KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Modular Semi-automatic Formal Verification of Critical Systems Software

Willem Penninckx

Supervisor:
Prof. dr. B. Jacobs
Prof. dr. ir. F. Piessens, co-supervisor

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

September 2017

Modular Semi-automatic Formal Verification of Critical Systems Software

Willem PENNINCKX

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

September 2017

Preface

This PhD was a challenging project. I was lucky to do this project in a good environment.

I thank Bart Jacobs for being my supervisor. The quality of his feedback and his accuracy are phenomenal. I thank Frank Piessens for being my co-supervisor. Knowing that someone is around with such an exceptional skill in quickly pinpointing the exact problem is reassuring.

I thank the chair, the members of my examination committee and the members and former member of my supervisory committee: Adhemar Bultheel, Dave Clarke, Bart Demoen, Marc Denecker, Marieke Huisman, and Eric Steegmans.

I thank the head of the research group: Wouter Joosen. I thank the members of Project Office: Katrien, Ghita, and Annick. I thank the experts in administration and communication: Denise, Liesbet, Nina, Fred, Karin, Margot, Ann, Anne-Sophie, Esther, Marleen, Karen S., Inge, and Karen V. I thank the experts of the computer infrastructure and networks: Anita, Jean, Bart S., Greg, Kris, and Steven W. I also thank everyone else who take care of administrative, managemental, practical, infrastructural or other activities such that research and education can happen. I would also like to thank the many people who, often silently, do administrative, managemental, practical, infrastructural or other such work besides their official activity.

I thank my office mates, former office mates, and office visitor: Jafar, Mahmoud, Amin, Gijs, Malte, Wilfried, Javier, and Adriaan. The office was always a nice place to work. I will miss the philosophical discussions and the games.

As a researcher one meets many interesting people. I will remember many people for good cooperation on didactical activities, discussing software verification, discussing life, being supportive, and their very colorful personality. Besides everyone mentioned above, I also thank Pieter A., Afshin, Willem D.G., Jesper, Kristof, Lieven, Dominique, Tung, Phil, Mario, Emad, Thomas H., Tom H., Georgios, Roald, Jef, Jonathan, Jan Tobias, Job, Mads, Marco, José, Zubair, Bob, Jan Smans, Klaas, Thomas v.B., Jo, Marko, Rinde, Alex vdB., Mathy, Dries, Frédéric, Neline, Tom V.G., and Fan.

I thank many students. Seeing them experiment with their first Python programs brings back good memories. They make the university more lively.

I thank Anneleen D.B., Irma K., Ingemar S., Aad, and Kevin Vh. for their support when I was ill.

Besides the classmates mentioned above, I thank Karel, Erika, Wouter, Jo, and Linard.

I thank Pieter W., my father, and everyone else who was involved in the early days when I got interested in software.

Special thanks go to Pieter & Nele, and my mother.

This work would not have been possible without funding. This work received funding from: Interuniversity Attraction Poles Programme Belgian State (Belgian Science Policy), Research Fund KU Leuven, EU FP7 project SecureChange, EU FP7 project NESSoS, Microsoft Research Cambridge as part of the Verified Software Initiative, EU FP7 FET-Open project ADVENT under grant number 308830, and the European Union’s Horizon 2020 research and innovation programme under grant agreement No 731453 (project VESSEDIA).

Abstract

In the first part of this thesis, we present a case study on successfully verifying the Linux USB BP keyboard driver. Our verification approach is *(a)* sound, *(b)* takes into account dynamic memory allocation, complex API rules and concurrency, and *(c)* is applied on a real kernel driver which was not written with verification in mind. We employ VeriFast, a software verifier based on separation logic. Besides showing that it is possible to verify this device driver, we identify the parts where the verification went smoothly and the parts where the verification approach requires further research to be carried out.

In the second part of this thesis, we present a program verification approach that uses an input/output style of reasoning. It can be applied both to programs that perform input/output, and programs that do not but instead manipulate memory. The approach is sound, modular, compositional (I/O actions can be defined on top of other actions) and supports concurrency. It uses Petri nets and separation logic. We have implemented the approach, both for programs that do and do not perform I/O, in the VeriFast verifier and sketched how it can be implemented in the Iris framework for programs that perform I/O.

Beknopte samenvatting

In het eerste deel van deze thesis voeren we een case study uit waarin we het USB BP toetsenbordstuurprogramma van Linux verifiëren. Onze verificatieaanpak is *(a)* sound, *(b)* ondersteunt dynamische geheugentoewijzing, complexe API regels en concurrency, en *(c)* is toegepast op een echt kernelstuurprogramma waarbij de makers van dat stuurprogramma geen rekening hielden met verificatie toen dat stuurprogramma geschreven werd. We gebruiken VeriFast, een programma voor programmaverificatie gebaseerd op separation logic. We tonen niet enkel aan dat het mogelijk is om zulk een apparaatstuurprogramma te verifiëren, maar we identificeren ook de delen waar verificatie vlot verliep en die waarbij verder onderzoek welkom is om de verificatieaanpak te verbeteren.

In het tweede deel van de thesis stellen we een verificatieaanpak voor die gebruik maakt van een manier van redeneren gebaseerd op het concept van invoer/uitvoer. Deze verificatieaanpak kan zowel toegepast worden op programma's die invoer/uitvoer doen, en programma's die dat niet doen en in plaats daarvan geheugen aanpassen. De verificatieaanpak is sound, modulair, compositioneel (invoer-/uitvoeracties kunnen gedefinieerd worden bovenop andere acties) en ondersteunt concurrency. Het gebruikt Petri netten en separation logic. We hebben de aanpak geïmplementeerd, zowel voor programma's die wel en die geen invoer/uitvoer doen, in VeriFast en geschetst hoe de verificatieaanpak kan geïmplementeerd worden in het raamwerk Iris.

Contents

List of Figures

Chapter 1

Introduction

While we used to say software is becoming commonplace, we should say now that software is already almost everywhere. Not just your personal computer runs software. Electric bikes, airplanes, televisions, nuclear reactors, cars, cellphones, elevators, washing machines, ... they all run on software. Software is written by humans and therefore can contain bugs, i.e. errors in the software, somewhat similar to an inconsistency in the plot of a book or an error in a thesis text. Such bugs do not have to be a big problem for some applications, but for others it can have very serious consequences.

For example, it was confirmed that in 2004 an elevator contained a software bug that caused the elevator to descend without completely closing the doors [44, Sec. 5.1.1].

In 2015 the US Federal Aviation Administration issued a directive which explained that the latest Boeing airplane, the 787 Dreamliner, contained a software bug consisting of an integer overflow [22]. If the airplane (or rather its generators) are not manually reset or shutdown after 248 days, it loses electric power which could result in loss of control of the airplane. If this happens during e.g. take-off, this can result in a catastrophy [27].

Bugs can also cause or play a role in economical losses. For example, a blackout in the US in 2003 was estimated to cost between $7 and $10 billion [77]. The blackout was at least largely caused by a race condition[1] in the alarm-

[1]We say software has a race condition if the result of a calculation in that software depends on the unpredictable order of two parallel or interleaved executions of two inner pieces of software inside the software. If unintended, a race condition can lead to incorrect behavior of the software.

management system. Because of this, the operator did not receive updated information with which he could have prevented failures.

Also everyday security and privacy is at risk because of bugs. In 2014, critical bugs were found in all major TLS libraries [41], i.e. pieces of software (reused in other software) responsible for encrypting online connections such as for online banking and accessing e-mail. Microsoft's Schannel had a remote code execution vulnerability [43] allowing an attacker to run arbitrary code on a server. Apple's SecureTransport suffered from an incorrect (and misindented) "goto fail" [21] allowing an attacker to spoof a server. OpenSSL — the only one gaining widespread mainstream media attention for its flaw — contained a buffer overread [53]. Other libraries had flaws as well.

Traditional strategies to achieve a level of safety for a mechanical device cannot be applied to software [67]. For strategies for mechanical devices it is normal to focus on physical faults such as wear, corrosion, mechanical stress, etc. for which the failure rate can even be known because of historic use of the components. Software does not have physical faults but can still have (many) design faults.

To (try to) address such design faults, multiple strategies exist for multiple phases or activities of the development process, but we will only focus on errors in the implementation, and not on errors in e.g. requirements analysis. Such strategies include various testing strategies and code review strategies, and avoiding or forbidding dangerous patterns such as global variables, lengthy functions, and high complexity in code according to some metric.

Depending on the safety and reliability required and on how which strategies are used, this might or might not be good enough. Butler and Finelli [9] argue that achieving higher reliability through testing, requires spending more time on testing, in such a way that for ultra reliable systems, the amount of time one should spend on testing becomes infeasible.

Testing is the process of executing the program multiple times, and each time the program is executed, checking whether the execution contains an error. So testing searches incorrect executions in the set of all executions. The reason testing becomes infeasible, is because this set of all executions can be huge, because of large inputs, timings on how concurrent pieces of software are executed in parallel and/or interleaved, an unbounded execution length of the program, and a huge amount of memory the program can address. Testing is then like searching a needle in a haystack, even though it might not feel like this when one is happy to not find a needle, which does not imply there are none.

Luckily, there are approaches that consider all executions, namely *sound formal verification* approaches for software.

- *Verification* is the act of checking something, in this context properties about the software such as that there are no executions that crash.
- *Formal* means we apply mathematical methods.
- An approach for checking something is *sound* if it finds all violations of the property checked, contrary to *unsound* approaches which can miss some. Testing is typically unsound since it can miss errors. Note that while a sound approach is preferable as is, other factors come into play when choosing which approaches to use, such as the monetary cost of applying the approach, which can mean an unsound approach can be preferred above a sound approach in some situations.
- While usually implicit, we should mention that we only consider *static* verification approaches, which means that the approach analyses the program itself without executing it. This contrasts with *dynamic* approaches that perform checks during execution of the program. A dynamic approach can detect an error during execution, while a static approach can detect the error while the program has never been executed at all. Similar to unsound approaches, dynamic approaches can be very valuable too.

In this thesis we focus on a (family of) sound formal verification approach(es) called Hoare logic(s). When using Hoare logic [31] one writes two logic formulas that constrain the program state: a precondition and a postcondition. The program is considered correct if for every state for which the precondition holds, and any program execution starting from such a state, the program does not crash and the postcondition is true for the state obtained at the end of the execution.

Separation logic [68, 49] is an extension of Hoare logic that adds support for easier verification of programs that have pointer manipulation and aliasing. Aliasing is the situation where two different variables in the program point to the same memory cell [12]. Separation logic deals with aliasing by allowing formulas to contain subformulas that describe separate parts of the memory state. This also allows local reasoning [50] by only describing the relevant parts of memory in the preconditions and postconditions, in contrast to describing the whole memory footprint of the whole program.

We should mention separation logic is not the first approach to support pointer manipulating programs, but as Bornat et al. points out, it "[makes] earlier attempts to prove pointer-mutating programs [...] look ridiculously complicated and ad-hoc"[7].

The observation that separation logic supports local reasoning, turns out to not only be applicable to reasoning about data, but also to reasoning about threads: Concurrent separation logic [51] makes the observation that in order to verify a parallel program consisting of two threads one can formulate an invariant that "describes" the shared data that is read and written by the threads in critical sections, and verify each thread independently. The resulting postcondition is then just the combination of the postconditions of the two threads.

Parkinson and Bierman [54] introduce support for abstraction: by using abstract predicates with arguments in specifications (preconditions and postconditions), and defining which concrete state (or other abstract predicates) these abstract predicates relate to, one can write specifications in terms of the abstract predicates instead of in terms of the inner memory representation, i.e. write specifications that hide the inner workings of the program (or library). This allows one to reason at the level of the abstraction.

Bornat et al. [7], building further on work by Boyland [8], add support for marking memory as shared amongst threads (read-only), and not shared (and therefore ead-write) while allowing combining shared read-only memory (i.e. merging the split permissions) into unshared read-write, in a context where the number of permission splits is not static.

Gotsman [28] adds support for locks that can be created, stored, and destructed during the execution of the program, and links the invariant to the lock, instead of to the parallel composition as in concurrent separation logic. This allows to use real-world locks such as POSIX mutexes.

While one can use the theoretical developments outlined above for program verification using using pen and paper, in practice this is not advisable and it is better to use software tools. In other words, we use software to verify software. Tools that use separation logic include VeriFast [34, 35], SmallFoot [4], jStar [20], Ynot [48] (a Coq [32, 63] library), and Grasshopper [64].

Using tools can be quicker than doing the same work on paper, but an important advantage of tools is also that they are more reliable: when performing verification on paper, one can easily make an error that goes undetected. Another important advantage of some tools is that while a piece of paper does not give much feedback, a tool does. A tool can be a great teacher that is always available and (if the tool is fast) provides a very short feedback loop.

The only way to know whether an approach is actually applicable in the real world, is to try to apply the approach on a real world application. In Chapter 2, we successfully apply the verification tool VeriFast on `usbkbd`. VeriFast is a tool that implements the separation logic based approaches outlined above. `usbkbd` is a USB keyboard driver that is shipped with the Linux kernel. The properties verified are that none of the driver's executions crash, that there are no unintended race conditions, and that it does not violate a number of constraints of the APIs it uses, of which a part is complex and asynchronous.

Something missing in this story in input/output (I/O): programs do not only perform calculations and read and write to memory, but also communicate with the outside world: writing to screen, reading from keyboard, reading from and writing to files, driving external hardware such as servos, and sending and receiving network messages. Output is the behaviour as observed by an outside actor (such as a human) and input is the behaviour of such actor as observed by the program. The act of reading input by the program can, depending on the setting, also be observed by the outside actor. The output typically depends on the input: the output on a display of a calculator should be different depending on which sum the user has entered as input. The order of the input/output performed is important: we want the printer to first print page one and then page two.

I/O verification is the act of verifying that the I/O performed by the program is correct: we do not want the program that is supposed to read files to actually format the harddisk or read the file in an incorrect order.

We develop a new approach to verify Input/Output (I/O) properties of programs (Chapter 3). While some work has been done on I/O verification in a more simple setting, such as only supporting programs that always terminate [74] and only allowing I/O in the main loop of the program [5], our approach does not suffer from these limitations and furthermore supports modularity and compositionality. Our approach also supports some degree of underspecification and blends in with other approaches that address other aspects of the software, such as aliasing, locks, etc.

The difference between I/O and non-I/O can be a thin line: the program that writes to a file, can be writing to a filesystem in RAM (such as Linux's `tmpfs`), to a filesystem on disk (which can both be considered as memory or as a device for which input/output commands are send/received), or to a filesystem on network (sending/receiving network messages, such as NFS).

The program that writes the file is unaware of whether the filesystem is implemented by performing lower-level I/O actions, or by manipulating memory: in both cases the program uses the same filesystem API. We do not want to verify the program twice (once for a filesystem API in an I/O setting, and once for a filesystem API in a memory-manipulating setting). Ideally, the filesytem API should have one specification, both the I/O implementation and in-memory implementation are verified against this specification, and the program that uses the filesystem API is only verified once.

A similar situation is the `java.io.InputStream` abstract class of Java's standard library, which provides a stream API. The user of this API can ask an object of the class for the next n bytes. This abstract class is implemented both by a class `java.io.ByteArrayInputStream` which allows reading from a buffer in memory (not performing I/O), and a class `java.io.FileInputStream` which allows reading from a file in a filesystem (performing filesystem I/O). To write a specification for `InputStream`, one would have to come up with a specification that both the subclass performing I/O and the subclass performing memory-manipulation can implement.

To the best of our knowledge, no verification approach exists that supports such uniform specifications for I/O and in-memory. In Chapter 4 we develop such an approach that uses the same I/O style specifications of Chapter 3.

Chapter 5 wraps up the conclusions made throughout the thesis.

Chapter 2

Case study: verified keyboard driver

Publication data

- W. Penninckx, J. T. Mühlberg, J. Smans, B. Jacobs, and F. Piessens. Sound formal verification of Linux's USB BP keyboard driver. In *NASA Formal Methods*, volume 7226, pages 210–215. Springer, April 2012

- P. Philippaerts, J. T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens. Software verification with VeriFast: Industrial case studies. *Science of Computer Programming*, 82(1):77–97, March 2014

Abstract

Case studies on formal software verification can be divided into two categories: while *(i)* unsound approaches may miss errors or report false-positive alarms due to coarse abstractions, *(ii)* sound approaches typically do not handle certain programming constructs like concurrency and/or suffer from scalability issues. This chapter presents a case study on successfully verifying the Linux USB BP keyboard driver. Our verification approach is *(a)* sound, *(b)* takes into account dynamic memory allocation, complex API rules and concurrency, and *(c)* is applied on a real kernel driver which was not written with verification in mind. We employ VeriFast, a software verifier based on separation logic. Besides showing that it is possible to verify this device driver, we identify the

parts where the verification went smoothly and the parts where the verification approach requires further research to be carried out.

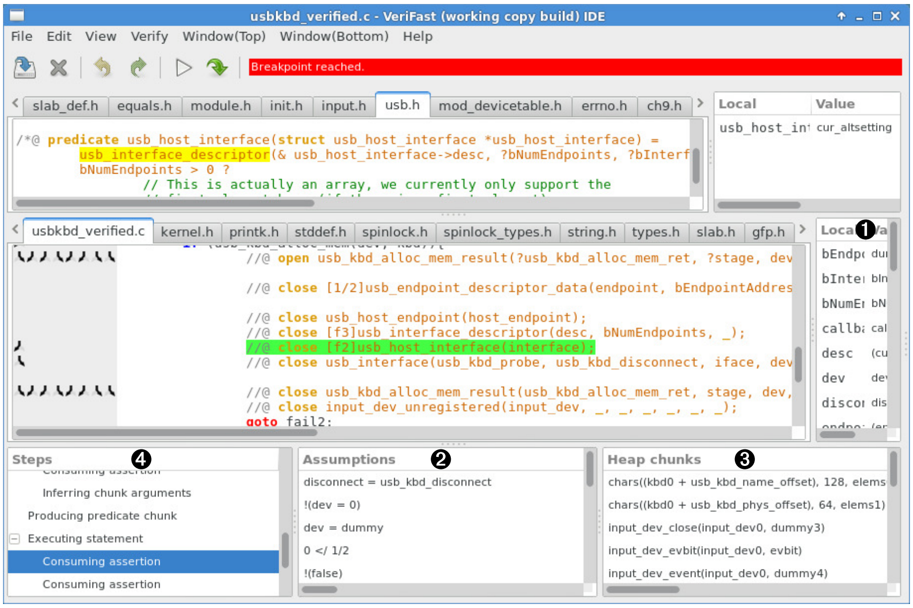
2.1 Introduction

The safety and security of today's omni-present computer systems critically depends on the reliability of operating systems (OS). Due to their complicated task of managing a system's physical resources, OSs are difficult to develop and to debug. As studies show, most defects causing operating systems to crash are not in the system's kernel but in the large number of OS extensions available [1, 11]. In Windows XP, for example, 85% of reported failures are caused by errors in device drivers [1]. As explained in [11], the situation is similar for Linux and FreeBSD: error rates reported for device drivers are up to seven times higher than error rates stated for the core components of these OSs.

A lot of research aims to prove the correctness of programs. However, not much work has been carried out to test whether the results of this research is applicable to complex real-world programs where correctness is important, like operating systems drivers. To work towards addressing this question, we apply a separation-logic based verifier, VeriFast [34], on a device driver taken from the Linux kernel.

The driver code subject to verification is Linux's USB Boot Protocol keyboard driver. While being small, this driver contains a bigger than expected subset of kernel driver complexity. It involves asynchronous callbacks, dynamic allocated memory, synchronization and usage of complex APIs. During verification, we identified and fixed a number of bugs. For these bugs we submitted patches that have been accepted by the driver's maintainer and are included in Linux[12].

In the remainder of this chapter we briefly introduce VeriFast and the device driver. We outline the verification of the driver and elaborate on the challenges involved. Finally, we discuss related work and draw conclusions.


usbkbd_verified.c - VeriFast (working copy build) IDE
File Edit View Verify Window(Top) Window(Bottom) Help
Breakpoint reached.
slab_def.h equals.h module.h init.h input.h usb.h mod_devicetable.h errno.h ch9.h
usbkbd_verified.c kernel.h printk.h stddef.h spinlock.h spinlock_types.h string.h types.h slab.h gfp.h
❶
❷
❸
❹

```
/*@ predicate usb_host_interface(struct usb_host_interface *usb_host_interface) =
      usb_interface_descriptor(& usb_host_interface->desc, ?bNumEndpoints, ?bInterf
      bNumEndpoints > 0 ?
              // This is actually an array, we currently only support the
```

Local	Value
usb_host_int	cur_altsetting

```
            //@ open usb_kbd_alloc_mem_result(?usb_kbd_alloc_mem_ret, ?stage, dev

            //@ close [1/2]usb_endpoint_descriptor_data(endpoint, bEndpointAddres

            //@ close usb_host_endpoint(host_endpoint);
            //@ close [f3]usb_interface_descriptor(desc, bNumEndpoints, _);
            //@ close [f2]usb_host_interface(interface);
            //@ close usb_interface(usb_kbd_probe, usb_kbd_disconnect, iface, dev

            //@ close usb_kbd_alloc_mem_result(usb_kbd_alloc_mem_ret, stage, dev,
            //@ close input_dev_unregistered(input_dev, _, _, _, _, _, _);
            goto fail2;
```

Local	Value
bEndp	du
bInte	bIn
bNumE	bN
callb	cal
desc	(cu
dev	de
disco	dis

Steps
- Inferring chunk arguments
- Producing predicate chunk
- Executing statement
  - Consuming assertion
  - Consuming assertion

Assumptions
- disconnect = usb_kbd_disconnect
- !(dev = 0)
- dev = dummy
- 0 </ 1/2
- !(false)

Heap chunks
- chars((kbd0 + usb_kbd_name_offset), 128, elems
- chars((kbd0 + usb_kbd_phys_offset), 64, elems1)
- input_dev_close(input_dev0, dummy3)
- input_dev_evbit(input_dev0, evbit)
- input_dev_event(input_dev0, dummy4)

Figure 2.1: The VeriFast IDE, showing for the current step (❹) in symbolic execution, the symbolic store (❶), the path conditions (❷), and the symbolic heap (❸).

2.2 VeriFast

The verifier we apply to the USB BP keyboard driver is VeriFast. To use VeriFast to verify a program, one writes extra annotations written as comments and thus ignored by the compiler. One kind of such annotations are preconditions and postconditions. For example:

```
int increase(int x, int y)
//@ requires x + y <= INT_MAX && x + y >= INT_MIN;
//@ ensures result == x + y;
{
  return x + y;
}
```

[1]https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=c196adf87514560f867492978ae350d4bbced0bd

[2]https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=a2b2c20ba2f6e22c065f401d63f7f883779cf642

The body of the function is written in C, but the annotations are written in another language where integer addition is always defined and not modulo a value different from one.

For this example, it suffices to say that the state of the program consists of the values assigned to the variables (here the arguments `x` and `y`), and to the return value. Some states satisfy the precondition, some do not.

When executing VeriFast with this example as input, VeriFast will perform some checks. One check is that for any initial state that satisfies the precondition, any execution starting from that state does not crash. Another check is that any such execution that terminates, yields a state that satisfies the postcondition.

For a compiler that uses 32 bit integers and for this trivial example, there are 18446744073709551616 possible initial states of which 13835058048839712768 satisfy the precondition, and every initial state has one possible execution (because the example does not have nondeterminism like memory allocation or concurrency). For a nontrivial program, checking each individual execution separately is infeasible.

In order to check all these executions, VeriFast uses *symbolic execution* [3]. Instead of using a concrete state consisting of a concrete store that maps variables to values, VeriFast uses a *symbolic state* that contains a *symbolic store* (see ❶ in Fig. 2.1 on the previous page) that maps variables to terms. In our example, the variables `x` and `y` could be mapped to the terms x and y. The symbolic state also contains a set of terms called *assumptions* or *path conditions* (❷ in Fig. 2.1 on the preceding page) that express the properties known so far in the symbolic execution. In our example, such a property can be $\mathsf{x}+\mathsf{y} \leq \mathsf{INT\_MAX} \wedge \mathsf{x}+\mathsf{y} \geq \mathsf{INT\_MIN}$. It is known because of the precondition.

Function calls are not verified by (symbolically) executing the body of the function. Instead, a function call is verified by checking that the precondition of the callee holds for the current symbolic state and assuming the postcondition of the callee. Loops are annotated by hand with loop invariants.

While one symbolic execution maps to potentially many concrete executions, a program can still have multiple symbolic executions. The precondition can contain a disjunction $P_1 \vee P_2$. In that case, VeriFast performs one symbolic execution where P_1 is added to path conditions, and another one where P_2 is added. Similarly, to verify an if-then-else statement, VeriFast performs a symbolic execution where the then-branch is executed and the condition is added to the set of assumptions, and another one where the else branch is executed and the negation of the condition is added.

So far we only considered assertions (such as the assertion of the precondition) that are terms. An assertion can also be a *points-to assertion* of the form $a \mapsto b$, where a and b are terms. This expresses that the heap consists of one cell that maps address a to b.

To check whether such an assertion holds for a symbolic state, the symbolic state also contains a *symbolic heap* (❸ in Fig. 2.1 on page 9) which is a multiset of assertions that are either points-to assertions or predicate assertions (explained later).

Another kind of assertion (not present in the symbolic heap) is the *separating conjunction* $P_1 * P_2$ where P_1 and P_2 are assertions. It expresses that the heap can be split into two disjoint parts where P_1 holds for one part and P_2 for the other [69].

To verify a function call, VeriFast also checks that the points-to assertions (and predicate assertions) that are present in the precondition of the callee are also present in the symbolic heap (possibly syntactically different), removes them from the symbolic heap, and adds the points-to assertions (and predicate assertions) that are present in the postcondition.

To provide abstraction and to describe complex memory data structures, the user can define *predicate*s in annotations. A predicate definition consists of a name and parameters, and a body which is an assertion. Assertions can contain predicate assertions, consisting of a predicate name and terms serving as arguments. The heap can also contain predicate assertions. The user has to write explicit annotations to fold and unfold (also called close and open) such assertions.

VeriFast also supports ghost data structures (to e.g. represent a mathematical list) and *lemma*s. Lemmas are similar to C functions and can modify the heap, but VeriFast checks that lemmas terminate and are only called from annotations (not from C code).

Concurrency is supported by associating a real number (called *fraction*) from $(0, 1]$ to every points-to assertion and predicate assertion, similar to [7]. So the heap can contain a points-to assertion for a certain fraction size. A points-to assertion of fraction size one denotes read and write permission, one of size less than one denotes only read permission. Multiple threads can obtain different fractions of the same permission.

Specifications for (spin)locks are done in a fashion similar to [28]: with a lock a handle and an invariant are associated. An invariant is a predicate that represents the permissions protected by the lock. A fraction of the handle allows acquiring the lock, which yields (adds to the symbolic heap) the invariant which represents the permissions protected by the lock.

The VeriFast tool is available at [37][3]. A formalization of a core subset of VeriFast [72] and a tutorial on VeriFast [36] are also available.

2.3 USB

Universal Serial Bus (USB) is a standard for interconnecting devices. Version 2.0 of this standard is described in [23]. A USB network consists of exactly one *host* (usually a desktop computer) and one or more USB devices [23, p. 16]. A *USB device* can be a USB *hub* or a *function*, such as a USB keyboard. With the host is always a hub associated. A USB network should be considered as a tree-formed network with the host at the root, the USB hubs at intermediate nodes and functions at the leaves. Communication is always initiated by the host; this implies that the host polls[23, p. 18] if it knows data might be available (but an interrupt-like interface may be built on top of polling).

A USB device has endpoints; an *endpoint* can be considered as the end of one communication channel from the USB host to the USB device [23, p. 33]. An endpoint has a number, and a direction: *IN* for data transmission from the device to the host[4] and *OUT* for transmission from the host to the device. Every USB device has an endpoint 0 IN and an endpoint 0 OUT. This is used for initialization and sending general commands.

An endpoint is associated with a *transfer type*; 4 transfer types exist: *isochronous* (stream-like, e.g. webcam), *control* (commands with irregular frequency), *interrupt* (latency is important, low frequency, e.g. mouse movements) and *bulk* (e.g. harddisk data transfer).

Endpoints are grouped into interfaces; an *interface* represents one functional unit of a device [23, p. 224]. For example, a keyboard with an integrated touchpad might have one interface for the keyboard itself and one interface for the touchpad. Every interface has an interface number and an *altsetting* number. Multiple interfaces with the same interface number but a different altsetting number can exist.

Interfaces are grouped into *configuration*s. Configurations can indicate different powering-settings. A device has one or more configurations (usually one [17]). When a device is in use, it is set to use one of its configurations.

[3]This is a newer version than the one used for performing verification in 2011; timings and lines of annotations reported are as of 2011.

[4]Even for communication that is conceptually from the device to the host, polling is still used.

USB devices provide information about themselves in *descriptors* [23, p. 260]. There exists a descriptor for the device, its configurations, the interfaces of the configurations and the endpoints. They are called device descriptor, configuration descriptor, interface disciptor and endpoint descriptor respectively.

For common devices like USB flash drives, separate USB specifications exist that describe protocols to communicate with these devices. This way, an end user does not need to install a vendor-specific driver for every USB device he uses, since usually the generic driver shipped with his operating system can be used.

If a new USB device is attached, the operating system should know which driver(s) to load. To easy this process, USB devices advertise a class, subclass and protocol number in their device descriptor and interface descriptor, which indicate what kind of device the device is. The operating system can use this information to (try to) load a device driver. For example, a standardized USB mouse might have class 3 (Human Interface Device) and protocol 2 (mouse) [24]; a flash drive might have class 8 (mass storage) and subclass 6 (SCSI) [25]. The class, subclass, and protocol numbers are defined in individual class specification documents[5].

2.4 Linux's USB API

2.4.1 Overview of structs

From a device driver's point of view, the following structs are important in Linux's USB API (see also Fig. 2.2 on the following page).

- **usb_device** represents a physical USB device. Remember that a USB device might consist of multiple functional units.
- **usb_interface** represents one functional unit of a USB device. A USB device driver is thus bound to `usb_interface`s, not to `usb_device`s.

  A `usb_interface` should not be considered as a USB interface, but rather as a grouping of different USB interfaces with the same interface number but different altsetting numbers.
- **usb_device_descriptor**, **usb_config_descriptor**, **usb_interface_-descriptor** and **usb_endpoint_descriptor** represent the descriptors.

[5] Available on http://www.usb.org/developers/devclass_docs/.

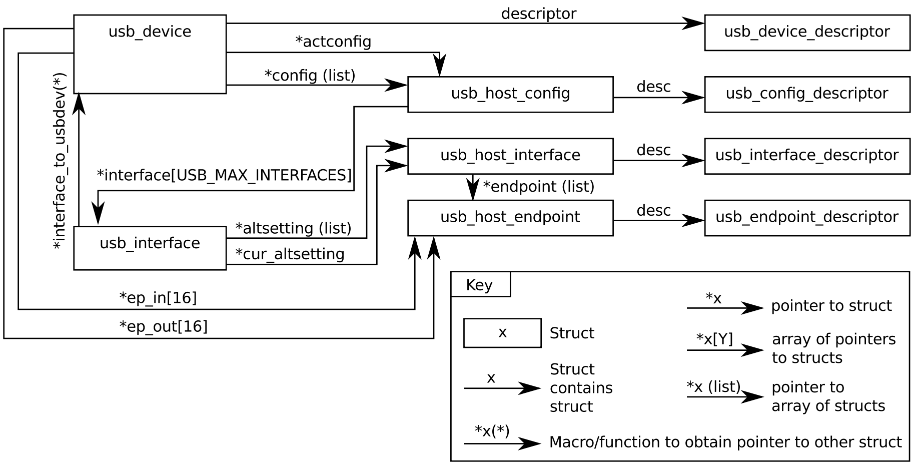


Figure 2.2: Overview of some structs in Linux's USB API

- **usb_host_config**, **usb_host_interface**, **usb_host_endpoint** wrap around `usb_config_descriptor`, `usb_interface_descriptor` and `usb_-endpoint_descriptor` respectively. **usb_host_interface** represents one altsetting of one interface.

2.4.2 Initialization and cleanup

A USB driver should register itself to the USB core by calling `usb_register`. This is typically done in the initialization function of the kernel module, which is called when the kernel module is loaded dynamically into the kernel.

`usb_register` expects a pointer to a struct as argument with the following fields:

- The name of the USB driver
- A pointer to a *probe* callback. This callback that will be called when a new USB interface (thus a USB function) is detected that this driver might support.
- A pointer to a *disconnect* callback. This callback that will be called when a USB function is disconnected which this driver was responsible for.
- A pointer to a table describing what kind of USB functions this driver supports. This table is called the *device ID table*.

The USB driver is typically unregistered (using `usb_deregister`), using the same pointer to the struct as used when registering. Deregistering the USB driver is typically done in the exit function of the USB driver kernel module, which gets called when the kernel module is unloaded.

2.4.3 Device ID table

Remember that the device ID table is passed to `usb_register`. When a new USB interface becomes available, the device ID tables of different kernel modules will be scanned. The descriptors of the USB device will be matched against the entries of the device ID table. If a match succeeds, the USB device driver (kernel module) will be "asked" if it really can support the USB interface. This is done by calling the probe callback of the USB driver. This way, only the probe function of interested USB drivers will be called. If a new USB driver is loaded, the above matching process is also performed for the device ID table of this new USB driver.

A USB device driver is supposed to include the macro `MODULE_DEVICE_TABLE (usb, usb_id_table)` in its sourcecode, with `usb_id_table` the device ID table, and "usb" left as is. This macro is used to known which kernel module should be loaded if a new USB device is connected[6]. Note that this is also useful for automatically loading kernel modules at boot time.

The device ID table of `usbkbd` is:

```
static struct usb_device_id usb_kbd_id_table [] = {
  { USB_INTERFACE_INFO(
      USB_INTERFACE_CLASS_HID,
      USB_INTERFACE_SUBCLASS_BOOT,
      USB_INTERFACE_PROTOCOL_KEYBOARD
    )
  },
  { } /* Terminating entry */
};
```

2.4.4 Probe and disconnect

A probe callback gets as first argument a pointer to an `usb_interface`. The job of the probe callback is to decide whether the given `usb_interface` is supported

[6]Source: `Documentation/usb/hotplug.txt` in the kernel source

by the particular USB device driver, and set up data structures such that the driver can perform its job as device driver for the given `usb_interface`. The entry of the device ID table that matched with the `usb_interface` is given as second argument to the probe callback. If the probe function judges it cannot handle the given `usb_interface`, a negative error code must be returned, and zero otherwise.

A device driver typically has an open function called when the device which the driver is responsible for is opened. The probe callback should not take over the job of the open function.

The USB API facilitates associating driver specific data to the `usb_interface` (with the functions `usb_set_intfdate` and `usb_get_intfdate`). `usb_set_-intfdata` is typically called in the probe callback. The USB driver can handle multiple USB interfaces concurrently, so this way the USB driver can associate state with the different USB interfaces.

2.4.5 URBs

Both sending data to and receiving data from a USB device is done using *USB Request Blocks* (*URB*s), which could best be considered as a data structure in the Linux API. To use URBs, they have to (1) be allocated, (2) be initialized, (3) be submitted. After submitting, they will (4) complete after a while. Submitting an URB does not block.

URB initialization requires specifying the transfer type (control, bulk, or interrupt), a direction (IN or OUT), a completion handler callback which called when the URB data is sent or received, a buffer of data that will be send or be filled, and whether this buffer is already DMA mapped, amongst other technicalities which we will not discuss here.

In case the device driver allocates DMA mapped memory itself it must free this in a special way (not with `kfree`).

An URB can be cancelled with `usb_kill_urb` This function blocks and makes sure the completion handler of the URB is not in execution when this function returns. If a driver does not use an URB anymore, it must call `usb_free_urb`.

2.4.6 Completion handlers

Remember that the completion handler callback is called when the URB finishes sending or receiving data. The completion handler callback receives the URB as argument. It examines the status of the URB (which is just the field `status` in the URB struct), because the URB might also complete because of an error. The completion handler can access the data inside the buffer.

Completion handlers (as well as the probe callback) are executed in interrupt context; code that executes for an interrupt is said to be executing in *interrupt context*. This applies both to code executed because it is an interrupt handler called when an interrupt happens (this is called the “top half”), or code that should be executed for an interrupt but can be scheduled to be executed a bit later and is interruptible by (other) interrupts (this is called the “bottom half”) [71, 42].

Code being executed in interrupt context cannot call certain other functions (also not indirectly by calling functions that call such functions), or can not call certain functions functions with(out) certain arguments or flags, such as `kmalloc` or the function to send an URB. Mutexes are also disallowed, but spinlocks are.

A completion handler for receiving data will typically send an URB, otherwise the communication with the USB device will simply stop here. For sending an URB, the original URB (that has completed now) can be reused.

2.5 Overview of how `usbkbd` works

The driver subject to verification is Linux’s USB Boot Protocol keyboard driver, named `usbkbd`[7]. This section gives a high-level overview of how the driver works (see Fig. 2.3 on the next page), leaving out details concerning concurrency and the exact API usage.

On loading, `usbkbd` registers itself with the USB API. When a new keyboard is attached, the API calls the `usb_kbd_probe` function of `usbkbd`. `usb_kbd_probe` checks whether the driver can handle the attached keyboard, and if so initializes a USB Request Block (URB). An URB is an asynchronous request that can be used to send or receive data from a USB device. The purpose of the URB initialized here is to receive key-presses and key-releases. This URB is named the IRQ URB. `usb_kbd_probe` initializes another URB for updating the LED status (e.g. numlock) named the LED URB. `usb_kbd_probe` then registers a new input device with the input API to make the keyboard available to applications. When the newly created input device is opened, `usbkbd`’s `usb_kbd_open` callback is invoked and `usb_kbd_open` submits the IRQ URB. When a key is pressed or

[7]The driver’s source file, `usbkbd.c`, is located in `drivers/hid/usbhid/` in the Linux kernel distribution available from `https://www.kernel.org/`.

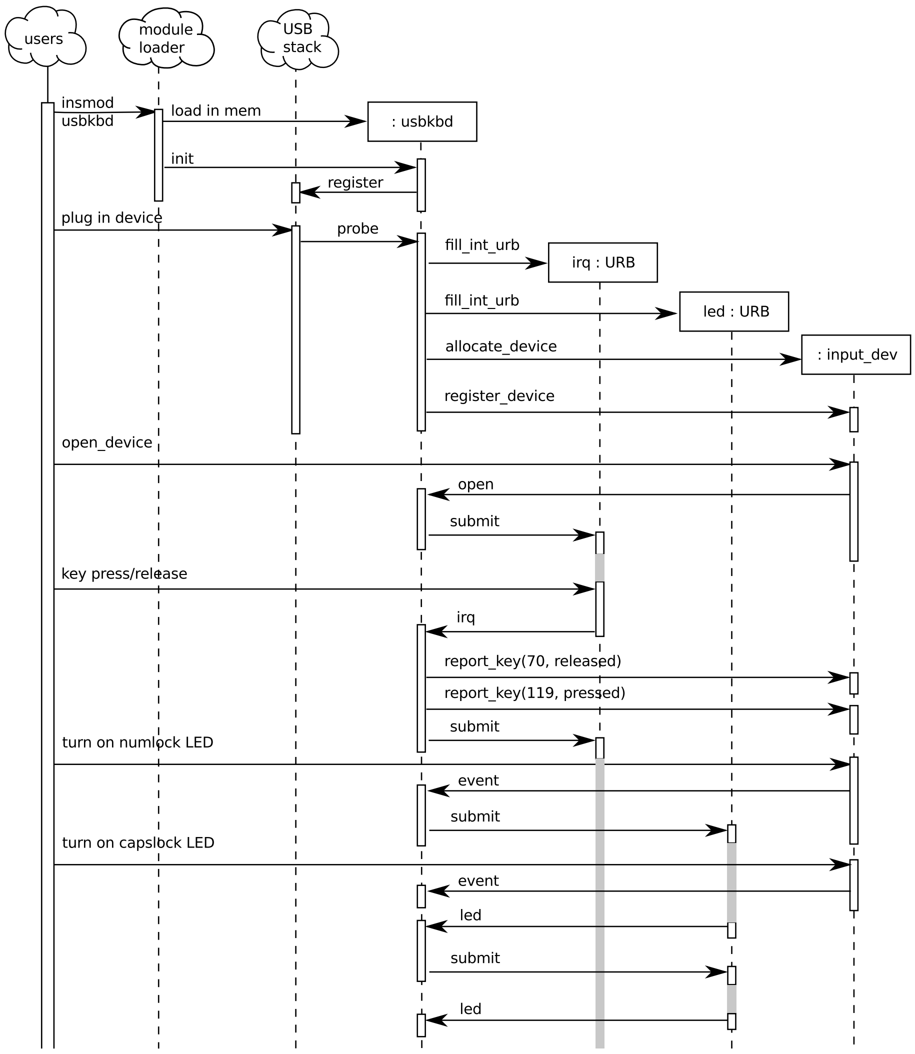


Figure 2.3: Flow of error free, low concurrent, single keyboard case

released, the URB completion callback `usb_kbd_irq` is called. `usb_kbd_irq` parses the data received from the keyboard and reports key-presses and releases to the input API. It then resubmits the URB. When the input API decides the LED status needs to be changed, the `usb_kbd_event` callback is invoked. This callback checks whether a LED URB is in progress, and if not submits the LED URB with the appropriate data. Otherwise, it stores the new LED info in a buffer. When the LED URB completion callback `usb_kbd_led` is called, this callback checks whether new LED info has appeared while the LED URB was in progress. If so, `usb_kbd_led` resubmits the LED URB with the new LED info.

2.6 Verifying the USB BP keyboard driver

Verification of the driver is against the original API. Wrapper functions are only used in a few cases where API functions return a struct (i.e. not a pointer to a struct) because this is currently not supported by VeriFast. The APIs that `usbkbd` uses are the USB API, the input API, spinlocks, and some generic functions like `memcpy`. Verification thus consists of (1) writing formal specifications for these APIs, based on official documentation and reading the API implementation for the underspecified or undocumented parts, and (2) of adding annotations to `usbkbd`. These annotations consists of contracts (pre- and postconditions written in separation logic), predicates to describe data structures, predicate family instances to instantiate callback function contracts, lemmas (i.e. ghost functions), and ghost-code like folding and unfolding predicates.

The verified properties are freedom of data races in the presence of concurrent callbacks, freedom of illegal memory accesses, and correct API usage. This does not include a formal proof of correctness of the hand-written API formalization. We do not verify functional correctness.

`usbkbd` is one of the smallest Linux kernel drivers. It consists of 426 lines of C code (including blanks and comments). VeriFast reports 329 lines of actual code and 822 lines of annotations. The API specifications count up to 769 lines of code. VeriFast can be launched for this driver with `verifast -I . -prover redux -c usbkbd_verified.c`. On an Intel L9400 1.86GHz running the verifier takes about one second. The annotated sources of `usbkbd`, specifications for the used APIs and the patches submitted to the driver's maintainer are shipped with VeriFast [37] under the directory `examples/usbkbd`.

Writing Specifications for the Input API and some generic functions like `kmalloc` was rather straightforward. API rules include forbidding double frees, requiring when registering input devices that the given callbacks are real function pointers with a contract not conflicting with some rules, etc.

Killable URBs were rather tricky to get verified for the LED URB. Because `usb_kbd_event` and `usb_kbd_led` both submit URBs, they are synchronized with a spinlock. A C boolean `led_urb_submitted` represents whether the URB is in progress, and thus also whether the URB data (necessary for URB submitting) is not owned by the lock invariant. After killing the URB, the URB data must be taken out of the lock invariant in order to free it, i.e. VeriFast must be convinced `led_urb_submitted` is false. We used a ghost-counter (associated with a predicate of which a uniqueness-proof must be provided on creation) named `cb_out_count` that yields a ticket on increase and ensures the counter is at least n high if n such tickets are owned. Another counter, `killcount`, keeps track of the number of URB submits. By making sure `killcount` tickets of `cb_out_count` are obtained when killing the URB, we can prove `cb_out_count` is at least as high as `killcount`. Because `cb_out_count` is maximum one less than `cb_out_count`, we know they are equal, which can only happen if the URB is not submitted.

The `usb_kbd_malloc` and `usb_kbd_free`'s Contracts take into account all possible combinations of failed and successful allocation and initialization, which makes their contracts long, and dependent on other parts of the annotations.

Flow Between Callbacks had to be reasoned about: permissions are passed between callbacks by setting up callbacks in other callbacks. Reasoning about flow between multiple callbacks easily gives the impression big parts of the program must be taken into account at the same time.

2.6.1 Killable asynchrounous resubmittable requests with completion callback

In this subsection, we describe a more challenging part of the verification in more detail.

Since both `usb_kbd_event` and `usb_kbd_led` submit the LED URB, they need to be synchronized. This is done by a boolean representing whether the URB is in progress and a spinlock protecting data including the boolean. `usb_kbd_led` will thus need a fraction of the lock handle. As a result, `usb_kbd_event` will need to give a fraction of his fraction of the lock handle to the `usb_submit_urb` as part of the callback data that the callback (here `usb_kbd_event`) will receive. Because the URB is submitted multiple times, the callback data (here fractions of the lock handle) will be given to the USB API multiple times. They will be given back when the URB is killed[8]. If an URB is submitted n times, n times the callback data will be returned when the URB is killed.

In order to enforce good API usage, the USB API's specifications ask to prove that the caller has submitted the URB n times if it wants the callback-data back n times. To facilitate this, `usb_submit_urb` returns a predicate that we call a ticket that indicates the URB is submitted. `usb_kill_urb` gives n times the callback predicate if it gets n tickets.

Note that we already introduced two counters: one counter that counts the number of times an URB is submitted (i.e. the amount of tickets), and one that counts the number of times a fraction of the lock handle is put in a callback data predicate. Since these two counters always have the same value, we only use one ghost counter. We call this counter `killcount`.

`usb_kbd_disconnect` must be able to free the LED URB. In order to do this, the struct containing the URB data must be obtained. This struct is contained in the lock invariant (i.e. "the data the lock protects") conditionally: if the boolean representing whether the URB is in progress is false, the lock contains the URB struct. Otherwise it does not since it is then owned by the URB API. Note that `usb_kbd_led` receives the URB struct such that it can resubmit the URB, but it can also store it in the spinlock if it does not resubmit.

Since `usb_kbd_disconnect` must obtain the URB struct, it must thus prove that the boolean is false. To be able to do this, we introduce a second counter `cb_out_count` which represent the number of times `usb_kbd_led` has returned without resubmitting the URB. A completion callback has "incoming data" (originally passed to `usb_submit_urb`), and "output data" (which will be given back by `usb_kill_urb`). If a completion handler does not resubmit, it must give back the output data in its postcondition. Otherwise, it must give back the incoming data and the predicate representing the URB struct such that the URB can be resubmitted by the USB API (note that resubmitting is thus deferred until the completion handler returns).

The lock invariant contains the claim that `cb_out_count` equals `killcount` if the boolean is false, and `cb_out_count` is one less than `killcount` otherwise.

So, `usb_kbd_disconnect` only needs to prove that the two counters are the same, such that it can prove the boolean is false and take the URB struct out of the lock invariant and free it. It is sufficient to prove (i.e. convince VeriFast) that `killcount` $\leq$ `cb_out_count`.

Since `usb_kbd_disconnect` kills the LED URB, it gets `killcount` times the

[8]This is a simplification. In fact, the cb-out is given back, not the cb-in. This is explained later on.

callback “output data”. We will use a special counter for `cb_out_count` that allows us to prove that `cb_out_count` is a least as big as `killcount`, provided that we have `killcount` times the callback “output data”.

Let us now look at how the `cb_out_count` counter works. This counter is just represented by a predicate with the counter value as one of the arguments. By increasing the counter value, a ticket associated with the counter is returned. Decreases eats one ticket. Given n tickets, the counter axioms state that the counter must be at least n high. In order to allow this to work, creating a counter requires a unique predicate, i.e. a predicate that can only have one instance for its arguments. Otherwise you could exchange tickets between counters, which would result in an important unsoundness. Creating a counter thus requires a predicate and a proof that this predicate is unique; both need to be passed as argument. Also note that we needs this uniqueness property to allow sharing counters inside multiple predicates. Otherwise it would be unknown whether two predicates containing the same counter are referring to the same counter value.

By putting the tickets of `cb_out_count` in the callback “output data”, we thus obtain `killcount` times this ticket, which allows us to prove that `cb_out_count` $\geq$ `killcount`.

While we were able to perform verification by working out the details of the above explained idea, we believe it would be nice if a more simple approach would be applicable.

It is also intresting to see that the part that was the hardest to get verified, was also the part that contained the bugs that we found in this driver. Originally, the driver did not perform any real synchronization between `usb_kbd_event` and `usb_kbd_led`, besides checking in a racy way whether the URB is in progress by reading a field of the URB struct (outside the completion handler), which is explicitly forbidden by the USB API documentation. The LED URB was originally also never killed.

2.7 Related work

Here we discuss related case studies and tools in the context of OS verification. The reader is referred to [34] for a discussion of the related work on VeriFast.

Several automated tools for verifying C programs have been introduced. Notably, CEGAR-based [13] model checkers such as BLAST [30] and SLAM/SDV [1] have been applied to check the conformance of device drivers with a set of API usage rules. In contrast with our work, these tools do not provide support for identifying errors with respect to the inherently concurrent execution environment device drivers are operating in. The tools also assume either that a program “does not have wild pointers” [1] or, as shown in [45], perform poorly when checking OS components for memory safety.

In [75] a model checker with support for pointers, bit-vector operations and concurrency is evaluated on a case study on Linux device drivers. The tool checks for buffer overflows, pointer safety, division by zero and user-written assertions. Yet, it requires a test harness with a fixed number of threads to be generated for each driver. VeriFast, in difference, handles concurrency implicitly and aims at verifying full functional correctness and implements assume-guarantee reasoning using generic API contracts. Therefore, VeriFast can check each function of a driver in isolation, which contributes to the scalability of our approach.

Bounded model checking and symbolic execution have been successfully applied to the source code [65, 39] and to the object code [46] of kernel modules. In contrast to the VeriFast approach, these techniques suffer from severe limitations with respect to reasoning about concurrently executing kernel threads.

Shape analysis has been applied to Windows [2] and Linux [76] drivers, and aims to automatically infer, e.g. whether a variable points to a cyclic or acyclic list. Shape analysis can be employed to verify pointer safety, guaranteeing that the shape of data structures is maintained throughout program execution. Ongoing work on VeriFast envisages the use of shape analysis to infer annotations [73].

A competing toolkit to VeriFast is the Verifying C Compiler (VCC) [16]. VCC verifies C programs annotated with contracts. VCC generates Boogie code, from which the Boogie program verifier generates verification conditions, which are then discharged by an SMT solver. VCC can be expected to require fewer annotations than VeriFast, however, at the expense of a less predictable search times. The toolkit has been employed in a case study on verifying the Microsoft Hypervisor.

Other approaches to OS verification involve modelling and interactive proof. Most notably, the L4.verified [29] project aims at producing a verified OS kernel by establishing refinement relations between several layers of Isabelle/HOL specifications, a prototypic kernel implementation in Haskell and the actual kernel implementation in C and assembly. This differs from our work as we do not employ refinement relations and verification is non-interactive.

2.8 Conclusions

We report on the successful verification of `usbkbd`, the USB Boot Protocol keyboard driver distributed with the Linux kernel, using the sound and efficient verification tool VeriFast. The verified properties are crash-freedom, race-freedom, and a set of API usage rules. The `usbkbd` driver presents a challenging case study as it involves concurrency and employs a complex API.

VeriFast requires the source code to be annotated with method contracts that are typically easy to write. Certain programming constructs that are difficult to annotate are discussed. During verification, we identified two bugs related to erroneous synchronization and a missing URB kill. Our case study shows that VeriFast is a powerful tool. Yet, the annotation overhead amounts to a total of 4.8 lines of annotations per line of code. About half of these annotations specify API contracts, that can potentially be reused in future case studies.

Verifying functional correctness and unload-safety is left for further work. Unload-safety includes making sure the kernel does not maintain a function-pointer to a callback of a module that is already unloaded. It is hard to tell whether our verification approach will scale for larger device drivers. More automation for writing or generating annotations with a high degree of decomposition might help. From our experience we conclude that execution speed of the verification tool will not impose problems for larger drivers.

List of symbols (I/O)

bio	A Basic Input Output (BIO) action	Def. 6 p. 31
$c \in \text{Commands}$	Commands	Def. 2 p. 30
$C \in \text{Chunks}$	Heap chunks	Def. 10 p. 37
$\mathcal{C} \in \text{Values} \rightarrow \text{Commands}$		Def. 3 p. 30
$f \in \text{FuncNames}$	A program function name	Def. 4 p. 31
fc	The program function context	Def. 5 p. 31
fC	The mathematical function used to look up user-written preconditions and postconditions.	Def. 30 p. 53
$h \in \text{Heaps}$	Heaps	Def. 11 p. 38
no_io	Dummy element in a program trace representing no I/O happens for this element	Def. 7 p. 33
no_op	No I/O action, used for optional I/O.	Sec. 3.6.4 p. 56
P	An assertion (also used for set of heaps before assertions are introduced)	Def. 27 p. 51
Places	Places of Petri nets	Def. 12 p. 38
PredTx	Set of predicate transformers	Def. 20 p. 46
Q	A function from Values to assertions (also used for functions from Values to set of heaps before assertions are introduced). See also: P.	
safe	Safe trace	Fig. 3.9 p. 47
t	See: v	

$v \in \text{Values}$		Def. 2 p. 30
wp	Weakest precondition	Def. 22 p. 46
τ	A program trace	Def. 7 p. 33
σ	An element in a program trace	Def. 7 p. 33
$c \Downarrow \tau , v$	Program c evaluates to trace τ and return value v	Fig. 3.1 p. 32
$\mathbb{T}$	A Petri net trace	Def. 9 p. 36
$h \uplus h$	Heap addition	Def. 14 p. 39
$h \xrightarrow{bio(v, v)} h$	Petri net / heap step performing a BIO	Def. 15 p. 39
$h \xrightarrow{\epsilon} h$	Petri net / heap step performing split, join, or no_op	Def. 15 p. 39
$h \Downarrow \mathbb{T}$	Petri net / heap execution	Fig. 3.3 p. 40
$\mathbb{T} \sim \tau$	Petri net trace simulates program trace	Def. 17 p. 40
$P \vDash c$	Program satisfies the specification consisting of a set of Petri nets	Def. 17 p. 40
$h \xRightarrow{bio(v, v)} h$	Finite epsilon steps followed by one BIO step	Def. 19 p. 45
$\lceil P \rceil$	Assertion that "allows" finite epsilon steps before P	Def. 24 p. 49
$\lceil Q \rceil$	Postcondition that "allows" finite epsilon steps before postcondition Q	Def. 25 p. 49
$\models \left\{ P \right\} c \left\{ Q \right\}$	Validity of a Hoare triple	Def. 26 p. 50
$h \vDash P$	Assertion P holds for heap h	Fig. 3.10 p. 51
$\vdash \left\{ P \right\} c \left\{ Q \right\}$	Provable Hoare triple according to the proof rules	Fig. 3.12 p. 53

Chapter 3

I/O verification

Publication data

- W. Penninckx, B. Jacobs, and F. Piessens. Sound, modular and compositional verification of the input/output behavior of programs. In J. Vitek, editor, *Programming Languages and Systems, European Symposium on Programming (ESOP 2015), London, UK, 14-16 April 2015*, pages 158–182. Springer Berlin Heidelberg, Apr. 2015
- W. Penninckx, B. Jacobs, and F. Piessens. Modular, compositional and sound verification of the input/output behavior of programs. CW Reports CW663, Department of Computer Science, KU Leuven, May 2014
- W. Penninckx and B. Jacobs. Sound, modular and compositional verification of the input/output behavior of programs: extended version. To be submitted

Abstract

We present a program verification approach for programs that perform I/O. The approach is sound, modular, compositional (I/O actions can be defined on top of other actions). It uses Petri nets and separation logic. We have implemented the approach in the VeriFast verifier and sketched how it can be implemented in the Iris framework.

3.1 Introduction

We present an I/O verification approach as applied to a simple programming language that can perform I/O and a Hoare logic with proof rules. The specifications allow to express which I/O the program is allowed to perform in which order and the proof rules allow to prove that the program does not violate this. If a proof using these proof rules exists, then we are sure that the program will never perform undesired I/O and will never perform I/O in an incorrect order. We do not prove that the program does perform any I/O; such liveness properties are outside the scope of the thesis. However, if the program terminates, we do know the program has performed all desired I/O (i.e. the I/O the program should do according to the specification of the program), and if the program does not terminate we still know all I/O performed is desired.

The approach supports programs that do not always terminate, since this is rather common for programs that perform I/O. Indeed, a text editor, a calculator, or a traffic light: they are not guaranteed to terminate since the user (or absence thereof) might simply never quit the application.

The approach also supports constraining the output depending on the input. For example, which digits a calculator is allowed to display (output) depends on which digits are entered (input).

Furthermore, the approach supports modularity, which means that one can combine verified pieces of software (e.g. libraries) without having to (re)do work to use them together. It is not necessary to verify them again.

Besides modularity, the approach also supports compositionality: it is possible to define I/O actions on top of more primitive I/O actions.

The high-level idea of the approach is as follows. We specify a program's allowed behaviors by means of potentially infinite typically non-circular Petri nets whose transitions are labeled by program actions and environment actions. Specifically, a program's specification describes a set of such Petri nets. The specifications are written using assertions that can contain corecursive predicates to support loops. A program satisfies such a specification if it simulates each of these Petri nets, in the sense that for each of the program's runs, each Petri net has a run where the program's actions match or else the environment's actions do not match. That is, one can underspecify a program's behavior by specifying nondeterministic Petri nets, and one can underspecify the environment's behavior by specifying multiple Petri nets with distinct environment actions.

To verify a program, we reason in terms of an instrumented program semantics where the program state includes a Petri net. In this semantics, an I/O primitive goes wrong if the Petri net cannot perform a corresponding transition, and otherwise updates the Petri net accordingly. This allows us to use a Hoare logic to prove that the program does not go wrong when executed under an arbitrary Petri net from the specified set. More specifically, we use a separation logic, which allows us to reason about the mutation of the Petri net while framing out and abstracting over parts of it, thus achieving modularity and compositionality.

The remainder of this chapter is organized as follows. We define the syntax (Sec. 3.2) and semantics (Sec. 3.3) of the programming language. We explain which role Petri nets play in writing specifications (Sec. 3.4). We define an instrumented semantics (Sec. 3.5) that links program execution with Petri net execution. We explain assertions and proof rules (Sec. 3.6). We give some examples in Sec. 3.7. Finally, we sketch how one can implement the I/O style verification approach in Iris (Sec. 3.8). Contrary to our approach which is made specifically for I/O, Iris [38] is a generic framework for reasoning about concurrent programs that manipulate a shared resource.

We end the chapter with a discussion of related work (Sec. 3.9) and a conclusion (Sec. 3.10).

3.2 Programming language: syntax

In this section we present the syntax of a simple programming language that supports I/O, which we use throughout this chapter to present our approach. Using the syntax of the language as defined in this section, we will define the semantics of this programming language in Sec. 3.3 on page 32.

The syntax of a command is as follows:

Definition 1: Syntax of Commands, $c \in \text{Commands}$

$$c ::= v \mid \textbf{let } c \textbf{ in } \mathcal{C} \mid f(\overline{v}) \mid bio(v)$$

Any command is a program.

We explain each element of this syntax:

Values v and t range over Values (not variables), which is defined as

Definition 2: $v, t \in \text{Values}$

$$\text{Values} = \mathbb{Z} \cup \mathbb{Z}^* \cup \{\text{true}, \text{false}, \text{unit}, \bot\} \cup \text{Places} \cup \text{FuncNames}$$

but we postpone Places and FuncNames for now (to Def. 4 on the next page and Def. 12 on page 38). So a value can be an integer, a list of integers, a boolean, unit, or $\bot$.

Executing a command produces a side-effect and a result value. A command can simply be a value. For example, 42 is a command that has no side-effect and yields the result value 42.

The programming language does not have operations ($+$, $-$, $*$, head, tail, $\dots$), but it is possible to use the metalogic[1] for operations: $1 + 1$ is a way to write the value 2. Similarly, we use the metalogic for if-then-else. For example: **if** $1 > 2$ **then** 7 **else** 3.

Let

Definition 3: $\mathcal{C} \in \text{Values} \to \text{Commands}$

$\mathcal{C}$ ranges over mathematical functions from Values to Commands.

So $\mathcal{C}$ in a command of the form **let** c **in** $\mathcal{C}$ is such a function.

The programming language does not have variables: one just writes

$$\textbf{let } 1 * 1 \textbf{ in } \lambda x.\, x + x$$

where $\lambda x.\, x + x$ is a lambda expression of the metalogic. We use the more convenient syntax

$$\textbf{let } x := 1 * 1 \textbf{ in } x + x$$

Function application f ranges over function names:

[1]For a programming language where "1+1" is a valid command in the programming language, writing "1+1" is ambiguous: it could be a command of the programming language, or it could be a mathematical addition as we used before studying programming languages. In the former case we say it is in the logic; in the latter case we say it is in the metalogic. For our specific programming language under consideration, "1+1" is not ambiguous because it is always in the metalogic.

Definition 4: $f \in \text{FuncNames}$

We assume a set FuncNames of function names.

We use overline to range over lists, so $\overline{v}$ ranges over lists of values. A command $f(\overline{v})$ is a function application of function f to arguments $\overline{v}$.

We write a value that is a list comma-separated, e.g. $1, 2, 3, 4$ is a value that is a list. For a function name f_1 and values v_1 and v_2, $f(v_1, v_2)$ is a function call. $f()$ is a function call where we use the empty list.

Note that f ranges over function names, not function definitions.

Definition 5: fc

We assume a given mathematical function, called fc, from FuncNames to mathematical functions from Values to Commands.

