Logical Methods in Computer Science
Vol. 11(3:19)2015, pp. 1–57
www.lmcs-online.org

Submitted Aug. 18, 2014
Published Sep. 22, 2015

FEATHERWEIGHT VERIFAST

FRÉDÉRIC VOGELS, BART JACOBS, AND FRANK PIESSENS

iMinds-DistriNet, Dept. C.S., KU Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium
e-mail address: {frederic.vogels, bart.jacobs, frank.piessens}@gmail.com

Abstract. VeriFast is a leading research prototype tool for the sound modular verification of safety and correctness properties of single-threaded and multithreaded C and Java programs. It has been used as a vehicle for exploration and validation of novel program verification techniques and for industrial case studies; it has served well at a number of program verification competitions; and it has been used for teaching by multiple teachers independent of the authors.

However, until now, while VeriFast's operation has been described informally in a number of publications, and specific verification techniques have been formalized, a clear and precise exposition of how VeriFast works has not yet appeared.

In this article we present for the first time a formal definition and soundness proof of a core subset of the VeriFast program verification approach. The exposition aims to be both accessible and rigorous: the text is based on lecture notes for a graduate course on program verification, and it is backed by an executable machine-readable definition and machine-checked soundness proof in Coq.

Introduction

For many classes of safety-critical or security-critical programs, such as operating system components, internet infrastructure, or embedded software, conventional quality assurance approaches such as testing, code review, or even model checking are insufficient to detect all bugs and achieve good confidence in their safety and security; for these programs, the newer technique of modular formal verification may be the most promising approach.

VeriFast is a sound modular formal verification approach for single-threaded and multithreaded imperative programs being developed at KU Leuven. The prototype tool that implements this approach[1] takes as input a C or Java program annotated with preconditions, postconditions, loop invariants, data structure definitions, and proof hints written in a variant of separation logic [39, 42], and symbolically executes each function/method. It either reports "0 errors found" or the source location of a potential error. If it reports "0 errors found", it is guaranteed (modulo bugs in the tool) that no execution of the program

2012 ACM CCS: [**Theory of computation**]: Logic—Logic and verification / Programming logic; Semantics and reasoning—Program reasoning.

Key words and phrases: program verification, separation logic, symbolic execution.

[1]Download it from `http://distrinet.cs.kuleuven.be/software/VeriFast/`.

will a) perform an illegal memory access such as a null pointer dereference, an access of unallocated memory, or an access of an array outside of its bounds; b) perform a data race, where two threads access the same variable concurrently without synchronization, and at least one access is a write operation; c) violate the user-specified function/method contracts or the contracts of the library or API functions/methods used by the program. If it reports an error, it shows a symbolic execution trace that leads to the error, including the symbolic state (store, heap, and path condition) at each step.

VeriFast has served as a vehicle for exploration and validation of a number of novel program verification techniques [26, 29, 49, 48] and for a number of industrial case studies [41]; it has served well at a number of program verification competitions[2]; and it has been used for teaching program verification by the authors as well as by independent instructors at other institutions[3].

Until now, while VeriFast's operation has been described informally in a number of publications [28, 27, 44], and specific verification techniques have been formalized [26, 29, 49, 48], a clear and precise exposition of how VeriFast works has not yet appeared.

In this article, we present a formal definition of a simplified version of the VeriFast program verification approach, called Featherweight VeriFast, as well as an outline for a proof of the soundness of this approach, *i.e.* that if verification of a program succeeds, then no execution of the program accesses unallocated memory. Featherweight VeriFast targets a simple toy programming language with routines, loops, and dynamic memory allocation and deallocation, and supports routine contracts, loop invariants, separation logic predicates, and symbolic execution. It captures some of the core aspects of the C programming language, but leaves out many complexities, including advanced concepts such as function pointers and concurrency, even though these are supported by VeriFast [26, 29]. The running example (introduced on p. 4) builds a large linked list in one routine and tears it down in another one; another example that appears (on p. 46) is the in-place reversal of a linked list. We use Featherweight VeriFast to verify the safety of both examples.

We hope that the definitions in this article are clear and the proofs are convincing; however, to address any shortcomings in this regard, we developed a machine-readable executable definition and machine-checked soundness proof of a slight variant of Featherweight VeriFast, called Mechanised Featherweight VeriFast, in the Coq proof assistant. It is available at `http://www.cs.kuleuven.be/~bartj/fvf/`. Furthermore, the executable nature of the definitions allowed us to test for errors in our programming language semantics and to verify that the formalized verification algorithm succeeds in verifying the example programs.

The structure of the article is as follows. In Section 1, we define the syntax of the input programming language, and we illustrate it with a few example programs. In Section 2, we illustrate and define the *concrete execution* of programs in this programming language. In Sections 3 and 4, we gradually introduce the VeriFast verification approach: in Section 4, we present the approach itself, called *symbolic execution*[4]; in Section 3, we present an intermediate type of execution, called *semiconcrete execution*, that sits between concrete execution and symbolic execution, which introduces some but not all features of the VeriFast approach. In Section 5, we discuss Mechanised Featherweight VeriFast. We end the article with an overview of related work in Section 6 and a conclusion in Section 7.

[2]See Endnote (a).

[3]See Endnote (b).

[4]We use this term in this article to denote the specific algorithm implemented by VeriFast. It is an instance of the general approach known in the literature as symbolic execution.

This article is based on a slide deck and lecture notes for a graduate course on program verification, and aims to be usable as an introduction to program verification. In the course, theory lectures based on this material are interleaved with hands-on lab sessions based on the VeriFast Tutorial [30]. Acknowledgements of related work are deferred to Section 6.

1. The Programming Language

In this section, we define the syntax of programs and then show an example program.

1.1. Syntax of Programs.

The programming language is as follows. An integer expression e is either an integer literal z, a variable x, an addition $e + e$, or a subtraction $e - e$. A boolean expression b is either an equality comparison $e = e$, a less-than comparison $e < e$, or a negation $\neg b$ of another boolean expression. A command c is either an assignment $x := e$ of an integer expression e to a variable x, a sequential composition $(c; c)$ of two commands (whose execution proceeds by first executing the first command and then the second command), a conditional command **if** b **then** c **else** c, a while loop **while** b **do** c, a routine call $r(\overline{e})$ (which calls routine r with argument list $\overline{e}$ (a line over a letter means a list of the things denoted by the letter))[5], a heap memory block allocation $x := \textbf{malloc}(n)$ (which allocates a block of heap memory of size n and stores the address of the new block in variable x), a memory read $x := [e]$ (which reads the value of the memory cell whose address is given by e and stores it in variable x), a memory write $[e] := e$ (which writes the value of the second expression into the memory cell whose address is given by the first expression), or a deallocation command $\textbf{free}(e)$ which releases the memory block allocated by **malloc** whose address is given by e. A routine definition *rdef* is of the form **routine** $r(\overline{x}) = c$ which declares $\overline{x}$ as the parameter list and c as the body of routine r.

Definition 1.1. Syntax of Programs

$$\begin{array}{rl}
 & z \in \mathbb{Z}, n \in \mathbb{N} \\
 & x \in \mathit{Vars} \\
e ::= & z \mid x \mid e + e \mid e - e \\
b ::= & e = e \mid e < e \mid \neg b \\
c ::= & x := e \mid (c; c) \mid \textbf{if } b \textbf{ then } c \textbf{ else } c \mid \textbf{while } b \textbf{ do } c \\
 & \mid r(\overline{e}) \mid x := \textbf{malloc}(n) \mid x := [e] \mid [e] := e \mid \textbf{free}(e) \\
\mathit{rdef} ::= & \textbf{routine } r(\overline{x}) = c
\end{array}$$

1.2. Example Program.

The example program of Figure 1 consists of routine definitions for routines `range` and `dispose` and a main command. Routine `range` has parameters `i`, `n`, and `result`; it builds a linked list that stores the integers from `i`, inclusive, to `n`, exclusive, and writes the address of the new linked list into the memory cell whose address is given by `result`. If `i` equals `n`, the value 0 is written to address `result`, denoting the empty linked list. Otherwise, a new linked list node is allocated with two fields; the first field holds the value of the node, and the second field holds the address of the next node. A recursive call of routine `range` is used to build the remaining nodes of the linked list.

[5]In this simple language, routines have no return value. A routine can pass a result to its caller by taking an address where the result should be stored as an argument.

```
routine range(i, n, result) =
   if i = n then
      [result] := 0
   else (
      head := malloc(2);
      [result] := head;
      [head] := i;
      range(i + 1, n, head + 1)
   )

routine dispose(list) =
   if list = 0 then
      dummy := dummy
   else (
      tail := [list + 1];
      free(list);
      dispose(tail)
   )

cell := malloc(1); range(0, 100000000, cell);
list := [cell]; free(cell); dispose(list)
```

Figure 1: Example Program

Routine dispose has the single parameter list. It frees the nodes of the linked list pointed to by list. If list is 0, this means the linked list is empty and nothing needs to be done. (Since in this programming language, each **if** command must specify a command for the **then** branch and for the **else** branch, we specify the command dummy := dummy for the **then** branch, which has no effect.) Otherwise, the first node is freed and then a recursive call of dispose is used to free the remaining nodes.

The main program calls range to build a linked list holding the numbers 0 through 99999999. Before doing so, however, it allocates a memory cell to hold the address of the new list. After the range call, the address of the list is read from the cell, the cell is freed, and finally the list nodes are freed using a call of routine dispose.

The purpose of Featherweight VeriFast is to verify that programs, like this one, never *fail* (i.e. access unallocated memory), i.e. that no execution of the program fails. The example program has an infinite number of executions: for each possible address of each linked list node, there is a separate execution. In one execution, the first node is allocated at address 1000, the second node at address 2000, etc. In another execution, the first node is allocated at address 123, the second node at address 234, etc. Featherweight VeriFast must check that none of these infinitely many executions fail.

Note: in a language like Java, the precise address at which an object is allocated cannot influence program execution, since the program can only compare two object references for equality; it cannot compare an object reference with an integer, check if one reference is less than another one, use literal addresses as object references, etc. However, in C, as well as in the programming language which we defined above, this is possible, so it is possible to write programs that fail or not depending on the address picked by **malloc**. Here is such a program:

$$x := \mathbf{malloc}(1); [42] := 0$$

If, in a given execution of this program, the address picked by **malloc** happens to be 42, the execution completes normally; otherwise, it fails.

Note also: While this aspect of memory allocation is peculiar to C, the fact that the language contains *nondeterministic* constructs, i.e. constructs whose observable behavior is not uniquely determined by the language specification, is universal to all programming languages: any language construct that accepts user input or otherwise interacts with the

$// \ s = \mathbf{0}, h = \mathbf{0}$
$\mathsf{pair} := \mathbf{malloc}(2);$
$// \ s = \mathbf{0}[\mathsf{pair} := 100], h = \{\!\!\{\mathsf{mb}(100, 2), 100 \mapsto 42, 101 \mapsto 24\}\!\!\}$
$[\mathsf{pair}] := 0;$
$// \ s = \mathbf{0}[\mathsf{pair} := 100], h = \{\!\!\{\mathsf{mb}(100, 2), 100 \mapsto 0, 101 \mapsto 24\}\!\!\}$
$\mathbf{free}(\mathsf{pair})$
$// \ s = \mathbf{0}[\mathsf{pair} := 100], h = \mathbf{0}$

where

$$f[x := y] = \text{function update} = \lambda z. \begin{cases} y & \text{if } z = x \\ f(z) & \text{otherwise} \end{cases}$$
$$\mathbf{0} = \lambda x.\ 0 = \text{empty store} = \text{empty heap} = \{\!\!\{\}\!\!\} = \text{empty multiset}$$
$$\{\!\!\{e_1, \ldots, e_n\}\!\!\} = \mathbf{0} + \{\!\!\{e_1\}\!\!\} + \cdots + \{\!\!\{e_n\}\!\!\}$$
$$M + \{\!\!\{e\}\!\!\} = M[e := M(e) + 1]$$

Figure 2: Example concrete execution trace

environment is nondeterministic from a verification point of view, since it leads to multiple possible executions, all of which need to be checked.

The main point illustrated by the example is that a program may have infinitely many executions, each of which may be very long (or even infinitely long), and all of these need to be checked for failure. This is true in all programming languages. Clearly, it is inefficient or impossible to naively check each execution separately. VeriFast (and Featherweight VeriFast) perform *modular symbolic execution* to achieve efficiency. After we define concrete execution precisely in Section 2, we introduce the Featherweight VeriFast constructs for modularity in Section 3 and the symbolic execution in Section 4.

2. Concrete Execution

In this section, we provide a formal definition of the behavior of programs of our programming language. We first introduce the notion of *concrete execution states* by means of two examples of concrete execution *traces* (sequences of states reached during an execution). We then introduce the notion of *outcomes*, which we use to express failure, nontermination, and nondeterminism. Finally, we use these concepts to define concrete execution of commands and safety of a program, and we discuss the verification problem.

2.1. Small Example Concrete Execution Trace.

The small example program in Figure 2 allocates a memory block of size 2, initializes the first element of the block to 0, and then frees the block. An example *execution trace* of this program is shown in comments before and after the code lines. An execution trace is a sequence of *execution states* (or *states* for short). In our simple programming language, a state consists of a *store* s and a *heap* h. A store is a function that maps variables to their current values; a heap is a *multiset* (or *bag*) of *heap chunks*. A multiset is like a set, except that it may contain elements more than once. Mathematically, it is a function that maps each potential element to the number of times it occurs in the multiset. A heap chunk (in concrete executions) is either a *points-to chunk* $\ell \mapsto v$ denoting that there is an allocated memory cell at address ℓ whose current value is v, or a *malloc block chunk* $\mathsf{mb}(\ell, n)$ denoting that a memory block of size n was allocated at address ℓ by **malloc**, i.e. that the memory cells at addresses ℓ through $\ell + n - 1$ are part of a single block, which will be freed as one unit when **free** is called with argument ℓ.

The example programming language makes a few simplifications compared to real machine states: memory cells may store arbitrary integers, rather than just bytes, and memory addresses may be arbitrary positive integers, rather than being bounded by the size of installed memory (or the size of the address space).

The initial store maps all variables to zero; the initial heap contains no heap chunks, i.e. it contains all heap chunks zero times, so it is a function that maps all heap chunks to zero. We denote a function that maps all arguments to zero by $\mathbf{0}$.

In the example execution trace, the **malloc** operation allocates the new block at address 100. Therefore, in the execution state after the **malloc** operation, the store maps the target variable pair of the **malloc** operation to 100, and the heap contains three heap chunks: the two points-to chunks that correspond to the two memory cells that constitute the newly allocated block, and the malloc block chunk that records that these two memory cells are part of the same block. As in C, the initial contents of the newly allocated memory cells are arbitrary; in the example trace, the contents are 42 and 24. (All numbers that were picked arbitrarily are shown in orange, to highlight that the program has infinitely many other executions, that pick these numbers differently.)

The notation $f[a := b]$ denotes the function that is like f except that it maps argument a to value b. The notation $\{\!\{e_1, e_2\}\!\}$ denotes the multiset with elements e_1 and e_2 (where possibly $e_1 = e_2$). Formally, $\{\!\{e_1, \dots, e_n\}\!\} = \mathbf{0} + \{\!\{e_1\}\!\} + \cdots + \{\!\{e_n\}\!\}$, where $M + \{\!\{e\}\!\} = M[e := M(e) + 1]$; i.e. the multiset $M + \{\!\{e\}\!\}$ is like M except that element e occurs once more than in M.

The second command, which initializes the memory cell at address pair to zero, causes the state to change in just one place: the value of the points-to chunk with address 100 changes from 42 to 0.

Finally, the **free** command removes the three heap chunks from the heap and leaves it empty; it does not modify any variables so the store remains unchanged.

2.2. **Large Example Concrete Execution Trace.**

Example 2.1. Larger Example Concrete Execution Trace See Figure 3.

Now, let's look at an execution trace of routine range from the example program introduced earlier. This trace is part of a larger program execution trace. We look at a particular call of range. As shown in the first state of the trace, the values of parameters i, n, and r are 5, 8, and 41. That is, the caller is asking range to build a linked list with three nodes holding the values 5, 6, and 7, respectively, and to store the address of the newly built linked list in the previously allocated memory cell at address 41. At the time of the call, the heap consists of some chunks h_0 plus a points-to chunk with address 41 and value 77. (We use both $M + M'$ and $M \uplus M'$ for multiset union, defined as $M + M' = M \uplus M' = \lambda e.\ M(e) + M'(e)$.)

The first statement is the **if** statement. It checks if $\mathsf{i} = \mathsf{n}$. Since this is not the case, we skip the **then** branch and execute the **else** branch. The state upon arrival in the **else** branch is unchanged.

```
**routine** range(i, n, r) =
$s{:}\mathbf{0}[\mathsf{i}{:}5, \mathsf{n}{:}8, \mathsf{r}{:}41], h{:}h_0 \uplus \{\!\{41 \mapsto 77\}\!\}$
**if** i = n **then** l := 0 **else** (
$s{:}\mathbf{0}[\mathsf{i}{:}5, \mathsf{n}{:}8, \mathsf{r}{:}41], h{:}h_0 \uplus \{\!\{41 \mapsto 77\}\!\}$
l := **malloc**(2);
$s{:}\mathbf{0}[\mathsf{i}{:}5, \mathsf{n}{:}8, \mathsf{r}{:}41, \mathsf{l}{:}50], h{:}h_0 \uplus \{\!\{41 \mapsto 77, \mathsf{mb}(50, 2), 50 \mapsto 88, 51 \mapsto 99\}\!\}$
[l] := i; range(i + 1, n, l + 1)
   ⋮  (Execution of 3 nested range calls)
$s{:}\mathbf{0}[\mathsf{i}{:}5, \mathsf{n}{:}8, \mathsf{r}{:}41, \mathsf{l}{:}50], h{:}h_0 \uplus \{\!\{41 \mapsto 77, \mathsf{mb}(50, 2), 50 \mapsto 5, 51 \mapsto 60,$
   $\mathsf{mb}(60, 2), 60 \mapsto 6, 61 \mapsto 70, \mathsf{mb}(70, 2), 70 \mapsto 7, 71 \mapsto 0\}\!\}$
);
[r] := l
$s{:}\mathbf{0}[\mathsf{i}{:}5, \mathsf{n}{:}8, \mathsf{r}{:}41, \mathsf{l}{:}50], h{:}h_0 \uplus \{\!\{41 \mapsto 50, \mathsf{mb}(50, 2), 50 \mapsto 5, 51 \mapsto 60,$
   $\mathsf{mb}(60, 2), 60 \mapsto 6, 61 \mapsto 70, \mathsf{mb}(70, 2), 70 \mapsto 7, 71 \mapsto 0\}\!\}$
where $h_0 : \{\!\{\mathsf{mb}(30), 30 \mapsto 3, 31 \mapsto 40, \mathsf{mb}(40, 2), 40 \mapsto 4\}\!\}$
```

Figure 3: Larger Example Concrete Execution Trace

The execution of the **malloc** block is as before, in the simple example. In this trace, the block is allocated at address 50, and the initial values of the memory cells are 88 and 99.

Then, the first cell of the new block is initialized to i. We do not show the resulting state; only the value of the points-to chunk with address 50 changes.

Then, we get the recursive call of range to build the rest of the linked list. We do not show the execution states reached during the execution of this recursive call (which itself contains two more calls, one nested within the other); we skip directly to the state reached upon return from the call.

At this point, two more linked list nodes have been allocated, at addresses 60 and 70 (in this trace). Also, the linked list is well-formed: the second cell (which serves as the next field) of the node at address 50 points to the node at address 60, the next field of the node at address 60 points to the node at address 70, and the next field of the node at address 70 is a null pointer, indicating the end of the linked list.

