance of concurrent code verification ever performed. In fact, there are several tools in the academic world, such as VeriFast [6, 7] and VerCors [15, 30] that have made use of the exact same ideas presented in this project (concurrent separation logic with fractional permissions). However, Gillian is very much in its infancy as far as concurrency is concerned, and therefore we consider comparisons with these much more developed tools, in which years of research and continuous efforts have been invested, to be unfair. We will still compare our performance results from testing to those of VeriFast, with the disclaimer that we can justify any performance benefits in favour of Gillian by considering the much more covering support that VeriFast offers in terms of target languages and programming constructs (locks, atomics etc.) and the complexity of the implementation associated with it.

5.1 Test Suite

Coming up with an extensive test suite that would not only exercise numerous types of behaviours in concurrent programming, but also show real value through their complexity and importance in showcasing fundamental programming concepts, has been a great challenge and a fulfilling experience. Not only did we implement perhaps the most difficult program ever written in WISL, in the form of concurrent lambda term substitution, but doing so required understanding of features of Gillian that had previously been unknown to us, such as lemmas and manual application of logical actions. Learning about these features and using them in our programs uncovered old bugs that had been lurking for years, undiscovered by the Gillian development team, some of which were mentioned in Chapter 4. Therefore, we consider this experience a very rewarding one for both parties: firstly, as a successful and complete individual effort for the student, with numerous learning opportunities, and secondly, in the form of new well tested features for Gillian and the bi-products of bug discovery and fixing.

5.1.1 Floating point arithmetic

We mentioned before that supporting fractional permissions gave rise to the obvious need of supporting floating point numbers and arithmetic, in order to operate with fractions. As this realisation came to mind, we decided to implement full support for floating point numbers and arithmetic in WISL, even though we will only require floating point division in the context of dividing the permission of a shared resource across multiple threads, as we shall see in one of our examples. Nevertheless, we have written a simple test suite meant to exercise all aspects of floating point operations in WISL, so as to make sure that our work with permissions (and any future development reliant on floating point arithmetic) would not be interrupted by tangential issues. Section 4.2 showed an example from our test suite, using floats inside a predicate as typed logical variables, and inside a function as program variables, showcasing both arithmetical and logical binary operators working on floats. The example in Listing A.44 tests all implemented operators and storing of floats inside heap cells.

5.1.2 Simple Concurrency

As mentioned before, correctness was the primary factor considered when testing our implementation, as it is a crucial aspect of any verification tool that we want to label as reliable. The first thing to test in terms of concurrency were some very simple functions accessing shared heap resources. Simple as they were, these examples exercised all the basic functionality needed in order to prove much more complicated examples: producers and consumers for cell assertions with permissions (making sure the removal mechanism of permissions worked correctly and owners can only retrieve a fraction from the heap as long as the heap holds at least that amount of permission, and then producing the cell assertion adds that permission to what the heap already contains), the unification of pre- and post-conditions in a par block, loading with any non zero permission and storing with full permission only.

First and foremost, we wrote two basic read and write functions. The read function works with any permission and returns 1 if the value pointed to by x is 0, and 0 otherwise. This is captured by the `is_zero` predicate. The write function requires full access to the cell, and writes value v to it. A write function whose specification does not enforce that the permission is equal to 1.0 fails with a missing resource error.

```
//should fail
[spec incorrect_write: #p]
{ (x == #x) * (v == #v) * (#p f<=# 1.0) * (#x -> (#p: #v))}
function incorrect_write(x, v) {
    [x] := v;
    return 0
}
{ (ret == 0) * (#x -> #v) }
```

Listing 5.1: Incorrect write function

```
Verifying one spec of procedure incorrect_write... FAILURE: Action store failed
with: [Missing Resource]
```

Listing 5.2: Output of the failure in the terminal

```
{ (x == #x) * (v == #v) * (#x -> (1.0: #y))}
function write(x, v) {
    [x] := v;
    return 0
}
{ (ret == 0) * (#x -> #v) }
```

Listing 5.3: Correct specification of write function

```
predicate is_zero(+x, y) {
    (x == 0) * (y == 1);
    (! (x == 0)) * (y == 0)
}
```

```
{ (x == #x) * (p == #p) * (#p f<=# 1.0) * (#x -> (#p: #v)) }
function read(x, p) {
    n := [x];
    if (n = 0) {
        r := 1
    } else {
        r := 0
    };
    return r
}
{ is_zero(#v, ret) * (#x -> (#p: #v)) }
```

Listing 5.4: Basic read function and predicate

We can now include these functions within par blocks and check that the producers and consumers work as expected. Firstly, reading and writing to disjoint locations should have no problem in reaching the correct post-condition, since the two threads do not interfere with each other. Any bugs we would find here would be caused solely by an incorrect implementation of the parallel composition interpreter. For the sake of brevity, we will only go through concurrent disjoint reads in detail. The implementation of concurrent disjoint writes and the corresponding logs can be found in Appendix A, Listing A.45 to Listing A.53.

```
{ emp }
function conc_disjoint_reads() {
    x := new(2);
    [x] := 0; [x + 1] := 0;
    par {
        fst := read(x, 0.5);
        snd := read(x + 1, 0.5)
    };
    [x] := fst; [x + 1] := snd;
    return x
}
{ ret -> 1, 1 }
```

Listing 5.5: Concurrent reads on disjoint locations

The following logs highlight all the intermediate states we are interested in seeing. We can see the correct behaviour being logged by Gillian: initially, memory location `_$l_12` contains two cells with full permission. We unify against the pre-condition of the first function call, resulting in a state where the first cell has half a permission left in the heap. Then, we unify against the pre-condition of the second function call and we remove half a permission from the second cell, as expected.

```
CMD: par [fst := "read"(x, 0.5); snd := "read"(gvar3, 0.5)]
PROCS: [conc_disjoint_reads]
STORE:
  (gvar0: {{  }})
  (gvar1: {{ ALoc _$l_12, Lit 1i }})
  (gvar2: {{  }})
  (gvar3: {{ ALoc _$l_12, Lit 1i }})
  (x: {{ ALoc _$l_12, Lit 0i }})
MEMORY:
  _$l_12 -> BOUND: Some (2, 1.)
            {0i: 0i [1.]
             1i: 0i [1.]}
```

Listing 5.6: State before executing par

```
Unifier.unify: Success
STATE:
  MEMORY:
    _$l_12 -> BOUND: Some (2, 1.)
              {0i: 0i [0.5]
               1i: 0i [1.]}
```

Listing 5.7: State after unifying the pre-condition of the first function

```
Unifier.unify: Success
STATE:
  MEMORY:
    _$l_12 -> BOUND: Some (2, 1.)
              {0i: 0i [0.5]
               1i: 0i [0.5]}
```

Listing 5.8: State after unifying the pre-condition of the second function

Remember, unification of states inside a par block works similarly to a call stack, so the post-conditions will be produced in reverse order. When unifying against the post-condition of the second function, we can observe the behaviour described in Section 4.1.2 when explaining the merging of SFVL blocks: a memory location described by a new logical variable is produced initially, and later, the two logical variables are discovered to be equal through a substitution, so the two blocks are merged, adding the permission and recreating the full cell. The same behaviour is observed when unifying against the post-condition of the first function call. Notice here how the return variables of both function calls are still present in the store, a behaviour which was particularly challenging to implement correctly, as described in Section 4.3.

```
STATE BEFORE SIMPLIFICATIONS:
STORE:
  (snd: LVar _lvar_18)
  (x: {{ ALoc _$l_12, Lit 0i }})
MEMORY:
  _$l_12 -> BOUND: Some (2, 1.)
            {0i: 0i [0.5]
             1i: 0i [0.5]}
  _$l_13 -> BOUND: None
            {_lvar_20: 0i [0.5]}

Filtered and fixed subst, to be applied to memory:
[ (_lvar_18: 1i),
                                                  (_lvar_19: _$l_12),
                                                  (_lvar_20: 1i),
                                                  (_$l_13: _$l_12) ]
WARNING: SFVL.union: merging with field in both lists (1i: 0i [0.5] and 0i [0.5]),
adding permissions.
```

Listing 5.9: State when unifying the post-condition of the second function

```
STATE BEFORE SIMPLIFICATIONS:
STORE:
  (fst: LVar _lvar_24)
  (snd: Lit 1i)
  (x: {{ ALoc _$l_12, Lit 0i }})
MEMORY:
  _$l_12 -> BOUND: Some (2, 1.)
            {0i: 0i [0.5]
```

```
             1i: 0i [1.]}
  _$l_15 -> BOUND: None
            {_lvar_26: 0i [0.5]}

Filtered and fixed subst, to be applied to memory:
[ (_lvar_24: 1i),
                                                        (_lvar_25: _$l_12),
                                                        (_lvar_26: 0i),
                                                        (_$l_15: _$l_12) ]
WARNING: SFVL.union: merging with field in both lists (0i: 0i [0.5] and 0i [0.5]),
adding permissions.
```

Listing 5.10: State when unifying the post-condition of the first function

The final state shows the heap cells containing full permissions, identical to the state before the par block (except, of course, for the return variables in the store).

```
STATE AFTER SIMPLIFICATIONS:
STORE:
  (fst: Lit 1i)
  (snd: Lit 1i)
  (x: {{ ALoc _$l_12, Lit 0i }})
MEMORY:
  _$l_12 -> BOUND: Some (2, 1.)
            {0i: 0i [1.]
             1i: 0i [1.]}
```

Listing 5.11: Final state

Having checked this correct behaviour in the case of disjoint operations, it is time to do the same thing for shared resources, which was the big goal of this project since the beginning. Below are two functions that perform reads and writes to the same location. The `read_and_add` function reads the value in cell pointed to by x and returns the sum of that value and the third parameter, w (see Listing A.54). We can see that half a permission to access x is passed to each function call inside the par block in the case of concurrent reads. This will verify correctly, since reading can be performed with an arbitrary permission. However, writing requires full permission to the cell, so concurrent writes to the same location should fail.

```
{ (x == #x) * (#x -> #v) * is_int(#v) }
function conc_reads_same_loc(x) {
    y := new(2);
    par {
        fst := read_and_add(x, 0.5, 5);
        snd := read_and_add(x, 0.5, 10)
    };
    [y] := fst; [y + 1] := snd;
    return y
}
{ (#x -> #v) * ret -> #v + 5, #v + 10}
```

Listing 5.12: Concurrent reads to the same location - succeeds

```
{ emp }
function conc_writes_same_loc() {
    x := new(1);
    par {
        u := write(x, 2);
        u := write(x, 3)
    };
    return x
}
{ ret == 2 \/ ret == 3 }
```

Listing 5.13: Concurrent writes to the same location - fails due to insufficient permission

We will focus on the reads first. Similarly to the case of disjoint reads, we are expecting the cell to be consumed from the heap when unifying the pre-conditions of the function calls, except this time, there will be the same cell being consumed, each time with 0.5 permission, resulting in its disappearance from the heap. The logs shown in Listing A.55, Listing A.56 and Listing A.57 showcase precisely this behaviour.

When it comes to producing the post-conditions, we expect, again, the same behaviour: the cells will be reproduced, in order, with 0.5 permissions in turn, potentially requiring merging of heap blocks due to unification of logical variables. We can see below that, indeed, the cell at location `_$l_6` gets produced with 0.5 permission and then that location gets unified with `_$l_1`. The same thing happens for `_$l_7`, except in this case, when unifying the two locations, there are cells in both of them at offset 0, so the blocks have to be merged, adding the permissions. This results in the final state, where the full permission on the cell has been re-established.

```
STATE BEFORE SIMPLIFICATIONS:
SPEC VARS: #v, #wisl__0, #wisl__1, #x
STORE:
  (snd: LVar _lvar_8)
  (x: LVar #x)
  (y: {{ ALoc _$l_5, Lit 0i }})
MEMORY:
  _$l_5 -> BOUND: Some (2, 1.)
           {0i: null [1.]
            1i: null [1.]}
  _$l_1 -> BOUND: None
           {}
  _$l_6 -> BOUND: None
           {_lvar_7: #v [0.5]}

Filtered and fixed subst, to be applied to memory:
[ (_lvar_6: _$l_1),
                                                              (_lvar_7: #wisl__1),
                                                              (_lvar_8: (10i i+ #v)),
                                                              (_$l_6: _$l_1) ]
```

Listing 5.14: State after unifying the post-condition of the second function

```
STATE BEFORE SIMPLIFICATIONS:
SPEC VARS: #v, #wisl__0, #wisl__1, #x
STORE:
  (fst: LVar _lvar_11)
  (snd: (Lit 10i i+ LVar #v))
  (x: LVar #x)
  (y: {{ ALoc _$l_5, Lit 0i }})
MEMORY:
  _$l_5 -> BOUND: Some (2, 1.)
           {0i: null [1.]
            1i: null [1.]}
  _$l_7 -> BOUND: None
           {_lvar_10: #v [0.5]}
  _$l_1 -> BOUND: None
           {#wisl__1: #v [0.5]}

Filtered and fixed subst, to be applied to memory:
```

```
[ (_lvar_10: #wisl__1),
                                                 (_lvar_11: (5i i+ #v)),
                                                 (_lvar_9: _$l_1),
                                                 (_$l_7: _$l_1) ]
WARNING: SFVL.union: merging with field in both lists (#wisl__1: #v [0.5] and #v [0.5]),
adding permissions.
```

Listing 5.15: State after unifying the post-condition of the first function

```
STATE AFTER SIMPLIFICATIONS:
STORE:
  (fst: (Lit 5i i+ LVar #v))
  (snd: (Lit 10i i+ LVar #v))
  (x: LVar #x)
  (y: {{ ALoc _$l_5, Lit 0i }})
MEMORY:
  _$l_5 -> BOUND: Some (2, 1.)
           {0i: null [1.]
            1i: null [1.]}
  _$l_1 -> BOUND: None
           {#wisl__1: #v [1.]}
```