Since the function context maps a function name to a function, and a function call just mentions the name of the function and not the definition, we can create loops by having fc map a function name and arguments to a command containing a call to that function name. For example, consider infloop as a function name and a mapping where

$$\text{fc}(\mathsf{infloop}) = \lambda x.\, \mathsf{infloop}(x+1)$$

Given this definition, all executions of the command $\mathsf{infloop}(1)$ are infinite.

I/O The programming language has explicit syntax for expressing input/output.

Definition 6: BIOs

We assume a set of Basic Input/Output (BIO) actions, and range over it with *bio*.

The command $bio(v)$ performs the BIO action *bio* with argument v, e.g. the program $\mathsf{putchar}(3)$ outputs the number three on the screen if putchar is considered a BIO. Remember that programs return values, e.g. the program 4 returns the value 4. A program consisting of a BIO call is considered to return the input value, e.g. the program $\mathsf{getchar}(\text{unit})$ returns the value read from

$$\frac{}{v \Downarrow \langle\rangle, v}\ \text{Val} \qquad \frac{c \Downarrow \tau_1, v_1 \quad \mathcal{C}(v_1) \Downarrow \tau_2, v_2}{\textbf{let}\ c\ \textbf{in}\ \mathcal{C} \Downarrow \tau_1 \cdot \tau_2, v_2}\ \text{Let}$$

$$\frac{(\text{fc}(f))(\overline{v_1}) \Downarrow \tau, v_2}{f(\overline{v_1}) \Downarrow \text{no\_io} \cdot \tau, v_2}\ \text{App} \qquad \frac{}{\mathit{bio}(v) \Downarrow \langle \mathit{bio}(v, v_r)\rangle, v_r}\ \text{Bio}$$

Figure 3.1: Step semantics

the keyboard if getchar is considered a BIO. All BIO actions are considered to be both input and output: a pure output action can be modelled as a BIO action that returns unit. If the output value (i.e. the value "given to the outside world") is irrelevant we sometimes omit it, e.g. we write getchar() instead of getchar(unit).

When applying our approach, BIOs can be used to model calls to a complex graphics or sound library, calls to the C standard library, system calls (calls to the operating system's kernel, implemented as software interrupts), or they could be calls to functions implemented to perform specific I/O CPU instructions or perform memory-mapped I/O. For simplicity of our formalization, BIOs are the lowest-level input/output action under consideration, but it could be a high-level action (to e.g. a library) or a low-level action (e.g. a system call) in practice. Because the programming language supports functions, it is possible to write programs on top of higher-level input-output actions, where these actions are implemented as functions that call BIOs or call other higher-level input-output actions.

3.3 Programming language: semantics

In this section we define the semantics of the programming language.

Because we are interested in input/output and we want to support nonterminating programs, we need a way to reason about the input/output performed by nonterminating programs. We do this by defining program traces as potentially infinite lists of actions.

More formally:

Definition 7: Program traces (τ)

We define program traces coinductively as follows:

$$\sigma ::= \text{no\_io} \mid \mathit{bio}(v, v)$$
$$\tau ::= \langle\rangle \mid \sigma \cdot \tau$$

We usually abbreviate "program trace" to "trace".

A trace is a sequence of input-output actions. For example, the trace $\mathit{bio}_1(v_1, v_2) \cdot \mathit{bio}_2(v_2, v_3) \cdot \langle\rangle$ expresses that first the action bio_1 happens with argument v_1 and return value v_2. v_1 is output (from the point of view of the program), and v_2 is input. In other words, v_1 is information that flows from the program to the world, and v_2 flows from the world to the program. After that, bio_2 happens with argument v_2 and result value v_3.

It will later be useful that traces of nonterminating programs are infinite, even if they do not perform any I/O. This is what no_io is used for: with a nonterminating program that does not perform I/O we associate the infinite trace $\text{no\_io} \cdot \text{no\_io} \cdot \text{no\_io} \cdot \ldots$, and not the finite trace $\langle\rangle$.

We write a finite trace like $\mathit{bio}_1(v_1, v_2) \cdot \mathit{bio}_2(v_3, v_4) \cdot \langle\rangle$ also as $\langle \mathit{bio}_1(v_1, v_2), \mathit{bio}_2(v_3, v_4)\rangle$.

Definition 8: Step semantics ($c \Downarrow \tau, v$)

Fig. 3.1 on the preceding page defines the step semantics of the programming language coinductively.

This is a coinductive big-step semantics similar[2] to [47]. $c \Downarrow \tau, v$ expresses that executing the program c can result in a (possibly infinite) trace of I/O actions τ where it (if it terminates) returns value v.

Bio Consider the BIO rule. An example instance of this rule is $\mathsf{putchar}(1) \Downarrow \langle\mathsf{putchar}(1, 0)\rangle, 0$. This expresses that the program $\mathsf{putchar}(1)$ has an execution where the I/O actions are described by the trace $\langle\mathsf{putchar}(1, 0)\rangle$, and where the program returns 0.

[2] We use I/O actions as items in the trace, instead of program state consisting of a mapping of variables to values.

Nondeterminism and input Note that $c \Downarrow \tau_1, v_1 \wedge c \Downarrow \tau_2, v_2$ does not imply that $\tau_1 = \tau_2$ nor that $v_1 = v_2$. For example, for the program $\mathsf{read}(0)$, both $\mathsf{read}(0) \Downarrow \langle \mathsf{read}(0,7) \rangle, 7$ and $\mathsf{read}(0) \Downarrow \langle \mathsf{read}(0,8) \rangle, 8$ hold. In the former execution, the command reads the value 7, and in the latter it reads the value 8. In other words, we consider BIO actions to be nondeterministic because a program that reads input can read different values on different runs and therefore multiple executions are possible for the same program.

Function call The App step rule describes how function calls are executed. Note that this step rule adds no_io to the trace. If the function call creates an infinite loop, the trace will therefore be infinite. This gives us the property that we always associate an infinite trace to a nonterminating program execution, even if the execution does not perform an infinite number of input/output actions.

Let Consider the Let step rule. It uses trace concatenation, written with the infix symbol $\cdot$, which is defined corecursively as follows: $(\sigma \cdot \tau_1) \cdot \tau_2 = \sigma \cdot (\tau_1 \cdot \tau_2)$ and $\langle\rangle \cdot \tau = \tau$.

Consider the program $\mathbf{let}\ c\ \mathbf{in}\ \mathcal{C}$, an execution $c \Downarrow \tau_1, v_1$ of c, and an execution $\mathcal{C}(v_1) \Downarrow \tau_2, v_2$ of $\mathcal{C}(v_1)$. The Let rule states that $\mathbf{let}\ c\ \mathbf{in}\ \mathcal{C}$ has an execution $\mathbf{let}\ c\ \mathbf{in}\ \mathcal{C} \Downarrow \tau_1 \cdot \tau_2, v_2$. Note that in case τ_1 is infinite, i.e. the execution of c that is under consideration does not terminate, then $\tau_1 \cdot \tau_2 = \tau_1$. So in that case τ_2 is ignored. In case τ_1 is finite, i.e. the execution of c that is under consideration terminates, then τ_2 is not ignored.

3.4 Specifications: Petri nets

In this section we study our flavor of Petri nets (Sec. 3.4.1), the heap representation of Petri nets and Petri net execution (Sec. 3.4.2), and the relation between program execution and Petri net execution (Sec. 3.4.3).

This prepares us for verifying modularly and compositionally that a program satisfies a specification given by a set of Petri nets (Sec. 3.5), proof rules, and assertions that describe sets of Petri nets (Sec. 3.6).

3.4.1 Petri nets

The kind of Petri nets we use in this thesis consists of

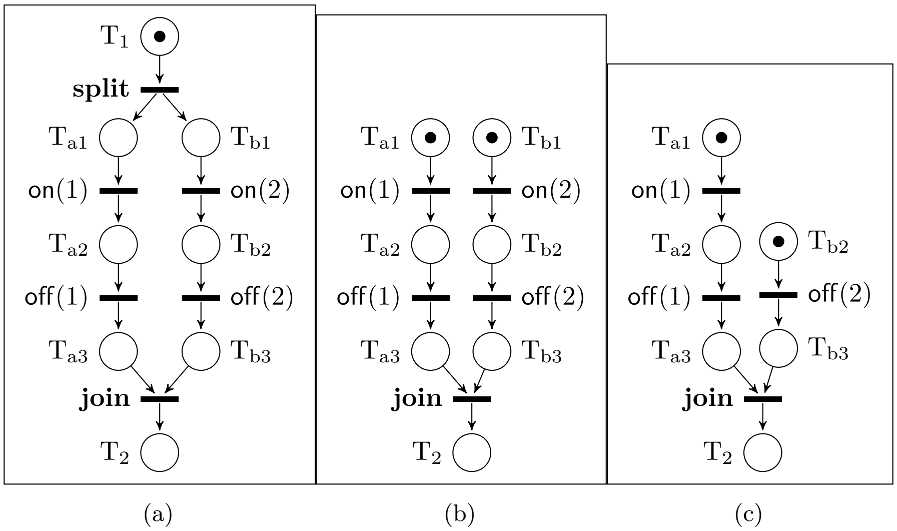


Figure 3.2: Example Petri net with two executions steps

- a set of places. Places are visualized as circles.
- a marking: a function from places to natural numbers. Markings are visualized as a number of dots (called tokens) in the circles that represent places.
- a set of transitions (does not overlap with the set of places). Transitions are visualized as bars.
- a set of arrows. An arrow goes from a place to a transition, or from a transition to a place.

A Petri net can be executed by performing a number of execution steps. An execution step manipulates the marking (although it is possible the resulting marking equals the starting one). Deviating from the standard definition of Petri nets, an execution step also removes one transition. To perform an execution step, select a transition for which all incoming arrows start from a place with at least one token. The resulting marking is the original one with one token removed from every place that has an outgoing arrow to the transition, and one token added to every place that has an incoming arrow from the transition. The selected transition is removed.

Since sometimes there are multiple transitions to choose from to perform an execution step, there can be multiple possible executions for a given Petri net. Execution is nondeterministic.

Fig. 3.2 on the preceding page shows an example sequence of three Petri nets, where successive Petri nets are related by an execution step.

The Petri nets we use in our approach have four kinds of transitions:

- BIO transitions are labelled by a BIO action and have one incoming and one outgoing arrow;
- split transitions have one incoming and two outgoing arrows;
- join transitions have two incoming arrows and one outgoing arrow; and
- no_op transitions are labelled with **no_op** and have one incoming and one outcoming arrow. They are dummy transitions that represent not performing any I/O. We will explain the use of no_op transitions later (Sec. 3.6.4).

An execution of a Petri net yields a (potentially infinite) Petri net trace of the transitions used in the steps of the execution.

Definition 9: Petri net trace ($\mathbb{T}$)

We define Petri net traces coinductively as follows:

$\mathbb{T} ::= \langle\rangle \mid \epsilon \cdot \mathbb{T} \mid bio(v_o, v_i) \cdot \mathbb{T}$

We define the trace of an execution of a Petri net as follows. We record every transition in the trace in the order in which the transitions were selected in the execution. For a transition that is a split, a join or a no_op, we write ϵ in the trace, and for an I/O action we write the BIO, input, and output value of the I/O action. For simplicity, we combine one program action (i.e. output action) and one environment action (input action) into one BIO action that does both input and output. In case the input value (or sometimes the output value) does not matter, we do not write it down.

Here are the traces of some of the executions of the Petri net of Fig. 3.2a on the previous page:

- $\langle \epsilon, \mathsf{on}(1), \mathsf{off}(1), \mathsf{on}(2), \mathsf{off}(2), \epsilon \rangle$

- $\langle \epsilon, \mathsf{on}(1), \mathsf{on}(2), \mathsf{off}(2), \mathsf{off}(1), \epsilon \rangle$
- $\langle \epsilon, \mathsf{on}(2), \mathsf{off}(2), \mathsf{on}(1), \mathsf{off}(1), \epsilon \rangle$

Here are traces for which no execution of the Petri net of Fig. 3.2a on page 35 yield the trace:

- $\langle \epsilon, \mathsf{off}(1), \mathsf{on}(2), \mathsf{off}(2), \epsilon \rangle$ (cannot occur)
- $\langle \epsilon, \mathsf{on}(1), \mathsf{off}(2), \mathsf{on}(2), \mathsf{off}(1), \epsilon \rangle$ (cannot occur)
- $\langle \epsilon, \mathsf{on}(1), \mathsf{on}(2), \mathsf{on}(1), \mathsf{off}(2), \mathsf{off}(1), \epsilon \rangle$ (cannot occur)

3.4.2 Petri nets as heaps

In this subsection we study how we represent a Petri net as a heap instead of as a drawing. Here, we do not use classic heaps that are (partial) function from memory addresses to values. Instead, we use heaps that are multisets of heap chunks.

In this subsection we also use the heap representation of a Petri net to formally define execution of a Petri net. The heap representation also serves as a first step towards defining an instrumented semantics (Sec. 3.5) where the program state includes a Petri net, and towards writing assertions (Sec. 3.6) that describe sets of Petri nets.

Definition 10: $C \in \text{Chunks}$

We define the set Chunks of *chunks*, ranged over by C:

$$C ::= \ \mathit{bio}(t, v, v, t) \mid \textbf{no\_op}(t, t) \mid \textbf{split}(t, t, t) \mid \textbf{join}(t, t, t) \mid \textbf{token}(t)$$

Remember both v and t range over Values.

A chunk $\mathit{bio}(t_1, v_i, v_o, t_2)$ represents the BIO transition for BIO *bio* with input value v_i and output value v_o and the arrow from place t_1 to that transition and the arrow from that transition to place t_2.

A chunk $\textbf{split}(t_1, t_2, t_3)$ represents a split transition and an arrow from place t_1 to that transition, an arrow from that transition to t_2 and an arrow from that transition to the place t_3.

A chunk $\mathbf{join}(t_1, t_2, t_3)$ represents a join transition and an arrow from place t_1 to that transition, an arrow from the place t_2 to that transition, and an arrow from that transition to the place t_3.

A chunk $\mathbf{no\_op}(t_1, t_2)$ represents a no_op transition and an arrow from the place t_1 to that transition, and an arrow from the transition to the place t_2.

A chunk $\mathbf{token}(t)$ expresses the Petri net marking has at least one token (i.e. in the visual representation the place has at least one dot). The number of tokens (or dots) in the place t in the Petri net depends on the number of $\mathbf{token}(t)$ chunks.

We call a potentially infinite multiset of chunks a heap. We range over heaps with h:

Definition 11: $h \in \text{Heaps}$

$$\text{Heaps} = \text{Chunks} \rightarrow \mathbb{N} \cup \{\infty\}$$

Definition 12: Places

We assume an infinite set Places. We assume it contains the elements $\mathrm{T}_1, \mathrm{T}_2, \mathrm{T}_3, \ldots \mathrm{T}_{\mathrm{a}1}, \mathrm{T}_{\mathrm{a}2}, \mathrm{T}_{\mathrm{a}3}, \ldots \mathrm{T}_{\mathrm{b}1}, \mathrm{T}_{\mathrm{b}2}, \mathrm{T}_{\mathrm{b}3}, \ldots, \mathrm{T}_{\mathrm{z}1}, \mathrm{T}_{\mathrm{z}2}, \mathrm{T}_{\mathrm{z}3}, \ldots$

So we can say $\mathrm{T}_3 \in \text{Places}$.

In this thesis, we only consider Petri nets where its set of places is a subset of Places. When defining Values earlier (Def. 2 on page 30), it was defined such that Values is a superset of Places.

Consider the following heap:

$$\{\!\!\{\mathbf{token}(\mathrm{T}_1), \mathbf{split}(\mathrm{T}_1, \mathrm{T}_{\mathrm{a}1}, \mathrm{T}_{\mathrm{b}1}), \mathsf{on}(\mathrm{T}_{\mathrm{a}1}, 1, \mathrm{T}_{\mathrm{a}2}), \mathsf{off}(\mathrm{T}_{\mathrm{a}2}, 1, \mathrm{T}_{\mathrm{a}3}),$$
$$\mathsf{on}(\mathrm{T}_{\mathrm{b}1}, 2, \mathrm{T}_{\mathrm{b}2}), \mathsf{off}(\mathrm{T}_{\mathrm{b}2}, 2, \mathrm{T}_{\mathrm{b}3}), \mathbf{join}(\mathrm{T}_{\mathrm{a}3}, \mathrm{T}_{\mathrm{b}3}, \mathrm{T}_2)\}\!\!\}.$$

The Petri net representation of this heap is the leftmost Petri net in Fig. 3.2 on page 35.

We do not write the input or output argument of BIOs when it is not interesting, so $\mathsf{on}(\mathrm{T}_{\mathrm{a}1}, 1, \mathrm{T}_{\mathrm{a}2})$ stands for $\mathsf{on}(\mathrm{T}_{\mathrm{a}1}, 1, \mathrm{unit}, \mathrm{T}_{\mathrm{a}2})$ and $\mathsf{getchar}(\mathrm{T}_{\mathrm{a}1}, 27, \mathrm{T}_{\mathrm{a}2})$ stands for $\mathsf{getchar}(\mathrm{T}_{\mathrm{a}1}, \mathrm{unit}, 27, \mathrm{T}_{\mathrm{a}2})$.

Definition 13: Addition on $\mathbb{N} \cup \{\infty\}$

We define addition on $\mathbb{N} \cup \{\infty\}$, written with infix operator $+$, as follows ($+_{\mathbb{N}}$ is addition of natural numbers):

$$\begin{aligned}
\infty + \infty &= \infty \\
\infty + n &= \infty \\
n + \infty &= \infty \\
n_1 + n_2 &= n_1 +_{\mathbb{N}} n_2
\end{aligned}$$

We define addition of heaps, written with infix operation $\uplus$, as follows:

Definition 14: Addition of heaps ($h \uplus h$)

$$h_1 \uplus h_2 = (\lambda C.\, h_1(C) + h_2(C))$$

Now that we have heaps, we define execution of Petri nets again, but this time in terms of heaps.

We define a relation between heaps: $h_1 \xrightarrow{bio(v_o,v_i)} h_2$ expresses that the heap h_1 allows to perform the BIO action *bio* with argument v_o. Furthermore, performing such action will return value v_i and the resulting (state of the) Petri net is expressed by h_2.

Definition 15: Petri net step ($h \xrightarrow{bio(v,v)} h$ and $h \xrightarrow{\epsilon} h$)

$$\{\!\!\{\mathbf{token}(t_1), bio(t_1, v_o, v_i, t_2)\}\!\!\} \uplus h \xrightarrow{bio(v_o,v_i)} \{\!\!\{\mathbf{token}(t_2)\}\!\!\} \uplus h$$

$$\{\!\!\{\mathbf{token}(t_1), \mathbf{split}(t_1, t_2, t_3)\}\!\!\} \uplus h \xrightarrow{\epsilon} \{\!\!\{\mathbf{token}(t_2), \mathbf{token}(t_3)\}\!\!\} \uplus h$$

$$\{\!\!\{\mathbf{token}(t_1), \mathbf{token}(t_2), \mathbf{join}(t_1, t_2, t_3)\}\!\!\} \uplus h \xrightarrow{\epsilon} \{\!\!\{\mathbf{token}(t_3)\}\!\!\} \uplus h$$

$$\{\!\!\{\mathbf{token}(t_1), \mathbf{no\_op}(t_1, t_2)\}\!\!\} \uplus h \xrightarrow{\epsilon} \{\!\!\{\mathbf{token}(t_2)\}\!\!\} \uplus h$$

$h_1 \xrightarrow{\epsilon} h_2$ expresses one step of the Petri net h_1 that does not perform I/O. This is used for split, join, and no_op transitions.

$$\frac{}{h \Downarrow \langle\rangle}\ \text{Stop} \qquad \frac{h \xrightarrow{\epsilon} h' \quad h' \Downarrow \mathbb{T}}{h \Downarrow \epsilon \cdot \mathbb{T}}\ \text{Epsilon} \qquad \frac{h \xrightarrow{bio(v_o,v_i)} h' \quad h' \Downarrow \mathbb{T}}{h \Downarrow bio(v_o, v_i) \cdot \mathbb{T}}\ \text{Bio}$$

Figure 3.3: Execution of a Petri net / heap

Definition 16: Petri net execution ($h \Downarrow \mathbb{T}$)

Executing a Petri net is defined coinductively in Fig. 3.3.

3.4.3 Programs and Petri nets

On the one hand we have defined a programming language, and how an execution of a program can perform I/O (Sec. 3.3). Executing the program yields a program trace of I/O actions. On the other hand we have defined Petri nets and how they are executed (Sec. 3.4.1 and Sec. 3.4.2). Executing a Petri net yields a Petri net trace.

In this subsection we study the relation between program execution and Petri net execution, and how we use a set of Petri nets as the specification of the allowed I/O [3]. The latter prepares us for writing assertions that describe sets of Petri nets (Sec. 3.6.1).

A program c satisfies the specification P consisting of a set of Petri nets, denoted $P \vDash c$, if for every execution of the program, and for every Petri net in that set of Petri nets, there exists an execution in the Petri net such that the trace of the execution of the Petri net simulates the trace of the execution of the program.

More formally:

Definition 17: Petri net trace simulates program trace ($\mathbb{T} \sim \tau$)

When a Petri net trace simulates a program trace, written $\mathbb{T} \sim \tau$, is defined coinductively in Fig. 3.4 on the facing page.

The Contra rule of Fig. 3.4 on the next page will be explained later (p. 44).

[3] For now we ignore specifying which I/O must have been performed if the program terminates.

$$\frac{\mathbb{T} \sim \tau}{\mathbb{T} \sim \text{no\_io} \cdot \tau}\ \text{NoIO} \qquad \frac{\mathbb{T} \sim \tau}{\epsilon^* \cdot bio(v_o, v_i) \cdot \mathbb{T} \sim bio(v_o, v_i) \cdot \tau}\ \text{Bio}$$

$$\frac{v_i \neq v_i'}{\epsilon^* \cdot bio(v_o, v_i') \cdot \mathbb{T} \sim bio(v_o, v_i) \cdot \tau}\ \text{Contra} \qquad \frac{}{\mathbb{T} \sim \langle\rangle}\ \text{Empty}$$

Figure 3.4: Relation between Petri net traces and program traces. ϵ^* ranges over finite Petri net traces that only consist of epsilons, e.g. $\langle \epsilon, \epsilon, \epsilon \rangle$ and $\langle\rangle$.

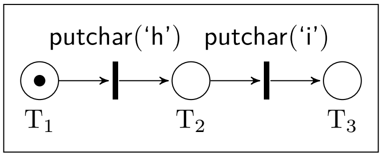


Figure 3.5: One Petri net that only has one execution

Definition 18: $P \vDash c$

For a set of heaps P (we write $P(h)$ to say h is in P) and a program c, we define whether c satisfies the specification P, written $P \vDash c$, as follows:

$$P \vDash c \iff \forall h, \tau, v.\ P(h) \wedge c \Downarrow \tau, v \Rightarrow \exists \mathbb{T}.\ h \Downarrow \mathbb{T} \wedge \mathbb{T} \sim \tau$$

We now illustrate this specification formalism by means of examples.

Simple Petri net We start with the simple situation where the set of Petri nets only contains one Petri net, input is not considered (only output), the Petri net only has one possible run, and the program is deterministic (only has one execution). Consider for example the set of Petri nets that only consists of the Petri net of Fig. 3.5 and the program (call it c_1)

$$\mathsf{putchar('h'); putchar('i')}$$

The Petri net has only one trace, namely $\mathbb{T}_1 = \langle \mathsf{putchar('h')}, \mathsf{putchar('i')} \rangle$ and the program has only one execution, namely $c_1 \Downarrow \tau_1, \text{unit}$ where $\tau_1 = \langle \mathsf{putchar('h')}, \mathsf{putchar('i')} \rangle$.

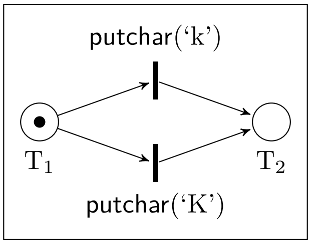


Figure 3.6: One Petri net with two executions

Looking at Fig. 3.4 on the previous page it is easy to see that $\mathbb{T}_1 \sim \tau_1$. Therefore, the program satisfies its specification, i.e. $\{\{\!\!\{\textbf{token}(\mathrm{T}_1), \mathsf{putchar}(\mathrm{T}_1, \text{'h'}, \mathrm{T}_2), \mathsf{putchar}(\mathrm{T}_2, \text{'i'}, \mathrm{T}_3$
c_1.

The following program does not satisfy the same specification:

$$\mathsf{putchar}(\text{'h'}); \mathsf{putchar}(\text{'w'})$$

Underspecification of the program Now we consider Petri nets that have multiple executions, such as the one in Fig. 3.6. It has the following traces:

- $\mathbb{T}_2 = \langle \mathsf{putchar}(\text{'k'}) \rangle$
- $\mathbb{T}_3 = \langle \mathsf{putchar}(\text{'K'}) \rangle$

For a program to satisfy the specification consisting of only the one Petri net of Fig. 3.6, the traces of the program's executions only need to simulate one of the traces of the Petri net. Therefore, the following programs satisfy the specification:

$$\mathsf{putchar}(\text{'k'})$$

$$\mathsf{putchar}(\text{'K'})$$

The implementation does not have to choose statically which Petri net trace to simulate. If the programming language had a random number generator (that does not perform I/O), then the following program would also satisfy the specification:

$$\textbf{let } x := \textbf{rand in if } x > 0 \textbf{ then } \mathsf{putchar}(\text{'K'}) \textbf{ else } \mathsf{putchar}(\text{'k'})$$

The following programs do not respect the specification:

$$\mathsf{putchar}(\text{'k'}); \mathsf{putchar}(\text{'K'})$$

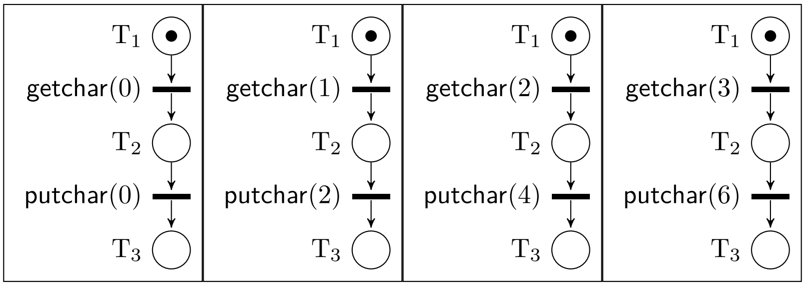


Figure 3.7: Set of Petri nets

putchar('l')

whisper('k')

Summarizing, Petri nets with multiple traces are useful to underspecify the program. This can be useful: For example, there might be two "OK" messages in a protocol because of historical reasons, and either can be sent as a reply. The person implementing the function that has this specification, can choose which one to actually send.

Split/join The Petri net of Fig. 3.2 on page 35 contains a split and a join. This way, it also has multiple traces. The program can then choose which one to follow. Split and join allow one to more easily specify interleavings of I/O actions. While for this simple example it is possible to just let the Petri net contain all the possible interleavings instead of using split and join, this way of avoiding split and join won't be possible anymore once we introduce compositionality (Sec. 3.6.3).

Input, underspecifying the environment, and multiple Petri nets A Petri net can have multiple executions (and therefore multiple traces) and we explained this is used for nondeterminism of the program. For nondeterminism of the environment, we use a set of Petri nets as specification instead of only one Petri net. This also allows to constrain the environment by leaving Petri nets out of the set.

Consider the example of Fig. 3.7. This is a set of four Petri nets. Informally, this specification expresses that the environment will input 0, 1, 2, or 3 and that the program will respond by twice outputting the received value.

For a program to satisfy this specification, for *every* Petri net of the set and for every execution of the program, the program trace of the program execution must simulate at least one Petri net trace of the Petri net. The reason this must be true for every Petri net (and not just for one) is because the program must behave well for every possible behavior of the world.

Consider the following program which we will call c_2:

let $x :=$ getchar() **in**
putchar($x \times 2$)

This program has infinitely many executions, including the following three:

- $c_2 \Downarrow \langle \mathsf{getchar}(0), \mathsf{putchar}(0) \rangle, \text{unit}$
- $c_2 \Downarrow \langle \mathsf{getchar}(1), \mathsf{putchar}(2) \rangle, \text{unit}$
- $c_2 \Downarrow \langle \mathsf{getchar}(2), \mathsf{putchar}(4) \rangle, \text{unit}$

It is reasonable that every execution "must behave properly" and not perform disallowed I/O. Consider the first execution: $c_2 \Downarrow \langle \mathsf{getchar}(0), \mathsf{putchar}(0) \rangle, \text{unit}$. It is clear that the trace of this execution simulates a trace of the leftmost Petri net of Fig. 3.7 on the preceding page since it is the same trace. We said earlier that the program execution must simulate a trace of *every* Petri net in the set, because the program must behave correctly for every behavior/choice of the environment. So it must also simulate a trace of the second Petri net. This is the case: according to the Contra rule of Fig. 3.4 on page 41, if a Petri net trace and a program trace agree up to an I/O action where they disagree on the input value of a BIO (but they agree on the output value and on the BIO itself), then the one trace simulates the other, regardless of the remainders of the traces. In other words: as soon as the environment violates the specification when executing the program, the program is allowed to perform any I/O.

Note that by using a set of Petri nets as a specification (contrary to using one Petri net as a specification) we do not only support underspecification of the environment (i.e. input), but also specifying I/O behavior that is different depending on the environment's behavior.

3.5 Instrumented semantics

To verify a program, we reason in terms of an instrumented program semantics (which we define in this section) where the program state includes a Petri net.

$$\frac{h_1 \overset{\epsilon}{\to} h_2 \qquad h_2 \overset{\epsilon}{\to}^* h_3}{h_1 \overset{\epsilon}{\to}^* h_3} \qquad\qquad \frac{}{h \overset{\epsilon}{\to}^* h}$$

Figure 3.8: Definition of $h_1 \overset{\epsilon}{\to}^* h_2$: it expresses that there is a finite number of epsilon steps from h_1 to h_2. (used in Def. 19)

In this semantics, an I/O primitive goes wrong if the Petri net cannot perform a corresponding transition, and otherwise updates the Petri net accordingly. This allows us to use a Hoare logic (Sec. 3.6) to prove that the program does not go wrong when executed under an arbitrary Petri net from the specified set. More specifically, we use a separation logic, which allows us to reason about the mutation of the Petri net while framing out and abstracting over parts of it, thus achieving modularity and compositionality.

Contrast to our definition of correctness in the previous subsection, correctness of a command is defined compositionally in terms of correctness of its subcommands. This allows verifying modularly and compositionally that a program satisfies a specification given by a set of Petri nets.

Specifically, in this section, we define an instrumented semantics in the form of a predicate transformer semantics that maps a command and a postcondition to the weakest precondition, similar to Dijkstra's weakest precondition semantics [18]. In the next section (Sec. 3.6) we use this semantics to define a Hoare logic. Before defining this semantics, we define some helper definitions.

We write $h_1 \xRightarrow{bio(v_o,v_i)} h_2$ to express that the heap h_1 can perform a finite number of epsilon steps (splits, joins, no_ops), followed by the BIO step. By doing so, the heap h_2 is obtained. More formally:

Definition 19: $h \xRightarrow{bio(v,v)} h$

$$h_1 \xRightarrow{bio(v_o,v_i)} h_2 = \exists h'.\ h_1 \overset{\epsilon}{\to}^* h' \land h' \xrightarrow{bio(v_o,v_i)} h_2$$

This definition uses Fig. 3.8.

We define predicate transformers as the functions that map a command and a postcondition on a precondition. More formally:

Definition 20: Predicate transformer (PredTx)

$$\text{PredTx} = \text{Cmds} \to (\text{Values} \to \mathcal{P}(\text{Heaps})) \to \mathcal{P}(\text{Heaps})$$

Here, Cmds is the set of all syntactically valid commands.

A predicate transformer *ptx* is monotone if it allows weakening the postcondition. Formally:

Definition 21: Monotone predicate transformer

A predicate transformer *ptx* is monotone iff

$$\forall Q, Q' \in (\text{Values} \to \mathcal{P}(\text{Heaps})), c \in \text{Commands}.$$
$$\big(\forall v, h.\ Q(v)(h) \Rightarrow Q'(v)(h)\big) \Rightarrow \big(\forall h.\ ptx(c, Q)(h) \Rightarrow ptx(c, Q')(h)\big)$$

Here, $\Rightarrow$ is implication. We write $Q(v)(h)$ to express h is a heap in the set of heaps $Q(v)$.

Definition 22: wp

We define wp as the weakest monotone predicate transformer that satisfies the following equations[4].

- $\text{wp}(\textbf{let}\ c\ \textbf{in}\ \mathcal{C}, Q) = \text{wp}(c, \lambda v.\, \text{wp}(\mathcal{C}(v), Q))$
- $\text{wp}(f(\overline{v}), Q) = \text{wp}(\text{fc}(f)(\overline{v}), Q)$
- $\text{wp}(bio(v_o), Q) = \lambda h.\, \exists v_i, h'.\ h \xRightarrow{bio(v_o, v_i)} h' \land Q(v_i)(h')$
- $\text{wp}(v, Q) = Q(v)$

wp maps a command and a postcondition to a precondition. One can think of the precondition as a set of heaps, i.e. a set of Petri nets or an I/O specification, and the postcondition as a function that maps a value to a set of heaps.

For a command that is a BIO and a postcondition, wp returns the heaps h for which it is possible to perform the BIO to reach a heap h' in the postcondition. Some examples:

[4] A unique weakest solution exists; see Appendix A.1.

$$\frac{h_1 \xRightarrow{bio(v_o,v_i)} h_2 \qquad \text{safe}(h_2,\tau,P)}{\text{safe}(h_1, bio(v_o,v_i)\cdot\tau, P)}\ \text{SafeBio}$$

$$\frac{h_1 \xRightarrow{bio(v_o,v_i')} h_2 \qquad v_i \neq v_i'}{\text{safe}(h_1, bio(v_o,v_i)\cdot\tau, P)}\ \text{SafeContradict} \qquad \frac{\text{safe}(h,\tau,P)}{\text{safe}(h, \text{no\_io}\cdot\tau, P)}\ \text{SafeNoIO}$$

$$\frac{\lceil P\rceil(h)}{\text{safe}(h, \langle\rangle, P)}\ \text{SafePost}$$

Figure 3.9: Definition of a safe trace for a start heap and a postcondition.

- $\text{wp}(\mathsf{getchar}(\text{unit}), \lambda v.\ \{\{\!\{\mathbf{token}(\text{T}_2)\}\!\}\}) \supseteq \{\{\!\{\mathbf{token}(\text{T}_1), \mathsf{getchar}(\text{T}_1, \text{unit}, 2, \text{T}_2)\}\!\}\}$
- $\text{wp}(\mathsf{getchar}(\text{unit}), Q) \supseteq P$ where
  $Q = \lambda v.\ \{h \mid \exists h_r.\ h = h_r \uplus \{\!\{\mathbf{token}(\text{T}_2)\}\!\}\}$
  $P = \{\{\!\{\mathbf{token}(\text{T}_1), \mathsf{getchar}(\text{T}_1, \text{unit}, 2, \text{T}_2)\}\!\}\}$

  Note that the postcondition allows leaking: while the postcondition constrains the heap to contain the heap chunk $\mathbf{token}(\text{T}_2)$, the postcondition allows the heap to contain other heap chunks as well.
- $\text{wp}(c, \lambda v.\ \{\{\!\{\mathbf{token}(\text{T}_3)\}\!\}\}) \supseteq P$ where
  $P = \{h \mid \exists i.\ h = \{\!\{\mathbf{token}(\text{T}_1), \mathsf{getchar}(\text{T}_1, i, \text{T}_2), \mathsf{putchar}(\text{T}_2, i * 2, \text{T}_3)\}\!\}\}$
  $c = \mathbf{let}\ x := \mathsf{getchar}()\ \mathbf{in}\ \mathsf{putchar}(x * 2)$

  Note that P includes the Petri nets of Fig. 3.7 p. 43.

Note that the weakest precondition semantics does not only define whether a command satisfies a specification, but also defines the semantics of commands.

3.5.1 Soundness

We show that the new compositional definition of correctness implies the old non-compositional one. In other words, we show that if a program c satisfies a specification P according to the weakest precondition semantics, then it also satisfies this specification according to the previous non-modular semantics:

$$P \subseteq \mathrm{wp}(c, Q) \Rightarrow P \vDash c$$

One can easily prove that $P \vDash c$ allows strengthening the precondition, i.e. $P \subseteq P' \wedge P' \vDash c \Rightarrow P \vDash c$. Therefore, it suffices to prove that

$$\mathrm{wp}(c, Q) \vDash c$$

By definition it is equivalent to show that:

$$\forall h, \tau, v.\ \mathrm{wp}(c, Q)(h) \wedge c \Downarrow \tau, v \Rightarrow \exists \mathbb{T}.\ h \Downarrow \mathbb{T} \wedge \mathbb{T} \sim \tau$$

In other words, if a heap is in the weakest precondition of the program for some postcondition, then every (potentially infinite) program trace obtained by executing the program is simulated by a Petri net trace obtained by executing the Petri net.

Instead of proving this soundness statement, we will prove a stronger soundness statement. We formulate the stronger soundness statement in two steps.

As the first step, we make the soundness statement stronger by also constraining the postcondition. Note that the conclusion ($\exists \mathbb{T}.\ h \Downarrow \mathbb{T} \wedge \mathbb{T} \sim \tau$) does not constrain the postcondition. To additionally constrain the postcondition, we instead show that:

$$\forall h, \tau, v.\ \mathrm{wp}(c, Q)(h) \wedge c \Downarrow \tau, v \Rightarrow \mathrm{safe}(h, \tau, Q(v))$$

Definition 23: safe

safe is defined coinductively in Fig. 3.9 on the previous page.

Informally, $\mathrm{safe}(h, \tau, P)$ expresses that a run of a program that yields the trace τ satisfies the following properties:

- The trace τ is simulated by a trace $\mathbb{T}$ that "starts" from h. As a result, the trace τ is only allowed to perform I/O actions that are allowed by the start heap h.
- If the program trace τ is finite, and the world did not violate promises expressed in h, then the postcondition P must be met for the heap obtained by performing the steps of $\mathbb{T}$ on h. Remember that only finite executions of programs yield finite program traces.

The start heap h that expresses the permissions of allowed I/O, can contain promises about the world, and the world can violate these promises.

For example, we have

$$\text{safe}(\{\!\!\{\textbf{token}(t_1), \text{getchar}(t_1, 0, 3, t_2)\}\!\!\}, \langle \text{getchar}(0, 4), \text{putchar}(7, 0)\rangle, P)$$

The heap $\{\!\!\{\textbf{token}(t_1), \text{getchar}(t_1, 0, 3, t_2)\}\!\!\}$ promises that the world will produce 3 when reading. In the trace the world returns 4, violating this promise. The program only has to behave correctly as long as the world does not violate such promises, hence it can perform the `putchar` action even though it did not have permission to do so.

We show that the soundness statement is made stronger and not weaker:

Theorem 1

$$\forall h, \tau, P.\ \text{safe}(h, \tau, P) \Rightarrow \exists \mathbb{T}.\ h \Downarrow \mathbb{T} \wedge \mathbb{T} \sim \tau$$

Proof. Full proof on p. 169. □

The second step in making the soundness statement stronger will be useful for a later theorem (Theorem 4 on page 59). This second step needs the following definitions.

Definition 24: $\lceil P \rceil$

For a precondition P, we define $\lceil P \rceil$ as

$$\lceil P \rceil = \lambda h.\ \exists h'.\ h \xrightarrow{\epsilon}^{*} h' \wedge P(h')$$

Intuitively, $\lceil P \rceil$ is the set of heaps that can "reach" a heap in P by performing a finite number of epsilon steps.

Definition 25: $\lceil Q \rceil$

For a postcondition Q, we define $\lceil Q \rceil$ as:

$$\lceil Q \rceil = \lambda v.\ \lceil Q(v) \rceil$$

As the second step in making the soundness statement stronger we write the soundness statement as follows (note that $\text{wp}(c, Q)(h) \Rightarrow \lceil \text{wp}(c, \lceil Q \rceil) \rceil(h)$):

Theorem 2

$$\forall c, Q, h, v, \tau.\ c \Downarrow \tau, v \wedge \lceil \mathrm{wp}(c, \lceil Q \rceil) \rceil(h) \Rightarrow \mathrm{safe}(h, \tau, Q(v))$$

Proof. Proof by induction nested inside coinduction. (Full proof on p. 170) □

3.6 Program logic: assertions and proof rules

In the previous section we studied when a program satisfies its specifications thanks to the instrumented semantics. In this section we define proof rules for this based on Hoare logic and assertions that describe sets of heaps.

3.6.1 Hoare triples and assertions

A Hoare triple consists of a precondition P, a program c and a postcondition Q and is written $\{P\}\, c\, \{Q\}$. P is an assertion. In the setting of this section, Q is a function from values to assertions such that Q can constrain the return value of the program.

We say a Hoare triple $\{P\}\, c\, \{Q\}$ is valid if for every heap (or Petri net) h that satisfies the precondition P and for every execution of c, the trace τ of this execution is simulated by a trace $\mathbb{T}$ of the Petri net. Furthermore, if the program terminates, then the postcondition Q is satisfied by the return value of the execution and the Petri net obtained by performing the steps of the trace $\mathbb{T}$ on h. More precisely ($h \vDash P$ means h satisfies the assertion P):

Definition 26: Validity of a Hoare triple ($\models \{P\}\, c\, \{Q\}$)

$$\begin{aligned}&\forall P, c, Q.\ \models \{P\}\, c\, \{Q\} \iff \\ &\quad \forall h, v, \tau.\ h \vDash P \wedge c \Downarrow \tau, v \Rightarrow \mathrm{safe}(h, \tau, (\lambda v, h.\ h \vDash Q(v)))\end{aligned}$$

For a Hoare triple $\{P\}\, c\, \{Q\}$, the precondition P is an assertion, and the postcondition Q is a function from values to assertions (this allows the postcondition to constrain the return value).

Assertions are defined syntactically as follows:

$$\begin{array}{c} \\ \hline\hline h \vDash \text{true} \end{array} \qquad \begin{array}{c} \\ \hline\hline h \uplus \{\!\{C\}\!\} \vDash C \end{array} \qquad \begin{array}{c} h_1 \vDash P_1 \quad h_2 \vDash P_2 \\ \hline\hline h_1 \uplus h_2 \vDash P_1 * P_2 \end{array}$$

$$\begin{array}{c} h \vDash (\text{predmap}(p))(\overline{v}) \\ \hline\hline h \vDash p(\overline{v}) \end{array} \qquad \begin{array}{c} \exists v \in \text{Values}.\ h \vDash \mathcal{P}(v) \\ \hline\hline h \vDash \exists \mathcal{P} \end{array} \qquad \begin{array}{c} h \vDash P \\ \hline\hline h \vDash P \vee P' \end{array}$$

$$\begin{array}{c} h \vDash P' \\ \hline\hline h \vDash P \vee P' \end{array}$$

Figure 3.10: Satisfaction relation of assertions

Definition 27: Assertions (P)

$P ::= \ v \mid P * P \mid C \mid p(\overline{v}) \mid P \vee P \mid \exists \mathcal{P}$

Definition 28: Assertion holds for a heap ($h \vDash P$)

Fig. 3.10 defines coinductively for which heaps an assertion holds.

In Fig. 3.10, $\mathcal{P}$ ranges over functions from values to assertions. One can ignore p for now; it ranges over copredicate names and will be explained on p. 55. Notice that assertion satisfaction is preserved by adding chunks: we can prove by coinduction that $h \vDash P \Rightarrow h \uplus h' \vDash P$.

Here is an example Hoare triple:

$$\left\{ \begin{array}{l} \textbf{token}(\mathrm{T}_1) * \\ \mathsf{putchar}(\mathrm{T}_1, \text{`}\mathsf{h}\text{'}, \mathrm{T}_2) * \\ \mathsf{putchar}(\mathrm{T}_2, \text{`}\mathsf{i}\text{'}, \mathrm{T}_3) \end{array} \right\}$$

```
putchar('h');
putchar('i')
```

$$\{\ \lambda\_.\,\textbf{token}(\mathrm{T}_3)\ \}$$

$$\frac{\forall h.\; h \vDash P_1 \Rightarrow h \vDash P_2}{P_1 \Rrightarrow P_2}$$

$$\frac{}{\mathbf{token}(t_1) * \mathbf{split}(t_1, t_2, t_3) \Rrightarrow \mathbf{token}(t_2) * \mathbf{token}(t_3)}$$

$$\frac{}{\mathbf{token}(t_1) * \mathbf{token}(t_2) * \mathbf{join}(t_1, t_2, t_3) \Rrightarrow \mathbf{token}(t_3)}$$

$$\frac{}{\mathbf{token}(t_1) * \mathbf{no\_op}(t_1, t_2) \Rrightarrow \mathbf{token}(t_2)} \qquad \frac{P \Rrightarrow P'}{P * R \Rrightarrow P' * R}$$

Figure 3.11: Rewrite rules

We write $c; c'$ to abbreviate **let** c **in** $\lambda\_. c'$. In this example Hoare triple, the precondition describes exactly one Petri net, namely the one of Fig. 3.5 on page 41.

The precondition of our example states that the program is allowed to write 'h', followed by writing 'i'. It does not allow the program to perform any other I/O action, or perform the allowed actions in any other order.

The postcondition of our example also describes a Petri net consisting of only one place with one token. If the Hoare triple is true, then the we know the program, if it terminates, will have "obtained" the token, which it can only do by performing the action of writing 'h' and writing 'i'. In other words, the postcondition describes that if the program terminates, it will have written 'h' followed by 'i' and not have done any other I/O.

Definition 29: Proof rules ($\vdash \{P\}\, c\, \{Q\}$)

Fig. 3.12 on the facing page lists the proof rules.

The Rewrite rule of Fig. 3.12 on the next page uses the rules of Fig. 3.11. The App rule of the proof rules (i.e. of Fig. 3.12) uses fC:

$$\frac{}{\vdash \{\text{true}\}\, v\, \{\lambda v'.\, v' = v\}}\ \text{Val}$$

$$\frac{\vdash \{P\}\, c\, \{Q_1\} \quad \forall v.\, \vdash \{Q_1(v)\}\, \mathcal{C}(v)\, \{Q\}}{\vdash \{P\}\, \textbf{let}\ c\ \textbf{in}\ \mathcal{C}\, \{Q\}}\ \text{Let} \qquad \frac{(P, Q) \in \text{fC}(f)(\overline{v})}{\vdash \{P\}\, f(\overline{v})\, \{Q\}}\ \text{App}$$

$$\frac{\vdash \{P\}\, c\, \{Q\}}{\vdash \{P * R\}\, c\, \{\lambda v.\, Q(v) * R\}}\ \text{Frame} \qquad \frac{\vdash \{P_1\}\, c\, \{Q\} \quad \vdash \{P_2\}\, c\, \{Q\}}{\vdash \{P_1 \vee P_2\}\, c\, \{Q\}}\ \text{Disj}$$

$$\frac{}{\vdash \{\textbf{token}(t_1) * \mathit{bio}(t_1, v_o, v_r, t_2)\}\, \mathit{bio}(v_o)\, \{\lambda v.\, v = v_r * \textbf{token}(t_2)\}}\ \text{Bio}$$

$$\frac{P \Rrightarrow P' \quad \vdash \{P'\}\, c\, \{Q\} \quad \forall v.\, Q(v) \Rrightarrow Q'(v)}{\vdash \{P\}\, c\, \{Q'\}}\ \text{Rewrite}$$

$$\frac{\forall v.\, \vdash \{\mathcal{P}(v)\}\, c\, \{Q\}}{\vdash \{\exists \mathcal{P}\}\, c\, \{Q\}}\ \text{Exists}$$

Figure 3.12: Proof rules

Definition 30: fC

We assume a function fC that maps program function names to mathematical functions that map a list of values (i.e. the arguments) to a set of pairs of a precondition and a postcondition.

We can prove the example Hoare triple straightforwardly using these rules.

3.6.2 Hoare triples with multiple starting Petri nets

As explained in Sec. 3.4.3, a specification consists of a set of Petri nets, but in the examples so far we only wrote assertions that describe one Petri net.

One way to deal with this is to write assertions that describe sets that contain more than one Petri net. You can easily do this, e.g. write as precondition $\textbf{token}(\mathrm{T}_1) * \exists v.\ \mathsf{getchar}(\mathrm{T}_1, v, \mathrm{T}_2)$.

This has the disadvantage that the postcondition cannot access v. We solve this by using quantification at the level of the Hoare triple, like so:

$$\forall v, t_1, t_2, t_3.$$
$$\left\{\begin{array}{l}\textbf{token}(t_1) * \\ \mathsf{getchar}(t_1, v, t_2) * \\ \mathsf{putchar}(t_2, v+1, t_3)\end{array}\right\}$$

```
let x := getchar() in
putchar(x + 1);
x
```

$$\left\{\ \lambda \mathit{res}.\, \textbf{token}(t_3) * \mathit{res} = v\ \right\}$$

This expresses that for any v, t_1, t_2, t_3 a Hoare triple holds. So for $v = 7, t_1 = \mathrm{T}_1$ and $t_2 = \mathrm{T}_2$ the following Hoare triple holds:

$$\left\{\begin{array}{l}\textbf{token}(\mathrm{T}_1) * \\ \mathsf{getchar}(\mathrm{T}_1, 7, \mathrm{T}_2) * \\ \mathsf{putchar}(\mathrm{T}_2, 8, \mathrm{T}_3)\end{array}\right\}$$

```
let x := getchar() in
putchar(x + 1);
x
```

$$\left\{\ \lambda \mathit{res}.\, \textbf{token}(\mathrm{T}_3 * \mathit{res} = 7)\ \right\}$$

We will not write the quantification (such as $\forall v$) explicitly for Hoare triples anymore: when writing a Hoare triple $\vdash \{P\}\, c\, \{Q\}$ with free variables $\overline{x}$, we mean $\forall \overline{x}.\ \vdash \{P\}\, c\, \{Q\}$.

Here is a more interesting example:

$$\left\{\begin{array}{l}\textbf{token}(t_1) * \\ \mathsf{getchar}(t_1, v, t_2) * \\ \textbf{if } v < 10 \textbf{ then} \\ \quad \mathsf{putchar}(t_2, \text{‘l’}, t_3) * \\ \quad \mathsf{putchar}(t_3, \text{‘o’}, t_4) \\ \textbf{else} \\ \quad \mathsf{putchar}(t_2, \text{‘h’}, t_3) * \\ \quad \mathsf{putchar}(t_3, \text{‘i’}, t_4)\end{array}\right\}$$

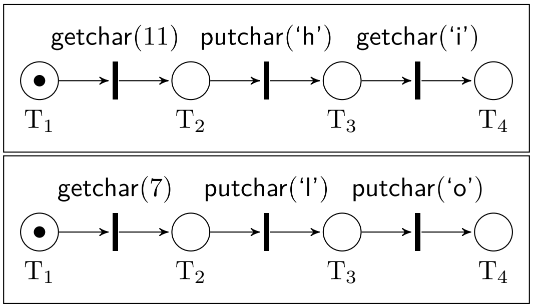


Figure 3.13: Multiple starting Petri nets for the same Hoare triple

```
let $x$ := getchar() in
if $x < 10$ then
  putchar('l');
  putchar('o')
else
  putchar('h');
  putchar('i')
$\{\ \lambda\_.\ \mathbf{token}(t_4)\ \}$
```

The specification of this program states that the program can read a number, and the subsequent actions are different depending on which number was entered. Two of the many corresponding Petri nets are in Fig. 3.13.

3.6.3 Copredicates

Consider the Unix `yes` program, which keeps on printing 'y' forever. To write a specification for `yes`, we would want the precondition to express permission to perform an infinite number of input/output actions. But of course we do not want the precondition to be of infinite length.

To deal with this, we use copredicates, which are somewhat similar to the predicates of [54]. The major difference is that copredicates allow one to express an infinite number of heap chunks, while predicates do not.

For `yes` we can write the specification using copredicates as follows:

```
copred yes_io($t_1$) = $\exists t_2$. putchar($t_1$, 'y', $t_2$) $*$ yes_io($t_2$)

function yes() =
```

$$\left\{ \begin{array}{l} \textbf{token}(t_1) * \\ \mathsf{yes\_io}(t_1) \end{array} \right\}$$

putchar('y');
yes()

$$\{\ \lambda\_.\,\text{false}\ \}$$

The assertion language syntax defined earlier (p. 51) allows one to include copredicate names with arguments, such as $\mathsf{yes\_io}(t_1)$. What does such an assertion mean? Assertions describe a set of heaps. The assertion semantics is defined coinductively in Fig. 3.10 on page 51. It says that a heap h is a model for the assertion $\mathsf{yes\_io}(t_1)$ if h is a model for the assertion $(\text{predmap}(\mathsf{yes\_io}))(t_1)$. predmap maps a predicate name to a function which maps an argument list to an assertion. In our example above, we have

$$\text{predmap}(\mathsf{yes\_io}) = \lambda\overline{v}.\begin{cases} \exists t_2.\ \mathsf{putchar}(t_1, \text{'y'}, t_2) * \mathsf{yes\_io}(t_2) & \text{if } \overline{v} = t_1 \\ \text{false} & \text{otherwise} \end{cases}$$

This is a bit awkward to write, so therefore above we wrote it in a more handy style.

So $(\text{predmap}(\mathsf{yes\_io}))(t_1)$ describes the same set of heaps as the assertion $\exists t_2.\ \mathsf{putchar}(t_1, \text{'y'}, t_2) * \mathsf{yes\_io}(t_2)$. This assertion also contains a copredicate name. Fig. 3.10 on page 51 defines the assertion semantics for copredicates, using predmap. Note that the assertion semantics of Fig. 3.10 on page 51 uses coinduction instead of induction. The heaps described by this assertion are of infinite size and contain an infinite number of chunks.

Copredicates are not only useful to describe large and infinite numbers of permissions, they also enable compositionality. We just defined the yes_io action in terms of lower-level actions.

Another use case of (co)predicates is abstraction: one can define a high-level action, without forcing the user to be exposed to the exact definition of this action.

3.6.4 no_op

The special **no_op** action is necessary when one wants to combine underspecification of the program with compositionality.

Consider the situation where the program is allowed to do one of two options, but we want one of the two options to be "empty": instead of doing either one action or another, do one action, or do nothing.

It might feel natural to just write this (incorrectly) as follows:

$$\begin{aligned}&\textbf{copred}\ \mathsf{maybe\_putchar\_io}(t_1, c, t_2) =\\&\quad \mathsf{putchar}(t_1, c, t_2) * t_1 = t_2\end{aligned}$$

This is not the right way to express optional I/O. Consider the following example:

$$\begin{aligned}&\textbf{copred}\ \mathsf{maybe\_a\_or\_maybe\_b\_io}(t_1, c, t_2) =\\&\quad \mathsf{maybe\_putchar\_io}(t_1, \text{‘}\mathsf{a}\text{’}, t_2) *\\&\quad \mathsf{maybe\_putchar\_io}(t_1, \text{‘}\mathsf{b}\text{’}, t_2)\end{aligned}$$

This does not express to maybe print ‘a’, or maybe print ‘b’. Instead, it expresses to print either ‘a’ and ‘b’, or ‘b’ and ‘a’, or only ‘b’, or only ‘a’, or nothing.

The correct way to write optional I/O like this:

$$\textbf{copred}\ \mathsf{maybe\_putchar\_io}(t_1, c, t_2) = \mathsf{putchar}(t_1, c, t_2) * \textbf{no\_op}(t_1, t_2)$$

One can think of **no_op** as a “dummy” I/O action that represents not actually doing input or output. With this definition of maybe_putchar_io, the definition of
maybe_a_or_maybe_b_io has the intended meaning.

3.6.5 Interleaving

Consider the example where we want to mix the actions of printing ‘a’ followed by printing ‘b’, with the actions of printing ‘c’ followed by printing ‘d’. Note that ‘a’ must always be printed before printing ‘b’, and ‘c’ must be printed before printing ‘d’. For this example, we could define an I/O action like this:

$$\begin{array}{llll}\multicolumn{4}{l}{\textbf{copred}\ \mathsf{mix\_ab\_and\_cd\_io}(t_1, t_2) =}\\ \mathsf{putchar}(t_1, \text{‘a’}, t_a) & * \mathsf{putchar}(t_a, \text{‘b’}, t_{11}) & * \mathsf{putchar}(t_{11}, \text{‘c’}, t_{12}) & * \mathsf{putchar}(t_{12}, \text{‘d’}, t_2)\\ & * \mathsf{putchar}(t_a, \text{‘c’}, t_{21}) & * \mathsf{putchar}(t_{21}, \text{‘b’}, t_{22}) & * \mathsf{putchar}(t_{22}, \text{‘d’}, t_2)\\ & & * \mathsf{putchar}(t_{21}, \text{‘d’}, t_{32}) & * \mathsf{putchar}(t_{32}, \text{‘b’}, t_2)\\ \mathsf{putchar}(t_1, \text{‘c’}, t_c) & * \mathsf{putchar}(t_c, \text{‘a’}, t_{41}) & * \mathsf{putchar}(t_{41}, \text{‘b’}, t_{42}) & * \mathsf{putchar}(t_{42}, \text{‘d’}, t_2)\\ & & * \mathsf{putchar}(t_{41}, \text{‘d’}, t_{52}) & * \mathsf{putchar}(t_{52}, \text{‘b’}, t_2)\\ & * \mathsf{putchar}(t_c, \text{‘d’}, t_{61}) & * \mathsf{putchar}(t_{61}, \text{‘a’}, t_{62}) & * \mathsf{putchar}(t_{62}, \text{‘b’}, t_2)\end{array}$$

We can instead write it using split and join:

```
**copred** mix_ab_and_cd_io$(t_1, t_2) =$
  $\mathbf{split}(t_1, t_{ab1}, t_{cd1}) *$
  putchar$(t_{ab1}, $'a'$, t_{ab2}) *$ putchar$(t_{ab2}, $'b'$, t_{ab3}) *$
  putchar$(t_{cd1}, $'c'$, t_{cd2}) *$ putchar$(t_{cd2}, $'d'$, t_{cd3}) *$
  $\mathbf{join}(t_{ab3}, t_{cd3}, t_2)$
```

For this example it is still somewhat practical to avoid split and join if we want to, by writing all possible interleavings by hand. For bigger examples this will not be practical anymore. Writing all interleavings by hand forces us to build higher-level I/O actions only on top of BIO actions (instead of also on top of other higher-level I/O actions), so it breaks compositionality. Split and join are therefore for bigger programs not just a nice-to-have but are necessary to support compositionality and interleaving at the same time. Combining interleaving and compositionality is illustrated by the following example:

```
**copred** printstr_io$(t_1, str, t_2) =$
  **if** $str =$ “” **then**
    $\mathbf{no\_op}(t_1, t_2)$
  **else**
    $\exists t_m.$ putchar$(t_1, \mathrm{head}(str), t_m) *$
    printstr_io$(t_m, \mathrm{tail}(str), t_2)$

**copred** mix_hello_and_BYE_io$(t_1, t_2) =$
  $\mathbf{split}(t_1, t_{h1}, t_{b1}) *$
  printstr_io$(t_{h1}, $“hello”$, t_{h2}) *$
  printstr_io$(t_{b1}, $“BYE”$, t_{b2}) *$
  $\mathbf{join}(t_{h2}, t_{b2}, t_2)$
```

The mix_hello_and_bye_io action expresses the action of printing “hello” and “BYE” interleaved. A precondition $\mathbf{token}(t_1) *$ mix_hello_and_BYE_io(t_1, t_2) allows the program to print one of “helloBYE”, “hBeYlElo”, “heBYloE”, and so on. It does not allow the program to print “ollehBYE”.

3.6.6 Soundness

We now prove soundness of the proof rules for Hoare triples.

Definition 31: Whether all function bodies are proven

We say *all function bodies are proven* if

$$\forall f, \overline{v}, P, Q.\ (P, Q) \in \mathrm{fC}(f)(\overline{v}) \Rightarrow \vdash \left\{P\right\} \mathrm{fc}(f)(\overline{v}) \left\{Q\right\}$$

Instead of using the definition of the weakest precondition directly to show that a heap satisfies the weakest precondition of a command and a postcondition, one can use the Hoare proof rules instead:

Theorem 3

If all function bodies are proven, then

$$\forall P, c, Q, h.\ \vdash \left\{P\right\} c \left\{Q\right\} \wedge h \vDash P \Rightarrow \left\lceil \mathrm{wp}(c, \lceil \lambda v, h'.\, h' \vDash Q(v) \rceil) \right\rceil (h)$$

Proof. Full proof in Appendix A.4 on page 171. □

Now we can easily prove the main soundness theorem:

Theorem 4: Soundness

If all function bodies are proven, then
$\forall P, c, Q.\ \vdash \left\{P\right\} c \left\{Q\right\} \Rightarrow \models \left\{P\right\} c \left\{Q\right\}$

Proof. Follows directly from Theorem 3 and Theorem 2 on page 50. □

The main soundness theorem states that if all Hoare triples of the program functions under consideration are provable using the proof rules, then for any Hoare triple that is provable the following property holds: for any heap that satisfies the precondition, and any execution of the program, the trace of I/O actions of the execution is allowed by that heap. Furthermore, if the execution terminates and the world did not violate promises of the initial heap, then the postcondition must hold for the return value.

We mechanically proved a soundness theorem in the Coq proof assistant, but it uses trace triples similar to [60] instead of the safe relation, it does not support nonterminating recursion in the proof rules, and assertion satisfaction does not allow leaking[5] (but the proof rules do). The proof uses the programming language semantics directly (instead of the weakest precondition semantics). It is available at [58].

3.7 Examples

We give some examples of programs performing I/O and their specifications.

3.7.1 Tee

The specification of Fig. 3.14 on the facing page allows the program to read from stdin (until EOF, represented as reading a negative number) and meanwhile writing what the program reads to stdout and also to stderr. The specifications do not enforce any buffer size, so the implementation can choose how much to buffer. In Fig. 3.14 on the next page, the implementation chooses a buffer size of two. The specification does not enforce whether writing to stderr or stdout has to happen first.