The final command writes the address 50 of the newly built linked list to the address 41 provided by the caller; this modifies only the points-to chunk with address 41 in the heap.

The points to remember about this example trace are that it is long (since it contains three nested routine executions, which we did not show), that its states have large heaps with many chunks (here, up to 15 chunks, if we include h_0), and that routine range has infinitely many more execution traces like this one, that pick the numbers shown in orange differently. In subsequent sections, we will define alternative ways of executing programs where programs have fewer and shorter executions, and execution states have fewer heap chunks.

2.3. Concrete Execution States.

We define the set *CStates* of concrete execution states. The concrete stores *CStores* are the functions from variables to integers. The concrete predicates (i.e. the concrete chunk names) are the points-to predicate $\mapsto$ and the malloc block predicate mb. The set of concrete chunks is the set of expressions of the form $p(\ell, v)$, where p is a concrete predicate, and ℓ and v are integers. We call p the *name* of the chunk, and ℓ and v the *arguments* of the chunk. The concrete heaps are the multisets of concrete chunks. The concrete states are the pairs of concrete stores and concrete heaps.

We often use the alternative syntax $\ell \mapsto v$ for the points-to chunk $\mapsto(\ell, v)$.

Definition 2.2. Concrete Execution States

$$\begin{aligned}
\mathit{CStores} = &\quad \mathit{Vars} \to \mathbb{Z} \\
\mathit{CPredicates} = &\quad \{\mapsto, \mathsf{mb}\} \\
\mathit{CChunks} = &\quad \{p(\ell, v) \mid p \in \mathit{CPredicates}, \ell, v \in \mathbb{Z}\} \\
\mathit{CHeaps} = &\quad \mathit{CChunks} \to \mathbb{N} \\
\mathit{CStates} = &\quad \mathit{CStores} \times \mathit{CHeaps}
\end{aligned}$$
$$\ell \mapsto v \text{ is alternative syntax for } \mapsto(\ell, v)$$

2.4. **Outcomes.** In this subsection, we introduce the notion of *outcomes*, which we use to express failure, nontermination, and nondeterminism. We first introduce the various types of outcomes by example. We then provide formal definitions of outcomes, without or with *answers*. Finally, we define the concepts of satisfaction of a postcondition by an outcome, coverage of an outcome by another outcome, and sequential composition of outcomes; we state some properties; and we introduce some notations.

2.4.1. *Outcomes by Example.* To define mathematically what the concrete executions of a given program are, we define the function exec, which takes as arguments a command and an input state, and returns the *outcome* of executing the command starting in the given input state. In simple cases, such as in the case of the assignment command $\mathsf{p} := 42$, the outcome is a single output state: executing this assignment in the state $(\mathbf{0}, \mathbf{0})$ with an empty store (i.e. one that maps all variables to zero) and an empty heap results in the single output state where the store maps p to value 42 and all other variables to zero, and where the heap is still empty. We call such an outcome a *singleton outcome*, and we denote the singleton outcome with output state σ using angle brackets: $\langle\sigma\rangle$.

Example 2.3. Concrete Execution: Singleton Outcomes

$$\mathsf{exec}(\mathsf{p} := 42)((\mathbf{0}, \mathbf{0})) = \langle(\mathbf{0}[\mathsf{p} := 42], \mathbf{0})\rangle$$

Note: we define and use function exec in a *curried* form: instead of defining it as a function of two parameters (a command and an input state), we define it as a function of one parameter (a command) that returns another function of one parameter (an input state) which itself returns an outcome. We call the latter kind of function (a function that takes an input state and returns an outcome) a *mutator*. Therefore, exec is a function that maps commands to mutators.

Example 2.4. Concrete Execution: Demonic Choice

$$\begin{aligned}
\mathsf{exec}(\mathsf{p} := \mathbf{malloc}(0))((\mathbf{0}, \mathbf{0})) \; = & \\
& \langle(\mathbf{0}[\mathsf{p} := 1], \{\!\!\{\mathsf{mb}(1, 0)\}\!\!\})\rangle \\
\otimes \quad & \langle(\mathbf{0}[\mathsf{p} := 2], \{\!\!\{\mathsf{mb}(2, 0)\}\!\!\})\rangle \\
\otimes \quad & \langle(\mathbf{0}[\mathsf{p} := 3], \{\!\!\{\mathsf{mb}(3, 0)\}\!\!\})\rangle \\
\otimes \quad & \cdots
\end{aligned}$$

$$
\begin{aligned}
&\mathsf{exec}(\mathsf{p} := \mathbf{malloc}(1))((\mathbf{0},\mathbf{0})) \ = \\
&\quad \langle(\mathbf{0}[\mathsf{p} := 1], \{\!\!\{\mathsf{mb}(1,1), 1 \mapsto 0\}\!\!\})\rangle \\
&\qquad \otimes \langle(\mathbf{0}[\mathsf{p} := 1], \{\!\!\{\mathsf{mb}(1,1), 1 \mapsto 1\}\!\!\})\rangle \otimes \cdots \\
&\quad \otimes \langle(\mathbf{0}[\mathsf{p} := 2], \{\!\!\{\mathsf{mb}(2,1), 2 \mapsto 0\}\!\!\})\rangle \\
&\qquad \otimes \langle(\mathbf{0}[\mathsf{p} := 2], \{\!\!\{\mathsf{mb}(2,1), 2 \mapsto 1\}\!\!\})\rangle \otimes \cdots \\
&\quad \otimes \langle(\mathbf{0}[\mathsf{p} := 3], \{\!\!\{\mathsf{mb}(3,1), 3 \mapsto 0\}\!\!\})\rangle \\
&\qquad \otimes \langle(\mathbf{0}[\mathsf{p} := 3], \{\!\!\{\mathsf{mb}(3,1), 3 \mapsto 1\}\!\!\})\rangle \otimes \cdots \\
&\quad \otimes \cdots
\end{aligned}
$$

The outcome of executing a command is not always a single state. Specifically, consider **malloc** commands: the command $\mathsf{p} := \mathbf{malloc}(0)$ allocates a new memory block of size zero. This means that it does not allocate any memory cells, but it does create an mb chunk at an address that is different from the address of existing mb chunks. When starting from an empty heap, this address may be any positive integer. For every distinct address chosen, there is a different output state. Notice that this choice can be considered *demonic*: the program should not fail even if an attacker who tries to make the program fail makes this choice. Therefore, the outcome returned by exec is a *demonic choice* over the integers, where the chosen number is used as the address of the new block in the output state of a singleton outcome. That is, the *operands* of the demonic choice outcome in this example are singleton outcomes. The demonic choice between outcomes o_1 and o_2 is denoted as $o_1 \otimes o_2$.

In the case of the command $\mathsf{p} := \mathbf{malloc}(1)$, the outcome is a demonic choice over both the address of the new block and the initial value of the new memory cell.

Example 2.5. Concrete Execution: Failure, Nontermination, Angelic Choice

$$\mathsf{exec}([0] := 33)((\mathbf{0},\mathbf{0})) = \bot$$

$$
\begin{aligned}
&\mathsf{exec}(\mathsf{recurse}())((\mathbf{0},\mathbf{0})) = \top \\
&\quad \text{where } \mathbf{routine}\ \mathsf{recurse}() = \mathsf{recurse}()
\end{aligned}
$$

$$\mathsf{exec}(\mathbf{backtrack}(c_1, c_2))(\sigma) = \mathsf{exec}(c_1)(\sigma) \oplus \mathsf{exec}(c_2)(\sigma)$$

Singleton outcomes and demonic choices are not the only kinds of outcomes; there are three more kinds.

Consider the command $[0] := 33$ when executed in the empty heap. This is an access of an unallocated memory cell, i.e. it is a failure. We denote the failure outcome by the symbol for "bottom": $\bot$.

Consider the routine call $\mathsf{recurse}()$ and assume that routine $\mathsf{recurse}$ is defined such that its body is simply a recursive call of itself. This command performs an infinite recursion; it does not terminate.[6] Nontermination is often considered undesirable; however, in many other cases, it is intentional: for example, a web server or a database server is not supposed to terminate unless and until the user instructs it to do so. In any case, VeriFast and Featherweight VeriFast do not verify termination. Featherweight VeriFast verifies only the absence of accesses of unallocated memory, so from this point of view a nonterminating command is a good thing, since it prevents the remainder of the program from executing,

[6]In the case of a real programming language, this would lead to a stack overflow error at run time, except if the compiler performs tail recursion optimization. Neither VeriFast nor Featherweight VeriFast verify the absence of stack overflows, so we ignore this issue in this formalization.

including any commands that might fail. Therefore, we represent nontermination with the symbol for "top", $\top$, the opposite of $\bot$.

Finally, to round out the "algebra of outcomes", we introduce also *angelic choice*. True angelic choice does not occur in concrete executions of our programming language[7]; however, some real programming languages do have a form of angelic choice. For example, the logic programming language Prolog allows the user to specify multiple alternative ways to solve a problem. At run time, Prolog will try first the first alternative; if it fails, it restores the program state and then tries the second alternative. The program as a whole succeeds if either alternative succeeds: it is as if an angel chooses the right alternative. Another example is a transactional database: if a schedule fails, the state is rolled back and another schedule is attempted.

We introduce angelic choice here to obtain a nice, complete algebra, but also because we will use angelic choice in our definition of the Featherweight VeriFast verification algorithm.

In summary, (concrete) mutators are functions from (concrete) input states to outcomes over (concrete) output states. (Later we will also use outcomes over other state spaces.) The concrete execution function exec maps commands to concrete mutators.

Definition 2.6. Type of Concrete Execution

$$\begin{array}{rcl} \mathit{CMutators} & = & \mathit{CStates} \to \mathit{Outcomes}(\mathit{CStates}) \\ \mathsf{exec} & \in & \mathit{Commands} \to \mathit{CMutators} \end{array}$$

2.4.2. *Outcomes: Definition.* An outcome ϕ over a state space $\mathcal{S}$ is either a singleton outcome $\langle\sigma\rangle$, with $\sigma \in \mathcal{S}$, or a demonic choice $\bigotimes \Phi$ over the outcomes in Φ, or an angelic choice $\bigoplus \Phi$ over the outcomes in Φ, where Φ is a set of outcomes over $\mathcal{S}$. We denote the set of outcomes over state space $\mathcal{S}$ as $\mathit{Outcomes}(\mathcal{S})$. $\bigotimes \Phi$ and $\bigoplus \Phi$ are called *infinitary* demonic and angelic choice, since the set Φ is potentially infinite.

Binary demonic choice, binary angelic choice, nontermination, and failure can be defined as special cases of infinitary demonic choice and infinitary angelic choice: binary choices are choices over the set collecting the two alternatives; nontermination is a demonic choice over zero alternatives (the attacker is stuck with no alternatives, which is a good thing); failure is an angelic choice over zero alternatives (the angel is stuck with no alternatives, which is a bad thing).

Definition 2.7. Outcomes

$$\begin{array}{rcll} \phi & ::= & \langle\sigma\rangle & \text{singleton outcome} \\ & | & \bigotimes \Phi & \text{demonic choice} \\ & | & \bigoplus \Phi & \text{angelic choice} \end{array}$$

$$\begin{array}{rcl} \sigma \in \mathcal{S} & \Rightarrow & \langle\sigma\rangle \in \mathit{Outcomes}(\mathcal{S}) \\ \Phi \subseteq \mathit{Outcomes}(\mathcal{S}) & \Rightarrow & \bigotimes \Phi \in \mathit{Outcomes}(\mathcal{S}) \\ \Phi \subseteq \mathit{Outcomes}(\mathcal{S}) & \Rightarrow & \bigoplus \Phi \in \mathit{Outcomes}(\mathcal{S}) \end{array}$$

[7]A degenerate form of angelic choice does occur: failure is equivalent to angelic choice over zero alternatives, as we will see later.

$$\begin{array}{rcll} \phi_1 \otimes \phi_2 & = & \bigotimes\{\phi_1, \phi_2\} & \text{binary demonic choice} \\ \phi_1 \oplus \phi_2 & = & \bigoplus\{\phi_1, \phi_2\} & \text{binary angelic choice} \\ \top & = & \bigotimes \emptyset & \text{nontermination} \\ \bot & = & \bigoplus \emptyset & \text{failure} \end{array}$$

2.4.3. *Outcomes with Answers.* Often, it is useful to consider mutators that have not just an output state but also an *answer*. We denote a singleton outcome with output state $\sigma \in \mathcal{S}$ and answer $a \in \mathcal{A}$ by $\langle \sigma, a \rangle$. For uniformity, we treat outcomes without answers like outcomes whose answer is the *unit value* tt, the sole element of the *unit set* unit. That is, we consider $\mathit{Outcomes}(\mathcal{S})$ a shorthand for $\mathit{Outcomes}(\mathcal{S}, \mathsf{unit})$, and $\langle \sigma \rangle$ a shorthand for $\langle \sigma, \mathsf{tt} \rangle$.

Definition 2.8. Outcomes with Answers

$$\begin{array}{rrll} \phi & ::= & \langle \sigma, a \rangle & \text{singleton outcome} \\ & | & \bigotimes \Phi & \text{demonic choice} \\ & | & \bigoplus \Phi & \text{angelic choice} \end{array}$$

$$\begin{array}{rcl} \sigma \in \mathcal{S} \wedge a \in \mathcal{A} & \Rightarrow & \langle \sigma, a \rangle \in \mathit{Outcomes}(\mathcal{S}, \mathcal{A}) \\ \Phi \subseteq \mathit{Outcomes}(\mathcal{S}, \mathcal{A}) & \Rightarrow & \bigotimes \Phi \in \mathit{Outcomes}(\mathcal{S}, \mathcal{A}) \\ \Phi \subseteq \mathit{Outcomes}(\mathcal{S}, \mathcal{A}) & \Rightarrow & \bigoplus \Phi \in \mathit{Outcomes}(\mathcal{S}, \mathcal{A}) \end{array}$$

$$\langle \sigma \rangle = \langle \sigma, \mathsf{tt} \rangle \in \mathit{Outcomes}(\mathcal{S}) = \mathit{Outcomes}(\mathcal{S}, \mathsf{unit})$$

2.4.4. *Outcomes: Satisfaction, Coverage.* A useful question to ask is whether an outcome $\phi \in \mathit{Outcomes}(\mathcal{S}, \mathcal{A})$ satisfies a given postcondition Q, where a postcondition can be modelled mathematically as the set of pairs of output states and answers that satisfy it, i.e. $Q \subseteq \mathcal{S} \times \mathcal{A}$. We denote this by $\phi\ \{Q\}$.

We define this recursively as follows:

- A singleton outcome $\langle \sigma, a \rangle$ satisfies postcondition Q if the output state σ and answer a satisfy Q, i.e. $(\sigma, a) \in Q$.
- A demonic choice $\bigotimes \Phi$ satisfies Q if all alternatives satisfy Q
- An angelic choice $\bigoplus \Phi$ satisfies Q if some alternative satisfies Q.

Notice that it follows from this definition that nontermination satisfies all postconditions (even the postcondition that does not accept any output state), and failure satisfies no postcondition (not even the postcondition that accepts all output states).

We also define *coverage* between outcomes: we say outcome ϕ *covers* outcome ϕ', denoted $\phi \Rrightarrow \phi'$, if for any postcondition Q, if ϕ satisfies Q, then ϕ' satisfies Q. Intuitively, this means ϕ is a "worse" outcome than ϕ'; if ϕ' is failure, then ϕ must be failure, but the converse does not hold: it is possible that ϕ is failure but ϕ' is not. Another way to look at this is to say that ϕ is a safe approximation of ϕ' for verification: if we prove that ϕ satisfies some postcondition, then it follows that ϕ' also satisfies it.

We lift outcome coverage pointwise to mutators: a mutator C *covers* a mutator C' if for each input state σ, the outcome of C started in state σ covers the outcome of C' started in state σ.

Definition 2.9. Outcomes: Satisfaction, Coverage

$$\begin{array}{c} \phi \in \mathit{Outcomes}(\mathcal{S}, \mathcal{A}) \quad Q \subseteq \mathcal{S} \times \mathcal{A} \\ \phi\ \{Q\} \quad (\text{“outcome } \phi \text{ satisfies postcondition } Q\text{”}) \end{array}$$

$$\begin{array}{rcl} \langle \sigma, a \rangle\ \{Q\} & \Leftrightarrow & (\sigma, a) \in Q \\ \bigotimes \Phi\ \{Q\} & \Leftrightarrow & \forall \phi \in \Phi.\ \phi\ \{Q\} \\ \bigoplus \Phi\ \{Q\} & \Leftrightarrow & \exists \phi \in \Phi.\ \phi\ \{Q\} \end{array}$$

$$\begin{array}{rcl} \phi \Rrightarrow \phi' & \Leftrightarrow & \forall Q.\ \phi\ \{Q\} \Rightarrow \phi'\ \{Q\} \\ C \Rrightarrow C' & \Leftrightarrow & \forall \sigma.\ C(\sigma) \Rrightarrow C'(\sigma) \end{array}$$

2.4.5. *Outcomes: Sequential Composition.* An important concept is the sequential composition $\phi; C$ of an outcome $\phi \in \mathit{Outcomes}(\mathcal{S})$ and a mutator $C \in \mathcal{S} \to \mathit{Outcomes}(\mathcal{S}')$. The intuition is straightforward: the output states of ϕ are passed as input states to C. The result is again an outcome. It is defined as follows:

- if ϕ is a singleton outcome $\langle \sigma \rangle$, the sequential composition is the outcome of passing σ as input to C
- if ϕ is a demonic or angelic choice, the sequential composition is the distribution of the sequential composition over the alternatives.

We also define sequential composition $C; C'$ of two mutators C and C': it is simply the mutator that, for a given input state σ, passes σ to C and composes the outcome sequentially with C'.

Definition 2.10. Outcomes: Sequential composition

$$-;- \ : \ \mathit{Outcomes}(\mathcal{S}) \to (\mathcal{S} \to \mathit{Outcomes}(\mathcal{S}')) \to \mathit{Outcomes}(\mathcal{S}')$$

$$\begin{array}{rcl} \langle \sigma \rangle; C & = & C(\sigma) \\ (\bigotimes \Phi); C & = & \bigotimes\{\phi \in \Phi.\ (\phi; C)\} \\ (\bigoplus \Phi); C & = & \bigoplus\{\phi \in \Phi.\ (\phi; C)\} \end{array}$$

$$C; C' = \lambda \sigma.\ C(\sigma); C'$$

We have the following important properties of sequential composition:

- Associativity: given three mutators C, C', and C'', first composing C and C' and then composing the resulting mutator with C'' is equivalent to first composing C' and C'' and then composing C with the resulting mutator.
- Monotonicity: if mutator C_1 is worse than mutator C_1', and mutator C_2 is worse than mutator C_2', then $C_1; C_2$ is worse than $C_1'; C_2'$.
- Satisfaction: the sequential composition $\phi; C$ satisfies the postcondition Q if and only if ϕ satisfies the postcondition that accepts the state σ if $C(\sigma)$ satisfies Q.

Lemma 2.11 (Associativity of Sequential Composition of Mutators)**.**

$$(C; C'); C'' = C; (C'; C'')$$

Lemma 2.12 (Monotonicity of Sequential Composition of Mutators)**.** *If* $C_1 \Rrightarrow C_1'$ *and* $C_2 \Rrightarrow C_2'$ *then* $C_1; C_2 \ \Rrightarrow C_1'; C_2'$.