Listing 5.16: Final state

In the case of concurrent writes, things fail due to insufficient permissions. When consuming the pre-condition of the first function call, the cell gets removed from the heap entirely, since the specification of write requires a permission of 1. Therefore, when trying to consume the pre-condition of the second function call inside the par block, we will find that there is no cell at that offset anymore, and thus fail with a `MissingResource` error. This is precisely the correct behaviour, as explained in Section 4.1.1.

```
--conc_writes_same_loc: 1--
CMD: par [u := "write"(x, 2i); u := "write"(x, 3i)]

STORE:
  (x: {{ ALoc _$l_17, Lit 0i }})
MEMORY:
  _$l_17 -> BOUND: Some (1, 1.)
            {0i: null [1.]}
```

Listing 5.17: Initial state - concurrent writes to the same location

```
Unifier.unify: Success
STATE:
  STORE:
    (v: Lit 3i)
    (x: {{ ALoc _$l_17, Lit 0i }})

  MEMORY:
    _$l_17 -> BOUND: Some (1, 1.)
              {}
```

Listing 5.18: State after consuming pre-condition of first function call

```
Unify assertion: (<cell>(_$l_0, #wisl__1, 1.; #y), (#y, 0))
Subst:
[ (v: 3i),
  (x: {{ _$l_17, 0i }}),
  (#v: 3i),
  (#wisl__0: l-nth({{ _$l_17, 0i }}, 0i)),
  (#wisl__1: l-nth({{ _$l_17, 0i }}, 1i)),
  (#x: {{ _$l_17, 0i }}),
  (_$l_0: l-nth({{ _$l_17, 0i }}, 0i)) ]
STATE:
  SPEC VARS:
  STORE:
    (v: Lit 3i)
    (x: {{ ALoc _$l_17, Lit 0i }})
  MEMORY:
    _$l_17 -> BOUND: Some (1, 1.)
              {}

Action getcell resulted in [(WislSHeap.MissingResource
                              (WislLActions.Cell, "_$l_17", (Some 0i)))]
WARNING: Unify Assertion Failed: (<cell>(_$l_0, #wisl__1, 1.; #y), (#y, 0))
```

Listing 5.19: Error when trying to consume pre-condition of second function call

With these simple examples covered, we have a good degree of confidence in the correctness of our implementation and can proceed to writing more complicated programs and show the full power of the model of fractional permissions.

5.1.3 Concurrent Binary Trees

The first application we will look at are binary trees, a well known data structure consisting of a node that can be either a leaf (here encoded by null), or a parent of two other subtrees. Algorithms working on binary trees are classic examples of divide and conquer problems, which are very suitable for concurrent implementations, since they are embarassingly parallelisable (we can spawn a thread for each independent sub-problem, in this case, each one of the two subtrees). Proving specifications of important algorithms operating on binary trees shows the true potential of WISLFP and the value of our project, as no previous sequential WISL program had been written using this type of data structure, let alone a concurrent implementation of such a program.

We will represent (logical) binary trees as lists, using their pre-order traversal, while the representation in memory will be the intuitive one: a pointer x will either be null, or point to a cell containing a value and two pointers to independent subtrees. Below is a representation of the binary tree predicate in WISLFP. We will enforce that the same permission value will be present across all nodes in the tree, and this will be represented by the p parameter. The need for this parameter to be passed into the predicate is evident: we want to be able to reuse the predicate for functions that can take any amount of permission on a binary tree, not just the full permission. Notice how both the pointer x and the permission p are in parameters (marked by the character +), since the cell assertion needs both the location and the permission to determine its out parameters. The list representation of the tree will be the learned out parameter in this case.

```
predicate binary_tree(+x, tree, +p: Float) {
    (x == null) * (tree == nil);
    (x -> (p: #a), (p: #l), (p: #r)) * binary_tree(#l, #left, p) *
        binary_tree(#r, #right, p) * (tree == #a::(#left @ #right))
}
```

Listing 5.20: Predicate representing binary trees in WISLFP

The first algorithm we implemented is calculating the height of a binary tree. The function implementation is very simple. A leaf node will always have height 0, while in the case of a parent node, all we need to do is recurse into the two subtrees. We can do this inside a par block, since the two subtrees are disjoint (there share no resources between each other). Having calculated the height of both subtrees, we just select the greater of the two and increment it by 1. Notice the use of the permission parameter `#p` as a specification parameter. As calculating the height requires read-only access to the nodes of the tree, any (positive) permission parameter is enough to satisfy the function's specification. We do not need to split the permission here, as both parallel function calls operate on different portions of the heap, so they can both use the entire `#p` we are given from the pre-condition.

```
predicate tree_height(+x, h: Int, +p: Float) {
    (x == null) * (h == 0);
    (x -> (p: #a), (p: #l), (p: #r)) * tree_height(#l, #hl, p)
        * tree_height(#r, #hr, p) * (#hl >=# #hr) * (h == #hl + 1);
    (x -> (p: #a), (p: #l), (p: #r)) * tree_height(#l, #hl, p)
        * tree_height(#r, #hr, p) * (#hl <# #hr) * (h == #hr + 1)
}

[ spec height_spec: #p ]
{ (t == #t) *  binary_tree(#t, #tree, #p) }
function height(t) {
    if (t = null) {
        r := 0
    } else {
        lt := [t + 1];
        rt := [t + 2];
        par {
            lh := height(lt) [height_spec: (#p: #p)];
            rh := height(rt) [height_spec: (#p: #p)]
        };
        r := max(lh, rh);
        r := r + 1
    };
    return r
}
{ tree_height(#t, ret, #p) }
```

Listing 5.21: A function that calculates the height of a binary tree

The more interesting problem we had to solve with this function (and, in general, with all binary tree and, later, lambda term algorithms) was expressing the goal that the function should achieve as its post-condition. Here, this takes the form of the `tree_height` predicate. The body of this predicate mirrors the one of the `lambda_term` predicate: we have a base case for the leaf node, when the height is 0, and for the parent node, we select the greater of the heights of its left and right subtrees and add 1 to it. WISL predicates support disjunction in the form of multiple clauses, so, unfortunately, a big part of the second clause had to be duplicated between the two recursive cases.

Next, we will implement a find function. Given a binary tree and a given integer value, the function returns true if the value is an element of the tree (in its flattened, list representation) and false otherwise. To express list membership, a simple `list_member` predicate was used from a previous WISL implementation of singly-linked lists. The rather interesting aspect about this function is the need to apply logic commands. This was a great learning experience for us when testing, as we had the opportunity to learn previously unexplored parts of the WISL language, how manual folding and unfolding works, how we can write our own lemmas and apply them. In this case, we do not claim authorship of the `list_member_concat` lemma, which was, again, copied from an implementation of singly-linked lists, but it served as a great example for writing and proving other lemmas that we needed in more complicated algorithms. The list membership predicate and the concatenation lemma can be found in Listing A.58.

Once again, the implementation is straightforward. In the case of a leaf node, we clearly cannot find the desired element. For parent nodes, we first look into the current node, to see if the element in the root of the tree is the one we are looking for. If so, we return true, otherwise, we spawn two threads to recurse in parallel into the two subtrees, and return the disjunction of their results. Once again, the embarassingly parallelisable nature of this data structure makes the application of the parallel composition rule straightforward. The more challenging part of the function was applying the logical commands. Since we have to produce the `list_member` predicate in the post-condition, we need a way to combine the two instances of this predicate for the left and right subtree. This is where the concatenation lemma comes into play, but before we can apply it, we need to successfully bind the list representations of the left and right subtrees to logical variables that can then be passed as parameters to the lemma. Note that we can only bind out parameters of a predicate, which is an obstacle that required ingeniousity in overcoming in other examples. Once we have applied the lemma, we have `list_member(left  right, #v, r)` in our state, so we can cons x in front of the concatenation of the left and right subtrees and fold the list membership predicate to successfully produce the postcondition.

```
[ spec find_spec: #p ]
{ (t == #t) * (v == #v) * binary_tree(#t, #tree, #p) }
function find(t, v) {
    if (t = null) {
        r := false
    } else {
        x := [t];
        if (x = v) {
            r := true
        } else {
            lt := [t + 1];
            rt := [t + 2];
            par {
                is_in_left := find(lt, v) [ find_spec: (#p: #p) ];
                is_in_right := find(rt, v) [ find_spec: (#p: #p) ]
            };
            r := is_in_left || is_in_right;
            [[assert {bind: #left, #right} binary_tree(lt, #left, #p)
                * binary_tree(rt, #right, #p)]];
            [[apply list_member_concat(#left, #right, #v)]];
            [[fold list_member(x :: (#left @ #right), #v, r)]]
        }
    };
    return r
}
{ binary_tree(#t, #tree, #p) * list_member(#tree, #v, ret) }
```

Listing 5.22: A function that finds element v in binary tree t

Our last binary tree algorithm replaces every occurrence of element v in binary tree t by w (see Listing A.59 for the implementation). Once again, the algorithm implementation is straightforward and very much mimics the structure of the find procedure, except instead of returning a boolean value, we modify the tree in place. Observe that, since we need to modify the contents of the tree, we require full permission on the `binary_tree` predicate. The more interesting parts to talk about are the `replaced` predicate used in the post-condition, and the application of logical commands.

The code of the `replaced` predicate and its corresponding concatenation are ommitted for brevity and can be found in Listing A.60. The predicate takes two lists (flattened representations of binary trees) and the v and w values and checks that every occurrence of v has been replaced by w. The empty lists case is obvious, whereas in the non-empty case, we have to make sure that if the head of the list is v in the first list, it has been replaced by w in the second, and that otherwise it remained unchanged. This is a very good example of how apparently complicated relations can be expressed very elegantly through WISL predicates. The concatenation lemma was written from scratch, and states that if every occurrence of v has been replaced with w in two lists, then we can say the same about their concatenation. The proof works by unfolding the predicate for one of the two pairs of lists and binding the head and tail variables in the non-empty cases. Since both non-empty cases recursively call the predicate into the tails of the two lists, we can apply the lemma on the bound tail lists and the second pair passed as parameter to the lemma. Thus, assuming the contract of the lemma holds, in the same way that recursively calling a function produces its post-condition, we can deduce that the predicate holds for the concatenation of the tails with `vs2` and `ws2`. From there, the interpreter can automatically fold back the predicate by adding the heads to the two concatenated lists, and thus obtain the desired result.

To see where this lemma becomes useful, we have to consider the state after the par block executes. We have produced the post-condition of the `replace` function for the left and right subtrees, but we do not have an obvious way of folding the `replaced` predicate for the full tree. Similarly to what we did in the `find` function, we can bind the two resulting subtrees that correspond to the original subtrees where every v has been replaced with w. However, all parameters of the `replaced` predicate are ins, and, as we mentioned before, only outs can be bound. The way to navigate this is to use the `binary_tree` predicate instead to bind the original subtrees (before the par block) and the replaced subtrees (after the par block), since the list representation of the tree is an out parameter in that predicate. As we know the post-condition of the function, we know the state will also contain an instance of the `replaced` predicate with the variables we just bound, so we can simply apply the lemma, passing them as arguments. The core engine then takes care of folding the predicates by cons-ing the root of the tree back onto the concatenation of the subtrees.

So far we have seen how we can verify parallel implementations of some fundamental and commonly-used algorithms on binary trees. While very exciting and evidently powerful in terms of expressiveness, these algorithms have not yet illustrated the most crucial phenomenon in the model of fractional permissions: permission splitting. As binary trees were embarassingly parallelisable data structures, all our parallel composition calls were operating on disjoint portions of the heap, so there was no need in splitting permissions of shared resources between threads. They could have exclusive access if needed (as was the case for `replace`). In the next section, however, we will address this limitation and show the full potential of fractional permissions.

5.1.4 Lambda Terms

Lambda term substitution is mentioned by Bornat et al. in [27] as the most conclusive application illustrating the usage and expressiveness of the model of fractional permissions. Our evaluation would be incomplete had we not included this famous example, so this section will discuss it in detail.

Section 2.4.2 describes the grammar of the lambda calculus and explains the process of dividing the permission of the substituted term in the algorithm. Here, we will focus on the WISL implementation of lambda terms.

Quite similarly to binary trees, we will represent lambda terms as pointers to a natural number between 0 and 2 (referred to as the type of the term, 0 for abstractions, 1 for applications, 2 for variables), and depending on the type, either one or two more values, representing the variable v (in the case of variables and abstractions) and the subterm(s) that this term is made of (one subterm for abstractions, two for applications). As before, we enforce that the entire tree construction of the lambda term has the same permission parameter p. The predicate is given below. The choice to pass v, as well as t1 and t2, even though at most two of these three will be set is to do with convenience. We can see that in the cases where there is no (second) subterm, that corresponding variable is marked as null. In the application case, we have no variable v, so it is marked as 0 (conventional value, it will not be used by any program logic). We can discover what variables will be set to their correct values by accessing the type parameter and using the variables according to the structure of the term.

```
predicate lambda_term(+x, v, t1, t2, type: Int, +p: Float) {
    (x -> (p: 0), (p: v), (p: t1)) * (type == 0) * (t2 == null)
        * lambda_term(t1, #v1, #t11, #t12, #type1, p);
    (x -> (p: 1), (p: t1), (p: t2)) * (type == 1) * (v == 0)
        * lambda_term(t1, #v1, #t11, #t12, #type1, p)
        * lambda_term(t2, #v2, #t21, #t22, #type2, p);
    (x -> (p: 2), (p: v)) * (type == 2) * (t1 == null) * (t2 == null)
}
```

Listing 5.23: The predicate representing lambda terms in WISLFP

We will first focus on the copy function. It takes a lambda term with any given permission and creates a copy of that term in memory (which will have full permission, since it is made of newly created heap cells). The implementation is given in Listing A.61.