Note that the specifications are written in a compositional manner: the tee action is built on top of reads and tee_outs actions, which are built on top of stdin, stdout, and stderr actions (which we consider BIOs in this example, though it does not have to).

A proof outline of the tee_out function and of the main function is given in Fig. 3.15 on page 62.

3.7.2 Read files mentioned in a file

The specifications of Fig. 3.16 on page 63 allow the program to read a file with filename "f", which contains a list of filenames, each of length one, until EOF is encountered (represented as reading a value less than zero). Filenames that are not in 7bit-ASCII or that consist of the zero character, are ignored. These files are read (in that order), and their contents is printed (also in the same order). The specifications do not enforce how much from "f" is read before the next file is read, or how many files are read before printing them, or how much from one of the input files is buffered before printing it.

[5] i.e. it is not true that given $h_1 \vDash P$ one can always conclude that $h_1 \uplus h_2 \vDash P$.

```
**copred** reads($t_1, text, t_3$) =
$\exists t_2, c, sub.$
   getchar($t_1, c, t_2$)
   $*$ **if** $c < 0$ **then** $text = \langle\rangle * t_3 = t_2$
     **else**
       (reads($t_2, sub, t_3$) $* \ text = c :: sub$)

**copred** tee_out($t_1, c, t_2$) =
$\exists t_{p1}, t_{p2}, t_{r1}, t_{r2}.$
   **split**($t_1, t_{p1}, t_{r1}$)
   $*$ stdout($t_{p1}, c, \_, t_{p2}$)
   $*$ stderr($t_{r1}, c, \_, t_{r2}$)
   $*$ **join**($t_{p2}, t_{r2}, t_2$)

**function** tee_out($c$) =
   { **token**($t_1$) $*$ tee_out($t_1, c, t_2$) }
     stdout($c$);
     stderr($c$)
   { $\lambda\_.$ **token**($t_2$) }

**copred** tee_outs($t_1, text, t_2$) =
$\exists t_{out}.$
   **if** $text = \langle\rangle$ **then** $t_2 = t_1$ **else** (
     tee_out($t_1$, head($text$), $t_{out}$)
     $*$ tee_outs($t_{out}$, tail($text$), $t_2$) )

**copred** tee($t_1, text, t_2$) =
$\exists t_{r1}, t_{w1}, t_{r2}, t_{w2}.$
   **split**($t_1, t_{r1}, t_{w1}$)
   $*$ reads($t_{r1}, text, t_{r2}$)
   $*$ tee_outs($t_{w1}, text, t_{w2}$)
   $*$ **join**($t_{r2}, t_{w2}, t_2$)

**copred** invariant($t_2$) =
$\exists t_{r1}, t_{r2}, t_{w1}, t_{w2}, text.$
   **token**($t_{r1}$)
   $*$ reads($t_{r1}, text, t_{r2}$)
   $*$ **token**($t_{w1}$)
   $*$ tee_outs($t_{w1}, text, t_{w2}$)
   $*$ **join**($t_{r2}, t_{w2}, t_2$)

**function** iter() =
   { invariant($t_2$) }
     **let** $c_1$ := getchar() **in**
     **if** $c_1 \geq 0$ **then**
       **let** $c_2$ := getchar() **in**
       tee_out($c_1$);
       **if** $c_2 \geq 0$ **then**
         tee_out($c_2$);
         iter()
       **else** unit
     **else** unit
   { $\lambda\_.$ **token**($t_2$) }

**function** main() =
   { **token**($t_1$) $*$ tee($t_1, text, t_2$) }
     iter()
   { $\lambda\_.$ **token**($t_2$) }
```

Figure 3.14: Specification and implementation of the Tee program

```
function tee_out(c) =
  { token(t_1)
    * tee_out(t_1, c, t_2) }
  { token(t_{p1})
    * stdout(t_{p1}, c, _, t_{p2})
    * token(t_{r1})
    * stderr(t_{r1}, c, _, t_{r2})
    * join(t_{p2}, t_{r2}, t_2) }
stdout(c);
  { token(t_{p2}) * token(t_{r1})
    * stderr(t_{r1}, c, _, t_{r2})
    * join(t_{p2}, t_{r2}, t_2) }
stderr(c)
  { token(t_{p2}) * token(t_{r2})
    * join(t_{p2}, t_{r2}, t_2) }
  { λ_. token(t_2) }

function iter() =
    { invariant(t_2) }
  let c_1 := read() in
  if c_1 ≥ 0 then (
      { token(t_{rb1})
        * reads(t_{rb1}, sub, t_{r2})
        * token(t_{w1})
        * tee_outs(t_{w1},
              c_1 :: sub, t_{w2})
        * join(t_{r2}, t_{w2}, t_2) }
    let c_2 := read() in
      { token(t_{rb2}) * token(t_{w1})
        * if c_2 ≥ 0 then
            reads(t_{rb2}, subsub, t_{r2})
            * tee_outs(t_{w1},
                  c_1 :: c_2 :: subsub, t_{w2})
        else (
            t_{r2} = t_{rb2}
            * tee_outs(t_{w1},
                  c_1 :: ⟨⟩, t_{w2}) )
        * join(t_{r2}, t_{w2}, t_2) }
    tee_out(c_1);
    if c_2 ≥ 0 then (
        { token(t_{rb2})
          * reads(t_{rb2}, subsub, t_{r2})
          * token(t_{wb1})
          * tee_outs(t_{wb1},
                c_2 :: subsub, t_{w2})
          * join(t_{r2}, t_{w2}, t_2) }
      tee_out(c_2);
        { invariant(t_2) }
      iter()
    ) else unit
  ) else unit
  { λ_. token(t_2) }

function main() =
    { token(t_1) * tee(t_1, text, t_2) }
  iter()
    { λ_. token(t_2) }
```

Figure 3.15: Proof outline of the tee_out and main function of the tee program ($\langle\rangle$ is the empty list)

```
copred freads($t_1, f, text, t_{end}$) =
$\exists c, sub, t_2$.
  fread($t_1, f, c, t_2$)
  $*$ if $c \geq 0$ then
    freads($t_2, f, sub, t_{end}$)
    $*$ $text = c :: sub$
  else (
    $t_{end} = t_2$
    $*$ $text = \langle\rangle$)

copred get_file($t_1, name, text,$
$t_{end}$) =
$\exists fh, t_2, t_3$.
  fopen($t_1, name, fh, t_2$)
  $*$ freads($t_2, fh, text, t_3$)
  $*$ fclose($t_3, fh, t_{end}$)

copred prints($t_1, text, t_{end}$) =
$\exists t_2$.
  if $text = \langle\rangle$ then
    no_op($t_1, t_{end}$)
  else (
    print($t_1$, head($text$), _, $t_2$)
    $*$ prints($t_2$, tail($text$), $t_{end}$))

copred get_files($t_1, fs, text, t_{end}$) =
$\exists text1, text2, t_2, f, sub$.
  if $fs = \langle\rangle$ then
    no_op($t_1, t_{end}$)
    $*$ $text = \langle\rangle$
  else (
    get_file($t_1, f :: \langle\rangle, text1, t_2$)
    $*$ get_files($t_2, sub, text2, t_{end}$)
    $*$ $fs = f :: sub$
    $*$ $text = text1 ++ text2$)

copred read_fname_list($t_1, fh,$
$fs, t_{end}$) =
$\exists c, t_2, sub$.
  fread($t_1, fh, c, t_2$)
  $*$ if $c \geq 0$ then
    read_fname_list($t_2, fh, sub, t_{end}$)
    $*$ if $c > 0 \wedge c \leq 127$ then
      $fs = c :: sub$
    else
      $fs = sub$
  else (
    $t_{end} = t_2$
    $*$ $fs = \langle\rangle$)

copred main($t_1, fname, t_{end}$) =
$\exists t_2, t_{meta}, t_{rw}, fs, t_{meta2}, t_r, t_w,$
$t_{meta3}, t_{r2}, t_{w2}, t_{rw2}, fh$ .
  fopen($t_1, fname, fh, t_2$)
  $*$ split($t_2, t_{meta}, t_{rw}$)
    $*$ read_fname_list($t_{meta}, fh,$
      $fs, t_{meta2}$)
    $*$ fclose($t_{meta2}, fh, t_{meta3}$)
    $*$ split($t_{rw}, t_r, t_w$)
      $*$ get_files($t_r, fs, text, t_{r2}$)
      $*$ prints($t_w, text, t_{w2}$)
    $*$ join($t_{r2}, t_{w2}, t_{rw2}$)
  $*$ join($t_{meta3}, t_{rw2}, t_{end}$)

function main() =
  $\left\{ \begin{array}{l} \textbf{token}(t_1) \\ * \text{main}(t_1, \text{`f'} :: \langle\rangle, t_2) \end{array} \right\}$
  ...
  $\{\ \lambda\_.\ \textbf{token}(t_2)\ \}$
```

Figure 3.16: Specification of a program that prints the contents of all files whose filenames are in a given list. This list is not static, it is read from a file “f”.

```
copred brackets($t_1, t_2$) =
$\exists t_{open}, t_{center}, t_{close}.$
    no_op($t_1, t_2$)
    $*$ print($t_1$, ‘(’, _, $t_{open}$)
    $*$ brackets($t_{open}, t_{center}$)
    $*$ print($t_{center}$, ‘)’, _, $t_{close}$)
    $*$ brackets($t_{close}, t_2$)
```

```
function main() =
    { token($t_1$) $*$ brackets($t_1, t_2$) }
    print(‘(’);
    print(‘)’)
    { token($t_2$) }
```

Figure 3.17: Specification of a program that is allowed to output any string of the matching brackets grammar.

The specifications are again built on top of each other, up to standard library functions like fopen and fclose. They can be considered as BIOs, but this does not have to be the case.

An implementation is left out, but is included in the VeriFast examples (see Sec. 3.7.5).

3.7.3 Print any string of the grammar of matching brackets

The specifications of Fig. 3.17 state the program is allowed to print any string consisting of open and closing brackets, as long as they match up. More precisely, the program is allowed to print any string of the following grammar:

$B ::= (B)B \mid \epsilon$

Here, ϵ denotes the empty string.

Note that many implementations are a valid implementation for these specifications, including those that utilize a random number generator.

3.7.4 Turing machine

This example defines a copredicate

$$\textbf{copred}\ \mathsf{tm}(t_1, \mathit{encoding}, \mathit{initialstate}, \mathit{tapeleft}, \mathit{taperight}, t_2)$$

The second argument is an encoding of a Turing machine. It is a linearization into a list of integers of the transition table of a Turing machine.

A Turing machine's transition table normally associates a state and a symbol to an action (move tape left, move tape right, do not move tape) and a symbol to be written on the tape. Depending on the current state, and the symbol under the read head of the tape, the transition table defines what the Turing machine must do next (in case of a deterministic machine) or can do next (in case of a non-deterministic one).

Instead of considering input and output as the state of the tape at startup and termination time of the Turing machine, we add an input and an output action. The input action reads something from the world and writes it to the current position on the tape. The output action reads the current symbol from the tape and "prints" it to the world. Both these actions do not move the tape. Adding these actions has the advantage that the Turing machine does not have to terminate and can still perform input/output, which would not be the case if we only consider output as the tape when a Turing machine ends.

Note that the Turing machine is an argument of the copredicate: it is not hardcoded and the predicate definition is usable for any Turing machine.

A program's implementation that has a specification of the form

$$\{\ \mathbf{token}(t_1) * \mathsf{tm}(t_1, \mathit{tm}, \mathit{initstate}, \mathit{tapeleft}, \mathit{taperight}, t_2)\ \}$$

$\dots$

$$\{\ \mathbf{token}(t_2)\ \}$$

is allowed to perform the input/output actions the Turing machine (given as argument *tm*) performs, but no other actions. The implementation is only allowed to terminate if the Turing machine terminates and upon termination it must have performed all the input/output actions the Turing machine performs when executing.

The copredicate supports underspecified input/output (the implementation can choose to do X or Y) by using non-deterministic Turing machines.

The exact definitions of this example are left out, but annotated verified C versions are shipped in the examples of VeriFast (see Sec. 3.7.5).

3.7.5 Mechanical verification of the examples

The examples of Sec. 3.7, Sec. 4.3, and Sec. 4.6.2 have been implemented in C and verified with the VeriFast [34] program verifier. These files are available in the directory `examples/io` shipped with VeriFast[37], or directly at `https://github.com/verifast/verifast/tree/v17.06/examples/io`.

Note that VeriFast does not just hunt for bugs. Instead it proves the program does not violate its specification (and does not crash), for every possible execution. In the context of the examples, this means VeriFast proves the implementation does not violate any properties expressed by the I/O style specification: the program never performs disallowed I/O actions and the I/O actions performed are correct and in the correct order.

VeriFast performs little automation. Predicates and copredicates are not always folded and unfolded automatically and the user has to write annotations by hand to fold and unfold. Therefore, VeriFast typically has fast execution times at the cost of high annotation overhead. This is also the case with the examples of this thesis.

The following table lists the size and annotation overhead of the examples, and how long VeriFast takes to verify them.

			Lines			
Example	Sec.	page	C	Annot.	Mixed	Time (ms)
Tee	3.7.1	60	27	55	1	292
Read files mentioned in a file	3.7.2	60	20	91	0	148
Matching brackets	3.7.3	64	4	20	0	134
Turing machine	3.7.4	64	7	58	0	149
Non-reusable *print_hi*	4.3	87	27	170	1	183
Writer cat reader	4.6.2	146	258	935	14	1703

The reported timings are using the Redux SMT solver on a Intel i5-2400 CPU (max value of 10 runs) using a native build of the VeriFast development version of 2017-03-30.

3.8 A monoid for verifying input/output

In this section we sketch how one can implement the I/O verification style in Iris. Iris [38] is a generic framework for reasoning about concurrent programs that manipulate a shared resource. As a shared resource we use the trace of I/O actions that have been performed so far. Performing an I/O action can thus be considered as modifying the shared state (i.e. the trace) by adding an I/O action at the end of the trace.

We consider three examples that we will verify using Iris. This is the first example:

$$\left\{ \mathbf{token}(t_1) * \mathsf{putchar}(t_1, c, t_2) \right\}$$

$\mathsf{putchar}(c)$

$$\left\{ \mathbf{token}(t_2) \right\}$$

It consists of a program that prints a character c. We will use Iris to verify that the implementation does not violate the specification, which states that the program is allowed to print c. It does not allow the program to print a character twice or to perform any other I/O actions. The specification also states that any execution of the program that terminates, must have printed c upon termination.

This is the second example:

$$\left\{ \mathbf{token}(t_1) * \mathsf{putchar}(t_1, \text{`}\mathsf{l}\text{'}, t_2) * \mathsf{putchar}(t_2, \text{`}\mathsf{o}\text{'}, t_3) \right\}$$

$\mathsf{putchar}(\text{`}\mathsf{l}\text{'});$

$\mathsf{putchar}(\text{`}\mathsf{o}\text{'})$

$$\left\{ \mathbf{token}(t_3) \right\}$$

The specification states the program is allowed to print the character ‘l’ and the character ‘o’ but only in this order. It does not allow to first print ‘o’. Remember that for I/O the order is often important. For example, it is important to first perform the I/O action of putting on the shield and only after that perform the I/O action of turning on the laser, instead of the other way around. The specification also states that any execution of the program that terminates, must have performed the specified I/O upon termination.

This is the third example:

$$\left\{\begin{array}{l}\mathbf{token}(t_1)\\ *\ \mathbf{split}(t_1,t_2,t_3) * \mathsf{putchar}(t_2,\text{‘h’},t_{22}) * \mathsf{putchar}(t_3,\text{‘i’},t_{32}) * \mathbf{join}(t_{22},t_{32},t_4)\end{array}\right\}$$

$\mathsf{putchar}(\text{‘h’}) \mathbin{||} \mathsf{putchar}(\text{‘i’})$

$$\left\{\ \mathbf{token}(t_4)\ \right\}$$

The specification says the program is allowed to print ‘h’ and to print ‘i’, but leaves the order unspecified: the program can choose to first print ‘h’ and then ‘i’, or to first print ‘i’ and then ‘h’. The specification also says that any terminating execution of the program must have printed both characters upon termination.

The implementation of the example prints ‘h’ and ‘i’ concurrently. Therefore there exists an execution that prints ‘h’ before printing ‘i’, and there also exists an execution that prints ‘i’ before printing ‘h’.

We will verify, using Iris, that these three example programs do not violate their specifications.

Monoid In order to use Iris, it must be provided an Iris style monoid. This monoid is used to represent knowledge about the shared resource, such as a heap or a counter. In our case, the shared resource is the trace of I/O actions that have happened so far.

An Iris style monoid is a quadruple that consists of:

- A carrier set
- An element in the carrier set called the error element
- An element in the carrier set called the neutral element
- A binary operation on the carrier set, that adheres to some properties.

Carrier set Before studying how such a monoid can be used, we define the I/O monoid. We first define the carrier set of the monoid informally.

The elements of the carrier set of the I/O monoid are triples $(\hat{\tau}, i, h)$.

$$\frac{h_1 \xrightarrow{bio(v_o,v_i)} h \qquad h \xrightarrow{\tau}^* h_2}{h_1 \xrightarrow{bio(v_o,v_i)\cdot\tau}^* h_2} \qquad\qquad \frac{h_1 \xrightarrow{bio(v_o,v_i)} h \qquad v_i \neq v_i'}{h_1 \xrightarrow{bio(v_o,v_i')\cdot\tau}^* h_2}$$

$$\frac{h_1 \xrightarrow{\epsilon} h \qquad h \xrightarrow{\tau}^* h_2}{h_1 \xrightarrow{\tau}^* h_2}$$

Figure 3.18: The relation $h \xrightarrow{\tau}^* h$

- $\hat{\tau}$ can be a trace τ, or it can be ϵ. If $\hat{\tau}$ is a trace, then it represents the actual trace of input/output actions performed so far (by all threads). If $\hat{\tau} = \epsilon$, then it represents absence of knowledge of the input/output actions performed so far. If $\hat{\tau} = \langle\rangle$, i.e. the trace is empty, it expresses knowledge that there is no input/output performed. Note the difference between absence of knowledge of I/O and knowledge of absence of I/O.
- h is a heap that represents the local permissions to perform input/output actions in the future. By "local" we mean only of the thread under consideration, not of all threads combined.
- i represents the set of permissions to perform input/output actions "initially" at the time we conceptually start recording $\hat{\tau}$ and $\hat{\tau}$ was therefore still the empty trace $\langle\rangle$. i can also be equal to ϵ. Then it expresses absence of knowledge of the initial permissions.

Not all such triples are a member of the carrier set. More specifically, we only consider triples where h is a subheap of one of the possible heaps that can be obtained by performing the I/O actions in $\hat{\tau}$ starting from i.

We now formally define the carrier set of the I/O monoid:

$$\begin{aligned}|M| = \{\bot\} \cup \{(\hat{\tau}, i, h) \mid\ & \hat{\tau} \in |\text{Ex}(\text{Traces})|^+ \land \\ & i \in \text{Heaps} \cup \{\epsilon\} \land \\ & h \in \text{Heaps} \land \\ & \left(\hat{\tau} \neq \epsilon \land i \neq \epsilon \Rightarrow \exists h' \in \text{Heaps}.\ i \xrightarrow{\hat{\tau}}^* h' \land h \subseteq h'\right) \\ \}\end{aligned}$$

Here, the relation $h_1 \xrightarrow{\tau}^* h_2$ expresses that one can perform the actions in τ given the permissions in h_1 and that performing these actions can yield h_2. It is defined inductively in Fig. 3.18 on the previous page.

$|\text{Ex}(\text{Traces})|^+$ is the set $\text{Traces} \cup \{\epsilon\}$. The set Traces is the set of all traces as defined in Sec. 3.3, although we don't actually need that traces can be infinite.

The error element of the monoid is $\bot$. The neutral element is $(\epsilon, \epsilon, \{\!\!\}\})$.

Operation Besides a carrier set, an Iris style monoid also has a binary operation. When considering elements of the monoid as representation of knowledge, performing the operation on the elements can be considered as combining knowledge. It is possible the two "knowledges" from the two elements contradict each other. In that case, the operation will return $\bot$, i.e. if the knowledge encoded in an element m_1 of the carrier set contradicts the knowledge of m_2, then $m_1 \cdot m_2 = \bot$.

The operation of the I/O monoid is defined as follows:

- $(\hat{\tau}_1, i_1, h_1) \cdot (\hat{\tau}_2, i_2, h_2) = (\hat{\tau}_1 \cdot \hat{\tau}_2, i_1 \cdot i_2, h_1 \uplus h_2)$ if this is an element of $|M|$.
- $m_1 \cdot m_2 = \bot$ for all other cases

This definition uses an operation on $|\text{Ex}(\text{Traces})|^+ \cup \{\bot\}$, that is defined as follows ($\hat{\tau}$ ranges over $|\text{Ex}(\text{Traces})|^+$):

- $\epsilon \cdot \hat{\tau} = \hat{\tau}$ and $\hat{\tau} \cdot \epsilon = \hat{\tau}$
- $\_ \cdot \_ = \bot$ for all other cases

The operation on the I/O monoid also uses an operation on $\text{Heaps} \cup \{\epsilon, \bot\}$, which is defined as follows (i ranges over $\text{Heaps} \cup \{\epsilon\}$, h ranges over Heaps):

- $i \cdot \epsilon = i$ and $i \cdot \epsilon = i$
- $h \cdot h = h$
- $\_ \cdot \_ = \bot$ for all other cases. Note that $h_1 \cdot h_2 = \bot$ if $h_1 \neq h_2$.

Assertions In Iris, the assertion $\lfloor m \rfloor$ denotes the physical state is represented by m where m is an element of the carrier of a monoid. In our case, $\lfloor \tau \rfloor$ means the actions in the trace τ are the I/O actions that have happened so far (in that order). In other use cases of Iris, the physical state could be a classic heap (mapping addresses to values) or the content of a counter, but we will only focus on I/O here.

In Iris, the assertion $\boxed{m}$ expresses "ownership" of m where m is an element of the carrier of a monoid. Multiple threads can have such ownerships. You can think of it as follows: there exists a ghost state m, and $m = m_1 \cdot m_2 \cdot m_3 \cdot ...$ where the ownerships $\boxed{m_1}$, $\boxed{m_2}$, $\boxed{m_3}$, ... are spread over threads.

A property of an assertion of the form $\boxed{m}$ is that $\boxed{m_1 \cdot m_2}$ is the same as the assertion $\boxed{m_1} * \boxed{m_2}$. So you can swap one with the other.

In our case, we will use assertions $\boxed{m}$ where m are elements of the carrier of the I/O monoid, i.e. $m \in |M|$.

We translate the assertions as used in the I/O verification approach to assertions in the Iris framework as follows:

- $\mathbf{token}(t) \Leftrightarrow \boxed{(\epsilon, \epsilon, \{\!\{\mathbf{token}(t)\}\!\})}$
- $bio(t_1, v_o, v_i, t_2) \Leftrightarrow \boxed{(\epsilon, \epsilon, \{\!\{bio(t_1, v_o, v_i, t_2)\}\!\})}$
- $\mathbf{split}(t_1, t_2, t_3) \Leftrightarrow \boxed{(\epsilon, \epsilon, \{\!\{\mathbf{split}(t_1, t_2, t_3)\}\!\})}$
- $\mathbf{join}(t_1, t_2, t_3) \Leftrightarrow \boxed{(\epsilon, \epsilon, \{\!\{\mathbf{join}(t_1, t_2, t_3)\}\!\})}$

Frame preserving updates Now let us see how we can update this ghost state. We cannot simply replace any ghost state $\boxed{m_1}$ into any $\boxed{m_2}$: it is possible some other thread has a $\boxed{m_3}$ that conflicts with $\boxed{m_2}$, in other words: $m_3 \cdot m_2 = \bot$.

More generally, Iris uses the concept of a *frame preserving update*. We say there is a frame preserving update from an element m of (the carrier set of) a monoid to a set B of elements of the same monoid, denoted $m \rightsquigarrow B$, iff for every element f of the monoid that does not conflict with m, there is an element in B that does not conflict with f. In other words:

$$m \rightsquigarrow B \Leftrightarrow \forall f. f \cdot m \neq \bot \Rightarrow \exists b \in B.\ f \cdot b \neq \bot$$

Let us look at which frame preserving updates hold for the I/O monoid.

The following frame preserving update corresponds to performing an I/O action:

$$(\tau, \epsilon, \{\!\{\mathbf{token}(t_1), bio(t_1, v, r, t_2)\}\!\} \uplus h) \rightsquigarrow \{(\tau +\!\!+\, bio(v, r) :: \langle\rangle, \epsilon, \{\!\{\mathbf{token}(t_2)\}\!\} \uplus h)\}$$

This corresponds to performing a split:

$$(\epsilon, \epsilon, \{\!\!\{\mathbf{token}(t_1), \mathbf{split}(t_1, t_2, t_3)\}\!\!\} \uplus h) \rightsquigarrow \{(\epsilon, \epsilon, \{\!\!\{\mathbf{token}(t_2), \mathbf{token}(t_3)\}\!\!\} \uplus h)\}$$

And this corresponds to performing a join:

$$(\epsilon, \epsilon, \{\!\!\{\mathbf{token}(t_1), \mathbf{token}(t_2), \mathbf{join}(t_1, t_2, t_3)\}\!\!\} \uplus h) \rightsquigarrow \{(\epsilon, \epsilon, \{\!\!\{\mathbf{token}(t_3)\}\!\!\} \uplus h)\}$$

Note that you cannot perform a frame preserving update to create more permissions. For example:

$$(\hat{\tau}, i, \{\!\!\{\mathbf{token}(t_1)\}\!\!\} \uplus h) \not\rightsquigarrow \{(\hat{\tau}, i, \{\!\!\{\mathbf{token}(t_1), \mathit{bio}(t_1, v, r, t_2)\}\!\!\} \uplus h)\}$$

This is the reason why elements of the carrier set of the I/O monoid remember the initial heap ("i"): it prevents creation of new permissions.

Also, you cannot perform a frame preserving update that performs I/O without actually modifying the trace of performed I/O. For example:

$$m \not\rightsquigarrow \{b\}$$

where

$$m = (\epsilon, \epsilon, \{\!\!\{\mathbf{token}(t_1), \mathit{bio}(t_1, v, r, t_2)\}\!\!\})$$

and

$$b = (\epsilon, \epsilon, \{\!\!\{\mathbf{token}(t_2)\}\!\!\})$$

This is not a frame preserving update since for $f = (\langle\rangle, \{\!\!\{\mathbf{token}(t_1), \mathit{bio}(t_1, v, r, t_2)\}\!\!\}, \{\!\!\{\}\!\!\})$ we have $f \cdot m \neq \bot$ while $f \cdot b = \bot$.

Invariant While an assertion $\boxed{\mathrm{m}}$ expresses ghost ownership, it does not directly state anything about the physical state. Iris uses invariants to link the ghost state with the physical state. For the I/O monoid, we use the following invariant:

$$\mathrm{IoInv} = \exists \tau \in \mathrm{Traces}.\ \boxed{(\tau, \epsilon, \{\!\!\{\}\!\!\})} * \lfloor \tau \rfloor$$

Now we illustrate with an example how the physical state can be updated. In this example, we sketch a proof outline of the first example program:

$$\left\{\ \mathbf{token}(t_1) * \mathsf{putchar}(t_1, c, t_2)\ \right\}$$
$\mathsf{putchar}(c)$
$$\left\{\ \mathbf{token}(t_2)\ \right\}$$

The proof outline sketch is as follows:

$$\left\{ \mathbf{token}(t_1) * \mathsf{putchar}(t_1, c, t_2) \right\}_{\mathcal{E} \uplus \{\mathrm{IoInv}\}}$$

$$\left\{ \boxed{(\epsilon, \epsilon, \{\!\{\mathbf{token}(t_1)\}\!\})} * \boxed{(\epsilon, \epsilon, \{\!\{\mathsf{putchar}(t_1, c, t_2)\}\!\})} \right\}_{\mathcal{E} \uplus \{\mathrm{IoInv}\}}$$

$$\left\{ \begin{array}{l} \boxed{(\epsilon, \epsilon, \{\!\{\mathbf{token}(t_1)\}\!\})} * \boxed{(\epsilon, \epsilon, \{\!\{\mathsf{putchar}(t_1, c, t_2)\}\!\})} * \\ \exists \tau \in \mathrm{Traces} \,.\, \boxed{(\tau, \epsilon, \{\!\{\}\!\})} * \lfloor \tau \rfloor \end{array} \right\}_{\mathcal{E}} \quad \text{(open invariant)}$$

$$\left\{ \boxed{(\epsilon, \epsilon, \{\!\{\mathbf{token}(t_1)\}\!\})} * \boxed{(\epsilon, \epsilon, \{\!\{\mathsf{putchar}(t_1, c, t_2)\}\!\})} * \boxed{(\tau, \epsilon, \{\!\{\}\!\})} * \lfloor \tau \rfloor \right\}_{\mathcal{E}}$$

$$\left\{ \boxed{(\epsilon, \epsilon, \{\!\{\mathbf{token}(t_1)\}\!\}) \cdot (\epsilon, \epsilon, \{\!\{\mathsf{putchar}(t_1, c, t_2)\}\!\}) \cdot (\tau, \epsilon, \{\!\{\}\!\})} * \lfloor \tau \rfloor \right\}_{\mathcal{E}}$$

$$\left\{ \boxed{(\tau, \epsilon, \{\!\{\mathbf{token}(t_1), \mathsf{putchar}(t_1, c, t_2)\}\!\})} * \lfloor \tau \rfloor \right\}_{\mathcal{E}}$$

$$\left\{ \boxed{(\tau +\!\!+ \mathsf{putchar}(c) :: \langle\rangle, \epsilon, \{\!\{\mathbf{token}(t_2)\}\!\})} * \lfloor \tau \rfloor \right\}_{\mathcal{E}} \quad \text{(frame preserving update)}$$

$\mathsf{putchar}(c)$

$$\left\{ \boxed{(\tau +\!\!+ \mathsf{putchar}(c) :: \langle\rangle, \epsilon, \{\!\{\mathbf{token}(t_2)\}\!\})} * \lfloor \tau +\!\!+ \mathsf{putchar}(c) :: \langle\rangle \rfloor \right\}_{\mathcal{E}}$$

$$\left\{ \boxed{(\epsilon, \epsilon, \{\!\{\mathbf{token}(t_2)\}\!\})} * \exists \tau' \in \mathrm{Traces} \,.\, \boxed{(\tau', \epsilon, \{\!\{\}\!\})} * \lfloor \tau' \rfloor \right\}_{\mathcal{E}}$$

$$\left\{ \boxed{(\epsilon, \epsilon, \{\!\{\mathbf{token}(t_2)\}\!\})} \right\}_{\mathcal{E} \uplus \{\mathrm{IoInv}\}}$$

$$\left\{ \mathbf{token}(t_2) \right\}_{\mathcal{E} \uplus \{\mathrm{IoInv}\}}$$

For the second example, we can reuse the previous example twice:

$$\left\{ \mathbf{token}(t_1) * \mathsf{putchar}(t_1, \text{`l'}, t_2) * \mathsf{putchar}(t_2, \text{`o'}, t_3) \right\}_{\mathcal{E} \uplus \{\mathrm{IoInv}\}}$$

$\mathsf{putchar}(\text{`l'})$

$$\left\{ \mathbf{token}(t_2) * \mathsf{putchar}(t_2, \text{`o'}, t_3) \right\}_{\mathcal{E} \uplus \{\mathrm{IoInv}\}}$$

$\mathsf{putchar}(\text{`o'})$

$$\left\{\ \textbf{token}(t_3)\ \right\}_{\mathcal{E}\,\uplus\,\{\text{IoInv}\}}$$

For the third example we have to perform some extra frame preserving updates to split and join the tokens:

$$\left\{\begin{array}{l}\textbf{token}(t_1) * \textbf{split}(t_1, t_2, t_3) * \mathsf{putchar}(t_2, \text{‘h’}, t_{22}) \\ * \ \mathsf{putchar}(t_3, \text{‘i’}, t_{32}) * \textbf{join}(t_{22}, t_{32}, t_4)\end{array}\right\}_{\mathcal{E}\,\uplus\,\{\text{IoInv}\}}$$

$$\left\{\begin{array}{l}\textbf{token}(t_2) * \textbf{token}(t_3) \\ * \ \mathsf{putchar}(t_2, \text{‘h’}, t_{22}) * \mathsf{putchar}(t_3, \text{‘i’}, t_{32}) \\ * \ \textbf{join}(t_{22}, t_{32}, t_4)\end{array}\right\}_{\mathcal{E}\,\uplus\,\{\text{IoInv}\}} \qquad \text{(frame preserving update)}$$

$$\begin{array}{l||l}
\left\{\begin{array}{l}\textbf{token}(t_2) \\ * \ \mathsf{putchar}(t_2, \text{‘h’}, t_{22})\end{array}\right\}_{\mathcal{E}\,\uplus\,\{\text{IoInv}\}} &
\left\{\begin{array}{l}\textbf{token}(t_3) \\ * \ \mathsf{putchar}(t_3, \text{‘i’}, t_{32})\end{array}\right\}_{\mathcal{E}\,\uplus\,\{\text{IoInv}\}} \\
\mathsf{putchar}(\text{‘h’}) & \mathsf{putchar}(\text{‘i’}) \\
\left\{\ \textbf{token}(t_{22})\ \right\}_{\mathcal{E}\,\uplus\,\{\text{IoInv}\}} &
\left\{\ \textbf{token}(t_{32})\ \right\}_{\mathcal{E}\,\uplus\,\{\text{IoInv}\}}
\end{array}$$

$$\left\{\ \textbf{token}(t_{22}) * \textbf{token}(t_{32}) * \textbf{join}(t_{22}, t_{32}, t_4)\ \right\}_{\mathcal{E}\,\uplus\,\{\text{IoInv}\}}$$

$$\left\{\ \textbf{token}(t_4)\ \right\}_{\mathcal{E}\,\uplus\,\{\text{IoInv}\}} \qquad \text{(frame preserving update)}$$

3.9 Related work

LTL and CTL are well known languages that allow to express temporal properties. Model checking [14, 66, 15] allows checking these properties expressed in these languages for a model. The model can be extracted automatically from the program one wants to verify. Such temporal languages allow to express liveness properties. Unfortunately, model checking suffers from the state explosion problem [15], which would occur in a setting with an unbounded number of threads. Our approach works in a concurrent setting (by simply integrating it with concurrent separation logic), and also works in a setting with an unbounded number of threads.

We use Petri nets as a visualization of heaps, but compositionality and support for expressing that the program should react differently depending on the input is in this thesis not achieved through Petri nets. These features are achieved through the assertion language and by using a set of assertions as the specification of a program. Coloured Petri nets [40] work the other way around by extending Petri nets itself to support compositionality. Note that our approach does not solely consists of a way to express properties one wishes to verify: it also includes a way to verify software.

Nakata and Uustalu [47] define a coinductive big step semantics for programs that do not necessarily terminate. A command is associated to a potentially infinite trace of states, where a state is a mapping from variable names to integer values. They do not focus on I/O, but we expect it should be feasible to extend their approach for I/O verification. We use their coinductive big step semantics for the formalization of the I/O approach for programs that do I/O (instead of memory manipulation). Their work uses postconditions that express properties of the trace directly, even for executions that never terminate, while we use preconditions to express the allowed I/O. Their postcondition based approach has the advantage that one can express that actions must have happened, even for nonterminating executions, while we only support verifying that actions have happened for terminating executions (although at function boundaries we do verify that the intended I/O has happened at that boundary). To support a postcondition based approach, they define an assertion language inspired by interval temporal logic, and an accompanying Hoare logic. Their approach allows to prove liveness properties, which we do not support.

Linear Time Calculus (LTC) [6] extends first order logic in order to support modeling dynamic systems. Actions are represented using predicates that get an argument that expresses when in time the action happens. This argument is a natural number. This is somewhat similar to our approach, but we use two "time" arguments (places) for actions, and it is not always known whether one place is before or after another.

While we have examples, we do not have a case study of any considerable size. Beuster, Henrich and Wagner [5] report on verifying I/O properties of an email client with a text based (but not commandline) UI. The approach identifies points in the main loop of the program and only supports I/O to the screen at these points. In bigger programs typically not only the main loop performs I/O: other functions and libraries perform I/O as well. Our approach is not restricted to programs that only perform I/O in the main loop. Furthermore, we support modularity and compositionality (I/O actions can be defined on top of other I/O actions) to make verification of bigger software more practical.

Iris [38] provides a framework for reasoning about concurrent programs that manipulate a shared resource. Performing I/O can be considered manipulating a shared resource. Sec. 3.8 explains a way to implement our approach in the

Iris framework.

3.10 Conclusions and future work

We developed an approach for verifying that programs that perform I/O only perform desired I/O in a desired order. Furthermore, we sketched how one could implement the verification style in the Iris framework.

The approach supports modularity, compositionality, concurrency, reading and input-dependent behaviour, reuse of specifications and reuse of code, so for verifying programs that perform I/O we hope the approach is applicable to more real-world programs than the provided examples. However, without any case study the feasibility is less certain. Such a case study is future work.

Future work regarding implementing the approach in Iris include implementing the approach in Iris' Coq framework.

Acknowledgements

We would like to thank Amin Timany for together fixing the I/O monoid after an error was found.

List of symbols (in-memory)

agree(C , C)	Chunks do not conflict	Def. 40 p. 107
atintro	Command to introduce an atomic space	Def. 53 p. 113
atdel	Command to delete an atomic space	Def. 53 p. 113
c	A command	Def. 33 p. 103, Def. 52 p. 113, Def. 53 p. 113, Def. 79 p. 140
C $\in$ Chunks	Heap chunks	Def. 36 p. 104, Def. 54 p. 113, Def. 58 p. 116, Def. 58 p. 116, Def. 80 p. 141
e	An expression. Note: an expression is also an assertion.	Def. 33 p. 103
emp	The assertion "true"	Def. 48 p. 111
er(c)	Erasure	Fig. 4.16 p. 148
erat(h)	Remove all atomic space chunks	Def. 73 p. 130
eval	The operation evaluation function	Def. 35 p. 104
g	The ghost level "ghost"	Def. 33 p. 103
gc(e , e , e)	A ghost cell (only if e is an value), or a ghost cell assertion	Def. 58 p. 116, Def. 59 p. 117

$\mathbf{gcf}(e, \overline{e})$	A ghost cell family heap chunk (only if e is a value), or a ghost cell family heap chunk assertion.	Def. 58 p. 116, Def. 59 p. 117
$h \in \text{Heaps}$	Heap containing (fractions of) heap chunks	Def. 37 p. 104
$\tilde{h}$	Either a heap, $\perp$ or **inf**	Def. 39 p. 107
$i \in \mathbb{Z}$	An integer value	
$I \in \{\text{inat}, \text{outat}\}$	Inside or outside atomic space	Sec. 4.3.8 p. 115
inat	Inside atomic space	Sec. 4.3.8 p. 115
inf	Used inside a trace or outcome for infinite atomic executions	Def. 39 p. 107, Def. 44 p. 109
$\text{inv}(h, I)$	Multiset of assertions of atomic space chunks	Def. 74 p. 130
none	Constructor used for when a place has no parent place	Def. 32 p. 90, p. 90
O	An outcome	Def. 44 p. 109
$\text{OK}(\tau)$	Outcome of trace and empty heap does not crash	Def. 85 p. 151
outat	Outside atomic space	Sec. 4.3.8 p. 115
P	An assertion	Def. 48 p. 111, Def. 55 p. 114, Def. 59 p. 117, Def. 64 p. 127, Def. 81 p. 141
$\mathbf{pa}(e, e)$	Prophecy assignment command	Def. 79 p. 140
$\mathbf{pc}()$	Prophecy create command	Def. 79 p. 140
place	Constructor for a place of a Petri net	Def. 32 p. 90, p. 90, p. 95, p. 118
$\mathbf{pr}(e, e)$	Prophecy chunk (only if e is a value) or prophecy chunk assertion	Def. 80 p. 141, Def. 81 p. 141
Q	An assertion. See: P.	
r	The ghost level “real”	Def. 33 p. 103
safe	Safe trace	Def. 76 p. 131

v	A value	Def. 33 p. 103, Def. 47 p. 110
$\bot$	The error value. Also used for crashing traces and outcomes.	Def. 33 p. 103, Def. 39 p. 107, Def. 44 p. 109
$\{\,\overline{f : v}\,\}$	A record value	Def. 33 p. 103
$l \in \{\mathrm{g}, \mathrm{r}\}$	Ghost level	Def. 33 p. 103
$v \mapsto v$	Pointsto heap chunk. See also: $e \mapsto e$.	Def. 36 p. 104
$\pi \in \mathbb{Q}^+$	Fraction	
$\{\!\![\,\pi\,]\,C\,\}\!\!\}$	A heap containing a π fraction of chunk C. Comma separated for multiple chunks. See also: Heaps.	
$\{\!\!\}\!\!\}$	The empty heap	
$c \Downarrow \tau , v$	Program execution	Fig. 4.3 p. 106, Fig. 4.7 p. 114, Fig. 4.9 p. 116, Fig. 4.13 p. 140
τ	A trace	Def. 39 p. 107
$h \uplus h$	Sum of heaps	Def. 41 p. 108
$\tau \,;\, \tau$	Sequential composition of traces	Def. 42 p. 108
$\tau_1 \in \tau_2 \mid\mid \tau_3$	τ_1 is a parallel composition of τ_2 and τ_3. "$\in$" is part of the notation.	Fig. 4.2 p. 105
$O + 1$	Outcome increase	Def. 45 p. 109
$h , \tau \to O$	Outcome of a trace	Def. 46 p. 109
$P * P$	Separating conjunction assertion	Def. 48 p. 111
$e \mapsto e$	Pointsto assertion. See also: $v \mapsto v$.	Def. 48 p. 111
$P \vee P$	Disjunction assetrion	Def. 48 p. 111
$\exists\, x \,.\, P$	Existential quantification assertion	Def. 48 p. 111
$[\,\pi\,]\, P$	Fractional assertion	Def. 48 p. 111
$e\,(\,\overline{e}\,)$	Predicate application assertion	Def. 48 p. 111
$h \vDash_{F} P$	Heap h is a model for a assertion P	Fig. 4.5 p. 110, Fig. 4.12 p. 132

$\vdash \left\{ P \right\} c \left\{ Q \right\}^{l}_{I}$	Hoare triple is provable according to the proof rules	Def. 50 p. 111, Fig. 4.8 p. 114, Fig. 4.10 p. 117, Fig. 4.14 p. 141
$P_1 \to P_2$	Assertion P_1 implies P_2	Def. 51 p. 111
$\langle\, c \,\rangle$	An atomic block command	Def. 52 p. 113
$\boxed{P}$	Atomic space chunk, or atomic space chunk assertion	Def. 54 p. 113, Def. 55 p. 114
$\Lambda \in \mathbb{N} \times \mathbb{N}$	Step	Def. 65 p. 128
F_Λ	Set of Hoare triples that hold for step Λ	Def. 78 p. 132
$\models \left\{ P \right\} c \left\{ Q \right\}_{I}$	Validity of a Hoare triple	Def. 75 p. 130

Chapter 4

I/O style verification of memory-manipulating programs

Abstract

We present a program verification approach that uses an input/output style of reasoning, to verify programs that do not perform I/O but instead manipulate memory. The approach is sound, modular, compositional (I/O actions can be defined on top of other actions) and supports concurrency. It uses Petri nets, separation logic, atomic spaces, ghost cells, higher order functions and prophecies. We have implemented the approach in the VeriFast verifier.

Publication data

- W. Penninckx and B. Jacobs. I/O style verification of memory-manipulating programs. To be submitted
- W. Penninckx and B. Jacobs. A higher-order-ish logic with prophecies for concurrency verification. To be submitted

4.1 Introduction

Some pieces of programs perform input/output (I/O) and some do not. A calculator performs I/O, a sorting algorithm implementation usually does not. The calculator reads from buttons and writes to a screen. The sorting algorithm reads from memory and writes to memory. In a way, both manipulate a shared resource. For pieces of programs that manipulate memory (and do not perform I/O), it is known how to verify correctness properties. We know how to prove that they do not crash and produce correct results, even in some difficult contexts such as concurrency and aliasing.

One could try to apply approaches for memory manipulation to both pieces of programs that do manipulate memory, and those that actually perform I/O. In this chapter, we work the other way around: we present a way to use a verification approach for input/output for pieces of programs that do not perform input/output, but manipulate memory. In other words, we present how one can reason in an I/O style when no I/O is involved.

This can be useful when an API can both be implemented using I/O and using memory-manipulation. For example, we might want to have one filesystem API, but the implementation can be a network filesystem (using I/O) or a filesystem in RAM.

Another use case is when the specification of a function should state the function “does an action” such as for the Template Method design pattern. The Template Method design pattern [26] consists of having a method (called the template method) in a superclass which calls abstract methods. The abstract methods are implemented in subclasses. The specification of the template method (in the superclass) is "to call methods", i.e. that an action happens, but it is unknown what these methods will be since subclasses can be added in the future. Our approach allows to specify such a template method (both when (some) subclasses do and (some) subclasses do not perform I/O), since the I/O style specifications allow to specify that actions happen.

We build upon the I/O verification approach of Chapter 3. This I/O verification approach provides a Hoare logic where the preconditions and postconditions describe sets of Petri nets. The transitions of the Petri net correspond to actions by the program and by the environment. Nondeterminism by (or underspecification of) the environment is modeled by describing multiple Petri nets and nondeterminism by the program is described by using Petri nets that have multiple executions. The program satisfies its specification if for every Petri net of the precondition, every execution of the program simulates an execution of that Petri net, which yields (if the program terminates) a Petri net that is in the set of the Petri nets of the postcondition.

Throughout this chapter we define a programming language, together with Hoare logic style proof rules and a soundness proof (except erasure). The programming language supports manipulating a memory heap and does not have built-in support for performing I/O. We show how one can use the I/O verification approach with I/O style specifications, in that programming language which does not support I/O itself. Pieces of the program can interact with each other: one function can read from a buffer while another function running concurrently in another thread writes to that buffer. Both functions have an I/O style specification. The caller can itself have an I/O style specification, but it is possible that this is not the case. If this is not the case, the caller can construct the I/O style precondition of the callee, and obtain knowledge of the memory state using the I/O style postcondition of the callee.

The approach presented here supports concurrency, split/join, interleaving of actions, interaction between threads (one thread performs writes and another thread reads what is written and acts differently depending on what it reads), and reuse of code and of specifications.

The programming language we define in this chapter supports concurrency, atomic blocks with atomic spaces, ghost cell families (ghost key-value mappings), higher-order functions, and prophecies (obtaining a logical value that eventually will be known to be equal to a value in the program).

The approach works as follows. Which place a token in a Petri net has reached tells the I/O actions performed so far. Petri nets support interleaving through multiple tokens. An invariant relates the performed I/O actions associated with such tokens to the data stored in memory. Verifying an application thus involves writing such an invariant specifically for the application. Functions that perform I/O must not invalidate this invariant and this is enforced by their specifications by having (a fraction of) an atomic space for this invariant in the tokens that are passed between functions as part of their specifications. For non-composite I/O actions a proof that performing the I/O action does not invalidate the invariant, is shipped in the I/O action description as a ghost function.

We build up the approach over multiple sections. As a warmup, Sec. 4.2 simply encodes the memory state in the places. A token assertion then states the memory state satisfies the information encoded in the place. In Sec. 4.3 we add support for concurrency and split/join. Instead of encoding the memory state in places, places encode the list of I/O actions executed in one path of the Petri net executed so far, either directly as such a list or in a simplified form such as the last character written in the path. This encoding of actions is called the *progress*. Since Petri nets can have parallelism because of multiple tokens, multiple such progresses can exist. Tokens own a fraction of the knowledge that an invariant exists and holds, together with a ½ fraction of a ghost cell containing the progress. The invariant contains the other ½ fractions of the ghost cells of the progresses and constrains the memory state based on the progresses. Sec. 4.4 adds support for reusability by including the application-specific part of the invariant in the places. I/O actions that are not composite ship a proof as a ghost function, which proves that performing this I/O action preserves the invariant. Sec. 4.5 explains how reading works by using prophecies. Sec. 4.6 explains how one can use multiple instances of the same data structure in the memory state by using a ghost cell per instance. The ghost cell tracks the content of the data structure. The invariant contains a ½ fraction of these ghost cells.

Sec. 4.7 defines erasure (removing the ghost code from commands that mix ghost code and real code) and Sec. 4.8 provides a soundness proof (except erasure). We end the chapter with a discussion of related work, future work, and conclusions (Sec. 4.9).

4.2 Warmup without concurrency

We start with a small example that shows how the I/O style verification approach can be applied to a program that just writes a character into a memory buffer. This example uses a minimal number of features; support for more features is studied later.

The program of the example is as follows:

```
$putchar = \lambda \mathsf{b}, \mathsf{c}.$
  $[\mathsf{b}] := \mathsf{c}$

$print\_hi = \lambda \mathsf{b}.$
  $putchar(\mathsf{b}, \text{‘h’});$
  $putchar(\mathsf{b}, \text{‘i’})$

$main = \lambda.$
  $\textbf{let } \mathsf{b} := \textbf{cons}(0) \textbf{ in}$
  $print\_hi(\mathsf{b});$
  $\textbf{let } \mathsf{x} := [\mathsf{b}] \textbf{ in}$
  $\textbf{dispose}(\mathsf{b});$
  $\mathsf{x}$
```

The example does not use the programming language defined earlier. We do not use BIO commands because the programs do not perform actual I/O, but we use memory allocation (**cons**$(\overline{e})$ allocates memory cells with consecutive addresses), deallocation (**dispose**(e) frees one memory cell), writing to memory cells ($[e] := e$) and reading from memory cells ($[e]$).

The functions (*putchar*, *print_hi*, *main*) are functions in the programming language itself (not in the metalogic). The *putchar* function performs a write. We implement it as just writing to a memory buffer. This simply overwrites the old content. *print_hi* prints 'h' followed by 'i'. It is implemented on top of the *putchar* function. *main* allocates a buffer, calls *print_hi* (which writes twice to that buffer) and deallocates the buffer. It returns the content the buffer had before deallocation.

In the programming language, the heap contains memory cells. Assertions can express the existence of a memory cell, e.g. the assertion $4 \mapsto 5$ expresses that on address 4, there is value 5.

An assertion can be a *predicate value application*, like $\mathit{token}(t_1)$. *token* and *putchar_io* are *predicate value*s. A predicate value is a lambda function in the assertion language itself that can be applied to arguments. Predicate values are somewhat similar to copredicates (Sec. 3.6.3), but cannot express an infinite number of permissions. To more visually distinguish predicate value definitions from function definitions, we write **predicate** before a predicate value definition.

This simple programming and assertion language suffices to verify the example in an I/O style, as follows.

predicate $\mathit{token} = \lambda \mathsf{b}, \mathsf{t}_1.\, \mathsf{b} \mapsto \mathsf{t}_1$

predicate $\mathit{putchar\_io} = \lambda \mathsf{t}_1, \mathsf{c}, \mathsf{t}_2.\, (\mathsf{t}_2 = \mathsf{c})$

$\mathit{putchar} = \lambda \mathsf{b}, \mathsf{c}.$

$\{\ \mathit{token}(\mathsf{b}, t_1) * \mathit{putchar\_io}(t_1, \mathsf{c}, t_2)\ \}$

$\{\ \mathsf{b} \mapsto t_1 * t_2 = \mathsf{c}\ \}$

$[\mathsf{b}] := \mathsf{c}$

$\{\ \mathsf{b} \mapsto t_2 * t_2 = \mathsf{c}\ \}$

$\{\ \mathit{token}(\mathsf{b}, t_2)\ \}$

predicate $\mathit{print\_hi\_io} = \lambda \mathsf{t}_1, \mathsf{t}_2.$
$\exists \mathsf{t}_\mathsf{h}. \mathit{putchar\_io}(\mathsf{t}_1, \text{'h'}, \mathsf{t}_\mathsf{h}) * \mathit{putchar\_io}(\mathsf{t}_\mathsf{h}, \text{'i'}, \mathsf{t}_2)$

$print\_hi = \lambda \mathsf{b}.$
$\{\ token(\mathsf{b}, t_1) * print\_hi\_io(t_1, t_2)\ \}$
  $putchar(\mathsf{b}, \text{‘h’});$
  $putchar(\mathsf{b}, \text{‘i’})$
$\{\ token(\mathsf{b}, t_2)\ \}$

$main = \lambda.$
$\{\ \mathbf{emp}\ \}$
  $\mathbf{let}\ \mathsf{b} := \mathbf{cons}(0)\ \mathbf{in}$
$\{\ \mathsf{b} \mapsto 0\ \}$
$\{\ token(\mathsf{b}, 0)\ \}$
$\{\ token(\mathsf{b}, 0) * putchar\_io(0, \text{‘h’}, \text{‘h’}) * putchar\_io(\text{‘h’}, \text{‘i’}, \text{‘i’})\ \}$
$\{\ token(\mathsf{b}, 0) * print\_hi\_io(0, \text{‘i’})\ \}$
  $print\_hi(\mathsf{b});$
$\{\ token(\mathsf{b}, \text{‘i’})\ \}$
$\{\ \mathsf{b} \mapsto \text{‘i’}\ \}$
  $\mathbf{let}\ \mathsf{x} := [\mathsf{b}]\ \mathbf{in}$
$\{\ \mathsf{b} \mapsto \text{‘i’} * \mathsf{x} = \text{‘i’}\ \}$
  $\mathbf{dispose}(\mathsf{b});$
$\{\ \mathsf{x} = \text{‘i’}\ \}$
  x
$\{\ \mathsf{res} = \text{‘i’}\ \}$

So, how does the example work? When considering input/output as a style of reasoning, we would consider input/output actions as observations/modifications of some shared resource, such as the network or a filesystem. In this example, the shared resource is not yet actually shared. It is just a buffer of one integer. An assertion $token(b, t)$ states that the information about the shared resource encoded in the place t is currently true for the buffer at address b. The encoding of information about the shared resource in a place is in this example done by having the place equal to the value of the shared resource. As an example, the place 7 expresses that the shared resource is equal to the value 7, and the assertion $token(b, 7)$ expresses that currently this is the case for the buffer b.

Consider an instance of an I/O action, namely $putchar\_io(4, 5, 5)$. It expresses the action of going from a buffer with value 4 (the first argument), to a buffer with value 5 (the last argument) by writing 5 (the second argument) to the buffer.

Observe the flow of the example program: *main* starts with a non-I/O style precondition. It calls *print_hi*. *print_hi* has an I/O style precondition and reaches its I/O style postcondition. *main* obtains the knowledge encoded in that I/O style postcondition, and terminates by reaching its non-I/O style postcondition, ensuring properties about the memory state in a non-I/O style.

print_hi only calls functions with an I/O style specification. For verifying *print_hi* it is fine to ignore whether the I/O actions will be actual I/O or memory manipulation: for both cases the verification is the same. This also means it is fine to later switch between I/O and memory-manipulation: verification of *print_hi* will not have to be done again. In other words, *print_hi* lives in an "I/O style" world and does not care whether I/O style means real I/O or memory manipulation.

putchar (called by *print_hi*) has an I/O style specification, but is implemented by manipulating memory.

4.3 Concurrency

I/O style verification gets more interesting when concurrency is involved. Consider two threads (or programs or functions) running concurrently, each with an I/O style specification. They both work on the same shared resource – say a shared memory buffer. Can we verify the end result (say the state of the buffer) after both threads have ended? It gets more interesting if one thread can read the memory that another thread writes to, but we postpone reading until Sec. 4.5 to build up the explanation more gradually.

4.3.1 I/O threads

The following example uses concurrency: the command $c_1 \,||\, c_2$ concurrently executes the command c_1 and the command c_2, and continues after both c_1 and c_2 are terminated. So in the following example, the function *print_hi* concurrently calls *print_h*(b) and *print_i*(b) (we postpone a concurrency-enabled definition of *putchar*):

```
print_h = λb.
  putchar(b, 'h')
```

```
$print\_i = \lambda$b.
  $putchar($b, ‘i’$)$

$print\_hi = \lambda$b.
  (
    $print\_h($b$)$
  ||
    $print\_i($b$)$
  )
```

Upon executing *print_hi*, ‘h’ and ‘i’ will be written concurrently to the buffer, so after executing *print_hi* the buffer will either contain ‘h’ or ‘i’.

Remember that split and join allow us to not constrain the ordering of a set of I/O actions relative to another set. So we would like to verify the example in a style as shown below (ignore the argument r for now)

```
$print\_hi = \lambda \mathsf{b} ; \mathsf{r}$ .
  $\left\{\begin{array}{l} token(\mathsf{b}, t_1) * split\_io(t_1, t_{h1}, t_{i1}) \\ * putchar\_io(t_{h1}, \text{‘h’}, t_{h2}, \mathsf{r}.\text{“h”}) \\ * putchar\_io(t_{i1}, \text{‘i’}, t_{i2}, \mathsf{r}.\text{“i”}) \\ * join\_io(t_{h2}, t_{i2}, t_2) \end{array}\right\}$
  let _ := $split()$ in ❶
  $\{ \ldots * token(\mathsf{b}, t_{i1}) * token(\mathsf{b}, t_{h1}) \}$
  (
    $\left\{\begin{array}{l} token(\mathsf{b}, t_{h1}) \\ * putchar\_io(t_{h1}, \text{‘h’}, t_{h2}, \mathsf{r}.\text{“h”}) \end{array}\right\}$ $print\_h(\mathsf{b} ; \mathsf{r}.\text{“h”})$ $\{ token(\mathsf{b}, t_{h2}) \}$
    ||
    $\left\{\begin{array}{l} token(\mathsf{b}, t_{i1}) \\ * putchar\_io(t_{i1}, \text{‘i’}, t_{i2}, \mathsf{r}.\text{“i”}) \end{array}\right\}$ $print\_i(\mathsf{b} ; \mathsf{r}.\text{“i”})$ $\{ token(\mathsf{b}, t_{i2}) \}$
  );
  $\{ token(\mathsf{b}, t_{h2}) * token(\mathsf{b}, t_{i2}) * join\_io(t_{h2}, t_{i2}, t_2) \}$ ❷
  let _ := $join()$ in unit ❸
  $\{ token(\mathsf{b}, t_2) \}$
```

This example starts from one token. This token is split in two (❶). Two parallel threads are started: each will use one token. One thread writes ‘h’ and the

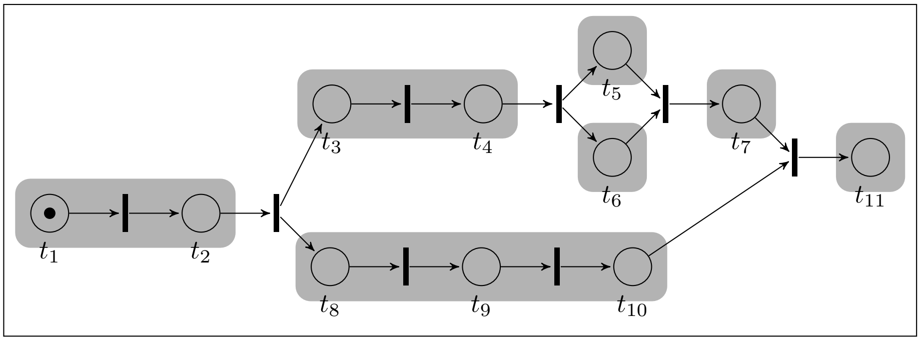


Figure 4.1: Every gray box is one I/O thread of this Petri net

other writes 'i' (concurrently). After both threads terminate, we have obtained two tokens: each from one thread (❷). The resulting tokens are joined (❸).

It is clear that our previous definitions of *token* and places for in-memory I/O will not suffice to do this. We also have to define *split_io*, *join_io*, *split*, and *join*, but we start with *token*.

To perform verification in such a style, we let the token "returned" by the second thread ($token(\mathsf{b}, t_{i2})$) express that the thread has printed 'i', without necessarily constraining what another thread has or has not done. The token obtained by joining the two tokens from the threads ($token(\mathsf{b}, t_2)$) will then express that the first thread has printed 'h' and the second 'i'.

In order to do that, we construct a new definition for places (Sec. 32 on the following page), and for that, we first introduce the concept of I/O threads.

Given a Petri net, we can partition the places such that in any partition there is no split or join between two places of that partition, while every partition is as large as possible. In other words, the boundaries of the partition are splits and joins. See Fig. 4.1 for an example. Every partition is what we call an I/O thread.

Given a Petri net and one I/O thread assigned as the "initial" one, one can identify every I/O thread by saying how it is related to the initial one, e.g. in Fig. 4.1 the I/O thread of t_3 is the left child[1] of the I/O thread of t_1. We will use this in our annotations, so we define a grammar for it:

$iot ::=$ init | left(iot) | right(iot) | join(iot, iot)

[1]one can arbitrary assign whether a child is the left or right child of a parent, as long as a parent does not have multiple left or multiple right children.

$v ::= \ldots \mid iot$

We will write iot(t) to express the I/O thread of place t. So in our example of Fig. 4.1 on the previous page: $\text{iot}(t_7) = \text{join}(\text{left}(\text{left}(\text{init})), \text{right}(\text{left}(\text{init})))$.

An I/O thread can correspond to a regular thread that is executed concurrently, but it does not have to. Also keep in mind that with an I/O style specification multiple Petri nets can be associated and these Petri nets can look entirely different from each other. Hence, with a specification multiple sets of I/O threads can be associated.

4.3.2 Places

Definition 32: Places

$place ::= \text{none} \mid \text{place}(iot, v, place, place, \lambda id.\, P, v_{id})$

Let us build up the above definition gradually[2]. We let a place state which I/O thread it belongs to (the dots (...) represent things we will add and explain later):

$place ::= \ldots \mid \text{place}(iot, \ldots)$

A value can be a place:

$v ::= \ldots \mid place$

We also let a place contain what the *progress* is of that place. The progress represents the actions that the I/O thread of the place has performed.