Lemma 2.13 (Satisfaction of Sequential Composition of Mutators)**.**

$$\phi; C\ \{Q\} \Leftrightarrow \phi\ \{\sigma \mid C(\sigma)\ \{Q\}\}$$

2.4.6. *Outcomes: Sequential Composition (with Answers).* We can generalize these concepts to the case of outcomes with answers. If ϕ is an outcome and $C(-)$ is a function from answers to mutators, i.e. a mutator parameterized by an answer, then we write the sequential composition of ϕ and $C(-)$ as $x \leftarrow \phi; C(x)$; that is, the answer of ϕ, bound to the variable x, is passed as an input argument to $C(-)$. The definition and the properties are a straightforward adaptation of the ones given above for outcomes without answers.

Definition 2.14. Outcomes: Sequential composition (with Answers)

$$\begin{array}{rcl} x \leftarrow -; -(x) & : & O(\mathcal{S}, \mathcal{A}) \to (\mathcal{A} \to \mathcal{S} \to O(\mathcal{S}', \mathcal{B})) \to O(\mathcal{S}', \mathcal{B}) \\ x \leftarrow \langle \sigma, a \rangle; C(x) & = & C(a)(\sigma) \\ x \leftarrow (\bigotimes \Phi); C(x) & = & \bigotimes\{\phi \in \Phi.\ (x \leftarrow \phi; C(x))\} \\ x \leftarrow (\bigoplus \Phi); C(x) & = & \bigoplus\{\phi \in \Phi.\ (x \leftarrow \phi; C(x))\} \end{array}$$

$$x \leftarrow C; C'(x) = \lambda\sigma.\ x \leftarrow C(\sigma); C'(x)$$

Lemma 2.15 (Associativity of Sequential Composition of Mutators with Answers)**.**

$$y \leftarrow (x \leftarrow C; C'(x)); C''(y) = x \leftarrow C; (y \leftarrow C'(x); C''(y))$$

Lemma 2.16 (Monotonicity of Sequential Composition of Mutators with Answers)**.** *If* $C_1 \Rrightarrow C_1'$ *and* $\forall a.\ C_2(a) \Rrightarrow C_2'(a)$ *then* $x \leftarrow C_1; C_2(x) \Rrightarrow x \leftarrow C_1'; C_2'(x)$*.*

Lemma 2.17 (Satisfaction of Sequential Composition of Mutators with Answers)**.**

$$x \leftarrow \phi; C(x)\ \{Q\} \Leftrightarrow \phi\ \{(\sigma, a) \mid C(a)(\sigma)\ \{Q\}\}$$

2.4.7. *Outcomes: Notations.* We introduce some additional notations and concepts that will be useful in the definition of the executions.

We lift demonic and angelic choice to mutators: if $\tilde{C}$ is a set of mutators, then $\bigotimes \tilde{C}$ is the demonic choice over these mutators. It is the mutator that, for a given input state σ, demonically chooses between the outcomes obtained by passing σ to the elements of $\tilde{C}$. Angelic choice over mutators is defined analogously.

We use the "variable binding" notation $\bigotimes i \in I.\ \phi_i$ to denote the demonic choice over the outcomes obtained by letting i range over I in ϕ_i. We also use this notation for angelic choice and for choices over mutators.

As an extension of the variable binding notation, we also allow boolean propositions to the left of the dot in demonic and angelic choices. If the proposition is true, this has no effect; otherwise, in the case of angelic choice, this means failure, and in the case of demonic choice, this means nontermination.

We define the primitive mutator $\mathsf{yield}\ a$ as the mutator that does not modify the state and answers a. We define noop as the mutator that does nothing; it merely answers the unit element tt.

We define *side-effect-only* sequential composition $C;, C'$ of two mutators C and C' as the mutator that first executes C, and then executes C', and whose answer is the answer of C. The answer of C' is ignored.

Notation 2.18. Outcomes: Notations

$$\bigotimes \tilde{C} = \lambda\sigma.\ \bigotimes\{C(\sigma) \mid C \in \tilde{C}\}$$
$$\bigoplus \tilde{C} = \lambda\sigma.\ \bigoplus\{C(\sigma) \mid C \in \tilde{C}\}$$

$$\bigotimes i \in I.\ \phi_i = \bigotimes\{\phi_i \mid i \in I\}$$
$$\bigoplus i \in I.\ \phi_i = \bigoplus\{\phi_i \mid i \in I\}$$

$$\bigotimes \mathsf{true}.\ \phi = \phi \qquad \bigotimes \mathsf{false}.\ \phi = \top$$
$$\bigoplus \mathsf{true}.\ \phi = \phi \qquad \bigoplus \mathsf{false}.\ \phi = \bot$$

$$\mathsf{yield}\ a = \lambda\sigma.\ \langle \sigma, a\rangle$$
$$\mathsf{noop} = \mathsf{yield}\ \mathsf{tt}$$

$$C;, C' = x \leftarrow C; C'; \mathsf{yield}\ x$$

2.5. **Some Auxiliary Definitions.** We introduce some further auxiliary notions that will be useful in the definition of concrete execution of commands.

The domain of a heap h is the set of *domain elements* of the form $p(\ell)$ where a value v exists such that $p(\ell, v)$ occurs in h.

The mutator $\mathsf{assume}(b)$, where b is a boolean expression, evaluates b in the given input store; if b evaluates to true, the mutator does nothing; otherwise, it does not terminate. We define evaluation $[\![b]\!]_s$ of a boolean expression b or arithmetic expression e under a store s as follows: $[\![e = e']\!]_s = ([\![e]\!]_s = [\![e']\!]_s)$, $[\![e < e']\!]_s = ([\![e]\!]_s < [\![e']\!]_s)$, $[\![\neg b]\!]_s = \neg[\![b]\!]_s$, $[\![z]\!]_s = z$, $[\![x]\!]_s = s(x)$, $[\![e + e']\!]_s = [\![e]\!]_s + [\![e']\!]_s$, and $[\![e - e']\!]_s = [\![e]\!]_s - [\![e']\!]_s$.

The mutator store simply returns the current store. The mutator $\mathsf{store} := s$ sets the current store to s. The mutator $\mathsf{with}(s, C)$ executes the mutator C under store s and then restores the original store. Its answer is the answer of C. The mutator $\mathsf{eval}(e)$ answers the value of e under the current store. The mutator $x := v$ updates the store, assigning value v to variable x.

Definition 2.19. Some Auxiliary Definitions

$$\begin{aligned}
\mathsf{dom}(h) &= \{p(\ell) \mid \exists v.\ p(\ell, v) \in h\} \\
\mathsf{assume}(b) &= \lambda(s, h).\ \bigotimes [\![b]\!]_s = \mathsf{true}.\ \langle (s, h)\rangle \\
\mathsf{store} &= \lambda(s, h).\ \langle (s, h), s\rangle \\
\mathsf{store} := s' &= \lambda(s, h).\ \langle (s', h)\rangle \\
\mathsf{with}(s', C) &= s \leftarrow \mathsf{store}; \mathsf{store} := s'; \mathsf{C};, \mathsf{store} := s \\
\mathsf{eval}(e) &= \lambda(s, h).\ \langle (s, h), [\![e]\!]_s\rangle \\
x := v &= \lambda(s, h).\ \langle (s[x := v], h)\rangle
\end{aligned}$$

We denote mutator C iterated n times as C^n. C iterated zero times does nothing; C iterated $n+1$ times is the sequential composition of C and C iterated n times. The demonic iteration C^* of C is C iterated a demonically chosen number of times.

Concrete consumption of a multiset h of chunks fails if the heap does not contain these chunks; otherwise, it removes them. Concrete production of a multiset h of chunks blocks if the heap already contains chunks with the same address, i.e. if the addresses of the chunks in h are not all pairwise distinct from the addresses of the chunks that are already in the heap. Otherwise, it adds the chunks to the heap. Concrete consumption and production of a single chunk α are defined in the obvious way.

$$\mathsf{exec}_0(c) = \top$$

$$\mathsf{exec}_{n+1}(x := e) = v \leftarrow \mathsf{eval}(e); x := v$$

$$\mathsf{exec}_{n+1}(c; c') = \mathsf{exec}_n(c); \mathsf{exec}_n(c')$$

$$\begin{aligned}&\mathsf{exec}_{n+1}(\textbf{if } b \textbf{ then } c \textbf{ else } c') = \\ &\quad \mathsf{assume}(b); \mathsf{exec}_n(c) \otimes \mathsf{assume}(\neg b); \mathsf{exec}_n(c')\end{aligned}$$

$$\begin{aligned}&\mathsf{exec}_{n+1}(\textbf{while } b \textbf{ do } c) = \\ &\quad (\mathsf{assume}(b); \mathsf{exec}_n(c))^*; \mathsf{assume}(\neg b)\end{aligned}$$

$$\begin{aligned}&\mathsf{exec}_{n+1}(r(\overline{e})) = \overline{v} \leftarrow \mathsf{eval}(\overline{e}); \mathsf{with}(\mathbf{0}[\overline{x} := \overline{v}], \mathsf{exec}_n(c)) \\ &\quad \text{where } \textbf{routine } r(\overline{x}) = c\end{aligned}$$

$$\begin{aligned}&\mathsf{exec}_{n+1}(x := \textbf{malloc}(n)) = \\ &\quad \bigotimes \ell, v_1, \ldots, v_n \in \mathbb{Z}. \\ &\qquad \mathsf{cproduce\_chunks}(\{\!\!\{\mathsf{mb}(\ell, n), \ell \mapsto v_1, \ldots, \ell + n - 1 \mapsto v_n\}\!\!\}); x := \ell\end{aligned}$$

$$\begin{aligned}&\mathsf{exec}_{n+1}(x := [e]) = \ell \leftarrow \mathsf{eval}(e); \\ &\quad \bigoplus v.\ \mathsf{cconsume\_chunk}(\ell \mapsto v); \mathsf{cproduce\_chunk}(\ell \mapsto v); x := v\end{aligned}$$

$$\begin{aligned}&\mathsf{exec}_{n+1}([e] := e') = \ell \leftarrow \mathsf{eval}(e); v \leftarrow \mathsf{eval}(e'); \\ &\quad \bigoplus v_0.\ \mathsf{cconsume\_chunk}(\ell \mapsto v_0); \mathsf{cproduce\_chunk}(\ell \mapsto v)\end{aligned}$$

$$\begin{aligned}&\mathsf{exec}_{n+1}(\textbf{free}(e)) = \ell \leftarrow \mathsf{eval}(e); \\ &\quad \bigoplus N \in \mathbb{N}, v_1, \ldots, v_N \in \mathbb{Z}. \\ &\qquad \mathsf{cconsume\_chunks}(\{\!\!\{\mathsf{mb}(\ell, N), \ell \mapsto v_1, \ldots, \ell + N - 1 \mapsto v_N\}\!\!\})\end{aligned}$$

$$\mathsf{exec}(c) = \bigotimes n \in \mathbb{N}.\ \mathsf{exec}_n(c)$$

Figure 4: Concrete Execution of Commands

Definition 2.20. Some Auxiliary Definitions

$$\begin{aligned}
C^0 &= \mathsf{noop} \\
C^{n+1} &= C; C^n \\
C^* &= \bigotimes n \in \mathbb{N}.\ C^n \\
\mathsf{cconsume\_chunks}(h') &= \lambda(s, h).\ \bigoplus h' \leq h.\ \langle (s, h - h') \rangle \\
\mathsf{cconsume\_chunk}(\alpha) &= \mathsf{cconsume\_chunks}(\{\!\!\{\alpha\}\!\!\}) \\
\mathsf{cproduce\_chunks}(h') &= \lambda(s, h).\ \bigotimes \mathsf{dom}(h) \cap \mathsf{dom}(h') = \emptyset.\ \langle (s, h \uplus h') \rangle \\
\mathsf{cproduce\_chunk}(\alpha) &= \mathsf{cproduce\_chunks}(\{\!\!\{\alpha\}\!\!\})
\end{aligned}$$

2.6. **Concrete Execution of Commands.**

Definition 2.21. Concrete Execution of Commands See Figure 4.

To define the concrete execution function exec, we first define a helper function exec_n, which is indexed by the maximum depth of the execution. If an execution exceeds the maximum depth, exec_n returns $\top$, i.e. the execution does not terminate.

Therefore, for any command c, $\mathsf{exec}_0(c)$ returns the mutator $\top$ (which is the mutator that for any input state returns the outcome $\top$).

Execution of an assignment $x := e$ evaluates e and binds variable x to its value.

Execution of a sequential composition $c; c'$ is the sequential composition of the execution of c and the execution of c'. (Notice that the two semicolons in this rule have different meanings: the former is part of the syntax of commands defined on Page 3; the latter is the function defined on Page 12 that takes two mutators and returns a mutator.)

Execution of an if-then-else command **if** b **then** c **else** c' demonically chooses between two branches: in the first branch, it is assumed that the condition b evaluates to true, and then command c is executed; in the second branch, it is assumed that b evaluates to false, and then c' is executed. Notice that this is equivalent to evaluating the condition and then, depending on whether it evaluates to true or false, executing c or c', respectively.

Execution of a loop **while** b **do** c first executes the body some demonically chosen number of times, after assuming that the loop condition holds, and then assumes that the condition does not hold.

Execution of a call $r(\overline{e})$ of routine r with argument list $\overline{e}$ first evaluates $\overline{e}$ to obtain values $\overline{v}$ and then executes the body c of r in a store which binds the parameters $\overline{x}$ of r to $\overline{v}$.

Execution of a memory block allocation command $x := \mathbf{malloc}(n)$ demonically picks an address ℓ and values $v_1, \ldots, v_n$ and produces the malloc block chunk and the n points-to chunks that constitute the newly allocated memory block. Finally, the execution binds variable x to the address ℓ.

Execution of a memory read command $x := [e]$ angelically picks a value v and tries to consume a points-to chunk at the address given by e and with value v. If it succeeds, it puts the chunk back and binds x to v.

Execution of a memory write command $[e] := e'$ angelically picks an old value v_0 and tries to consume a points-to chunk at the address given by e and with value v_0. If it succeeds, it puts the chunk back with an updated value.

Execution of a memory block deallocation command $\mathbf{free}(e)$ first evaluates expression e to an address ℓ and then tries to consume a malloc block chunk and a corresponding number of points-to chunks at address ℓ.

Execution of a command demonically chooses a maximum depth and then executes the command up to that depth. Notice that this is equivalent to executing the command without a depth bound.

2.7. **Safety of a Program.**

We say that a program is *safe* if no execution of the program accesses unallocated memory, i.e. no execution fails, when started from the empty state σ_0. (Notice that the failure outcome is the only outcome that does not satisfy postcondition "true".)

The verification problem addressed by Featherweight VeriFast is to check whether a command c is a safe program.

	exec	scexec	symexec
Recursion	Yes	No	No
Looping	Yes	No	No
Branching	Infinite	Infinite	Finite
Is Algorithm	No	No	Yes

$$\begin{tikzcd}\mathsf{exec} \arrow[r, "\substack{\text{Assertions}\\ \text{Predicates}\\ \text{Routine contracts}\\ \text{Loop invariants}}"] & \mathsf{scexec} \arrow[r, "\substack{\text{Symbols}\\ \text{Path Condition}\\ \text{Fresh Symbols}\\ \text{Theorem Prover}}"] & \mathsf{symexec}\end{tikzcd}$$

Figure 5: Solving the Verification Problem

Definition 2.22. Safety of a Program

$$\begin{array}{rcl} \sigma_0 & = & (\mathbf{0}, \mathbf{0}) \\ a \triangleright f & = & f(a) \\ \mathsf{safe\_program}(c) & = & \sigma_0 \triangleright \mathsf{exec}(c)\ \{\text{true}\} \end{array}$$

Definition 2.23 (The Verification Problem)**.**

$$\mathsf{safe\_program}(c)$$

2.8. **Solving the Verification Problem.** See Figure 5.

How to solve the verification problem? Naively computing the full traces of all executions of a program is impossible, since traces may be very large or even infinite (due to recursion and loops), and there may be infinitely many executions (due to the nondeterminism of memory allocation, causing execution to split into infinitely many branches, one for each choice of address). Therefore, concrete execution itself cannot serve as an algorithm for checking program safety.

To obtain an algorithm, we define new kinds of executions that do not exhibit infinitely long traces and/or infinite branching. Specifically, in Section 3 we define *semiconcrete execution* (scexec), where we use routine contracts and loop invariants, expressed as *assertions* that use *predicates* to denote data structures of potentially unbounded size, to limit the length of execution traces. Specifically, semiconcrete execution executes each routine separately, starting from an arbitrary initial state that satisfies the precondition, and checking that each final state satisfies the postcondition. Correspondingly, a routine call is executed using the callee's contract instead of its body. Similarly, a loop body is executed separately, starting from an arbitrary state that satisfies the loop invariant, and checking that each final state again satisfies the loop invariant. Execution of a loop first checks that the loop invariant holds on entry to the loop, and then updates the state to an arbitrary final state that satisfies the loop invariant. Since routine body and loop body executions are no longer inlined into the executions of their callers or loops, all executions have finite length.

However, semiconcrete execution still exhibits infinite branching; therefore, in Section 4 we define the actual verification algorithm of Featherweight VeriFast, which we call *symbolic execution* (symexec). It builds on semiconcrete execution but eliminates infinite branching through the use of *symbols* and a *path condition*, such that a single symbol can be used to represent an infinite number of concrete values. Infinite branching is thus replaced by picking a *fresh symbol*. A *theorem prover* is used to decide equalities between terms and other conditions involving symbols under a given path condition.

Our solution to the verification problem is then to execute the program symbolically. Crucially, the executions are designed such that if symbolic execution of a program succeeds ($\mathsf{sym\text{-}safe\_program}(c)$), then semiconcrete execution succeeds ($\mathsf{sc\text{-}safe\_program}(c)$), and if semiconcrete execution succeeds, then concrete execution succeeds ($\mathsf{safe\_program}(c)$). These properties are called the soundness of symbolic execution and the soundness of semiconcrete execution, respectively. In the next two sections, we define these executions and sketch a proof of their soundness.

Definition 2.24 (Soundness)**.**

$$\mathsf{safe\_program}(c) \Leftarrow \mathsf{sc\text{-}safe\_program}(c) \Leftarrow \mathsf{sym\text{-}safe\_program}(c)$$

3. Semiconcrete Execution

In this section, we define semiconcrete execution, which introduces routine contracts and loop invariants to limit the length of execution traces. Routine contracts and loop invariants are specified using a language of *assertions*, which specify both the *facts* (boolean expressions) and the *resources* (heap chunks) that are required or provided by a routine or loop body. To specify potentially unbounded data structures, *predicates* are used, which are named, parameterized assertions which may be recursive, i.e. mention themselves in their definition.

The structure of this section is as follows. First, we introduce the new concepts involved in semiconcrete execution using a number of example programs and execution traces. Then, we formally define semiconcrete execution. Finally, we sketch an approach for proving that if a program is safe under semiconcrete execution, then it is safe under concrete execution, i.e. semiconcrete execution is a sound approximation for checking the safety of a program under concrete execution.

3.1. **Annotations by Example.**

In this subsection, we introduce the kinds of program annotations required by Featherweight VeriFast by means of some examples.

Example 3.1. Annotations: Simple Example

```
routine swap(cell1, cell2)
  req cell1 ↦ ?v1 ∗ cell2 ↦ ?v2
  ens cell1 ↦ v2 ∗ cell2 ↦ v1
  =
    value1 := [cell1];
    value2 := [cell2];
    [cell1] := value2;
    [cell2] := value1
```

The example above shows a simple routine swap that swaps the values of two memory cells whose addresses are given by arguments cell1 and cell2. The body first reads the cells' original values into variables and then writes each cell's original value into the other cell. The routine has been annotated with a *routine contract* consisting of a *precondition* (also known as a *requires clause*, denoted using keyword **req**) and a *postcondition* (also known as an *ensures clause*, denoted using keyword **ens**). The precondition describes the set of initial states accepted by the routine; the postcondition describes the set of final states generated by the routine when started from an initial state that satisfies the precondition.