Once again, the difficult part lies not in the code, but in expressing the post-condition. In terms of the algorithm, we have three cases, corresponding to the three types of lambda terms: in the case of a variable v, our new term will contain just two cells, with v at offset 1 (offset 0 is reserved for the type, assigned at the end, irrespective of the number of cells the term will contain). Otherwise, we are dealing with either an abstraction or an application, which both require three cells to represent in heap. In the case of an application, we can easily read the subterms at offsets 1 and 2 and recursively call copy on each one of them in parallel, in much the same way we did with the left and right subtrees of a binary tree (since they are disjoint, the entire `#p` available can be given to both subterms). The two copies are thus created and put in place of our new term. The abstraction case is merely a combination of the previous two cases, with the variable being assigned to offset 1 and a copy of the subterm at offset 2.

The `is_copy` predicate looks quite complicated at first. It has 8 parameters, the first 4 describing the pointers to the two terms and their respective permissions, and the rest being the learned out parameters of the lambda term in the pre-condition. The philosophical reason for why the latter are kept in the predicate is to ensure that the term we copied is indeed the same term we received in the pre-condition (otherwise, we could have modified t and the post-condition would still hold as long as the returned term is a copy of the "new" t). However, since this function's specification is parameterised in terms of the permission, we cannot modify the term (the permission is not guaranteed to be 1). There is a practical reason as well behind this choice, which will become apparent when we discuss substitution. For now, we need to express the `is_copy` predicate in a way that ensures both memory locations are included (so as to not leak resources). The predicate is given below. As we can see, it completely mirrors the structure of the `lambda_term` predicate in terms of the three clauses and the recursive calls. This is a major advantage, as the similar structure makes the engine capable of folding the predicate to establish the post-condition without any manual help from lemmas (as we had in the case of binary trees).

```
predicate is_copy(+x, +y, +p1: Float, +p2: Float, v, t1, t2, type: Int) {
    (x -> (p1: 0), (p1: v), (p1: t1)) * (y -> (p2: 0), (p2: v), (p2: #t2))
        * (type == 0) * (t2 == null)
        * is_copy(t1, #t2, p1, p2, #v, #t11, #t12, #type);
    (x -> (p1: 1), (p1: t1), (p1: t2)) * (y -> (p2: 1), (p2: #t21), (p2: #t22))
        * (type == 1) * (v == 0)
        * is_copy(t1, #t21, p1, p2, #v1, #t11, #t12, #type1)
        * is_copy(t2, #t22, p1, p2, #v2, #t_21, #t_22, #type2);
    (x -> (p1: 2), (p1: v)) * (y -> (p2: 2), (p2: v)) * (type == 2)
        * (t1 == null) * (t2 == null)
}
```

Listing 5.24: The `is_copy` predicate

Copying a lambda term is quite intuitive and brings nothing new to the table. We had already seen examples of disjoint concurrency working perfectly for binary trees. However, we need this function in order to perform lambda term substitution, which, as we shall see, does not enjoy this property of disjointness anymore.

We start by describing the `substituted` predicate, shown in Listing 5.25, which states that `t_out` is the same as `t_in`, except every occurrence of v has been replaced with t. The permission parameters are `p1` for `t_in`, `p2` for `t_out`, and `pt` for `t`. All parameters of the predicate are in parameters, as learning `t_out` would have meant duplicating the resource of the entire resulting lambda term, and thus we would have needed to compare that with the returned term using the `is_copy` predicate. This version of the predicate allows to check that the term we produced is indeed a substitution of the original term.

One important aspect we had to take into account was writing a post-condition for the function that would not leak resources (would describe all the cells present in the heap). This is particularly difficult in the case of the `substituted` predicate. This is because in some cases we have to perform the substitution, such as in the application case, and other times, we need to keep the term intact, such as for an abstraction where the bound variable is the same as `v`. We cannot simply copy the `#t` term referenced by `t_in` to `t_out`, as that would create aliasing. Instead, we have to create a new subterm `#t2` for `t_out` and ensure that this is an exact copy of `#t`. Extra care is needed when using the `is_copy` predicate, since its definition contains cell assertions, and duplicating any of them would make the predicate equivalent to false, as the * operator describes *disjoint* portions of the heap. For this exact reason, we have to specifically include the `t` lambda term (through the `is_copy` or `lambda_term` predicates) in the cases where it is not substituted, in order to not leak those portions of the heap. Particularly interesting is the application case, where we divide the permission on the term `t` in half when recursing, as both subterms need to substitute t. Eventually, those two subterms would reach their base cases of the predicate (where no more substitutions are applied), and they would produce the cell assertions for the term `t`. Each occurrence of `t`, whether it is through the `lambda_term` predicate in the first and last clauses, or through the `is_copy` predicate for the case where we replace variable v with `t` (the fourth clause), so `t_out` becomes a copy of `t` itself, will have a permission of `pt / 2^(num_splittings)`, where the number of splittings is equal to the number of application subterms encountered on the path to that base case. We trust the reader can observe that there will be exactly `2^(num_splittings)` such cell assertions, and their permissions would sum up to the entire `pt`, making the term fully contained inside the definition of the predicate.

```
predicate substituted(+t_in, +t, +v, +p1: Float, +p2: Float, +pt: Float, +t_out) {
    (t_in -> (p1: 0), (p1: v), (p1: #t)) * (t_out -> (p2: 0), (p2: v), (p2: #t2))
        * is_copy(#t, #t2, p1, p2, #tv, #tt1, #tt2, #type1)
        * lambda_term(t, #w, #t11, #t12, #type2, pt);
    (t_in -> (p1: 0), (p1: #v), (p1: #t)) * (! (#v == v))
        * (t_out -> (p2: 0), (p2: #v), (p2: #t_out))
        * substituted(#t, t, v, p1, p2, pt, #t_out);
    (t_in -> (p1: 1), (p1: #t1), (p1: #t2))
        * (t_out -> (p2: 1), (p2: #t1_out), (p2: #t2_out))
        * substituted(#t1, t, v, p1, p2, pt f/ 2.0, #t1_out)
        * substituted(#t2, t, v, p1, p2, pt f/ 2.0, #t2_out);
    (t_in -> (p1: 2), (p1: v)) * is_copy(t, t_out, pt, p2, #v, #t1, #t2, #type);
    (t_in -> (p1: 2), (p1: #v)) * (! (v == #v)) * (t_out -> (p2: 2), (p2: #v))
        * lambda_term(t, #w, #t1, #t2, #type, pt)
}
```

Listing 5.25: The `substituted` predicate

The implementation of the substitution function is given below. Unlike the example presented by Bornat et al. in their paper, we will not be modifying the lambda term in place, since it would be impossible to derive a post-condition that would correctly express a correct substitution. As a result, we take any permission parameter for the `x` lambda term, as well as for the `y` (in accordance to the paper in the case of the latter). Following the structure of the predicate, if the term is a variable `w`, we replace it by (a copy of) `y` if `w` is equal to the variable `v` we want to substitute, otherwise we simply copy the entire term. Notice the need to manually unfold the `is_copy` predicate in the second case. This is because of the structure of the `substituted` predicate, which contains the two cell assertions for `t_in` and `t_out` explicitly unfolded, as well as the `lambda_term` predicate for `t`. By unfolding `is_copy`, we obtain precisely these separation logic assertions in our state, after which the folding of `substituted` can be done automatically. The same unfolding is not needed in the `v = w` case, as the `substituted` predicate actually contains an instance of `is_copy` parameterised with `t` and `t_out`, so the folding of the post-condition can be done with no need for manual help.

We will discuss the (slightly simpler) abstraction case, before we get to the substitution. In the case `x` is an abstraction, we have to check the bound variable `w`. If it is the same as `v`, then we must not substitute it in the subterm, so instead, we just copy the subterm and attach it to our result. If however, `w` is different from `v`, we have to perform the substitution. Once again, the post-conditions of `copy` and `subst` produced in these cases match exactly the corresponding clauses of the `substituted` predicate we need to fold, so the verification is simple.

Finally, in the application case, we have to substitute the variable inside both subterms. Since they are different subterms, we can do this in parallel with no problem. However, we need to pass the same term `y` to both recursive calls. What can we do in this case? The answer is the one we have been waiting for: we split the permission on `y` in halves. Each recursive call to subst will instantiate its post-condition with half of our current permission on `y` in the place of `#py`. Notice the need to bind this new logical variable in an assertion in order to use it later, and the usecase of the floating-point division that justified implementing the arithmetic. However, we are not able to unify against the pre-condition of the `subst` function, because the `lambda_term` predicate for `y` is instantiated with the full `#py` permission. Our engine is not smart enough to deduce that the predicate can be split into two identical parts with permissions summing up to `#py`. Thus, we need to apply a lemma that can do this for us. The splitting permission lemma can be found in Listing A.62, which states that if we have an instance of a lambda term with permission `p`, we can split it into 2 different instances of the same term with permissions `p1` and `p2`, as long as `p1 + p2 = p`. Its proof is not at all complicated. It works by first unfolding the `lambda_term` predicate, and, since we need only treat the recursive cases (as the engine can automatically infer the property for the non-recursive one, by checking that the cell assertions combine to the full permission), we follow the structure of those clauses. We bind the logical variables found in each recursive call to the predicate (which will contain the full permission p), and recursively apply the lemma on those instances, with the same `p1` and `p2` that we are given as parameters to the lemma.

```
[spec subst_spec: #px, #py]
{ (x == #x) * (y == #y) * (v == #v) * lambda_term(#x, #vx, #tx1, #tx2, #type_x, #px)
    * lambda_term(#y, #vy, #ty1, #ty2, #type_y, #py) }
function subst(x, y, v) {
    type := [x];
    if (type = 2) {
        w := [x + 1];
        if (v = w) {
            r := copy(y)[copy_spec: (#p: #py)]
        } else {
            r := copy(x)[copy_spec: (#p: #px)];
            [[unfold is_copy(#x, r, #px, 1.0, #vx, #tx1, #tx2, #type_x)]]
        }
    } else {
        r := new(3);
        [r] := type;
        if (type = 1) {
            t1 := [x + 1]; t2 := [x + 2];
            [[assert {bind: #py_half} (#py_half == #py f/ 2.0)]];
            [[apply split_lambda_term_permission(#y, #vy, #ty1, #ty2, #type_y,
                #py, #py_half, #py_half)]];
            par {
                ct1 := subst(t1, y, v)[subst_spec: (#px: #px), (#py: #py_half)];
                ct2 := subst(t2, y, v)[subst_spec: (#px: #px), (#py: #py_half)]
            };
            [r + 1] := ct1; [r + 2] := ct2
        } else {
            w := [x + 1];
            t := [x + 2];
            if (v = w) {
                ct := copy(t)[copy_spec: (#p: #px)]
            } else {
                ct := subst(t, y, v)[subst_spec: (#px: #px), (#py: #py)]
            };
            [r + 1] := w;
            [r + 2] := ct
        }
    };
    return r
}
{ substituted(#x, #y, #v, #px, 1.0, #py, ret) }
```

Listing 5.26: Implementation of lambda term substitution in WISLFP

Now we can apply our lemma to split the `lambda_term` predicate on `y` into two different instances, each with `#py_half` permission. This makes the unification of the pre-conditions inside the par call a formality. Finally, the post-conditions produced perfectly mirror the structure of the application case in the `substituted` predicate, so the folding is performed by the engine automatically.

With this famous example, we have seen a truly remarkable way in which fractional permissions can be applied to verification of concurrent programs with ease and elegance. We believe we have undergone a thorough and complete testing process, which not only uncovered bugs in legacy code that were fixed as a result, but also showed how solutions to famous, complex programming problems can be fully verified. This demonstrates the significant value of our contributions to the Gillian ecosystem, as the tool suddenly moved away from purely didactic examples to algorithms that can be applied in real-world, complex applications. Of course, the current work represents only the initial step towards this goal, as the choice of the target language is, clearly, unusable in production. Rather, the importance of our achievements is in providing a proof of concept in the area of concurrent program verification and making the initial effort towards putting Gillian at the same level with its competitors, through the simplest gateway first, in the form of WISL, hoping that similar efforts will be undergone for the other target languages in the near future.

5.2 Performance Testing

An important way to estimate the value that this project brings to the Gillian ecosystem is rooted in performance. Are the new provided features forming a good trade-off with the incurred performance overhead? Can the tool still be used within what is deemed acceptable in terms of time of execution and latency? This section will explore the process of performance testing that was performed on the new instantiation of Gillian, and compare the observed elapsed times with the ones that the old WISL instantiation on some previously-existing regression test suites, and also measure the performance of the new features demonstrated through the examples showed in the previous section.

5.2.1 Comparison with the Old Version on Regression Test Suites

The old WISL instantiation had a number of test programs used for didactic purposes (some of them to showcase the tool and its debugger extension during the Scalable Software Verification Course offered to Computing students at Imperial College London). They are mostly centered around the singly-linked list data structure. In fact, the `list_member` predicate used in the previous section was part of this test suite.

The `SLL_ex_complete.wisl` file, under `debugger-vscode-extension/sampleWorkspace`, contains an extensive set of both recursive and iterative implementations of functions operating on singly-linked lists, together with the predicates and lemmas used to prove their specifications. There are 18 functions in this file, exercising all aspects of the core engine and the memory model, from recursion, while loops and invariants, to loading and storing to memory, producing and consuming complex predicates, performing logical actions etc. Listing A.63 provides a short example of one of the functions in `SLL_ex_complete.wisl`, in both its iterative and recursive implementations. Throughout development, this file was used as a regression test suite, to test the correctness of our implementation against the old version of WISL, in order to ascertain soundness was preserved. We will use this file to compare the performance of the old WISL instantiation to the new one, WISLFP.