$place ::= \ldots \mid \text{place}(iot, v, \ldots)$

The progress can be encoded in various ways, but we will stick to simply a list of characters in this example. So in our *print_hi* example, the progress of t_1, of t_{h1} and of t_{i1} is the empty list, the progress of t_{h2} is $\langle$'h'$\rangle$, the progress of t_{i2} is $\langle$'i'$\rangle$ and the progress of t_2 is again the empty list.

A place also remembers the "parent" place(s); that is the last place of the previous I/O thread(s):

[2]To not explain everything at once, we spread the explanation over multiple pages: p. 90, p. 95, p. 118.

$\mathit{place} ::= \text{none} \mid \text{place}(\mathit{iot}, v, \mathit{place}, \mathit{place}, \ldots)$

This allows us to know the progress of terminated I/O threads. For example, as soon as we have obtained $\mathit{token}(t_3)$ (in Fig. 4.1 on page 89) we can now still know what the progress of $\text{iot}(t_1)$ is.

In case the place exists because of a join, it has two parent places. Note that in the example of Fig. 4.1 on page 89, t_3 and t_4 have the same parent place, namely t_2.

This also works with nested splits and joins, and joins after joins. In the example of Fig. 4.1 on page 89, t_{11} will still encode the progress that has been made for t_1.

For the *print_hi* example, the places are as follows:

$$
\begin{aligned}
&t_1 = \text{place}(\text{init}, \langle\rangle, \text{none}, \text{none}, \ldots)\\
&t_{h1} = \text{place}(\text{left}(\text{init}), \langle\rangle, t_1, \text{none}, \ldots)\\
&t_{h2} = \text{place}(\text{left}(\text{init}), \langle\text{'h'}\rangle, t_1, \text{none}, \ldots)\\
&t_{i1} = \text{place}(\text{right}(\text{init}), \langle\rangle, t_1, \text{none}, \ldots)\\
&t_{i2} = \text{place}(\text{right}(\text{init}), \langle\text{'i'}\rangle, t_1, \text{none}, \ldots)\\
&t_2 = \text{place}(\text{join}(\text{left}(\text{init}), \text{right}(\text{init})), \langle\rangle, t_{h2}, t_{i2}, \ldots)
\end{aligned}
$$

We use the notation $t.\mathit{fieldname}$ to retrieve a field of the place t, for example $t.\text{parent1}$ returns the first parent of t (i.e. the third field).

4.3.3 Tokens, ghost cell families, fractions and atomic spaces

Places are just pieces of information; we will use tokens to link this information to the memory state. To do that, we introduce the concept of *ghost cell families*. The high-level idea is that the token will contain a $1/2$ fraction of a ghost cell that contains the progress of the I/O thread, and an invariant will contain the other $1/2$ fraction. The invariant can then link this knowledge with the memory state. We will fill in the details to make this work and explain the new concepts.

Fractions A fractional permission [7, 8] is an assertion, such as $[1/2]30 \mapsto 40$. This assertion expresses that at address 30 there is value 40, and that the thread under consideration is not necessarily the only thread that knows this. Therefore, the thread can read from address 30 and this will return value 40, but we do not want the thread to write to it because that could violate another thread's knowledge that at address 30 is value 40. However, after the other thread terminates, the thread that is still alive can write to it, if there is no third thread that owns a fraction. To track whether another thread owns a fraction, we use the fraction size, which is $1/2$ in the example of the assertion $[1/2]30 \mapsto 40$. If the fraction size is one, then there is only one thread that has a fraction (namely of size one) and it can write to it. If it is less than one, there might be another thread that has a fraction. Let's illustrate fractions with an example:

```
$\{\ \textbf{emp}\ \}$
$\textbf{let}\ \mathsf{x} := \textbf{cons}(7)\ \textbf{in}$
$\{\ [1]\mathsf{x} \mapsto 7\ \}$ ❶
$\{\ [1/2]\mathsf{x} \mapsto 7 * [1/2]\mathsf{x} \mapsto 7\ \}$ ❷
(
    $\{\ [1/2]\mathsf{x} \mapsto 7\ \}$ [x] $\{\ [1/2]\mathsf{x} \mapsto \_\ \}$ ❸
    ||
    $\{\ [1/2]\mathsf{x} \mapsto 7\ \}$ [x] $\{\ [1/2]\mathsf{x} \mapsto 7\ \}$ ❸
);
$\{\ [1/2]\mathsf{x} \mapsto \_ * [1/2]\mathsf{x} \mapsto 7\ \}$
$\{\ [1]\mathsf{x} \mapsto 7\ \}$ ❹
[x] := 9 ❺
$\{\ [1]\mathsf{x} \mapsto 9\ \}$
```

In this example, after allocating the memory cell we have a fraction of size one of the memory cell (❶). We split this fraction into two fractions (❷). A $1/2$ fraction is given to one thread, and the other $1/2$ fraction is given to the other thread, and both threads return a $1/2$ fraction (❸) but one thread specifies the cell contains 7 ($[1/2]\mathsf{x} \mapsto 7$) and the other thread does not specify the contents of the cell ($[1/2]\mathsf{x} \mapsto \_$). We can merge these fractions (❹). Since the memory cell cannot contain both the value 7 and a different value at the same time, we know its content must be 7. Since we now again have a fraction of size one, we can write to it (❺).

In case the fraction size is one, we usually do not write the fraction size. For example the assertion $[1]\mathsf{x} \mapsto 7$ can be written as $\mathsf{x} \mapsto 7$.

Ghost cell families Ghost cells are like regular heap cells, except they only serve verification. They also are not identified by an address, but by a pair of an ID and a name/key. To create such an ID, one can use the *ghost command* **create_gcf**:

$\{\ \textbf{emp}\ \}$

$\textbf{let}\ \mathsf{id} := \textbf{create\_gcf}()\ \textbf{in}$

$\{\ \textbf{gcf}(\mathsf{id}, \langle\rangle)\ \}$

$\textbf{gcf}(id, \langle\rangle)$ expresses that the ghost cell family with the ID *id* exists, and that the empty list of keys is in use. Keeping track of the list of used keys allows one to prevent creating two cells with the same key and ID.

Once the ID is created, one can create the cells:

$\{\ \textbf{gcf}(\mathsf{id}, \langle\rangle)\ \}$

$\textbf{gcf\_cons}(\mathsf{id}, \text{“somekey”}, \text{“somevalue”});$

$\{\ \textbf{gcf}(\mathsf{id}, \langle\text{“somekey”}\rangle) * \textbf{gc}(\mathsf{id}, \text{“somekey”}, \text{“somevalue”})\ \}$

$\textbf{gcf\_cons}(\mathsf{id}, \text{“answer”}, 42)$

$$\left\{\begin{array}{l} \textbf{gcf}(\mathsf{id}, \langle\text{“somekey”}, \text{“answer”}\rangle) \\ * \textbf{gc}(\mathsf{id}, \text{“somekey”}, \text{“somevalue”}) * \textbf{gc}(\mathsf{id}, \text{“answer”}, 42) \end{array}\right\}$$

A heap can contain ghost cells, i.e. $\textbf{gc}(v_{id}, v_{key}, v_{val})$ is the heap chunk that is the ghost cell with ID v_{id}, key v_{key} and value v_{val}.

When you have a full fraction (i.e. fraction size 1) of such a cell, you can write to it:

$\{\ \textbf{gc}(\mathsf{id}, \text{“somekey”}, \text{“somevalue”})\ \}$

$(\mathsf{id}, \text{“somekey”}) := 12$

$\{\ \textbf{gc}(\mathsf{id}, \text{“somekey”}, 12)\ \}$

Ghost code **create_gcf**() and **gcf_cons**(...) are pieces of ghost code: we treat it like regular code during verification, but we can remove it to execute the program. One could say ghost code is not actually part of the program; it is just there in order to verify the program.

So there are two kinds of code: real code, and ghost code. It is important that real code does not use return values from ghost code since the ghost code should be removable prior to executing the program. In order to deal with this, we mark variables as ghost variables or real variables by using superscript r or g: x^{r} is a real variable and y^{g} is a ghost variable. When writing a variable name without marking it as real or ghost, such as x, it is still either real or ghost, we just did not specify whether it was real or ghost.

So in the following example id is a ghost variable:

$\textbf{let } \mathsf{id}^{\mathrm{g}} := \textbf{create\_gcf}() \textbf{ in } c$

Besides marking variables as real or ghost, we also mark code as real or ghost. This works as follows: ghost code cannot contain real code, but real code can contain ghost code. The whole program itself is always considered real code. So the only way to have ghost code, is to embed it into real code. The only way to embed ghost code is by using the let syntax with a ghost variable. For example $\textbf{let } \mathsf{v}^{\mathrm{g}} := c_1 \textbf{ in } c_2$ embeds c_1 as ghost code.

The rules mentioned above (ghost code cannot contain real code, and real code cannot mention ghost variables) are not enforced by the syntax of the programming language, and must be checked separately. Sec. 4.7 explains how such rules are checked and how to safely remove the ghost code.

Atomic spaces We use assertions of the form $[\pi]\boxed{P}$, where π is a fraction size and P is another assertion. Such an assertion $[\pi]\boxed{P}$ expresses that P holds and is called an *atomic space*. Atomic spaces are similar to shared regions [19] and Iris' invariants [38]. An atomic space assertion expresses that on the one hand you can rely on the fact that P holds, and on the other hand you have to make sure it keeps holding. This is useful in a multithreaded environment: multiple threads can have a fraction of $\boxed{P}$, so each of these threads knows P holds. A thread should not make modifications to the heap chunks that are covered by P that would stop P from holding, because other threads rely on their assumption that P keeps holding.

The ghost command **atintro** creates an atomic space chunk $\boxed{P}$. Executing this command revokes permission (from a verification point of view) to access the part of the heap covered by P. That part of the heap is only accessible in an atomic command $\langle c\rangle$. When fractions of $\boxed{P}$ are given to different threads, the assertion P is then the same for each thread: one can not convert $[\pi]\boxed{P}$ into $[\pi]\boxed{P'}$ when P is not the same as P', even if for the current heap both P and P' hold.

The ghost command **atdel** removes an atomic space chunk. This (from a verification point of view) gives back access to the part of the heap covered by P.

Token A property of fractions of (ghost) cells is that given $[\pi_1]\mathbf{gcf}(id, k, v_1)$ and $[\pi_2]\mathbf{gcf}(id, k, v_2)$, we know $v_1 = v_2$. We use this as follows. A token has a $1/2$ fraction of a ghost cell with as key the I/O thread and as value the progress. An invariant contains the other $1/2$ fraction.

We also let a token contain its parent token(s) such that given a token, we also know the progresses of the parents and their parents and so on. For an I/O thread that is the result of a join, the token contains the tokens of both parents. For an I/O thread that is the result of a split, a $1/2$ fraction of the parent token is contained and the sibling contains the other $1/2$ fraction.

We define *token* as follows:

```
$\textbf{predicate}\ \mathit{token\_rec} = \textbf{rec}\ \mathsf{token\_rec}(\mathsf{t}).$
  $[1/2]\mathbf{gc}(\mathsf{t}.\mathrm{id}, \mathsf{t}.\mathrm{iot}, \mathsf{t}.\mathrm{progress})$
  $*\ \textbf{if}\ \mathrm{is\_left}(\mathsf{t}.\mathrm{iot}) \lor \mathrm{is\_right}(\mathsf{t}.\mathrm{iot})\ \textbf{then}\ [1/2]\mathit{token\_rec}(\mathsf{t}.\mathrm{parent1})$
    $\textbf{else if}\ \mathrm{is\_join}(\mathsf{t}.\mathrm{iot})\ \textbf{then}\ \mathit{token\_rec}(\mathsf{t}.\mathrm{parent1})$
                                        $*\ \mathit{token\_rec}(\mathsf{t}.\mathrm{parent2})$
         $\textbf{else emp}$

$\textbf{predicate}\ \mathit{token} = \lambda \mathsf{b}, \mathsf{t}.$
  $[\mathsf{t}.\mathrm{iot}.\mathrm{f}]\boxed{\mathit{buffer\_invar}(\mathsf{t}.\mathrm{id}, \mathsf{b})}$
  $*\ \mathit{token\_rec}(\mathsf{t})$
```

Note that *token_rec* is a recursive predicate. We use the syntactic sugar $\textbf{rec}\ p(\overline{x}).\, P$ for $\lambda \overline{x}.\, r(r, \overline{x})$ where $r = \lambda \mathsf{r}, \overline{x}.\, P[\mathsf{r}(\mathsf{r}, \overline{z})/p(\overline{z})]$.

To make this definition of *token* work, a place also contains the ID (of the ghost cell family). This ID is the last argument of the place constructor:

$\mathit{place} ::= \mathrm{none} \mid \mathrm{place}(\mathit{iot}, v, \mathit{place}, \mathit{place}, v_{id})$

token contains a certain fraction size of $\boxed{\mathit{buffer\_invar}(\ldots)}$. This fraction size is defined in such a way that $\boxed{\mathit{buffer\_invar}(\ldots)}$ is shared over all tokens. For an initial token, the fraction size is 1. If that token is split in two, then both have fraction size $1/2$. If they are joined, the token that is the result of the join has fraction size 1 of the buffer again. The notation $iot.\mathrm{f}$ calculates such a fraction size and is defined inductively as follows:

$$\begin{array}{rcl}
\mathrm{init.f} & = & 1 \\
\mathrm{left}(iot).\mathrm{f} & = & iot.\mathrm{f}/2 \\
\mathrm{right}(iot).\mathrm{f} & = & iot.\mathrm{f}/2 \\
\mathrm{join}(iot_1, iot_2).\mathrm{f} & = & iot_1.\mathrm{f} + iot_2.\mathrm{f}
\end{array}$$

The definition of *token* uses is_left, is_right, and is_join. is_left is defined as follows: is_left(iot) is true iff there is some iot' such that $iot = \mathrm{left}(iot')$. is_right and is_join are defined analogously.

Here is an example invariant for the *print_hi* example code above (p. 88) that prints 'h' in one thread and 'i' in another thread:

```
**predicate** $buffer\_invar = \lambda \mathsf{id}, \mathsf{b}.$
$\exists \mathsf{l}, \mathsf{r}, \mathsf{ltodo}, \mathsf{rtodo}, \mathsf{c}.$
  $\mathsf{b} \mapsto \mathsf{c}$
  $* [1/2]\mathbf{gc}(id, \mathrm{left}(\mathrm{init}), \mathsf{l}) * [1/2]\mathbf{gc}(id, \mathrm{right}(\mathrm{init}), \mathsf{r})$
  $* \mathsf{l} +\!\!+ \mathsf{ltodo} = \langle\text{'h'}\rangle * \mathsf{r} +\!\!+ \mathsf{rtodo} = \langle\text{'i'}\rangle$
  $*$ **if** $\mathsf{l} \neq \langle\rangle \wedge \mathsf{r} \neq \langle\rangle$ **then** $\mathsf{c} = \mathrm{last}(\mathsf{l}) \vee \mathsf{c} = \mathrm{last}(\mathsf{r})$
    **else if** $\mathsf{r} \neq \langle\rangle$      **then** $\mathsf{c} = \mathrm{last}(\mathsf{r})$
    **else if** $\mathsf{l} \neq \langle\rangle$      **then** $\mathsf{c} = \mathrm{last}(\mathsf{l})$
    **else emp**
```

Note how the invariant has knowledge about the progresses, but also knows the memory representation of the shared resource ($\mathsf{b} \mapsto \mathsf{c}$). It links the knowledge about the progresses to the state of the shared resource.

buffer_invar does not contain ghost cells for I/O threads whose progress is intended to always be $\langle\rangle$, such as init and join(left(init), right(init)). While it is possible to include e.g. $[1/2]\mathbf{gc}(id, \mathrm{init}, \langle\rangle)$ in *buffer_invar*, this is not necessary. Without, *buffer_invar* is still strong enough to enforce that the implementation does not do undesired writes.

4.3.4 Split and join

```
**predicate** $split\_io = \lambda \mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3.$
  $\mathsf{t}_2 = \mathrm{place}(\mathrm{left}(\mathsf{t}_1.\mathrm{iot}), \langle\rangle, \mathsf{t}_1, \mathrm{none}, \mathsf{t}_1.\mathrm{id})$
  $* [1/2]\mathbf{gc}(\mathsf{t}_2.\mathrm{id}, \mathsf{t}_2.\mathrm{iot}, \langle\rangle)$
  $* \mathsf{t}_3 = \text{place}(\text{right}(\mathsf{t}_1.\text{iot}), \langle\rangle, \mathsf{t}_1, \text{none}, \mathsf{t}_1.\text{id})$
  $* [1/2]\mathbf{gc}(\mathsf{t}_3.\text{id}, \mathsf{t}_3.\text{iot}, \langle\rangle);$
```

```
$split = \lambda.$
  $\{\ token(b, t_1) * split\_io(t_1, t_2, t_3)\ \}$
  skip
  $\{\ token(b, t_2) * token(b, t_3)\ \}$
```

$split\_io(t_1, t_2, t_3)$ assigns a fixed value to t_2 and t_3 that only depends on t_1. $split\_io(t_1, t_2, t_3)$ also includes a $1/2$ fraction of the ghost cell that tracks the progress of t_2, and a $1/2$ fraction of the ghost cell for t_3. The *split* function takes out these $1/2$ fractions of the ghost cells and puts them in $token(b, t_2)$ and $token(b, t_3)$. Remember that the other $1/2$ fraction of the ghost cells are in the invariant. Also notice that the ghost cells say the progresses for t_2 and for t_3 are the empty list.

Join is similar:

```
predicate $join\_io = \lambda \mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3.$
  $\mathsf{t}_3 = \text{place}(\text{join}(\mathsf{t}_1.\text{iot}, \mathsf{t}_2.\text{iot}), \langle\rangle, \mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_1.\text{id})$
  $* \mathsf{t}_1.\text{id} = \mathsf{t}_2.\text{id}$
  $* [1/2]\mathbf{gc}(\mathsf{t}_3.\text{id}, \mathsf{t}_3.\text{iot}, \langle\rangle)$
```

```
$join = \lambda.$
  $\{\ join\_io(t_1, t_2, t_3) * token(b, t_1) * token(b, t_2)\ \}$
  skip
  $\{\ token(b, t_3)\ \}$
```

4.3.5 I/O actions

Now, in order to have a function that given a token in a precondition returns a token in the postcondition, this function will either have to call functions that manipulate tokens, or have to do it itself. Here is an example of the latter. We look at the implementation first (you can ignore the parameter r for now[3]):

$putchar = \lambda \mathsf{b}, \mathsf{c}\ ; \mathsf{r}\ .$

[3]At this point you do not need to understand the parameter r, but we include it in the examples to have consistent examples. To not explain everything at once, we postpone explaining the parameter r until p. 100 (❷).

$\langle [\mathsf{b}] := \mathsf{c} \rangle$

Execution of a command $\langle c \rangle$ is atomic: executing $\langle c \rangle$ executes the command c without any interleaving of other threads. In real-world applications one would use a less course-grained synchronization mechanism such as mutexes, but for studying the I/O approach this suffices. The nice property of an atomic block such as $\langle [\mathsf{b}] := \mathsf{c} \rangle$ is that, while performing verification, we can allow it to write to data protected by atomic spaces, and even violate the assertion of the atomic space, as long as the invariant holds again when the atomic block ends. For example, in case of an atomic space $[\pi]\boxed{P}$, the atomic block can access the memory covered by P, can modify it such that P does not hold anymore, but must make sure P holds again when the atomic block ends. We will use this in this subsection.

For the specification we would like something like this:

```
$putchar = \lambda \mathsf{b}, \mathsf{c} \,;\, \mathsf{r}\,.$
    $\{\ token(\mathsf{b}, t_1) * putchar\_io(t_1, \mathsf{c}, t_2, \mathsf{r})\ \}$
    $\langle [\mathsf{b}] := \mathsf{c} \rangle$
    $\{\ token(\mathsf{b}, t_2)\ \}$
```

For this specification we need to define *putchar_io*. Here's what we ideally would like to have for *putchar_io* (for now, ignore the line $\mathsf{r}.\text{“t1”} = \mathsf{t}_1$):

```
$\textbf{predicate}\ putchar\_io = \lambda \mathsf{t}_1, \mathsf{c}, \mathsf{t}_2, \mathsf{r}.$
    $\mathsf{r}.\text{“t1”} = \mathsf{t}_1$
    $* \mathsf{t}_2 = \mathsf{t}_1[\text{progress} := \mathsf{t}_1.\text{progress} +\!+ \langle \mathsf{c} \rangle]$
```

It says that t_2 has the same parents and I/O thread as t_1, but its progress is longer. The longer progress includes that c now also has been printed.

Sadly, we are running ahead now with this powerful definition and will stick to a more restricted definition before we learn how to deal with the more powerful version in Sec. 4.4.

The more restricted version is as follows:

```
$\textbf{predicate}\ putchar\_io = \lambda \mathsf{t}_1, \mathsf{c}, \mathsf{t}_2, \mathsf{r}.$
    $\mathsf{r}.\text{“t1”} = \mathsf{t}_1$ ❶
    $* \mathsf{t}_2 = \mathsf{t}_1[\text{progress} := \langle \mathsf{c} \rangle]$
```

$$\begin{aligned}
&* \mathsf{t}_1.\text{progress} = \langle\rangle \\
&* \big(\mathsf{t}_1.\text{iot} = \text{left}(\text{init}) \wedge \mathsf{c} = \text{‘}\mathsf{h}\text{’} \\
&\quad \vee \mathsf{t}_1.\text{iot} = \text{right}(\text{init}) \wedge \mathsf{c} = \text{‘}\mathsf{i}\text{’}\big)
\end{aligned}$$

It is basically the same, except it is tailored to the invariant. This makes it application-specific because the invariant is application-specific (again, we lift this restriction in Sec. 4.4).

Now that we know *putchar*'s specification and the definitions used in its specification, we verify it. Here's a proof outline (a question mark before a variable variable denotes this is the first occurrence of the variable):

$$\begin{aligned}
&\mathit{putchar} = \lambda \mathsf{b}, \mathsf{c}\ ; \mathsf{r}\ .\ ❷ \\
&\quad \{\ \mathit{token}(\mathsf{b}, ?t_1) * \mathit{putchar\_io}(t_1, \mathsf{c}, ?t_2, \mathsf{r})\ \} \\
&\quad \left\{ \begin{array}{l} [t_1.\text{iot.f}]\boxed{\mathit{buffer\_invar}(t_1.\text{id}, \mathsf{b})} * \mathit{token\_rec}(t_1) \\ * \mathit{putchar\_io}(t_1, \mathsf{c}, t_2, \mathsf{r}) \end{array} \right\} \\
&\quad \langle \\
&\qquad \left\{ \begin{array}{l} \mathit{buffer\_invar}(t_1.\text{id}, \mathsf{b}) * \mathit{token\_rec}(t_1) \\ * \mathit{putchar\_io}(t_1, \mathsf{c}, t_2, \mathsf{r}) \end{array} \right\} \\
&\qquad //\ \text{Case } t_1.\text{iot} = \text{left}(\text{init}) \text{ (the other case is analogous)} \\
&\qquad \left\{ \begin{array}{l} \mathsf{b} \mapsto \_ \\ * \mathbf{gc}(t_1.\text{id}, \text{left}(\text{init}), \langle\rangle) *\ [1/2]\mathbf{gc}(t_1.\text{id}, \text{right}(\text{init}), ?\mathit{rg}) \\ * t_2 = t_1[\text{progress} := \langle \mathsf{c} \rangle] * \mathsf{c} = \text{‘}\mathsf{h}\text{’} * t_1.\text{iot} = \text{left}(\text{init}) * \mathsf{r}.\text{“}\mathsf{t1}\text{”} = t_1 \\ * \mathit{rg} \mathbin{++} ?\mathit{rtodo} = \langle \text{‘}\mathsf{i}\text{’} \rangle \\ * [1/2]\mathit{token\_rec}(t_1.\text{parent1}) \end{array} \right\} \\
&\qquad [\mathsf{b}] := \mathsf{c}; \\
&\qquad \textbf{let } \_ := (\mathsf{r}.\text{“}\mathsf{t1}\text{”}.\text{id}, \mathsf{r}.\text{“}\mathsf{t1}\text{”}.\text{iot}) := \mathsf{c} :: \langle\rangle \textbf{ in } \text{unit } ❸ \\
&\qquad \left\{ \begin{array}{l} \mathsf{b} \mapsto \mathsf{c} \\ * \mathbf{gc}(t_2.\text{id}, \text{left}(\text{init}), \langle \text{‘}\mathsf{h}\text{’} \rangle) * [1/2]\mathbf{gc}(t_2.\text{id}, \text{right}(\text{init}), \mathit{rg}) \\ * [1/2]\mathit{token\_rec}(t_2.\text{parent1}) \end{array} \right\} \\
&\qquad \left\{ \begin{array}{l} \mathit{buffer\_invar}(t_2.\text{id}, \mathsf{b}) \\ * \mathit{token\_rec}(t_2) \end{array} \right\} \\
&\quad \rangle \\
&\quad \{\ [t_2.\text{iot.f}]\boxed{\mathit{buffer\_invar}(t_2.\text{id}, \mathsf{b})} * \mathit{token\_rec}(t_2)\ \} \\
&\quad \{\ \mathit{token}(\mathsf{b}, t_2)\ \}
\end{aligned}$$

Informally, the proof outline works as follows. By entering the atomic section, we "gain access" to the invariant. The invariant contains a $1/2$ fraction of the ghost cell for the progress. The token contains the other $1/2$ fraction. So inside the atomic block we have a full ghost cell (i.e. not a small fraction of the ghost cell). Note that there is a write to that ghost cell (see ❸), i.e. the progress gets updated. This allows us to create the new token (which needs a $1/2$ fraction of the ghost cell with the new progress), which we need for the postcondition. Before leaving the atomic block, the invariant needs to be restored. Luckily *putchar_io* was heavily constrained to give us enough knowledge to do this.

In order to write to the ghost cell, the ID of the ghost cell and the key of the ghost cell must be known. The ID is t_1.id, but t_1 is a variable in the assertions, not a variable in the program and the program cannot access it. That is why the function has an extra argument r (see ❷). This argument is only used for ghost code. To be able to easily remove such arguments, we separate them from the real arguments with a semicolon (see ❷). The variable r is a record, and the definition of *putchar_io* constrains r to map the key "t1" to the value $\mathsf{t_1}$ (see ❶). That is why the code (❸) uses r."t1".id to access the ID to update the ghost cell.

Besides functions that manipulate tokens themselves, some functions only call other I/O style functions, like the ones below which are very easy to verify:

$print\_h = \lambda \mathsf{b}\ ; \mathsf{r}\ .$

$\quad \{\ token(\mathsf{b}, t_1) * putchar\_io(t_1, \text{‘h’}, t_2, \mathsf{r})\ \}$

$\quad putchar(\mathsf{b}, \text{‘h’}\ ; \mathsf{r}\ )$

$\quad \{\ token(\mathsf{b}, t_2)\ \}$

$print\_i = \lambda \mathsf{b}\ ; \mathsf{r}\ .$

$\quad \{\ token(\mathsf{b}, t_1) * putchar\_io(t_1, \text{‘i’}, t_2, \mathsf{r})\ \}$

$\quad putchar(\mathsf{b}, \text{‘i’}\ ; \mathsf{r}\ )$

$\quad \{\ token(\mathsf{b}, t_2)\ \}$

$print\_hi = \lambda \mathsf{b}\ ; \mathsf{r}\ .$

$$\left\{ \begin{array}{l} token(\mathsf{b}, t_1) * split\_io(t_1, t_{h1}, t_{i1}) \\ * putchar\_io(t_{h1}, \text{‘h’}, t_{h2}, \mathsf{r}.\text{“h”}) * putchar\_io(t_{i1}, \text{‘i’}, t_{i2}, \mathsf{r}.\text{“i”}) \\ * join\_io(t_{h2}, t_{i2}, t_2) \end{array} \right\}$$

$\quad \textbf{let}\ \_ := split()\ \textbf{in}$

$\quad \big(putchar(\mathsf{b}, \text{‘h’}\ ; \mathsf{r}.\text{“h”}\ ) \,||\, putchar(\mathsf{b}, \text{‘i’}\ ; \mathsf{r}.\text{“i”}\ )\big);$

```
    let _ := $join()$ in unit
    $\{\ token(\mathsf{b}, t_2)\ \}$
```

4.3.6 Main

Consider the following implementation and specification:

```
$main = \lambda.$
    $\{\ \mathbf{emp}\ \}$
    let b := cons(0) in
    $print\_hi(\mathsf{b})$;
    [b]
    $\{\ \mathsf{res} = \text{‘h’} \lor \mathsf{res} = \text{‘i’}\ \}$
```

main allocates a buffer, calls *print_hi* (which writes to this buffer), and returns the content of the buffer.

In order to call *print_hi*, *print_hi*'s precondition must hold before it is called. The verification of *main* will thus have to make sure *print_hi*'s I/O precondition holds, even though *main* does not have an I/O style precondition itself. The *main* function also does the inverse: after obtaining the I/O style postcondition of *print_hi*, *main* extracts knowledge about the memory state. It then expresses such knowledge in its non-I/O style postcondition.

To verify *main*, we add ghost code to make the precondition of *print_hi* hold before it is called, and we add ghost code to be able to "convert" the knowledge obtained from the I/O style postcondition of *print_hi* to the non-I/O style postcondition of *main*.

```
$main = \lambda.$
    $\{\ \mathbf{emp}\ \}$
    let b := cons(0) in
    let id := create_gcf() in
    let iot1 := init in
    let ioth := left(iot1) in
    let ioti := right(iot1) in
    let iot2 := join(ioth, ioti) in
let _ := $\mathbf{gcf\_cons}(\mathsf{id}, \mathsf{iot1}, \langle\rangle)$;
    $\mathbf{gcf\_cons}(\mathsf{id}, \mathsf{ioth}, \langle\rangle)$;
    $\mathbf{gcf\_cons}(\mathsf{id}, \mathsf{ioti}, \langle\rangle)$;
    $\mathbf{gcf\_cons}(\mathsf{id}, \mathsf{iot2}, \langle\rangle)$;
    atintro in
$\left\{ \begin{array}{l} \mathbf{gc}(\mathsf{id}, \mathsf{iot1}, \langle\rangle) * [1/2]\mathbf{gc}(\mathsf{id}, \mathsf{ioth}, \langle\rangle) * [1/2]\mathbf{gc}(\mathsf{id}, \mathsf{ioti}, \langle\rangle) \\ * \mathbf{gc}(\mathsf{id}, \mathsf{iot2}, \langle\rangle) * \boxed{\mathit{buffer\_invar}(\mathsf{id}, \mathsf{b})} \end{array} \right\}$
let t1 := $\mathit{place}(\mathsf{iot1}, \langle\rangle, \mathrm{none}, \mathrm{none}, \mathsf{id})$ in
let th1 := $\mathit{place}(\mathsf{ioth}, \langle\rangle, \mathsf{t1}, \mathrm{none}, \mathsf{id})$ in
let th2 := $\mathit{place}(\mathsf{ioth}, \langle\text{‘h’}\rangle, \mathsf{t1}, \mathrm{none}, \mathsf{id})$ in
let ti1 := $\mathit{place}(\mathsf{ioti}, \langle\rangle, \mathsf{t1}, \mathrm{none}, \mathsf{id})$ in
let ti2 := $\mathit{place}(\mathsf{ioti}, \langle\text{‘i’}\rangle, \mathsf{t1}, \mathrm{none}, \mathsf{id})$ in
let t2 := $\mathit{place}(\mathsf{iot2}, \langle\rangle, \mathsf{th2}, \mathsf{ti2}, \mathsf{id})$ in
let r := {“h” : {“t1” : th1}, “i” : {“t1” : ti1}} in
$\left\{ \begin{array}{l} \mathit{token}(\mathsf{b}, \mathsf{t1}) * \mathit{split\_io}(\mathsf{t1}, \mathsf{th1}, \mathsf{ti1}) * \mathit{putchar\_io}(\mathsf{th1}, \text{‘h’}, \mathsf{th2}, \mathsf{r}.\text{“h”}) \\ * \mathit{putchar\_io}(\mathsf{ti1}, \text{‘i’}, \mathsf{ti2}, \mathsf{r}.\text{“i”}) * \mathit{join\_io}(\mathsf{th2}, \mathsf{ti2}, \mathsf{t2}) \end{array} \right\}$
$\mathit{print\_hi}(\mathsf{b}\ ; \mathsf{r}\ )$
$\{\ \mathit{token}(\mathsf{b}, \mathsf{t2})\ \}$
$\left\{\ \mathit{token\_rec}(\mathsf{t2}) * \boxed{\mathit{buffer\_invar}(\mathsf{id}, \mathsf{b})}\ \right\}$
let _ := atdel in
$\left\{ \begin{array}{l} \mathsf{b} \mapsto ?c \\ * \mathbf{gc}(\mathsf{id}, \mathrm{left}(\mathrm{init}), \langle\text{‘h’}\rangle) * \mathbf{gc}(\mathsf{id}, \mathrm{right}(\mathrm{init}), \langle\text{‘i’}\rangle) \\ * (c = \text{‘h’} \vee c = \text{‘i’}) \end{array} \right\}$
[b]
$\{\ \mathsf{res} = \text{‘h’} \vee \mathsf{res} = \text{‘i’}\ \}$
```

The ghost code that we insert in *main* creates the ghost cells (**create_gcf**() and **gcf_cons**(...)). It also introduces the atomic space: the **atintro** ghost code allows one to “convert” the assertion $\mathit{buffer\_invar}(\mathsf{id}, \mathsf{b})$ into $\boxed{\mathit{buffer\_invar}(\mathsf{id}, \mathsf{b})}$.

4.3.7 Formalization: Concurrent programming language

To explain part of the approach for I/O style verification of memory manipulating programs, we used a programming language that we so far only explained informally. Before we continue, we define the part of the programming language used so far more formally.

We define a programming language from scratch, i.e. not an extension of the previously formally defined language of Chapter 3.

Definition 33: Syntax

Values, ghost levels, expressions, operations, and commands are defined (in that order) syntactically as follows:

$$v, t ::= \ i \mid \bar{i} \mid \text{true} \mid \text{false} \mid \text{unit} \mid \lambda\overline{x^{\text{r}}};\overline{y^{\text{g}}}.\, c \mid \bot \mid \{\overline{f : v}\} \ \mid \mathit{iot} \mid \mathit{place}$$
$$l ::= \ \text{g} \mid \text{r}$$
$$e ::= \ v \mid x \mid \mathit{op}(\overline{e})$$
$$\mathit{op} ::= \ + \mid - \mid * \mid = \mid \text{last} \mid \text{cons} \mid \ {+}{+} \ \mid \ldots$$
$$c ::= \ e \mid \textbf{let}\ x := c\ \textbf{in}\ c \mid e(\overline{e};\overline{e}) \mid \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c \mid c \,||\, c \mid [e] \mid [e] := e \mid \textbf{cons}(\overline{e})$$

A program is a closed command, i.e. a command with no free variables.

i ranges over integers. $\{\overline{f : v}\}$ is a record: a partial function from strings to values.

Operations and expressions Notice that we defined operations syntactically. So + is just considered a syntactic thing, not a mathematical function.

An expression can consists of an operation application, such as $+(1, 1)$. We often write operation expressions in infix notation. For example we write $1 + 1$ for the expression $+(1, 1)$. To actually evaluate operation application expressions, we assume a function called eval:

Definition 34: eval

We assume a function called eval that given an operation and a list of arguments, returns a value.

Definition 35: Expression evaluation

Evaluation of a closed expression is defined inductively as follows:

- $[\![v]\!] = v$
- $[\![op(\overline{E})]\!] = \mathrm{eval}(op, [\![\overline{E}]\!])$

Here we write $[\![E_1, E_2, \ldots, E_n]\!]$ for $[\![E_1]\!], [\![E_2]\!], \ldots, [\![E_n]\!]$.

We assume eval to be a total function on all values. For nonsense calculations eval returns the error value ($\bot$). This is just a value like any other value. For example $[\![42/0]\!] = \bot$, $[\![1 + \mathrm{true}]\!] = \bot$, $[\![\mathrm{tail}(\langle\rangle)]\!] = \bot$, and $[\![\bot + 7]\!] = \bot$.

Lambda values Lambda values $(\lambda \overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}}.\, c)$ have two sets of arguments ($\overline{x^{\mathrm{r}}}$ and $\overline{y^{\mathrm{g}}}$). The first set are the real arguments and the second set are ghost arguments. This distinction will be necessary when we study erasure (Sec. 4.7).

Heap We verify memory behavior in an I/O style, so we do not need a BIO command for doing input/output, because everything happens in memory.

Heap chunks map addresses to values (we postpone formalizing ghost cell families until Sec. 4.3.9):

Definition 36: Heap chunks ($C \in \mathrm{Chunks}$)

We define heap chunks (also called "chunks") as follows:

$C ::= \; v \mapsto v$

We write Chunks for the set of all heap chunks.

A heap maps a chunk to a fraction:

Definition 37: Heap ($h \in \mathrm{Heaps}$)

$\mathrm{Heaps} = \mathrm{Chunks} \to \mathbb{Q}^{+}$

We range over Heaps with h.

$$\frac{\tau \in \tau_1 \,||\, \tau_2}{(h, \tilde{h}) \cdot \tau \in ((h, \tilde{h}) \cdot \tau_1) \,||\, \tau_2} \text{ParLeft} \qquad \frac{\tau \in \tau_1 \,||\, \tau_2}{(h, \tilde{h}) \cdot \tau \in \tau_1 \,||\, ((h, \tilde{h}) \cdot \tau_2)} \text{ParRight}$$

$$\frac{}{\epsilon \in \epsilon \,||\, \epsilon} \text{ParEmpty}$$

Figure 4.2: Parallel composition (used in Fig. 4.3 on the following page)

So Heaps is the set of all heaps, ranged over by h. We write heaps like $\{\![[1/2]4 \mapsto 5, 6 \mapsto 7]\!\}$, which means that the heap maps the chunk $4 \mapsto 5$ to the fraction $1/2$, the chunk $6 \mapsto 7$ to the fraction 1, and all other chunks to fraction 0.

Program evaluation A program can be an expression like $1 + 1$. The program $1 + 1$ should evaluate to 2. Let us look now how this is defined more formally. The semantics of the programming language is defined in a big step style: we write $c \Downarrow \tau, v$ to express that an execution of the command c can yield the return value v (ignore τ for now).

Definition 38: Step semantics ($c \Downarrow \tau, v$)

Execution of a command (i.e. the step semantics) is defined coinductively in Fig. 4.3 on the next page.

We will extend the definition of execution of a command later (Fig. 4.7 p. 114, Fig. 4.9 p. 116, Fig. 4.13 p. 140) to add more features. Let us look at the Exp step rule. It states that a command of the form e evaluates to $\epsilon, [\![e]\!]$. (again, ignore ϵ for now). For example, $1 + 1 \Downarrow \epsilon, 2$. In other words: the program $1 + 1$ can evaluate to $\epsilon, 2$.

Note that we wrote "can evaluate" instead of "will evaluate": the programming language is not deterministic, so in some cases it is possible there are multiple executions of a program, each returning a different value. In other words, it is possible that $c \Downarrow \tau_1, v_1$ and $c \Downarrow \tau_2, v_2$ where $v_1 \neq v_2$.

This nondeterminism is the case for memory allocation (which can yield a different memory address allocated) and parallel execution (which can have multiple interleavings). But how can one define a big step semantics that

$$\frac{c_1 \Downarrow \tau_1, \_ \qquad c_2 \Downarrow \tau_2, \_ \qquad \tau \in \tau_1 \,||\, \tau_2}{c_1 \,||\, c_2 \Downarrow \tau, \mathrm{unit}} \text{ Par}$$

$$\frac{c_1 \Downarrow \tau_1, v_1 \qquad c_2[v_1/x] \Downarrow \tau_2, v_2}{\mathbf{let}\ x := c_1\ \mathbf{in}\ c_2 \Downarrow \tau_1; \tau_2, v_2} \text{ Let}$$

$$\frac{}{[v_1] := v_2 \Downarrow (h \uplus \{\!\!\{v_1 \mapsto \_\}\!\!\}, h \uplus \{\!\!\{v_1 \mapsto v_2\}\!\!\}) \cdot \epsilon, \mathrm{unit}} \text{ Mut}$$

$$\frac{\nexists v'. h(v_1 \mapsto v') \geq 1}{[v_1] := v_2 \Downarrow (h, \bot) \cdot \epsilon, \bot} \text{ MutErr} \qquad \frac{c[\overline{v_1}/\overline{x}][\overline{v_2}/\overline{y}] \Downarrow \tau, v}{(\lambda \overline{x}; \overline{y}.\, c)(\overline{v_1}; \overline{v_2}) \Downarrow (h, h) \cdot \tau, v} \text{ App}$$

$$\frac{\nexists \overline{x}, \overline{y}, c.\ v = \lambda \overline{x}; \overline{y}.\, c}{v(\overline{v_1}; \overline{v_2}) \Downarrow (h, \bot) \cdot \epsilon, \bot} \text{ AppErr} \qquad \frac{c_1 \Downarrow \tau, v}{\mathbf{if}\ \mathrm{true}\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \Downarrow \tau, v} \text{ Then}$$

$$\frac{v \neq \mathrm{true} \qquad c_2 \Downarrow \tau, v'}{\mathbf{if}\ v\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \Downarrow \tau, v'} \text{ Else} \qquad \frac{}{e \Downarrow \epsilon, [\![e]\!]} \text{ Exp}$$

$$\frac{h(v \mapsto v') > 0}{[v] \Downarrow (h, h) \cdot \epsilon, v'} \text{ Lookup} \qquad \frac{\nexists v'. h(v \mapsto v') > 0}{[v] \Downarrow (h, \bot) \cdot \epsilon, \bot} \text{ LookupErr}$$

$$\frac{h' = \{\!\!\{n \mapsto v_0, n+1 \mapsto v_1, \ldots, n+k \mapsto v_k\}\!\!\}}{\mathbf{cons}(v_0, v_1, \ldots, v_k) \Downarrow (h, h \uplus h') \cdot \epsilon, n} \text{ Cons}$$

Figure 4.3: Step semantics. We identify closed expressions with their values.

associates a command with the final result in such a concurrent setting? To deal with this, we use *trace*s of pairs of heaps[4], somewhat similar to [70]. Formally, this is defined as follows:

Definition 39: Traces (τ)

$$\tilde{h} ::= h \mid \bot \mid \textbf{inf}$$
$$\tau ::= \epsilon \mid (h, \tilde{h}) \cdot \tau$$

Here, the definition of τ must be interpreted coinductively. To understand this definition, we start with the following example of a trace:

$(\{\!\!\{1 \mapsto 2\}\!\!\}, \{\!\!\{1 \mapsto 7\}\!\!\}) \cdot (\{\!\!\{1 \mapsto 7\}\!\!\}, \{\!\!\{1 \mapsto 7, 3 \mapsto 4\}\!\!\}) \cdot \epsilon.$

This trace expresses that the program starts with the heap $\{\!\!\{1 \mapsto 2\}\!\!\}$. The first pair in the trace expresses that the run of the program transforms the heap by writing 7 to address 1. The second pair of heaps says memory is allocated at address 3 with content 4.

In the example trace, the last item of a pair equals the first item of the next pair. This must be the case when the trace represents all heap modifications of all threads of a program. But the power of the trace semantics is that we can also consider a subset of the set of threads of a program by studying traces where the last item of a pair does not equal the first item of the next pair. It is possible another thread did something between the second item of a pair and the first item of the next pair. Consider this example:

$\tau_1 = (\{\!\!\{1 \mapsto 2\}\!\!\}, \{\!\!\{1 \mapsto 7\}\!\!\}) \cdot (\{\!\!\{1 \mapsto 10\}\!\!\}, \{\!\!\{1 \mapsto 15, 3 \mapsto 4\}\!\!\}) \cdot \epsilon.$

We see that between $\{\!\!\{1 \mapsto 7\}\!\!\}$ and $\{\!\!\{1 \mapsto 10\}\!\!\}$, another thread must have written 10 to address 1. Note that many other things might have happened in between, maybe yet another thread allocated something and then deallocated that.

Let us look at the Cons step rule of the step semantics. It uses the $\uplus$ operation: $h_1 \uplus h_2$ yields the heap that has all elements of both h_1 and h_2, if h_1 and h_2 do not have a conflicting opinion on which address is mapped to which value. More formally:

Definition 40: agree(C, C)

$$\text{agree}(v_1 \mapsto v_2, v_1' \mapsto v_2') = v_1 \neq v_1' \lor v_2 = v_2'$$

[4] and of $\bot$ and **inf**

Definition 41: Sum of heaps ($h \uplus h$)

$$h_1 \uplus h_2 = \begin{array}{l} \textbf{if } \forall C_1 \in h_1, C_2 \in h_2.\text{agree}(C_1, C_2) \textbf{ then } (\lambda C.h_1(C) + h_2(C)) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\; \textbf{else } \text{undefined} \end{array}$$

Going back to the Cons step rule, the step rule states that execution of a memory allocation yields a trace of only one pair of heaps, where the second item of the pair adds the newly allocated memory.

Now we look at the Let step rule. It states that, if c_1 yields trace τ_1 and c_2 yields trace τ_2, then executing c_2 after c_1 yields $\tau_1; \tau_2$.

You can think of $\tau_1; \tau_2$ as just adding τ_2 after τ_1. In case τ_1 is infinite, $\tau_1; \tau_2$ is equal to τ_1. More formally:

Definition 42: Sequential composition of traces ($\tau; \tau$)

We define $\tau_1; \tau_2$ corecursively as follows

$$((h, \tilde{h}) \cdot \tau); \tau' = (h, \tilde{h}) \cdot (\tau; \tau')$$
$$\epsilon; \tau = \tau$$

The Par step rule describes parallel composition: the command $c_1||c_2$ executes c_1 in parallel with c_2. The step rule states that if c_1 evaluates to a trace τ_1, and c_2 evaluates to a trace τ_2, then take one of the potentially many traces that are an interleaving of τ_1 with τ_2, and $c_1||c_2$ will evaluate to that trace. It will probably also evaluate to other traces.

Definition 43: Parallel composition of traces ($\tau \in \tau \,||\, \tau$)

When a trace is an interleaving of two other traces is defined coinductively in Fig. 4.2 on page 105.

Remember that in a trace, it is possible that the last item of a pair of the trace does not equal the first item of the next pair. In case we are looking at the traces produced by a full program – so no other threads are executed concurrently with it – this will not be the case. For the full program, we ignore such traces because they cannot happen. To do that, we use the concept of an outcome.

$$\frac{}{h, (h, \bot) \cdot \tau \to (0, \bot)} \qquad \frac{h_2, \tau \to O}{h_1, (h_1, h_2) \cdot \tau \to O + 1} \qquad \frac{}{h, \epsilon \to (0, h)}$$

$$\frac{}{h, \mathbf{inf} \cdot \tau \to \mathbf{inf}}$$

Figure 4.4: Outcomes

Definition 44: Outcomes (O)

Outcomes are defined inductively as follows:

$O ::= (n, h) \mid (n, \bot) \mid \mathbf{inf}$

The first outcome, (n, h), states the execution terminates with heap h after n steps. The second outcome, $(n, \bot)$ states the execution crashes after n (non-crashing) steps. The third outcome, **inf**, states the execution goes into an infinite loop.

Given a heap with which the program starts and a trace of an execution of the program, we associate an outcome with it.

Definition 45: Outcome increase ($O + 1$)

We define increasing an outcome (written $O + 1$) as follows:

- $(n, h) + 1 = (n + 1, h)$
- $(n, \bot) + 1 = (n + 1, \bot)$
- $\mathbf{inf} + 1 = \mathbf{inf}$

Definition 46: Outcome of a trace ($h, \tau \to O$))

With a trace and a startheap, we associate an outcome, as defined coinductively in Fig. 4.4. It uses Def. 45.

$$\frac{h = h_1 \uplus h_2 \quad h_1 \vDash_F P_1 \quad h_2 \vDash_F P_2}{h \vDash_F P_1 * P_2}\ \text{Sep} \qquad \frac{}{\{\![\pi]C\}\!\} \vDash_F [\pi]C}\ \text{Chunk}$$

$$\frac{(h \vDash_F P_1) \vee (h \vDash_F P_2)}{h \vDash_F P_1 \vee P_2}\ \text{Or} \qquad \frac{h \vDash_F [\pi]P[v/x]}{h \vDash_F [\pi]\exists x.P}\ \text{Exists} \qquad \frac{[\![e]\!] = \text{true}}{\{\![\,]\!\} \vDash_F [\pi]e}\ \text{Exp}$$

$$\frac{h \vDash_F [\pi]P_1 * [\pi]P_2}{h \vDash_F [\pi](P_1 * P_2)}\ \text{PiStar} \qquad \frac{h \vDash_F [\pi]P_1 \vee [\pi]P_2}{h \vDash_F [\pi](P_1 \vee P_2)}\ \text{PiOr}$$

$$\frac{h \vDash_F [\pi]P[\overline{v}/\overline{x}] \quad \text{fv}(P) \subseteq \overline{x}}{h \vDash_F [\pi](\lambda\overline{x}.\,P)(\overline{v})}\ \text{Pred} \qquad \frac{h \vDash_F [\pi_1 \times \pi_2]P}{h \vDash_F [\pi_1][\pi_2]P}\ \text{NestFrac}$$

Figure 4.5: Satisfaction relation of assertions. We identify closed expressions with their values.

Because $n \in \mathbb{N}$, we know (n, h) and $(n, \bot)$ never describe an infinite execution.

Assertions and proof rules We extend values with predicate values:

Definition 47: Values extended with predicate values

$v ::= \ \ldots \mid \lambda\overline{x}.\,P$

An assertion can be an expression, a separating conjuction, a points-to assertion, a disjunction, an existential quantification, an operation, a fractional assertion [7, 8], or a predicate value application (similar to [54] besides that we allow predicates to be values).

Definition 48: Assertions (P, Q)

Assertions are defined syntactically as follows:

$$P, Q ::= \ e \mid P * P \mid e \mapsto e \mid P \vee P \mid \exists x.P \mid [\pi]P \mid e(\overline{e})$$

emp is a shorthand for the assertion true.

A heap can be a model for a closed assertion.

Definition 49: Satisfaction relation of assertions ($h \vDash_F P$)

Fig. 4.5 on the preceding page defines inductively when a heap is a model for an assertion.

The subscript (F in $h \vDash_F P$) can be ignored for now; it will be important when we study higher order functions (Sec. 4.4.6).

Definition 50: Proof rules ($\vdash \{P\}\, c\, \{Q\}_I^l$)

Proof rules are defined inductively in Fig. 4.6 on the following page.

Each rule with conclusion $\vdash \{P\}\, c\, \{Q\}_I^l$ has an implicit side condition $\mathrm{fv}(P) = \emptyset$ and $\mathrm{fv}(Q) \subseteq \{\mathsf{res}\}$.

One can ignore the subscript and superscript (l and I in $\vdash \{P\}\, c\, \{Q\}_I^l$) for now, they will be important when we study atomic commands (Sec. 4.3.8 p. 115), higher order functions (Sec. 4.4.6), and erasure (Sec. 4.7).

The Conseq proof rule uses $P \to Q$, which expresses that "P implies Q". More formally:

Definition 51: $P \to Q$

$$P \to Q \iff \forall h, F.\ h \vDash_F P \Rightarrow h \vDash_F Q$$

$$\frac{\vdash \{P\}\, c\, \{Q * R\}_I^l}{\vdash \{P\}\, c\, \{Q\}_I^l}\ \text{Leak} \qquad \frac{\vdash \{P\}\, c_1\, \{Q\}_I^l}{\vdash \{P\}\, \textbf{if}\ \text{true}\ \textbf{then}\ c_1\ \textbf{else}\ c_2\, \{Q\}_I^l}\ \text{Then}$$

$$\frac{\vdash \{P\}\, c_2\, \{Q\}_I^l \qquad v \neq \text{true}}{\vdash \{P\}\, \textbf{if}\ v\ \textbf{then}\ c_1\ \textbf{else}\ c_2\, \{Q\}_I^l}\ \text{Else} \qquad \frac{[\![e]\!] = v}{\vdash \{\textbf{emp}\}\, e\, \{\mathsf{res} = v\}_I^l}\ \text{Exp}$$

$$\frac{\vdash \{P\}\, c[\overline{v_1}/\overline{x^{\mathrm{r}}}][\overline{v_2}/\overline{y^{\mathrm{g}}}]\, \{Q\}_I^l}{\vdash \{P\}\, (\lambda \overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}}.\, c)(\overline{v_1}; \overline{v_2})\, \{Q\}_I^l}\ \text{AppSimple} \qquad \frac{}{\vdash \{\text{false}\}\, c\, \{Q\}_I^l}\ \text{False}$$

$$\frac{}{\vdash \{v \mapsto \_\}\, [v] := v'\, \{v \mapsto v'\}_I^{\mathrm{r}}}\ \text{Mut} \qquad \frac{\forall v.\, \vdash \{P[v/x]\}\, c\, \{Q\}_I^l}{\vdash \{\exists x. P\}\, c\, \{Q\}_I^l}\ \text{Exists}$$

$$\frac{\vdash \{P_1\}\, c\, \{Q\}_I^l \qquad \vdash \{P_2\}\, c\, \{Q\}_I^l}{\vdash \{P_1 \vee P_2\}\, c\, \{Q\}_I^l}\ \text{Disj} \qquad \frac{\vdash \{P\}\, c\, \{Q\}_I^l \qquad \mathrm{fv}(R) = \emptyset}{\vdash \{P * R\}\, c\, \{Q * R\}_I^l}\ \text{Frame}$$

$$\frac{P \to P' \qquad \vdash \{P'\}\, c\, \{Q'\}_I^l \qquad Q' \to Q}{\vdash \{P\}\, c\, \{Q\}_I^l}\ \text{Conseq}$$

$$\frac{\vdash \{P\}\, c_1\, \{Q'\}_I^{\mathrm{r}} \qquad \forall v.\, \vdash \{Q'[v/\mathsf{res}]\}\, c_2[v/x^{\mathrm{r}}]\, \{Q\}_I^{\mathrm{r}}}{\vdash \{P\}\, \textbf{let}\ x^{\mathrm{r}} := c_1\ \textbf{in}\ c_2\, \{Q\}_I^{\mathrm{r}}}\ \text{LetR}$$

$$\frac{\vdash \{P\}\, c_1\, \{Q'\}_I^{\mathrm{g}} \qquad \forall v.\, \vdash \{Q'[v/\mathsf{res}]\}\, c_2[v/x^{\mathrm{g}}]\, \{Q\}_I^l}{\vdash \{P\}\, \textbf{let}\ x^{\mathrm{g}} := c_1\ \textbf{in}\ c_2\, \{Q\}_I^l}\ \text{LetG}$$

$$\frac{}{\vdash \{[\pi]v \mapsto v'\}\, [v]\, \{\mathsf{res} = v' * [\pi]v \mapsto v'\}_I^l}\ \text{Lookup}$$

$$\frac{\overline{v} = v_0, v_1, \ldots, v_k}{\vdash \{\textbf{emp}\}\, \textbf{cons}(\overline{v})\, \{\mathsf{res} \mapsto v_0 * \mathsf{res} + 1 \mapsto v_1 * \ldots * \mathsf{res} + k \mapsto v_k\}_I^{\mathrm{r}}}\ \text{Cons}$$

$$\frac{\vdash \{P_1\}\, c_1\, \{Q_1\}_I^l \qquad \vdash \{P_2\}\, c_2\, \{Q_2\}_I^l \qquad \mathrm{fv}(Q_1) = \mathrm{fv}(Q_2) = \emptyset}{\vdash \{P_1 * P_2\}\, c_1 \,||\, c_2\, \{Q_1 * Q_2\}_I^l}\ \text{Par}$$

Figure 4.6: Proof rules

4.3.8 Formalization: Atomic blocks

In this subsection we formalize atomic blocks (which were introduced informally in Sec. 4.3.5 on page 98) and atomic spaces (which were introduced informally in Sec. 4.3.3 on page 94).

We extend the syntax with atomic blocks:

Definition 52: Commands ext. with atomic block command

$c ::= \ldots \mid \langle c \rangle$

Remember that an atomic block is executed atomic, i.e. no other thread will do anything while the atomic block is being executed. As a result, when executing an atomic block, one can be sure that no other thread will manipulate the heap.

Remember that an atomic space assertion is an assertion of the form $[\pi]\boxed{P}$, where P is another assertion and π is a fraction size. As long as we know that $[\pi]\boxed{P}$ holds, we know that P is true. For example, if we have the assertion $[\pi]\boxed{\exists \mathsf{x}.\ 7 \mapsto \mathsf{x} * \mathsf{x} > 10}$, we know that, at every point in the execution of the program, the address 7 will contain a value that is greater than 10.

Now we combine the concept of an atomic block with the concept of an atomic space assertion. The sub-assertion P of the atomic space assertion must hold before executing the atomic block, and after the atomic block terminates. It is fine to invalidate P during execution of the atomic block, as long as P holds again when the atomic block ends. For example, with the atomic space assertion $[\pi]\boxed{\exists \mathsf{x}.7 \mapsto \mathsf{x} * \mathsf{x} > 10}$ it is fine to have a program $\langle [7] := 2; [7] := 11 \rangle$.

We add a command to introduce an atomic space, and another command to delete an atomic space:

Definition 53: Commands ext. with atomic space commands

$c ::= \ldots \mid \textbf{atintro} \mid \textbf{atdel}$

We add a special type of heap chunk, called an atomic space chunk:

Definition 54: Chunks extended with atomic space chunk

$C ::= \ldots \mid \boxed{P}$

$$\frac{c \Downarrow \tau, v \qquad h_1, \tau \to (n, h_2)}{\langle c \rangle \Downarrow (h_1, h_2) \cdot \epsilon, v}\ \text{Atom} \qquad \frac{c \Downarrow \tau, v \qquad h, \tau \to (n, \bot)}{\langle c \rangle \Downarrow (h, \bot) \cdot \epsilon, v}\ \text{AtomErr}$$

$$\frac{c \Downarrow \tau, v \qquad h, \tau \to \textbf{inf}}{\langle c \rangle \Downarrow (h, \textbf{inf}) \cdot \epsilon, \text{unit}}\ \text{AtomInf} \qquad \frac{}{\textbf{atintro} \Downarrow (h, h) \cdot \epsilon, \text{unit}}\ \text{AtomIntro}$$

$$\frac{}{\textbf{atdel} \Downarrow (h, h) \cdot \epsilon, \text{unit}}\ \text{AtomDel}$$

Figure 4.7: Step semantics for atomic blocks

$$\frac{\vdash \left\{P\ * P_1 * \ldots * P_k\right\} c \left\{Q\ * P_1 * \ldots * P_k\right\}^{l}_{\text{inat}} \qquad \forall i, j.\ i \neq j \Rightarrow P_i \neq P_j}{\vdash \left\{P * [\pi_1]\boxed{P_1} * \ldots * [\pi_k]\boxed{P_k}\right\} \langle c \rangle \left\{Q * [\pi_1]\boxed{P_1} * \ldots * [\pi_k]\boxed{P_k}\right\}^{l}_{\text{outat}}}\ \text{Atom}$$

$$\frac{}{\vdash \left\{P\right\} \textbf{atintro} \left\{\boxed{P}\right\}^{\text{g}}_{\text{outat}}}\ \text{AtIntro} \qquad \frac{}{\vdash \left\{\boxed{P}\right\} \textbf{atdel} \left\{P\right\}^{\text{g}}_{\text{outat}}}\ \text{AtDel}$$

Figure 4.8: Proof rules for atomic blocks

We add a corresponding atomic space chunk assertion:

Definition 55: Asn. ext. with atomic space chunk assertion

$$P ::= \ \ldots \mid \boxed{P}$$

The command **atintro** does not actually do something while executing the program: the heap before the command is the same as the heap after it. This is reflected in the AtomIntro step rule in Fig. 4.7. Now switch from the concrete execution point of view to the verification point of view: the AtIntro proof rule in Fig. 4.8 states that, if one has access to some memory described by an assertion P before executing **atintro**, we lose this access (in the postcondition), but gain the knowledge that there exists an atomic space with assertion P.

So a heap chunk of the form $\boxed{P}$ is only used for verification; during concrete execution it will never exist.

Definition 56: Program execution extended for atomic blocks

Fig. 4.7 on the preceding page extends the step semantics for atomic blocks.

The AtomErr step rule states that, if a command c evaluates to a trace, and this trace crashes (according to the definition of outcomes), then $\langle c\rangle$ also crashes.

Similarly, AtomInf states that if c goes into an infinite loop, then $\langle c\rangle$ also goes into an infinite loop.

The Atom step rule says that if c evaluates to a trace τ, and starting from heap h_1 the trace τ ends in n steps with heap h_2 without crashing (or formally: $h_1, \tau \to (n, h_2)$) then $\langle c\rangle$ yields a trace of only one step: (h_1, h_2). For example, if executing c yields a very long trace $(h_1, h_a) \cdot (h_a, h_b) \cdot \ldots \cdot (h_z, h_2) \cdot \epsilon$, then executing $\langle c\rangle$ yields the short trace $(h_1, h_2) \cdot \epsilon$. Note that for the long trace τ, because of the definition of outcomes, we only consider traces for which every second item of a tuple of the trace equals the first item of the next tuple. So we do not consider $\tau = (h_1, h_a) \cdot (h_b, h_2) \cdot \epsilon$ when $h_a \neq h_b$, because then $h_1, \tau \to (n, h_2)$ will not hold.

Also remember that execution is not deterministic: it is possible that both the Atom and the AtomErr step rule are applicable.