The precondition of routine swap states that the routine requires two distinct memory cells to be present in the heap, one at address cell1 and the other at address cell2. Furthermore, it introduces two *ghost variables* v1 and v2: it binds v1 to the original value of the cell at address cell1 and v2 to the original value of the cell at address cell2. In general, when a variable appears in an assertion immediately preceded by a question mark, this is called a *variable pattern*. A variable pattern $?x$ introduces the variable x and binds it to the value found in the heap corresponding to the position where the variable pattern appears.

In the example, the purpose of introducing the variables v1 and v2 in the precondition is so that they can be used in the postcondition to specify the relationship between the initial state and the final state of the routine. Specifically, the postcondition specifies that in the final state, the same memory cells are still present in the heap, and their value has changed such that the new value of the cell at address cell1 equals the original value of the cell at address cell2 and vice versa.

Notice that the assertions that serve as the precondition and the postcondition of routine swap specify only resources (heap chunks). In general, assertions may also specify facts (boolean expressions). Correspondingly, there are two kinds of elementary assertions: boolean expressions and *predicate assertions*. Elementary assertions can be composed using the *separating conjunction* $*$. Its meaning is that the facts on the left and the facts on the right are both true, and that furthermore the resources on the left and the resources on the right are both present *separately*, i.e. the heap can be split into two parts such that the resources specified by the left-hand side of the assertion are in one part and the resources specified by the right-hand side of the assertion are in the other part. Notice how, in this respect, separating conjunction differs from ordinary logical conjunction (AND): we have that a is equivalent to $a \wedge a$, but we do not have that a is equivalent to $a * a$. In particular, $a * a$ specifies that the heap contains *two occurrences* of each resource specified by a, which generally is not possible, and therefore $a * a$ is generally unsatisfiable. This also means that the precondition of routine swap implies that cell1 and cell2 denote distinct addresses.

Now, consider again the example routine range that we introduced earlier. Recall that this routine builds a linked list holding the values between argument i, inclusive, and argument n, exclusive, and writes the address of the first node into the previously allocated memory cell whose address is given by argument result.

We show a contract for this routine in Figure 6. The precondition specifies that a memory cell must exist at the address given by argument result. The postcondition specifies that this memory cell still exists, and that it now points to a linked list. The latter guarantee is specified using the *predicate* list, defined above. The definition of the predicate declares one parameter, l, and a body, which is an assertion. The body performs a case analysis on whether l equals 0. If so, it specifies only the trivial fact that 0 equals 0, i.e. it does not specify anything. Otherwise, it specifies that the heap contains a malloc block chunk of

```
predicate list(l) =
  if l = 0 then 0 = 0 else
    mb(l, 2) ∗ l ↦ ?v ∗ l + 1 ↦ ?n ∗ list(n)

routine range(i, n, result)
  req result ↦ ?dummy
  ens result ↦ ?list ∗ list(list)
  =
    if i = n then head := 0 else (
      head := malloc(2);
      [head] := i;
      range(i + 1, n, head + 1)
    );
    close list(head); [result] := head
```

Figure 6: Annotations: Predicates

size 2 at address l, as well as two memory cells, at addresses l and $l + 1$, as well as another linked list pointed to by the memory cell at address $l + 1$.

Technically, what happens is that the predicate definition introduces a new kind of chunk, or more specifically, a new chunk name, and allows this chunk name to be used in predicate assertions. As a result, in semiconcrete execution, there are two kinds of predicates: the built-in predicates **mb** and $\mapsto$, and the user-defined predicates. Correspondingly, the heap contains two kinds of chunks: those whose name is a built-in predicate, and those whose name is a user-defined predicate. The purpose of chunks corresponding to user-defined predicates is to "bundle up" zero or more malloc block chunks and points-to chunks, along with some facts. Such "bundling up" is necessary for writing contracts for routines that manipulate data structures of unbounded size. For example, it is impossible to write a postcondition for routine range without using user-defined predicates: a postcondition that contains m points-to assertions cannot describe a linked list of length greater than m, so such a postcondition does not hold for a call of range where $n - i > m$.

The built-in chunks are created by the **malloc** statement. How are the user-defined chunks created? To enable the creation of user-defined chunks, semiconcrete execution introduces a new form of commands into the programming language, called **close** commands. The command **close** $p(\overline{e})$ requires that p is a user-defined predicate; it removes from the heap the chunks described by the body of the predicate, and checks the facts required by the body of the predicate, and then adds a user-defined chunk whose name is p and whose arguments are the values of $\overline{e}$. That is, the command bundles up the resources and facts described by the body of predicate p into a chunk named p.

In the example, the body of routine range, after allocating the first node and performing the recursive call to build the rest of the linked list, performs a **close** operation to bundle the three chunks of the first node and the list chunk that represents the rest of the linked list together into a single list chunk.

3.2. **Syntax of Annotations.** In summary, the programming language syntax extensions introduced by semiconcrete execution are as follows.

```
routine range(i, n, r)
  req r ↦ ?dummy ens r ↦ ?list ∗ list(list)
s:0[i:5, n:8, r:41], h:0
produce(r ↦ ?dummy)
s:0[i:5, n:8, r:41], h:{41↦77}
if i = n then l := 0 else (
l := malloc(2);
s:0[i:5, n:8, r:41, l:50], h:{41↦77,mb(50, 2),50↦88,51↦99}
[l] := i; range(i + 1, n, l + 1)
consume(l+1↦?dummy); produce(l+1↦?list ∗ list(list))
s:0[i:5, n:8, r:41, l:50], h:{41↦77,mb(50, 2),50↦5,51↦60,list(60)}
); close list(l); [r] := l
s:0[i:5, n:8, r:41, l:50], h:{41↦50,list(50)}
consume(r ↦ ?list ∗ list(list))
s:0[i:5, n:8, r:41, l:50], h:0
```

Figure 7: Semiconcrete Execution: Example Trace

A program may now declare a number of *routine specifications rspec* of the form **routine** $r(\overline{x})$ **req** a **ens** a', which associate with the routine name r and parameter list $\overline{x}$ the precondition a and postcondition a', which are assertions. Furthermore, the syntax of loops is extended to include a loop invariant clause **inv** a, where a is an assertion. Furthermore, a program may declare a number of predicate definitions, which associate a predicate name and a list of parameters with a body, which is an assertion. An assertion a is a boolean expression b, a predicate assertion $p(\overline{e}, \overline{?x})$ (where p is either a built-in predicate or a user-defined predicate), a separating conjunction $a * a'$, or a conditional assertion **if** b **then** a **else** a'. Two new commands are introduced: the **open** command and the **close** command. The **open** command performs the inverse operation of the **close** command: it unbundles a user-defined chunk, i.e. it removes the user-defined chunk from the heap and adds the chunks described by the body of the predicate.

Definition 3.2. Annotations

$$
\begin{array}{rcl}
 & & q \in \mathit{UserDefinedPredicates} \\
p & ::= & \mapsto \ | \ \mathsf{mb} \ | \ q \\
a & ::= & b \ | \ p(\overline{e}, \overline{?x}) \ | \ a * a \ | \ \textbf{if } b \textbf{ then } a \textbf{ else } a \\
\mathit{preddef} & ::= & \textbf{predicate } q(\overline{x}) = a \\
c & ::= & \cdots \ | \ \textbf{while } b \textbf{ inv } a \textbf{ do } c \ | \ \textbf{open } q(\overline{e}) \ | \ \textbf{close } q(\overline{e}) \\
\mathit{rspec} & ::= & \textbf{routine } r(\overline{x}) \textbf{ req } a \textbf{ ens } a
\end{array}
$$

$$e \mapsto ?x \text{ is alternative syntax for } \mapsto(e, ?x)$$

3.3. **Semiconcrete Execution: Example Trace.** Recall the example concrete execution trace of routine range in Figure 3. Recall that the notable features of this trace are that the trace is long, since it contains three nested executions of routine range; that the heap is large, since it includes the entire heap that existed on entry to the routine, as well as all of the chunks produced by all of the nested calls; and that there is infinite branching.

We show an example semiconcrete execution trace for routine range in Figure 7. Recall that semiconcrete execution executes each routine separately. Therefore, the above trace is not an excerpt from a larger program trace; rather, it is a complete trace of the execution of routine range.

Execution starts in a state where the store binds each parameter to an arbitrary argument value and the heap is empty. It then *produces* the precondition: it adds the resources and assumes the facts specified by the precondition. When producing a predicate assertion $p(\overline{e}, \overline{?x})$, the values of the arguments corresponding to the variable patterns $\overline{?x}$ are arbitrary. In the example, a points-to chunk at the address given by parameter result is added to the heap.

The execution of the **malloc** command and the memory write command are the same as in the concrete execution.

The routine call is executed not by inlining a nested execution of the body of the routine, but by using the contract: the precondition is *consumed*, and then the postcondition is *produced*. Consuming an assertion means removing the heap chunks and checking the facts specified by the assertion. If a fact specified by the assertion is false, execution fails. The net effect is that the points-to chunk at address 51 gets some arbitrary value (60 in this trace) and a list chunk is added whose argument is 60.

The **close** command collapses the four chunks representing the linked list into a single chunk list(50).

Finally, after execution of the routine body is complete, the postcondition is consumed. It removes all of the heap chunks and leaves the heap empty.

Generally, in semiconcrete execution, if the heap is left nonempty after a routine execution, this indicates a memory leak, since the memory described by the remaining chunks can no longer be accessed by any subsequent operation in the program execution. Indeed, of the heap chunks that exist at the end of a routine body execution, only the ones described by the postcondition become available to the caller; the others can no longer be retrieved in any way. Therefore, as the final step of a routine execution, semiconcrete execution checks that the heap is empty; if not, routine execution fails.

3.4. **Semiconcrete Execution: Types.**

The set *SCStates* of semiconcrete states is defined above; the only difference with the concrete states is that the predicates now include the user-defined predicates, and consequently the chunks now include the user-defined chunks.

To formally define semiconcrete command execution, we will define a function scexec from commands to mutators, similar to function exec for concrete execution. Additionally, we define functions consume and produce that formalize what it means to consume and produce an assertion, respectively.

Definition 3.3. Semiconcrete Execution: Types

$$\begin{aligned}
\mathit{SCStores} &= \mathit{Vars} \to \mathbb{Z} \\
\mathit{SCPredicates} &= \{\mapsto, \mathsf{mb}\} \cup \mathit{UserDefinedPredicates} \\
\mathit{SCChunks} &= \{p(\overline{v}) \mid p \in \mathit{SCPredicates}, \overline{v} \in \mathbb{Z}\} \\
\mathit{SCHeaps} &= \mathit{SCChunks} \to \mathbb{N} \\
\mathit{SCStates} &= \mathit{SCStores} \times \mathit{SCHeaps} \\
\mathit{SCMutators} &= \mathit{SCStates} \to \mathit{Outcomes}(\mathit{SCStates})
\end{aligned}$$

$$\begin{array}{ll}\mathsf{scexec} & \in \mathit{Commands} \to \mathit{SCMutators} \\ \mathsf{consume} & \in \mathit{Assertions} \to \mathit{SCMutators} \\ \mathsf{produce} & \in \mathit{Assertions} \to \mathit{SCMutators}\end{array}$$

3.5. **Some Auxiliary Definitions.** The definition of semiconcrete execution uses the following auxiliary mutators, in addition to the ones used by the definition of concrete execution. Semiconcrete consumption $\mathsf{consume\_chunks}(h)$ of a multiset of chunks h fails if the heap does not contain these chunks, and otherwise removes them from the heap. It is identical to concrete consumption of chunks. Semiconcrete production $\mathsf{produce\_chunks}(h)$ of a multiset of chunks h adds the chunks to the heap. It differs from concrete production in that it does not check that the added chunks do not clash with existing chunks in the heap. Semiconcrete consumption and production of a single chunk α are defined in the obvious way. The mutator $\mathsf{assert}(b)$ asserting a boolean expression b fails if b, evaluated in the current store, is false, and otherwise does nothing.

Definition 3.4. Some Auxiliary Definitions

$$\begin{array}{rl}\mathsf{consume\_chunks}(h') &= \lambda(s,h).\ \bigoplus h' \le h.\ \langle (s, h - h')\rangle \\ \mathsf{consume\_chunk}(\alpha) &= \mathsf{consume\_chunks}(\{\!\!\{\alpha\}\!\!\}) \\ \mathsf{produce\_chunks}(h') &= \lambda(s,h).\ \langle (s, h \uplus h')\rangle \\ \mathsf{produce\_chunk}(\alpha) &= \mathsf{produce\_chunks}(\{\!\!\{\alpha\}\!\!\}) \\ \mathsf{assert}(b) &= \lambda(s,h).\ \bigoplus [\![b]\!]_s = \mathsf{true}.\ \langle (s,h)\rangle\end{array}$$

3.6. **Producing Assertions.** Production of an assertion is defined as follows.

Production of a boolean expression means assuming it. Recall from the definition of assume on Page 14 that assuming a boolean expression is equivalent to a no-op if it evaluates to true, and equivalent to nontermination if it evaluates to false. The effect is that all final states generated by production satisfy the expression.

Production of a predicate assertion means demonically choosing a value for each variable pattern, binding the pattern variable to it, and adding the specified chunk to the heap.

Producing a separating conjunction means first producing the left-hand side and then producing the right-hand side. Notice that the variable bindings introduced by the left-hand side are active when producing the right-hand side. Notice also that this definition correctly captures the *separating* aspect of the separating conjunction: if a chunk is specified by both the left-hand side and the right-hand side, two occurrences of it end up in the heap.

Producing a conditional assertion is defined analogously to executing a conditional statement.

Definition 3.5. Producing Assertions

$$\mathsf{produce}(b) = \mathsf{assume}(b)$$

$$\mathsf{produce}(p(\overline{e}, \overline{?x})) = \overline{v} \leftarrow \mathsf{eval}(\overline{e}); \bigotimes \overline{v}'.\ \mathsf{produce\_chunk}(p(\overline{v}, \overline{v}')); \overline{x} := \overline{v}'$$

$$\mathsf{produce}(a * a') = \mathsf{produce}(a); \mathsf{produce}(a')$$

$$\mathsf{produce}(\textbf{if } b \textbf{ then } a \textbf{ else } a') = \mathsf{assume}(b); \mathsf{produce}(a) \otimes \mathsf{assume}(\neg b); \mathsf{produce}(a')$$

3.7. **Consuming Assertions.** Consumption of an assertion is defined as follows.

Consuming a boolean expression is equivalent to a no-op if the expression evaluates to true under the current store; otherwise, it is equivalent to failure.

Consuming a predicate assertion fails unless there exists a value for each variable pattern such that the specified chunk can be consumed. If so, each pattern variable is bound to the corresponding value.

Consuming a separating conjunction first consumes the left-hand side and then consumes the right-hand side. Notice that this correctly reflects the *separating* aspect of the separating conjunction: if the left-hand side and the right-hand side specify the same chunk, consumption fails unless the heap contains two occurrences of the chunk, which is generally impossible.

Consuming a conditional assertion is defined analogously to executing a conditional statement.

Definition 3.6. Consuming Assertions

$$\mathsf{consume}(b) = \mathsf{assert}(b)$$

$$\begin{aligned}&\mathsf{consume}(p(\overline{e}, \overline{?x})) = \\ &\quad \overline{v} \leftarrow \mathsf{eval}(\overline{e}); \bigoplus \overline{v}'.\ \mathsf{consume\_chunk}(p(\overline{v}, \overline{v}')); \overline{x} := \overline{v}'\end{aligned}$$

$$\mathsf{consume}(a * a') = \mathsf{consume}(a); \mathsf{consume}(a')$$

$$\begin{aligned}&\mathsf{consume}(\textbf{if } b \textbf{ then } a \textbf{ else } a') = \\ &\quad \mathsf{assume}(b); \mathsf{consume}(a) \otimes \mathsf{assume}(\neg b); \mathsf{consume}(a')\end{aligned}$$

3.8. **Semiconcrete Execution of Commands.** Recall that the concrete execution function exec is defined in terms of the helper function exec_n. The latter function is defined by recursion on n.

In contrast, the semiconcrete execution function scexec is defined directly, by recursion on the structure of the command. Doing so for the concrete execution function would not have been possible, since the execution of a routine call involves the execution of the callee's body, which obviously is not part of the structure of the call command itself. However, since in semiconcrete execution routine call involves only production and consumption of assertions, this simple approach is possible here.

Definition 3.7. Semiconcrete Execution of Commands See Figures 8 and 9.

Execution of assignments, sequential compositions, and conditional assertions is the same as in concrete execution.

Execution of a routine call $r(\overline{e})$ looks up routine r's precondition a and postcondition a', to be interpreted under a parameter list $\overline{x}$, and sets up a store that binds the parameters to the values of the arguments. In this store, it first consumes the precondition and then produces the postcondition. Notice that the variable bindings generated during consumption of the precondition are active during production of the postcondition, since the output store of the consumption operation serves as the input store of the production operation.