In many projects, performance testing is a difficult matter. Accurate benchmarks have to be acquired and tested on different machines multiple times, so as to ensure statistical consistency and eliminate bias. This would have been a difficult and long process, especially for a tool that is not deployed on the cloud, with limited relevant benchmarks to choose from. Luckily for us, the Gillian CLI outputs the elapsed time at the beginning and the end of the verification process. We will use this as the measure of our performance, running `SLL_ex_complete.wisl` file 10 times and averaging the results. The machine used for testing is a modern Dell XPS 15, with an Intel Core i7 11-th generation containing 8 cores running at 4.9 GHz, and 32 GB of RAM. Listing A.64 presents a shortened example of the outputs that the Gillian CLI prints when running `esy x wisl verify ./debugger-vscode-extension/sampleWorkspace/SLL_ex_complete.wisl`, which is the command to execute Gillian with the `esy` building tool, in the verification mode, using the `WISL` memory model, feeding the `SLL_ex_complete.wisl` file as input for verification.

The results of verifying the `SLL_ex_complete.wisl` file using the old WISL memory model, show an average time of `0.4448277` seconds across 10 iterations. In fact, the time taken by the console to print the output might be longer than the actual measured execution time of the engine. There are 25 procedures: 10 recursive functions, 7 recursive functions, each with their own while loop that Gillian verifies as a separate procedure, and 1 function that contains neither recursive calls nor while loops. This is clearly an impressive result that we would like to match.

To accurately measure the time elapsed for the WISLFP instantiation, we switch from the master branch that these results were obtained on, and checkout the `fractional_permissions` branch, on which this work was performed (and not merged into master, for logistics reasons). Executing the same command, replacing `wisl` with `wislfp`, signalling Gillian to use the WISLFP instantiation, we obtain an average time of `0.4583692`, across 10 iterations. This is within 1.5 *hundreths* of a second of the original implementation, a clearly negligible amount. We can conclude that the newly added support for concurrency does not impact performance on verification of the average, sequential program of WISL, and that the user experience remains virtually unchanged.

5.2.2 Performance of Concurrent Program Verification

The next thing that this project attempts to prove is that verification of concurrent programs can be done efficiently, with basically no impact to the user experience and negligible latencies. In order to do so, we will analyse the three concurrent test suites presented in Section 5.1 and measure their performance. We will use the same environment mentioned in the previous section and, as before, we will average out the times across 10 iterations.

The `simple_concurrency.wisl` file contains 8 functions, 2 of which fail (as expected) under verification. Running the WISLFP instantiation of Gillian on this file, we obtain an average time of `0.0399565` seconds. That is less than 4 *hundreths* of a second, equivalent to less than half a hundreth for one function. This is remarkably fast, albeit for a very simple test suite.

The `concurrent_binary_tree.wisl` file contains 4 functions (the `height`, `find`, `replace` seen before and a simple `max` function) and 2 lemmas. These functions are considerably more complex than the ones in `simple_concurrency.wisl`, both in terms of their body of code and behaviours exercised, and in terms of their specifications. Despite all this, the average measured time is `0.1376696`, which is a little over a tenth of a second. This shows that users can verify complex specifications of concurrent programs (also containing logical actions) almost instantaneously.

Finally, `lambda_terms.wisl` contains 2 functions (`copy` and `subst`), 1 lemma and the most complex set of predicates in our test suite. On top of this, the substitution function exercises the splitting of permissions in half in the application case. Therefore, this would, in theory, achieve higher coverage of our tool, which is useful not only in correctness testing methods like fuzzing, but also in performance testing, as a slow path might be triggered in a small percentage of cases. Regardless, for 10 runs of this benchmark, Gillian still achieves a remarkable average execution time of `0.3192437` seconds. If we exclude the lemma, we obtain an overestimate of approximately 1.5 tenths of a second per function. Even though such experiments never scale exactly linearly, we could theoretically consider the implementation of an entire concurrent library, that can contain an approximate number of 100 very diverse functions, not all of which will have the complexity of `subst`, to verify in approximately 15 seconds. This is more than enough for a classic performance SLA that would enable us to include Gillian as part of a more complex CI/CD architecture in production. We have thus shown that adding support for concurrency to Gillian not only did not impact the baseline performance of existing benchmarks, but also provided valuable functionality at minimal costs of latency.

5.3 Comparison to Other Tools

It is somewhat unfair to compare Gillian, a tool still very much in its infancy, to any other much bigger projects in which there have been invested much more resources, both in terms of finances, time, and sustained development. Tools such as Verifast and VerCors are available to the general public and are capable of verifying complex programming constructs (including concurrency) in established languages, such as Java and C. While Gillian does have a C instantiation that has been used to verify industry code, its support is nowhere near as complete as that of its competitors. For instance, it does not support any concepts related to concurrency and multi-threading. In comparison, VerCors and Verifast have programs in their test suites making use of constructs such as atomics, locks, monitors, p_threads, fork-join concurrency etc. This project is the first step towards adding support for any form of concurrency to Gillian, in a purely didactic language. Comparing the newly added support in terms of complexity to the capabilities of VerCors and Verifast would obviously suggest a significant advantage in the favour of the latter two. However, the goal of our project was more related to improving the current Gillian architecture and making a first step to bridge the gap to its rivals.

A significant comparison that we can make to the other tools is in terms of performance. Is Gillian behaving better or worse on average than its competitors, in terms of the time of execution?

We downloaded the Verifast repository [7] on the same machine used to benchmark Gillian and ran their test-suite. The results on their example programs exercising concurrency concepts vary from 0.03 seconds up to 23.86 seconds on a particularly slow example (see Listing A.65). The average was still under 1 second, which is comparable to the time obtained by Gillian on its most complex benchmark (lambda terms). We can safely assume that Verifast's test suite is more complex than ours, but showing results comparable to real world standards, even for simpler examples, is encouraging. This shows the value of our work in comparison to more advanced academic tools and highlights the potential room for growth and improvement in the same direction.

5.4 Limitations

While this project successfully achieves its main goal in providing a mechanism to verify shared memory concurrency in Gillian, there are a number of limitations that we would have liked to address, in a less time-restricted scenario.

Inferring permission parameters automatically. While support for fractional permissions was successfully added to the tool, they are not automatically inferred. As mentioned in Section 4.1.2, permissions are either passed as program arguments to the function or as specification parameters, which places an extra responsibility on the programmers to be aware of the concept of permissions and express it explicitly in their code. We would like to automate this process and make the tool capable of automatically inferring the permission value needed for a concurrent function and liberate programmers from the burden of writing verifier-related boilerplate code.

Other permission models. This project only implements the model of fractional permissions in WISL, which, while useful for several interesting examples, as we have seen, it is not omnipotent. Several concurrency idioms like readers and writers cannot be expressed with the help of fractional permissions. If time allowed, we would have liked to implement the counting permission model in Gillian, which would have enabled us to verify other types of concurrent programs, such as the readers and writers problem (provided some extensions to the language are made, with respect to critical sections), and would have made for a more expressive and powerful tool.

5.5 Reflection

The project was a very rewarding experience, both in terms of engineering and programming skills exercised, but also in terms of the practical research opportunity it provided. Being able to work in a niche area of Computing in the form of software verification unveiled the true potential of combining knowledge and passion. The results were a successful, novel implementation, which constitutes a significant advantage to a sizeable software artefact, and a memorable journey for the student, where working felt relaxing and joyful.

Despite having a slow start to the project, when significant time was spent becoming familiar with the codebase, we believe that the available time was spent wisely. Early conversations with Sacha proved crucial to understanding the internals of the tool, and his mentoring, patience, and guidance throughout the process of discovering the hidden details of Gillian made development a much more straightforward and pleasant experience. From the moment the first contributions to the codebase were pushed to Gillian's Github repository, development progressed steadily, with a healthy plan in mind, that ensured the project's goals would be realised with plenty of time left in hand. If this project were repeated with the gift of hindsight, research of the internals of Gillian would be performed sooner, to allow for even faster development and enough time for an extension, such as the counting permissions model.

Chapter 6

Conclusion and Next Steps

This project was a successful attempt at providing the very first, fundamental support for verification of concurrent programs to the Gillian verifier. Chapter 2 provided the theoretical foundations required for our project, starting from basic Hoare Logic, continuing with Concurrent Separation Logic and finishing with permission models. In Chapter 3, we became familiar with the architecture of our tool and its working mechanisms and gave the reader an insight into the engineering additions required to implement the model of fractional permissions. Chapter 4 detailed the implementation, showing how the new WISL memory model was written, how the language was reshaped into the new, extended WISLFP, and how parallel composition was added to the core engine to support not only its WISLFP equivalent, but also any future work in concurrency for any other target language. To showcase the value of our contributions, in Chapter 5, we verified parallel implementations of fundamental algorithms operating on an even more important data structure, in the form of binary trees, and demonstrated how fractional permissions can be used to elegantly prove specifications for lambda term substitution.

This project's objective was to provide a foundational work in the development of concurrent program verification for Gillian, through the path of least resistance (in the form of WISL). As always, there are numerous extensions that can be attempted to build on or accompany our work:

- **Inferring permissions.** As mentioned in Chapter 5, inferring the required values for the permission parameters automatically would be a challenging, but valuable piece of research that would enable the programmers to focus entirely on their code, without thinking about the verification tool and permission models.

- **Synchronisation.** A lot of useful, industry-like concurrent code contains synchronisation mechanisms and algorithms relying on language primitives, such as atomics, locks, semaphores and monitors. WISLFP currently has no support for synchronisation primitives, which limits the spectrum of programs available for verification. Adding such support to WISLFP would enable us to implement complex thread-safe data structures (lists, binary search trees, red-black trees etc.), which would make a significant difference to the expressiveness of the language.

- **More complex concurrency concepts.** There is limitless room for improvement in terms of concurrency capabilities. Things like thread pools, fork-join concurrency, the Linux wait system call will always enrich the set of concurrent behaviours we are able to express.

- **Real world target languages.** The majority of the concepts mentioned above would prove much more valuable if implemented in a language like C, as it would automatically open the door to verifying complex real world code, as opposed to (still) toy, didactic examples like those we saw in WISL.

We leave all of these are suggestions and opportunities for future development of Gillian and hope that our project will open the door to exciting advancements in the world of verification and in the development of our tool.

Chapter 7

Ethical issues

Gillian is a program analysis tool that can verify correctness of programs and identify potential problems (bugs) of said programs, should these exist. Gillian has already been able to find 5 bugs in production JavaScript and C code within AWS [11]. Out of these, 2 were classified as security vulnerabilities. This consitutues a proof for the capability of Gillian to find vulnerabilities in software systems used by millions of users. The focus of this project is on verifying concurrent code, which would extend the area of programs that can be verified and of identifiable bugs. Concurrency bugs are arguably more obscure and harder to exploit as security vulnerabilities than regular programmer mistakes like accessing a buffer out of bounds, which immediately highlights a risk for data theft. However, denial of service attacks by effectively writing a concurrency stress tests which increase the chances of manifesting potential concurrency bugs (data-races that can crash the system, livelocks that prevent the application from making any progress etc.) identified through verification are still possible. It appears that the new support for fractional permission concurrency within Gillian would make it more attractive to malicious users. However, there are a number of very practical arguments which make the use of our tool for illegal purposes highly improbable:

- Attackers rarely have access to the source code of the target system. In order to find potential vulnerabilities, the source code has to be accessible for Gillian to analyse and verify. In order to get into such a situation, the attacker would have to circumvent all the existing security measures in place, which brings us to the next point.

- Most companies have serious counter-measures in place to prevent attacks on security critical parts of their infrastructure. The difficulty an attacker would have to face in order to get full access to the source code defeats the purpose of any possible damage that could be done through a denial of service attack. Furthermore, it is to be expected that any secrets a possible exploit could leak would probably already be available to the attacker once access to the source code (arguably higher privilege) is obtained.

- It is expected that developers would verify security critical code with our tool and fix any vulnerabilities it would highlight before an attacker has the opportunity to exploit them. The open-source nature of the project should make this task that much easier, eliminating bureaucratic or financial reasons for which this would not happen.

We conclude that the existent risk for our tool to be misused is just as small as the risk of any software being misused. That is, there are no special security considerations that would make us think our tool can be found at the root of a cyber security attack. Similar tools exist in the industry, as mentioned before, such as Meta's Infer, and up to now there have been no reported security issues arising from the use of such verification tools.