The Atom proof rule states that, in order to prove $\langle c\rangle$ with a precondition and a postcondition, it suffices to prove c instead. Additionally, while proving c, one can access the part of the heap "covered" by the atomic space of the precondition, as long as they still hold in the postcondition. To avoid gaining access twice to resources covered by atomic spaces, we disallow using this proof rule to prove the premise of the proof rule. This has the limitation one cannot nest atomic blocks ($\langle\langle c\rangle\rangle$), but we haven't needed this in the examples we considered. Gaining access twice to resources covered by an atomic space is prevented by annotating Hoare triples with whether they are considered inside or outside an atomic block: $\{P\}\, c\, \{Q\}^{l}_{\text{outat}}$ denotes we are outside an atomic block, $\{P\}\, c\, \{Q\}^{l}_{\text{inat}}$ denotes we are inside. I is used to range over $\{\text{inat}, \text{outat}\}$.

Note that the Atom proof rule only requires a fraction of the atomic space (i.e. the precondition contains $[\pi]\boxed{P}$ instead of $\boxed{P}$): this way one can spread fractions of atomic spaces over threads, and each thread can still access the resources covered by the atomic spaces in its atomic blocks.

$$\frac{\mathbf{gcf}(v, \_) \notin h}{\mathbf{create\_gcf}() \Downarrow (h, h \uplus \{\!\!\{\mathbf{gcf}(v, \langle\rangle)\}\!\!\}) \cdot \epsilon, v}$$

$$\frac{v_{key} \notin \overline{v} \qquad h' = h \uplus \{\!\!\{\mathbf{gcf}(v_{id}, v_{key} :: \overline{v}), \mathbf{gc}(v_{id}, v_{key}, v)\}\!\!\}}{\mathbf{gcf\_cons}(v_{id}, v_{key}, v) \Downarrow (h \uplus \{\!\!\{\mathbf{gcf}(v_{id}, \overline{v})\}\!\!\}, h') \cdot \epsilon, \text{unit}}$$

$$\frac{}{(v_{id}, v_{key}) := v \Downarrow (h \uplus \{\!\!\{\mathbf{gc}(v_{id}, v_{key}, \_)\}\!\!\}, h \uplus \{\!\!\{\mathbf{gc}(v_{id}, v_{key}, v)\}\!\!\}) \cdot \tau, v}$$

$$\frac{\mathbf{gc}(v_{id}, v_{key}, \_) \notin h}{(v_{id}, v_{key}) := v \Downarrow (h, \bot) \cdot \epsilon, \bot}$$

Figure 4.9: Ghost cell families step semantics

The **atdel** command, just like **atintro**, does not modify the heap while being executed. Now switch to the verification point of view (see AtDel rule in Fig. 4.8 on page 114): when one has access to knowledge that an atomic space exists (and this knowledge is not a fraction), i.e. when one has $\boxed{P}$, one can swap this with access to the memory covered by P.

Definition 57: Proof rules extended for atomics

Fig. 4.8 on page 114 extends the proof rules for atomic blocks, **atintro**, and **atdel**.

4.3.9 Formalization: Ghost cell families

In this subsection we formalize ghost cell families. Ghost cell families were informally explained in Sec. 4.3.3.

We add heap chunks for ghost cell families and ghost cells:

Definition 58: Heap chunks extended for ghost cell families

$C ::= \ \ldots \mid \mathbf{gcf}(v, \overline{v}) \mid \mathbf{gc}(v, v, v)$

$$\frac{}{\vdash \{\mathbf{emp}\}\ \mathbf{create\_gcf}()\ \{\mathbf{gcf}(\mathsf{res}, \langle\rangle)\}^{\mathrm{g}}_{I}}\ \text{CreateGcf}$$

$$\frac{v_{key} \notin \overline{v}}{\vdash \left\{\mathbf{gcf}(v_{id}, \overline{v})\right\}\ \mathbf{gcf\_cons}(v_{id}, v_{key}, v) \left\{\begin{matrix}\mathbf{gcf}(v_{id}, v_{key} :: \overline{v}) \\ * \ \mathbf{gc}(v_{id}, v_{key}, v)\end{matrix}\right\}^{\mathrm{g}}_{I}}\ \text{GcfCons}$$

$$\frac{}{\vdash \{\mathbf{gc}(v_{id}, v_{key}, \_)\}\ (v_{id}, v_{key}) := v\ \{\mathbf{gc}(v_{id}, v_{key}, v)\}^{\mathrm{g}}_{I}}\ \text{GcfMut}$$

Figure 4.10: Ghost cell families proof rules

We add the corresponding assertions:

Definition 59: Assertions extended for ghost cell families

$P ::= \ \dots \mid \mathbf{gcf}(e, \overline{e}) \mid \mathbf{gc}(e, e, e)$

Definition 60: Commands extended for ghost cell families

$c ::= \ \dots \mid \mathbf{create\_gcf}() \mid \mathbf{gcf\_cons}(e, e, e) \mid (e, e) := e$

Definition 61: Step rules extended for ghost cell families

Step rules for ghost cell families are listed in Fig. 4.9 on the facing page.

Definition 62: Proof rules extended for ghost cell families

Proof rules for ghost cell families are in Fig. 4.10.

The examples in this thesis only write to ghost cells but do not read from it. So we only include a command to write to a ghost cell ($(e, e) := e$) and not one to read from a ghost cell, but it easy to add such a command.

4.4 Reusability

In the previous examples, the proof of putchar is not reusable across applications. The verification of *putchar* uses knowledge of how *buffer_invar* is defined, but part of the definition of *buffer_invar* is application-specific. The application-specific part consists of a $1/2$ fraction of the ghost cells for the progresses, and how these progresses are related to each other and to the buffer content. This part is indeed application-specific: a different application can have a different Petri net and therefore different I/O threads and different progresses. The progresses can be related differently to each other and to the buffer content.

Ideally, we would like to write a specification for *putchar* and verify it once and for all, and then reuse it in multiple programs that have different specifications and therefore also different progresses and relationships between them.

We solve this problem by doing three things.

First, we move the application-specific part of *buffer_invar* to a user-defined predicate called the I/O invariant, which is passed as an extra argument to *buffer_invar*:

$$
\begin{array}{l}
\textbf{predicate}\ \textit{buffer\_invar} = \lambda \mathsf{id}, \mathsf{io\_inv}, \mathsf{b}. \\
\exists \mathsf{v}. \\
\qquad \textit{buf}(\mathsf{b}, \mathsf{v}) \\
\qquad * \ \mathsf{io\_inv}(\mathsf{id}, \mathsf{v})
\end{array}
$$

Here, $\textit{buf}(\mathsf{b}, \mathsf{v})$ is the memory state, such as $\mathsf{b} \mapsto \mathsf{v}$ (i.e. at address b is value v).

We change the definition of *token* such that it passes the I/O invariant to *buffer_invar* as an argument. *token* itself extracts the I/O invariant from the place.

$$
\begin{array}{l}
\textbf{predicate}\ \textit{token} = \lambda \mathsf{b}, \mathsf{t}. \\
\quad [t.\mathrm{iot.f}] \boxed{\textit{buffer\_invar}(\mathsf{t}.\mathrm{id}, \mathsf{t}.\mathrm{invar}, \mathsf{b})} \\
\quad * \ \textit{token\_rec}(\mathsf{t})
\end{array}
$$

split, *join*, *token_rec*, and *buffer* remain unchanged.

Second, we put the I/O invariant in the place: it is the one but last argument of the place constructor:

$$\textit{place} ::= \ \mathrm{none} \mid \mathrm{place}(\textit{iot}, \textit{v}, \textit{place}, \textit{place}, \lambda \textit{id}.\, P, \textit{v}_{\textit{id}})$$

This argument is an assertion lambda.

Because the place constructor now has an extra argument, we have to update the definition of *split_io* and *join_io* because they use this constructor:

$$
\begin{array}{l}
\textbf{predicate } \mathit{split\_io} = \lambda \mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3. \\
\quad \mathsf{t}_2 = \mathrm{place}(\mathrm{left}(\mathsf{t}_1.\mathrm{iot}), \langle\rangle, \mathsf{t}_1, \mathrm{none}, \mathsf{t}_1.\mathrm{invar}, \mathsf{t}_1.\mathrm{id}) \\
\quad * [{}^1\!/\!_2]\mathbf{gc}(\mathsf{t}_2.\mathrm{id}, \mathsf{t}_2.\mathrm{iot}, \langle\rangle) \\
\quad * \mathsf{t}_3 = \mathrm{place}(\mathrm{right}(\mathsf{t}_1.\mathrm{iot}), \langle\rangle, \mathsf{t}_1, \mathrm{none}, \mathsf{t}_1.\mathrm{invar}, \mathsf{t}_1.\mathrm{id}) \\
\quad * [{}^1\!/\!_2]\mathbf{gc}(\mathsf{t}_3.\mathrm{id}, \mathsf{t}_3.\mathrm{iot}, \langle\rangle);
\end{array}
$$

$$
\begin{array}{l}
\textbf{predicate } \mathit{join\_io} = \lambda \mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3. \\
\quad \mathsf{t}_3 = \mathrm{place}(\mathrm{join}(\mathsf{t}_1.\mathrm{iot}, \mathsf{t}_2.\mathrm{iot}), \langle\rangle, \mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_1.\mathrm{invar}, \mathsf{t}_1.\mathrm{id}) \\
\quad * \mathsf{t}_1.\mathrm{id} = \mathsf{t}_2.\mathrm{id} \land \mathsf{t}_1.\mathrm{invar} = \mathsf{t}_2.\mathrm{invar} \\
\quad * [{}^1\!/\!_2]\mathbf{gc}(\mathsf{t}_3.\mathrm{id}, \mathsf{t}_3.\mathrm{iot}, \langle\rangle)
\end{array}
$$

Third, because putchar cannot otherwise update the resources covered by the I/O invariant, i.e. prove that the I/O invariant still holds after performing the I/O action, a proof that the I/O invariant still holds after the I/O action is shipped in the I/O action *putchar_io*. This is easier to understand with an example, which we will do in the next subsections. Because we want to illustrate reusability, the example is split up into multiple smaller examples, each one reusable, and each one using the previous one, if any. We define a reusable buffer (without I/O style specifications) (Sec. 4.4.1), then on top of that we define a reusable putchar (with I/O style specifications) (Sec. 4.4.2), on top of that a reusable print_hi (Sec. 4.4.3), and on top of that a main function with non-I/O style specification that calls print_hi with I/O style specification (Sec. 4.4.5).

4.4.1 Example: Reusable buffer without I/O

We define a simple buffer (in non-I/O style) so that we can build an I/O style *putchar* on top of it. To define the simple buffer, we define a predicate to represent the memory footprint of the buffer. We also define functions to create, write to, read from, and dispose (free) a buffer.

$$\textbf{predicate } \mathit{buf} = \lambda \mathsf{b}, \mathsf{v}.\, \mathsf{b} \mapsto \mathsf{v}$$

$$\mathit{create\_buf} = \lambda.\ \{\ \mathbf{emp}\ \}\, \mathbf{cons}(0)\, \{\ \mathit{buf}(\mathsf{res}, 0)\ \}$$

$$\mathit{buf\_write} = \lambda \mathsf{b}, \mathsf{v}.\ \{\ \mathit{buf}(\mathsf{b}, \_)\ \}\ [\mathsf{b}] := \mathsf{v}\ \{\ \mathit{buf}(\mathsf{b}, \mathsf{v})\ \}$$

$$\mathit{buf\_read} = \lambda \mathsf{b}.\ \{\ \mathit{buf}(\mathsf{b}, \mathsf{v})\ \}\ [\mathsf{b}]\ \{\ \mathit{buf}(\mathsf{b}, \mathsf{v}) * \mathsf{res} = \mathsf{v}\ \}$$

4.4.2 Example: Reusable *putchar*

Remember that the approach has an I/O invariant, which is defined by the user of *putchar*.

The implementation of *putchar* will have to update the resources covered by the I/O invariant, but it is not fixed what this I/O invariant is. In the old approach, *putchar*'s verification proved that the I/O invariant still holds after the I/O action by replacing the I/O invariant with its definition. This old approach will not work anymore since it's unknown what the definition of the I/O invariant is. The solution we use is to let the user of *putchar* provide a proof that this I/O action is allowed by the I/O invariant. This proof is "shipped" in the I/O action as a function value with a specification (see ❺ below). *putchar* then calls (❻) this function value. We call such function values *updater function values*, or simply *updater functions*.

The precondition of the updater function value (❺) contains:

- The I/O invariant, which typically contains a $1/2$ fraction of the ghost cells that track the progresses of the I/O threads, including the I/O thread of the place under consideration (t_1). The I/O invariant has the memory representation as an argument.
- *token_rec*(t_1), which contains the other $1/2$ fraction of the ghost cell that tracks the progress of the I/O thread of t_1. *token_rec*(t_1) also contains the knowledge that the content of this ghost cell equals the progress encoded in t_1.

The postcondition of the updater function value contains:

- Again the I/O invariant, but this time the value that is the memory representation after applying the I/O action is used as argument for the memory representation. For this I/O action that writes to a one-cell buffer, this is just the value written (c).
- *token_rec*(t_2).

So the updater function has access to the full (i.e. fraction size one) ghost cell that tracks the progress, and must update it from the progress encoded in t_1 to the progress encoded in t_2. The updater function must not only update the ghost cell but also show that the I/O invariant holds after doing so, for the new memory representation that is an argument of the I/O invariant.

The reason the precondition and postcondition of the updater function value uses *token_rec* instead of just passing the ghost cell ($([1/2]\mathbf{gc}(t_1.\text{id}, t_1.\text{iot}, t_1.\text{progress}))$ is because the updater function value might need knowledge of progresses of previous I/O threads.

```
$\textbf{predicate}\ \textit{putchar\_io} = \lambda \mathsf{t_1}, \mathsf{c}, \mathsf{t_2}, \mathsf{r}.$
  $\mathsf{t_2} = \mathsf{t_1}[\text{progress} := \mathsf{t_1}.\text{progress} +\!\!+ \langle \mathsf{c} \rangle]$
  $* \ \mathsf{r}.\text{``t1''} = \mathsf{t_1}$
  $* \ \{\ \mathsf{t_1}.\text{invar}(\mathsf{t_1}.\text{id}, \_) * \textit{token\_rec}(\mathsf{t_1})\ \}$
      $\mathsf{r}.\text{``update\_fun''}()$ ❺
    $\{\ \mathsf{t_2}.\text{invar}(\mathsf{t_2}.\text{id}, \mathsf{c}) * \textit{token\_rec}(\mathsf{t_2})\ \}$

$\textit{putchar} = \lambda \mathsf{b}, \mathsf{c}\ ; \mathsf{r}\ .$
   $\{\ \textit{token}(\mathsf{b}, ?t_1) * \textit{putchar\_io}(t_1, \mathsf{c}, ?t_2, \mathsf{r})\ \}$
   $\left\{ \begin{array}{l} \textit{token\_rec}(t_1) * [t_1.\text{iot.f}] \boxed{\textit{buffer\_invar}(t_1.\text{id}, t_1.\text{invar}, \mathsf{b})} \\ *\ \textit{putchar\_io}(t_1, \mathsf{c}, t_2, \mathsf{r}) \end{array} \right\}$
   $\langle$
     $\left\{ \begin{array}{l} \textit{token\_rec}(t_1) * \ \textit{buf}(\mathsf{b}, ?v) * t_1.\text{invar}(t_1.\text{id}, v) \\ * \ \mathsf{r}.\text{``t1''} = t_1 * t_2 = t_1[\text{progress} := t_1.\text{progress} +\!\!+ \langle \mathsf{c} \rangle] \\ * \ \{\ t_1.\text{invar}(t_1.\text{id}, \_) * \textit{token\_rec}(t_1)\ \} \\ \quad \mathsf{r}.\text{``update\_fun''}() \\ \quad \{\ t_2.\text{invar}(t_2.\text{id}, \mathsf{c}) * \textit{token\_rec}(t_2)\ \} \end{array} \right\}$
     $\textit{buf\_write}(\mathsf{b}, \mathsf{c});$
     $\textbf{let}\ \ \_^{\text{g}}\ := \mathsf{r}.\text{``update\_fun''}()\ \textbf{in}\ \text{unit}$ ❻
     $\{\ \textit{token\_rec}(t_2) * \textit{buf}(\mathsf{b}, \mathsf{c}) * t_2.\text{invar}(t_2.\text{id}, \mathsf{c})\ \}$
   $\rangle$
   $\left\{\ \textit{token\_rec}(t_2) * [t_2.\text{iot.f}] \boxed{\textit{buffer\_invar}(t_2.\text{id}, t_2.\text{invar}, \mathsf{b})}\ \right\}$
   $\{\ \textit{token}(\mathsf{b}, t_2)\ \}$
```

4.4.3 Example: Reusable *print_hi* with I/O style specifications

The following functions can simply be verified by using the specifications of the functions they call. They do not have to take into account whether the callees are implemented in-memory or not.

```
$print\_h = \lambda \mathsf{b}\ ; \mathsf{r}\ .$
    $\{\ token(\mathsf{b}, t_1) * putchar\_io(t_1, \text{‘h’}, t_2, \mathsf{r})\ \}$
    $putchar(\mathsf{b}, \text{‘h’}\ ; \mathsf{r}\ )$
    $\{\ token(\mathsf{b}, t_2)\ \}$

$print\_i = \lambda \mathsf{b}\ ; \mathsf{r}\ .$
    $\{\ token(\mathsf{b}, t_1) * putchar\_io(t_1, \text{‘i’}, t_2, \mathsf{r})\ \}$
    $putchar(\mathsf{b}, \text{‘i’}\ ; \mathsf{r}\ )$
    $\{\ token(\mathsf{b}, t_2)\ \}$

$\textbf{predicate}\ print\_hi\_io = \lambda \mathsf{t}_1, \mathsf{t}_2, \mathsf{r}.$
$\exists \mathsf{t}_{\mathsf{h}1}, \mathsf{t}_{\mathsf{i}1}, \mathsf{t}_{\mathsf{h}2}, \mathsf{t}_{\mathsf{i}2}.$
    $split\_io(\mathsf{t}_1, \mathsf{t}_{\mathsf{h}1}, \mathsf{t}_{\mathsf{i}1})$
    $* putchar\_io(\mathsf{t}_{\mathsf{h}1}, \text{‘h’}, \mathsf{t}_{\mathsf{h}2}, \mathsf{r}.\text{“h”})$
    $* putchar\_io(\mathsf{t}_{\mathsf{i}1}, \text{‘i’}, \mathsf{t}_{\mathsf{i}2}, \mathsf{r}.\text{“i”})$
    $* join\_io(\mathsf{t}_{\mathsf{h}2}, \mathsf{t}_{\mathsf{i}2}, \mathsf{t}_2)$

$print\_hi = \lambda \mathsf{b}\ ; \mathsf{r}\ .$
    $\{\ token(\mathsf{b}, t_1) * print\_hi\_io(t_1, t_2, \mathsf{r})\ \}$
    $\textbf{let}\ \_^{\mathrm{g}} := split()\ \textbf{in}$
    $(\ print\_h(\mathsf{b}\ ; \mathsf{r}.\text{“h”}\ )\ ||\ print\_i(\mathsf{b}\ ; \mathsf{r}.\text{“i”}\ )\ );$
    $\textbf{let}\ \_^{\mathrm{g}} := join()\ \textbf{in}\ \text{unit}$
    $\{\ token(\mathsf{b}, t_2)\ \}$
```

4.4.4 Example: *main* specification without I/O style spec

Consider the following specification of the *main* function.

$\{\ \textbf{emp}\ \}$

$main()$
$\{\ \mathsf{res} = \text{‘h’} \vee \mathsf{res} = \text{‘i’}\ \}$

In the next section, we show how one can verify an implementation for this specification that calls the reusable function with I/O style specifications.

4.4.5 Example: *main* implementation calling I/O style specified functions

Multiple implementations for the *main* function are possible. In this section we focus on the following one:

```
$main = \lambda.$
  $\{\ \mathbf{emp}\ \}$
  $\mathbf{let}\ \mathsf{b} := create\_buf()\ \mathbf{in}$
  $print\_hi(\mathsf{b});$
  $buf\_read(\mathsf{b})$
  $\{\ \mathsf{res} = \text{‘h’} \vee \mathsf{res} = \text{‘i’}\ \}$
```

The main difference from the implementation in the non-reusable setting is that *main* now calls the reusable version of *print_hi*.

When verifying the *main* function, we have an extra responsibility: we have to define the I/O invariant. Furthermore, we have to define the updater functions (❼ and ❽) that provide the proof that the I/O actions preserve the I/O invariant.

In this example, the I/O invariant states the progress of the left I/O thread is $\langle \text{‘h’} \rangle$ or the empty list, and the progress of the right I/O thread is $\langle \text{‘i’} \rangle$ or the empty list. In other words, one I/O thread writes $\langle \text{‘h’} \rangle$ and another writes $\langle \text{‘i’} \rangle$. The content of the memory buffer is a last written character (by one of the two I/O threads), or unconstrained if both I/O threads have not written anything yet.

```
$\mathbf{predicate}\ io\_invar = \lambda \mathsf{id}, \mathsf{v}.$
$\exists\, \mathsf{l}, \mathsf{r}, \mathsf{l\_todo}, \mathsf{r\_todo}.$
  $[1/2]\mathbf{gc}(\mathsf{id}, \mathrm{left}(\mathrm{init}), \mathsf{l})$
  $* \ [1/2]\mathbf{gc}(\mathsf{id}, \mathrm{right}(\mathrm{init}), \mathsf{r})$
  $* \ \mathsf{l} +\!+ \mathsf{l\_todo} = \langle \text{‘h’} \rangle$
  $* \ \mathsf{r} +\!+ \mathsf{r\_todo} = \langle \text{‘i’} \rangle$
  $* \textbf{if } \mathsf{l} \neq \langle\rangle \wedge \mathsf{r} \neq \langle\rangle \textbf{ then}$
      $\mathsf{v} = \mathrm{last}(\mathsf{l}) \vee \mathsf{v} = \mathrm{last}(\mathsf{r})$
    $\textbf{else if } \mathsf{l} \neq \langle\rangle \textbf{ then}$
      $\mathsf{v} = \mathrm{last}(\mathsf{r})$
    $\textbf{else if } \mathsf{r} \neq \langle\rangle \textbf{ then}$
      $\mathsf{v} = \mathrm{last}(\mathsf{l})$

$main = \lambda.$
  $\{\ \textbf{emp}\ \}$
  $\textbf{let } \mathsf{b} := \mathsf{create\_buf}() \textbf{ in}$
  $\textbf{let } \mathsf{id} := \textbf{gcf\_init}() \textbf{ in}$
  $\textbf{let } \mathsf{iot1} := \mathrm{init} \textbf{ in}$
  $\textbf{let } \mathsf{ioth} := \mathrm{left}(\mathsf{iot1}) \textbf{ in}$
  $\textbf{let } \mathsf{ioti} := \mathrm{right}(\mathsf{iot1}) \textbf{ in}$
  $\textbf{let } \mathsf{iot2} := \mathrm{join}(\mathsf{ioth}, \mathsf{ioti}) \textbf{ in}$
  $\textbf{let } \mathsf{t1} := \mathrm{place}(\mathsf{iot1}, \langle\rangle, \mathrm{none}, \mathrm{none}, \mathit{io\_invar}, \mathsf{id}) \textbf{ in}$
  $\textbf{let } \mathsf{th1} := \mathrm{place}(\mathsf{ioth}, \langle\rangle, \mathsf{t1}, \mathrm{none}, \mathit{io\_invar}, \mathsf{id}) \textbf{ in}$
  $\textbf{let } \mathsf{th2} := \mathrm{place}(\mathsf{ioth}, \langle\text{‘h’}\rangle, \mathsf{t1}, \mathrm{none}, \mathit{io\_invar}, \mathrm{id}) \textbf{ in}$
  $\textbf{let } \mathsf{ti1} := \mathrm{place}(\mathsf{ioti}, \langle\rangle, \mathsf{t1}, \mathrm{none}, \mathit{io\_invar}, \mathrm{id}) \textbf{ in}$
  $\textbf{let } \mathsf{ti2} := \mathrm{place}(\mathsf{ioti}, \langle\text{‘i’}\rangle, \mathsf{t1}, \mathrm{none}, \mathit{io\_invar}, \mathsf{id}) \textbf{ in}$
  $\textbf{let } \mathsf{t2} := \mathrm{place}(\mathsf{iot2}, \langle\rangle, \mathsf{th2}, \mathsf{ti2}, \mathit{io\_invar}, \mathsf{id}) \textbf{ in}$

  $\textbf{let } \_ := \textbf{gcf\_cons}(\mathsf{id}, \mathsf{iot1}, \langle\rangle) \textbf{ in}$
  $\textbf{let } \_ := \textbf{gcf\_cons}(\mathsf{id}, \mathsf{ioth}, \langle\rangle) \textbf{ in}$
  $\textbf{let } \_ := \textbf{gcf\_cons}(\mathsf{id}, \mathsf{ioti}, \langle\rangle) \textbf{ in}$
  $\textbf{let } \_ := \textbf{gcf\_cons}(\mathsf{id}, \mathsf{iot2}, \langle\rangle) \textbf{ in}$
  $\textbf{let } \_ := \textbf{atintro in}$

  $\textbf{let } \mathsf{update\_h} := \lambda.$ ❼
    $\{\ \mathsf{th1}.\mathrm{invar}(\mathsf{id}, \_) * \mathit{token\_rec}(\mathsf{th1})\ \}$
    $(\mathsf{id}, \mathrm{left}(\mathrm{init})) := \langle\text{‘h’}\rangle$
    $\{\ \mathsf{th2}.\mathrm{invar}(\mathsf{id}, \text{‘h’}) * \mathit{token\_rec}(\mathsf{th2})\ \}$
  $\textbf{in}$
  $\textbf{let } \mathsf{update\_i} := \lambda.$ ❽
    $\{\ \mathsf{ti1}.\mathrm{invar}(\mathsf{id}, \_) * \mathit{token\_rec}(\mathsf{ti1})\ \}$
    $(\mathsf{id}, \mathrm{right}(\mathrm{init})) := \langle \text{‘i’} \rangle$
    $\{\ \mathsf{ti2}.\mathrm{invar}(\mathsf{id}, \text{‘i’}) * token\_rec(\mathsf{ti2})\ \}$
  $\textbf{in}$
  $\textbf{let}\ \mathsf{r} := \{\ \text{“h”} : \{\text{“update\_fun”} : \mathsf{update\_h}, \text{“t1”} : \mathsf{th1}\},$
                 $\text{“i”} : \{\text{“update\_fun”} : \mathsf{update\_i}, \text{“t1”} : \mathsf{ti1}\}\}\ \textbf{in}$
  $\{\ token(\mathsf{b}, \mathsf{t1}) * print\_hi\_io(\mathsf{t1}, \mathsf{t2}, \mathsf{r})\ \}$
  $print\_hi(\mathsf{b}\ ; \mathsf{r}\ )$
  $\{\ token(\mathsf{b}, \mathsf{t2})\ \}$
  $\left\{\ \boxed{buffer\_invar(\mathsf{id}, io\_invar, \mathsf{b})} * token\_rec(\mathsf{t2})\ \right\}$
  $\textbf{let}\ \_ := \textbf{atdel in}$
  $\{\ buf(\mathsf{b}, c) * (c = \text{‘h’} \lor c = \text{‘i’})\ \}$
  $buf\_read(\mathsf{b})$
  $\{\ \mathsf{res} = \text{‘h’} \lor \mathsf{res} = \text{‘i’}\ \}$
```

4.4.6 Formalization: Higher order functions

In this section we formalize higher order functions. Higher order functions (such as *putchar* in Sec. 4.4.2) are functions that accept another function (such as an updater function) as argument. To support this, a value can be a function such that it can be passed to another function.

The use of higher order functions for I/O style verification is explained in Sec. 4.4.2.

Consider the following example:

```
$\textbf{let}\ \mathsf{f} := \lambda \mathsf{g}.$
  $\mathsf{g}(1)$
$\textbf{in}\ \mathrm{unit}$
```

It defines a function f. f has a parameter g. The body of f calls g with argument 1.

To verify this, we provide a specification for g in the precondition of f.

$$\dfrac{\forall \overline{v_1}, \overline{v_2}, \overline{v_3}.\ \vdash \left\{\begin{matrix} P[\overline{v_1}/\overline{x}] \\ [\overline{v_2}/\overline{y}] \\ [\overline{v_3}/\overline{z}] \end{matrix}\right\} \begin{matrix} c[\overline{v_1}/\overline{x}] \\ [\overline{v_2}/\overline{y}] \end{matrix} \left\{\begin{matrix} Q[\overline{v_1}/\overline{x}] \\ [\overline{v_2}/\overline{y}] \\ [\overline{v_3}/\overline{z}] \end{matrix}\right\}_I^l \qquad \overline{z} = \mathrm{fv}(P * Q) \setminus (\overline{x^{\mathrm{r}}} \cup \overline{y^{\mathrm{g}}})}{\vdash \left\{\mathbf{emp}\right\} \lambda \overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}}.\, c \left\{\mathsf{res} = \lambda \overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}}.\, c * (\left\{P\right\} (\lambda \overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}}.\, c)(\overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}}) \left\{Q\right\}_I^l)\right\}_I^l} \text{ FVI}$$

$$\dfrac{\overline{z} = \mathrm{fv}(P * Q) \setminus (\overline{x^{\mathrm{r}}} \cup \overline{y^{\mathrm{g}}})}{\vdash \left\{\begin{matrix} P[\overline{v_1}/\overline{x}][\overline{v_2}/\overline{y}][\overline{v_3}/\overline{z}] \\ * (\left\{P\right\} v(\overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}}) \left\{Q\right\}_I^l) \end{matrix}\right\} v(\overline{v_1}; \overline{v_2}) \left\{\begin{matrix} Q[\overline{v_1}/\overline{x}][\overline{v_2}/\overline{y}][\overline{v_3}/\overline{z}] \\ * (\left\{P\right\} v(\overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}}) \left\{Q\right\}_I^l) \end{matrix}\right\}_I^l} \text{ FVA}$$

Figure 4.11: Proof rules for higher order functions

```
let f := λg.
    { { emp } g(x) { res > x } }
    g(1)
    { res > 1 }
in unit
```

The function call to g is then verified using this specification. The FVA proof rule for function value application in Fig. 4.11 does exactly this: if the precondition of a function value holds, then after calling the function value the postcondition of the function value holds.

Definition 63: Proof rules extended for higher order functions

Fig. 4.11 extends the proof rules for higher order functions.

The caller of f can pass any function value as argument, as long as the caller can prove that argument satisfies the specification.

We extend the assertion language with an assertion that states a function value conforms to a certain specification:

Definition 64: Assertions extended for higher order functions

$$P ::= \ldots \mid \{P\}\, v(\overline{x}; \overline{y})\, \{Q\}_I^l$$

We want such a Hoare triple assertion to hold if the Hoare triple is valid. Informally, a Hoare triple is valid, written $\models \{P\}\, c\, \{Q\}$, if for every heap h that is a model of P, any execution of c "does not go wrong" and, if it terminates, ends with a heap that satisfies Q.

What does it mean for an execution to not go wrong? Consider a heap h that satisfies the precondition P and an execution of c that starts from h and yields a trace τ and result value v. Whether this execution goes wrong or not, depends largely on the trace τ.

Examples:

- $h = \{\!\!\{\}\!\!\}$, $\tau = (\{\!\!\{\}\!\!\}, \bot) \cdot \epsilon$. The execution goes wrong because the trace expresses (using $\bot$) that the execution crashes.
- $h = \{\!\!\{\}\!\!\}$, $\tau = (\{\!\!\{\}\!\!\}, \mathrm{inf}) \cdot \epsilon$. The execution is fine (does not goes wrong) because it immediately goes into an infinite loop. This infinite loop is caused by an atomic block that goes into an infinite loop (and does not crash because then the trace would be $(\{\!\!\{\}\!\!\}, \bot) \cdot \ldots$).
- $h = \{\!\!\{\boxed{1 \mapsto 3}\}\!\!\}$, $\tau = (\{\!\!\{1 \mapsto 3\}\!\!\}, \{\!\!\{1 \mapsto 3\}\!\!\}) \cdot \epsilon$. The execution does not go wrong.
- $h = \{\!\!\{\}\!\!\}$, $\tau = (\{\!\!\{1 \mapsto 3\}\!\!\}, \{\!\!\{1 \mapsto 3\}\!\!\}) \cdot \epsilon$. The execution does not go wrong.
- $h = \{\!\!\{\boxed{1 \mapsto 3}\}\!\!\}$, $\tau = (\{\!\!\{1 \mapsto 3\}\!\!\}, \{\!\!\{1 \mapsto 4\}\!\!\}) \cdot \ldots$. The execution goes wrong because it does not preserve the atomic space given in the start heap $\{\!\!\{\boxed{1 \mapsto 3}\}\!\!\}$.
- $h = \{\!\!\{2 \mapsto 3\}\!\!\}$, $\tau = (\{\!\!\{2 \mapsto 3, 3 \mapsto 4\}\!\!\}, \{\!\!\{2 \mapsto 3, 3 \mapsto 5\}\!\!\}) \cdot \epsilon$. The execution goes wrong. $3 \mapsto 4$ is not owned by the start heap, only $2 \mapsto 3$ is. $3 \mapsto 4$ can be owned by another thread. Therefore, the thread under consideration should not modify it.
- $h = \{\!\!\{\boxed{\exists \mathsf{x}. 1 \mapsto \mathsf{x} * \mathsf{x} > 1}, 2 \mapsto 3\}\!\!\}$, $\tau = (\{\!\!\{1 \mapsto 2, 2 \mapsto 3, 7 \mapsto 8\}\!\!\}, \{\!\!\{1 \mapsto 3, 2 \mapsto 4, 7 \mapsto 8\}\!\!\}) \cdot (\{\!\!\{1 \mapsto 3, 2 \mapsto 4, 7 \mapsto 10\}\!\!\}, \{\!\!\{1 \mapsto 3, 2 \mapsto 5, 7 \mapsto 8\}\!\!\}) \cdot \epsilon$. The execution does not go wrong.

More general, consider the situation $h, c \Downarrow (h_1, h_2) \cdot \tau', v$. The heaps in the trace $(h_1, h_2) \cdot \tau'$ are from the point of view of execution (not verification), and do not contain atomic space chunks (such as $\boxed{P}$). This is consistent with the step semantics: consider for example the AtomIntro step rule in Fig. 4.7 p. 114. It does not actually add an atomic space chunk to the heap; it does not modify the heap. The heaps in this trace can, however, contain ghost cells.

Contrary to the heaps in the trace, the heap h is from the point of view of verification and represents the part of the heap that is owned by the thread that yields the trace $(h_1, h_2) \cdot \tau'$. h can contain atomic space chunks and fractional chunks.

We split the heap h_1 into three parts:

- The part that is h, but without the atomic space chunks
- The part that is covered by the atomic space chunks
- The part that is owned by other threads

If the execution does not go wrong, we expect h_2 to consist of these three parts:

- The part that "evolved" from h (but without the atomic space chunks). This contains the modifications of the heap performed by the thread.
- The part that is covered by the atomic space chunks, which might now be different as long as it satisfies the assertion of the atomic space chunk.
- The part that is owned by other threads, which must stay the same.

Now we define this more formally. We will write definitions that do not look at complete executions, but only at a first number of steps. These steps are split up in two parts: the part of the steps outside an atomic block, and the part inside. So when we say we only consider (m, n) steps of an execution, it means we only look at the first m steps outside an atomic block, and only at the first n steps inside an atomic block.

Definition 65: Steps

We range over steps with Λ: Λ ranges over $\mathbb{N} \times \mathbb{N}$.

We define an ordering:

Definition 66: Ordering on steps

$$(m, n) < (m', n') \iff m < m' \vee (m = m' \wedge n < n')$$

We define two addition operations:

Definition 67: $\Lambda +_{\text{outat}} k$

$(m, n) +_{\text{outat}} k = (m + k, n)$

Definition 68: $\Lambda +_{\text{inat}} k$

$(m, n) +_{\text{inat}} k = (m, n + k)$

Here, k, m, n range over $\mathbb{N}$

Subtraction is similar, but is not a total operation ($(0, 0) -_{\text{outat}} 1$ is undefined).

Definition 69: $\Lambda -_{\text{outat}} k$

$(m + k, n) -_{\text{outat}} k = (m, n)$

Definition 70: $\Lambda -_{\text{inat}} k$

$(m, n + k) -_{\text{inat}} k = (m, n)$

Operations to retrieve fields:

Definition 71: $\Lambda.\text{outat}$ **operation**

$(m, n).\text{outat} = m$

Definition 72: $\Lambda.\text{inat}$ **operation**

$(m, n).\text{inat} = n$

erat returns the heap obtained by removing all atomic space chunks:

Definition 73: $\mathrm{erat}(h)$

$$(\mathrm{erat}(h))(C) = \textbf{ if } \exists P.\ C = \boxed{P} \textbf{ then } 0 \textbf{ else } h(C)$$

inv returns the multiset of assertions of atomic space chunks:

Definition 74: inv

$$\begin{aligned}\mathrm{inv}(h, I) = \lambda P.\ \textbf{if } I = \mathrm{inat} \textbf{ then } 0 \\ \textbf{else } \lceil h(\boxed{P}) \rceil\end{aligned}$$

For example, $\mathrm{inv}(\{\!\!\{\boxed{\exists \mathsf{x}.\ 1 \mapsto x}, \boxed{\exists \mathsf{x}.\ 1 \mapsto x}, 2 \mapsto 3\}\!\!\}, \mathrm{outat}) = \{\!\!\{\exists \mathsf{x}.\ 1 \mapsto x, \exists \mathsf{x}.\ 1 \mapsto x\}\!\!\}$.

Validity of a Hoare triple is defined as follows:

Definition 75: Validity of a Hoare triple ($\models \{P\}\, c\, \{Q\}_I$)

$$\begin{aligned}&\models^{\Lambda} \{P\}\, c\, \{Q\}_I \iff \\ &\quad \forall h.\ h \models_{F_\Lambda} P \Rightarrow \\ &\quad \forall \tau, v.\ c \Downarrow \tau, v \Rightarrow \\ &\quad \mathrm{safe}(\Lambda, h, \tau, I, Q[v/\mathsf{res}])\end{aligned}$$

$$\models \{P\}\, c\, \{Q\}_I \iff \forall \Lambda.\ \models^{\Lambda} \{P\}\, c\, \{Q\}_I$$

$\mathrm{safe}(\Lambda, h, \tau, I, Q)$ defines what it means for an execution to not go wrong, when the execution has trace τ, starts from heap h, is in an atomic block if $I = \mathrm{inat}$ and outside an atomic block if $I = \mathrm{outat}$ and has the desired postcondition Q.

Definition 76: safe

$$
\begin{array}{l}
\text{safe}'((0, \_), h, \tau, \text{outat}, Q) = \text{true} \\
\text{safe}'((\_, 0), h, \tau, \text{inat}, Q) = \text{true} \\
\text{safe}'(\Lambda +_I 1, h, (h_1, \textbf{inf}) \cdot \tau, I, Q) = \text{true} \\
\text{safe}'(\Lambda +_I 1, h, (h_1, \bot) \cdot \tau, I, Q) = \\
\quad \forall h_r, h_I.\ h_1 = \text{erat}(h \uplus h_I \uplus h_r) \wedge h_I \vDash_{F_{\Lambda +_I 1}} \circledast \text{inv}(h \uplus h_I \uplus h_r, I) \Rightarrow \text{false} \\
\text{safe}'(\Lambda +_I 1, h, (h_1, h_2) \cdot \tau, I, Q) = \\
\quad \forall h_r, h_I.\ h_1 = \text{erat}(h \uplus h_I \uplus h_r) \wedge h_I \vDash_{F_{\Lambda +_I 1}} \circledast \text{inv}(h \uplus h_I \uplus h_r, I) \Rightarrow \\
\quad \exists h', h'_I.\ h_2 = \text{erat}(h' \uplus h'_I \uplus h_r) \wedge h'_I \vDash_{F_\Lambda} \circledast \text{inv}(h' \uplus h'_I \uplus h_r, I) \\
\quad \wedge\ \text{safe}(\Lambda, h', \tau, I, Q) \\
\quad \wedge\ \big(I = \text{inat} \Rightarrow \text{inv}(h \uplus h_I \uplus h_r, \text{outat}) = \text{inv}(h' \uplus h'_I \uplus h_r, \text{outat})\big) \\
\text{safe}'(\Lambda +_I 1, h, \epsilon, I, Q) = \\
\quad \exists h_Q, h_r.\ h = h_Q \uplus h_r \wedge \\
\quad h_Q \vDash_{F_{\Lambda +_I 1}} Q \\
\text{safe}(\Lambda, h, \tau, I, Q) = \\
\quad \forall \Lambda' \leq \Lambda.\ \text{safe}'(\Lambda', h, \tau, I, Q)
\end{array}
$$

safe is used both for traces that occur inside an atomic block (and hence such traces will be squashed into one pair), and traces that occur outside an atomic block (and might therefore contain such a pair of heaps that is the result of squashing a trace). The one but last argument expresses whether we are in an atomic block or not.

In an atomic block, it is disallowed to create new atomic space chunks, as enforced by

$$I = \text{inat} \Rightarrow \text{inv}(h \uplus h_I \uplus h_r, \text{outat}) = \text{inv}(h' \uplus h'_I \uplus h_r, \text{outat})$$

When inside the context of an atomic block, the assertions of the atomic spaces can be violated. So when we said the heaps in the trace have three parts where one part represents the chunks covered by the atomic space, this part does not exist (or is empty). Therefore, the function inv has a second parameter representing whether it is used in the context of an atomic block, and returns the empty set in that case.

Definition 77: Sat. rel. of asn extended for higher order functions

The satisfaction relation of assertions is extended by the inference rule of Fig. 4.12 on the following page.

$$\frac{\{P\}\, v(\overline{x};\overline{y})\, \{Q\}_I^l \in F}{\{\} \vDash_F [\pi]\, \{P\}\, v(\overline{x};\overline{y})\, \{Q\}_I^l}\ \text{Fun}$$

Figure 4.12: Satisfaction relation of assertions for higher order functions (extends Fig. 4.5 on page 110)

It basically says that the assertion expressing that a certain Hoare triple holds, is true if the Hoare triple is in a given set F. This given set is supposed to express the set of Hoare triples that hold.

For this definition to be useful, we need to define such a set.

We define such a set - or rather such sets - as follows:

Definition 78: F_Λ

$$\begin{aligned} F_\Lambda = \Big\{ \{P\}\, (\lambda \overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}}.\, c)(\overline{x};\overline{y})\, \{Q\}_I^l \mid \\ \forall \overline{v_1}, \overline{v_2}, \overline{v_3}.\ \forall \overline{z}.\ \overline{z} = \mathrm{fv}(P) \cup \mathrm{fv}(Q) \setminus (\overline{x} \cup \overline{y}) \Rightarrow \\ \forall \Lambda'.\ \Lambda' < \Lambda \Rightarrow \\ \vDash^{\Lambda'} \{P[\overline{v_1}/\overline{x}][\overline{v_2}/\overline{y}][\overline{v_3}/\overline{z}]\}\, c[\overline{v_1}/\overline{x}][\overline{v_2}/\overline{y}]\, \{Q[\overline{v_1}/\overline{x}][\overline{v_2}/\overline{y}][\overline{v_3}/\overline{z}]\}_I \Big\} \end{aligned}$$

It immediately follows from this definition that $F_{(0,0)}$ is the set of all Hoare triples, regardless of whether they are valid or not. Any syntactically correct Hoare triple is in this set, including $\{\mathbf{emp}\}$ true $\{\text{false}\}$.

Note that the definition of F_Λ only talks about $\Lambda' < \Lambda$ (not $\Lambda' \leq \Lambda$). Thanks to this, we do not end up with a circular definition.

4.5 Reading

So far, we only wrote functions that modify the shared state, such as *putchar*, but we did not write functions that inspect it. When inspecting or reading the shared state, the result depends on how the shared state has been modified before, maybe by other threads.

4.5.1 Prophecies

We deal with reading by using *prophecy chunks*. More precisely, the approach is crafted to be able to use prophecy chunks for reading. First we explain prophecies by example by showing how they can be used.

A prophecy chunk can be created, but only in ghost code, like so:

$\{\ \mathbf{emp}\ \}$

$\mathbf{let}\ \ \mathsf{id}^{\mathrm{g}}\ := \mathbf{pc}()\ \mathbf{in}$ unit

$\left\{\ \exists v.\ \mathbf{pr}(\ \mathsf{id}^{\mathrm{g}}\ , v)\ \right\}$

Creating a prophecy ($\mathbf{pc}()$) returns an identifier *id*, and puts a chunk $\mathbf{pr}(id, v)$ on the heap that links this identifier to a value v. The value is called the *prophetic value*. This heap chunk is a special kind of chunk: it is not a regular points-to chunk (not $\_ \mapsto \_$) and it is not accessible memory. At this point, nothing is known about the prophetic value: it might be greater than 42, or maybe not. The prophetic value is not accessible in real code and (in our programming language) also not in ghost code: it only exists in annotations and in that heap chunk.

Besides using prophecy chunks in annotations, one can use the prophecy assignment command (**pa**) to assign a value to the prophetic value:

$\{\ \mathbf{emp}\ \}$

$\mathbf{let}\ \ \mathsf{x}^{\mathrm{g}}\ := \mathbf{pc}()\ \mathbf{in}$

$\{\ \exists v.\ \mathbf{pr}(\ \mathsf{x}^{\mathrm{g}}\ , v)\ \}$

$\{\ \mathbf{pr}(\ \mathsf{x}^{\mathrm{g}}\ , ?v)\ \}$

$\mathbf{let}\ \ \_^{\mathrm{g}}\ := \mathbf{pa}(\ \mathsf{x}^{\mathrm{g}}\ , 123)\ \mathbf{in}$ unit

$\{\ v = 123\ \}$

After assigning a value to the prophetic value, the prophetic value equals the value. Also, the prophecy chunk is lost to prevent assigning a different value to the prophetic value later, which would be unsound. Note that you cannot assign the prophetic value to itself, or assign a prophetic value v plus one to itself. That would be unsound as well since it would yield $v + 1 = v$. This is prevented because the prophetic value cannot be used in code (both real and ghost), only in annotations.

To summarize, a prophetic value is a value that you can refer to in annotations before you know what actual value it will be. You can then assign to the prophetic value later. Prophecy chunks exist to facilitate the use of prophetic values.

4.5.2 Recursive functions

We want to write a function with an I/O style contract that performs reading. We implement this function as a recursive function.

For recursive functions, we use a **let rec** syntax, like the example below:

```
let rec f := λx ; y .
  f(x + 1 ; y + 1 ) in
f(7 ; 8 )
```

This program defines a function f whose body calls itself. Then, the program calls f. Executing this program causes an infinite loop.

4.5.3 Example: reading

We define a peekchar function that reads from the buffer of one integer. peekchar is blocking, i.e. it will wait until data is actually written into the buffer before returning a read value.

```
let rec peekchar := λb ; r .
  let read :=
  ⟨
    let read′ := buf_read(b) in
    read′
  ⟩ in
  if read ≠ 0 then
    read
  else
    peekchar(b ; r )
```

The implementation of peekchar works as follows. The peekchar function enters an atomic block, in which it copies the integer from the buffer to a local variable read′. After the atomic block the read value is returned, or peekchar is called recursively in case no value (i.e. the value zero) has been read.

The definition of the *peekchar_io* action is as follows.

```
predicate $peekchar\_io = \lambda t_1, c, t_2, r.$
  $t_2 = t_1[\text{progress} := t_1.\text{progress} +\!+ \langle c \rangle]$
  $r.\text{“t1”} = t_1$
  $* \ \mathbf{pr}(r.\text{“prid”}, c)$ ❶
  $* \ \{ \ t_1.\text{invar}(id, c) * token\_rec(t_1) * c \neq 0 * c = x \ \}$
    $r.\text{“update\_fun”}( \ ; x \ )$
    $\{ \ t_2.\text{invar}(id, c) * token\_rec(t_2) \ \}$
```

The argument c of this I/O action is the value that will be read. We do not know in advance what this value will be. The definition of the I/O action contains an assertion ❶ that states that a prophecy chunk exists for this value.

The verification of peekchar uses this prophecy as follows (see code below). Inside the atomic block, after reading a value from the buffer, we insert ghost code. The ghost code checks whether the integer read is not zero, i.e. the buffer is not empty. If the buffer is not empty, the ghost code assigns the integer that is read to the prophetic value ❷. Then, the ghost code calls the updater function. The updater function updates the progress that tracks the characters read. The postcondition of the updater function guarantees that the I/O invariant still holds. The I/O invariant typically contains a 1/2 fraction of the ghost cell that contains this progress.

```
let rec peekchar := $\lambda b \ ; r \ .$
  $\{ \ token(b, ?t_1) * peekchar\_io(t_1, ?c, ?t_2, r) \ \}$
  let read :=
  $\langle$
    $\left\{ \begin{array}{l} buf(b, ?v) \\ * \ token\_rec(t_1) \\ * \ t_1.\text{invar}(id, v) \\ * \ peekchar\_io(t_1, c, t_2, r) \end{array} \right\}$
    let read′ := $buf\_read(b)$ in
    let $\_^{g}$ :=
        if read' ≠ 0 then
          pa(r."prid", read'); ❷
          $\left\{\begin{array}{l} \mathit{buf}(\mathsf{b}, \mathsf{read}') \\ * \ \mathit{token\_rec}(t_1) \\ * \ t_1.\mathrm{invar}(\mathit{id}, \mathsf{read}' \\ * \ \{\ t_1.\mathrm{invar}(\mathit{id}, c) * \mathit{token\_rec}(t_1) * \ c \neq 0 * c = x \ \} \\ \quad \mathsf{r}.\text{“update\_fun”}(\ ; x\ ) \\ \quad \{\ t_2.\mathrm{invar}(\mathit{id}, c) * \mathit{token\_rec}(t_2)\ \} \\ * \ c = \mathsf{read}' \end{array}\right\}$
          r."update_fun"( ; read' )
        else unit
      in
      read'
    ⟩ in
    $\left\{\begin{array}{l} [t_1.\mathrm{iot.f}]\boxed{\mathit{buffer\_invar}(t_1.\mathrm{id}, t_1.\mathrm{invar}, \mathsf{b})} \\ * \ \textbf{if}\ \mathsf{read} = 0\ \textbf{then}\ \mathit{token\_rec}(t_1) * \mathit{peekchar\_io}(\mathsf{b}, t_1, c, t_2, \mathsf{r}) \\ \quad \textbf{else}\ \mathit{token\_rec}(t_2) * \mathsf{read} = c \end{array}\right\}$
    $\left\{\begin{array}{l} \textbf{if}\ \mathsf{read} = 0\ \textbf{then}\ \mathit{token}(\mathsf{b}, t_1) * \mathit{peekchar\_io}(\mathsf{b}, t_1, c, t_2, \mathsf{r}) \\ \qquad\qquad\qquad \textbf{else}\ \mathit{token}(\mathsf{b}, t_2) * \mathsf{read} = c \end{array}\right\}$
    if read ≠ 0 then
      read
    else
      peekchar(b ; r )
  $\{\ \mathit{token}(\mathsf{b}, t_2) * \mathsf{res} = c\ \}$
```

Contrary to *putchar*, peekchar does not modify the buffer. However, similar to *putchar*, peekchar can modify the progress. Remember that the updater function for *putchar* modifies the ghost cell that tracks the progress of the I/O thread, such that the progress reflects that the character is written. Similarly, the updater function for peekchar modifies the ghost cell that tracks the progress, to reflect that the character is read.

Just like we encode the characters written in a progress which is encoded in a place, we encode the characters read in a progress which is also encoded in a place. In the examples we use here, there is no I/O thread that both reads and writes, so it is fine to encode the bytes read and the bytes written both as just a list of integers. In case you want to mix reading and writing in one I/O thread, you will need another encoding than just a list of integers, such that one progress can track both characters read and written in the same I/O thread.

Tracking progress of characters read, allows the I/O invariant to constrain them. Let's build an example *readwrite* where one thread reads a character (using peekchar), concurrently with one thread that writes a character.

Depending on how *readwrite* is used, it can yield different results. *readwrite* might be used concurrently with another function, and the buffer might already contain some content before *readwrite* is called.

In the particular case where *readwrite* is not called concurrently with other code, and the buffer is initially empty, the character read will be equal to the character written.

If we want to prove, while verifying a caller of *readwrite*, that the character read is equal to the character written, we will have to come up with an I/O invariant that helps to prove this. For this particular example, the I/O invariant can track both the progress of the writing I/O thread and the progress of the reading I/O thread. The I/O invariant can then constrain that what is read is a prefix of what is written. This way, while verifying the caller of *readwrite*, we can, thanks to the I/O invariant, prove the desired property.

Note that coming up with an I/O invariant is necessary when verifying the caller of *readwrite*, not when verifying *readwrite* itself.

Let's turn this idea into code. The *readwrite* function is as follows:

```
$readwrite = \lambda \mathsf{b}\ ; \mathsf{r}\ .$
    $\left\{ \begin{array}{l} token(\mathsf{b}, t1) * split\_io(t1, tr1, tw1) \\ * putchar\_io(tw1, c, tw2, \mathsf{r}.\text{“print”}) \\ * peekchar\_io(tr1, c', tr2, \mathsf{r}.\text{“peek”}) \\ * join\_io(tr2, tw2, t2) \end{array} \right\}$
  $\textbf{let}\ \mathsf{x} := \textbf{cons}(0)\ \textbf{in}$
  $\textbf{let}\ \_^{\mathrm{g}} := split()\ \textbf{in}$
  $\big([\mathsf{x}] := \mathsf{peekchar}(\mathsf{b}\ ; \mathsf{r}.\text{“peek”})\ \ ||\ \ print\_i(\mathsf{b}\ ; \mathsf{r}.\text{“print”})\big);$
  $\textbf{let}\ \_^{\mathrm{g}} := join()\ \textbf{in}$
  $[\mathsf{x}]$
   $\{\ token(\mathsf{b}, t2) * \mathsf{res} = c'\ \}$
```

Verifying this function is easy since it has an I/O style specification and only calls I/O style functions.

The implementation of *main* simply creates the buffer and calls *readwrite* on an empty buffer. Because the buffer is empty, the character read will be equal to the character written.

```
$main = \lambda.$
  $\{\ \textbf{emp}\ \}$
  $\textbf{let}\ \mathsf{b} := \mathit{create\_buf}()\ \textbf{in}$
  $\textbf{let}\ \mathsf{c}' := \mathit{readwrite}(\mathsf{b})\ \textbf{in}$
  $\mathsf{c}'$
  $\{\ \mathsf{res} = \text{‘}\mathsf{i}\text{’}\ \}$
```

The specification of *main* is not in an I/O style. The postcondition simply states the return value is ‘i’.

To be able to verify *main*, we need to come up with an I/O invariant and updater functions.

We define the I/O invariant as follows:

```
$\textbf{predicate}\ \mathit{io\_invar} = \lambda \mathsf{id}, \mathsf{v}.$
$\exists\, \mathsf{v}, \mathsf{l}, \mathsf{r}, \mathsf{l\_todo}, \mathsf{r\_todo}.$
      $*\ [1/2]\textbf{gc}(\mathsf{id}, \mathrm{left}(\mathrm{init}), \mathsf{l})$
      $*\ [1/2]\textbf{gc}(\mathsf{id}, \mathrm{right}(\mathrm{init}), \mathsf{r})$
      $*\ \mathsf{l} \mathbin{++} \mathsf{l\_todo} = \langle \text{‘}\mathsf{i}\text{’} \rangle$
      $*\ \mathsf{r} \mathbin{++} \mathsf{r\_todo} = \mathsf{l}$
      $*\ \textbf{if}\ \mathsf{l} = \langle\rangle\ \textbf{then}$
          $\mathsf{v} = 0$
        $\textbf{else}$
          $\mathsf{v} = \mathrm{last}(\mathsf{l}) \wedge \mathsf{v} \neq 0$
```

The I/O invariant uses two progresses: one progress for left(init) for the I/O thread that writes, and one progress for right(init) for the I/O thread that reads. It states that the bytes read must be a prefix of the bytes written. It allows reading to lag behind on writing, which is necessary in a multithreaded setting.

The proof outline for *main* and the updater functions are as follows.

```
$main = \lambda.$
  $\{\ \textbf{emp}\ \}$
  $\textbf{let}\ \mathsf{b} := \mathit{create\_buf}()\ \textbf{in}$
```

```
let id := create_gcf() in
let prid := pc() in
{ ... * pr(prid, ?c) }

let iot1 := init in
let iotw := left(iot1) in
let iotr := right(iot1) in
let iot2 := join(iotw, iotr) in
let t1 := place(iot1, ⟨⟩, none, none, io_invar, id) in
let tw1 := place(iotw, ⟨⟩, t1, none, io_invar, id) in
let tw2 := place(iotw, ⟨'i'⟩, t1, none, io_invar, id) in
let tr1 := place(iotr, ⟨⟩, t1, none, io_invar, id) in
{ ... * ?tr2 = place(iotr, ⟨c⟩, t1, none, io_invar, id) }
{ ... * ?t2 = place(iot2, ⟨⟩, tw2, tr2, io_invar, id) }

let _ := gcf_cons(id, iot1, ⟨⟩) in
let _ := gcf_cons(id, iotw, ⟨⟩) in
let _ := gcf_cons(id, iotr, ⟨⟩) in
let _ := gcf_cons(id, iot2, ⟨⟩) in
let _^g := atintro in
let updater_w := λ.
    { io_invar(id, _) * token_rec(tw1) }
    (id, left(init)) := ⟨'i'⟩
    { io_invar(id, 'i') * token_rec(tw2) }
in
let updater_r := λ ; x .
    { io_invar(id, c) * token_rec(tr1) * c ≠ 0 ∧ c = x }
    (id, right(init)) := ⟨x⟩
    { io_invar(id, c) * token_rec(tr2) }
in
let r := {  "print" : {"update_fun" : updater_w, "t1" : tw1},
            "peek" : {"update_fun" : updater_r, "t1" : tr1, "prid" : prid}} in
```

$$\frac{\mathbf{pr}(v_{id}, \_) \notin h}{\mathbf{pc}() \Downarrow (h, h \uplus \{\!\!\{\mathbf{pr}(v_{id}, v)\}\!\!\}) \cdot \epsilon, v_{id}}\ \text{Pc}$$

$$\frac{}{\mathbf{pa}(v_{id}, v) \Downarrow (h \uplus \{\!\!\{\mathbf{pr}(v_{id}, v)\}\!\!\}, h) \cdot \epsilon, \text{unit}}\ \text{Pa}$$

$$\frac{\nexists v'.\ h(\mathbf{pr}(v_1, v')) \geq 1}{\mathbf{pa}(v_1, v_2) \Downarrow (h, \bot) \cdot \epsilon, \bot}\ \text{PaErr}$$

Figure 4.13: Step semantics for prophecies

$$\left\{\begin{array}{l} \mathit{token}(\mathsf{b}, \mathsf{t1}) * \mathit{split\_io}(\mathsf{t1}, \mathsf{tr1}, \mathsf{tw1}) \\ * \mathit{putchar\_io}(\mathsf{tw1}, \text{‘i’}, \mathsf{tw2}, \mathsf{r}.\text{“print”}) \\ * \mathit{peekchar\_io}(\mathsf{tr1}, c, \mathit{tr2}, \mathsf{r}.\text{“peek”}) \\ * \mathit{join\_io}(\mathit{tr2}, \mathsf{tw2}, \mathit{t2}) \end{array}\right\}$$

$\textbf{let}\ \mathsf{c}' := \mathit{readwrite}(\text{‘i’}\ ; \mathsf{r}\ )\ \textbf{in}$

$\{\ \mathit{token}(\mathsf{b}, \mathit{t2}) * \mathsf{c}' = c\ \}$

$\textbf{let}\ \ \_^{\mathrm{g}}\ := \textbf{atdel in}$

$\{\ \mathsf{c}' = c \land c = \text{‘i’}\ \}$

c'

$\{\ \mathsf{res} = \text{‘i’}\ \}$

4.5.4 Formalization: Prophecies

In this subsection we formalize prophecies. Prophecies were introduced informaly in Sec. 4.5.1.

We extend the syntax of commands with a command to create prophecies and a command to assign to prophecies:

Definition 79: Commands extended for prophecies

$$c ::= \ \ldots \mid \mathbf{pc}() \mid \mathbf{pa}(e, e)$$

$$\frac{}{\vdash \left\{\mathbf{emp}\right\} \mathbf{pc}() \left\{\exists v.\mathbf{pr}(\mathsf{res}, v)\right\}_I^{\mathrm{g}}}\ \text{Pc}$$

$$\frac{}{\vdash \left\{\mathbf{pr}(v_{id}, v_1)\right\} \mathbf{pa}(v_{id}, v_2) \left\{v_1 = v_2\right\}_I^{\mathrm{g}}}\ \text{Pa}$$

Figure 4.14: Proof rules for prophecies

A heap chunk can now also be a prophecy chunk:

Definition 80: Chunks extended for prophecies

$C ::= \ldots \mid \mathbf{pr}(v, v)$

Definition 81: Assertions extended for prophecies

$P ::= \ldots \mid \mathbf{pr}(e, e)$

A chunk $\mathbf{pr}(v_1, v_2)$ states a prophetic value v_2 exists with ID v_1. The **pa** command is used to assign a value to a prophetic value. To specify to which prophetic value should be assigned, it is not an option to have the prophetic value as an argument to **pa** since the prophetic value is not accessible from code (not from ghost code, not from real code). The IDs are used to specify which prophetic value one wants to assign to. A command $\mathbf{pa}(v_1, v_2)$ assigns the value v_2 to the prophetic value with ID v_1.

Definition 82: Step semantics extended for prophecies

Fig. 4.13 on the preceding page lists the step semantics for prophecies.

The Pc step rule makes sure there are never two prophecy chunks with the same ID by requiring that no prophecy chunk with the ID already exists in the heap. The prophetic value is unconstrained: the execution of $\mathbf{pc}()$ that yields prophetic value 4 is a valid execution. For example, the execution $\mathbf{pc}() \Downarrow (\{\!\!\{\}\!\!\}, \{\!\!\{\mathbf{pr}(1, 4)\}\!\!\}) \cdot \epsilon, 1$ is a possible execution: this we can prove directly using the Pc step rule. It is the execution where the a prophetic value 4 with ID 1 is created.