Execution of a while loop is relatively complex. It proceeds as follows:

- The loop invariant is consumed.
- An arbitrary new value is assigned to each variable modified by the loop body.
- Execution chooses demonically between two branches:

$\mathsf{scexec}(x := e) = v \leftarrow \mathsf{eval}(e); x := v$

$\mathsf{scexec}(c; c') = \mathsf{scexec}(c); \mathsf{scexec}(c')$

$\mathsf{scexec}(\textbf{if } b \textbf{ then } a \textbf{ else } a') =$
$\quad \mathsf{assume}(b); \mathsf{scexec}(c) \otimes \mathsf{assume}(\neg b); \mathsf{scexec}(c')$

$\mathsf{scexec}(\textbf{while } e \textbf{ inv } a \textbf{ do } c) = \text{See Figure 9}$

$\mathsf{scexec}(r(\overline{e})) = \overline{v} \leftarrow \mathsf{eval}(\overline{e}); \mathsf{with}(\mathbf{0}[\overline{x} := \overline{v}], \mathsf{consume}(a); \mathsf{produce}(a'))$
$\quad \text{where } \textbf{routine } r(\overline{x}) \textbf{ req } a \textbf{ ens } a'$

$\mathsf{scexec}(x := \textbf{malloc}(n)) =$
$\quad \bigotimes \ell, v_1, \ldots, v_n \in \mathbb{Z}.$
$\qquad \mathsf{produce\_chunks}(\{\!\!\{\mathsf{mb}(\ell, n), \ell \mapsto v_1, \ldots, \ell + n - 1 \mapsto v_n\}\!\!\}); x := \ell$

$\mathsf{scexec}(x := [e]) = \ell \leftarrow \mathsf{eval}(e);$
$\quad \bigoplus v.\ \mathsf{consume\_chunk}(\ell \mapsto v); \mathsf{produce\_chunk}(\ell \mapsto v); x := v$

$\mathsf{scexec}([e] := e') = \ell \leftarrow \mathsf{eval}(e); v \leftarrow \mathsf{eval}(e');$
$\quad \bigoplus v_0.\ \mathsf{consume\_chunk}(\ell \mapsto v_0); \mathsf{produce\_chunk}(\ell \mapsto v)$

$\mathsf{scexec}(\textbf{free}(e)) = \ell \leftarrow \mathsf{eval}(e);$
$\quad \bigoplus N \in \mathbb{N}, v_1, \ldots, v_N \in \mathbb{Z}.$
$\qquad \mathsf{consume\_chunks}(\{\!\!\{\mathsf{mb}(\ell, N), \ell \mapsto v_1, \ldots, \ell + N - 1 \mapsto v_N\}\!\!\})$

$\mathsf{scexec}(\textbf{open } p(\overline{e})) = \overline{v} \leftarrow \mathsf{eval}(\overline{e});$
$\quad \mathsf{consume\_chunk}(p(\overline{v})); \mathsf{with}(\mathbf{0}[\overline{x} := \overline{v}], \mathsf{produce}(a))$
$\quad \text{where } \textbf{predicate } p(\overline{x}) = a$

$\mathsf{scexec}(\textbf{close } p(\overline{e})) = \overline{v} \leftarrow \mathsf{eval}(\overline{e});$
$\quad \mathsf{with}(\mathbf{0}[\overline{x} := \overline{v}], \mathsf{consume}(a)); \mathsf{produce\_chunk}(p(\overline{v}))$
$\quad \text{where } \textbf{predicate } p(\overline{x}) = a$

Figure 8: Semiconcrete Execution of Commands

– In the first branch, execution proceeds as follows:
  - ∗ The heap is emptied, so that heap chunks not described by the loop invariant are not available to the loop body.
  - ∗ The loop invariant is produced, but the resulting variable bindings are discarded.
  - ∗ It is assumed that the loop condition holds.
  - ∗ The loop body is executed.
  - ∗ The loop invariant is consumed.
  - ∗ A leak check is performed, i.e. execution fails if the heap is not empty; otherwise, execution blocks.

$$
\begin{array}{lcl}
\mathsf{targets}(x := e) & = & \{x\} \\
\mathsf{targets}(c_1; c_2) & = & \mathsf{targets}(c_1) \cup \mathsf{targets}(c_2) \\
\mathsf{targets}(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) & = & \mathsf{targets}(c_1) \cup \mathsf{targets}(c_2) \\
\mathsf{targets}(r(\overline{e})) & = & \emptyset \\
\mathsf{targets}(\mathbf{while}\ b\ \mathbf{inv}\ a\ \mathbf{do}\ c_0) & = & \mathsf{targets}(c_0) \\
\mathsf{targets}(x := \mathbf{malloc}(n)) & = & \{x\} \\
\mathsf{targets}(x := [e]) & = & \{x\} \\
\mathsf{targets}([e] := e') & = & \emptyset \\
\mathsf{targets}(\mathbf{free}(e)) & = & \emptyset
\end{array}
$$

$$\mathsf{havoc}(\overline{x}) = \lambda(s,h).\ \bigotimes \overline{v} \in \mathbb{Z}.\ \langle (s[\overline{x} := \overline{v}], h) \rangle$$
$$\mathsf{leakcheck} = \lambda(s,h).\ \bigoplus h = \mathbf{0}.\ \top$$

```
$\mathsf{scexec}(\mathbf{while}\ b\ \mathbf{inv}\ a\ \mathbf{do}\ c) =$
  $s \leftarrow \mathsf{store}; \mathsf{with}(s, \mathsf{consume}(a));$
  $\mathsf{havoc}(\mathsf{targets}(c));$
  (
    $\mathsf{heap} := \mathbf{0};$
    $s \leftarrow \mathsf{store}; \mathsf{with}(s, \mathsf{produce}(a));$
    $\mathsf{assume}(b); \mathsf{scexec}(c);$
    $s \leftarrow \mathsf{store}; \mathsf{with}(s, \mathsf{consume}(a));$
    $\mathsf{leakcheck}$
  $\otimes$
    $s \leftarrow \mathsf{store}; \mathsf{with}(s, \mathsf{produce}(a));$
    $\mathsf{assume}(\neg b)$
  )
```

Figure 9: Semiconcrete Execution of Loops

– In the second branch, the loop invariant is produced (but the resulting variable bindings are discarded), and it is assumed that the loop condition does not hold.

The definition uses the auxiliary functions targets, havoc, and leakcheck.

Function targets maps a command to the set of variables modified by the command.

Function $\mathsf{havoc}(\overline{x})$ demonically chooses a value for each variable in $\overline{x}$ and assigns it to the corresponding variable.

Function leakcheck fails if the heap is nonempty, and otherwise blocks, i.e. does not terminate.

Semiconcrete execution of memory block allocation, memory read, memory write, and memory block deallocation are the same as in concrete execution.

Execution of an **open** command **open** $p(\overline{e})$ first consumes the chunk whose name is p and whose arguments are the values of $\overline{e}$ and then produces the body of predicate p, the latter in a store that binds the predicate parameters $\overline{x}$ to the values of the argument expressions $\overline{e}$.

Conversely, execution of a **close** command **close** $p(\overline{e})$ first consumes the body of predicate p in a store that binds the predicate parameters $\overline{x}$ to the values of the argument expressions $\overline{e}$. Then it produces the chunk whose name is p and whose arguments are the values of $\overline{e}$.

3.9. **Validity of Routines.** In concrete execution, safety of a program simply means that execution of the main command starting from an empty state does not fail. In semiconcrete execution, safety of a program means that two things are true: 1) execution of the main command starting from the empty state does not fail; and 2) all routines are *valid*.

Validity of a routine means that its body satisfies its contract. More specifically, it means that the *routine validity mutator* does not fail, when started from an empty state. The routine validity mutator for a given routine r proceeds as follows: 1) it sets up a store that binds each of the routine's parameters to a demonically chosen value; 2) it produces the routine precondition; 3) it semiconcretely executes the routine body; 4) it consumes the routine postcondition; 5) it checks for leaks.

Definition 3.8. Validity of Routines

$$\begin{array}{l}
\mathsf{valid}(r) = \\
\quad (\mathbf{0}, \mathbf{0}) \triangleright \\
\quad \bigotimes \overline{v}. \\
\quad \mathsf{with}(\mathbf{0}[\overline{x} := \overline{v}], \\
\quad\quad s' \leftarrow \mathsf{with}(\mathbf{0}[\overline{x} := \overline{v}], \mathsf{produce}(a); \mathsf{store}); \\
\quad\quad \mathsf{scexec}(c); \\
\quad\quad \mathsf{with}(s', \mathsf{consume}(a')) \\
\quad); \\
\quad \mathsf{leakcheck} \\
\quad \{\mathrm{true}\} \\
\quad \text{where } \textbf{routine } r(\overline{x}) \textbf{ req } a \textbf{ ens } a' = c
\end{array}$$

Notice that the postcondition is consumed starting from the store saved after producing the precondition. This ensures that the variable bindings generated by producing the precondition are visible when consuming the postcondition.

3.10. **Semiconcrete Execution: Program Safety.** As stated before, safety of a program in semiconcrete execution means that execution of the main command succeeds when started from the empty state, *and* that all routines are valid.

Definition 3.9. Semiconcrete Execution: Program Safety

$$\mathsf{sc\text{-}safe\_program}(c) \quad = \quad (\forall r.\ \mathsf{valid}(r)) \land \sigma_0 \triangleright \mathsf{scexec}(c)\ \{\mathrm{true}\}$$

where r ranges over the declared routines of the program.

3.11. **Soundness.** Now that we have defined safety of a program in semiconcrete execution, we discuss its relationship with safety of the program in concrete execution. The intended relationship is that if a program is safe in semiconcrete execution (i.e. all routines are valid and the main command does not fail when executed semiconcretely starting from the empty state), then it is safe in concrete execution (i.e. the main command does not fail when executed concretely starting from the empty state). We call this property the *soundness* of semiconcrete execution. In the remainder of this section, we sketch a proof of this property.

3.11.1. *Properties of Assertion Consumption and Production.* First, we discuss some properties of assertion consumption and production. To gain more insight into consumption and production, we here offer an alternative definition of them, in terms of *consumption and production arrows* $\xrightarrow{a}_{\mathsf{c}}, \xrightarrow{a}_{\mathsf{p}} \subseteq \mathit{SCStates} \times \mathit{SCStates}$, defined inductively using the inference rules shown below. $\sigma \xrightarrow{a}_{\mathsf{c}} \sigma'$ means that consumption of assertion a starting from state σ succeeds and results in state σ'. Similarly, $\sigma \xrightarrow{a}_{\mathsf{p}} \sigma'$ means that production of assertion a starting from state σ results in state σ'.

Definition 3.10. The Consumption Arrow

$$\frac{[\![b]\!]_s = \mathsf{true}}{(s,h) \xrightarrow{b}_{\mathsf{c}} (s,h)} \qquad \frac{h = \{\!\!\{p([\![\overline{e}]\!]_s, \overline{v})\}\!\!\} \uplus h'}{(s,h) \xrightarrow{p(\overline{e},?\overline{x})}_{\mathsf{c}} (s[\overline{x} := \overline{v}], h')} \qquad \frac{(s,h) \xrightarrow{a}_{\mathsf{c}} (s',h') \qquad (s',h') \xrightarrow{a'}_{\mathsf{c}} (s'',h'')}{(s,h) \xrightarrow{a * a'}_{\mathsf{c}} (s'',h'')}$$

$$\frac{[\![b]\!]_s = \mathsf{true} \qquad (s,h) \xrightarrow{a}_{\mathsf{c}} (s',h')}{(s,h) \xrightarrow{\textbf{if } b \textbf{ then } a \textbf{ else } a'}_{\mathsf{c}} (s',h')} \qquad \frac{[\![b]\!]_s = \mathsf{false} \qquad (s,h) \xrightarrow{a'}_{\mathsf{c}} (s',h')}{(s,h) \xrightarrow{\textbf{if } b \textbf{ then } a \textbf{ else } a'}_{\mathsf{c}} (s',h')}$$

Definition 3.11. The Production Arrow

$$\frac{[\![b]\!]_s = \mathsf{true}}{(s,h) \xrightarrow{b}_{\mathsf{p}} (s,h)} \qquad \frac{h' = \{\!\!\{p([\![\overline{e}]\!]_s, \overline{v})\}\!\!\} \uplus h}{(s,h) \xrightarrow{p(\overline{e},?\overline{x})}_{\mathsf{p}} (s[\overline{x}:=\overline{v}], h')} \qquad \frac{(s,h) \xrightarrow{a}_{\mathsf{p}} (s',h') \qquad (s',h') \xrightarrow{a'}_{\mathsf{p}} (s'',h'')}{(s,h) \xrightarrow{a * a'}_{\mathsf{p}} (s'',h'')}$$

$$\frac{[\![b]\!]_s = \mathsf{true} \qquad (s,h) \xrightarrow{a}_{\mathsf{p}} (s',h')}{(s,h) \xrightarrow{\textbf{if } b \textbf{ then } a \textbf{ else } a'}_{\mathsf{p}} (s',h')} \qquad \frac{[\![b]\!]_s = \mathsf{false} \qquad (s,h) \xrightarrow{a'}_{\mathsf{p}} (s',h')}{(s,h) \xrightarrow{\textbf{if } b \textbf{ then } a \textbf{ else } a'}_{\mathsf{p}} (s',h')}$$

Notice that the only difference between the two definitions is the different positions of h and h' in the rule for predicate assertions. Consumption of predicate assertions removes matching chunks, whereas production adds matching chunks.

Notice that in both cases, there are generally multiple output states for any given input state: in both cases, there is a distinct output state for each distinct binding of values to pattern variables in predicate assertions. However, this is much more common in the case of production than in the case of consumption, since in the case of consumption multiple bindings are possible only if the heap contains multiple chunks that match the predicate assertion.

Given the consumption and production arrows, we can give an alternative definition of the consumption and production mutators, as shown below.

Lemma 3.12 (Consumption and Production and the Arrows)**.**

$$\begin{aligned} \mathsf{consume}(a) &= \lambda\sigma.\ \bigoplus \sigma', \sigma \xrightarrow{a}_{\mathsf{c}} \sigma'.\ \langle\sigma'\rangle \\ \mathsf{produce}(a) &= \lambda\sigma.\ \bigotimes \sigma', \sigma \xrightarrow{a}_{\mathsf{p}} \sigma'.\ \langle\sigma'\rangle \end{aligned}$$

Notice that the consumption mutator chooses angelically among the output states, and fails if there are none; production chooses demonically among the output states, and blocks if there are none.

We can easily prove some important properties of the consumption and production arrows. Firstly, consumption is local: if consumption succeeds, then it also succeeds if more chunks are available, and those additional chunks remain untouched.

Lemma 3.13 (Consumption Locality)**.**

$$(s, h) \xrightarrow{a}_{\mathsf{c}} (s', h') \Rightarrow (s, h \uplus h'') \xrightarrow{a}_{\mathsf{c}} (s', h' \uplus h'')$$

Secondly, consumption is monotonic: if it succeeds, then the resulting heap is a submultiset of the original heap, and consumption also succeeds if only the consumed chunks are available, and then it yields the empty heap.

Lemma 3.14 (Consumption Monotonicity)**.**

$$(s, h) \xrightarrow{a}_{\mathsf{c}} (s', h') \Rightarrow \exists h''.\ h = h' \uplus h'' \wedge (s, h'') \xrightarrow{a}_{\mathsf{c}} (s', \mathbf{0})$$

Thirdly, production is the converse of consumption: production adds back the chunks removed by consumption.

Lemma 3.15 (Production after Consumption (Arrows))**.**

$$(s, h) \xrightarrow{a}_{\mathsf{c}} (s', \mathbf{0}) \Rightarrow (s, h'') \xrightarrow{a}_{\mathsf{p}} (s', h'' \uplus h)$$

All of these properties are proved easily by induction on the assertion.

From these properties of the consumption and production arrows, we can easily derive corresponding properties of the consumption and production mutators:

Lemma 3.16 (Consumption and Production (with Post-stores))**.**

$$\begin{array}{l} s_1 \leftarrow \mathsf{with}(s, \mathsf{consume}(a); \mathsf{store}); \\ s_2 \leftarrow \mathsf{with}(s, \mathsf{produce}(a); \mathsf{store}); \\ C(s_1, s_2) \\ \Rrightarrow \\ \bigoplus s'.\ C(s', s') \end{array}$$

Lemma 3.17 (Consumption and Production)**.**

$$\mathsf{with}(s, \mathsf{consume}(a)); \mathsf{with}(s, \mathsf{produce}(a)) \Rrightarrow \mathsf{noop}$$

The first lemma states that consuming an assertion and then producing the same assertion starting from the same store, and then performing some mutator $C(-,-)$ parameterized by the output stores of the consumption and production, safely approximates doing nothing to the heap and angelically picking a store and performing C using this store for both parameters. The second lemma is a simplified version that ignores the output stores: consuming an assertion and then producing the same assertion starting from the same store safely approximates doing nothing.

3.11.2. *Locality and Modifies.* Two important but simple properties of semiconcrete execution are that it is local and that it modifies only the command's targets. Locality means that execution under some initial heap and then adding more chunks safely approximates first adding those chunks and then executing.

Definition 3.18. Locality

$$\mathsf{local}\ C \quad \Leftrightarrow \quad \forall h.\ C;, \mathsf{produce\_chunks}(h) \Rrightarrow \mathsf{produce\_chunks}(h); C$$

Lemma 3.19 (Locality of Semiconcrete Execution)**.**

$$\mathsf{local}\ \mathsf{scexec}(c)$$

Definition 3.20 (Modifies)**.**

$$s \overset{\overline{x}}{\sim} s' \Leftrightarrow s[\overline{x} := 0] = s'[\overline{x} := 0]$$

$$\mathsf{modified}_{\overline{x}}(s') = \lambda(s, h).\ \bigoplus s \overset{\overline{x}}{\sim} s'.\ \mathsf{noop}$$

$$\mathsf{modifies}_{\overline{x}}\ C \quad \Leftrightarrow \quad \forall s.\ \mathsf{modified}_{\overline{x}}(s); C \Rrightarrow C;, \mathsf{modified}_{\overline{x}}(s)$$

Lemma 3.21 (Semiconcrete Execution Modifies Targets)**.**

$$\mathsf{modifies}_{\mathsf{targets}(c)}\ \mathsf{scexec}(c)$$

3.11.3. *Heap Refinement.* Having discussed the properties of assertion consumption and production, we now discuss the relationship between semiconcrete command execution and concrete command execution. For this purpose, we need to characterize the relationship between semiconcrete states and concrete states.

We say that a concrete heap h_c *refines* a semiconcrete heap h, denoted $h_\mathsf{c} \lhd h$, if h can be obtained from h_c by *closing* some finite number of user-defined predicate chunks. This is expressed formally using the three inference rules shown below.

Definition 3.22 (Heap refinement)**.**

$$\frac{h_\mathsf{c} \lhd h \qquad \textbf{predicate}\ p(\overline{x}) = a \qquad (\mathbf{0}[\overline{x} := \overline{v}], h) \xrightarrow{a}_\mathsf{c} (s', \mathbf{0})}{h_\mathsf{c} \lhd \{\!\!\{p(\overline{v})\}\!\!\}} \qquad \frac{}{h_\mathsf{c} \lhd h_\mathsf{c}} \qquad \frac{h_\mathsf{c} \lhd h \qquad h'_\mathsf{c} \lhd h'}{h_\mathsf{c} \uplus h'_\mathsf{c} \lhd h \uplus h'}$$

The first rule states that if a concrete heap h_c refines a heap h that satisfies the body a of some predicate p, with no chunks left, when consumed under a store that binds the predicate parameters $\overline{x}$ to some argument list $\overline{v}$, then it refines the singleton heap containing just the chunk $p(\overline{v})$. The second rule states that any heap refines itself. The third rule states that heap refinement is compatible with heap union.

Notice that there are typically many concrete heaps that refine a given semiconcrete heap. Consider for example the semiconcrete heap $\{\!\!\{\mathsf{list}(50)\}\!\!\}$, where predicate list is defined as in the example earlier. Any concrete heap that contains exactly a linked list starting at address 50 refines this semiconcrete heap. There are infinitely many such concrete heaps, corresponding to different list lengths, different addresses of nodes, and different values stored in the nodes.

The following property of heap refinement allows us to fold and unfold predicate definitions:

Lemma 3.23 (Open, Close)**.**

$$h_\mathsf{c} \lhd h \uplus \{\!\!\{p(\overline{v})\}\!\!\} \Leftrightarrow \exists s', h'.\ (\mathbf{0}[\overline{x} := \overline{v}], h') \xrightarrow{a}_\mathsf{c} (s', \mathbf{0}) \wedge h_\mathsf{c} \lhd h \uplus h'$$

where **predicate** $p(\overline{x}) = a$

3.11.4. *Soundness of Semiconcrete Execution of Commands.* Given the refinement relation, we can define a *refinement mutator* κ that takes a semiconcrete state as input and outputs a demonically chosen concrete state such that the output heap refines the input heap.

Definition 3.24. Refinement Mutator

$$\kappa = \lambda(s,h).\ \bigotimes h_{\mathrm{c}} \in \mathit{CHeaps}, h_{\mathrm{c}} \lhd h.\ \langle (s, h_{\mathrm{c}}) \rangle$$

Given the refinement mutator, we can state the main lemma for the soundness of semiconcrete execution.

Lemma 3.25 (Soundness of Semiconcrete Execution of Commands)**.** *If* $\forall r.\ \mathsf{valid}(r)$*, then*

$$\mathsf{scexec}(c); \kappa \Rrightarrow \kappa; \mathsf{exec}(c)$$

Proof. It is sufficient to prove

$$\forall n, c.\ \mathsf{scexec}(c); \kappa \Rrightarrow \kappa; \mathsf{exec}_n(c)$$

By induction on n. The base case is trivial. Assume $\forall c.\ \mathsf{scexec}(c); \kappa \Rrightarrow \kappa; \mathsf{exec}_n(c)$. The goal is $\forall c.\ \mathsf{scexec}(c); \kappa \Rrightarrow \kappa; \mathsf{exec}_{n+1}(c)$. By case analysis on c. □

It roughly states that, assuming that all routines are valid, executing a command semiconcretely starting from some semiconcrete state is worse than executing it concretely starting from a demonically chosen corresponding concrete state.