Appendix A

Listings

```
(* state.ml *)

module type S = sig
  (** Type of GIL values *)
  type vt [@@deriving yojson, show]

  (** Type of GIL general states *)
  type t [@@deriving yojson]

  (** Type of GIL substitutions *)
  type st

  (** Type of GIL stores *)
  type store_t

  (** Type of GIL stores *)
  type heap_t

  ...
end
```

Listing A.1: Interface for GIL General States

```
(* SState.ml *)

module type S = sig
  include State.S

  val make_s :
    init_data:init_data ->
    store:store_t ->
    pfs:PFS.t ->
    gamma:Type_env.t ->
    spec_vars:SS.t ->
    t
  ...
end

module Make (SMemory : SMemory.S) :
  S
    with type st = SVal.SESubst.t
     and type vt = SVal.M.t
```

```
      and type store_t = SStore.t
      and type heap_t = SMemory.t
      and type m_err_t = SMemory.err_t
      and type init_data = SMemory.init_data = struct
   type vt = SVal.M.t [@@deriving yojson, show]
   type st = SVal.M.et
   type heap_t = SMemory.t [@@deriving yojson]
   type store_t = SStore.t [@@deriving yojson]
   ...

   let execute_action
       ?(unification = false)
       (action : string)
       (state : t)
       (args : vt list) : action_ret =
     let open Syntaxes.List in
     let heap, store, pfs, gamma, vars = state in
     let pc = Gpc.make ~unification ~pfs ~gamma () in
     let+ Gbranch.{ value; pc } = SMemory.execute_action action heap pc args in
     match value with
     | Ok (new_heap, vs) ->
         let store = SStore.copy store in
         let new_state = (new_heap, store, pc.pfs, pc.gamma, vars) in
         Ok (new_state, vs)
     | Error err -> Error (StateErr.EMem err)

end
```

Listing A.2: Implementation of Symbolic States, parametric in the Symbolic Memory Model

```
type 'label t = 'label TypeDef__.cmd =
  | Skip  (** Skip *)
  | Assignment of string * Expr.t  (** Assignment *)
  | LAction of string * string * Expr.t list  (** Local Actions *)
  | Logic of LCmd.t  (** GIL Logic commands *)
  | Goto of 'label  (** Unconditional goto *)
  | GuardedGoto of Expr.t * 'label * 'label  (** Conditional goto *)
  | Call of
      string * Expr.t * Expr.t list * 'label option * logic_bindings_t option
      (** Procedure call *)
  | ECall of string * Expr.t * Expr.t list * 'label option
      (** External Procedure call *)
  | Apply of string * Expr.t * 'label option
      (** Application-style procedure call *)
  | Arguments of string  (** Arguments of the current function *)
  | PhiAssignment of (string * Expr.t list) list  (** PHI assignment *)
  | ReturnNormal  (** Normal return *)
  | ReturnError  (** Error return *)
  | Fail of string * Expr.t list  (** Failure *)
```

Listing A.3: Type of GIL commands, pre-project

```
let run_spec
      (* logical bindings of variables *)
      ?(existential_bindings : (string * vt) list option)
      (name : string) (* function identifier *)
      (params : string list) (* the name of function parameters *)
      (up : UP.t) (* unification plan *)
      (astate : t) (* the abstract state *)
      (x : string option) (* the variable to which the return value is assigned *)
      (args : vt list) (* actual parameter values *)
      : ((t * Flag.t) list, SUnifier.err_t list) result
```

Listing A.4: run_spec, which runs the specification of a function when called

```
let cell ~loc ~offset ~value =
  let cell = str_ga Cell in
  Asrt.GA (cell, [ loc; offset ], [ value ])

let bound ~loc ~bound =
  let bound_ga = str_ga Bound in
  let bound = Expr.int bound in
  Asrt.GA (bound_ga, [ loc ], [ bound ])
```

Listing A.5: Cell and Bound assertions, pre-project

```
(** Gets a first key-value pair that satisfies a predicate *)
let get_first (f : field_name -> bool) (sfvl : t) :
    (field_name * field_value) option

(** Adds by testing something equal is not already there *)
let add_with_test
    ~(equality_test : field_name -> field_name -> bool)
    (ofs : field_name)
    (new_val : field_value)
    (sfvl : t)
```

Listing A.6: Retrieval and insertion of symbolic values in the SFVL

```
let union =
  Expr.Map.union (fun k fvl fvr ->
      L.(
        verbose (fun m ->
            m
              "WARNING: SFVL.union: merging with field in both lists (%s: %s \
               and %s), choosing left."
              ((Fmt.to_to_string Expr.pp) k)
              ((Fmt.to_to_string Expr.pp) fvl)
              ((Fmt.to_to_string Expr.pp) fvr)));
      Some fvl)
```

Listing A.7: Retrieval and insertion of symbolic values in the SFVL, pre-project

```
let get_cell ~pfs ~gamma heap loc ofs =
  match Hashtbl.find_opt heap loc with
  | None -> Error (MissingResource (Cell, loc, Some ofs))
  | Some Block.Freed -> Error (UseAfterFree loc)
  | Some (Allocated { data; bound }) -> (
      let maybe_out_of_bound =
        match bound with
        | None -> false
        | Some n ->
```

```
            let n = Expr.int n in
            let open Formula.Infix in
            Solver.sat ~unification:false ~pfs ~gamma n #<= ofs
      in
      if maybe_out_of_bound then Error (OutOfBounds (bound, loc, ofs))
      else
        match SFVL.get ofs data with
        | Some v -> Ok (loc, ofs, v)
        | None -> (
            match
              SFVL.get_first
                (fun name -> Solver.is_equal ~pfs ~gamma name ofs)
                data
            with
            | Some (o, v) -> Ok (loc, o, v)
            | None -> Error (MissingResource (Cell, loc, Some ofs))))
```

Listing A.8: The handler for `GetCell`, prior to the project

```
let rem_cell heap loc offset =
  match Hashtbl.find_opt heap loc with
  | None -> Error (MissingResource (Cell, loc, Some offset))
  | Some Block.Freed -> Error (UseAfterFree loc)
  | Some (Allocated { bound; data }) ->
      let data = SFVL.remove offset data in
      let () = Hashtbl.replace heap loc (Allocated { bound; data }) in
      Ok ()
```

Listing A.9: The handler for `RemCell`, prior to the project

```
let set_cell ~pfs ~gamma heap loc_name ofs v =
  match Hashtbl.find_opt heap loc_name with
  | None ->
      let data = SFVL.add ofs v SFVL.empty in
      let bound = None in
      let () = Hashtbl.replace heap loc_name (Allocated { data; bound }) in
      Ok ()
  | Some Block.Freed -> Error (UseAfterFree loc_name)
  | Some (Allocated { data; bound }) ->
      let maybe_out_of_bound =
        match bound with
        | None -> false
        | Some n ->
            let n = Expr.int n in
            let open Formula.Infix in
            Solver.sat ~unification:false ~pfs ~gamma n #<= ofs
      in
      if maybe_out_of_bound then Error (UseAfterFree loc_name)
      else
        let equality_test = Solver.is_equal ~pfs ~gamma in
        let data = SFVL.add_with_test ~equality_test ofs v data in
        let () = Hashtbl.replace heap loc_name (Allocated { data; bound }) in
        Ok ()
```

Listing A.10: The handler for `SetCell`, prior to the project

```
let compile_binop b =
  WBinOp.(
    match b with
    | EQUAL -> BinOp.Equal
    | LESSTHAN -> BinOp.ILessThan
    | LESSEQUAL -> BinOp.ILessThanEqual
    | PLUS -> BinOp.IPlus
    | MINUS -> BinOp.IMinus
    | TIMES -> BinOp.ITimes
    | DIV -> BinOp.IDiv
    | MOD -> BinOp.IMod
    (* TODO: Add FPLUS, FMINUS, FTIMES, FDIV, FMOD *)
    | AND -> BinOp.BAnd
    | OR -> BinOp.BOr
    | LSTNTH -> BinOp.LstNth
    (* operators that do not exist in gil are compiled separately *)
    | _ ->
        failwith
          (Format.asprintf "compile_binop should not be used to compile %a" pp b))
```

Listing A.11: Compiling binary operators, pre-project

```
let rec compile_stmt_list ?(fname = "main") ?(is_loop_prefix = false) stmtl =
match stmtl with
| { snode = Logic invariant; _ } :: while_stmt :: rest -> ...
| { snode = While (e, sl); sid = sid_while; sloc } :: rest -> ...
| { snode = VarAssign (v, e); sid; sloc } :: rest ->
(* TODO: { snode = Par funcs; sid; sloc } :: rest -> ... *)
...
```

Listing A.12: Compiling a list of statements, pre-project

```
let union =
  Expr.Map.union (fun k (fvl : field_value) (fvr : field_value) ->
      L.(
        verbose (fun m ->
            m
              "WARNING: SFVL.union: merging with field in both lists (%s: %s \
               and %s), adding permissions."
              ((Fmt.to_to_string Expr.pp) k)
              ((Fmt.to_to_string fv_pp) fvl)
              ((Fmt.to_to_string fv_pp) fvr)));
      let { value; permission = lperm } = fvl in
      let { permission = rperm; _ } = fvr in
      Some { value; permission = Expr.Infix.(lperm +. rperm) })
```

Listing A.13: Merging of SFVLs, after the project

```
let cell ~loc ~offset ~value ~permission =
  let cell = str_ga Cell in
  Asrt.GA (cell, [ loc; offset; permission ], [ value ])

let bound ~loc ~bound ~permission =
  let bound_ga = str_ga Bound in
  let bound = Expr.int bound in
  Asrt.GA (bound_ga, [ loc; permission ], [ bound ])
```

Listing A.14: Cell and Bound assertions, after the project

```
let load ~pfs ~gamma heap loc ofs =
  let open Syntaxes.Result in
  let+ _, _, v =
    access_cell ~unification:false ~pfs ~gamma heap loc ofs (fun _ -> None)
  in
  v
```

Listing A.15: Implementation of the `Load` local action

```
(* Helper function: Checks if the offset exists in the SFVL first by performing a performance effi
   and then using the solver. If an entry is found, it applies the success_case function, otherwis
let check_sfvl ~pfs ~gamma ofs data none_case success_case =
  let none =
    match
      SFVL.get_first (fun name -> Solver.is_equal ~pfs ~gamma name ofs) data
    with
    | None -> none_case ()
    | Some (o, SFVL.{ value; permission }) -> success_case o value permission
  in
  let some SFVL.{ value; permission } = success_case ofs value permission in
  Option.fold ~none ~some (SFVL.get ofs data)
```

Listing A.16: The `check_sfvl` function

```
let get_bound ~unification ~pfs ~gamma heap loc out_perm =
  match Hashtbl.find_opt heap loc with
  | Some Block.Freed -> Error (UseAfterFree loc)
  | None -> Error (MissingResource (Cell, loc, None))
  | Some (Allocated { bound = None; _ }) ->
      Error (MissingResource (Bound, loc, None))
  | Some (Allocated { bound = Some bound; _ }) ->
      let _, q = bound in
      let fl = Formula.Infix.(q#<.out_perm) in
      let has_less_perm =
        Solver.check_satisfiability ~unification (fl :: PFS.to_list pfs) gamma
      in
      if has_less_perm then
        let missing_permission = Expr.Infix.(out_perm -. q) in
        Error (MissingResource (Bound, loc, Some missing_permission))
      else Ok bound
```

Listing A.17: The implementation of `GetBound`, after the project

```
let rem_bound ~pfs ~gamma heap loc out_perm =
  match Hashtbl.find_opt heap loc with
  | Some Block.Freed -> Error (UseAfterFree loc)
  | None -> Error (MissingResource (Cell, loc, None))
  | Some (Allocated { bound = None; _ }) ->
      Error (MissingResource (Bound, loc, None))
  | Some (Allocated { data; bound = Some (n, q) }) ->
      let new_perm = Expr.Infix.(q -. out_perm) in
      let bound =
        if Solver.is_equal ~pfs ~gamma new_perm (Expr.num 0.0) then None
        else Some (n, new_perm)
      in
      let () = Hashtbl.replace heap loc (Allocated { data; bound }) in
      Ok ()
```

Listing A.18: The implementation of `RemBound`, after the project

```
(* Helper function: Performs the preliminary checks common to the set_cell and store
   operations and applies the "block_missing" operation if the block cannot be found
   in the heap "in_bounds" operation if the access to the allocated cell is within bounds *)
let overwrite_cell ~unification ~pfs ~gamma heap loc_name ofs block_missing in_bounds =
  match Hashtbl.find_opt heap loc_name with
  | None -> block_missing ()
  | Some Block.Freed -> Error (UseAfterFree loc_name)
  | Some (Allocated { data; bound }) -> (
      match bound with
      | None -> in_bounds data None
      | Some (n, perm) ->
          let expr_n = Expr.int n in
          let open Formula.Infix in
          if Solver.sat ~unification ~pfs ~gamma expr_n #<= ofs then
            Error (MissingResource (Bound, loc_name, Some ofs))
          else in_bounds data (Some (n, perm)))
```

Listing A.19: The `overwrite_cell` helper function

```
(* Helper function: Extends the block data with a new cell at offset ofs with the value v. *)
let extend_block ~pfs ~gamma heap loc_name ofs value data bound permission =
  let equality_test = Solver.is_equal ~pfs ~gamma in
  let data =
    SFVL.add_with_test ~equality_test ofs SFVL.{ value; permission } data
  in
  let () = Hashtbl.replace heap loc_name (Block.Allocated { data; bound }) in
  let fl = Formula.Infix.(permission #>. (Expr.num 0.0)) in
  Ok [ fl ]
```

Listing A.20: The `extend_block` helper function

```
let store ~pfs ~gamma heap loc_name ofs v =
  let block_missing () = Error (MissingResource (Cell, loc_name, Some ofs)) in
  let in_bounds (data : SFVL.t) bound =
    let full_perm = Expr.num 1.0 in
    let fl q = Formula.Infix.(q#<.full_perm) in
    let can_be_less q =
      Solver.check_satisfiability (fl q :: PFS.to_list pfs) gamma
    in
    let none_case () = Error (MissingResource (Cell, loc_name, Some ofs)) in
    let some_case ofs _ permission =
      if can_be_less permission then
        let missing_permission = Expr.Infix.(full_perm -. permission) in
        Error (MissingResource (Cell, loc_name, Some missing_permission))
      else extend_block ~pfs ~gamma heap loc_name ofs v data bound permission
    in
    check_sfvl ~pfs ~gamma ofs data none_case some_case
  in
  match
    overwrite_cell ~unification:false ~pfs ~gamma heap loc_name ofs
      block_missing in_bounds
  with
  | Error e -> Error e
  | Ok _ -> Ok ()
```