So what happens if we actually wanted to assign value 5 to the prophetic value 4? In case we assign 5 to the prophetic value, we have a command such as **let** $id := \mathbf{pc}()$ **in** $\mathbf{pa}(id, 5)$. While it is true that

$$\mathbf{pc}() \Downarrow (\{\!\!\}, \{\!\!\{\mathbf{pr}(1,4)\}\!\!\}) \cdot \epsilon, 1$$

it is not true that

$$\mathbf{pa}(1,5) \Downarrow (\{\!\!\{\mathbf{pr}(1,4)\}\!\!\}, \{\!\!\{\mathbf{pr}(1,5,\text{assig})\}\!\!\}) \cdot \epsilon, \text{unit} \qquad \text{(not true)}$$

So there simply does not exist an execution where a prophetic value 4 is created and later value 5 is assigned to it. There does, however, exist an execution where prophetic value 4 is created and later value 4 is assigned to it. Note that the assignment of the prophetic value does not have to be a hardcoded constant such as 4: it could be the result of a complex calculation.

Definition 83: Extended proof rules for prophecies

Fig. 4.14 on the preceding page extends the proof rules with two rules for prophecies.

The Pc proof rule simply states that after creating a prophecy chunk and value, the prophetic value exists and the prophecy chunk is on the heap. Because of the existential quantification in the assertion of the postcondition, we simply know a prophetic value exists but we do not know what its value is or is not.

The Pa proof rule states that after assigning to a prophetic value, the prophetic value is equal to the chosen value. This equality in the postcondition conceptually brings the prophetic value “to the ghost code”: before the assignment the ghost code cannot talk about the prophetic value, but after the assignment the prophetic value is known to be equal to the chosen value, therefore talking about the chosen value after that point is the same as talking about the prophetic value. This allows one to have a program that returns a value, where the postcondition states the return value is equal to the prophetic value. Normally the program cannot return the prophetic value because it does not have access to it, but after assigning a value to the prophetic value it can return the value.

4.5.5 Formalization: Recursive functions

In this subsection we formalize recursive functions. Recursive functions were explained informally in Sec. 4.5.2 and used in Sec. 4.5.3.

Consider the following example.

$$\frac{\begin{array}{c}\forall \overline{v'_x}, \overline{v'_y}, \overline{v'_z}.\ \vdash \left\{\begin{array}{c} P[\overline{v'_x}/\overline{x}] \\ [\overline{v'_y}/\overline{y}] \\ [\overline{v'_z}/\overline{z}] \\ * A(A) \end{array}\right\} c[G, \overline{v'_x}/g, \overline{x}][\overline{v'_y}/\overline{y}] \left\{\begin{array}{c} Q[\overline{v'_x}/\overline{x}] \\ [\overline{v'_y}/\overline{y}] \\ [\overline{v'_z}/\overline{z}] \end{array}\right\}^l_I \\ A = \lambda a.\ \{g = G * a(a) * P\}\ g(g, \overline{x}; \overline{y})\ \{Q\}^l_I \\ \overline{z} = \mathrm{fv}(P * Q) \setminus (\overline{x} \cup \overline{y}) \\ \{g, a\} \cap \overline{z} = \emptyset \\ \mathsf{res} \notin \mathrm{fv}(P) \\ G = \lambda g, \overline{x}; \overline{y}.\ c \end{array}}{\vdash \Big\{\mathbf{emp}\Big\}\ \lambda\overline{x}; \overline{y}.\ \mathbf{let}\ g = \lambda g, \overline{x}; \overline{y}.\ c\ \mathbf{in}\ g(g, \overline{x}; \overline{y}) \Big\{\{P\}\ \mathsf{res}\ \{Q\}^l_I\Big\}^l_I}\ \text{Rec}$$

Figure 4.15: Derived proof rule for recursive functions

```
let f := λx ;y .
   let g := λg, x ;y .
      g(g, x ;y ) ❶
   in
   g(g, x ;y ) ❷
in
f(2 ;3 ) ❸
```

Executing this example would result in an infinite loop.

In this example, a function f is defined. Its body contains a definition of a function g. g expects to get itself as an argument g. So the body of g can call itself (see ❶). The body of f contains, after defining g, a function call to g (see ❷) where it passes g as an argument to g. After f is defined, f can just be called (see ❸) without having to worry that internally recursion happens.

For functions that perform recursion, we use the following syntactic sugar:

let rec $f := \lambda\overline{x}\ ;\overline{y}\ .\, c_f$ **in** c'

We only use this when $\overline{x}$ and $\overline{y}$ do not contain f, and occurrences of f in c_f are function applications. So we do not consider cases where code in c_f passes f to another function or compares it to other values.

This syntax is syntactic sugar for:

```
**let** $f := \lambda \overline{x}\ ; \overline{y}\ .$
  **let** $g := \lambda g, \overline{x}\ ; \overline{y}\ .$
    $c_g$
  **in**
  $g(g, \overline{x}\ ; \overline{y}\ )$
**in**
$c'$
```

where c_g is constructed by replacing recursive calls $f(\overline{e_x}; \overline{e_y})$ in c_f with $g(g, \overline{e_x}; \overline{e_y})$.

To prove a Hoare triple for a program of this shape, we introduce a derived proof rule, see Fig. 4.15 on the previous page. Intuitively, it requires to prove a Hoare triple for the body of the recursive function (c_g), where the precondition includes a Hoare triple assertion such that c_g can call itself.

A proof that this derived proof rule holds, is in Appendix B.2 on page 207.

4.6 Multiple instances of the same data structure

In Sec. 4.4.1 we defined functions to manipulate a buffer. These functions had non-I/O style specifications. This buffer and functions were then used in I/O style functions such as *putchar*. *putchar* was then used in an application, in such a way that *putchar* can be reused in a different application without changing its specification.

Ideally, we would like to be able to (re)use *putchar* in a setting where there are multiple buffers. We want to go even further, and support accessing the different buffers concurrently.

We did not support having multiple instances of the buffer, since *buffer_invar* (used in *token*) contained hardcoded knowledge that there is only one buffer.

To solve this problem, we put each instance of the buffer in a different atomic space (see ❶ below). We also create a ghost cell for each buffer, of which we we put a $1/2$ fraction in the atomic space of the buffer. The assertion of the atomic space enforces that the ghost cell equals the content of the buffer. The other $1/2$ fractions of the ghost cells that track the content of the buffers are in the I/O invariant (❷). This way, the I/O invariant "knows" the contents of all the buffers.

predicate $token = \lambda \mathsf{b}, \mathsf{t}.$

$$\begin{aligned}&[\mathsf{t}.\mathrm{iot.f}]\boxed{\mathsf{t}.\mathrm{invar}(\mathsf{t}.\mathrm{id})}\\ &* \ \mathit{token\_rec}(\mathsf{t})\end{aligned}$$

$$\begin{aligned}&\textbf{predicate}\ \mathit{buffer} = \lambda \mathsf{id}, \mathsf{name}, \mathsf{b}.\\ &\quad \boxed{\exists \mathsf{v}.\ \mathit{buf}(\mathsf{b}, \mathsf{v}) * [1/2]\mathbf{gc}(\mathsf{id}, \mathsf{name}, \mathsf{v})}\ ❶\end{aligned}$$

$$\begin{aligned}&\textbf{predicate}\ \mathit{example\_io\_inv} = \lambda \mathsf{id}.\\ &\exists \mathsf{buf1}, \mathsf{buf2}, \mathsf{prog1}, \mathsf{prog2}.\\ &\quad \mathbf{gc}(\mathsf{id}, \mathrm{left}(\mathrm{init}), \mathsf{prog1})\\ &\quad * \ \mathbf{gc}(\mathsf{id}, \mathrm{right}(\mathrm{init}), \mathsf{prog2})\\ &\quad * \ \mathbf{gc}(\mathsf{id}, \text{“}\mathsf{buf1}\text{”}, \mathsf{buf1})\ ❷\\ &\quad * \ \mathbf{gc}(\mathsf{id}, \text{“}\mathsf{buf2}\text{”}, \mathsf{buf2})\ ❷\\ &\quad * \ \ldots\ //\ (\text{constrains}\ \mathsf{buf1}, \mathsf{buf2}, \mathsf{prog1}, \mathsf{prog2})\end{aligned}$$

4.6.1 Example: *putchar* (multiple instances supported)

Since there are potentially multiple instances of the buffer, the precondition of *putchar* now contains the specific buffer to which *putchar* writes ($[?\pi]\mathit{buffer}(\ldots)$).

$$\begin{aligned}&\mathit{putchar} = \lambda \mathsf{b}, \mathsf{c}\ ; \mathsf{r}\ .\\ &\quad \left\{ \begin{array}{l} \mathit{token}(?t_1) * [?\pi]\mathit{buffer}(t_1.\mathrm{id}, ?\mathit{buf\_name}, \mathsf{b})\\ * \ \mathit{putchar\_io}(\mathit{buf\_name}, t_1, \mathsf{c}, ?t_2, \mathsf{r}) \end{array} \right\}\\ &\quad \langle\ \mathit{buf\_write}(\mathsf{b}, \mathsf{c})\ \rangle\\ &\quad \{\ \mathit{token}(t_2) * [\pi]\mathit{buffer}(t_2.\mathrm{id}, \mathsf{buf\_name}, \mathsf{b})\ \}\end{aligned}$$

The updater function's precondition contains a 1/2 fraction of the ghost cell that tracks the contents of the buffer. The other 1/2 fraction is in the I/O invariant. The updater function writes to this ghost cell, and shows that the I/O invariant still holds after doing so. It also still updates the ghost cell that tracks the progress.

$$\begin{aligned}&\textbf{predicate}\ \mathit{putchar\_io} = \lambda \mathsf{buf\_name}, \mathsf{t}_1, \mathsf{c}, \mathsf{t}_2, \mathsf{r}.\\ &\quad \mathsf{t}_2 = \mathsf{t}_1[\mathrm{progress} := \mathsf{t}_1.\mathrm{progress} +\!\!+ \langle \mathsf{c} \rangle]\\ &\quad * \ \mathsf{r}.\text{“}\mathsf{t1}\text{”} = \mathsf{t}_1\\ &\quad * \ \{\ \mathsf{t}_1.\mathrm{invar}(t_1.\mathrm{id}) * [1/2]\mathbf{gc}(t_1.\mathrm{id}, \mathsf{buf\_name}, \_) * \mathit{token\_rec}(\mathsf{t}_1)\ \}\\ &\qquad \mathsf{r}.\text{“}\mathsf{update\_fun}\text{”}()\end{aligned}$$

$$\{\ \mathsf{t}_2.\text{invar}(t_2.\text{id}) * [1/2]\mathbf{gc}(t_1.\text{id}, \mathsf{buf\_name}, \mathsf{c}) * \textit{token\_rec}(\mathsf{t}_2)\ \}$$

We adapt the verification of *putchar* to its new specification and to the new predicate definitions:

$$
\begin{array}{l}
\textit{putchar} = \lambda \mathsf{b}, \mathsf{c}\ ; \mathsf{r}\ .\\
\quad \left\{ \begin{array}{l} \textit{token}(?t_1) * [?\pi]\textit{buffer}(t_1.\text{id}, ?\textit{buf\_name}, \mathsf{b}) \\ * \textit{putchar\_io}(\textit{buf\_name}, t_1, \mathsf{c}, ?t_2, \mathsf{r}) \end{array} \right\}\\
\quad \langle\\
\qquad \left\{ \begin{array}{l} \textit{token\_rec}(t_1) * \textit{buf}(\mathsf{b}, ?v) * [1/2]\mathbf{gc}(t_1.\text{id}, \textit{buf\_name}, v) * t_1.\text{invar}(t_1.\text{id}) \\ * \mathsf{r}.\text{``}\mathsf{t1}\text{''} = t_1 * t_2 = t_1[\text{progress} := t_1.\text{progress} +\!\!+ \langle \mathsf{c} \rangle] \\ * \ \{\ t_1.\text{invar}(t_1.\text{id}) * [1/2]\mathbf{gc}(t_1.\text{id}, \textit{buf\_name}, \_) * \textit{token\_rec}(t_1)\ \} \\ \quad \mathsf{r}.\text{``}\mathsf{update\_fun}\text{''}() \\ \quad \{\ t_2.\text{invar}(t_2.\text{id}) * [1/2]\mathbf{gc}(t_1.\text{id}, \textit{buf\_name}, \mathsf{c}) * \textit{token\_rec}(t_2)\ \} \end{array} \right\}\\
\qquad \textit{buf\_write}(\mathsf{b}, \mathsf{c});\\
\qquad \textbf{let}\ \ \_^{\mathrm{g}}\ := \mathsf{r}.\text{``}\mathsf{update\_fun}\text{''}()\ \textbf{in}\ \text{unit}\\
\qquad \{\ \textit{token\_rec}(t_2) * \textit{buf}(\mathsf{b}, \mathsf{c}) * [1/2]\mathbf{gc}(t_1.\text{id}, \textit{buf\_name}, \mathsf{c}) * t_2.\text{invar}(t_2.\text{id})\ \}\\
\quad \rangle\\
\quad \{\ \textit{token}(t_2) * [\pi]\textit{buffer}(t_2.\text{id}, \textit{buf\_name}, \mathsf{b})\ \}
\end{array}
$$

4.6.2 Example: Writer cat reader

The code of the example of this section is too big to fit in paper format, but is verified with the VeriFast program verifier (see Sec. 3.7.5). It is written as C files with annotations and follows the I/O verification approach.

The example consists of three parts, each running in a different thread. The first part, the writer, writes to a buffer. The second part, called cat, reads from that buffer and writes what it reads to a second buffer. The third part, the reader, reads from the second buffer and calculates the sum of the values read. It returns this sum.

The program is built compositionally.

- At the lower layer is a circular buffer with a regular (non-I/O style) specification.

- A getchar and a putchar function are implemented on top of the circular buffer. Synchronization is not done by atomics but by using a mutex and condition variables for the buffer, and a ghost mutex for the I/O invariant. The specifications are I/O style.
- The writer, cat, and reader functions have I/O style specifications and are implemented on top of getchar and putchar. writer just writes the numbers 1, 2, 3, 4. cat and reader do not know what they will read.
- The main function does not have an I/O style specification. It creates the two buffers and starts the threads such that writer, cat, and reader functions run concurrently. main waits until they terminate. It then returns what reader returned. The postcondition of main states the return value is equal to 10. The verification of main manages to extract this information from the I/O style postconditions of writer, cat and reader.

4.7 Formalization: Erasure

Programs can contain commands that are only meant for verification, namely commands for ghost cell families, prophecies and atomic spaces. Erasure of a program means removing these parts from the program. Execution of an erased program is the same as executing the original (not erased) program, except the original program can have ghost cells chunks, prophecy chunks and atomic space chunks in the heap. The return values and writing to and reading from points-to heap cells remain the same.

Definition 84: Erasure ($\text{er}(c)$)

Erasure of a command is defined in Fig. 4.16 on the next page.

This definition relies on the availability of a termination checker or a way to prove termination: we want to only consider programs where the ghost code terminates. This termination checker or proof system needs to be sound but does not have to be complete: if it can show termination of the ghost code of interest it is good enough. Termination checking is outside the scope of this thesis, see e.g. [33].

We do not want the result of ghost code to be used in real code. For example, we do not want code that first creates a ghost cell family, and then writes to memory the ID of the ghost cell family.

This is enforced as follows.

$$\begin{array}{rcl}
\mathrm{er}(\mathbf{let}\ x^{\mathrm{r}} := c_1\ \mathbf{in}\ c_2) & = & \mathbf{let}\ x^{\mathrm{r}} := \mathrm{er}(c_1)\ \mathbf{in}\ \mathrm{er}(c_2) \\
\mathrm{er}(\mathbf{let}\ x^{\mathrm{g}} := c_1\ \mathbf{in}\ c_2) & = & \mathrm{er}(c_2)\ \text{(undefined if } \neg\text{terminationchecker}(c_1)) \\
\mathrm{er}(x^{\mathrm{g}}) & = & \text{(undefined)} \\
\mathrm{er}(x^{\mathrm{r}}) & = & x^{\mathrm{r}} \\
\mathrm{er}(e :: \overline{e}) & = & \mathrm{er}(e) :: \mathrm{er}(\overline{e}) \\
\mathrm{er}(\langle\rangle) & = & \langle\rangle \\
\mathrm{er}(v) & = & v \\
\mathrm{er}(e_1(\overline{e_2}; \overline{e_3})) & = & \mathrm{er}(e_1)(\mathrm{er}(\overline{e_2})) \\
\mathrm{er}(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) & = & \mathbf{if}\ \mathrm{er}(e)\ \mathbf{then}\ \mathrm{er}(c_1)\ \mathbf{else}\ \mathrm{er}(c_2) \\
\mathrm{er}(c_1 \,||\, c_2) & = & \mathrm{er}(c_1) \,||\, \mathrm{er}(c_2) \\
\mathrm{er}([e]) & = & [\mathrm{er}(e)] \\
\mathrm{er}([e_1] := e_2) & = & [\mathrm{er}(e_1)] := \mathrm{er}(e_2) \\
\mathrm{er}(\mathbf{cons}(\overline{e})) & = & \mathbf{cons}(\mathrm{er}(\overline{e})) \\
\mathrm{er}(\lambda \overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}}.\, c) & = & \lambda \overline{x^{\mathrm{r}}}.\, \mathrm{er}(c) \\
\mathrm{er}(\langle c \rangle) & = & \langle \mathrm{er}(c) \rangle \\
\mathrm{er}(\mathbf{create\_gcf}()) & = & \text{(undefined)} \\
\mathrm{er}(\mathbf{gcf\_cons}(e_1, e_2, e_3)) & = & \text{(undefined)} \\
\mathrm{er}((e_1, e_2) := e_3) & = & \text{(undefined)} \\
\mathrm{er}(\mathbf{pc}()) & = & \text{(undefined)} \\
\mathrm{er}(\mathbf{pa}(e_1, e_2)) & = & \text{(undefined)}
\end{array}$$

Figure 4.16: Definition of erasure

First, Hoare triples are annotated as ghost or real, using a superscript g or r: $\{P\}\, c\, \{Q\}^{\mathrm{r}}$ is annotated as real and $\{P\}\, c\, \{Q\}^{\mathrm{g}}$ is annotated as ghost. We use l to range over $\{\mathrm{g}, \mathrm{r}\}$.

Second, for non-compound commands, the proof rules enforce that Hoare triples for ghost commands are marked as ghost, and for real commands are marked as real. For example, the proof rules in Fig. 4.10 p. 117 allow to prove $\{P\}\, \mathbf{create\_gcf}()\, \{Q\}^{\mathrm{g}}_{I}$ but not $\{P\}\, \mathbf{create\_gcf}()\, \{Q\}^{\mathrm{r}}_{I}$.

Third, we do not want real code nested inside ghost code. The only compound command that allows mixing ghost and real code is **let** $x := c_1$ **in** c_2. Here, c_1 can be ghost while c_2 can be real. Looking at the proof rules in Fig. 4.6 p. 112 we can see that this is allowed, but these proof rules do not allow c_1 to be real while c_2 is ghost.

Fourth, for compound commands, the proof rules enforce that there is no "data flow" from ghost commands to real commands, but data flow from real commands to ghost commands is okay. To do that, we split variables into ghost variables and real variables. A superscript indicates whether a variable is ghost or real: x^{r} is a real variable and y^{g} is ghost. Real code should not use ghost variables, but ghost code can use real variables.

Fifth, since we support higher order function, we should prevent ghost code to call a function that is passed as an argument if that function contains real code. The proof rule to call a function in Fig. 4.11 p. 126 enforces this. A Hoare triple assertion $\{P\}\, v(\overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}})\, \{Q\}^{l}$ also has a ghost level l. Given such an assertion in the precondition, the function call is allowed if the caller has the same ghost level as the callee according to the assertion. Also note that in the definition of F_Λ, we require the Hoare triple to be provable with the given ghost level. If this was left out from the definition, one could swap a ghost Hoare triple chunk with a real Hoare triple chunk using the consequence proof rule.

4.8 Soundness

We prove soundness of the approach (except erasure).

Lemma 1: Safe interleaving

$$\begin{aligned}
&\forall h_A, \tau_A, Q_A, h_B, \tau_B, Q_B, \Lambda, I.\\
&\quad \mathrm{safe}(\Lambda, h_A, \tau_A, I, Q_A) \wedge\\
&\quad \mathrm{safe}(\Lambda, h_B, \tau_B, I, Q_B) \wedge\\
&\quad \tau \in \tau_A \,||\, \tau_B \Rightarrow\\
&\quad \mathrm{safe}(\Lambda, h_A \uplus h_B, \tau, I, Q_A * Q_B).
\end{aligned}$$

Proof. Well-founded induction on Λ. Case analysis on τ. (Full proof on p. 185) □

Lemma 2: Safe seq

$$\begin{aligned}
&\forall \Lambda, h, \tau_A, I, Q_1, Q_2.\\
&\big(\mathrm{safe}(\Lambda, h, \tau_A, I, Q_1) \wedge (\forall \Lambda. \forall h_Q. h_Q \vDash_{F_m} Q_1 \Rightarrow \mathrm{safe}(\Lambda, h_Q, \tau_B, I, Q_2)\big)\\
&\Rightarrow \mathrm{safe}(\Lambda, h, \tau_A; \tau_B, I, Q_2).
\end{aligned}$$

Proof. Well-founded induction on Λ. Case analysis on τ_A and τ_B. (Full proof on p. 187) □

Lemma 3: Safe leak

$\forall \Lambda, h, \tau, I, Q, R.\ \mathrm{safe}(\Lambda, h, \tau, I, Q * R) \Rightarrow \mathrm{safe}(\Lambda, h, \tau, I, Q).$

Proof. Well-founded induction on Λ. Case analysis on τ. (Full proof on p. 189) □

Lemma 4: Safe frame

$\forall \Lambda, h, Q, R, I, \tau, h_R.$
$\mathrm{safe}(\Lambda, h, \tau, I, Q) \wedge h_R \vDash_{F_\Lambda} R \Rightarrow \mathrm{safe}(\Lambda, h \uplus h_R, \tau, I, Q * R)$

Proof. Well-founded induction on Λ. Case analysis on τ. (Full proof on p. 190) □

Lemma 5: Safe conseq

$\forall \Lambda, h, \tau, I, Q, P.$
$(P \rightarrow Q) \Rightarrow$
$\mathrm{safe}(\Lambda, h, \tau, I, P) \Rightarrow$
$\mathrm{safe}(\Lambda, h, \tau, I, Q).$

Proof. Well-founded induction on Λ. Case analysis on τ. (Full proof on p. 191) □

Lemma 6: Safe outcome

$\forall \Lambda, h_1, \tau, n, h_2, \Lambda, Q, h_r.$
$h_1, \tau \rightarrow (n, h_2)$
$\wedge\ \mathrm{safe}(\Lambda +_{\mathrm{inat}} n +_{\mathrm{inat}} 1, h, \tau, \mathrm{inat}, Q)$
$\wedge\ h_1 = \mathrm{erat}(h \uplus h_r)$
$\Rightarrow$
$\exists h', h_{rr}.\ h_2 = \mathrm{erat}(h' \uplus h_{rr} \uplus h_r)$
$\wedge\ h' \vDash_{F_\Lambda} Q$
$\wedge\ \mathrm{inv}(h \uplus h_r, \mathrm{outat}) = \mathrm{inv}(h' \uplus h_{rr} \uplus h_r, \mathrm{outat}).$

Proof. Induction on n. (Full proof on p. 191) □

Lemma 7: Unsafe inat outcome

$\forall h_1, \tau, n, h, Q, h_r.\ h_1, \tau \to (n, \bot) \wedge h_1 = h \uplus h_r \Rightarrow$
$\neg\text{safe}((0, n+1), h, \tau, \text{inat}, Q)$.

Proof. Induction on n. (Full proof on p. 192) □

Lemma 8: Hoare triple validity

$\forall P, c, Q, I.\ \vdash \{P\}\, c\, \{Q\}_I \Rightarrow\ \models \{P\}\, c\, \{Q\}_I$.

Proof. Induction on $\vdash \{P\}\, c\, \{Q\}_I$ using lemmas. (Full proof on p. 193) □

Definition 85: OK

$\text{OK}(\tau) = \forall O.\ \emptyset, \tau \to O \Rightarrow \nexists n. O = (n, \bot)$

Lemma 9: Unsafe outcome

$\forall h, \tau, Q.\ \neg\text{OK}(\tau) \Rightarrow \exists \Lambda. \neg\text{safe}(\Lambda, \emptyset, \tau, \text{outat}, \mathbf{emp})$

Proof. Induction on the number of steps until error. (Full proof on p. 206) □

Lemma 10: Error preservation (erasure)

$\forall c, \tau, v, P, Q, I.\ \text{er}(c) \Downarrow \tau, v \wedge \neg\text{OK}(\tau) \wedge \vdash \{P\}\, c\, \{Q\}_I^{\text{r}} \Rightarrow \exists \tau'.\ c \Downarrow \tau', v \wedge \neg\text{OK}(\tau')$

To prove this, one needs an approach for proving termination of ghost code, which is outside the scope of this thesis. Remember that we only consider programs where the ghost code terminates.

Theorem 5: Soundness (in-memory)

$$\begin{array}{l}\forall c.\ \vdash \{\mathbf{emp}\}\, c\, \{\mathbf{emp}\}^{\mathrm{r}}_{\mathrm{outat}} \Rightarrow \\ \quad \forall \tau, v.\ \mathrm{er}(c) \Downarrow \tau, v \Rightarrow \\ \quad \mathrm{OK}(\tau)\end{array}$$

Proof. Proof by contradiction using lemmas. (Full proof on p. 207) □

4.9 Conclusions, related work, and future work

We used an I/O style of writing specifications and verifying programs that perform I/O, to verify programs that do not perform I/O. We implemented the approach on top of the VeriFast program verifier for verifying C programs. We formalized a programming language that supports nontermination, concurrency, atomic blocks, and prophecies. We formulated Hoare logic style proof rules for this programming language, showed how the approach can be applied in this programming language, and proved soundness of the proof rules, except erasure, which is future work.

Iris [38] provides a framework for reasoning about a shared resource, such as the heap. We expect that our approach (except reading) can be implemented in Iris rather straightforwardly. This is future work. Supporting prophecies or finding a way to support reading in Iris is also future work.

Around the same time that we developed the I/O verification approach (Chapter 3) and the I/O style verification approach for memory manipulating programs (this chapter), histories [10] and futures [52] were developed.

The approach of histories [10] is an approach for functional verification of terminating executions of concurrent software. The approach uses a process algebra that supports parallel and sequential composition, choice, if-then-else, named processes with arguments, and actions with arguments for which preconditions and postconditions are written. Performed actions are registered in such process algebra term. Annotations declare which blocks of code corresponds to which action. The tool verifies whether such code block respects the specification of the action consisting of the precondition and postcondition. A history can be destructed. After destruction, one receives standard memory chunk permission to the memory of which the contents is tracked by the history. Upon destruction, a prover for the process algebra is used to automatically verify whether desired properties hold for the process algebra term that represents the history of performed actions, e.g. to conclude from the history that a calculated value equals the desired value. Thread local reasoning is supported by splitting and merging such process algebra terms such that each thread can independently use one. Furthermore, separate process algebra term can be used for tracking disjoint sets of memory locations. Splitting and merging such histories use dummy synchronization actions to remember at which point a history was split. For expressivity, the process algebra term can also contain a process name with arguments – this allows recursion in the specification consisting of such a process algebra term: the postcondition of a function can state the history is a process algebra term that contains such a process name with arguments. In the approach, for such a process name a precondition and postcondition is written, together with a body consisting of a process algebra term. In the verification approach, the verifier checks whether the process algebra term respects its specification consisting of such a precondition and postcondition, as follows. First, the body is normalized to not contain parallel composition ($||$). This is automatic and might not always work. Second, the normalized body is converted to Java code. Third, the Java code is verified with respect to the precondition and postcondition.

The approach of futures [52] builds further on the idea of histories and adds support for nontermination by switching from a postcondition based approach to a precondition based one: the process algebra term represents permissions to perform actions. To support thread-local reasoning, it supports swapping parallel composition ($||$) of the process algebra with separating conjunction ($*$) from separation logic (instead of sychronization actions). A future contains the actions to be performed in the future, as a process algebra term. When all actions have been performed, the process algebra term is then epsilon. The future remembers the process algebra name for which it is created. The future can only by initialized for a process name with arguments, so the initial process algebra term consists of a process name application. So upon destruction we know the postcondition of the process name holds.

Histories and futures do not support a uniform specification style for I/O and in-memory applications, but we expect histories and futures to be adaptable to also support I/O verification. Compared to our approach, it uses a process algebra to express the actions done and allowed, while we express this using a separation logic with abstract predicates. Because a process algebra is used, it supports verification of liveness properties (which we do not) by performing verification on the level of the process algebra term directly by a verifier for the process algebra, but verification on the level of the process algebra term is not modular.

While the I/O verification approach itself (Chapter 3) does not suffer from heavy annotation overhead, the current implementation of the idea behind our verification approach for memory manipulating programs does. Histories and futures do not seem to have a heavy annotation overhead. Most of the annotations (such as defining the places) are boilerplate; reducing this annotation overhead is future work.

Chapter 5

Conclusion

Errors in software can lead to loss of life (airplanes, electric bikes, hospital equipment, . . .), heavy economical losses (power outages, self destructing rockets, . . .), serious security issues (protocol implementations, parsers, . . .), and daily frustration (crashes of desktop software, . . .). Software is just data. Contrary to physical equipment that can suffer from mechanical faults (wear, corrosion, . . .) software can not.

But since software is just data, we should be able to create the perfect data – in theory. In practice: how does the creator of a piece of software know, after having created the software, that it is free of bugs: that there are e.g. no unintended race conditions or forgotten input validations? "I do not remember having forgotten something" does not sound very convincing. This is not new or unknown: all or almost all programmers do not just write code, they also run their software to check that it works, simply because the probability that the software (or a nontrivial modification thereof) works correctly immediately after writing it is rather low. One step further is automatic tests (i.e. software that runs target software and checks that the results or behavior of the target software is correct), but in nontrivial software, not finding an error through testing does not mean there are no errors.

To know with much more certainty that a piece of software is free of bugs, there are formal sound verification approaches: they allow to prove absence of (classes of) bugs.

In Chapter 2 we tried (and succeeded) in applying such a verification approach on software taken "from the wild": we verified absence of unintended race conditions, conformance to a set of (complex) API rules of, and absence of crashes and illegal memory accesses of a USB keyboard driver from the Linux kernel, using the verification tool VeriFast.

The case study illustrates its usefulness in catching bugs: thanks to the case study, we found two bugs in the driver. They were very well hidden and it was only because of performing verification that we found them. We wrote patches to fix these bugs. The patches are now included in Linux.

We now know the approach can deal with the complex USB API. The approach can also deal with an unbounded number of concurrent threads, which not all verification approaches can deal with.

While the execution time of the verification tool (about one second for the whole driver) does not impose any problems, the number of annotation lines required is substantial: there are about two lines of annotation for one line of C code for the driver itself, plus about the same number of lines of annotations for API specifications as for annotations for the driver itself. The work on API annotations can be reused for other drivers (but (parts of) API unused by this driver and used by another driver will have to be annotated as well when verifying such other driver). Source code that is not complex and does not use a complex API can be verified rather quickly. Complex code, including code that uses a complex API, takes a lot longer to verify. This is as expected but hard to quantify.

Work remains to be done in this area. The case study does not verify functional correctness: if a callback in the driver receives a keypress from the USB API, and the callback sends the keypress to the input API, we do not verify that the callback sends the keypress of the correct key to the input API: if the implementation would always send to the input API that the F5 key is pressed, even if actually another key is pressed, the verification approach would technically not catch this bug. Even worse: we do not verify that any keypress is reported at all. A case study that also verifies functional correctness is future work. Unload-safety (verifying that the kernel does not maintain pointers to memory or functions of the driver after the driver is dynamically unloaded) is also future work.

Besides the case study, we also created a verification approach to verify input/output (I/O) properties of programs (Chapter 3). Input/output is the behavior of the program that can be observed by the outside world. For example, given the software that drives an elevator, we can observe the doors opening, closing, lights going on and off, and the engine starting and stopping. We care about the correct behavior and also its order: first close the door, only after that start the engine. The verification approach does not only allow to specify such intended behavior, but also to verify that any execution of the program does not violate such specification.

Such specifications are formulated as the intended behavior (if button pressed, first close doors; afterwards: if closing doors was successful, then start engine), but not from a crosscutting point of view (at any point in time, do not drive engine while doors are open).

We identified a number of requirements for I/O verification, of which our approach supports the following ones: support verifying the order in which I/O happens, support specifying the I/O behavior depending on the input (e.g. if high number is read, write something else than if a low number is read), support programs that do not always terminate, support underspecification (the program can do A or B), and support modularity and compositionality (define I/O actions on top of other I/O actions) to make the approach more scalable. The approach allows to model and verify complex I/O, as illustrated by the examples such as the one that models arbitrary Turing machines.

There are requirements which our I/O verification approach does not meet. The approach does not verify that any I/O happens or happens in time, nor takes into account any physical constraints (if the brakes are hit, the elevator, car or drone will take some time to stop depending on the brakes and the speed and mass of the object that should stop). These requirements are less pressing for the type of I/O the approach was developed for (writing and reading from files, reading from keyboards and mice, and writing to a computer screen), but are very important in other settings. Adding support for these requirements is future work.

We also implemented the approach in VeriFast. This allows to apply the approach on C programs, which we did for some example programs. The approach is simple and practical in VeriFast.

More technically, we also mathematically formalized it, which allows us to prove soundness: now we know that if a program is verified with the approach, the program indeed does not violate its I/O specifications. We also proved this in Coq (a program that can check correctness of mathematical proofs, if the proof is formulated in a strict and verbose way). Although the the verification approach as implemented in Coq does support modeling nonterminating executions, it does not support verifying nonterminating executions. This is future work. We sketched how the I/O verification approach can be implemented in Iris, a framework for reasoning about shared resources in a concurrent setting. This is only on paper: We do not have a a Coq formalization of this that uses the Coq formalization of Iris; this is future work.

Converting equivalent specifications is future work (e.g. verify a function that has an I/O specification whose body consists of calling another function with equivalent I/O specification). A case study is also future work.

In Chapter 4 we developed an approach to use the same verification style of verifying I/O for programs that do not perform I/O at all, but instead only read and write to memory that can be shared between threads. Besides concurrency, the approach supports reuse of specifications and of code.

Since the same style of I/O specifications is used, one can switch between verifying an implementation that performs memory manipulation (e.g. a filesystem implemented in RAM) and an implementation that performs I/O (e.g. a filesystem implemented by reading/writing network messages), while keeping the same specifications.

We wrote a formalization of a programming language that supports nontermination, concurrency, atomic blocks (blocks of code that are never executed concurrently with other code), and prophecies (allow to mention during verification values at a point in the execution of the program where such value is not yet assigned and thus not yet know what the value will be) together with proof rules to verify specifications. We proved soundness of these proof rules (except erasure). We explained how the approach for I/O style verification of memory-manipulating programs can be applied in the context of this programming language.

We implemented this I/O style verification approach in VeriFast and performed verification of some examples in VeriFast. The amount of annotation overhead for these small examples is high. While we developed an idea for I/O style verification, and illustrated that it works, the current implementation of this idea requires boilerplate annotations (such as defining the places). As an upside, the implementation of the idea of the approach can be applied in VeriFast without making modifications to VeriFast, but we think a more simple implementation of the idea with less annotation overhead should be possible. Developing this is future work. A case study is also future work.

Appendix A

Proofs I/O verification approach

A.1 Unique weakest precondition

In Sec. 22 on p. 46 we defined wp as the weakest monotone solution of a set of equations.

Here we prove that such a solution exists and is unique.

Definition 86: MonoPredTx

We define MonoPredTx as the set of predicate transformers that are monotone.

We range over predicate transformers with *ptx*, and over monotone predicate transformers with *mptx*.

Definition 87: WP

We define WP as the set of predicate transformers *ptx* that satisfy the following equations and are monotone.

- $ptx(\textbf{let } c \textbf{ in } \mathcal{C}, Q) = ptx(c, \lambda v.\, ptx(\mathcal{C}(v), Q))$
- $ptx(f(\overline{v}), Q) = ptx(\mathrm{fc}(f)(\overline{v}), Q)$
- $ptx(bio(v_o), Q) = \lambda h.\, \exists v_i, h'.\, h \xRightarrow{bio(v_o,v_i)} h' \wedge Q(v_i)(h')$
- $ptx(v, Q) = Q(v)$

Definition 88: ordering on PredTx

$ptx_1 \leq ptx_2 \iff \forall c, Q, h.\ ptx_1(c, Q)(h) \Rightarrow ptx_2(c, Q)(h)$

You can think of this ordering as the subset ordering where we consider a predicate transformer as a set of triples (c, Q, h).

Lemma 11

MonoPredTx, $\leq$ is a complete lattice.

Proof. Choose arbitrary $X \subseteq$ MonoPredTx. We have to prove that there is an element in MonoPredTx that is the least upper bound of X (and that there is a highest lower bound; proving this is analogous).

Construct the predicate transformer *mptxl* as follows:
$mptxl(c, Q)(h) = \exists x \in X.\ x(c, Q)(h)$. You can think of it as the union of the predicate transformers in X. *mptxl* is the least upper bound of X. *mptxl* is monotone because every x in X is monotone. □

Definition 89: F

We define a function $\mathrm{F} : \mathrm{PredTx} \to \mathrm{PredTx}$.

- $\mathrm{F}(ptx)(\textbf{let } c \textbf{ in } \mathcal{C}, Q) = ptx(c, \lambda v.\, ptx(\mathcal{C}(v), Q))$
- $\mathrm{F}(ptx)(f(\overline{v}), Q) = ptx(\mathrm{fc}(f)(\overline{v}), Q)$
- $\mathrm{F}(ptx)(bio(v_o), Q) = \lambda h.\, \exists v_i, h'.\, h \xrightarrow{bio(v_o,v_i)} h' \wedge Q(v_i)(h')$
- $\mathrm{F}(ptx)(v, Q) = Q(v)$

Note that, if ptx is in MonoPredTx, then $\mathrm{F}(ptx)$ is in MonoPredTx.

Lemma 12

F is monotone in MonoPredTx, i.e. $\forall mptx, mptx' \in \mathrm{MonoPredTx}.\ mptx \leq mptx' \Rightarrow F(mptx) \leq F(mptx')$.

Proof. Choose arbitrary monotone predicate transformers $mptx$, $mptx'$ such that $mptx \leq mptx'$.

Choose arbitrary c, Q, h.

We have to prove that $\mathrm{F}(mptx)(c, Q)(h) \Rightarrow \mathrm{F}(mptx')(c, Q)(h)$.

Case analysis on c:

- $c = \textbf{let } c' \textbf{ in } \mathcal{C}$. Follows from $mptx \leq mptx'$ and from monotonicity of $mptx$.
- $c = f(\overline{v})$. Follows from $mptx \leq mptx'$.
- $c = bio(v)$. Follows directly from definition of F.
- $c = v$. Follows directly from definition of F.

□

Because Lemma 11 on the facing page and Lemma 12 we know F is a monotone function on a complete lattice. Therefore we can apply

Knaster-Tarski's fixpoint theorem to obtain that F has a greatest fixpoint, which equals the least upper bound of the postfixpoints[1].

Definition 90: wp

We define wp as the least upper bound of the postfixpoints of F:

$$\begin{aligned} wp(c,Q)(h) = \exists mptx.\ & (\forall c', Q', h'. mptx(c',Q')(h') \Rightarrow \mathrm{F}(mptx)(c',Q')(h')) \\ & \wedge mptx(c,Q)(h) \end{aligned}$$

Note that $\mathrm{wp} = \mathrm{F}(\mathrm{wp})$.

Example: consider a function f that just calls itself, i.e. $\mathrm{fc}(f)() = f()$. To show that $\mathrm{wp}(f(), Q)(h)$ we must identify a *mptx* such that some properties hold as defined in Definition 90. A good choice is $mptx(c',Q')(h') = (c' = f())$.

A.2 Safe implies trace simulation

Definition 91

The program trace $\mathrm{no\_io}^\infty$ is defined coinductively as follows: $\mathrm{no\_io}^\infty = \mathrm{no\_io} \cdot \mathrm{no\_io}^\infty$

Definition 92

The program trace $\mathrm{no\_io}^n$ is defined inductively as follows:

- $\mathrm{no\_io}^0 = \langle\rangle$
- $\mathrm{no\_io}^{n+1} = \mathrm{no\_io} \cdot \mathrm{no\_io}^n$

[1]Given a set S with a partial ordering $\leq$ and a function $f : S \rightarrow S$, we call an element $e \in S$ a postfixpoint of f iff $e \leq f(e)$.

Definition 93

We define finiteness of a program trace τ (written $\text{finite}(\tau)$), inductively as follows:

- $\text{finite}(\langle\rangle) = \text{true}$
- $\text{finite}(bio(v_o, v_i) \cdot \tau') = \text{finite}(\tau')$
- $\text{finite}(\text{no\_io} \cdot \tau') = \text{finite}(\tau')$

Lemma 13

$\forall \tau.\ \neg\text{finite}(\tau) \land \neg(\exists n, \tau', bio, v_o, v_i.\ \tau = \text{no\_io}^n \cdot bio(v_o, v_i) \cdot \tau') \Rightarrow \tau = \text{no\_io}^\infty$

Proof. Proof by coinduction. Case analysis on τ.

- $\tau = \langle\rangle$. Cannot occur.
- $\tau = bio'(v_o', v_i') \cdot \tau''$. Cannot occur.
- $\tau = \text{no\_io} \cdot \tau''$ for some τ''. Because of coinduction hypothesis: $\tau'' = \text{no\_io}^\infty$. Therefore, $\tau = \text{no\_io} \cdot \tau'' = \text{no\_io}^\infty$.

$\square$

Lemma 14

$\forall \tau.\ \tau \neq \text{no\_io}^\infty \land \neg\text{finite}(\tau) \Rightarrow \exists n, \tau', bio, v_o, v_i.\ \tau = \text{no\_io}^n \cdot \text{bio}(v_o, v_i) \cdot \tau'$

Proof. Proof by contradiction: suppose $\neg\exists n, \tau', bio, v_o, v_i.\ \tau = \text{no\_io}^n \cdot \text{bio}(v_o, v_i) \cdot \tau'$.

Use Lemma 13 to obtain $\tau = \text{no\_io}^\infty$. Contradiction. $\square$

Lemma 15

$\forall \tau.$
$(\exists n.\ \tau = \text{no\_io}^n) \vee$
$\tau = \text{no\_io}^\infty \vee$
$\exists n, bio, v_o, v_i, \tau'.\ \tau = \text{no\_io}^n \cdot bio(v_o, v_i) \cdot \tau'$

Proof. Because of axiom of excluded middle: $\text{finite}(\tau) \vee \neg\text{finite}(\tau)$.

- $\neg\text{finite}(\tau)$. Because of axiom of excluded middle: $\tau = \text{no\_io}^\infty \vee \neg\tau = \text{no\_io}^\infty$.
  - $\tau = \text{no\_io}^\infty$. Trivial.
  - $\neg\tau = \text{no\_io}^\infty$. Apply Lemma 14 on the preceding page.
- $\text{finite}(\tau)$. Induction on $\text{finite}(\tau)$.
  - $\tau = \text{no\_io} \cdot \tau'$ for some τ'. Case analysis on the induction hypothesis
    * $\tau' = \text{no\_io}^n$ for some n. Then $\tau = \text{no\_io}^{n+1}$.
    * $\tau' = \text{no\_io}^\infty$. Trivial. (alternatively: prove by induction that this case cannot occur)
    * $\tau' = \text{no\_io}^n \cdot bio(v_o, v_i) \cdot \tau''$ for some n, bio, v_o, v_i, τ''. Then $\tau = \text{no\_io}^{n+1} \cdot bio(v_o, v_i) \cdot \tau''$.
  - $\tau = bio(v_o, v_i) \cdot \tau'$ for some bio, v_o, v_i, τ'. Trivial.
  - $\tau = \langle\rangle$. Trivial.

□

Lemma 16

$\forall n, bio, v_o, v_i, P.\ \text{safe}(h, \text{no\_io}^n \cdot bio(v_o, v_i) \cdot \tau', P) \Rightarrow \text{safe}(h, bio(v_o, v_i) \cdot \tau', P)$

Proof. Proof by induction on n. □

Definition 94

The Petri net trace ϵ^m is defined inductively as follows:

- $\epsilon^m = \langle\rangle$
- $\epsilon^{m+1} = \epsilon \cdot \epsilon^m$

Definition 95

We define $h \xrightarrow{\epsilon^m} h'$ inductively as follows:

- $h \xrightarrow{\epsilon^0} h$
- $h_1 \xrightarrow{\epsilon^{m+1}} h_3 = \exists h_2.\ h_1 \xrightarrow{\epsilon} h_2 \wedge h_2 \xrightarrow{\epsilon^m} h_3$

One can prove easily that if $h \xrightarrow{\epsilon}^* h'$, we can identify an m such that $h \xrightarrow{\epsilon^m} h'$.

Definition 96

Given some h, τ, P such that $\text{safe}(h, \tau, P)$, then we define $\text{topetri}(\tau)$ corecursively as follows.

Because of Lemma 15 on page 164 we can perform the following case analysis.

- $\tau = \text{no\_io}^n$ for some n. Then $\text{topetri}(\tau) = \langle\rangle$.
- $\tau = \text{no\_io}^\infty$. Then $\text{topetri}(\tau) = \langle\rangle$.
- $\tau = \text{no\_io}^n \cdot bio(v_o, v_i) \cdot \tau'$ for some n, bio, v_o, v_i, τ'.

  Apply Lemma 16 on page 164 to obtain $\text{safe}(h, bio(v_o, v_i) \cdot \tau', P)$, on which we perform case analysis:

  – SafeBio. Because of the premise of this inference rule: $h \xRightarrow{bio(v_o,v_i)} h'$ and $\text{safe}(h', \tau', P)$ for some h'.

    Because $h \xRightarrow{bio(v_o,v_i)} h'$ we know that $h \xrightarrow{\epsilon^m} h''$ and $h'' \xrightarrow{bio(v_o,v_i)} h'$ for a certain m and h''.

    $\text{topetri}(\tau) = \epsilon^m \cdot bio(v_o, v_i) \cdot \text{topetri}(\tau')$.

  – SafeContradict. Because of the premise of this inference rule: $h \xRightarrow{bio(v_o,v_i')} h'$ for some v_i', h'.

    Because $h \xRightarrow{bio(v_o,v_i')} h'$ we know that $h \xrightarrow{\epsilon^m} h''$ and $h'' \xrightarrow{bio(v_o,v_i')} h'$ for a certain m and h''.

    $\text{topetri}(\tau) = \epsilon^m \cdot bio(v_o, v_i') \cdot \langle\rangle$.

  – SafeNoIO. This case cannot occur.

  – SafePost. This case cannot occur.

Lemma 17

$\forall h_1, h_2, h_3, bio, v_o, v_i, m, \mathbb{T}.$
$h_1 \xrightarrow{\epsilon^m} h_2 \wedge h_2 \xrightarrow{bio(v_o,v_i)} h_3 \wedge h_3 \Downarrow \mathbb{T} \Rightarrow$
$h_1 \Downarrow \epsilon^m \cdot bio(v_o, v_i) \cdot \mathbb{T}$

Proof. Proof by induction on m (use m times the Epsilon inference rule and one time the Bio inference rule of Fig. 3.3 on page 40). □

Lemma 18

$\forall h, \tau, P, \mathbb{T}.\ \text{safe}(h, \tau, P) \wedge \mathbb{T} = \text{topetri}(\tau) \Rightarrow h \Downarrow \mathbb{T}$

Proof. Proof by coinduction.

Case analysis by using Lemma 15 on page 164.

- $\tau = \text{no\_io}^n$ for some n. Then $\mathbb{T} = \langle\rangle$. Trivial.
- $\tau = \text{no\_io}^\infty$. Then $\mathbb{T} = \langle\rangle$. Trivial.
- $\tau = \text{no\_io}^n \cdot bio(v_o, v_i) \cdot \tau'$ for some n, bio, v_o, v_i, τ'.

  Apply Lemma 16 on page 164 to obtain $\text{safe}(h, bio(v_o, v_i) \cdot \tau', P)$, on which we perform case analysis:

  - SafeBio.

    Because of the premise of this inference rule: $h \xRightarrow{bio(v_o,v_i)} h'$ and $\text{safe}(h', \tau', P)$ for some h'.

    Because $h \xRightarrow{bio(v_o,v_i)} h'$ we know that $h \xrightarrow{\epsilon^m} h''$ and $h'' \xrightarrow{bio(v_o,v_i)} h'$ for a certain m and h''.

    By Definition 96 on the facing page: $\mathbb{T} = \epsilon^m \cdot bio(v_o, v_i) \cdot \text{topetri}(\tau')$.

    Let $\mathbb{T}' = \text{topetri}(\tau')$.

    Because of the coinductive hypothesis: $h' \Downarrow \mathbb{T}'$.

    Use Lemma 17 on the preceding page: $h \Downarrow \epsilon^m \cdot bio(v_o, v_i) \cdot \mathbb{T}'$ (note that it is guarded because we always apply the Bio inference rule).

    Rewrite to obtain $h \Downarrow \mathbb{T}$.

  - SafeContradict.

    Because of the premise of this inference rule: $h \xRightarrow{bio(v_o,v_i')} h'$ for some v_i', h'.

    Because $h \xRightarrow{bio(v_o,v_i')} h'$ we know that $h \xrightarrow{\epsilon^m} h''$ and $h'' \xrightarrow{bio(v_o,v_i')} h'$ for a certain m and h''.

    By definition of topetri: $\mathbb{T} = \epsilon^m \cdot bio(v_o, v_i') \cdot \langle\rangle$.

    Use Lemma 17 on the facing page (and $h' \Downarrow \langle\rangle$):
    $h \Downarrow \epsilon^m \cdot bio(v_o, v_i') \cdot \langle\rangle$. (note that it is guarded because we always apply the Bio inference rule).

    Rewrite to obtain $h \Downarrow \mathbb{T}$.

  - SafeNoIO. This case cannot occur.

  – SafePost. This case cannot occur.

□

Lemma 19

$\forall h, \tau, P, \mathbb{T}.\ \mathrm{safe}(h, \tau, P) \wedge \mathbb{T} = \mathrm{topetri}(\tau) \Rightarrow \mathbb{T} \sim \tau.$

Proof. Proof by coinduction.

Case analysis by using Lemma 15 on page 164.

- $\tau = \mathrm{no\_io}^n$ for some n. $\mathbb{T} = \langle\rangle$. What we want to prove follows easily (apply n times the NoIO rule of Fig. 3.4 on page 41 and one time the Empty rule).
- $\tau = \mathrm{no\_io}^\infty$. Then $\mathbb{T} = \langle\rangle$.

  $\tau = \mathrm{no\_io} \cdot \mathrm{no\_io}^\infty$.

  Because of the coinduction hypothesis: $\mathbb{T} \sim \mathrm{no\_io}^\infty$.

  Apply the NoIO rule: $\mathbb{T} \sim \mathrm{no\_io} \cdot \mathrm{no\_io}^\infty$.

  Rewrite: $\mathbb{T} \sim \tau$.
- $\tau = \mathrm{no\_io}^n \cdot bio(v_o, v_i) \cdot \tau'$ for some n, bio, v_o, v_i, τ'.

  Case analysis on n.

  – $n = 0$.

    Case analysis on $\mathrm{safe}(h, \tau, P)$.

    ∗ SafeBio.
      Because of the premise of this inference rule and because $n = 0$:
      $h \xRightarrow{bio(v_o,v_i)} h'$ and $\mathrm{safe}(h', \tau', P)$ for some h'.
      Because $h \xRightarrow{bio(v_o,v_i)} h'$ we know that $h \xrightarrow{\epsilon^m} h''$ and
      $h'' \xrightarrow{bio(v_o,v_i)} h'$ for a certain m and h''.
      By definition of topetri: $\mathbb{T} = \epsilon^m \cdot bio(v_o, v_i) \cdot \mathrm{topetri}(\tau')$.
      Let $\mathbb{T}' = \mathrm{topetri}(\tau')$.
      Because of the coinductive hypothesis: $\mathbb{T}' \sim \tau'$.
      Because of the Bio inference rule (Fig. 3.4 on page 41):
      $\epsilon^m \cdot bio(v_o, v_i) \cdot \mathbb{T}' \sim bio(v_o, v_i) \cdot \tau'$.
      We can rewrite this to $\mathbb{T} \sim \tau$.

  - SafeContradict. Because of the premise of this inference rule, and because $n = 0$: $h \xRightarrow{bio(v_o,v_i')} h'$ for some v_i', h'.
    Because $h \xRightarrow{bio(v_o,v_i')} h'$ we know that $h \xrightarrow{\epsilon^m} h''$ and $h'' \xrightarrow{bio(v_o,v_i')} h'$ for a certain m and h''.
    By definition: $\mathbb{T} = \epsilon^m \cdot bio(v_o, v_i') \cdot \langle\rangle$.
    Because of the Contra inference rule (Fig. 3.4 on page 41):
    $\epsilon^m \cdot bio(v_o, v_i') \cdot \langle\rangle \sim bio(v_o, v_i) \cdot \tau'$.
    We can rewrite this to $\mathbb{T} \sim \tau$.
  - SafeNoIO. This case cannot occur.
  - SafePost. This case cannot occur.

- $n > 0$. Let $\tau_2 = \text{no\_io}^{n-1} \cdot bio(v_o, v_i) \cdot \tau'$. Note that $\tau = \text{no\_io} \cdot \tau_2$.
  Because safe(h, τ, P): safe(h, τ_2, P).
  By definition of topetri: $\mathbb{T} = \text{topetri}(\tau_2)$.
  The coinduction hypothesis states that $\mathbb{T} \sim \tau_2$.
  By applying the NoIO inference rule (Fig. 3.4 on page 41) we obtain $\mathbb{T} \sim \tau$.

□

Theorem 6

$\forall h, \tau, P.\ \text{safe}(h, \tau, P) \Rightarrow \exists \mathbb{T}.\ h \Downarrow \mathbb{T} \wedge \mathbb{T} \sim \tau$

Proof. Let $\mathbb{T} = \text{topetri}(\tau)$.

Because of Lemma 18 on page 167: $h \Downarrow \mathbb{T}$.

Because of Lemma 19 on the preceding page: $\mathbb{T} \sim \tau$. □

A.3 Weakest precondition implies safe

Lemma 20

$\forall h, \tau_1, \tau_2, P_1, P_2.\ (\text{safe}(h, \tau_1, P_1) \wedge (\forall h'.\ P_1(h') \Rightarrow \text{safe}(h', \tau_2, P_2))$
$\Rightarrow \text{safe}(h, \tau_1 \cdot \tau_2, P_2)$

Proof. Proof by coinduction. Case analysis on safe(h, τ_1, P_1).

- SafeBio.

  In this case, $h_1 \xRightarrow{bio(v_o,v_i)} h_2$ and $\text{safe}(h_2, \tau, P)$ for some h_2, bio, v_o, v_i.

  We want to apply the SafeBio inference rule to obtain $\text{safe}(h_1, \tau_1 \cdot \tau_2, P)$, so it suffices to prove $\text{safe}(h_2, \tau_1 \cdot \tau_2)$. This follows from the coinduction hypothesis.

- SafeNoIO, SafeContradict: similar.

- SafePost.

  In this case, $\tau_1 = \langle\rangle$ and $P_1(h)$.

  We also know $\forall h'.\ P_1(h') \Rightarrow \text{safe}(h', \tau_2, P_2)$ since it was given.

  Using this together with $P_1(h)$ we know $\text{safe}(h, \tau_2, P_2)$.

  Since $\tau_1 = \langle\rangle$ we obtain $\text{safe}(h, \tau_1 \cdot \tau_2, P_2)$.

□

Theorem 7

$$\forall c, Q, h, v, \tau.\ c \Downarrow \tau, v \wedge \lceil \text{wp}(c, \lceil Q \rceil) \rceil(h) \Rightarrow \text{safe}(h, \tau, Q(v))$$

Proof. Proof by coinduction. We perform (nested) induction on c.

- $c = v'$ for some v'.

  Because $c \Downarrow \tau, v$ we know $\tau = \langle\rangle$ and $v' = v$.

  We have to prove $\lceil Q \rceil(v)(h)$.

  This follows immediately from $\lceil \text{wp}(v', \lceil Q \rceil) \rceil(h)$.

- $c = \textbf{let } c_1 \textbf{ in } \mathcal{C}$ for some $c_1, \mathcal{C}$.

  Let $Q_1 = \lambda v.\, \text{wp}(\mathcal{C}(v), \lceil Q \rceil)$.

  Note that $\text{wp}(c, \lceil Q \rceil) = \text{wp}(\mathrm{c_1}, \mathrm{Q_1})$.

  Because $c \Downarrow \tau, v$ we know $\tau = \tau_1 \cdot \tau_2$ and $c_1 \Downarrow \tau_1, v_1$ and $\mathcal{C}(v_1) \Downarrow \tau_2, v$ for some τ_1, τ_2, v_1.

  The induction hypothesis states that
  $\forall h_1.\ \lceil \text{wp}(c_1, \lceil Q_1 \rceil) \rceil(h_1) \Rightarrow \text{safe}(h_1, \tau_1, Q_1(v_1))$ and
  $\forall h_2.\ \lceil \text{wp}(\mathcal{C}(v_1), \lceil Q \rceil) \rceil(h_2) \Rightarrow \text{safe}(h_2, \tau_2, Q(v))$.

  Because $\lceil \text{wp}(c, \lceil Q \rceil) \rceil(h)$ and $\text{wp}(c, \lceil Q \rceil) = \text{wp}(c_1, Q_1)$: $\lceil \text{wp}(c_1, Q_1) \rceil(h)$.

  Because wp is monotone: $\lceil \text{wp}(c_1, \lceil Q_1 \rceil) \rceil(h)$.

Combined with the first part of the induction hypothesis we obtain: $\text{safe}(h, \tau_1, Q_1(v_1))$.

Because of the second part of the induction hypothesis (note that $\text{wp}(\mathcal{C}(v_1), \lceil Q \rceil)(h_2) = Q_1(v_1)(h_2)$) we know that $\forall h_2.\ Q_1(v_1)(h_2) \Rightarrow \text{safe}(h_2, \tau_2, Q(v))$.

We now have all the ingredients to apply Lemma 20 on page 169 to obtain $\text{safe}(h, \tau_1 \cdot \tau_2, Q(v))$.

- $c = bio(v_o)$ for some bio, v_o.

  Because $c \Downarrow \tau, v$: $\tau = \langle bio(v_o, v) \rangle$.

  Because $\lceil \text{wp}(c, \lceil Q \rceil) \rceil(h)$ there is some v_i, h' such that $h \xRightarrow{bio(v_o,v_i)} h'$ and $\lceil Q \rceil(v_i)(h')$.

  Case analysis on whether $v_i = v$.

  - $v_i = v$.
    Because $\lceil Q \rceil(v_i)(h')$ we know $\text{safe}(h', \langle\rangle, Q(v))$.
    Therefore $\text{safe}(h, \tau, Q(v))$.
  - $v_i \neq v$. $\text{safe}(h, \tau, Q(v))$ follows directly.

  We have to prove $\text{safe}(h, \tau, Q(v))$.

- $c = f(\overline{v})$ for some $f, \overline{v}$.

  Becuase $c \Downarrow \tau, v$ we know $\text{fc}(f)(\overline{v}) \Downarrow \tau_1, v$ and $\tau = \text{no\_io} \cdot \tau_1$ for some τ_1.

  Because $\lceil \text{wp}(f(\overline{v}), \lceil Q \rceil) \rceil(h)$: $\lceil \text{wp}(\text{fc}(f)(\overline{v}, \lceil Q \rceil) \rceil(h)$.

  Use the SafeNoIO inference rule and the coinduction hypothesis to obtain $\text{safe}(c, \tau, Q(v))$.

□

A.4 Proven Hoare triple implies weakest precondition

We split wp in two parts: one for commands (wpc) and one for functions (wpf).

We define the set of function predicate transformers. A function predicate transformers returns a precondition given a program function, a list of arguments, and a postcondition.

Definition 97: FunPredTx

$$\text{FunPredTx} = \text{FuncNames} \to \text{Values}^* \to (\text{Values} \to \mathcal{P}(\text{Heaps})) \to \mathcal{P}(\text{Heaps})$$

We range over FunPredTx with *ptf*.

Definition 98: wpc

We define $\text{wpc}(\mathit{ptf}, c, Q)$ by recursion on c:

- $\text{wpc}(\mathit{ptf}, \textbf{let } c \textbf{ in } \mathcal{C}, Q) = \text{wpc}(\mathit{ptf}, c, \lambda v.\, \text{wpc}(\mathit{ptf}, \mathcal{C}(v), Q))$
- $\text{wpc}(\mathit{ptf}, f(\overline{v}), Q) = \mathit{ptf}(f, \overline{v}, Q)$
- $\text{wpc}(\mathit{ptf}, \mathit{bio}(v_o), Q) = \lambda h.\, \exists v_i, h'.\, h \xRightarrow{\mathit{bio}(v_o, v_i)} h' \wedge Q(v_i)(h')$
- $\text{wpc}(\mathit{ptf}, v, Q) = Q(v)$

We write $Q_1 \to Q_2$ to express that postcondition Q_2 is weaker than Q_2. More formally:

Definition 99: Weaker postcondition

$$Q_1 \to Q_2 \iff \forall v, h.\, Q_1(v)(h) \Rightarrow Q_2(v)(h)$$

We use the same notation to express a precondition is weaker than another:

Definition 100: Weaker precondition

$$P_1 \to P_2 \iff \forall h.\, P_1(h) \Rightarrow P_2(h)$$

To define *wpf*, we use Knaster-Tarski's fixpoint theorem. To do so, we define a function on a complete lattice, and some helper definitions.