The lemma can be proven by induction on the depth of concrete execution and a case analysis on the command. Most cases are trivial; the nontrivial cases are routine calls, while loops, and **open** and **close** commands. The proofs of the latter cases use the properties of consumption and production.

Below, we sketch the proof in some more detail for the cases of routine calls and while loops.

Proof (Routine Calls). Assume routine definition

$$\textbf{routine}\ r(\overline{x})\ \textbf{req}\ a\ \textbf{ens}\ a' = c$$

The goal is $\mathsf{scexec}(r(\overline{e})); \kappa \Rrightarrow \kappa; \mathsf{cexec}_{n+1}(r(\overline{e}))$. This expands to

$$\overline{v} \leftarrow \mathsf{eval}(\overline{e}); \mathsf{with}(\mathbf{0}[\overline{x} := \overline{v}], \mathsf{consume}(a); \mathsf{produce}(a')); \kappa$$
$$\Rrightarrow \kappa; \overline{v} \leftarrow \mathsf{eval}(\overline{e}); \mathsf{with}(\mathbf{0}[\overline{x} := \overline{e}], \mathsf{cexec}_n(c))$$

We have $\mathsf{eval}(e);, \kappa \Rrightarrow \kappa; \mathsf{eval}(e)$. Furthermore, we have monotonicity of sequential composition of mutators with respect to coverage. Therefore, it is sufficient to fix values $\overline{v}$ and prove

$$\mathsf{with}(\mathbf{0}[\overline{x} := \overline{v}], \mathsf{consume}(a); \mathsf{produce}(a')); \kappa \Rrightarrow \kappa; \mathsf{with}(\mathbf{0}[\overline{x} := \overline{v}], \mathsf{cexec}_n(c))$$

Let $s = \mathbf{0}[\overline{x} := \overline{v}]$. Furthermore, we abbreviate with, $\mathsf{consume}$, $\mathsf{produce}$, scexec, and exec as w, c, p, sce, and e, respectively. The goal then becomes

$$\mathsf{w}(s, \mathsf{c}(a); \mathsf{p}(a')); \kappa \Rrightarrow \kappa; \mathsf{w}(s, \mathsf{e}_n(c))$$

By the induction hypothesis, we have $\mathsf{sce}(c); \kappa \Rrightarrow \kappa; \mathsf{e}_n(c)$ and therefore $\mathsf{w}(s, \mathsf{sce}(c)); \kappa \Rrightarrow \mathsf{w}(s, \mathsf{e}_n(c))$. By transitivity and monotonicity of coverage, it is sufficient to prove
$\mathsf{w}(s, \mathsf{c}(a); \mathsf{p}(a')) \Rrightarrow \mathsf{w}(s, \mathsf{sce}(c))$.

By validity of r, we have

$$(\mathbf{0},\mathbf{0}) \rhd \bigotimes \overline{v}.\ \mathsf{w}(\mathbf{0}[\overline{x} := \overline{v}], s' \leftarrow \mathsf{w}(\mathbf{0}[\overline{x} := \overline{v}], \mathsf{p}(a); \mathsf{store}); \\ \mathsf{sce}(c); \mathsf{w}(s', \mathsf{c}(a'))); \mathsf{leakcheck}\ \{\mathrm{true}\}$$

We abbreviate $\mathsf{w}(s, C; \mathsf{store})$ by $\mathsf{ws}(s, C)$. Furthermore, we instantiate the demonic choice using our fixed $\overline{v}$, and we use $s = \mathbf{0}[\overline{x} := \overline{v}]$, obtaining

$$(\mathbf{0},\mathbf{0}) \rhd \mathsf{w}(s, s' \leftarrow \mathsf{ws}(s, \mathsf{p}(a)); \mathsf{sce}(c); \mathsf{w}(s', \mathsf{c}(a'))); \mathsf{leakcheck}\ \{\mathrm{true}\}$$

It is easy to see that it follows that for any store s_0 and heap h_0, we have $(s_0, \mathbf{0}) \rhd \mathsf{w}(s, s' \leftarrow \mathsf{ws}(s, \mathsf{p}(a)); \mathsf{sce}(c); \mathsf{w}(s', \mathsf{c}(a'))); \mathsf{produce}(h_0)\ \{s_1, h_1.\ s_1 = s_0 \wedge h_1 = h_0\}$. By locality of assertion consumption, semiconcrete execution, and assertion production, we can shift $\mathsf{produce}(h_0)$ to the front, obtaining

$$(s_0, h_0) \rhd \mathsf{w}(s, s' \leftarrow \mathsf{ws}(s, \mathsf{p}(a)); \mathsf{sce}(c); \mathsf{w}(s', \mathsf{c}(a')))\ \{s_1, h_1.\ s_1 = s_0 \wedge h_1 = h_0\}$$

Hence, $\mathsf{noop} \Rrightarrow \mathsf{w}(s, s' \leftarrow \mathsf{ws}(s, \mathsf{p}(a)); \mathsf{sce}(c); \mathsf{w}(s', \mathsf{c}(a')))$.

The goal now follows by simple rewriting, using the rewriting lemmas seen above for consumption followed by production:

$$\begin{aligned}
&\mathsf{w}(s, \mathsf{c}(a); \mathsf{p}(a')) \\
\Rrightarrow\ & s_1 \leftarrow \mathsf{ws}(s, \mathsf{c}(a)); \mathsf{w}(s_1, \mathsf{p}(a')) \\
\Rrightarrow\ & s_1 \leftarrow \mathsf{ws}(s, \mathsf{c}(a)); \mathsf{noop}; \mathsf{w}(s_1, \mathsf{p}(a')) \\
\Rrightarrow\ & s_1 \leftarrow \mathsf{ws}(s, \mathsf{c}(a)); \mathsf{w}(s, s' \leftarrow \mathsf{ws}(s, \mathsf{p}(a)); \mathsf{sce}(c); \mathsf{w}(s', \mathsf{c}(a'))); \mathsf{w}(s_1, \mathsf{p}(a')) \\
\Rrightarrow\ & s_1 \leftarrow \mathsf{ws}(s, \mathsf{c}(a)); s_2 \leftarrow \mathsf{ws}(s, \mathsf{p}(a)); \mathsf{w}(s, \mathsf{sce}(c)); \mathsf{w}(s_2, \mathsf{c}(a')); \mathsf{w}(s_1, \mathsf{p}(a')) \\
\Rrightarrow\ & \mathsf{w}(s, \mathsf{sce}(c)); \mathsf{w}(s'', \mathsf{c}(a')); \mathsf{w}(s'', \mathsf{p}(a')) \\
\Rrightarrow\ & \mathsf{w}(s, \mathsf{sce}(c))
\end{aligned}$$

□

Proof (Loops). The goal is

$$\mathsf{sce}(\textbf{while}\ b\ \textbf{inv}\ a\ \textbf{do}\ c); \kappa \Rrightarrow \kappa; \mathsf{e}_{n+1}(\textbf{while}\ b\ \textbf{inv}\ a\ \textbf{do}\ c)$$

Expanding the definitions, and further abbreviating $\mathsf{modified}$, havoc, assume, $\mathsf{leakcheck}$, $\mathsf{heap} := \mathbf{0}$, $\mathsf{targets}(c)$, $s \leftarrow \mathsf{store}; \mathsf{with}(s, \mathsf{consume}(a))$, and $s \leftarrow \mathsf{store}; \mathsf{with}(s, \mathsf{produce}(a))$ as m, h, a, lck, clh, $\overline{x}$, $\mathsf{cc}(a)$, and $\mathsf{pc}(a)$, our goal reduces to

$$\mathsf{cc}(a); \mathsf{h}(\overline{x}); (\mathsf{clh}; \mathsf{pc}(a); \mathsf{a}(b); \mathsf{sce}(c); \mathsf{cc}(a); \mathsf{lck} \otimes \mathsf{pc}(a); \mathsf{a}(\neg b)); \kappa \Rrightarrow \kappa; (\mathsf{a}(b); \mathsf{e}_n(c))^*; \mathsf{a}(\neg b)$$

Using the property $(\forall s.\ \mathsf{m}_{\overline{x}}(s); C \Rrightarrow C') \Rightarrow C \Rrightarrow C'$, and fixing s, it is sufficient to prove

$$\mathsf{m}_{\overline{x}}(s); \mathsf{cc}(a); \mathsf{h}(\overline{x}); (\mathsf{clh}; \mathsf{pc}(a); \mathsf{a}(b); \mathsf{sce}(c); \mathsf{cc}(a); \mathsf{lck} \otimes \mathsf{pc}(a); \mathsf{a}(\neg b)); \kappa \\ \Rrightarrow \kappa; (\mathsf{a}(b); \mathsf{e}_n(c))^*; \mathsf{a}(\neg b)$$

We now prove the following lemma.

Lemma 3.26. *Assume* $\mathsf{local}\ C$ *and* $\mathsf{modifies}_{\overline{x}}\ C$. *We have*

$$\mathsf{m}_{\overline{x}}(s); \mathsf{h}(\overline{x}); \mathsf{clh}; C; \mathsf{lck} \Rrightarrow \bot \quad \vee \quad \mathsf{m}_{\overline{x}}(s); \mathsf{h}(\overline{x}) \Rrightarrow C$$

Proof. We assume the left-hand disjunct is false and we prove the right-hand disjunct. From this assumption it follows that there exists an initial state (s_0, h_0) such that

$$(s_0, h_0) \rhd \mathsf{m}_{\overline{x}}(s); \mathsf{h}(\overline{x}); \mathsf{clh}; C; \mathsf{lck}\ \{\mathrm{true}\}$$

It follows that $s_0 \overset{\overline{x}}{\sim} s$ and for any $s_1 \overset{\overline{x}}{\sim} s_0$ we have $(s_1, \mathbf{0}) \triangleright C\ \{s', h'.\ h' = \mathbf{0}\}$. Hence, by $\mathsf{modifies}_{\overline{x}}\ C$, we have $(s_1, \mathbf{0}) \triangleright C\ \{s', h'.\ s' \overset{\overline{x}}{\sim} s_1 \wedge h' = \mathbf{0}\}$. Hence, by $\mathsf{local}\ C$, we have, for any h_1, $(s_1, h_1) \triangleright C\ \{s', h'.\ s' \overset{\overline{x}}{\sim} s_1 \wedge h' = h_1\}$. From this our goal follows. □

We have $\mathsf{modifies}_{\overline{x}}\ \mathsf{pc}(a); \mathsf{a}(b); \mathsf{sce}(c); \mathsf{cc}(a)$ and $\mathsf{local}\ \mathsf{pc}(a); \mathsf{a}(b); \mathsf{sce}(c); \mathsf{cc}(a)$; applying the lemma, we obtain

$$\begin{aligned}&\mathsf{m}_{\overline{x}}(s); \mathsf{h}(\overline{x}); \mathsf{clh}; \mathsf{pc}(a); \mathsf{a}(b); \mathsf{sce}(c); \mathsf{cc}(a); \mathsf{lck} \Rrightarrow \bot \\ &\qquad\qquad\qquad \vee\ \mathsf{m}_{\overline{x}}(s); \mathsf{h}(\overline{x}) \Rrightarrow \mathsf{pc}(a); \mathsf{a}(b); \mathsf{sce}(c); \mathsf{cc}(a)\end{aligned}$$

We consider both cases. In the first case, the goal follows trivially. In the remainder of the proof, we assume the second case.

Using the property $C_2 \Rrightarrow C_3 \Rightarrow C_1 \otimes C_2 \Rrightarrow C_3$, we drop the left-hand side of the demonic choice in our goal. Our goal becomes

$$\mathsf{m}_{\overline{x}}(s); \mathsf{cc}(a); \mathsf{h}(\overline{x}); \mathsf{pc}(a); \mathsf{a}(\neg b); \kappa \Rrightarrow \kappa; (\mathsf{a}(b); \mathsf{e}_n(c))^*; \mathsf{a}(\neg b)$$

Applying the induction hypothesis, we have $(\mathsf{a}(b); \mathsf{sce}(c))^*; \mathsf{a}(\neg b); \kappa \Rrightarrow \kappa; (\mathsf{a}(b); \mathsf{e}_n(c))^*; \mathsf{a}(\neg b)$. By transitivity of coverage and monotonicity of mutator sequential composition with respect to coverage, it is sufficient to prove

$$\mathsf{m}_{\overline{x}}(s); \mathsf{cc}(a); \mathsf{h}(\overline{x}); \mathsf{pc}(a) \Rrightarrow (\mathsf{a}(b); \mathsf{sce}(c))^*$$

Note that to prove $C' \Rrightarrow C^*$, it is sufficient to prove $C' \Rrightarrow \mathsf{noop}$ and $C' \Rrightarrow C; C'$. Applying this rule to the goal, the first subgoal is easy to prove (using the properties of consumption followed by production). Our remaining goal is

$$\mathsf{m}_{\overline{x}}(s); \mathsf{cc}(a); \mathsf{h}(\overline{x}); \mathsf{pc}(a) \Rrightarrow \mathsf{a}(b); \mathsf{sce}(c); \mathsf{m}_{\overline{x}}(s); \mathsf{cc}(a); \mathsf{h}(\overline{x}); \mathsf{pc}(a)$$

The goal now follows by simple rewriting, using the rewriting lemmas seen above for consumption followed by production, as well as the properties $\mathsf{m}_{\overline{x}}(s) \Rrightarrow \mathsf{m}_{\overline{x}}(s); \mathsf{m}_{\overline{x}}(s)$, $\mathsf{h}(\overline{x}) \Rrightarrow \mathsf{h}(\overline{x}); \mathsf{h}(\overline{x})$, and $\mathsf{modifies}_{\overline{x}}\ \mathsf{sce}(c)$:

$$\begin{array}{ll} & \mathsf{m}_{\overline{x}}(s); \mathsf{cc}(a); \mathsf{h}(\overline{x}); \mathsf{pc}(a) \\ \Rrightarrow & \mathsf{m}_{\overline{x}}(s); \mathsf{cc}(a); \mathsf{m}_{\overline{x}}(s_0); \mathsf{h}(\overline{x}); \mathsf{h}(\overline{x}); \mathsf{pc}(a) \\ \Rrightarrow & \mathsf{m}_{\overline{x}}(s); \mathsf{cc}(a); \mathsf{pc}(a); \mathsf{a}(b); \mathsf{sce}(c); \mathsf{cc}(a); \mathsf{h}(\overline{x}); \mathsf{pc}(a) \\ \Rrightarrow & \mathsf{m}_{\overline{x}}(s); \mathsf{a}(b); \mathsf{sce}(c); \mathsf{cc}(a); \mathsf{h}(\overline{x}); \mathsf{pc}(a) \\ \Rrightarrow & \mathsf{a}(b); \mathsf{sce}(c); \mathsf{m}_{\overline{x}}(s); \mathsf{cc}(a); \mathsf{h}(\overline{x}); \mathsf{pc}(a) \end{array}$$

□

3.12. **Soundness of Semiconcrete Execution.**

Theorem 3.27 (Soundness of Semiconcrete Execution)**.**

$$\mathsf{sc\text{-}safe\_program}(c) \Rightarrow \mathsf{safe\_program}(c)$$

The soundness of semiconcrete execution follows directly from the soundness of semiconcrete execution of commands. Therefore, we are now halfway on our way towards a formalization and soundness proof of Featherweight VeriFast. Semiconcrete execution is not suitable as a verification algorithm since it performs infinite branching. In the next section, we formalize and sketch a soundness proof of Featherweight VeriFast's symbolic execution algorithm, which builds on semiconcrete execution but introduces *symbols* to eliminate infinite branching.

```
routine range(i, n, r)
   req r ↦ ?dummy ens r ↦ ?list ∗ list(list)
$\Phi{:}\{i,n,r\}, s{:}\mathbf{0}[\mathsf{i}{:}i, \mathsf{n}{:}n, \mathsf{r}{:}r], h{:}\mathbf{0}$          $\Phi{:}\{\dots,\varsigma,\dots\} = \Phi{:}\{\dots,\varsigma=\varsigma,\dots\}$
sproduce(r ↦ ?dummy)
$\Phi{:}\{i,n,r,d\}, s{:}\mathbf{0}[\mathsf{i}{:}i, \mathsf{n}{:}n, \mathsf{r}{:}r], h{:}\{\!\{r\mapsto d\}\!\}$
if i = n then l := 0 else (
$\Phi{:}\{i,n,r,d, i\neq n\}, s{:}\mathbf{0}[\mathsf{i}{:}i, \mathsf{n}{:}n, \mathsf{r}{:}r], h{:}\{\!\{r\mapsto d\}\!\}$
l := malloc(2);
$\Phi{:}\{i,n,r,d,l,v,v', i\neq n,0<l\}, s{:}\mathbf{0}[\mathsf{i}{:}i, \mathsf{n}{:}n, \mathsf{r}{:}r, \mathsf{l}{:}l], h{:}\{\!\{r\mapsto d,\mathsf{mb}(l,2),l\mapsto v,l+1\mapsto v'\}\!\}$
[l] := i; range(i + 1, n, l + 1)
sconsume(l+1↦?dummy); sproduce(l+1↦?list ∗ list(list))
$\Phi{:}\{i,n,r,d,l,v,v',l', i\neq n,0<l\}, s{:}\mathbf{0}[\mathsf{i}{:}i, \mathsf{n}{:}n, \mathsf{r}{:}r, \mathsf{l}{:}l]$,
$h{:}\{\!\{r\mapsto d,\mathsf{mb}(l,2),l\mapsto i,l+1\mapsto l',\mathsf{list}(l')\}\!\}$
);  close list(l); [r] := l
$\Phi{:}\{i,n,r,d,l,v,v',l', i\neq n,0<l\}, s{:}\mathbf{0}[\mathsf{i}{:}i, \mathsf{n}{:}n, \mathsf{r}{:}r, \mathsf{l}{:}l], h{:}\{\!\{r\mapsto l,\mathsf{list}(l)\}\!\}$
sconsume(r ↦ ?list ∗ list(list))
$\Phi{:}\{i,n,r,d,l,v,v',l', i\neq n,0<l\}, s{:}\mathbf{0}[\mathsf{i}{:}i, \mathsf{n}{:}n, \mathsf{r}{:}r, \mathsf{l}{:}l], h{:}\mathbf{0}$
```

Figure 10: Symbolic Execution: Example Trace

4. Symbolic Execution

In this section, we introduce symbolic execution by example, and then provide formal definitions. Finally, we sketch a soundness proof.

4.1. **Symbolic Execution: Example Trace.**

Recall the example semiconcrete execution trace for the example routine range in Figure 7. Notice that while the length of this trace is linear in the size of the body of routine range, there are infinitely many such traces, since each number shown in orange is picked by demonic choice among all integers (potentially with some constraints).

We introduce *symbolic execution* to arrive at an execution with a finite number of traces of limited length. Instead of demonically choosing among an infinite set of integers, symbolic execution uses a fresh *symbol* to represent an arbitrary number. Symbolic execution states are like semiconcrete execution states, except that a *term* may be used instead of a literal value in the store and the heap. A term is either a literal number, a symbol, or an operation (addition or subtraction) applied to two terms. In addition to replacing numbers by terms, symbolic execution adds a third component to the state: the *path condition*. This is a set of *formulae* that define the set of *relevant interpretations* of the symbols used in the store and the heap. A formula is either an equality between terms ($t = t'$), an inequality between terms ($t < t'$), or the negation of another formula.

Example 4.1. Symbolic Execution: Example Trace See Figure 10

In Figure 10 we show the symbolic execution trace for routine range corresponding to the semiconcrete execution trace shown before. Note: do not confuse the program variables and the symbols. The former are shown in an upright font; the latter are shown in a slanted font. In the symbolic execution trace, the letters shown in orange do not denote branching (i.e. demonic choices); rather, they show freshly picked symbols.