Listing A.21: Implementation of the Store local action

```
let alloc (heap : t) size =
  let loc = ALoc.alloc () in
  let rec get_list current_offset =
    if current_offset < 0 then []
    else
      ( Expr.int current_offset,
        SFVL.{ value = Expr.Lit Literal.Null; permission = Lit (Num 1.0) } )
      :: get_list (current_offset - 1)
  in
  let l = get_list (size - 1) in
  let sfvl = SFVL.of_list l in
  let block =
    Block.Allocated { data = sfvl; bound = Some (size, Lit (Num 1.0)) }
  in
  let () = Hashtbl.replace heap loc block in
  loc
```

Listing A.22: Alloc implementation

```
let alloc heap _pfs _gamma (size : int) =
  let loc = WislSHeap.alloc heap size in
  Ok [ (heap, [ Expr.Lit (Literal.Loc loc); Expr.Lit (Literal.Int Z.zero) ], [], []) ]
```

Listing A.23: Alloc handler, pre-project

```
let alloc heap _pfs _gamma (size : int) =
  let loc = WislSHeap.alloc heap size in
  Ok [ (heap, [ Expr.ALoc loc; Expr.Lit (Literal.Int Z.zero) ], [], []) ]
```

Listing A.24: Alloc handler, after the project

```
let dispose ~unification ~pfs ~gamma (heap : t) loc =
  match Hashtbl.find_opt heap loc with
  | None -> Error (MissingResource (Cell, loc, None))
  | Some Freed -> Error (DoubleFree loc)
  | Some (Allocated { data = _; bound = None; _ }) ->
      Error (MissingResource (Bound, loc, None))
  | Some (Allocated { data; bound = Some (i, perm) }) ->
      let full_perm = Expr.num 1.0 in
      let fl q = Formula.Infix.(q#<.full_perm) in
      let can_be_less q =
        Solver.check_satisfiability ~unification (fl q :: PFS.to_list pfs) gamma
      in
      if can_be_less perm then Error (MissingResource (Bound, loc, None))
      else
        let open Syntaxes.Result in
        let check_entry ac i =
          let* () = ac in
          match SFVL.get (Expr.int i) data with
          | None -> Error (MissingResource (Bound, loc, None))
          | Some { permission = q; _ } ->
              if can_be_less q then
                let missing_permission = Expr.Infix.(full_perm -. q) in
                Error (MissingResource (Cell, loc, Some missing_permission))
              else Ok ()
        in
        let+ () = Seq.init i Fun.id |> Seq.fold_left check_entry (Ok ()) in
        set_freed_with_logging heap loc
```

Listing A.25: Dispose implementation

```
let float = digit* '.' digit*
rule read =
  parse
  | "spec" { SPEC (curr lexbuf) }
  | "Float" { TFLOAT (curr lexbuf) }
  | ">#"     { LGREATER }
  | "f>#"    { FLGREATER }
  ...
```

Listing A.26: Lexing with Ocamllex

```
function_call_list:
  sl = separated_nonempty_list(SEMICOLON, function_call) { sl }
statement:
...
| lstart = PAR; LCBRACE; fs = function_call_list; lend = RCBRACE;
    {
      let bare_stmt = WStmt.Par (fs) in
      let loc = CodeLoc.merge lstart lend in
      WStmt.make bare_stmt loc
    }
```

Listing A.27: Parsing with Menhir

```
let compile_pointsto =
(* A lot of details that we will omit *)
  let eloc, eoffs = (Expr.LVar loc, gil_add expr_offset curr) in
  let bound =
    if start && block then [ Constr.bound ~loc:eloc ~bound:(List.length lle) ]
    else []
  in
  match lle with
  | [] -> failwith "Error"
  | le :: r ->
     let e2 = compile_lexpr le in
     (Constr.cell ~loc:eloc ~offset:eoffs ~value:e2)
```

Listing A.28: Compiling points-to, pre-project

```
let compile_pointsto =
(* A lot of details that we will omit *)
  let eloc, eoffs = (Expr.LVar loc, gil_add expr_offset curr) in
  let bound =
    if start && block then
      [
        Constr.bound ~loc:eloc ~bound:(List.length lle)
          ~permission:(Expr.num 1.0);
      ]
    else []
  in
  match lle with
  | [] -> failwith "Error"
  | le :: r ->
    let perm, le = le in
    let e2 = compile_lexpr le in
    let e3 = compile_lexpr perm in
    (Constr.cell ~loc:eloc ~offset:eoffs ~value:e2 ~permission:e3)
```

Listing A.29: Compiling points-to, after the project

```
{ (x == #x) * (p == #p) * (#p f<=# 1.0) * (#x -> (#p: #v)) }
function read(x, p) {
...
}
{ (#x -> (#p: #v)) * (ret == #v) }

function swap() {
    [x] := 0;
    [x + 1] := 1;
    par {
        fst := read(x, 0.5);
        snd := read(x + 1, 0.5)
    };
    [x] := snd;
    [x + 1] := fst;
}
```

Listing A.30: Example of a parallel composition block

```
type tt =
  | Skip
  | VarAssign of string * WExpr.t
  | New of string * int
  | Dispose of WExpr.t
  | Lookup of string * WExpr.t (* x := [e] *)
  | Update of WExpr.t * WExpr.t (* [e] := [e] *)
  | FunCall of string * string * WExpr.t list
      * (string * (string * WLExpr.t) list) option
    (* The last bit is only for internal use *)
  | While of WExpr.t * t list
  | If of WExpr.t * t list * t list
  | Par of t list
  | Logic of WLCmd.t
and t = { sid : int; sloc : CodeLoc.t; snode : tt }
```

Listing A.31: The new WStmt.ml node

```
| { snode = Par funcs; sid; sloc } :: rest ->
      let lambda f =
        match f with
        | { snode = FunCall (x, fn, el, to_bind); _ } ->
            let var_name, fct_name, args, bindings, cmdles =
              create_func_call x fn el to_bind
            in
            (Cmd.{ var_name; fct_name; args; err_lab = None; bindings }, cmdles)
        | _ ->
            failwith
              "Parallel composition called with a node different from FunCall!"
      in
      let zipped = List.map lambda funcs in
      let fcs = List.map (fun (f, _) -> f) zipped in
      let cmdles = List.concat @@ List.map (fun (_, s) -> s) zipped in
      let cmd = Cmd.Par fcs in
      let annot =
        WAnnot.make ~origin_id:sid ~origin_loc:(CodeLoc.to_location sloc) ()
      in
      let comp_rest, new_functions = compile_list rest in
      (List.concat cmdles @ [ (annot, None, cmd) ] @ comp_rest, new_functions)
```

Listing A.32: Compiling a par block

```
predicate points_to_sub(+x, +a: Float, +b: Float) {
    (a f>=# b) * (#v == a f- b) * (x -> #v);
    (a f<# b) * (#v == b f- a) * (x -> #v)
}

{ (x == #x) * (y == #y) }
function floating_subtraction(x, y) {
    r := new(1);
    if (x f< y) {
        z := y f- x
    } else {
        z := x f- y
    };
    [r] := z;
    return r
}
{ points_to_sub(ret, #x, #y) }
```

Listing A.33: Example of a function and a predicate manipulating floats

```
let compile_binop b = match b with
  | PLUS -> BinOp.IPlus
  | FPLUS -> BinOp.FPlus

let compile_val v = match v with
  | Int n -> Literal.Int (Z.of_int n)
  | Float x -> Literal.Num x
  ...
```

Listing A.34: Compiling floating point numbers and operators

```
match cmd with
 | Call { var_name; fct_name; args; err_lab; bindings } ->
    eval_call var_name fct_name args err_lab bindings eval_state
 | Par fcs -> eval_par_call fcs eval_state
```

Listing A.35: Matching for parallel composition and function calls in the GIL interpreter

```
let eval_par_call fcs eval_state =
  let { eval_expr; _ } = eval_state in
  let lambda Cmd.{ var_name; fct_name; args; bindings; _ } =
    let pid = eval_expr fct_name in
    let v_args = List.map eval_expr args in
    (var_name, pid, v_args, bindings)
in eval_par_proc_call (List.map lambda fcs) eval_state
```

Listing A.36: Extracting the assignment variable, pid, args and bindings of functions in a par block

```
let eval_call x e args j subst eval_state =
    let { eval_expr; _ } = eval_state in
    DL.log (fun m -> m "Call");
    let pid = eval_expr e in
    let v_args = List.map eval_expr args in
    let result = eval_proc_call x pid v_args j subst eval_state in
    result
```

Listing A.37: Extracting the procedure identifier and arguments for a function call

```
let f x pid v_args j subst eval_state =
          let { prog; cs; state; _ } = eval_state in
          let pid = get_pid_or_error pid state in
          let spec, params = get_spec_and_params prog pid state in
          let caller = Call_stack.get_cur_proc_id cs in
          let () = CallGraph.add_proc_call call_graph caller pid in
          let args = build_args v_args params in
          let is_internal_proc proc_name =
            (Prog.get_proc_exn prog.prog proc_name).proc_internal
          in
          let symb_exec_proc =
            symb_exec_proc x pid v_args j params args eval_state
          in
          let spec_exec_proc () =
            match spec with
            | Some spec ->
                exec_with_spec spec x j args pid subst symb_exec_proc eval_state
            | None -> exec_without_spec pid symb_exec_proc eval_state
          in
          match Exec_mode.biabduction_exec !Config.current_exec_mode with
          | true -> (
              match
                ( pid = caller,
                  is_internal_proc pid,
                  Call_stack.recursive_depth cs pid >= !Config.bi_unroll_depth
                )
              with
              (* In bi-abduction, reached max depth of recursive calls *)
              | _, _, true -> []
              (* In bi-abduction, recursive call *)
              | true, false, _ -> symb_exec_proc ()
                 | true, false, false
                   when List.length
                          (List.filter is_internal_proc (Call_stack.get_cur_procs cs))
                        < !Config.bi_no_spec_depth -> symb_exec_proc () *)
              | _ -> spec_exec_proc ())
          | false -> spec_exec_proc ()
```

Listing A.38: Extracting the procedure identifier and arguments for a function call

```
let par_f fcs eval_state =
  let { prog; state; _ } = eval_state in
  let fcs = List.map (fun (var_name, pid, v_args, bindings) ->
    let pid = get_pid_or_error pid state in
    let spec, params = get_spec_and_params prog pid state in
    let args = build_args v_args params in
    match spec with
    | Some spec -> (spec, var_name, pid, args, bindings)
    | None -> failwith "Found function without specification in a parallel call")
    fcs
  in exec_par_with_spec fcs eval_state
```

Listing A.39: `eval_par_proc_call` calls `par_f`, which extracts the specification

```
let exec_par_with_spec fcs eval_state =
  let { state; i; b_counter; cs; branch_path; _ } = eval_state in
  let process_ret = process_ret "par" None eval_state in
  let fcs = List.map (fun (spec, x, pid, args, subst) ->
    let subst = eval_subst_list state subst in
    L.verbose (fun fmt -> fmt "ABOUT TO USE THE SPEC OF %s" pid);
    (spec, x, args, subst)) fcs
  in let rets : ((State.t * Flag.t) list, state_err_t list) result =
    State.run_par_spec fcs state
  in match rets with
  | Ok rets -> ( L.verbose (fun fmt ->
        fmt "Run_par_spec returned %d Results" (List.length rets));
    let b_counter = if List.length rets > 1 then b_counter + 1 else b_counter
    in match rets with
    | (ret_state, fl) :: rest_rets ->
        let others = List.map (fun (ret_state, fl) ->
          process_ret false ret_state fl b_counter None) rest_rets
          in process_ret true ret_state fl b_counter (Some others) :: others
    | _ -> L.fail (Format.asprintf "ERROR: Unable to use specification of a \
            function in parallel composition"))
  | Error errors -> let errors = errors |> List.map (fun e -> Exec_err.ESt e) in
    [ CConf.ConfErr { cs; proc_idx = i; state; errors; branch_path} ]
```

Listing A.40: `exec_par_with_spec` runs the specification in the `PState` and processes the results

```
let exec_with_spec spec x j args pid subst symb_exec_proc eval_state =
  let { state; i; b_counter; cs; branch_path; _ } = eval_state in
  let process_ret = process_ret pid j eval_state in
  match !symb_exec_next with
  | true ->
     symb_exec_next := false;
```

```
      symb_exec_proc ()
  | false -> (
    let subst = eval_subst_list state subst in
    L.verbose (fun fmt -> fmt "ABOUT TO USE THE SPEC OF %s" pid);
    let rets : ((State.t * Flag.t) list, state_err_t list) result =
      State.run_spec spec state x args subst
    in match rets with
    | Ok rets -> ( L.verbose (fun fmt ->
          fmt "Run_spec returned %d Results" (List.length rets));
      let b_counter = if List.length rets > 1 then b_counter + 1 else b_counter
      in match rets with
      | (ret_state, fl) :: rest_rets ->
        let others = List.map (fun (ret_state, fl) ->
            process_ret false ret_state fl b_counter None) rest_rets
        in process_ret true ret_state fl b_counter (Some others) :: others
      (* Run spec returned no results *)
      | _ -> (
        match spec.data.spec_incomplete with
        | true -> L.normal (fun fmt -> fmt "Proceeding with symbolic execution.");
            symb_exec_proc ()
        | false -> L.fail (Format.asprintf "ERROR: Unable to use specification of \
                    function %s" spec.data.spec_name)))
  | Error errors ->
    let errors = errors |> List.map (fun e -> Exec_err.ESt e) in
    [ ConfErr { callstack = cs; proc_idx = i; error_state = state; errors; branch_path } ])
```