We define an ordering on FunPredTx:

Definition 101

$\mathit{ptf_1} \leq \mathit{ptf_2} \Leftrightarrow \forall f, \overline{v}, Q.\ \mathit{ptf_1}(f, \overline{v}, Q) \rightarrow \mathit{ptf_2}(f, \overline{v}, Q)$

Definition 102: Monotonicity of function predicate transformers

We say a function predicate transformer *ptf* is monotone iff it allows weakening the postcondition. In other words:

$\forall f, \overline{v}, Q_1, Q_2.\ Q_1 \rightarrow Q_2 \Rightarrow \mathit{ptf}(f, \overline{v}, Q_1) \rightarrow \mathit{ptf}(f, \overline{v}, Q_2)$.

Definition 103: MonoFunPredTx

We define MonoFunPredTx as the set of function predicate transformers that are monotone.

Lemma 21

MonoFunPredTx is a complete lattice.

Proof. Similar to proof of Lemma 11 on page 160. □

Definition 104: Ff

We define a function Ff : FunPredTx → FunPredTx.

$\mathrm{Ff}(\mathit{ptf}) = \lambda f, \overline{v}, Q.\ \mathrm{wpc}(\mathit{ptf}, \mathrm{fc}(f)(\overline{v}), Q)$

Lemma 22

If *ptf* is monotone, then wpc(*ptf*) is monotone.

Proof. Proof by induction on *c*. □

Lemma 23

If $\mathit{ptf} \in \text{MonoFunPredTx}$, then $\text{Ff}(\mathit{ptf}) \in \text{MonoFunPredTx}$.

Proof. Use Lemma 22 on the preceding page. □

Lemma 24

Ff is monotone in MonoFunPredTx, i.e. $\forall \mathit{ptf_1}, \mathit{ptf_2} \in \text{MonoFunPredTx}.\ \mathit{ptf_1} \leq \mathit{ptf_2} \Rightarrow \text{Ff}(\mathit{ptf_1}) \leq \text{Ff}(\mathit{ptf_2})$.

Proof. Given $\text{Ff}(\mathit{ptf_1})(f, \overline{v}, Q)(h)$ we have to prove $\text{Ff}(\mathit{ptf_2})(f, \overline{v}, Q)(h)$.

Because $\text{Ff}(\mathit{ptf_1})(f, \overline{v}, Q)(h)$: $\text{wpc}(\mathit{ptf_1}, \text{fc}(f)(\overline{v}), Q)(h)$.

Case analysis on $\text{fc}(f)(\overline{v})$.

- $\text{fc}(f)(\overline{v}) = f'(\overline{v'})$.
  Because $\text{wpc}(\mathit{ptf_1}, \text{fc}(f)(\overline{v}), Q)(h)$: $\mathit{ptf_1}(f', \overline{v'}, Q)(h)$.
  Because $\mathit{ptf_1} \leq \mathit{ptf_2}$: $\mathit{ptf_2}(f', \overline{v'}, Q)(h)$
  Therefore: $\text{wpc}(\mathit{ptf_2}, f'(\overline{v'}), Q)(h)$.
  Rewrite: $\text{wpc}(\mathit{ptf_2}, \text{fc}(f, \overline{v}), Q)(h)$.
  Therefore: $\text{Ff}(\mathit{ptf_2}, f, \overline{v})(h)$.
- Other cases: follows directly from definition of Ff and wpc.

□

Now that we have a complete lattice and a monotone function Ff on that lattice, we can use Knaster-Tarski to obtain that Ff has a unique greatest fixpoint. We define *wpf* as this fixpoint:

Definition 105: *wpf*

We define *wpf* as the greatest fixpoint of Ff.

Note that $\text{Ff}(\mathit{wpf}) = \mathit{wpf}$.

Now that we have split up wp into wpc and wpf, we show that in order to obtain $\text{wp}(c, Q)(h)$, it suffices to obtain $\text{wpc}(\text{wpf}, c, Q)(h)$.

Lemma 25

$\forall c, Q.\ \mathrm{wpc}(\mathrm{wpf}, c, Q) \rightarrow \mathrm{wp}(c, Q)$

Proof. Choose arbitrary h such that $\mathrm{wpc}(\mathrm{wpf}, c, Q)(h)$. According to Definition 90 on page 162, we have to identify a *ptx* that satisfies some properties. We choose $ptx = \mathrm{wpc}(\mathrm{wpf})$. We prove the following properties:

- wpc(wpf) must be monotone. This follows from Lemma 22 on page 173. wpf is monotone because it is an element in MonoFunPredTx.
- $(\mathrm{wpc}(\mathrm{wpf}))(c, Q)(h)$: given.
- $\forall c', Q', h'.\ (\mathrm{wpc}(\mathrm{wpf}))(c', Q')(h') \Rightarrow \mathrm{F}(\mathrm{wpc}(\mathrm{wpf}))(c', Q')(h')$.

  Note that we use F as defined in Definition 89 on page 161.

  Case analysis on c'.

  - $c' = f(\overline{v})$.
    It suffices to show that (unfold F): $(\mathrm{wpc}(\mathrm{wpf}))(\mathrm{fc}(f)(\overline{v}), Q')(h')$.
    Because $(\mathrm{wpc}(\mathrm{wpf}))(c', Q')(h')$: $\mathrm{wpf}(f, \overline{v}, Q')(h')$.
    Because $\mathrm{wpf} = \mathrm{Ff}(\mathrm{wpf})$: $\mathrm{wpc}(\mathrm{wpf}, \mathrm{fc}(f)(\overline{v}), Q')(h')$.
  - Other cases: follows directly from definition of wpc and F.

□

Definition 106

$\mathrm{fr}(Q, h_F) = \lambda v, h.\ \exists h'.\ h = h' \uplus h_F \wedge Q(v)(h')$

We define a new notation: $\mathring{P}$ is the set of heaps that satisfy the assertion P.

Definition 107

$\mathring{P} = (\lambda h.\ h \vDash P)$

We use a similar notation for postconditions:

Definition 108

$\mathring{Q} = (\lambda v, h.\, h \vDash Q(v))$

We define a function predicate transformer that intuitively performs a lookup in the table of user-written preconditions and postconditions (fC), instead of looking at the body of the function.

Definition 109: prfptf

$$\begin{aligned}&\mathrm{prfptf}(f, \overline{v}, Q) = \lambda h.\, \exists P', Q', h', h_F.\\ &\quad h = h' \uplus h_F \wedge \lceil \mathring{P}' \rceil(h') \wedge (P', Q') \in \mathrm{fC}(f)(\overline{v}) \wedge\\ &\quad (\forall v, h''.\, h'' \vDash Q'(v) \Rightarrow Q(v)(h'' \uplus h_F))\\ &\quad \wedge Q = \lceil Q \rceil\end{aligned}$$

Lemma 26

prfptf is monotone.

Proof. Follows easily from definition of prfptf. □

Definition 110: wpcp

We define wpcp(ptf, c, Q) by recursion on c.

- wpcp$(\mathit{ptf}, \textbf{let } c \textbf{ in } \mathcal{C}, Q) = \mathrm{wpcp}(\mathit{ptf}, c, \lambda v.\, \mathrm{wpcp}(\mathit{ptf}, \mathcal{C}(v), Q))$
- wpcp$(\mathit{ptf}, f(\overline{v}), Q) = \lceil \mathit{ptf}(f, \overline{v}, \lceil Q \rceil) \rceil$
- wpcp$(\mathit{ptf}, \mathit{bio}(v_o), Q) = \lambda h.\, \exists v_i, h'.\, h \xRightarrow{\mathit{bio}(v_o, v_i)} h' \wedge \lceil Q(v_i) \rceil(h')$
- wpcp$(\mathit{ptf}, v, Q) = Q(v)$

The difference between wpcp and wpc is that wpcp allows epsilon-steps after a BIO while wpc does not, and wpcp allows epsilon steps before and after function calls.

Lemma 27: Monotonicity of wpcp

If a given *ptf* is monotone, then wpcp(*ptf*) is monotone. In other words:

$\forall ptf, c, Q_1, Q_2.\ ptf \in \text{MonoFunPredTx} \wedge Q_1 \rightarrow Q_2 \Rightarrow \text{wpcp}(ptf, c, Q_1) \rightarrow \text{wpcp}(ptf, c, Q_2)$

Proof. Proof by induction on c. □

Lemma 28

$\forall P, P'.\ (P \Rrightarrow P') \Rightarrow (\mathring{P} \rightarrow \lceil \mathring{P'} \rceil)$

Proof. Choose arbitrary h such that $\mathring{P}(h)$, i.e. $h \vDash P$.

Induction on $P \Rrightarrow P'$.

- $P \rightarrow P'$. Trivial.
- $P = P_1 * R$, $P' = P_1' * R$, $P_1 \Rrightarrow P_1'$.

  Because $h \vDash P$ there is some h_1, h_R such that $h = h_1 \uplus h_R$, $h_1 \vDash P_1$, and $h_R \vDash R$.

  Because of the induction hypothesis: $\mathring{P_1} \rightarrow \lceil \mathring{P_1'} \rceil$.

  Therefore: $\lceil \mathring{P_1'} \rceil(h_1)$.

  So there is some h_1' such that $h_1 \xrightarrow{\epsilon}^* h_1'$ and $h_1' \vDash P_1'$.

  So we also have $h_1 \uplus h_R \xrightarrow{\epsilon}^* h_1' \uplus h_R$ and $h_1' \uplus h_R \vDash P_1' * R$.

  Therefore, $\lceil P_1' \mathring{*} R \rceil(h_1 \uplus h_R)$, which we can rewrite to $\lceil \mathring{P'} \rceil(h)$.
- $P = \textbf{token}(t_1) * \textbf{split}(t_1, t_2, t_3)$, $P' = \textbf{token}(t_2) * \textbf{token}(t_3)$.

  Because $h \vDash P$: $h = \{\!\{\textbf{token}(t_1), \textbf{split}(t_1, t_2, t_3)\}\!\}$

  Let $h' = \{\!\{\textbf{token}(t_2), \textbf{token}(t_3)\}\!\}$.

  We know $h \xrightarrow{\epsilon}^* h'$ and $\mathring{P'}(h')$.

  Therefore $\lceil \mathring{P'} \rceil(h)$.
- Other cases: similar.

□

Lemma 29

$\forall Q, Q', h_F.\ Q \to Q' \Rightarrow \mathrm{fr}(\lceil Q \rceil, h_F) \to \mathrm{fr}(\lceil Q' \rceil, h_F)$

Proof. Choose arbitrary v, h such that $\mathrm{fr}(\lceil Q \rceil, h_F)(v)(h)$.

By definition of fr there exists some h_Q such that $h = h_Q \uplus h_F$ and $\lceil Q \rceil(v)(h_Q)$.

Since $Q \to Q'$, we also know $\lceil Q \rceil \to \lceil Q' \rceil$.

Because of the latter and $\lceil Q \rceil(v)(h_Q)$: $\lceil Q' \rceil(v)(h_Q)$.

By definition of fr: $\mathrm{fr}(\lceil Q' \rceil, h_F)(v)(h)$ □

Lemma 30

$\mathrm{wpcp}(\mathrm{prfptf}, c, \lceil Q \rceil) \to \lceil \mathrm{wpcp}(\mathrm{prfptf}, c, Q) \rceil$

Proof. Induction on c. For each case, choose arbitrary h such that $\mathrm{wpcp}(\mathrm{prfptf}, c, \lceil Q \rceil)(h)$.

- $c =$ **let** c' **in** $\mathcal{C}$. It suffices to prove that $\lceil \mathrm{wpcp}(\mathrm{prfptf}, c', \lambda v.\, \mathrm{wpcp}(\mathrm{prfptf}, \mathcal{C}(v), Q)) \rceil(h)$.

  Because $\mathrm{wpcp}(\mathrm{prfptf}, c, \lceil Q \rceil)(h)$:
  $\mathrm{wpcp}(\mathrm{prfptf}, c', \lambda v.\, \mathrm{wpcp}(\mathrm{prfptf}, \mathcal{C}(v), \lceil Q \rceil))(h)$.

  Because of the first induction hypothesis, Lemma 27 on the preceding page and Lemma 26 on page 176:
  $\mathrm{wpcp}(\mathrm{prfptf}, c', \lceil \lambda v.\, \mathrm{wpcp}(\mathrm{prfptf}, \mathcal{C}(v), Q) \rceil)(h)$.

  Because of the second induction hypothesis:
  $\lceil \mathrm{wpcp}(\mathrm{prfptf}, c', \lambda v.\, \mathrm{wpcp}(\mathrm{prfptf}, \mathcal{C}(v), Q)) \rceil(h)$.

  Because of the definition of wpcp:
  $\lceil \mathrm{wpcp}(\mathrm{prfptf}, c, Q) \rceil(h)$.

- Other cases are trivial or follow immediately from the induction hypothesis.

□

Lemma 31

$$\forall h_R, h_F, R, Q.\ h_R \vDash R \wedge \mathrm{fr}(\lceil \mathring{Q} \rceil, h_R \uplus h_F) \to \mathrm{fr}(\lceil \lambda v, h.\, h \vDash Q(v) * R \rceil, h_F)$$

Proof. Choose arbitrary h, v such that $\mathrm{fr}(\lceil \mathring{Q} \rceil, h_R \uplus h_F)(v)(h)$.

Therefore, there is some h_Q such that $h = h_Q \uplus h_R \uplus h_F$ and $\lceil \mathring{Q} \rceil(v)(h_Q)$.

Because of the latter, there is some h'_Q such that $h_Q \xrightarrow{\epsilon}^* h'_Q$ and $h'_Q \vDash Q(v)$.

Therefore, $h'_Q \uplus h_R \vDash Q(v) * R$.

Also, $h_Q \uplus h_R \xrightarrow{\epsilon}^* h'_Q \uplus h_R$

So, $\lceil \lambda v, h.\, h \vDash Q(v) * R \rceil(v)(h_Q \uplus h_R)$.

To show $\mathrm{fr}(\lceil \lambda v, h.\, h \vDash Q(v) * R \rceil, h_F)(v)(h)$ we need to identify an h'' such that $h = h'' \uplus h_F$ and $\lceil \lambda v, h.\, h \vDash Q(v) * R \rceil(v)(h'')$. This holds for $h'' = h_Q \uplus h_R$. □

Lemma 32

$$\forall P, c, Q, h.\ \vdash \{P\}\, c\, \{Q\} \wedge h \vDash P \Rightarrow \forall h_F.$$
$$\lceil \mathrm{wpcp}(\mathrm{prfptf}, c, \mathrm{fr}(\lceil \mathring{Q} \rceil, h_F)) \rceil(h \uplus h_F)$$

Proof. Proof by induction on $\vdash \{P\}\, c\, \{Q\}$.

- Val, Disj, Bio and Exists proof rule. Trivial.
- Rewrite. $P \Rrightarrow P'$, $\forall v.\ Q'(v) \Rrightarrow Q(v)$ and $\vdash \{P'\}\, c\, \{Q'\}$ for some P', Q'.

  Choose arbitrary h, h_F such that $h \vDash P$ (and therefore $\mathring{P}(h)$).

  Apply Lemma 28 on page 177 to obtain $\mathring{P} \to \lceil \mathring{P}' \rceil$.

  So we also know $\lceil \mathring{P}' \rceil(h)$ (and therefore $h \vDash P'$).

  Therefore there is some h' such that $h \xrightarrow{\epsilon}^* h'$ and $h' \vDash P'$.

  So we can apply the induction hypothesis to obtain $\lceil \mathrm{wpcp}(\mathrm{prfptf}, c, \mathrm{fr}(\lceil \mathring{Q}' \rceil, h_F)) \rceil(h' \uplus h_F)$.

  Because $h \xrightarrow{\epsilon}^* h'$ (and therefore $h \uplus h_F \xrightarrow{\epsilon}^* h' \uplus h_F$) we also know $\lceil \mathrm{wpcp}(\mathrm{prfptf}, c, \mathrm{fr}(\lceil \mathring{Q}' \rceil, h_F)) \rceil(h \uplus h_F)$.

Now apply Lemma 28 on page 177 to obtain $\forall v.\ \mathring{Q}(v) \rightarrow \lceil \mathring{Q'}(v) \rceil$.

So we also know $\mathring{Q} \rightarrow \lceil \mathring{Q}' \rceil$.

Apply Lemma 29 on page 178 to obtain $\mathrm{fr}(\lceil \mathring{Q} \rceil, h_F) \rightarrow \mathrm{fr}(\lceil \mathring{Q}' \rceil, h_F)$.

Apply Lemma 27 on page 177 (and Lemma 26 on page 176) to obtain: $\lceil \mathrm{wpcp}(\mathrm{prfptf}, c, \mathrm{fr}(\lceil \mathring{Q} \rceil, h_F)) \rceil (h \uplus h_F)$.

- App. Choose arbitrary h_F. Because of the definition of wpcp, we have to prove $\lceil \mathrm{prfptf}(f, \overline{v}, \lceil \mathrm{fr}(\lceil \mathring{Q} \rceil, h_F) \rceil) \rceil (h \uplus h_F)$.

  It is sufficient to prove that $\mathrm{prfptf}(f, \overline{v}, \lceil \mathrm{fr}(\lceil \mathring{Q} \rceil, h_F) \rceil)(h \uplus h_F)$.

  We have to identify a P', Q', h', h'_F such that $h \uplus h_F = h' \uplus h'_F$ and $\lceil \mathring{P'} \rceil (h')$ and $(P', Q') \in \mathrm{fC}(f)(\overline{v})$ and $(\forall v, h''.\ h'' \vDash Q'(v) \Rightarrow \lceil \mathrm{fr}(\lceil \mathring{Q} \rceil, h_F) \rceil (v)(h'' \uplus h'_F))$.

  We choose $P' = P, Q' = Q, h' = h$, and $h'_F = h_F$. All conjuncts follow immediately.

- Let. $c = \textbf{let}\ c'\ \textbf{in}\ \mathcal{C}$ for some c', $\mathcal{C}$.

  Choose arbitrary h such that $h \vDash P$. Choose arbitrary h_F.

  The induction hypothesis states that

  $$\lceil \mathrm{wpcp}(\mathrm{prfptf}, c', \mathrm{fr}(\lceil \mathring{Q}_1 \rceil, h_F)) \rceil (h \uplus h_F)$$

  and

  $$\forall v, h_Q.\ h_Q \vDash Q_1(v) \Rightarrow \lceil \mathrm{wpcp}(\mathrm{prfptf}, \mathcal{C}(v), \mathrm{fr}(\lceil \mathring{Q} \rceil, h_F)) \rceil (h_Q \uplus h_F)$$

  for some Q_1.

  It suffices to prove that $\lceil \lceil \mathrm{wpcp}(\mathrm{prfptf}, c', \lambda v.\ \mathrm{wpcp}(\mathrm{prfptf}, \mathcal{C}(v), \mathrm{fr}(\lceil \mathring{Q} \rceil, h_F))) \rceil \rceil (h \uplus h_F)$.

  Because of Lemma 30 on page 178 it suffices to prove that $\lceil \mathrm{wpcp}(\mathrm{prfptf}, c', \lceil \lambda v.\ \mathrm{wpcp}(\mathrm{prfptf}, \mathcal{C}(v), \mathrm{fr}(\lceil \mathring{Q} \rceil, h_F)) \rceil) \rceil (h \uplus h_F)$.

  Because of the first part of the induction hypothesis, Lemma 27 on page 177 and Lemma 26 on page 176, it suffices to prove that $\mathrm{fr}(\lceil \mathring{Q}_1 \rceil, h_F) \rightarrow \lceil \lambda v.\ \mathrm{wpcp}(\mathrm{prfptf}, \mathcal{C}(v), \mathrm{fr}(\lceil \mathring{Q} \rceil, h_F)) \rceil$.

  Choose arbitrary h', v such that $\mathrm{fr}(\lceil \mathring{Q}_1 \rceil, h_F)(v)(h')$. We want to show $\lceil \lambda v.\ \mathrm{wpcp}(\mathrm{prfptf}, \mathcal{C}(v), \mathrm{fr}(\lceil \mathring{Q} \rceil, h_F)) \rceil (v)(h')$.

  Because $\mathrm{fr}(\lceil \mathring{Q}_1 \rceil, h_F)(v)(h')$ there is some h_1, h_2 such that $h' = h_1 \uplus h_F$ and $h_1 \xrightarrow{\epsilon}{}^{*} h_2$ and $h_2 \vDash Q_1(v)$.

  Because of the second part of the induction hypothesis: $\lceil \mathrm{wpcp}(\mathrm{prfptf}, \mathcal{C}(v), \mathrm{fr}(\lceil \mathring{Q} \rceil, h_F)) \rceil (h_2 \uplus h_F)$.

  Because also $h_1 \uplus h_F \xrightarrow{\epsilon}{}^{*} h_2 \uplus h_F$ and $h' = h_1 \uplus h_F$: $\lceil \mathrm{wpcp}(\mathrm{prfptf}, \mathcal{C}(v), \mathrm{fr}(\lceil \mathring{Q} \rceil, h_F)) \rceil (h')$.

- Frame. $P = P' * R, Q = \lambda v.\, Q'(v) * R$ for some P', Q'.

  Choose arbitrary h such that $h \vDash P$. Therefore, there is some h'_P, h_R such that $h = h'_P \uplus h_R$ and $h'_P \vDash P'$ and $h_R \vDash R$.

  Choose arbitrary h_F.

  Because of the induction hypothesis:
  $\lceil \text{wpcp}(\text{prfptf}, c, \text{fr}(\lceil \mathring{Q}' \rceil, h_R \uplus h_F)) \rceil (h'_P \uplus (h_R \uplus h_F))$.

  Because of Lemma 31 on page 179 we know
  $\text{fr}(\lceil \mathring{Q}' \rceil, h_R \uplus h_F) \to \text{fr}(\lceil \lambda v, h.\, h \vDash Q'(v) * R \rceil, h_F)$.

  So we can use monotonicity of wpcp(prfptf) (Lemma 27 on page 177 and Lemma 26 on page 176) to obtain:

  $$\lceil \text{wpcp}(\text{prfptf}, c, \text{fr}(\lceil \lambda v, h.\, h \vDash Q'(v) * R \rceil, h_F)) \rceil (h'_P \uplus (h_R \uplus h_F))$$

  We can rewrite this to $\lceil \text{wpcp}(\text{prfptf}, c, \text{fr}(\lceil \mathring{Q} \rceil, h_F)) \rceil (h \uplus h_F)$

□

Lemma 33

$\forall c, Q.\ \lceil \text{wpcp}(\text{prfptf}, c, Q) \rceil \to \text{wpc}(\text{prfptf}, c, \lceil Q \rceil)$

Proof. Induction on c.

- $c = \textbf{let}\ c'\ \textbf{in}\ \mathcal{C}$. The induction hypothesis states that

  $$\forall Q'.\ \lceil \text{wpcp}(\text{prfptft}, c', Q') \rceil \to \text{wpc}(\text{prfptf}, c', \lceil Q' \rceil)$$

  and

  $$\forall Q', v.\ \lceil \text{wpcp}(\text{prfptf}, \mathcal{C}(v), Q') \rceil \to \text{wpc}(\text{prfptf}, \mathcal{C}(v), \lceil Q' \rceil)$$

  Choose arbitrary h such that $\lceil \text{wpcp}(\text{prfptf}, c, Q) \rceil (h)$.

  Because of the definition of wpcp:
  $\lceil \text{wpcp}(\text{prfptf}, c', \lambda v.\, \text{wpcp}(\text{prfptf}, \mathcal{C}(v), Q)) \rceil (h)$.

  Because of the first part of the induction hypothesis:
  $\text{wpc}(\text{prfptf}, c', \lambda v.\, \lceil \text{wpcp}(\text{prfptf}, \mathcal{C}(v), Q) \rceil)(h)$.

  Because of the second part of the induction hypothesis, Lemma 22 on page 173 and Lemma 26 on page 176:
  $\text{wpc}(\text{prfptf}, c', \lambda v.\, \text{wpc}(\text{prfptf}, \mathcal{C}(v), \lceil Q \rceil))(h)$.

  Because of the definition of wpc: $\text{wpc}(\text{prfptf}, c, \lceil Q \rceil)(h)$.

- Other cases: trivial.

□

Lemma 34

If all function bodies are proven, then prfptf $\leq$ F(prfptf)

Proof. Given $\text{prfptf}(f, \overline{v}, Q)(h)$ for some $f, \overline{v}, Q, h$, we have to prove $\text{wpc}(\text{prfptf}, \text{fc}(f)(\overline{v}), Q)(h)$.

Because $\text{prfptf}(f, \overline{v}, Q)(h)$ we know $Q = \lceil Q \rceil$. So it suffices to prove that $\text{wpc}(\text{prfptf}, \text{fc}(f)(\overline{v}), \lceil Q \rceil)(h)$.

Because Lemma 33 on the previous page and because $Q = \lceil Q \rceil$, it suffices to prove that $\lceil \text{wpcp}(\text{prfptf}, \text{fc}(f)(\overline{v}), \lceil Q \rceil) \rceil(h)$.

Because $\text{prfptf}(f, \overline{v}, Q)(h)$: There is some P', Q', h', h_F such that $h = h' \uplus h_F$ and $\lceil \mathring{P}' \rceil(h')$ and $(P', Q') \in \text{fC}(f)(\overline{v})$ and

$$\forall v, h''.\ h'' \vDash Q'(v) \Rightarrow Q(v)(h'' \uplus h_F) \tag{A.1}$$

Because $\lceil \mathring{P}' \rceil(h')$ there is some h_ϵ such that $h' \overset{\epsilon}{\rightarrow}^* h_\epsilon$ and $P' \vDash h_\epsilon$.

Because all function bodies are proven we can apply Lemma 32 on page 179 to obtain
$\lceil \text{wpcp}(\text{prfptf}, \text{fc}(f)(\overline{v}), \text{fr}(\lceil \mathring{Q}' \rceil, h_F) \rceil(h_\epsilon \uplus h_F)$.

Therefore, $\lceil \text{wpcp}(\text{prfptf}, \text{fc}(f)(\overline{v}), \text{fr}(\lceil \mathring{Q}' \rceil, h_F) \rceil(h)$.

Because of monotonicity of wpcp(prfptf) (Lemma 27 on page 177 and Lemma 26 on page 176) it suffices to prove that
$\forall v, h''.\ \text{fr}(\lceil \mathring{Q}' \rceil, h_F)(v)(h'') \Rightarrow \lceil Q \rceil(v)(h'')$.

Choose arbitrary v, h'' such that $\text{fr}(\lceil \mathring{Q}' \rceil, h_F)(v)(h'')$.

Because $\text{fr}(\lceil \mathring{Q}' \rceil, h_F)(v)(h'')$: there is some h_1, h_2 such that
$h'' = h_1 \uplus h_F \wedge h_1 \overset{\epsilon}{\rightarrow}^* h_2 \wedge h_2 \vDash Q'(v)$.

Because (A.1): $Q(v)(h_2 \uplus h_F)$.

Because $h_1 \overset{\epsilon}{\rightarrow}^* h_2$: $h_1 \uplus h_F \overset{\epsilon}{\rightarrow}^* h_2 \uplus h_F$ and hence $\lceil Q \rceil(h_1 \uplus h_F)$.

Therefore, $\lceil Q \rceil(h'')$. □

Lemma 35

If all function bodies are proven, then prfptf $\leq$ wpf

Proof. wpf is the greatest fixpoint of F. We use Knaster-Tarski to obtain that wpf is equal to the least upper bound of the postfixpoints of F.

According to Lemma 34 on the preceding page, ptrptf is a postfixpoint of F, so prfptf $\leq$ wpf. □

Theorem 8

If all function bodies are proven, then

$$\forall P, c, Q, h.\ \vdash \big\{P\big\}\, c\, \big\{Q\big\} \wedge h \vDash P \Rightarrow \Big\lceil \mathrm{wp}(c, \lceil \lambda v, h'.\, h' \vDash Q(v) \rceil) \Big\rceil (h)$$

Proof. Because of Lemma 25 on page 175 it suffices to prove that $\lceil \mathrm{wpc}(\mathrm{wpf}, c, \lceil \mathring{Q} \rceil) \rceil (h)$.

Because of Lemma 32 on page 179 we know $\lceil \mathrm{wpcp}(\mathrm{prfptf}, c, \lceil \mathring{Q} \rceil) \rceil (h)$.

Apply Lemma 33 on page 181 to obtain $\lceil \mathrm{wpc}(\mathrm{prfptf}, c, \lceil \mathring{Q} \rceil) \rceil (h)$.

Because of Lemma 35 and the definition of wpc we know $\lceil \mathrm{wpc}(\mathrm{wpf}, c, \lceil \mathring{Q} \rceil) \rceil (h)$. □

Appendix B

Proofs in-memory I/O

B.1 Soundness proof in-memory I/O

Lemma 36: Safe interleaving

$$\begin{array}{l} \forall h_A, \tau_A, Q_A, h_B, \tau_B, Q_B, \Lambda, I. \\ \quad \text{safe}(\Lambda, h_A, \tau_A, I, Q_A) \wedge \\ \quad \text{safe}(\Lambda, h_B, \tau_B, I, Q_B) \wedge \\ \quad \tau \in \tau_A \,||\, \tau_B \Rightarrow \\ \quad \text{safe}(\Lambda, h_A \uplus h_B, \tau, I, Q_A * Q_B). \end{array}$$

Proof. We perform well-founded induction on Λ. The induction hypothesis states

$$\begin{array}{l} \forall \Lambda' < \Lambda.\ \forall h_A, \tau_A, Q_A, h_B, \tau_B, Q_B. \\ \quad \text{safe}(\Lambda', h_A, \tau_A, I, Q_A) \wedge \\ \quad \text{safe}(\Lambda', h_B, \tau_B, I, Q_B) \wedge \\ \quad \tau \in \tau_A \,||\, \tau_B \Rightarrow \\ \quad \text{safe}(\Lambda', h_A \uplus h_B, \tau, I, Q_A * Q_B). \end{array}$$

We have to prove $\text{safe}(\Lambda, h_A \uplus h_B, \tau, I, Q_A * Q_B)$.

Choose arbitrary $\Lambda' \leq \Lambda$. We want to prove $\text{safe}'(\Lambda', h_A \uplus h_B, \tau, I, Q_A * Q_B)$. Assume $\Lambda'.I > 0$ (the case $\Lambda'.I = 0$ is trivial).

We perform case analysis on τ.

- $\tau = \epsilon$.

  Then $\tau_A = \tau_B = \epsilon$.

  Because of $\text{safe}'(\Lambda', h_A, \epsilon, I, Q_A)$ we know
  $h_A = h_{rA} \uplus h_{QA} \wedge h_{QA} \vDash_{F_{\Lambda'}} Q_A$ for some h_{rA}, h_{QA}.

  Analogous, $h_B = h_{rB} \uplus h_{QB} \wedge h_{QB} \vDash_{F_{\Lambda'}} Q_B$ for some h_{rB}, h_{QB}.

  Let $h_r = h_{rA} \uplus h_{rB}$.

  Let $h_Q = h_{QA} \uplus h_{QB}$.

  Combining $h_A = h_{rA} \uplus h_{QA}$ and $h_B = h_{rB} \uplus h_{QB}$ we obtain
  $h_A \uplus h_B = h_{rA} \uplus h_{QA} \uplus h_{rB} \uplus h_{QB} = h_r \uplus h_Q$ (which we want to prove).

  $h_Q \vDash_{F_{\Lambda'}} Q_A * Q_B$ (which we want to prove) because $h_{QA} \vDash_{F_{\Lambda'}} Q_A$ and $h_{QB} \vDash_{F_{\Lambda'}} Q_B$.

- $\tau = (h_1, \bot) \cdot \tau'$ for some h_1, τ'.

  This case does not occur because then $\neg\text{safe}'(\Lambda', h_A, \tau_A, I, Q_A)$ or $\neg\text{safe}'(\Lambda', h_B, \tau_B, I, Q_B)$ (remember $\Lambda'.I \neq 0$).

- $\tau = (h_1, h_2) \cdot \tau'$ for some h_1, h_2, τ'.

  Assume $\tau_A = (h_1, h_2) \cdot \tau'_A$ for some τ'_A (the other case is analogous).

  Choose arbitrary h_r, h_I such that
  $h_1 = \text{erat}((h_A \uplus h_B) \uplus h_I \uplus h_r) \wedge h_I \vDash_{F_{\Lambda'}} \circledast\text{inv}(h_A \uplus h_B \uplus h_I \uplus h_r, I)$.

  Let $h_{rA} = h_B \uplus h_r$.

  Then $h_1 = \text{erat}(h_A \uplus h_I \uplus h_{rA})$.

  Combining this with $\text{safe}'(\Lambda', h_A, \tau_A, I, Q_A)$ we know there exists some h'_A, h'_I such that $h_2 = \text{erat}(h'_A \uplus h'_I \uplus h_{rA})$, and
  $h'_I \vDash_{F_{\Lambda'}} \circledast\text{inv}(h'_A \uplus h'_I \uplus h_{rA}, I)$ and
  $I = \text{inat} \Rightarrow \text{inv}((h_A \uplus h_B) \uplus h_I \uplus h_r, \text{outat}) = \text{inv}(h'_A \uplus h'_I \uplus h_{rA}, \text{outat})$
  (which we all three want to prove) and $\text{safe}(\Lambda' -_I 1, h'_A, \tau'_A, I, Q_A)$.

  Let $h' = h'_A \uplus h_B$.

  Then we have the following equality (which we wanted to prove):
  $h_2 = \text{erat}(h'_A \uplus h'_I \uplus h_{rA})$ (obtained earlier)
  $= \text{erat}(h'_A \uplus h'_I \uplus h_r \uplus h_B)$ (by definition of h_{rA})
  $= \text{erat}(h' \uplus h'_I \uplus h_r)$ (by definition of h').

  Next we need to prove $\text{safe}(\Lambda' -_I 1, h', \tau', I, Q_A * Q_B)$.

  We already know $\text{safe}(\Lambda' -_I 1, h'_A, \tau'_A, I, Q_A)$. We also know $\text{safe}(\Lambda, h_B, \tau_B, I, Q_B)$ and therefore $\text{safe}(\Lambda' -_I 1, h_B, \tau_B, I, Q_B)$ (use $\Lambda' -_I 1 < \Lambda$). Now we can apply the induction hypothesis.

- $\tau = (h_1, \mathbf{inf}) \cdot \tau'$ for some h_1, τ'. This case follows directly from the definition of safe$'$.

$\square$

Lemma 37: Safe seq

$\forall \Lambda, h, \tau_A, I, Q_1, Q_2.$
$\big(\text{safe}(\Lambda, h, \tau_A, I, Q_1) \wedge (\forall \Lambda. \forall h_Q. h_Q \vDash_{F_m} Q_1 \Rightarrow \text{safe}(\Lambda, h_Q, \tau_B, I, Q_2))\big)$
$\Rightarrow \text{safe}(\Lambda, h, \tau_A; \tau_B, I, Q_2).$

Proof. Choose arbitrary Λ, I. We perform well-founded induction on Λ. The induction hypothesis states that

$\forall \Lambda' < \Lambda.$
$\forall h, \tau_A, Q_1, Q_2.$
$\text{safe}(\Lambda', h, \tau_A, I, Q_1) \wedge \big(\forall \Lambda. \forall h_Q.\ h_Q \vDash_{F_m} Q_1 \Rightarrow \text{safe}(\Lambda, h_Q, \tau_B, I, Q_2)\big)$
$\Rightarrow \text{safe}(\Lambda', h, \tau_A; \tau_B, I, Q_2)$

Given

$$\text{safe}(\Lambda, h, \tau_A, I, Q_1) \tag{B.1}$$

and

$$\forall \Lambda. \forall h_Q.\ h_Q \vDash_{F_m} Q_1 \Rightarrow \text{safe}(\Lambda, h_Q, \tau_B, I, Q_2) \tag{B.2}$$

we want to prove $\text{safe}(\Lambda, h, \tau_A; \tau_B, I, Q_2)$.

Choose arbitrary $\Lambda' \leq \Lambda$. We want to prove $\text{safe}'(\Lambda', h, \tau_A; \tau_B, I, Q_2)$. In case $\Lambda' = (0,0)$ this is trivial, so we assume $\Lambda' > (0,0)$.

Let $\tau_C = \tau_A; \tau_B$.

We perform case analysis on τ_A and τ_B.

- $\tau_A = (h_1, h_2) \cdot \tau'_A$ for some h_1, h_2, τ'_A.

  Therefore $\tau_C = (h_1, h_2) \cdot (\tau'_A; \tau_B)$.

  Choose arbitrary h_r, h_I such that $h_1 = \text{erat}(h \uplus h_I \uplus h_r)$ and $h_I \vDash_{F_{\Lambda'}} \circledast \text{inv}(h \uplus h_I \uplus h_r, I)$.

  Because (B.1) there is some h', h'_I such that $h_2 = \text{erat}(h' \uplus h'_I \uplus h_r)$ and $h'_I \vDash_{F_{\Lambda'}} \circledast \text{inv}(h' \uplus h'_I \uplus h_r, I)$ and $I = \text{inat} \Rightarrow \text{inv}(h \uplus h_I \uplus h_r, \text{outat}) = \text{inv}(h' \uplus h'_I \uplus h_r, \text{outat})$ (which we all wanted to prove) and

$$\text{safe}(\Lambda' -_I 1, h', \tau'_A, I, Q_1) \tag{B.3}$$

Using (B.3) and (B.2) we can apply the induction hypothesis to obtain $\text{safe}(\Lambda' -_I 1, h', \tau'_A; \tau_B, I, Q_1)$ (which we wanted to prove).

- $\tau_A = \epsilon \wedge \tau_B = (h_1, h_2) \cdot \tau'_B$ for some h_1, h_2, τ'_B. Then $\tau_C = (h_1, h_2) \cdot \tau'_B$. Choose arbitrary h_{rC}, h_{IC} such that

$$h_1 = \text{erat}(h \uplus h_{IC} \uplus h_{rC}) \tag{B.4}$$

and $h_{IC} \vDash_{F_{\Lambda'}} \circledast\text{inv}(h \uplus h_{IC} \uplus h_{rC}, I)$.

Because (B.1) and $\tau_A = \epsilon$ we know there is some h_{QA}, h_{rA} such that

$$h = h_{QA} \uplus h_{rA} \tag{B.5}$$

and

$$h_{QA} \vDash_{F_{\Lambda'}} Q_1 \tag{B.6}$$

Because (B.2) and (B.6):

$$\text{safe}'(\Lambda', h_{QA}, \tau_B, I, Q_2) \tag{B.7}$$

Let $h_{rB} = h_{rC} \uplus h_{rA}$ and $h_{IB} = h_{IC}$.

We have:
$h_1 = \text{erat}(h \uplus h_{IC} \uplus h_{rC})$ (because (B.4))
$= \text{erat}((h_{QA} \uplus h_{rA}) \uplus h_{IC} \uplus h_{rC})$ (because (B.5))
$= \text{erat}(h_{QA} \uplus h_{IC} \uplus h_{rB})$

Similarly, out of $h_{IC} \vDash_{F_{\Lambda'}} \circledast\text{inv}(h \uplus h_{IC} \uplus h_{rC}, I)$ we can conclude $h_{IC} \vDash_{F_{\Lambda'}} \circledast\text{inv}(h_{QA} \uplus h_{IC} \uplus h_{rB}, I)$.

Combine this with (B.7) to obtain:
$\exists h'_B, h'_{IB}.$
$h_2 = \text{erat}(h'_B \uplus h'_{IB} \uplus h_{rB})$
$\wedge\ h'_{IB} \vDash_{F_{\Lambda'}} \circledast\text{inv}(h'_B \uplus h'_{IB} \uplus h_{rB}, I)$
$\wedge\ \text{safe}(\Lambda' -_I 1, h_{QA}, \tau_B, I, Q_2)$
$\wedge\ I = \text{inat} \Rightarrow \text{inv}(h_{QA} \uplus h_{IC} \uplus h_{rB}, \text{outat}) = \text{inv}(h'_B \uplus h'_{IB} \uplus h_{rB}, \text{outat})$

This conjunction is what we wanted to prove.

- $\tau_A = \epsilon \wedge \tau_B = \epsilon$. Therefore, $\tau_C = \epsilon$.

Because (B.1), $\tau_A = \epsilon$, and $\Lambda' > (0, 0)$ there exists some h_{QA}, h_{rA} such that

$$h = h_{QA} \uplus h_{rA} \tag{B.8}$$

and

$$h_{QA} \vDash_{F_{\Lambda'}} Q_1 \tag{B.9}$$

Because (B.2) and (B.9): $\text{safe}'(\Lambda', h_{QA}, \tau_B, I, Q_2)$.

Combine this with $\tau_B = \epsilon$ and the definition of safe′ to obtain that there is some h_{QB}, h_{rB} such that

$$h_{QA} = h_{QB} \uplus h_{rB} \tag{B.10}$$

and $h_{QB} \vDash_{F_{\Lambda'}} Q_2$ (we wanted to prove the latter).

Let $h_{rC} = h_{rA} \uplus h_{rB}$. Then:
$h = h_{QA} \uplus h_{rA}$ (given in (B.9))
$= h_{QB} \uplus h_{rB} \uplus h_{rA}$ (because of (B.10))
$= h_{QB} \uplus h_{rC}$ which we wanted to prove.

□

Lemma 38: Safe leak

$\forall \Lambda, h, \tau, I, Q, R.\ \text{safe}(\Lambda, h, \tau, I, Q * R) \Rightarrow \text{safe}(\Lambda, h, \tau, I, Q).$

Proof. We perform well-founded induction on Λ. Choose arbitrary $\Lambda' \leq \Lambda$. We have to prove $\text{safe}'(\Lambda', h, \tau, I, Q)$. Assume $\Lambda' > (0, 0)$ (the case $\Lambda' = (0, 0)$ is trivial). We perform case analysis on τ.

- $\tau = (h_1, h_2) \cdot \tau'$ for some h_1, h_2, τ'.

  Choose arbitrary h_r, h_I such that
  $h_1 = \text{erat}(h \uplus h_I \uplus h_r) \wedge h_I \vDash_{F_{\Lambda'}} \circledast\text{inv}(h \uplus h_I \uplus h_r, I)$.

  Because $\text{safe}'(\Lambda', h, \tau, I, Q * R)$: $\exists h', h'_I.\ h_2 = \text{erat}(h' \uplus h'_I \uplus h_r)$
  $\wedge\ h'_I \vDash_{F_{\Lambda'}} \circledast\text{inv}(h' \uplus h'_I \uplus h_r, I) \wedge \text{safe}(\Lambda' -_I 1, h', \tau', I, Q * R)$
  $\wedge\ I = \text{inat} \Rightarrow \text{inv}(h \uplus h_I \uplus h_R, \text{outat}) = \text{inv}(h' \uplus h'_I \uplus h_R, \text{outat})$.

  Apply the induction hypothesis to obtain $\text{safe}(\Lambda' -_I 1, h', \tau', I, Q)$.

- $\tau = \epsilon$.

  Because $\text{safe}'(\Lambda', h, \tau, I, Q * R)$: $\exists h_{r1}, h'_1.\ h = h_{r1} \uplus h'_1 \wedge h'_1 \vDash_{F_{\Lambda'}} Q * R$.

  Because $h'_1 \vDash_{F_{\Lambda'}} Q * R$, we know
  $\exists h_Q, h_R. h_Q \vDash_{F_{\Lambda'}} Q \wedge h_R \vDash_{F_{\Lambda'}} R \wedge h'_1 = h_Q \uplus h_R$.

  Let $h_{r2} = h_{r1} \uplus h_R$. Let $h'_2 = h_Q$.

  Then: $h = h_{r2} \uplus h'_2$ (which we want to prove) because
  $h = h_{r1} \uplus h'_1 = h_{r1} \uplus h_R \uplus h_Q = h_{r2} \uplus h'_2$.

- $\tau = (h_1, \textbf{inf}) \cdot \tau'$ for some τ': trivial.

- $\tau = (h, \bot) \cdot \tau'$ for some h, τ'.

  This case does not occur because then $\neg\text{safe}'(\Lambda', h, \tau, I, Q * R)$.

$\square$

Lemma 39: Safe frame

$\forall \Lambda, h, Q, R, I, \tau, h_R.$
$\text{safe}(\Lambda, h, \tau, I, Q) \wedge h_R \vDash_{F_\Lambda} R \Rightarrow \text{safe}(\Lambda, h \uplus h_R, \tau, I, Q * R)$

Proof. Choose arbitrary Λ, I. We perform well-founded induction on Λ. Choose arbitrary $\Lambda' \leq \Lambda$. We have to prove $\text{safe}'(\Lambda', h \uplus h_R, \tau, I, Q * R)$. The case $\Lambda' = (0, 0)$ is trivial. We perform case analysis on τ.

- $\tau = (h_1, h_2) \cdot \tau'$ for some h_1, h_2, τ'.

  Choose arbitrary h_r, h_I such that $h_1 = \text{erat}((h \uplus h_R) \uplus h_I \uplus h_r)$ and $h_I \vDash_{F_{\Lambda'}} \circledast\text{inv}((h \uplus h_R) \uplus h_I \uplus h_r, I)$.

  Because $\text{safe}'(\Lambda', h, \tau, I, Q)$ there is some h', h'_I such that $h_2 = \text{erat}(h' \uplus h'_I \uplus (h_R \uplus h_r))$ and $h_I \vDash_{F_{\Lambda'}} \circledast\text{inv}(h' \uplus h'_I \uplus (h_R \uplus h_r), I)$ and
  $I = \text{inat} \Rightarrow \text{inv}(h \uplus h_I \uplus (h_r \uplus h_R), \text{outat}) = \text{inv}(h' \uplus h'_I \uplus (h_R \uplus h_r), \text{outat})$ and $\text{safe}(\Lambda' -_I 1, h', \tau', I, Q)$.

  Therefore, $h_2 = \text{erat}((h' \uplus h_R) \uplus h'_I \uplus h_r)$ and
  $h_I \vDash_{F_{\Lambda'}} \circledast\text{inv}((h' \uplus h_R) \uplus h'_I \uplus h_r, I)$ and
  $I = \text{inat} \Rightarrow \text{inv}((h \uplus h_R) \uplus h_I \uplus h_r, \text{outat}) = \text{inv}((h' \uplus h_R) \uplus h'_I \uplus h_r, \text{outat})$
  (which we all three want to prove).

  Because $h_R \vDash_{F_\Lambda} R$ and $\Lambda' \leq \Lambda$, we know $h_R \vDash_{F_{\Lambda'}} R$

  We can now apply the induction hypothesis to obtain
  $\text{safe}(\Lambda' -_I 1, h' \uplus h_R, \tau', I, Q * R)$.

- $\tau = \epsilon$.

  Because of $\text{safe}'(\Lambda', h, \tau, I, Q)$: $\exists h_r, h_Q.\ h = h_r \uplus h_Q \wedge h_Q \vDash_{F_{\Lambda'}} Q$.
  Therefore, $h \uplus h_R = h_r \uplus (h_Q \uplus h_R) \wedge h_Q \uplus h_R \vDash_{F_{\Lambda'}} Q * R$.

$\square$

Lemma 40: Safe conseq

$\forall \Lambda, h, \tau, I, Q, P.$
$(P \rightarrow Q) \Rightarrow$
$\text{safe}(\Lambda, h, \tau, I, P) \Rightarrow$
$\text{safe}(\Lambda, h, \tau, I, Q).$

Proof. We perform well-founded induction on Λ. Choose arbitrary $\Lambda' \leq \Lambda$. We have to prove $\text{safe}'(\Lambda', h, \tau, I, Q)$. The case $\Lambda' = (0, 0)$ is trivial. We perform case analysis on τ.

- $\tau = (h_1, h_2) \cdot \tau'$ for some h_1, h_2, τ'.

  Choose arbitrary h_r, h_I such that $h_1 = \text{erat}(h \uplus h_I \uplus h_r)$ and $h_I \vDash_{F_{\Lambda'}} \circledast \text{inv}(h \uplus h_I \uplus h_r, I)$.

  Because $\text{safe}'(\Lambda', h, \tau, I, P)$: $\exists h', h'_I.\ h_2 = \text{erat}(h' \uplus h'_I \uplus h_r)$ and $h_I \vDash_{F_{\Lambda'}} \circledast \text{inv}(h' \uplus h'_I \uplus h_r, I)$ and $I = \text{inat} \Rightarrow \text{inv}(h \uplus h_I \uplus h_r, \text{outat}) = \text{inv}(h' \uplus h'_I \uplus h_r, \text{outat})$ (which we all three wanted to prove) and $\text{safe}(\Lambda' -_I 1, h, \tau', I, P)$.

  Because of the induction hypothesis: $\text{safe}(\Lambda' -_I 1, h, \tau', I, Q)$.

- $\tau = \epsilon$.

  Because $\text{safe}'(\Lambda', h, \tau, I, P)$: $\exists h', h_r.\ h = h' \uplus h_r \wedge h' \vDash_{F_{\Lambda'}} P$.

  Because $P \rightarrow Q$: $h' \vDash_{F_{\Lambda'}} Q$.

□

Lemma 41: Safe outcome

$\forall \Lambda, h_1, \tau, n, h_2, \Lambda, Q, h_r.$
$h_1, \tau \rightarrow (n, h_2)$
$\wedge\ \text{safe}(\Lambda +_{\text{inat}} n +_{\text{inat}} 1, h, \tau, \text{inat}, Q)$
$\wedge\ h_1 = \text{erat}(h \uplus h_r)$
$\Rightarrow$
$\exists h', h_{rr}.\ h_2 = \text{erat}(h' \uplus h_{rr} \uplus h_r)$
$\wedge\ h' \vDash_{F_\Lambda} Q$
$\wedge\ \text{inv}(h \uplus h_r, \text{outat}) = \text{inv}(h' \uplus h_{rr} \uplus h_r, \text{outat}).$

Proof. We perform induction on n.

- $n = 0$.

  Because $h_1, \tau \rightarrow (n, h_2)$ we know that $\tau = \epsilon$ and $h_1 = h_2$.

  Because safe$(\Lambda +_{\text{inat}} n +_{\text{inat}} 1, h, \tau, \text{inat}, Q)$ there is some h', h_{rr} such that $h = h' \uplus h_{rr}$ and $h' \vDash_{F_\Lambda} Q$. We wanted to prove the latter.

  We know $h_1 = \text{erat}(h \uplus h_r)$. Replace h to obtain
  $h_1 = h_2 = \text{erat}(h' \uplus h_{rr} \uplus h_r)$ (which we wanted to prove).

  Use $h = h' \uplus h_{rr}$ to obtain $\text{inv}(h \uplus h_r, \text{outat}) = \text{inv}(h' \uplus h_{rr} \uplus h_r, \text{outat})$.

- $n > 0$. Case analysis on τ.

  – $\tau = (h_1, h_2') \cdot \tau'$ for some h_2', τ'.
    Because safe$(\Lambda +_{\text{inat}} n +_{\text{inat}} 1, h, \tau, \text{inat}, Q)$ and $h_1 = \text{erat}(h \uplus h_r)$:
    there is some h_A', h_{IA}' such that
    $h_2' = \text{erat}(h_A' \uplus h_{IA}' \uplus h_r)$
    and $h_{IA}' \vDash_{\Lambda +_{\text{inat}} n} \text{inv}(h_A' \uplus h_{IA}' \uplus h_r, \text{inat})$
    and safe$(\Lambda +_{\text{inat}} n, h_A', \tau', \text{inat}, Q)$
    and $\text{inv}(h \uplus h_r, \text{outat}) = \text{inv}(h_A' \uplus h_r, \text{outat})$.
    Note that therefore $h_{IA}' = \emptyset$.
    Because $h_1, \tau \rightarrow (n, h_2)$ and $\tau = (h_1, h_2') \cdot \tau'$ we know
    $h_2', \tau' \rightarrow (n-1, h_2)$
    We now have all the requirements to apply the induction hypothesis.

  – $\tau = \epsilon$. This case cannot happen because then $h_1, \epsilon \rightarrow (n, h_2)$ with $n > 0$ (which is impossible).

  – $\tau = (h_1, \bot)$. Similarly, this case cannot happen.

  – $\tau = (h_1, \textbf{inf})$. Similarly, this case cannot happen.

□

Lemma 42: Unsafe inat outcome

$\forall h_1, \tau, n, h, Q, h_r.\ h_1, \tau \rightarrow (n, \bot) \wedge h_1 = h \uplus h_r \Rightarrow$
$\neg\text{safe}((0, n+1), h, \tau, \text{inat}, Q)$.

Proof. We perform induction on n.

- $n = 0$. In that case $\tau = (h_1, \bot) \cdot \tau'$ for some τ'. What we want to prove immediately follows from the definition of safe.

- $n > 0$.

  Proof by contradiction: assume $\mathrm{safe}((0, n+1), h, \tau, \mathrm{inat}, Q)$.

  We perform case analysis on τ.

  - $\tau = \epsilon$. This case is not possible because $h_1, \tau \to (n, \bot)$.
  - $\tau = (h_1, h_2) \cdot \tau'$ for some h_2, τ'.

    Because $\mathrm{safe}((0, n+1), h, \tau, \mathrm{inat}, Q)$ there is some h' such that $h_2 = h' \uplus h_r$ and $\mathrm{safe}(n, h', \tau', \mathrm{inat}, Q)$.

    Because $h_1, \tau \to (n, \bot)$ and $\tau = (h_1, h_2) \cdot \tau'$ we know $h_2, \tau' \to (n-1, \bot)$.

    Now apply the induction hypothesis to obtain $\neg\mathrm{safe}((0, n), h', \tau', \mathrm{inat}, Q)$.

    We have obtained a contradiction.
  - $\tau = (h_1', \bot) \cdot \tau'$ for some h_1', τ'.

    This case cannot happen because $n > 0$ and $h_1, \tau \to (n, \bot)$.
  - $\tau = (h_1', \mathrm{inf}) \cdot \tau'$ for some h_1', τ'.

    This case cannot happen because $h_1, \tau \to (n, \bot)$.

□

Lemma 43: Hoare triple validity

$$\forall P, c, Q, I.\ \vdash \{P\}\, c\, \{Q\}_I \Rightarrow\ \models \{P\}\, c\, \{Q\}_I.$$

Proof. We perform induction on $\vdash \{P\}\, c\, \{Q\}_I$. For each case, choose arbitrary Λ (we want to prove $\models^\Lambda \{P\}\, c\, \{Q\}_I$).

- Par proof rule. $P = P_1 * P_2$, $c = c_1 \,||\, c_2$, and $Q = Q_1 * Q_2$ for some $P_1, P_2, c_1, c_2, Q_1, Q_2$.

  Choose arbitrary h, τ, v such that $h \vDash_{F_\Lambda} P_1 * P_2$ and $c \Downarrow \tau, v$.

  Because of the Par rule of the step semantics, $\tau \in \tau_1 \,||\, \tau_2$, $c_1 \Downarrow \tau_1, \_$ and $c_2 \Downarrow \tau_2, \_$ for some τ_1, τ_2.

  We want to prove $\mathrm{safe}(\Lambda, h, \tau, I, Q)$.

  Because $h \vDash_{F_\Lambda} P$, there is some h_1, h_2 such that $h_1 \vDash_{F_\Lambda} P_1$ and $h_2 \vDash_{F_\Lambda} P_2$.

Because of the induction hypothesis, we know $\models^{\Lambda} \{P_1\}\, c_1\, \{Q_1\}$, therefore safe$(\Lambda, h_1, \tau_1, I, Q_1)$.

Similarly, safe$(\Lambda, h_2, \tau_2, I, Q_2)$.

We can now apply Lemma 1 on page 149 to obtain safe(Λ, h, τ, I, Q).

- LetR proof rule. $c = \textbf{let } x := c_1 \textbf{ in } c_2$ for some x, c_1, c_2.

  Because of the induction hypothesis:

$$\models \big\{P\big\}\, c_1\, \big\{Q'\big\}_I \tag{B.11}$$

  for some Q', and

$$\forall v.\ \models \big\{Q'[v/\mathsf{res}]\big\}\, c_2[v/x]\, \big\{Q\big\}_I \tag{B.12}$$

  We have to prove $\models^{\Lambda} \big\{P\big\}\, c\, \big\{Q\big\}_I$.

  Choose arbitrary h such that

$$h \vDash_{F_\Lambda} P \tag{B.13}$$

  Choose arbitrary τ, v such that $c \Downarrow \tau, v$.

  We have to prove safe$(\Lambda, h, \tau, I, Q[v/\mathsf{res}])$.

  Because of the step rules: $\tau = \tau_1; \tau_2$ and

$$c_1 \Downarrow \tau_1, v_1 \tag{B.14}$$

$$c_2[v_1/x] \Downarrow \tau_2, v \tag{B.15}$$

  for some τ_1, τ_2, v_1.

  Because (B.11), (B.13), (B.14) we can conclude safe$(\Lambda, h, \tau_1, I, Q1[v_1/\mathsf{res}])$.

  Because (B.12) and (B.15) we know $\forall h_Q.\ h_Q \vDash_{F_\Lambda} Q'[v_1/\mathsf{res}] \Rightarrow$ safe$(\Lambda, h_Q, \tau_2, I, Q[v/\mathsf{res}])$.

  We can now apply Lemma 2 on page 149.

- LetG proof rule. Analogous to LetR proof rule.

- Then proof rule. Follows from induction hypothesis.

- Else proof rule. Follows from induction hypothesis.

- Frame proof rule. $P = P' * R$, $Q = Q' * R$ for some P', Q', R.

  Choose arbitrary h such that $h \vDash_{F_\Lambda} P' * R$. Therefore, there is some h_P, h_R such that $h_P \vDash_{F_\Lambda} P'$ and $h_R \vDash_{F_\Lambda} R'$

  Choose arbitrary τ, v such that $c \Downarrow \tau, v$.

  Because of the induction hypothesis: $\text{safe}(\Lambda, h_P, \tau, I, Q[v/\mathsf{res}])$.

  Apply Lemma 4 on page 150 to obtain $\text{safe}(\Lambda, h, \tau, I, (Q' * R)[v/\mathsf{res}])$.

- Mut proof rule. $P = v_1 \mapsto \_$, $c = [v_1] := v_2$, $Q = v_1 \mapsto v_2$ for some v_1, v_2.

  Choose arbitrary h such that $h \vDash_{F_\Lambda} P$.

  Therefore, $h = \{\!\{v_1 \mapsto v_0\}\!\}$ for some v_0.

  Choose arbitrary τ, v such that $c \Downarrow \tau, v$.

  We want to prove $\text{safe}(\Lambda, h, \tau, I, Q)$.

  Choose arbitrary $\Lambda' \leq \Lambda$. Assume $\Lambda' > (0,0)$ (the case $\Lambda' = (0,0)$ is trivial).

  Case analysis on $c \Downarrow \tau, v$.

  - Mut rule of the step semantics.
    $\tau = (h_R \uplus \{\!\{v_1 \mapsto v_0'\}\!\}, h_R \uplus \{\!\{v_1 \mapsto v_2\}\!\}) \cdot \epsilon$ for some v_0, h_R.
    Choose arbitrary h_I, h_r such that
    $h_R \uplus \{\!\{v_1 \mapsto v_0'\}\!\} = \text{erat}(h \uplus h_I \uplus h_r)$ and $h_I \vDash_{F_{\Lambda'}} \circledast\text{inv}(h \uplus h_I \uplus h_r)$
    It follows that $\{\!\{v_1 \mapsto v_0'\}\!\} = h$ (because of the definition of $\uplus$ for heaps).
    Let $h' = \{\!\{v_1 \mapsto v_2\}\!\}$ and $h_I' = h_I$.
    It follows from rewriting equalities that
    $h_R \uplus \{\!\{v_1 \mapsto v_2\}\!\} = \text{erat}(h' \uplus h_I' \uplus h_r)$ and $h_I' \vDash_{F'_\Lambda} \circledast\text{inv}(h' \uplus h_I' \uplus h_r)$ and $I = \text{inat} \Rightarrow \text{inv}(h \uplus h_I \uplus h_r, \text{outat}) = \text{inv}(h' \uplus h_I' \uplus h_r, \text{outat})$ (which we all three wanted to prove).
    $\text{safe}(\Lambda' -_I 1, h', \epsilon, I, Q)$ follows immediately because of the definition of safe$'$.

  - MutErr step rule. $\tau = (h_1, \bot) \cdot \epsilon$ for some h_1.
    It suffices to prove that there is no h_I, h_r such that
    $h_1 = \text{erat}(h \uplus h_I \uplus h_r)$.
    Prove by contradiction: assume such h_I, h_r exist.
    Since $h = \{\!\{v_1 \mapsto v_0\}\!\}$: $h_1 = \{\!\{v_1 \mapsto v_0\}\!\} \uplus \text{erat}(h_I \uplus h_r)$.
    This contradicts with the premise of the MutErr step rule, which states that h_1 does not have a heap cell with address v_1.

- Pa proof rule. $P = \mathbf{pr}(v_{id}, v_1)$, $c = \mathbf{pa}(v_{id}, v_2)$, $Q = (v_1 = v_2)$, for some v_{id}, v_1, v_2.

  Choose arbitrary h such that $h \vDash_{F_\Lambda} P$.

  Therefore, $h = \{\!\!\{\mathbf{pr}(v_{id}, v_1)\}\!\!\}$.

  Choose arbitrary τ, v such that $c \Downarrow \tau, v$.

  We want to prove safe(Λ, h, τ, I, Q).

  Choose arbitrary $\Lambda' \leq \Lambda$. Assume $\Lambda' > (0,0)$ (the case $\Lambda' = (0,0)$ is trivial).

  Case analysis on $c \Downarrow \tau, v$.