Besides the use of symbols, notice the path condition Φ: it starts out empty; in the **else** branch of the **if** statement, the formula $\mathsf{i} \neq \mathsf{n}$ is added; and the **malloc** statement adds the formula $0 < \mathsf{l}$.

4.2. **Symbolic Execution: Types.** The set $\mathsf{SStates}$ of symbolic execution states is defined below. Terms are like expressions, except that they may mention symbols, which represent a fixed value, instead of program variables, whose value may change through assignments. Similarly, formulae correspond to boolean expressions.

Symbolic states are like semiconcrete states, except that terms are used instead of values in the store and as chunk arguments; furthermore, the state includes an additional component, called the path condition, which is a set of formulae.

Definition 4.2. Symbolic Execution: Types

$$\begin{array}{rl}
& \varsigma \in \mathit{Symbols} \\
t, \hat{\ell}, \hat{v} \in \mathit{Terms} & ::= z \mid \varsigma \mid t + t \mid t - t \\
\varphi \in \mathit{Formulae} & ::= t = t \mid t < t \mid \neg\varphi \\
& \\
\hat{s} \in \mathit{SStores} = & \mathit{Vars} \to \mathit{Terms} \\
\mathit{SPredicates} = & \{\mapsto, \mathsf{mb}\} \cup \mathit{UserDefinedPredicates} \\
\mathit{SChunks} = & \{p(\overline{\hat{v}}) \mid p \in \mathit{SPredicates}, \overline{\hat{v}} \in \mathit{Terms}\} \\
\hat{h} \in \mathit{SHeaps} = & \mathit{SChunks} \to \mathbb{N} \\
\mathit{PathConditions} = & \mathcal{P}(\mathit{Formulae}) \\
\mathit{SStates} = & \mathit{PathConditions} \times \mathit{SStores} \times \mathit{SHeaps} \\
\mathit{SMutators} = & \mathit{SStates} \to \mathit{Outcomes}(\mathit{SStates}) \\
& \\
\mathit{sconsume}(a) \in & \mathit{Assertions} \to \mathit{SMutators} \\
\mathit{sproduce}(a) \in & \mathit{Assertions} \to \mathit{SMutators} \\
\mathit{symexec}(c) \in & \mathit{Commands} \to \mathit{SMutators}
\end{array}$$

4.3. **Symbolic Execution: Auxiliary Definitions.** As we did for concrete execution and semiconcrete execution, we introduce a few auxiliary definitions for use in the definition of symbolic execution. They are as follows.

In concrete and semiconcrete execution, assuming a boolean expression evaluates the expression in the current store and blocks if it evaluates to false. In symbolic execution, this is not possible, since evaluation of a boolean expression under a symbolic store yields a formula rather than a boolean value. Symbolic execution, therefore, asks an *SMT solver*, a type of automatic theorem prover, to try to prove that the formula is inconsistent with the path condition. If it succeeds, symbolic execution blocks. Otherwise, the formula is added to the path condition, in order to record that on the remainder of the current symbolic execution path, of all possible interpretations of the symbols used in the symbolic state, only the ones that satisfy the formula are relevant.

We write $\Phi \vdash_{\mathrm{SMT}} \varphi$ to denote that the SMT solver succeeds in proving that the set of formulae Φ implies the formula φ.

Similarly, asserting a boolean expression in symbolic execution means evaluating it to a formula under the current symbolic store and asking the SMT solver to try to prove that the formula follows from the path condition. If it succeeds, execution proceeds normally; otherwise, it fails.

The set $\mathsf{Used}(\Phi)$ denotes the set of symbols ς for which a formula $\varsigma = \varsigma$ appears in the path condition Φ. In a well-formed symbolic state, all symbols used in the symbolic state are in this set.

$\mathsf{fresh}(\Phi)$ denotes some symbol that is not in $\mathsf{Used}(\Phi)$. It is defined using a *choice function* ϵ, which maps each nonempty set to some element of that set.

The mutator fresh picks some symbol ς that is not yet used by the current symbolic state, records that it is now being used by adding a formula $\varsigma = \varsigma$ to the path condition, and yields the symbol as its answer.

We define the notation $\bigoplus t.\ C(t)$, where C is a mutator parameterized by a term, to denote angelic choice over all terms that only use symbols that are already being used by the current symbolic state. $\mathrm{FS}(t)$ denotes the set of free symbols that appear in term t, i.e. the set of symbols used by t.[8]

Symbolic consumption $\mathsf{sconsume\_chunks}(\hat{h})$ of a multiset $\hat{h}$ of symbolic terms differs from concrete and semiconcrete consumption in that it does not simply look for the exact chunks $\hat{h}$ in the current heap; rather, it looks for chunks for which the SMT solver succeeds in proving that their argument terms are equal under all relevant interpretations of the symbols. For example, suppose the heap contains a chunk $\mathsf{list}(\mathsf{l})$ and the path condition contains a formula $\mathsf{l} = \mathsf{l}'$; then consumption of a chunk $\mathsf{list}(\mathsf{l}')$ succeeds, even though the exact chunk $\mathsf{list}(\mathsf{l}')$ does not appear in the symbolic heap. Symbolic production is simpler; as in semiconcrete execution, it simply adds the specified chunks to the heap. Symbolic consumption and production of a single symbolic chunk $\hat{\alpha}$ are defined in the obvious way.

Definition 4.3. Symbolic Execution: Auxiliary Definitions

$$
\begin{array}{l}
\mathsf{sassume}(\varphi) = \lambda(\Phi, \hat{s}, \hat{h}).\ \bigotimes \Phi \nvdash_{\mathrm{SMT}} \neg\varphi.\ \langle(\Phi \cup \{\varphi\}, \hat{s}, \hat{h})\rangle \\
\mathsf{sassume}(b) = \hat{s} \leftarrow \mathsf{sstore}; \mathsf{sassume}([\![b]\!]_{\hat{s}}) \\
\mathsf{sassert}(b) = \lambda(\Phi, \hat{s}, \hat{h}).\ \bigoplus \Phi \vdash_{\mathrm{SMT}} [\![b]\!]_{\hat{s}}.\ \langle(\Phi, \hat{s}, \hat{h})\rangle \\
\mathsf{Used}(\Phi) = \{\varsigma \in \mathit{Symbols} \mid (\varsigma = \varsigma) \in \Phi\} \\
\mathsf{fresh}(\Phi) = \epsilon(\{\varsigma \in \mathit{Symbols} \mid \varsigma \notin \mathsf{Used}(\Phi)\}) \\
\mathsf{fresh} = \lambda(\Phi, \hat{s}, \hat{h}).\ \mathbf{let}\ \varsigma = \mathsf{fresh}(\Phi)\ \mathbf{in}\ \langle(\Phi \cup \{\varsigma = \varsigma\}, \hat{s}, \hat{h}), \varsigma\rangle \\
\\
\bigoplus t.\ C(t) = \Phi \leftarrow \mathsf{pc}; \bigoplus t \in \mathit{Terms}, \mathrm{FS}(t) \subseteq \mathsf{Used}(\Phi).\ C(t) \\
\mathsf{sconsume\_chunks}(\hat{h}') = \lambda(\Phi, \hat{s}, \hat{h}).\ \bigotimes \hat{h}'' \leq \hat{h}, \Phi \vdash_{\mathrm{SMT}} \hat{h}'' = \hat{h}'.\ \langle(\Phi, \hat{s}, \hat{h} - \hat{h}'')\rangle \\
\mathsf{sconsume\_chunk}(\hat{\alpha}) = \mathsf{sconsume\_chunks}(\{\!\!\{\hat{\alpha}\}\!\!\}) \\
\mathsf{sproduce\_chunks}(\hat{h}') = \lambda(\Phi, \hat{s}, \hat{h}).\ \langle(\Phi, \hat{s}, \hat{h} \uplus \hat{h}')\rangle \\
\mathsf{sproduce\_chunk}(\hat{\alpha}) = \mathsf{sproduce\_chunks}(\{\!\!\{\hat{\alpha}\}\!\!\})
\end{array}
$$

where

$$\epsilon(X) \quad = \quad \text{some element of } X$$

[8]Since the syntax of terms does not include any binding constructs, all symbols that appear in a term are free symbols of the term.

4.4. **Symbolic Execution: Definition.** The definition of symbolic execution is entirely analogous to that of semiconcrete execution, except that symbolic versions of the auxiliary mutators are used and that each demonic choice over all values is replaced by picking a fresh symbol.

Definition 4.4. Producing Assertions

$$\begin{array}{l}
\mathsf{sproduce}(b) = \mathsf{sassume}(b) \\[6pt]
\mathsf{sproduce}(p(\overline{e}, \overline{?x})) = \\
\quad \overline{\hat{v}} \leftarrow \mathsf{seval}(e); \overline{\hat{v}}' \leftarrow \mathsf{fresh}; \mathsf{sproduce\_chunk}(p(\overline{\hat{v}}, \overline{\hat{v}}')); \overline{x} := \overline{\hat{v}}' \\[6pt]
\mathsf{sproduce}(a * a') = \mathsf{sproduce}(a); \mathsf{sproduce}(a') \\[6pt]
\mathsf{sproduce}(\textbf{if } b \textbf{ then } a \textbf{ else } a') = \\
\quad \mathsf{sassume}(b); \mathsf{sproduce}(a) \otimes \mathsf{sassume}(\neg b); \mathsf{sproduce}(a')
\end{array}$$

Definition 4.5. Consuming Assertions

$$\begin{array}{l}
\mathsf{sconsume}(b) = \mathsf{sassert}(b) \\[6pt]
\mathsf{sconsume}(p(\overline{e}, \overline{?x})) = \\
\quad \overline{\hat{v}} \leftarrow \mathsf{seval}(e); \bigoplus \overline{\hat{v}}'.\ \mathsf{sconsume\_chunk}(p(\overline{\hat{v}}, \overline{\hat{v}}')); \overline{x} := \overline{\hat{v}}' \\[6pt]
\mathsf{sconsume}(a * a') = \mathsf{sconsume}(a); \mathsf{sconsume}(a') \\[6pt]
\mathsf{sconsume}(\textbf{if } b \textbf{ then } a \textbf{ else } a') = \\
\quad \mathsf{sassume}(b); \mathsf{sconsume}(a) \otimes \mathsf{sassume}(\neg b); \mathsf{sconsume}(a')
\end{array}$$

Definition 4.6. Symbolic Execution of Commands See Figures 11 and 12.

Definition 4.7. Validity of Routines

```
svalid(r) =
   (∅, 0, 0) ▷
   $\overline{\hat{v}} \leftarrow$ fresh;
   with($\mathbf{0}[\overline{x} := \overline{\hat{v}}]$,
      $\hat{s}' \leftarrow$ with($\mathbf{0}[\overline{x} := \overline{\hat{v}}]$, sproduce($a$); sstore);
      symexec($c$);
      with($\hat{s}'$, sconsume($a'$))
   );
   sleakcheck
   {true}
   where routine $r(\overline{x})$ req $a$ ens $a' = c$
```

Definition 4.8. Symbolic Execution: Program Safety

$$\mathsf{sym\text{-}safe\_program}(c) \quad = \quad (\forall r.\ \mathsf{svalid}(r)) \wedge (\emptyset, \mathbf{0}, \mathbf{0}) \triangleright \mathsf{symexec}(c)\ \{\mathrm{true}\}$$

$\mathsf{symexec}(x := e) = \hat{v} \leftarrow \mathsf{seval}(e); x := \hat{v}$

$\mathsf{symexec}(c; c') = \mathsf{symexec}(c); \mathsf{symexec}(c')$

$\mathsf{symexec}(\textbf{if } b \textbf{ then } a \textbf{ else } a') =$
$\quad \mathsf{sassume}(b); \mathsf{symexec}(c) \otimes \mathsf{sassume}(\neg b); \mathsf{symexec}(c')$

$\mathsf{symexec}(\textbf{while } e \textbf{ inv } a \textbf{ do } c) = \text{See Figure 12}$

$\mathsf{symexec}(r(\overline{e})) =$
$\quad \overline{\hat{v}} \leftarrow \mathsf{eval}(\overline{e}); \mathsf{with}(\mathbf{0}[\overline{x} := \overline{\hat{v}}], \mathsf{sconsume}(a); \mathsf{sproduce}(a'))$
$\quad \text{where } \textbf{routine } r(\overline{x}) \textbf{ req } a \textbf{ ens } a'$

$\mathsf{symexec}(x := \textbf{malloc}(n)) =$
$\quad \hat{\ell}, \hat{v}_1, \ldots, \hat{v}_n \leftarrow \mathsf{fresh}; \mathsf{sassume}(0 < \hat{\ell});$
$\quad \mathsf{sproduce\_chunks}(\{\!\!\{\mathsf{mb}(\hat{\ell}, n), \hat{\ell} \mapsto \hat{v}_1, \ldots, \hat{\ell} + n - 1 \mapsto \hat{v}_n\}\!\!\}); x := \hat{\ell}$

$\mathsf{symexec}(x := [e]) =$
$\quad \hat{\ell} \leftarrow \mathsf{seval}(e); \bigoplus \hat{v}.\ \mathsf{sconsume\_chunk}(\hat{\ell} \mapsto \hat{v}); \mathsf{sproduce\_chunk}(\hat{\ell} \mapsto \hat{v}); x := \hat{v}$

$\mathsf{symexec}([e] := e') =$
$\quad \hat{\ell}, \hat{v} \leftarrow \mathsf{seval}(e, e'); \bigoplus \hat{v}'.\ \mathsf{sconsume\_chunk}(\hat{\ell} \mapsto \hat{v}'); \mathsf{sproduce\_chunk}(\hat{\ell} \mapsto \hat{v})$

$\mathsf{symexec}(\textbf{free}(e)) = \hat{\ell} \leftarrow \mathsf{seval}(e);$
$\quad \bigoplus n, \hat{v}_1, \ldots, \hat{v}_n.\ \mathsf{sconsume\_chunks}(\{\!\!\{\mathsf{mb}(\hat{\ell}, n), \hat{\ell}_1 \mapsto \hat{v}_1, \ldots, \hat{\ell}_n \mapsto \hat{v}_n\}\!\!\})$

$\mathsf{symexec}(\textbf{open } p(\overline{e})) = \overline{\hat{v}} \leftarrow \mathsf{eval}(\overline{e});$
$\quad \mathsf{sconsume\_chunk}(p(\overline{\hat{v}})); \mathsf{with}(\mathbf{0}[\overline{x} := \overline{\hat{v}}], \mathsf{sproduce}(a))$
$\quad \text{where } \textbf{predicate } p(\overline{x}) = a$

$\mathsf{symexec}(\textbf{close } p(\overline{e})) = \overline{\hat{v}} \leftarrow \mathsf{eval}(\overline{e});$
$\quad \mathsf{with}(\mathbf{0}[\overline{x} := \overline{\hat{v}}], \mathsf{sconsume}(a)); \mathsf{sproduce\_chunk}(p(\overline{\hat{v}}))$
$\quad \text{where } \textbf{predicate } p(\overline{x}) = a$

Figure 11: Symbolic Execution of Commands

4.5. **Soundness.** We now argue the soundness of symbolic execution with respect to semiconcrete execution, i.e. that symbolic execution is a safe approximation of semiconcrete execution, and therefore if symbolic execution does not fail, then semiconcrete execution does not fail. To do so, we need to characterize the relationship between symbolic states and semiconcrete states. We do so by means of the concept of an *interpretation*.

Definition 4.9. Soundness of symbolic execution: Definitions

```
$\mathsf{shavoc}(\overline{x}) = \overline{\hat{v}} \leftarrow \mathsf{fresh}; \overline{x} := \overline{\hat{v}}$
$\mathsf{sleakcheck} = \lambda(\Phi, \hat{s}, \hat{h}).\ \bigoplus \hat{h} = \mathbf{0}.\ \top$

$\mathsf{symexec}(\mathbf{while}\ b\ \mathbf{inv}\ a\ \mathbf{do}\ c) =$
  $\hat{s} \leftarrow \mathsf{sstore}; \mathsf{with}(\hat{s}, \mathsf{sconsume}(a));$
  $\mathsf{shavoc}(\mathsf{targets}(c));$
  $($
    $\mathsf{sheap} := \mathbf{0};$
    $\hat{s} \leftarrow \mathsf{sstore}; \mathsf{with}(\hat{s}, \mathsf{sproduce}(a));$
    $\mathsf{sassume}(b); \mathsf{symexec}(c);$
    $\hat{s} \leftarrow \mathsf{sstore}; \mathsf{with}(\hat{s}, \mathsf{sconsume}(a));$
    $\mathsf{sleakcheck}$
  $\otimes$
    $\hat{s} \leftarrow \mathsf{sstore}; \mathsf{with}(\hat{s}, \mathsf{sproduce}(a))$
    $\mathsf{sassume}(\neg b);$
  $)$
```

Figure 12: Symbolic Execution of Loops

$$\begin{array}{rcl}
I \in \mathit{Interps} & = & \mathit{Symbols} \rightharpoonup \mathbb{Z} = \mathit{Symbols} \to \mathbb{Z} \cup \{\mathsf{undef}\} \\
\operatorname{dom} I & = & \{\varsigma \mid I(\varsigma) \neq \mathsf{undef}\} \\
I \subseteq I' & = & \forall \varsigma.\ I(\varsigma) = \mathsf{undef} \lor I(\varsigma) = I'(\varsigma) \\
I((\Phi, \hat{s}, \hat{h})) & = & \begin{cases} (s, h) & \text{if } \operatorname{dom} I = \mathsf{Used}(\Phi) \land [\![\Phi, \hat{s}, \hat{h}]\!]_I = \mathsf{true}, s, h \\ \mathsf{undef} & \text{otherwise} \end{cases} \\
\rho_I & = & \lambda\hat{\sigma}.\ \bigotimes I' \supseteq I, \sigma, I'(\hat{\sigma}) = \sigma.\ \langle \sigma \rangle \\
C \rightsquigarrow_I C' & = & C;, \rho_I \Rrightarrow \rho_I; C' \\
C(-) \rightsquigarrow_I C'(-) & = & \forall I' \supseteq I, t, v, [\![t]\!]_{I'} = v.\ C(t) \rightsquigarrow_{I'} C'(v)
\end{array}$$

An interpretation is a partial function from symbols to program values. By *partial function*, we mean that it maps each symbol either to a program value (an integer) or to the special value undef. By this, we reflect that at each point during symbolic execution, only some of the symbols are in use and the others may be picked by a future execution of mutator fresh.

We say an interpretation I' *extends* another interpretation I, denoted $I' \supseteq I$, if for each symbol for which I is defined, I' is defined and I' maps it to the same value as I.

We define the evaluation $[\![-]\!]_I$ of a term, a formula, a path condition, a symbolic store, or a symbolic heap under an interpretation I as the partial function that yields undef if the interpretation yields undef for any of the symbols that appear in the input, and the output obtained by replacing all symbols by their value otherwise.

We also use an interpretation as a partial function from symbolic states to semiconcrete states, as follows. For an interpretation I and a symbolic state $(\Phi, \hat{s}, \hat{h})$, if the domain of I is exactly $\mathsf{Used}(\Phi)$, and Φ evaluates to true under I, and the symbolic store and heap $\hat{s}$ and $\hat{h}$ evaluate to a semiconcrete store and heap s and h under I, then the value of $(\Phi, \hat{s}, \hat{h})$ under I is (s, h), and otherwise it is undefined. Notice that this means that the interpretation of a symbolic state is undefined if the symbolic state is not well-formed, i.e. if it uses symbols ς for which no formula $\varsigma = \varsigma$ appears in the path condition.

We now define the *interpretation mutator* ρ_I that, for a given symbolic state $\hat{\sigma}$, demonically chooses an extension I' of I for which $I'(\hat{\sigma})$ is defined and sets the resulting semiconcrete state as the current state.