Listing A.41: Executing a single function call with specifications

```
let rec run_par_spec_aux fcs astate =
match fcs with
  | [] -> Ok []
  | (existential_bindings, name, params, up, x, args) :: rest -> (
    L.verbose (fun m -> m "INSIDE RUN spec of %s with the following UP:@\n%a@\n"
        name UP.pp up);
    let old_store = get_store astate in
    let new_store = try Store.init (List.combine params args)
      with Invalid_argument _ ->
        let message = Fmt.str "Running spec of %s which takes %i parameters \
            with the following %i arguments : %a" name (List.length params)
            (List.length args) (Fmt.Dump.list Val.pp) args
        in raise (Invalid_argument message)
    in
    let astate' = set_store astate new_store in
    let existential_bindings = Option.value ~default:[] existential_bindings
    in
    let existential_bindings =
      List.map (fun (x, v) -> (Expr.LVar x, v)) existential_bindings
    in
    let store_bindings = Store.bindings new_store in
    let store_bindings = List.map (fun (x, v) -> (Expr.PVar x, v)) store_bindings
    in
    let subst = ESubst.init (existential_bindings @ store_bindings) in
    L.verbose (fun m -> m "About to use the spec of %s with the following \
            UP:@\n%a@\n" name UP.pp up);
    match SUnifier.unify astate' subst up FunctionCall with
    | Ok rets ->
        let open Syntaxes.Result in
        let ( let++ ) x f = List.concat_map f x in
        let lambda acc (frame_state, subst, posts) =
          let* acc = acc in
          let frame_state = set_store frame_state (Store.copy old_store) in
          let* rets = run_par_spec_aux rest frame_state in
          let frames =
            match rets with
            | [] -> [ frame_state ]
            | l -> List.map fst l
          in
          L.verbose (fun m -> m "Returned from recursive call with length %s"
            (string_of_int @@ List.length rets));
          let fl, posts =
            match posts with
            | Some (fl, posts) -> (fl, posts)
            | _ -> let msg = Printf.sprintf "SYNTAX ERROR: Spec of %s \
                    does not have a postcondition" name
              in
              L.normal (fun m -> m "%s" msg);
              raise (Failure msg)
          in
          (* OK FOR DELAY ENTAILMENT *)
          let res =
            let++ frame_state = frames in
            let frame_store = get_store frame_state in
            let frame_state =
              set_store frame_state (Store.copy new_store)
            in
            let++ final_state =
              SUnifier.produce_posts frame_state subst posts
            in
            let final_store = get_store final_state in
            let v_ret = Store.get final_store Names.return_variable in
            let final_state =
              set_store final_state (Store.copy frame_store)
            in
            let v_ret = Option.value
                ~default:(Option.get (Val.from_expr (Lit Undefined)))
                v_ret
            in
            let final_state = update_store final_state x v_ret in
            let _, final_states = simplify ~unification:true final_state in
            List.map
              (fun final_state ->
                ( snd
                    (Option.get
                       (SUnifier.unfold_concrete_preds final_state)),
                  fl ))
              final_states
          in Ok (acc @ res)
        in List.fold_left lambda (Ok []) rets
    | Error errs -> let msg = Fmt.str "WARNING: Failed to unify against \
            the precondition of procedure %s" name
        in
        L.normal (fun m -> m "%s" msg);
        Error errs)
```

Listing A.42: Running the specification for a par call

```
let run_spec_aux
  ?(existential_bindings : (string * vt) list option)
  (name : string)
  (params : string list)
  (up : UP.t)
  (astate : t)
  (x : string option)
  (args : vt list) : ((t * Flag.t) list, SUnifier.err_t list) result =
L.verbose (fun m ->
    m "INSIDE RUN spec of %s with the following UP:@\n%a@\n" name UP.pp up);
let old_store = get_store astate in
let new_store = try Store.init (List.combine params args)
  with Invalid_argument _ -> let message = Fmt.str "Running spec of %s \
      which takes %i parameters with the following %i arguments : %a"
      name (List.length params) (List.length args) (Fmt.Dump.list Val.pp) args
    in raise (Invalid_argument message)
in
let astate' = set_store astate new_store in
let existential_bindings = Option.value ~default:[] existential_bindings in
let existential_bindings =
  List.map (fun (x, v) -> (Expr.LVar x, v)) existential_bindings
in
let store_bindings = Store.bindings new_store in
let store_bindings = List.map (fun (x, v) -> (Expr.PVar x, v)) store_bindings
in
let subst = ESubst.init (existential_bindings @ store_bindings) in
L.verbose (fun m -> m "About to use the spec of %s with the following
    UP:@\n%a@\n" name UP.pp up);
match SUnifier.unify astate' subst up FunctionCall with
| Ok rets ->
    let acceptable =
      match !Config.Verification.exact with
      | false -> true
      | true ->  let check_variant (state : t) (subst : st) =
        let bstate, _, _, variants = state in
        let variant = Hashtbl.find_opt variants name in
        match variant with
        | None -> true
        | Some None -> failwith ("Error: " ^ name^ ": Recursive call in exact
            verification without a variant")
        | Some (Some variant) -> (
            let new_variant = ESubst.subst_in_expr_opt subst variant in
            let () = L.verbose (fun fmt -> fmt "New variant: %a"
                (Fmt.Dump.option Expr.pp) new_variant)
            in
            match new_variant with
            | None -> failwith ("ERROR: " ^ name ^ ": recursive call
                variant substitution failure")
            | Some new_variant -> (
                let variant_check = Expr.UnOp
                    (UnOp.UNot, BinOp (new_variant, ILessThan, variant))
                in
                let variant_check = Val.from_expr variant_check in
                match variant_check with
                | None -> failwith ("ERROR: " ^ name ^ ": variant could
                    not be converted to a value")
                | Some variant_check ->
```

```
                  not (State.sat_check bstate variant_check)))
       in List.for_all
         (fun (state, subst, _) -> check_variant state subst)
         rets
     in
     if acceptable then
       Ok
         ((* Successful Unification *)
         let ( let++ ) x f = List.concat_map f x in
         let++ frame_state, subst, posts = rets in
         let fl, posts =
           match posts with
           | Some (fl, posts) -> (fl, posts)
           | _ -> let msg = Printf.sprintf "SYNTAX ERROR: Spec of %s does \
                not have a postcondition" name
               in
               L.normal (fun m -> m "%s" msg);
               raise (Failure msg)
         in
         (* OK FOR DELAY ENTAILMENT *)
         let++ final_state =
           SUnifier.produce_posts frame_state subst posts
         in
         let final_store = get_store final_state in
         let v_ret = Store.get final_store Names.return_variable in
         let final_state = set_store final_state (Store.copy old_store) in
         let v_ret = Option.value
             ~default:(Option.get (Val.from_expr (Lit Undefined)))
             v_ret
         in
         let final_state = update_store final_state x v_ret in
         let _, final_states = simplify ~unification:true final_state in
         List.map (fun final_state ->
             ( snd (Option.get (SUnifier.unfold_concrete_preds final_state)),
               fl ))
           final_states)
     else
       let err = [] in
       Error err
   | Error errs -> let msg = Fmt.str "WARNING: Failed to unify against \
         the precondition of procedure %s" name
     in
     L.normal (fun m -> m "%s" msg);
     Error errs
```

Listing A.43: Running the specification for a single function call

```
{ emp }
function use_float_ops() {
    x := 5.5;
    y := 2.0;

    r := new(5);
    [r] := x f+ y;
    [r + 1] := x f- y;
    [r + 2] := x f* y;
    [r + 3] := x f/ y;
    [r + 4] := x f% y;
    return r
}
{ ret -> 7.5, 3.5, 11.0, 2.75, 1.5 }
```

Listing A.44: Floating point operations in WISL

```
{ emp }
function conc_disjoint_writes() {
    x := new(3);
    par {
        u := write(x, 1);
        u := write(x + 1, 2);
        u := write(x + 2, 3)
    };
    return x
}
{ ret -> 1, 2, 3 }
```

Listing A.45: Concurrent disjoint writes

```
CMD: par [u := "write"(x, 1i); u := "write"(gvar0, 2i); u := "write"(gvar1,
                                                                      3i)]
LOC: ./wislfp/tests/simple_concurrency.wisl:64:4
PROCS: [conc_disjoint_writes]
LOOPS: [] ++ []
BRANCHING: 0

SPEC VARS:
STORE:
  (gvar0: {{ ALoc _$l_8, Lit 1i }})
  (gvar1: {{ ALoc _$l_8, Lit 2i }})
  (x: {{ ALoc _$l_8, Lit 0i }})

MEMORY:
  _$l_8 -> BOUND: Some (3, 1.)
           {0i: null [1.]
            1i: null [1.]
            2i: null [1.]}
```

Listing A.46: Logs before executing par block

```
Unifier.unify: Success

STATE:
    MEMORY:
    _$l_8 -> BOUND: Some (3, 1.)
             {1i: null [1.]
              2i: null [1.]}
```

Listing A.47: State after unifying the pre-condition of the first function

```
Unifier.unify: Success

STATE:
    MEMORY:
    _$l_8 -> BOUND: Some (3, 1.)
            {2i: null [1.]}
```

Listing A.48: State after unifying the pre-condition of the second function

```
Unifier.unify: Success

STATE:
    MEMORY:
    _$l_8 -> BOUND: Some (3, 1.)
            {}
```

Listing A.49: State after unifying the pre-condition of the third function

```
STATE BEFORE SIMPLIFICATIONS:
STORE:
  (u: Lit 0i)
  (x: {{ ALoc _$l_8, Lit 0i }})

MEMORY:
  _$l_8 -> BOUND: Some (3, 1.)
           {}

  _$l_9 -> BOUND: None
           {_lvar_13: 3i [1.]}

Filtered and fixed subst, to be applied to memory:
[ (_lvar_12: _$l_8),
                                                       (_lvar_13: 2i),
                                                       (_$l_9: _$l_8) ]
```

Listing A.50: State after unifying the post-condition of the third function

```
STATE BEFORE SIMPLIFICATIONS:
STORE:
  (u: Lit 0i)
  (x: {{ ALoc _$l_8, Lit 0i }})

MEMORY:
  _$l_10 -> BOUND: None
            {_lvar_15: 2i [1.]}

  _$l_8 -> BOUND: Some (3, 1.)
           {2i: 3i [1.]}

Filtered and fixed subst, to be applied to memory:
[ (_lvar_14: _$l_8),
                                                       (_lvar_15: 1i),
                                                       (_$l_10: _$l_8) ]
```

Listing A.51: State after unifying the post-condition of the second function

```
STATE BEFORE SIMPLIFICATIONS:
STORE:
  (u: Lit 0i)
  (x: {{ ALoc _$l_8, Lit 0i }})

MEMORY:
  _$l_8 -> BOUND: Some (3, 1.)
          {1i: 2i [1.]
           2i: 3i [1.]}

  _$l_11 -> BOUND: None
           {_lvar_17: 1i [1.]}

Filtered and fixed subst, to be applied to memory:
[ (_lvar_16: _$l_8),
                                                    (_lvar_17: 0i),
                                                    (_$l_11: _$l_8) ]
```

Listing A.52: State after unifying the post-condition of the first function

```
STATE AFTER SIMPLIFICATIONS:
STORE:
  (gvar0: {{ ALoc _$l_8, Lit 1i }})
  (gvar1: {{ ALoc _$l_8, Lit 2i }})
  (u: Lit 0i)
  (x: {{ ALoc _$l_8, Lit 0i }})

MEMORY:
  _$l_8 -> BOUND: Some (3, 1.)
          {0i: 1i [1.]
           1i: 2i [1.]
           2i: 3i [1.]}
```

Listing A.53: Final state

```
predicate is_int(+x: Int) {
    emp
}

{ (x == #x) * (p == #p) * (w == #w) * is_int(#w) * is_int(#v)
    * (#p f<=# 1.0) * (#x -> (#p: #v)) }
function read_and_add(x, p, w) {
    n := [x];
    m := n + w;
    return m
}
{ (#x -> (#p: #v)) * (ret == #v + #w) }
```

Listing A.54: The specification of the read_and_add function

```
--conc_reads_same_loc: 1--
CMD: par [fst := "read_and_add"(x, 0.5, 5i); snd := "read_and_add"(x, 0.5, 10i)]

STORE:
  (x: LVar #x)
  (y: {{ ALoc _$l_5, Lit 0i }})

MEMORY:
  _$l_5 -> BOUND: Some (2, 1.)
           {0i: null [1.]
            1i: null [1.]}

  _$l_1 -> BOUND: None
           {#wisl__1: #v [1.]}
```

Listing A.55: State before the par call - concurrent reads on the same location

```
Unifier.unify: Success
STATE:
  MEMORY:
    _$l_5 -> BOUND: Some (2, 1.)
             {0i: null [1.]
              1i: null [1.]}

    _$l_1 -> BOUND: None
             {#wisl__1: #v [0.5]}
```

Listing A.56: State after unifying the pre-condition of the first function

```
Unifier.unify: Success
STATE:
    MEMORY:
      _$l_5 -> BOUND: Some (2, 1.)
               {0i: null [1.]
                1i: null [1.]}

      _$l_1 -> BOUND: None
               {}
```

Listing A.57: State after unifying the pre-condition of the second function

```
predicate list_member(+vs, +v, r : Bool){
  (vs == []) * (r == false);
  (vs == v :: #rest) * (r == true);
  (vs == #v :: #rest) * (! (#v == v)) * list_member(#rest, v, r)
}
lemma list_member_concat {
  statement:
    forall vs1, vs2, v.
      list_member(vs1, v, #r1) * list_member(vs2, v, #r2) |-
        list_member(vs1 @ vs2, v, (#r1 || #r2))

  proof:
    unfold list_member(vs1, v, #r1);
    if (not (vs1 = [])) {
      if (hd vs1 = v) {
        fold list_member(vs1 @ vs2, v, true)
      } else {
        assert {bind: #nv1, #nvs1, #nr1} (vs1 == #nv1 :: #nvs1)
          * list_member(#nvs1, v, #nr1);
        apply list_member_concat(#nvs1, vs2, v)
      }
    }
}
```