  - Pa rule of the step semantics.
    $\tau = (h_R \uplus \{\!\!\{\mathbf{pr}(v_{id}, v_2)\}\!\!\}, h_R \uplus \{\!\!\{\mathbf{pr}(v_{id}, v_2, \text{assig})\}\!\!\}) \cdot \epsilon$ for some h_R.
    Choose arbitrary h_I, h_r such that
    $h_R \uplus \{\!\!\{\mathbf{pr}(v_{id}, v_2)\}\!\!\} = \text{erat}(h \uplus h_I \uplus h_r)$ and $h_I \vDash_{F_{\Lambda'}} \circledast\text{inv}(h \uplus h_I \uplus h_r)$
    It follows that $\{\!\!\{\mathbf{pr}(v_{id}, v_2)\}\!\!\} = h$ (because of the definition of $\uplus$ for heaps).
    Therefore, $v_1 = v_2$.
    Let $h' = \{\!\!\{\mathbf{pr}(v_{id}, v_2, \text{assig})\}\!\!\}$ and $h'_I = h_I$.
    It follows from rewriting equalities that
    $h_R \uplus \{\!\!\{\mathbf{pr}(v_{id}, v_2, \text{assig})\}\!\!\} = \text{erat}(h' \uplus h'_I \uplus h_r)$ and
    $h'_I \vDash_{F'_\Lambda} \circledast\text{inv}(h' \uplus h'_I \uplus h_r)$ and
    $I = \text{inat} \Rightarrow \text{inv}(h \uplus h_I \uplus h_r, \text{outat}) = \text{inv}(h' \uplus h'_I \uplus h_r, \text{outat})$ (which we all three wanted to prove).
    safe$(\Lambda' -_I 1, h', \epsilon, I, Q)$ follows immediately because of the definition of safe$'$. Note that we use here that safe$'$ supports leaking.
  - PaErr step rule. $\tau = (h_1, \bot) \cdot \epsilon$ for some h_1.
    It suffices to prove that there is no h_I, h_r such that
    $h_1 = \text{erat}(h \uplus h_I \uplus h_r)$.
    Prove by contradiction: assume such h_I, h_r exist.
    Since $h = \{\!\!\{\mathbf{pr}(v_{id}, v_1)\}\!\!\}$: $h_1 = \{\!\!\{\mathbf{pr}(v_{id}, v_1)\}\!\!\} \uplus \text{erat}(h_I \uplus h_r)$.
    This contradicts with the premise of the PaErr step rule, which states that h_1 does not have an unassigned prophecy chunk cell with ID v_{id}.

- Leak proof rule.

  Follows from Lemma 3 on page 150.

- Exists proof rule. $P = \exists x.P'$ for some P'.

  Because of the induction hypothesis, $\forall v.\ \models^{\Lambda} \{P'[v/x]\}\, c\, \{Q\}_I$.

  Therefore,

  $$\forall v.\ \forall h.\ h \vDash_{F_\Lambda} P'[v/x] \Rightarrow \forall \tau, v.c \Downarrow \tau, v \Rightarrow \text{safe}(\Lambda, h, \tau, I, Q) \qquad \text{(B.16)}$$

  Choose arbitrary h such that $h \vDash_{F_\Lambda} \exists x.P'$. Because $h \vDash_{F_\Lambda} \exists x.P'$ we know there is some v such that $h \vDash_{F_\Lambda} P'[v/x]$. Now we can apply (B.16).

- Conseq proof rule.

  Given $\models^{\Lambda} \{P'\}\, c\, \{Q'\}_I$, $P \to P'$, $Q' \to Q$. We want to prove $\models^{\Lambda} \{P\}\, c\, \{Q\}_I$.

  Choose arbitrary h, v, τ such that $h \vDash_{F_\Lambda} P$ and $c \Downarrow \tau, v$.

  Because $P \to P'$ we know $h \vDash_{F_\Lambda} P'$.

  Because $\models^{\Lambda} \{P'\}\, c\, \{Q'\}_I$ we know $\text{safe}(\Lambda, h, \tau, I, Q')$.

  Because of Lemma 5 on page 150, we know $\text{safe}(\Lambda, h, \tau, I, Q)$ (which we wanted to prove).

- Exp proof rule.

  $P = \mathbf{emp}$, $c = e$, $Q = (\mathsf{res} = v)$, $v = [\![e]\!]$ for some e, v.

  Choose arbitrary h such that $h \vDash_{F_\Lambda} P$. Therefore, $h = \{\!\}\!$.

  Choose arbitrary τ, v' such that $c \Downarrow \tau, v'$. Because of the step rules: $v' = v$ and $\tau = \epsilon$.

  Choose arbitrary $m' \leq m$. We have to prove $\text{safe}'(m', \{\!\}\!, \epsilon, I, (res = v)[v/\mathsf{res}])$. In case $m' > 0$, we have to identify an h_r, h_Q such that $\{\!\}\! = h_r \uplus h_Q$ and $h_Q \vDash_{F_{\Lambda'}} v = v$. This holds for $h_r = h_Q = \{\!\}\!$.

- AppSimple proof rule. $c = (\lambda \overline{x}; \overline{y}.\, c')\ (\overline{v_1}; \overline{v_2})$ for some $\overline{x}, \overline{y}, c', \overline{v_1}, \overline{v_2}$.

  Choose arbitrary h, τ, v such that $h \vDash_{F_\Lambda} P$ and $(\lambda \overline{x}; \overline{y}.\, c')(\overline{v_1}; \overline{v_2}) \Downarrow \tau, v$.

  Because of the App step rule: $c'[\overline{v_1}/\overline{x}][\overline{v_2}/\overline{y}] \Downarrow \tau', v$ where $\tau = (h_a, h_a) \cdot \tau'$ for some h_a, τ'.

  Because of $\models^{\Lambda} \{P\}\, c'[\overline{v_1}/\overline{x}][\overline{v_2}/\overline{y}]\, \{Q\}_I$ we know $\text{safe}(\Lambda - 1, h, \tau', I, Q[v/\mathsf{res}])$.

  Choose arbitrary $\Lambda' \leq \Lambda$. We have to prove $\text{safe}'(\Lambda', h, (h_a, h_a) \cdot \tau', I, Q[v/\mathsf{res}])$.

  Choose arbitrary h_r, h_I such that $h_a = \text{erat}(h \uplus h_I \uplus h_r) \wedge h_I \vDash_{F_{\Lambda'}} \circledast\text{inv}(h \uplus h_I \uplus h_r, I)$.

We have to come up with a h', h'_I such that $h_a = \mathrm{erat}(h' \uplus h'_I \uplus h_r)$ (and some other conditions which we will prove next). Let $h' = h$ and $h'_I = h_I$.

We have to prove that
$I = \mathrm{inat} \Rightarrow \mathrm{inv}(h \uplus h_I \uplus h_r, \mathrm{outat}) = \mathrm{inv}(h' \uplus h'_I \uplus h_r, \mathrm{outat})$ which holds trivially.

Next we need to prove $h'_I \vDash_{F_\Lambda} \circledast\mathrm{inv}(h' \uplus h_r, I)$ and
$\mathrm{safe}(\Lambda - 1, h', \tau', I, Q[v/\mathsf{res}])$, which we both obtained already above.

- Lookup proof rule. $P = [\pi]v \mapsto v'$, $c = [v]$, and
$Q = (res = v' * [\pi]v \mapsto v')$ for some v, v'.

  Choose arbitrary h such that $h \vDash_{F_\Lambda} P$. Therefore, $h = \{\![\pi]v \mapsto v'\}\!\}$.

  Choose arbitrary τ, v_{res} such that $c \Downarrow \tau, v_{res}$.

  We perform case analysis on the step rule of $c \Downarrow \tau, v_{res}$.

  - LookupErr step rule. $\tau = (h_l, \bot) \cdot \epsilon$ and $(v \mapsto \_) \notin h_l$ for some h_l.

    We have to prove $\mathrm{safe}(\Lambda, h, (h_l, \bot) \cdot \epsilon, I, Q[v_{\mathsf{res}}/\mathsf{res}])$.

    Looking at the definition of safe$'$, we want to prove something for arbitrary h_r, h_I where $h_l = \mathrm{erat}(h \uplus h_I \uplus h_r)$.

    Such an h_r and h_I do not exist since $(v \mapsto \_) \in h$ while
$(v \mapsto \_) \notin h_l$.

  - Lookup step rule. $\tau = (h_l, h_l) \cdot \epsilon$, $h_l(v \mapsto v'') > 0$ and $v_{res} = v''$ for some v''.

    Choose arbitrary $\Lambda' \leq \Lambda$. We want to prove
$\mathrm{safe}'(\Lambda', h, (h_l, h_l) \cdot \epsilon, I, Q[v_{\mathsf{res}}/\mathsf{res}])$. Assume $\Lambda' > (0, 0)$ (the case $\Lambda' = (0, 0)$ is trivial).

    Choose arbitrary h_r, h_I such that $h_l = \mathrm{erat}(h \uplus h_I \uplus h_r)$ and
$h_I \vDash_{F_{\Lambda'}} \circledast\mathrm{inv}(h \uplus h_I \uplus h_r, I)$.

    We have to identify a h', h'_I such that $h_l = \mathrm{erat}(h' \uplus h'_I \uplus h_r)$ and
$h'_I \vDash_{F_{\Lambda'}} \circledast\mathrm{inv}(h\prime \uplus h'_I \uplus h_r, I)$ and
$I = \mathrm{inat} \Rightarrow \mathrm{inv}(h \uplus h_I \uplus h_r, \mathrm{outat}) = \mathrm{inv}(h' \uplus h'_I \uplus h_r, \mathrm{outat})$ (and another property that we will prove next). This conjunction is satisfied by choosing $h' = h, h'_I = h_I$.

    Next we need to prove $\mathrm{safe}(\Lambda' -_I 1, h, \epsilon, I, Q[v_{\mathsf{res}}/\mathsf{res}])$.

    In case $\Lambda' -_I 1 > 0$, we have to identify a h'_r, h_Q such that
$h = h'_r \uplus h_Q \wedge h_Q \vDash_{F_{\Lambda' -_I 1}} Q[v_{\mathsf{res}}/\mathsf{res}]$. This is satisfied by choosing $h'_r = \{\!\}\!\}, h_Q = h$.

- Cons proof rule. $c = \mathbf{cons}(\overline{v})$,
$Q = \mathsf{res} \mapsto v_0 * \mathsf{res} + 1 \mapsto v_1 * \ldots * \mathsf{res} + k \mapsto v_k$, $\overline{v} = v_0, v_1, \ldots, v_k$, and
$P = \mathbf{emp}$ for some $\overline{v}, k, v_0, v_1, \ldots, v_k$.

Choose arbitrary τ, v such that $\mathbf{cons}(\overline{v}) \Downarrow \tau, v$.

Because of the step rule:
$\tau = (h_s, h_s \uplus h'_s) \cdot \epsilon \wedge h'_s = \{\!\!\{ n \mapsto v_0, n+1 \mapsto v_1, \ldots, n+k \mapsto v_k \}\!\!\}$ for some n, h_s, h'_s.

Choose arbitrary $\Lambda' \leq \Lambda$.

We want to prove safe'$(\Lambda', \{\!\!\{\}\!\!\}, \tau, I, Q[v/\mathsf{res}])$. Assume $\Lambda' > (0,0)$ (the case $\Lambda' = (0,0)$ is trivial).

Choose arbitrary h_r, h_I such that
$h_s = \mathrm{erat}(\{\!\!\{\}\!\!\} \uplus h_I \uplus h_r) \wedge h_I \vDash_{F_{\Lambda'}} \circledast\mathrm{inv}(\{\!\!\{\}\!\!\} \uplus h_I \uplus h_r, I)$.

We need to identify a h', h'_I such that
$h_s \uplus h'_s = \mathrm{erat}(h' \uplus h'_I \uplus h_r) \wedge h_I \vDash_{F_{\Lambda'}} \circledast\mathrm{inv}(h' \uplus h'_I \uplus h_r, I) \wedge I = \mathrm{inat} \Rightarrow \mathrm{inv}(h' \uplus h'_I \uplus h_r, \mathrm{outat}) = \mathrm{inv}(h' \uplus h'_I \uplus h_r, \mathrm{outat})$ (and another property that we will prove next). This holds by choosing $h' = h'_s$ and $h'_I = h_I$.

Next we have to prove safe$(\Lambda' -_I 1, h', \epsilon, I, Q[v/\mathsf{res}])$. We need to identify some h'_r, h'' such that $h' = h'_r \uplus h'' \wedge h'' \vDash_{F_\Lambda} Q[v/\mathsf{res}]$.

Let $h'_r = \emptyset$ and $h'' = h'_s$.

$h'' \vDash_{F_\Lambda} Q[v/\mathsf{res}]$ holds because of the equalities of h'_s and Q.

$h' = h'_r \uplus h''$ holds because:

$$\begin{aligned} h' &= h'_s \text{ (definition of } h'\text{)} \\ &= h'_r \uplus h'_s \text{ (because } h'_r = \emptyset\text{)} \\ &= h'_r \uplus h'' \text{ (definition of } h''\text{)} \end{aligned}$$

- Pc proof rule.

  $c = \mathbf{pc}()$, $Q = \exists \mathsf{x}.\, \mathbf{pr}(\mathsf{res}, \mathsf{x})$, and $P = \mathbf{emp}$.

  Choose arbitrary τ, v_{id} such that $\mathbf{cons}(\overline{v}) \Downarrow \tau, v$.

  Because of the step rule: $\tau = (h_s, h_s \uplus h'_s) \cdot \epsilon \wedge h'_s = \{\!\!\{ \mathbf{pr}(v, v_{pr}) \}\!\!\}$ for some h_s, h'_s, v_{id}.

  Choose arbitrary $\Lambda' \leq \Lambda$.

  We want to prove safe'$(\Lambda', \{\!\!\{\}\!\!\}, \tau, I, Q[v/\mathsf{res}])$. Assume $\Lambda' > (0,0)$ (the case $\Lambda' = (0,0)$ is trivial).

  Choose arbitrary h_r, h_I such that
  $h_s = \mathrm{erat}(\{\!\!\{\}\!\!\} \uplus h_I \uplus h_r) \wedge h_I \vDash_{F_{\Lambda'}} \circledast\mathrm{inv}(\{\!\!\{\}\!\!\} \uplus h_I \uplus h_r, I)$.

  We need to identify a h', h'_I such that
  $h_s \uplus h'_s = \mathrm{erat}(h' \uplus h'_I \uplus h_r) \wedge h_I \vDash_{F_{\Lambda'}} \circledast\mathrm{inv}(h' \uplus h'_I \uplus h_r, I) \wedge I = \mathrm{inat} \Rightarrow \mathrm{inv}(h' \uplus h'_I \uplus h_r, \mathrm{outat}) = \mathrm{inv}(h' \uplus h'_I \uplus h_r, \mathrm{outat})$ (and another property that we will prove next). This holds by choosing $h' = h'_s$ and $h'_I = h_I$.

Next we have to prove safe($\Lambda' -_I 1, h', \epsilon, I, Q[v/\mathsf{res}]$). We need to identify some h'_r, h'' such that $h' = h'_r \uplus h'' \wedge h'' \vDash_{F_\Lambda} Q[v/\mathsf{res}]$.

Let $h'_r = \emptyset$ and $h'' = h'_s$.

$h'' \vDash_{F_\Lambda} Q[v/\mathsf{res}]$ holds: apply the Exists rule of Fig. 4.5 p. 110: now we have to prove $h'' \vDash_{F_\Lambda} \mathbf{pr}(v, v_{pr})$, which holds because
$h'' = h'_s = \{\!\!\{\mathbf{pr}(v, v_{pr})$.

$h' = h'_r \uplus h''$ holds because:

$h' = h'_s$ (definition of h')
$= h'_r \uplus h'_s$ (because $h'_r = \emptyset$)
$= h'_r \uplus h''$ (definition of h'')

- Disj proof rule. $P = P_1 \vee P_2$ for some P_1, P_2.

  Choose arbitrary h such that $h \vDash_{F_m} P$.

  Because $P = P_1 \vee P_2$: $h \vDash_{F_\Lambda} P_1$ or $h \vDash_{F_\Lambda} P_2$. We perform case analysis on this disjunction: assume $h \vDash_{F_\Lambda} P_1$ (the case $h \vDash_{F_\Lambda} P_2$ is analogous).

  Choose arbitrary τ, v such that $c \Downarrow \tau, v$.

  We have to prove safe($\Lambda, h, \tau, I, Q[v/\mathsf{res}]$). This follows from
  $\models^\Lambda \{P_1\}\, c\, \{Q\}_I$ (which we know because of the induction hypothesis).

- False proof rule. We have to prove $\models^\Lambda \{\text{false}\}\, c\, \{Q\}_I$. According to its definition, we have to prove some property for all heaps h where
  $h \vDash_{F_\Lambda}$ false. Such a heap h does not exist.

- Atom proof rule.
  $P = P' * [\pi_1]\boxed{P_1} * \ldots * [\pi_k]\boxed{P_k}$,
  $Q = Q' * [\pi_1]\boxed{P_1} * \ldots * [\pi_k]\boxed{P_k}$,
  $c = \langle c' \rangle$,
  $I = \text{outat}$
  for some $P', P_1, \ldots, P_k, \pi_1, \ldots, \pi_k, Q', Q_1, \ldots, Q_k$ where
  $P_1 \neq P_2 \neq \ldots \neq P_k$.

  Note that since the proof rules have the implicit side condition that the precondition of the Hoare triple of the conclusion has no free variables, we know $\mathsf{res} \notin \text{fv}(P_1 * \ldots * P_k)$.

  Choose arbitrary h such that $h \vDash_{F_\Lambda} P' * [\pi_1]\boxed{P_1} * \ldots * [\pi_2]\boxed{P_k}$.

  Therefore, there is some $h_{P'}$ such that $h_{P'} \vDash_{F_{\Lambda'}} P'$ and
  $h = h_{P'} \uplus \{\!\!\{[\pi_1]\boxed{P_1}, \ldots, [\pi_k]\boxed{P_k}\}\!\!\}$.

  Choose arbitrary τ, v such that $\langle c' \rangle \Downarrow \tau, v$.

Choose arbitrary $\Lambda' \leq \Lambda$. We have to prove
$\text{safe}'(\Lambda', h, \tau, \text{outat}, Q'[v/\mathsf{res}] * [\pi_1]\boxed{P_1} * \ldots * [\pi_k]\boxed{P_k})$.

Assume $\Lambda'.\text{outat} > 0$ (the case $\Lambda'.\text{outat} = 0$ is trivial).

We perform case analysis on $\langle c' \rangle \Downarrow \tau, v$.

– Atom step rule.
  Because of this step rule,
  $\tau = (h_1, h_2) \cdot \epsilon$ for some h_1, h_2
  and $c' \Downarrow \tau', v$ for some τ'
  and $h_1, \tau' \to (n, h_2)$ for some n.
  Choose arbitrary h_I, h_r such that
  $h_1 = \text{erat}(h \uplus h_I \uplus h_r) \land h_I \vDash_{F_{\Lambda'}} \circledast\text{inv}(h \uplus h_I \uplus h_r, \text{outat})$.
  Therefore
  $h_I \vDash_{F_{\Lambda'}} P_1 * \ldots * P_k * \circledast\text{inv}(h \uplus h_I \uplus h_r, \text{outat}) \setminus \{P_1, \ldots, P_k\}$.
  So there is some h_{P1k}, h_{IR} such that $h_{P1k} \vDash_{F_{\Lambda'}} P_1 * \ldots * P_k$ and

$$h_{IR} \vDash_{F_{\Lambda'}} \circledast\text{inv}(h \uplus h_I \uplus h_r, \text{outat}) \setminus \{P_1, \ldots, P_k\} \tag{B.17}$$

  and $h_I = h_{P1k} \uplus h_{IR}$.
  Let $h_B = h_{P'} \uplus h_{P1k}$ and $h_{Br} = \{\!\!\{[\pi_1]\boxed{P_1}, \ldots, [\pi_k]\boxed{P_k}\}\!\!\} \uplus h_{IR} \uplus h_r$.
  Note that $h_1 = \text{erat}(h_B \uplus h_{Br})$.
  Let $\Lambda_C = (\Lambda'.\text{outat} - 1, \Lambda'.\text{inat})$.
  Let $\Lambda_B = \Lambda_C +_{\text{inat}} (n + 1)$.
  Because of the induction hypothesis,
  $\text{safe}(\Lambda_B, h_B, \tau', \text{inat}, Q'[v/\mathsf{res}] * P_1 * \ldots * P_k)$.
  We can now apply Lemma 6 on page 150 to obtain that there is some h'_B, h'_{rr} such that
  $h_2 = \text{erat}(h'_B \uplus h'_{rr} \uplus (\{\!\!\{[\pi_1]\boxed{P_1}, \ldots, [\pi_k]\boxed{P_k}\}\!\!\} \uplus h_{IR} \uplus h_r))$
  $\land\ h'_B \vDash_{F_{\Lambda_C}} Q_1[v/\mathsf{res}] * P_1 * \ldots * P_k$
  and

$$\text{inv}(h_B \uplus h_{Br}, \text{outat}) = \text{inv}(h'_B \uplus h'_{rr} \uplus h_{Br}, \text{outat}) \tag{B.18}$$

  Because $h'_B \vDash_{F_{\Lambda_C}} Q'[v/\mathsf{res}] * P_1 * \ldots * P_k$ there is some h'_{BQ}, h'_{BP1k} such that $h'_B = h_{BQ} \uplus h_{BP1k}$ and $h'_{BQ} \vDash_{F_{\Lambda_C}} Q'[v/\mathsf{res}]$ and

$$h'_{BP1k} \vDash_{F_{\Lambda_C}} P_1 * \ldots * P_k \tag{B.19}$$

  Let $h' = h'_{BQ} \uplus h'_{rr} \uplus \{\!\!\{[\pi_1]\boxed{P_1} * \ldots * [\pi_k]\boxed{P_k}\}\!\!\}$.
  Let $h'_I = h_{IR} \uplus h'_{BP1k}$.

We have:
$h_2 = \mathrm{erat}(h'_B \uplus h_{rr} \uplus \{\![\pi_1]\boxed{P_1} * \ldots * [\pi_k]\boxed{P_k}\}\!\} \uplus h_{IR} \uplus h_r)$
$= \mathrm{erat}((h'_{BQ} \uplus h'_{BP1k}) \uplus h_{rr} \uplus \{\![\pi_1]\boxed{P_1} * \ldots * [\pi_k]\boxed{P_k}\}\!\} \uplus h_{IR} \uplus h_r)$
$= \mathrm{erat}((h'_{BQ} \uplus h_{rr} \uplus \{\![\pi_1]\boxed{P_1} * \ldots * [\pi_k]\boxed{P_k}\}\!\}) \uplus (h'_{BP1k} \uplus h_{IR}) \uplus h_r)$
$= \mathrm{erat}(h' \uplus h'_I \uplus h_r)$ (which we wanted to prove).
We also have

$$\mathrm{inv}((h) \uplus (h_I) \uplus h_r) = \mathrm{inv}(h' \uplus h'_I \uplus h_r) \qquad \text{(B.20)}$$

because
$\mathrm{inv}((h) \uplus (h_I) \uplus h_r)$
$= \mathrm{inv}((h_{P'}) \uplus (h_{P1k} \uplus h_{IR}) \uplus h_r)$
$= \mathrm{inv}((h_{P'} \uplus h_{P1k}) \uplus (h_{IR} \uplus h_r))$
$= \mathrm{inv}(h_B \uplus h_{Br})$
$= \mathrm{inv}(h'_B \uplus h_{Br} \uplus h'_{rr})$ (because of (B.18))
$= \mathrm{inv}((h'_{BQ} \uplus h'_{BP1k}) \uplus (h_{IR} \uplus h_r \uplus \{\![\pi_1]\boxed{P_1} * \ldots * [\pi_k]\boxed{P_k}\}\!\}) \uplus h'_{rr})$
$= \mathrm{inv}((h'_{BQ} \uplus \{\![\pi_1]\boxed{P_1} * \ldots * [\pi_k]\boxed{P_k}\}\!\} \uplus h'_{rr}) \uplus (h'_{BP1k} \uplus h_{IR}) \uplus h_r)$
$= \mathrm{inv}(h' \uplus h'_I \uplus h_r)$.
Because (B.17) and $\Lambda_C < \Lambda'$:
$h_{IR} \vDash_{F_{\Lambda_C}} \circledast\mathrm{inv}(h \uplus h_I \uplus h_r, \mathrm{outat}) \setminus \{P_1, \ldots, P_k\}$.
Rewrite using (B.20):
$h_{IR} \vDash_{F_{\Lambda_C}} \circledast\mathrm{inv}(h' \uplus h'_I \uplus h_r, \mathrm{outat}) \setminus \{P_1, \ldots, P_k\}$.
Combined with (B.19) and $\{P_1, \ldots, P_k\} \subseteq \mathrm{inv}(h' \uplus h'_I \uplus h_r, \mathrm{outat})$ we obtain: $h'_I = h_{IR} \uplus h'_{BP1k} \vDash_{F_{\Lambda_C}} \circledast\mathrm{inv}(h' \uplus h'_I \uplus h_r, \mathrm{outat})$ (which we wanted to prove).

- AtomErr step rule. This case cannot happen. In this case, $h_1, \tau \rightarrow (n, \bot)$ for some n. Apply Lemma 7 on page 151 to obtain $\neg\mathrm{safe}((0, n+1), h, \tau, \mathrm{inat}, Q)$. This contradicts with the induction hypothesis.
- AtomInf step rule. Follows from definition of safe′ (note that because of the step rule we know $\tau = (h_1, \mathbf{inf}) \cdot \epsilon$).

• AtIntro proof rule.

Choose arbitrary h such that $h \vDash_{F_\Lambda} P$.

Choose arbitrary τ, v such that $\mathbf{atintro} \Downarrow \tau, v$.

Therefore, $\tau = (h_1, h_1) \cdot \epsilon$ for some h_1.

Choose arbitrary $\Lambda' \leq \Lambda$. We have to prove $\mathrm{safe}'(\Lambda', h, \tau, \mathrm{outat}, Q)$.

Assume $\Lambda'.\mathrm{outat} > 0$ (the case $\Lambda'.\mathrm{outat} = 0$ is trivial).

Choose arbitrary h_I, h_r such that

$h_1 = \text{erat}(h \uplus h_I \uplus h_r)$ and
$h_I \vDash_{F_{\Lambda'}} \circledast\text{inv}(h \uplus h_I \uplus h_r, \text{outat})$.

Let $h'_I = h_I \uplus h$ and $h' = \{\!\!\{ \boxed{P} \}\!\!\}$.

$$h_1 = \text{erat}(h \uplus h_I \uplus h_r)$$

$$= \text{erat}(h'_I \uplus h_r)$$

$$= \text{erat}(h' \uplus h'_I \uplus h_r)$$

Remember $h_I \vDash_{F_{\Lambda'}} \circledast\text{inv}(h \uplus h_I \uplus h_r, \text{outat})$
Therefore: $h \uplus h_I \vDash_{F_{\Lambda'}} P * \circledast\text{inv}(h \uplus h_I \uplus h_r, \text{outat})$
Therefore: $h'_I \vDash_{F_{\Lambda'}} P * \circledast\text{inv}(h \uplus h_I \uplus h_r, \text{outat})$
Therefore: $h'_I \vDash_{F_{\Lambda'}} \circledast\text{inv}(\{\!\!\{ \boxed{P} \}\!\!\} \uplus h \uplus h_I \uplus h_r, \text{outat})$
Therefore: $h'_I \vDash_{F_{\Lambda'}} \circledast\text{inv}(h' \uplus h'_I \uplus h_r, \text{outat})$

We also have to prove $\text{safe}(\Lambda' -_{\text{outat}} 1, h', \epsilon, \text{outat}, Q)$.

Choose arbitrary $\Lambda'' \leq \Lambda' -_{\text{outat}} 1$.

We have to prove $\text{safe}'(\Lambda'', h', \epsilon, \text{outat}, Q)$. Assume we are not in the trivial case $\Lambda''.\text{outat} = 0$.

Remember $Q = \boxed{P}$. We have to identify a h_Q, h_r such that $h' = h_Q \uplus h_r$ and $h_Q \vDash_{F_{\Lambda''}} Q$. This holds by choosing $h_Q = h$ and $h_r = \emptyset$.

- AtDel proof rule. $P = \boxed{P'}$, $Q = P'$, $I = \text{outat}$, $c = \textbf{atdel}$.

  Note that since the proof rules have the implicit side condition that the precondition of the Hoare triple of the conclusion has no free variables, we know $\mathsf{res} \notin \text{fv}(P')$.

  Choose arbitrary h such that $h \vDash_{F_\Lambda} \boxed{P'}$.

  Choose arbitrary τ, v such that $c \Downarrow \tau, v$.

  We have to prove $\text{safe}(\Lambda, h, \tau, I, Q[v/\text{res}])$.

  Because $c = \textbf{atdel}$: $\tau = (h_1, h_1) \cdot \epsilon$ for some h_1, and $v = \text{unit}$.

  Choose arbitrary $\Lambda' \leq \Lambda$.

  Choose arbitrary h_r, h_I such that $h_1 = \text{erat}(h \uplus h_I \uplus h_r)$ and
  $h_I \vDash_{F_{\Lambda'}} \circledast\text{inv}(h \uplus h_I \uplus h_r, \text{outat})$.

  Because of the latter and because $h = \{\!\!\{ \boxed{P'} \}\!\!\}$:
  $h_I \vDash_{F'_\Lambda} P' * \circledast\text{inv}(h_I \uplus h_r, \text{outat})$.

  Therefore, there is some h', h'_I such that $h_I = h' \uplus h'_I$, $h' \vDash_{F'_\Lambda} P'$, and
  $h'_I \vDash_{F'_\Lambda} \circledast\text{inv}(h_I \uplus h_r, \text{outat})$.

Rewrite h_I in the latter to obtain $h'_I \vDash_{F'_\Lambda} \circledast\mathrm{inv}(h' \uplus h'_I \uplus h_r, \mathrm{outat})$, which we wanted to prove.

Remember $h_1 = \mathrm{erat}(h \uplus h_I \uplus h_r)$. Because $h = \{\!\!\{ \boxed{P'} \}\!\!\}$:
$h_1 = \mathrm{erat}(h_I \uplus h_r)$.

Because $h_I = h' \uplus h'_I$: $h_1 = \mathrm{erat}(h' \uplus h'_I \uplus h_r)$.

We also have to prove that $\mathrm{safe}(\Lambda' -_I 1, h', \epsilon, I, Q)$. Choose arbitrary $\Lambda'' \leq \Lambda' -_I 1$.

Let $h_Q = h'$ and $h_{rQ} = \emptyset$.

We have to prove that $h' = h_Q \uplus h_{rQ}$ and $h_Q \vDash_{F_{\Lambda''}} Q$. The former is trivial. For the latter use that $Q = P'$ and that $h' \vDash_{F_{\Lambda'}} P'$.

- CreateGcf, GcfCons, and GcfMut proof rule. Follows directly from definitions.

- FVI proof rule.
  $P = \mathbf{emp}$,
  $c = \lambda\overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}}. c'$,
  $Q = \left(\mathsf{res} = \lambda\overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}}. c * (\{P'\}\ (\lambda\overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}}. c')(\overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}})\ \{Q'\}^l_I)\right)$
  for some $\overline{x^{\mathrm{r}}}, \overline{y^{\mathrm{g}}}, c', Q', l$.

  Let $\overline{z} = \mathrm{fv}(P' * Q') \setminus (\overline{x} \cup \overline{y})$

  The induction hypothesis states that $\forall \overline{v_1}, \overline{v_2}, \overline{v_3}, \models$
  $\{P'[\overline{v_1}/\overline{x}][\overline{v_2}/\overline{y}][\overline{v_3}/\overline{z}]\}\ c'[\overline{v_1}/\overline{x}][\overline{v_2}/\overline{y}]\ \{Q'[\overline{v_1}/\overline{x}][\overline{v_2}/\overline{y}][\overline{v_3}/\overline{z}]\}^l_I$.
  Choose arbitrary h such that $h \vDash_{F_\Lambda} P$. Choose arbitrary τ, v such that $c \Downarrow \tau, v$. It follows that $h = \{\!\!\}\!\!\}, \tau = \epsilon, v = c$.

  Choose arbitrary $\Lambda' \leq \Lambda$.

  We have to prove that $\mathrm{safe}'(\Lambda', \{\!\!\}\!\!\}, \epsilon, I, Q[c/\mathsf{res}])$.

  In order to do so, we we have to identify an h_Q, h_r such that $\{\!\!\}\!\!\} = h_Q \uplus h_r$ and $h_Q \vDash_{F_{\Lambda'}} Q[c/\mathsf{res}]$.

  Let $h_Q = h_r = \{\!\!\}\!\!\}$. We have to prove $\{\!\!\}\!\!\} \vDash_{F_{\Lambda'}} Q[c/\mathsf{res}]$, i.e.
  $\{\!\!\}\!\!\} \vDash_{F_{\Lambda'}} c = c * (\{P'\}\ c(\overline{x}; \overline{y})\ \{Q'\}^l_I)$.

  It suffices to prove $\{P'\}\ (\lambda\overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}}. c')(\overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}})\ \{Q'\}^l_I \in F_{\Lambda'}$.

  Therefore, it suffices to prove $\forall \overline{v_1}, \overline{v_2}, \overline{v_3}, \Lambda''.\ \Lambda'' < \Lambda' \Rightarrow \models^{\Lambda''}$
  $\{P'[\overline{v_1}/\overline{x}][\overline{v_2}/\overline{y}][\overline{v_3}/\overline{z}]\}\ c'[\overline{v_1}/\overline{x}][\overline{v_2}/\overline{y}]\ \{Q'[\overline{v_1}/\overline{x}][\overline{v_2}/\overline{y}][\overline{v_3}/\overline{z}]\}_I$. This follows from the induction hypothesis.

- FVA proof rule.
  $P = P'[\overline{v_1}/\overline{x^{\mathrm{r}}}][\overline{v_2}/\overline{y^{\mathrm{g}}}][\overline{v_3}/\overline{z}] * (\{P'\}\ v(\overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}})\ \{Q'\}^l_I)$,

$c = v(\overline{v_1}; \overline{v_2})$,
$Q = Q'[\overline{v_1}/\overline{x^{\mathrm{r}}}][\overline{v_2}/\overline{y^{\mathrm{g}}}][\overline{v_3}/\overline{z}] * (\{P'\}\, v(\overline{x^{\mathrm{r}}}; \overline{y^{\mathrm{g}}})\, \{Q'\}_I^l)$
$\overline{z} = \mathrm{fv}(P' * Q') \setminus (\overline{x} \cup \overline{y})$
for some P', $\overline{v_1}$, $\overline{v_2}$, $\overline{v_3}$, $\overline{x^{\mathrm{r}}}$, $\overline{y^{\mathrm{g}}}$, v, Q'.

Choose arbitrary h such that $h \vDash_{F_\Lambda} P$. Choose arbitrary τ, v' such that $c \Downarrow \tau, v'$.

Choose arbitrary $\Lambda' \leq \Lambda$.

We have to prove $\mathrm{safe}'(\Lambda', h, \tau, I, Q[v'/\mathsf{res}])$.

Assume $\Lambda'.I > 0$ (the case $\Lambda'.I = 0$ is trivial).

Because $h \vDash_{F_\Lambda} P$ there is some h_P, h_H such that $h = h_P \uplus h_H$ and $h_P \vDash_{F_\Lambda} P'[\overline{v_1}/\overline{x^{\mathrm{r}}}][\overline{v_2}/\overline{y^{\mathrm{g}}}][\overline{v_3}/\overline{z}]$ and $h_H \vDash_{F_\Lambda} \{P'\}\, v(\overline{x}; \overline{y})\, \{Q'\}_I^l$.

Because of the latter: $\{P'\}\, v(\overline{x}; \overline{y})\, \{Q'\}_I^l \in F_\Lambda$.

Therefore:

$$v = \lambda \overline{x}; \overline{y}.\, c' \tag{B.21}$$

(for some c') and

$$\vDash^{\Lambda -_I 1} \{P'[\overline{v_1}/\overline{x}][\overline{v_2}/\overline{y}][\overline{v_3}/\overline{z}]\}\, c'[\overline{v_1}/\overline{x}][\overline{v_2}/\overline{y}]\, \{Q'[\overline{v_1}/\overline{x}][\overline{v_2}/\overline{y}][\overline{v_3}/\overline{z}]\}_I \tag{B.22}$$

Because $c \Downarrow \tau, v'$ and $c = v(\overline{v_1}; \overline{v_2})$ and (B.21), we know $\tau = (h_1, h_1) \cdot \tau'$ for some h_1, τ'.

Choose arbitrary h_r, h_I such that $h_1 = \mathrm{erat}(h \uplus h_I \uplus h_2)$ and $h_I \vDash_{F_{\Lambda'}} \circledast\mathrm{inv}(h \uplus h_I \uplus h_r, I)$.

It suffices to prove that $\mathrm{safe}(\Lambda' -_I 1, h, \tau', I, Q[v'/\mathsf{res}])$.

Because $(\lambda \overline{x}; \overline{y}.\, c')(\overline{v_1}; \overline{v_2}) \Downarrow \tau, v'$ and $\tau = (h_1, h_1) \cdot \tau'$, the step rules give us that $c'[\overline{v_1}/\overline{x}][\overline{v_2}/\overline{y}] \Downarrow \tau', v'$.

Combined with $h_P \vDash_{F_\Lambda} P'[\overline{v_1}/\overline{x^{\mathrm{r}}}][\overline{v_2}/\overline{y^{\mathrm{g}}}][\overline{v_3}/\overline{z}]$ and (B.22) we obtain: $\mathrm{safe}(\Lambda -_I 1, h_P, \tau', I, Q'[\overline{v_1}/\overline{x^{\mathrm{r}}}][\overline{v_2}/\overline{y^{\mathrm{g}}}][\overline{v_3}/\overline{z}][v'/\mathsf{res}])$.

Combine with $h_H \vDash_{F_{\Lambda -_I 1}} \{P'\}\, v(\overline{x}; \overline{y})\, \{Q'\}_I^l$ (because $h_H \vDash_{F_\Lambda} \{P'\}\, v\ \overline{x}; \overline{y}\, \{Q'\}_I^l$) and Lemma 4 on page 150 to obtain: $\mathrm{safe}(\Lambda -_I 1, h, \tau', I, Q[v'/\mathsf{res}])$, which we wanted to prove.

□

Lemma 44: Unsafe outcome

$\forall h, \tau, Q.\ \neg\mathrm{OK}(\tau) \Rightarrow \exists \Lambda. \neg\mathrm{safe}(\Lambda, \emptyset, \tau, \mathrm{outat}, \mathbf{emp})$

Proof. Because $\neg\text{OK}(\tau)$ we know there is some n such that $\emptyset, \tau \to (n, \bot)$.

It suffices to prove
$\forall n, h, h_R, \tau.\ \text{erat}(h) \uplus h_R, \tau \to (n, \bot) \Rightarrow \neg\text{safe}'((n+1, 0), h, \tau, \text{outat}, \mathbf{emp})$.

We prove this by induction on n.

- $n = 0$.

  Because $\text{erat}(h) \uplus h_R, \tau \to (0, \bot)$ there is some τ' such that $\tau = (\text{erat}(h) \uplus h_R, \bot) \cdot \tau'$.

  $\neg\text{safe}((1, 0), h, \tau, \text{outat}, \mathbf{emp})$ then follows directly from the definition of safe$'$.

- $n > 0$.

  The induction hypothesis states that
  $\forall h, h_R, \tau.\ \text{erat}(h) \uplus h_R, \tau \to (n-1, \bot) \Rightarrow \neg\text{safe}((n, 0), h, \tau, \text{outat}, \mathbf{emp})$.

  Proof by contradiction: assume

  $$\text{safe}((n+1, 0), h, \tau, \text{outat}, \mathbf{emp}). \tag{B.23}$$

  Because $\text{erat}(h) \uplus h_R, \tau \to (n, \bot)$ and $n > 0$, there is some h_2, τ' such that $\tau = (\text{erat}(h) \uplus h_R, h_2) \cdot \tau'$.

  Choose arbitrary h_I, h_r such that $\text{erat}(h) \uplus h_R = \text{erat}(h \uplus h_I \uplus h_r)$ and $h_I \vDash_{F_{(n+1,0)}} \circledast\text{inv}(h \uplus h_I \uplus h_r, \text{outat})$.

  Because (B.23) there is some h', h'_I such that $h_2 = \text{erat}(h' \uplus h_I \uplus h_r)$ and $\text{safe}((n, 0), h', \tau, \text{outat}, \mathbf{emp})$.

  Let $h'_R = \text{erat}(h'_I \uplus h_r)$

  Note that $h_2 = \text{erat}(h') \uplus h'_R$ and $\text{erat}(h') \uplus h'_R, \tau \to (n-1, \bot)$.

  Therefore we can apply the induction hypothesis to obtain $\neg\text{safe}((n, 0), h', \tau, \text{outat}, \mathbf{emp})$.

  We have reached a contradiction with (B.23).

□

Theorem 9: Soundness (in-memory)

$\forall c.\ \vdash \{\mathbf{emp}\}\, c\, \{\mathbf{emp}\}^{\text{r}}_{\text{outat}} \Rightarrow$
$\quad \forall \tau, v.\ \text{er}(c) \Downarrow \tau, v \Rightarrow$
$\quad \text{OK}(\tau)$

Proof. Choose arbitrary c, τ, v such that $\vdash \left\{\mathbf{emp}\right\} c \left\{\mathbf{emp}\right\}_{\text{outat}} \wedge \text{er}(c) \Downarrow \tau, v$. We want to prove that $\text{OK}(\tau)$.

Proof by contradiction: assume $\neg\text{OK}(\tau)$.

Because of Lemma 10 on page 151, we know that there is some τ', v' such that $c \Downarrow \tau', v' \wedge \neg\text{OK}(\tau')$.

Because Lemma 9 on page 151: $\exists\Lambda, \Lambda.\neg\text{safe}(\Lambda, \emptyset, \tau', \emptyset, \mathbf{emp})$.

Because of Lemma 8 on page 151: $\models \{P\}\, c\, \{Q\}$.

Using the definition of validity of a Hoare triple, we obtain:
$\forall\Lambda.\ \text{safe}(\Lambda, \emptyset, \tau', \emptyset, \mathbf{emp})$.

We have reached a contradiction. $\square$

B.2 Recursion

We want to prove the derived inference rule of Fig. 4.15 p. 143, which we repeat here for convenience:

$$\dfrac{\begin{array}{c}\forall \overline{v'_x}, \overline{v'_y}, \overline{v'_z}.\ \vdash \left\{\begin{array}{c} P[\overline{v'_x}/\overline{x}] \\ [\overline{v'_y}/\overline{y}] \\ [\overline{v'_z}/\overline{z}] \\ * A(A) \end{array}\right\} c[G, \overline{v'_x}/g, \overline{x}][\overline{v'_y}/\overline{y}] \left\{\begin{array}{c} Q[\overline{v'_x}/\overline{x}] \\ [\overline{v'_y}/\overline{y}] \\ [\overline{v'_z}/\overline{z}] \end{array}\right\}_I^l \\ A = \lambda a.\ \left\{g = G * a(a) * P\right\} g(g, \overline{x}; \overline{y}) \left\{Q\right\}_I^l \\ \overline{z} = \text{fv}(P * Q) \setminus (\overline{x} \cup \overline{y}) \\ \{g, a\} \cap \overline{z} = \emptyset \\ \mathsf{res} \notin \text{fv}(P) \\ G = \lambda g, \overline{x}; \overline{y}.\, c \end{array}}{\vdash \left\{\mathbf{emp}\right\} \lambda\overline{x}; \overline{y}.\, \mathbf{let}\ g = \lambda g, \overline{x}; \overline{y}.\, c\ \mathbf{in}\ g(g, \overline{x}; \overline{y}) \left\{\left\{P\right\} \mathsf{res} \left\{Q\right\}_I^l\right\}_I^l}\ \text{Rec}$$

So given the premises, we want to prove
$\vdash \left\{\mathbf{emp}\right\} \lambda\overline{x}; \overline{y}.\, \mathbf{let}\ g = \lambda g, \overline{x}; \overline{y}.\, c\ \mathbf{in}\ g(g, \overline{x}; \overline{y}) \left\{\left\{P\right\} \mathsf{res} \left\{Q\right\}_I^l\right\}_I^l$.

Apply the FVI proof rule (backwards). It suffices to prove

$$\forall \overline{v_x}, \overline{v_y}, \overline{v_z}.\ \vdash \left\{\begin{matrix} P[\overline{v_x}/\overline{x}] \\ [\overline{v_y}/\overline{y}] \\ [\overline{v_z}/\overline{z}] \end{matrix}\right\} \textbf{let } g = \lambda g, \overline{x}; \overline{y}.\, c \textbf{ in } g(g, \overline{v_x}; \overline{v_y}) \left\{\begin{matrix} Q[\overline{v_x}/\overline{x}] \\ [\overline{v_y}/\overline{y}] \\ [\overline{v_z}/\overline{z}] \end{matrix}\right\}_I^l$$

Here, $\overline{z}$ are the variables that are free in P and/or in Q, and furthermore are different from $\overline{x}$ and $\overline{y}$.

Note that A is a predicate value, and that $A(A)$ is an assertion.

To gain better insight, let us study when $A(A)$ is true for a heap, and see how we do not get into an infinite loop when doing so.

Using the Pred inference rule of Fig. 4.5 on page 110, $h \vDash_F A(A)$ is true if

$$h \vDash_F \{g = G * A(A) * P\}\, g(g, \overline{x}; \overline{y})\, \{Q\}_I^l \tag{B.24}$$

is true. To know when (B.24) is true, we can not use the Pred inference rule another time, because now the assertion

$$\{g = G * A(A) * P\}\, g(g, \overline{x}; \overline{y})\, \{Q\}_I^l \tag{B.25}$$

is not a predicate value. So we do not get stuck into an infinite loop of applying the Pred inference rule over and over.

The assertion (B.25) is a Hoare triple assertion, so (B.24) is true if $\{g = G * A(A) * P\}\, g(g, \overline{x}; \overline{y})\, \{Q\}_I^l \in F$, according to the Fun inference rule (see Fig. 4.12 p. 132).

Note that the assertion (B.25) contains a predicate value application ($A(A)$) in the precondition of the Hoare triple assertion. This predicate value application is an assertion itself: it is written in the syntax of the assertion language. One can not apply it at the meta level.

Back to proving the derived inference rule. Choose arbitrary $\overline{v_x}, \overline{v_y}, \overline{v_z}$.

Let $P' = P[\overline{v_x}/\overline{x}][\overline{v_y}/\overline{y}][\overline{v_z}/\overline{z}]$.

Let $Q' = Q[\overline{v_x}/\overline{x}][\overline{v_y}/\overline{y}][\overline{v_z}/\overline{z}]$.

We apply the Let proof rule (again backwards). We now have to prove

- $\vdash \{P'\}\, \lambda g, \overline{x}; \overline{y}.\, c\, \{P' * \mathsf{res} = \lambda g, \overline{x}; \overline{y}.\, c * A(A)\}_I^l$
  Use the Frame rule. Now we have to prove
  $\vdash \{\mathbf{emp}\}\, \lambda g, \overline{x}; \overline{y}.\, c\, \{\mathsf{res} = \lambda g, \overline{x}; \overline{y}.\, c * A(A)\}_I^l$
  Use the FVI rule. Now we have to prove

$\forall v_g, \overline{v'_x}, \overline{v'_y}, \overline{v'_z}.\ \vdash$

$$\left\{\begin{matrix} P[v_g, \overline{v'_x}/g, \overline{x}] \\ [\overline{v'_y}/\overline{y}] \\ [\overline{v'_z}/\overline{z}] \\ * \, v_g = G * A(A) \end{matrix}\right\} c[v_g, \overline{v'_x}/g, \overline{x}][\overline{v'_y}/\overline{y}] \left\{\begin{matrix} Q[v_g, \overline{v'_x}/g, \overline{x}] \\ [\overline{v'_y}/\overline{y}] \\ [\overline{v'_z}/\overline{z}] \\ \end{matrix}\right\}_I^l$$

Remember that $\overline{z}$ are the free variables in P and/or Q that are not in $\overline{x}$ and not in $\overline{y}$.

We can simplify this (use that g is not a free variable in P and also not in Q): $\forall \overline{v'_x}, \overline{v'_y}, \overline{v'_z}.\ \vdash$
$\left\{A(A) * P[\overline{v'_x}/\overline{x}][\overline{v'_y}/\overline{y}][\overline{v'_z}/\overline{z}]\right\} c[G, \overline{v'_x}/g, \overline{x}][\overline{v'_y}/\overline{y}] \left\{Q[\overline{v'_x}/\overline{x}][\overline{v'_y}/\overline{y}][\overline{v'_z}/\overline{z}]\right\}_I^l$
which is given.

- $\forall v_g.\ \vdash \left\{v_g = G * P' * A(A)\right\} v_g(v_g, \overline{v_x}; \overline{v_y}) \left\{Q'\right\}_I^l$

  Choose arbitrary v_g. If $v_g \neq G$ use the False proof rule. Otherwise, we have to prove: $\vdash \left\{G = G * P' * A(A)\right\} G(G, \overline{v_x}; \overline{v_y}) \left\{Q'\right\}_I^l$

  Rewrite this (use the Conseq rule):
  $\vdash \left\{G = G * P' * A(A) * A(A)\right\} G(G, \overline{v_x}; \overline{v_y}) \left\{Q'\right\}_I^l$

  Rewrite again: $\vdash$

$$\left\{\begin{matrix} (g = G * P * A(A))[G, \overline{v_x}/g, \overline{x}][\overline{v_y}/\overline{y}][\overline{v_z}/\overline{z}] \\ * \, A(A) \\ \end{matrix}\right\} G(G, \overline{v_x}; \overline{v_y}) \left\{\begin{matrix} Q[G, \overline{v_x}/g, \overline{x}] \\ [\overline{v_y}/\overline{y}] \\ [\overline{v_z}/\overline{z}] \end{matrix}\right\}_I^l$$

  Apply the FVA rule.

Bibliography

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, 2006.

[2] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV 2007*, volume 4590 of *LNCS*, pages 178–192, Heidelberg, 2007. Springer.

[3] J. Berdine, C. Calcagno, and P. W. O'Hearn. *Symbolic Execution with Separation Logic*, pages 52–68. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[4] J. Berdine, C. Calcagno, and P. O'hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.

[5] G. Beuster, N. Henrich, and M. Wagner. Real world verification – Experiences from the Verisoft email client. In G. Sutcliffe, R. Schmidt, and S. Schulz, editors, *Proceedings of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning (ESCoR 2006)*, volume 192 of *CEUR Workshop Proceedings*, pages 112–125. CEUR-WS.org, Aug. 2006.

[6] B. Bogaerts, J. Jansen, M. Bruynooghe, B. De Cat, J. Vennekens, and M. Denecker. Simulating dynamic systems using linear time calculus theories. *Theory and Practice of Logic Programming*, 14:477–492, 7 2014.

[7] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accouting in separation logic. In *POPL*, 2005.

[8] J. Boyland. Checking interference with fractional permissions. In *International Static Analysis Symposium*, pages 55–72. Springer, 2003.

[9] R. W. Butler and G. B. Finelli. The infeasibility of experimental quantification of life-critical software reliability. *SIGSOFT Softw. Eng. Notes*, 16(5):66–76, Sept. 1991.

[10] R. Calinescu and B. Rumpe, editors. *Software Engineering and Formal Methods - 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings*, volume 9276 of *Lecture Notes in Computer Science*. Springer, 2015.

[11] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *SOSP '01*, pages 73–88, New York, 2001. ACM.

[12] D. Clarke, T. Wrigstad, and J. Noble, editors. *Aliasing in Object-Oriented Programming*. Springer-Verlag Berlin Heidelberg, 2013.

[13] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[14] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1981. Springer-Verlag.

[15] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. Model checking and the state explosion problem. In B. Meyer and M. Nordio, editors, *Tools for Practical Software Verification*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer Berlin Heidelberg, 2012.

[16] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs '09*, volume 5674 of *LNCS*, pages 23–42, Heidelberg, 2009. Springer.

[17] J. Corbet, A. Rubini, and G. Kroah-Hartmann. *Linux Device Drivers*. O'Reilly, 3rd edition, 2005.

[18] E. W. Dijkstra. Guarded commands, non-determinacy and a calculus for the derivation of programs. In *Language Hierarchies and Interfaces*, pages 111–124. Springer, 1976.

[19] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *European Conference on Object-Oriented Programming*, pages 504–528. Springer, 2010.

[20] D. Distefano and M. J. Parkinson J. jstar: Towards practical verification for Java. In *ACM Sigplan Notices*, volume 43, pages 213–226. ACM, 2008.

[21] P. Ducklin. Anatomy of a “goto fail” – Apple’s SSL bug explained, plus an unofficial patch for OS X! `https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/`.

[22] Federal Aviation Administration. `https://www.gpo.gov/fdsys/pkg/FR-2015-05-01/pdf/2015-10066.pdf`.

[23] U. I. Forum. Universal Serial Bus specification revision 2.0, 2000.

[24] U. I. Forum. Device class definition for Human Interface Devices (HID), 2001.

[25] U. I. Forum. Universal Serial Bus mass storage class specification overview, 2010.

[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[27] S. Gibbs. US aviation authority: Boeing 787 bug could cause ’loss of control’. 05 2016. `https://www.theguardian.com/business/2015/may/01/us-aviation-authority-boeing-787-dreamliner-bug-could-cause-loss-of-control`.

[28] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *APLAS’07*, 2007.

[29] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. M. Petters. Towards trustworthy computing systems: taking microkernels to the next level. *SIGOPS Oper. Syst. Rev.*, 41:3–11, July 2007.

[30] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV ’02*, volume 2402 of *LNCS*, pages 382–399, Heidelberg, 2002. Springer.

[31] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, 1969.

[32] Inria. The Coq proof assistant. `https://coq.inria.fr/`.

[33] B. Jacobs, D. Bosnacki, and R. Kuiper. Modular Termination Verification. In J. T. Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 664–688, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[34] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods 2011*, volume 6617 of *LNCS*, pages 41–55, Heidelberg, 2011. Springer.

[35] B. Jacobs, J. Smans, and F. Piessens. The VeriFast program verifier: A tutorial. `https://people.cs.kuleuven.be/~bart.jacobs/verifast/tutorial.pdf`.

[36] B. Jacobs, J. Smans, and F. Piessens. The VeriFast program verifier: A tutorial. `https://doi.org/10.5281/zenodo.887907`.

[37] B. Jacobs (editor). Verifast 17.06, June 2017. `https://doi.org/10.5281/zenodo.819853`.

[38] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 637–650, New York, NY, USA, 2015. ACM.

[39] M. Kim and Y. Kim. Concolic testing of the multi-sector read operation for flash memory file system. In *SBMF '09*, volume 5902 of *LNCS*, pages 251–265, Heidelberg, 2009. Springer.

[40] L. M. Kristensen, S. Christensen, and K. Jensen. The practitioner's guide to coloured Petri nets. *International Journal on Software Tools for Technology Transfer*, 2:98–132, 1998.

[41] J. Leyden. Annus horribilis for TLS! All the bigguns now officially pwned in 2014. 11 2014. `http://www.theregister.co.uk/2014/11/12/ms_crypto_library_megaflaw/`.

[42] R. Love. *Linux Kernel Development*. Novell Press, second edition, 2005.

[43] Microsoft. Vulnerability in Schannel could allow remote code execution, 2014. `https://technet.microsoft.com/library/security/MS14-066`.

[44] Ministry of Land, Infrastructure, Transport and Tourism. City heights Takeshiba elevator accident investigation report (translated title). `http://www.mlit.go.jp/common/000048837.pdf`.

[45] J. T. Mühlberg and G. Lüttgen. BLASTing Linux code. In *FMICS '06*, volume 4346 of *LNCS*, pages 211–226, Heidelberg, 2007. Springer.

[46] J. T. Mühlberg and G. Lüttgen. Verifying compiled file system code. In *SBMF '09*, volume 5902 of *LNCS*, pages 306–320, Heidelberg, 2009. Springer.

[47] K. Nakata and T. Uustalu. A Hoare logic for the coinductive trace-based big-step semantics of While. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, pages 488–506, Berlin, Heidelberg, 2010. Springer-Verlag.

[48] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *ACM SIGPLAN International Conference on Functional Programming*, 2008.

[49] P. O'Hearn. A primer on separation logic (and automatic program verification and analysis). In T. Nipkow, O. Grumberg, and B. Hauptmann, editors, *Software Safety and Security; Tools for Analysis and Verification*, number 33 in NATO Science for Peace and Security Series. IOS Press, 2012. Marktoberdorf Summer School 2011 Lecture Notes.

[50] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.

[51] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[52] W. Oortwijn, M. Huisman, S. Blom, M. Zaharieva-Stojanovski, and D. Gurov. An abstraction technique for describing concurrent program behaviour. In *9th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE) 2017*. To appear. `http://wwwhome.ewi.utwente.nl/~marieke/vstte2017.pdf`.

[53] OpenSSL. TLS heartbeat read overrun. `https://www.openssl.org/news/secadv/20140407.txt`.

[54] M. Parkinson and G. Bierman. Separation logic and abstraction. In *Proceedings of the 32nd Symposium on Principles of Programming Languages*, pages 247–258. ACM, 2005.

[55] W. Penninckx and B. Jacobs. A higher-order-ish logic with prophecies for concurrency verification. To be submitted.

[56] W. Penninckx and B. Jacobs. I/O style verification of memory-manipulating programs. To be submitted.

[57] W. Penninckx and B. Jacobs. Sound, modular and compositional verification of the input/output behavior of programs: extended version. To be submitted.

[58] W. Penninckx and B. Jacobs. Coq formalization and soundness proof for an input/output verification approach, Dec. 2016. `https://doi.org/10.5281/zenodo.887578`.

[59] W. Penninckx, B. Jacobs, and F. Piessens. Modular, compositional and sound verification of the input/output behavior of programs. CW Reports CW663, Department of Computer Science, KU Leuven, May 2014.

[60] W. Penninckx, B. Jacobs, and F. Piessens. Sound, modular and compositional verification of the input/output behavior of programs. In J. Vitek, editor, *Programming Languages and Systems, European Symposium on Programming (ESOP 2015), London, UK, 14-16 April 2015*, pages 158–182. Springer Berlin Heidelberg, Apr. 2015.

[61] W. Penninckx, J. T. Mühlberg, J. Smans, B. Jacobs, and F. Piessens. Sound formal verification of Linux's USB BP keyboard driver. In *NASA Formal Methods*, volume 7226, pages 210–215. Springer, April 2012.

[62] P. Philippaerts, J. T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens. Software verification with VeriFast: Industrial case studies. *Science of Computer Programming*, 82(1):77–97, March 2014.

[63] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. *Software Foundations.* `https://www.cis.upenn.edu/~bcpierce/sf`.

[64] R. Piskac, T. Wies, and D. Zufferey. Grasshopper: Complete heap verification with mixed specifications. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 124–139. Springer, 2014.

[65] H. Post, C. Sinz, and W. Küchlin. Towards automatic software model checking of thousands of Linux modules – a case study with Avinux. *Softw. Test. Verif. Reliab.*, 19:155–172, 2009.

[66] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[67] F. Redmill. Understanding the use, misuse and abuse of safety integrity levels. `https://www.cems.uwe.ac.uk/~a2-lenz/n-gunton/worksheets/3A.SILs.pdf`.

[68] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Symposium on Logic in Computer Science*, pages 55–74, Washington, 2002. IEEE.

[69] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02*, pages 55–74, Washington, 2002. IEEE.

[70] A. J. Turon and M. Wand. A separation logic for refining concurrent objects. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 247–258, New York, NY, USA, 2011. ACM.

[71] S. Vankateswaren. *Essential Linux Device Drivers.* Prentice Hall, 2008.

[72] F. Vogels, B. Jacobs, and F. Piessens. Featherweight verifast. *Logical Methods in Computer Science*, 11(3), 2015.

[73] F. Vogels, B. Jacobs, F. Piessens, and J. Smans. Annotation inference for separation logic based verifiers. In *FMOODS 2011*, volume 6722 of *LNCS*, pages 319–333, Heidelberg, 2011. Springer.

[74] R. Wisnesky, G. Malecha, and G. Morrisett. Certified web services in Ynot. In *5th International Workshop on Automated Specification and Verification of Web Systems*, July 2009.

[75] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent Linux device drivers. In *ASE '07*, pages 501–504, New York, 2007. ACM.

[76] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *CAV 2008*, volume 5123 of *LNCS*, pages 385–398, Heidelberg, 2008. Springer.

[77] M. Zhivich and R. K. Cunningham. The real cost of software errors. *IEEE Security Privacy*, 7(2):87–90, March 2009.

List of publications

Articles to be submitted

- W. Penninckx and B. Jacobs. Sound, modular and compositional verification of the input/output behavior of programs: extended version. To be submitted
- W. Penninckx and B. Jacobs. I/O style verification of memory-manipulating programs. To be submitted
- W. Penninckx and B. Jacobs. A higher-order-ish logic with prophecies for concurrency verification. To be submitted

Articles in internationally reviewed academic journals

- P. Philippaerts, J. T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens. Software verification with VeriFast: Industrial case studies. *Science of Computer Programming*, 82(1):77–97, March 2014

Papers at international scientific conferences and symposia, published in full in proceedings

- B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods 2011*, volume 6617 of *LNCS*, pages 41–55, Heidelberg, 2011. Springer
- W. Penninckx, J. T. Mühlberg, J. Smans, B. Jacobs, and F. Piessens. Sound formal verification of Linux's USB BP keyboard driver. In *NASA Formal Methods*, volume 7226, pages 210–215. Springer, April 2012

- W. Penninckx, B. Jacobs, and F. Piessens. Sound, modular and compositional verification of the input/output behavior of programs. In J. Vitek, editor, *Programming Languages and Systems, European Symposium on Programming (ESOP 2015), London, UK, 14-16 April 2015*, pages 158–182. Springer Berlin Heidelberg, Apr. 2015

Internal reports

- W. Penninckx, B. Jacobs, and F. Piessens. Modular, compositional and sound verification of the input/output behavior of programs. CW Reports CW663, Department of Computer Science, KU Leuven, May 2014

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
IMEC-DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Leuven
https://www.cs.kuleuven.be