Listing A.58: List membership predicate and concatenation lemma

```
{ (t == #t) * (v == #v) * (w == #w) * binary_tree(#t, #tree, 1.0) }
function replace(t, v, w) {
    if (t = null) {
        skip
    } else {
        x := [t];
        if (x = v) {
            [t] := w
        } else {
            skip
        };
        lt := [t + 1];
        rt := [t + 2];
        [[assert {bind: #left_tree, #right_tree} binary_tree(lt, #left_tree, 1.0)
            * binary_tree(rt, #right_tree, 1.0)]];
        par {
            u := replace(lt, v, w);
            u := replace(rt, v, w)
        };
        [[assert {bind: #left_r_tree, #right_r_tree} binary_tree(lt, #left_r_tree, 1.0)
            * binary_tree(rt, #right_r_tree, 1.0)]];
        [[apply replaced_concat(#left_tree, #left_r_tree,
            #right_tree, #right_r_tree, #v, #w)]]
    };
    return 0
}
{ binary_tree(#t, #r_tree, 1.0) * replaced(#tree, #r_tree, #v, #w) }
```

Listing A.59: A function that replaces element `v` in binary tree `t` with `w`

```
predicate replaced(+vs, +ws, +v, +w) {
    (vs == []) * (ws == []);
    (vs == v :: #restvs) * (ws == w :: #restws)
        * replaced(#restvs, #restws, v, w);
    (vs == #v :: #restvs) * (ws == #v :: #restws) * (! (#v == v))
        * replaced(#restvs, #restws, v, w)
}

lemma replaced_concat {
    statement:
      forall vs1, ws1, vs2, ws2, v, w.
        replaced(vs1, ws1, v, w) * replaced(vs2, ws2, v, w) |-
            replaced(vs1 @ vs2, ws1 @ ws2, v, w)

    proof:
      unfold replaced(vs1, ws1, v, w);
      if (not (vs1 = [])) {
        assert {bind: #nv, #nw, #nvs1, #nws1} (vs1 == #nv :: #nvs1)
            * (ws1 == #nw :: #nws1) * replaced(#nvs1, #nws1, v, w);
        apply replaced_concat(#nvs1, #nws1, vs2, ws2, v, w)
      }
}
```

Listing A.60: The `replaced` predicate and its corresponding concatenation lemma

```
[spec copy_spec: #p]
{ (x == #x) * lambda_term(#x, #v, #t1, #t2, #type, #p) }
function copy(x) {
    type := [x];
    if (type = 2) {
        r := new(2);
        v := [x + 1];
        [r + 1] := v
    } else {
        r := new(3);
        if (type = 1) {
            t1 := [x + 1];
            t2 := [x + 2];
            par {
                r1 := copy(t1) [copy_spec: (#p: #p)];
                r2 := copy(t2) [copy_spec: (#p: #p)]
            };
            [r + 1] := r1;
            [r + 2] := r2
        } else {
            v := [x + 1];
            t1 := [x + 2];
            [r + 1] := v;
            r1 := copy(t1) [copy_spec: (#p: #p)];
            [r + 2] := r1
        }
    };
    [r] := type;
    return r
}
{ is_copy(#x, ret, #p, 1.0, #v, #t1, #t2, #type) }
```

Listing A.61: Lambda term copying

```
lemma split_lambda_term_permission {
  statement:
    forall x, v, t1, t2, type, p, p1, p2.
      (p == p1 f+ p2) * (p1 f># 0.0) * (p2 f># 0.0)
        * lambda_term(x, v, t1, t2, type, p) |-
        lambda_term(x, v, t1, t2, type, p1) * lambda_term(x, v, t1, t2, type, p2)

  proof:
    unfold lambda_term(x, v, t1, t2, type, p);
    if (not (type = 2)) {
      if (type = 1) {
        assert {bind: #v1, #t11, #t12, #type1, #v2, #t21, #t22, #type2}
          lambda_term(t1, #v1, #t11, #t12, #type1, p)
          * lambda_term(t2, #v2, #t21, #t22, #type2, p);
        apply split_lambda_term_permission(t1, #v1, #t11, #t12, #type1, p, p1, p2);
        apply split_lambda_term_permission(t2, #v2, #t21, #t22, #type2, p, p1, p2)
      } else {
        assert {bind: #v1, #t11, #t12, #type1}
          lambda_term(t1, #v1, #t11, #t12, #type1, p);
        apply split_lambda_term_permission(t1, #v1, #t11, #t12, #type1, p, p1, p2)
      };
      fold lambda_term(x, v, t1, t2, type, p1);
      fold lambda_term(x, v, t1, t2, type, p2)
    }
}
```

Listing A.62: Splitting permission lemma

```
predicate SLL(+x, vs) {
  (x == null) * (vs == []);
  (x -b> #v, #next) * SLL(#next, #vs) *
  (vs == #v :: #vs)
}

// 06. Calculating the length of a given SLL
{ (x == #x) *  SLL(#x, #vs) }
function SLL_length(x) {
  n := 0;
  if (x = null){
    n := 0
  } else {
    t := [x + 1];
    n := SLL_length(t);
    n := 1 + n
  };
  return n
}
{ ret == len(#vs) }

// 06. Calculating the length of a given SLL
{ (x == #x) * SLL(x, #vx) }
function SLL_length_iter(x) {
  y := x;
  n := 0;
  [[invariant {bind: n, y, #nvx,  #nvy}
      SLLseg(x, y,  #nvx) * SLL(y, #nvy) *
         (#vx == (#nvx@#nvy)) * (n == len #nvx) ]];
  while (not (y = null)) {
    [[ assert {bind: #y} y == #y ]];
    [[ assert {bind: #v, #z} #y -b> #v, #z ]];
    y := [y+1];
    n := n+1;
    [[ apply SSLseg_append(x, #nvx, #v, y) ]]
  };
  [[ unfold SLL(null, #nvy)]];
  [[ apply SLLseg_to_SLL(x) ]];
  return n
}
{ SLL(#x, #vx) * (ret == len(#vx)) }
```

Listing A.63: Extract of the `SLL_ex_complete.wisl` regression test suite

```
Obtained 30 symbolic tests in total
Running symbolic tests: 0.047551
Verifying lemma SLLseg_concat_SLL... Success
...
Verifying one spec of procedure SLL_concat_iter_loop0... s s Success
All specs succeeded: 0.439783
```

Listing A.64: Results of verifying `SLL_ex_complete.wisl` with the old WISL instantiation

```
PASS: examples/busywaiting/clhlock$ verifast -c clhlock.c (0.11s)
PASS: examples/busywaiting/ioliveness$ verifast -c echo.c (0.03s)
PASS: examples/busywaiting/ioliveness$ verifast -c echo_live.c (0.03s)
PASS: examples/busywaiting/ioliveness$ verifast -c echo_live_mt.c (0.07s)
PASS: examples/busywaiting$ verifast -c await.c (0.05s)
PASS: examples/jayanti$ verifast threading.o popl20_prophecies.o atomics.c
    jayanti.c client.c (23.86s)
PASS: examples/monitors$ verifast -prover z3v4.5 -disable_overflow_check -shared
    ghost_counters.c  (0.17s)
PASS: examples/monitors$ verifast -prover z3v4.5 -disable_overflow_check -shared
    ghost_counters.c monitors.o queues.o buffer.c  (0.32s)
PASS: examples/monitors$ verifast -prover z3v4.5 -disable_overflow_check -shared
    ghost_counters.c monitors.o queues.o bounded_buffer.c  (1.30s)
PASS: examples/monitors$ verifast -prover z3v4.5 -disable_overflow_check -shared
    monitors.o barrier.c  (0.13s)
PASS: examples/monitors$ verifast -prover z3v4.5 -disable_overflow_check -shared
    ghost_counters.c monitors.o barbershop.c  (1.49s)
PASS: examples/monitors$ verifast -prover z3v4.5 -disable_overflow_check -shared
    ghost_counters.c monitors.o readers_writers.c  (0.58s)
```

Listing A.65: Running part of the concurrency-related test-suite of Verifast

Bibliography

[1] Le Lann G. An analysis of the Ariane 5 flight 501 failure-a system engineering perspective. In: Proceedings International Conference and Workshop on Engineering of Computer-Based Systems; 1997. p. 339-46.

[2] Lynch J. The Worst Computer Bugs in History: The Ariane 5 Disaster; 2017. https://www.bugsnag.com/blog/bug-day-ariane-5-disaster.

[3] Leveson NG. Medical Devices: the Therac-25; 1985. .

[4] King JC. Symbolic Execution and Program Testing. Commun ACM. 1976 jul;19(7):385–394. Available from: https://doi.org/10.1145/360248.360252.

[5] Cadar C, Godefroid P, Khurshid S, Pasareanu CS, Sen K, Tillmann N, et al. Symbolic execution for software testing in practice: preliminary assessment. 2011 33rd International Conference on Software Engineering (ICSE). 2011:1066-71.

[6] Jacobs B, Smans J, Philippaerts P, Vogels F, Penninckx W, Piessens F. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In: Proceedings of the Third International Conference on NASA Formal Methods. NFM'11. Berlin, Heidelberg: Springer-Verlag; 2011. p. 41–55.

[7] Bart Jacobs, Frank Piessens, et al . VeriFast source code;. https://github.com/verifast/verifast [Accessed 2023/01/24].

[8] Calcagno C, Distefano D. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In: Bobaru M, Havelund K, Holzmann GJ, Joshi R, editors. NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18–20, 2011, Proceedings. vol. 6617 of Lecture Notes in Computer Science. Springer; 2011. p. 459-65.

[9] The Infer Team. About Infer; 2017. https://fbinfer.com/docs/about-Infer [Accessed 2023/01/23].

[10] Fragoso Santos J, Maksimović P, Ayoun SE, Gardner P. Gillian, Part i: A Multi-Language Platform for Symbolic Execution. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2020. New York, NY, USA: Association for Computing Machinery; 2020. p. 927–942. Available from: https://doi.org/10.1145/3385412.3386014.

[11] Maksimović P, Ayoun SE, Santos JF, Gardner P. Gillian, Part II: Real-World Verification for JavaScript and C. In: Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II. Berlin, Heidelberg: Springer-Verlag; 2021. p. 827–850. Available from: https://doi.org/10.1007/978-3-030-81688-9_38.

[12] Maksimović P, Santos JF, Ayoun SÉ, Gardner P. Gillian: A Multi-Language Platform for Unified Symbolic Analysis. ArXiv. 2021;abs/2105.14769.

[13] Santos JF, Maksimović P, Ayoun SE, Gardner P. Gillian: Compositional Symbolic Execution for All; 2020.

[14] Azalea Raad. The Theory and Practice of Concurrent Programming; 2021. Imperial College London.

[15] Blom S, Darabi S, Huisman M, Oortwijn W. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In: Polikarpova N, Schneider S, editors. Integrated Formal Methods. Cham: Springer International Publishing; 2017. p. 102-10.

[16] Hoare CAR. An Axiomatic Basis for Computer Programming. Commun ACM. 1969 oct;12(10):576–580. Available from: https://doi.org/10.1145/363235.363259.

[17] Philippa Gardner. Scalable Software Verification; 2022. Imperial College London.

[18] O'Hearn PW, Reynolds JC, Yang H. Local Reasoning about Programs that Alter Data Structures. In: Annual Conference for Computer Science Logic; 2001. .

[19] O'Hearn PW, Pym DJ. The Logic of Bunched Implications. Bulletin of Symbolic Logic. 1999;5:215 244.

[20] The Infer Team. Infer - Separating logic and bi-abduction;. https://fbinfer.com/docs/separation-logic-and-bi-abduction [Accessed 2023/01/24].

[21] O'Hearn PW. Resources, Concurrency, and Local Reasoning. Theorertical Computer Science. 2007 apr;375(1–3):271–307. Available from: https://doi.org/10.1016/j.tcs.2006.12.035.

[22] Hoare CAR. In: Hansen PB, editor. Towards a Theory of Parallel Programming. New York, NY: Springer New York; 2002. p. 231-44. Available from: https://doi.org/10.1007/978-1-4757-3472-0_6.

[23] Owicki SS, Gries D. Verifying properties of parallel programs: an axiomatic approach. Commun ACM. 1976;19:279-85.

[24] Brookes S. A semantics for concurrent separation logic. Theoretical Computer Science. 2007;375(1):227-70. Festschrift for John C. Reynolds's 70th birthday. Available from: https://www.sciencedirect.com/science/article/pii/S0304397506009248.

[25] Boyland J. In: Clarke D, Noble J, Wrigstad T, editors. Fractional Permissions. Berlin, Heidelberg: Springer Berlin Heidelberg; 2013. p. 270-88. Available from: https://doi.org/10.1007/978-3-642-36946-9_10.

[26] O'Hearn PW, Power AJ, Takeyama M, Tennent RD. Syntactic control of interference revisited. Theoretical Computer Science. 1999;228(1):211-52. Available from: https://www.sciencedirect.com/science/article/pii/S0304397598003594.

[27] Bornat R, Calcagno C, O'Hearn P, Parkinson M. Permission Accounting in Separation Logic. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '05. New York, NY, USA: Association for Computing Machinery; 2005. p. 259–270. Available from: https://doi.org/10.1145/1040305.1040327.

[28] Boyland J. Checking Interference with Fractional Permissions. In: Cousot R, editor. Static Analysis. Berlin, Heidelberg: Springer Berlin Heidelberg; 2003. p. 55-72.

[29] François Pottier, Yann Régis-Gianas. Menhir Parser Generator Library Home Page;. http://gallium.inria.fr/~fpottier/menhir/ [Accessed 2023/06/12].

[30] Stefan Blom, Saeed Darabi, et al . VerCors source code;. https://github.com/utwente-fmt/vercors [Accessed 2023/06/12].