Given this mutator, we define the concept of *safe approximation* $C \rightsquigarrow_I C'$ of a semiconcrete mutator C' by a symbolic mutator C under an interpretation I. This holds if $C;, \rho_I$ covers $\rho_I; C'$.

We extend this notion to the case of a symbolic operator $C(-)$ parameterized by a term and a semiconcrete operator $C'(-)$ parameterized by a value. It holds if for any extension I' of I, and for any term whose value is defined under I', $C(t)$ safely approximates $C'(\llbracket t \rrbracket_{I'})$ under I'.

Definition 4.10. Logical Consequence

$$\Phi \vDash \varphi \quad \Leftrightarrow \quad \forall I.\ \llbracket \Phi \rrbracket_I = \mathsf{true} \Rightarrow \llbracket \varphi \rrbracket_I = \mathsf{true}$$

Assumption 4.11 (SMT Solver Soundness)**.**

$$\Phi \vdash_{\mathrm{SMT}} \varphi \quad \Rightarrow \quad \Phi \vDash \varphi$$

Soundness of symbolic execution relies on one assumption: that the SMT solver is sound. That is, if the SMT solver reports success in proving that a formula follows from a path condition, then it must be the case that this formula does indeed follow from this path condition. We say a formula follows from a path condition if all interpretations that satisfy the path condition satisfy the formula.

It is not necessary for soundness of symbolic execution that the SMT solver be *complete*, i.e. that it succeed in proving all true facts. In fact, symbolic execution is sound even when using an SMT solver that does not even try and always reports failure to prove a fact. However, in that case symbolic execution itself is highly incomplete, i.e. it fails even if concrete execution does not fail. Indeed, we do not claim completeness of Featherweight VeriFast.

Given these concepts, we can state the soundness lemmas of symbolic execution:

Lemma 4.12 (Soundness)**.**

$$\begin{array}{rcl}
C(-) \rightsquigarrow_I C'(-) \Rightarrow \hat{v} \leftarrow \mathsf{fresh}; C(\hat{v}) & \rightsquigarrow_I & \bigotimes v.\ C'(v) \\
C(-) \rightsquigarrow_I C'(-) \Rightarrow \bigoplus \hat{v}.\ C(\hat{v}) & \rightsquigarrow_I & \bigoplus v.\ C'(v) \\
\mathsf{sassume}(b), \mathsf{sassert}(b) & \rightsquigarrow_I & \mathsf{assume}(b), \mathsf{assert}(b) \\
\llbracket \hat{h} \rrbracket_I = h \Rightarrow \mathsf{sconsume}(\hat{h}), \mathsf{sproduce}(\hat{h}) & \rightsquigarrow_I & \mathsf{consume}(h), \mathsf{produce}(h) \\
\mathsf{sconsume}(a), \mathsf{sproduce}(a) & \rightsquigarrow_I & \mathsf{consume}(a), \mathsf{produce}(a) \\
\mathsf{symexec}(c) & \rightsquigarrow_I & \mathsf{scexec}(c) \\
\mathsf{svalid}(r) & \Rightarrow & \mathsf{valid}(r) \\
\mathsf{sym\text{-}safe\_program}(c) & \Rightarrow & \mathsf{sc\text{-}safe\_program}(c)
\end{array}$$

Mutator fresh safely approximates demonic choice of a value; angelic choice of a term that uses only symbols already being used by the current symbolic state safely approximates angelic choice of a value; symbolic assumption and assertion safely approximate semiconcrete assumption and assertion; symbolic consumption and production of heap chunks safely approximate semiconcrete consumption and production of their interpretations; and symbolic execution safely approximates semiconcrete execution. The soundness theorem follows directly.

Proving the properties stated above is mostly easy; below we go into some detail of two of the more interesting proofs: soundness of fresh and soundness of $\mathsf{sassume}$.

Lemma 4.13 (Soundness of fresh)**.**

$$C(-) \rightsquigarrow_I C'(-) \Rightarrow \hat{v} \leftarrow \mathsf{fresh}; C(\hat{v}) \rightsquigarrow_I \bigotimes v.\ C'(v)$$

Proof. We assume the premise and we unfold the definition of safe approximation, of mutator coverage, and of outcome coverage. Fix an input symbolic state $(\Phi, \hat{s}, \hat{h})$ and a postcondition Q. Unfold the definition of fresh. Let ς be the fresh symbol. Assume $(\Phi \cup \{\varsigma = \varsigma\}, \hat{s}, \hat{h}) \triangleright C(\varsigma); \rho_I\ \{Q\}$. It is sufficient to prove $(\Phi, \hat{s}, \hat{h}) \triangleright \rho_I; \bigotimes v.\ C'(v)\ \{Q\}$. Unfolding the definition of ρ_I in the goal, fix an interpretation $I' \supseteq I$ and a semiconcrete state (s, h) such that $I'((\Phi, \hat{s}, \hat{h})) = (s, h)$. Further fix a value v picked by the demonic choice in the goal. It is sufficient to prove that $(s, h) \triangleright C'(v)\ \{Q\}$. We build a new interpretation I'' by binding the fresh symbol ς to value v: $I'' = I'[\varsigma := v]$. It follows that $I''((\Phi \cup \{\varsigma = \varsigma), \hat{s}, \hat{h})) = (s, h)$. Using I'', we can rewrite our goal into the following form:

$$(\Phi \cup \{\varsigma = \varsigma\}, \hat{s}, \hat{h}) \triangleright \rho_{I''}; C'(v)\ \{Q\}$$

The goal now matches the consequent of our premise $C(-) \rightsquigarrow_I C'(-)$ after unfolding the definition of safe approximation, mutator coverage, and outcome coverage. Finally, the antecedent matches our assumption. □

Lemma 4.14 (Soundness of $\mathsf{sassume}$)**.**

$$\mathsf{sassume}(b) \rightsquigarrow_I \mathsf{assume}(b)$$

Proof. Unfold the definition of safe approximation, mutator coverage, outcome coverage, and ρ_I. Fix an input symbolic state $(\Phi, \hat{s}, \hat{h})$, a postcondition Q, an extension $I' \supseteq I$, and a semiconcrete state (s, h) such that $I'((\Phi, \hat{s}, \hat{h})) = (s, h)$. Unfold the definition of $\mathsf{sassume}$. Assume $\Phi \nvdash_{\mathrm{SMT}} \neg[\![b]\!]_{\hat{s}} \Rightarrow (\Phi \cup \{[\![b]\!]_{\hat{s}}\}, \hat{s}, \hat{h}) \triangleright \rho_I\ \{Q\}$. Unfold the definition of assume. Assume $[\![b]\!]_s = \mathsf{true}$. Our goal reduces to $(s, h) \in Q$.

Since $[\![\Phi]\!]_{I'} = \mathsf{true}$ and $[\![[\![b]\!]_{\hat{s}}]\!]_{I'} = \mathsf{true}$, we have $\Phi \nvDash \neg[\![b]\!]_{\hat{s}}$. By soundness of the SMT solver, it follows that $\Phi \nvdash_{\mathrm{SMT}} \neg[\![b]\!]_{\hat{s}}$. Hence, by our assumption above, $(\Phi \cup \{[\![b]\!]_{\hat{s}}\}, \hat{s}, \hat{h}) \triangleright \rho_I\ \{Q\}$. In this fact, we unfold ρ_I and instantiate the demonic choice with I'. Since $I'((\Phi \cup \{[\![b]\!]_{\hat{s}}\}, \hat{s}, \hat{h})) = (s, h)$, we obtain $(s, h) \in Q$. □

Theorem 4.15 (Soundness of Featherweight VeriFast)**.**

$$\mathsf{sym\text{-}safe\_program}(c) \Rightarrow \mathsf{safe\_program}(c)$$ □

Combining the soundness of symbolic execution with respect to semiconcrete execution and the soundness of semiconcrete execution with respect to concrete execution, we obtain the soundness of Featherweight VeriFast: if symbolic execution does not fail, then concrete execution does not fail.

5. Mechanisation

Above we presented a formal definition of Featherweight VeriFast and we gave the highlights of a proof of its soundness. We hope that the definitions are clear and the proof outline is convincing. However, the definition, while formal (in the sense of: consisting of symbols rather than natural language), is written in the general language of mathematics and not in any particular explicitly defined *formal logic*, with a well-defined formal language of *formulae* and a well-defined formal language of *proofs* that specifies which formulae are *logically true*. Therefore, the precise meaning of the definition might not be clear to all readers. A fortiori, the soundness proof is not expressed in such a formal language of proofs, and therefore, there is always the possibility that some of the inferences made are *invalid* and the conclusion is *false*; i.e., it is not an argument that will necessarily convince all readers.

To address these limitations, we developed a definition and soundness proof of a slight variant of Featherweight VeriFast, called Mechanised Featherweight VeriFast, in the machine-readable formal language of the interactive proof assistant Coq. Coq is a computer program that takes as input a set of files containing definitions and proofs expressed in its formal language, and checks that these definitions and proofs are indeed well-formed. Since we have successfully checked our development with Coq, we can have very high confidence that the theorems that we have proven are indeed true, with respect to the given definitions.

Note that it is still possible that Mechanised Featherweight VeriFast contains errors: it might still be the case that the stated definitions and theorems are not the ones that we *intended*; for example, if we made an error in the definition of the concrete execution such that concrete execution always blocks, or we made an error in the definition of the symbolic execution such that symbolic execution always fails, then the soundness theorem holds vacuously and does not really tell us anything meaningful. We partially address this issue by including a small *test suite* in our development, where we run the concrete execution and the symbolic execution on specific example programs, and test that concrete execution does indeed sometimes fail as expected, and that symbolic execution does indeed sometimes succeed as expected. Still, we should remain skeptical, and confidence in the *relevance* of a formally proven statement can never be 100%. It can be improved further by enlarging the test suite and/or by proving additional properties of the various executions, e.g. by relating MFVF's concrete execution to another programming language semantics found in the literature.

While MFVF follows FVF very closely in most respects, there are a few differences, mainly motivated by the fact that we wanted MFVF to be *executable* so as to be able to test it easily, whereas for FVF *simplicity* is more important. Also, MFVF has a few minor additional features, which were left out of FVF, again for the sake of simplicity.

In the remainder of this section, we briefly discuss the main differences between MFVF and FVF and the executability of MFVF, we show the soundness theorem, and we point the reader to the full Coq sources which are available online.

5.1. **Differences between MFVF and FVF: Syntax.**

In Figure 13 we show the syntax of the programming language and the annotations accepted by MFVF. The differences with FVF are shown in red; as the reader can see, they are very minor.

The main difference is that MFVF supports *routine return values*; when executing a routine call $x := r(\overline{e})$, after execution of the routine body ends, the value assigned by the routine body to variable result is assigned to variable x of the caller.

A minor difference is in the syntax of **open** commands: MFVF allows the command to leave some of the chunk arguments unspecified. The command **open** $q(\overline{e}, \overline{?\_})$ opens some chunk that matches the pattern $q(\overline{e}, \overline{?\_})$.

$$
\begin{array}{rl}
 & z \in \mathbb{Z}, n \in \mathbb{N} \\
 & x \in \mathit{Vars} \\
e ::= & z \mid x \mid e + e \\
b ::= & e = e \mid e < e \mid \neg b \\
c ::= & x := e \mid (c; c) \mid \textbf{if } b \textbf{ then } c \textbf{ else } c \mid \textbf{skip} \mid \textbf{message } \mathit{text} \\
 & \mid x := r(\overline{e}) \mid x := \textbf{malloc}(n) \mid x := [e] \mid [e] := e \mid \textbf{free}(e) \\
 & \mid \textbf{while } b \textbf{ inv } a \textbf{ do } c \mid \textbf{open } q(\overline{e}, \overline{?\_}) \mid \textbf{close } q(\overline{e}) \\
\mathit{rdef} ::= & \textbf{routine } r(\overline{x}) = c \\
 & \\
 & q \in \mathit{UserDefinedPredicates} \\
p ::= & \mapsto \mid \mathsf{mb} \mid q \\
a ::= & b \mid p(\overline{e}, \overline{?x}) \mid a * a \mid \textbf{if } b \textbf{ then } a \textbf{ else } a \\
\mathit{preddef} ::= & \textbf{predicate } q(\overline{x}) = a \\
\mathit{rspec} ::= & \textbf{routine } r(\overline{x}) \textbf{ req } a \textbf{ ens } a
\end{array}
$$

Figure 13: Syntax of Mechanised Featherweight VeriFast's input language

Two new commands are added. The **skip** command does nothing; it is equivalent to $x := x$. The command **message** *text* prints message *text* to the console. This command is useful in MFVF for testing the executions.

5.2. **Differences between MFVF and FVF: Executions.** The main difference between the executions (concrete execution, semiconcrete execution, and symbolic execution) of MFVF and those of FVF is in the definition and use of the auxiliary mutators for the consumption of heap chunks ($\mathsf{cconsume\_chunks}(h)$, $\mathsf{consume\_chunks}(h)$, and $\mathsf{sconsume\_chunks}(\hat{h})$ in FVF). In FVF, these mutators take as an argument the precise multiset of heap chunks (up to provable equality for symbolic execution) to be consumed. However, at a typical use site, only part of the argument list of a chunk is fixed, and the remaining arguments are to be looked up in the heap. In FVF, this is achieved by angelically choosing these remaining arguments.

For example, consider symbolic execution of a heap lookup command:

FVF:
$\mathsf{symexec}(x := [e]) =$
$\quad \hat{\ell} \leftarrow \mathsf{seval}(e); \bigoplus \hat{v}.\ \mathsf{sconsume\_chunk}(\hat{\ell} \mapsto \hat{v}); \mathsf{sproduce\_chunk}(\hat{\ell} \mapsto \hat{v}); x := \hat{v}$
$\mathsf{sconsume\_chunk} \in \mathit{SHeaps} \to \mathit{SOutcomes}(\mathsf{unit})$

MFVF:
$\mathsf{symexec}(x := [e]) =$
$\quad \hat{\ell} \leftarrow \mathsf{seval}(e); [\hat{v}] \leftarrow \mathsf{sconsume\_chunk}(\mapsto, [\hat{\ell}], 1); \mathsf{sproduce\_chunk}(\hat{\ell} \mapsto \hat{v}); x := \hat{v}$
$\mathsf{sconsume\_chunk} \in \mathit{SPredicates} \to \mathit{Terms}^* \to \mathbb{N} \to \mathit{SOutcomes}(\mathit{Terms}^*)$

In FVF, symbolic execution of a command of the form $x := [e]$ that reads the memory cell at address e evaluates e to obtain term $\hat{\ell}$, then angelically chooses some term $\hat{v}$, and then attempts to consume the points-to chunk that maps address $\hat{\ell}$ to this angelically chosen term $\hat{v}$. This consumption operation succeeds if a points-to chunk exists in the symbolic heap such that the SMT solver succeeds in proving that its arguments are equal to $\hat{\ell}$ and $\hat{v}$, respectively.

This definition is perfectly fine, except that angelically choosing a term from the set of all terms (that use only symbols that are already being used by the current symbolic state) is not directly executable, since that set is infinite, and even if it was finite, it would be highly inefficient. Therefore, in MFVF, a slightly more complex but directly executable version of the chunk consumption mutators is used. These mutators consume only a single chunk at a time, and they take as arguments the predicate name, the list of fixed chunk arguments, and the number of non-fixed chunk arguments; they return the values of the non-fixed arguments of the chunk that was consumed as their answer. Correspondingly, in MFVF, symbolic execution of $x := [e]$, rather than angelically choosing a term for the value of the cell, retrieves that term as the answer of the sconsume_chunk auxiliary mutator.

5.3. **Executability.** This concludes the discussion of the differences between MFVF and FVF. We now discuss some specific encoding choices made when defining MFVF to obtain *executable* definitions of symbolic execution and concrete execution.

The most important such choice is in the definition of the type of outcomes. The definition of inductive type **outcome** in MFVF is shown below.

```
Inductive type_name := n_Empty_set | n_bool | n_Z | n_T(T : Type).

Fixpoint Itype_name(n : type_name) : Type := match n with
  | n_Empty_set ⇒ Empty_set
  | n_bool ⇒ bool
  | n_Z ⇒ Z
  | n_T T ⇒ T
  end.

Inductive set(X : Type) := set_(n : type_name)(f : Itype_name n → X).

Inductive outcome(S A : Type) :=
| single(s : S)(a : A)
| demonic(os : set (outcome S A))
| angelic(os : set (outcome S A))
| message(msg : string)(o : outcome S A).
```

It corresponds exactly to the definition of outcomes given earlier for FVF (except for the extra case of messages): an outcome ϕ is either a singleton outcome $\langle\sigma, a\rangle$ with output state σ and answer a, or a demonic choice $\bigotimes\Phi$ or angelic choice $\bigoplus\Phi$ over a set of outcomes Φ. However, there are two interesting aspects about this definition, and more specifically, about the type **set** used for the sets of outcomes.

First of all, we had to choose this type carefully to obtain a proper inductive definition. The simplest approach for defining a type for sets of elements of some type X is as follows:

```
Definition set X := X → Prop.
```

That is, a set of elements of type X is simply a predicate over type X. However, using this definition of sets in the definition of outcomes would cause Coq to reject the definition of outcomes, since it would not be a proper inductive definition. Indeed, it would allow us to write demonic ($\lambda\_.$ True), denoting the demonic choice over *all* outcomes, including demonic ($\lambda_.$True) itself, defeating the crucial notion that each value of an inductive type is built from *smaller* values of that type, and thus rendering proof by induction unsound.

Perhaps the simplest possible definition for a type of sets of elements of type X that is compatible with inductive definitions is the following:

$$\texttt{Inductive}\ \textbf{set}(X : \texttt{Type}) := \mathsf{set\_}(I : \texttt{Type})(f : I \to X).$$

This type allows a set to be constructed by providing an *index type* I and a function f that maps each value of type I to some value of type X. For example, the set containing exactly the integers 24 and 42 can be constructed as follows:

$$\mathsf{set\_}\ \textbf{bool}\ (\lambda b.\ \mathsf{if}\ b\ \mathsf{then}\ 24\ \mathsf{else}\ 42)$$

Using this type of sets in the definition of outcomes would be accepted by Coq.

However, another problem would still remain: we would like to write a Coq function that takes the outcome of symbolically executing some program starting from the empty symbolic state, and decides if that outcome satisfies postcondition True, i.e., if symbolic execution has failed or not. An outcome satisfies postcondition True iff the outcome is a singleton outcome, or it is a demonic choice over some set of outcomes, each of which satisfies postcondition True, or it is an angelic choice over some set of outcomes, at least one of which satisfies postcondition True (or it is a message outcome and its continuation satisfies postcondition True). So, for demonic and angelic choice over some set of outcomes, we need to be able to enumerate the elements of the set. Given the definition of sets above, it would be necessary to enumerate the elements of the index type I. Unfortunately, however, this is not generally possible: the index type might be infinite.

However, MFVF's definition of symbolic execution uses only very restricted forms of demonic or angelic choice: it only uses blocking, failure, and binary choice, i.e., choices over zero elements or two elements. So, if in symbolic execution we use as index types only the type **Empty_set** and the type **bool**, can we write our Coq function? Unfortunately, still no, because this would require our function to perform a case analysis on a comparison between the index type of a set and the types **Empty_set** or **bool**, and Coq's execution engine does not support this. Coq's execution engine supports only pattern matching on values of inductive types, and types themselves are not values of inductive types.

The solution we adopted for this problem is to not directly allow arbitrary types to be specified as the index type when constructing a set, but rather to define an inductive type **type_name** of *type names*, with names for type **Empty_set**, for type **bool**, and for the type **Z** of integers, and a fallback case n_T for arbitrary types. We also defined an interpretation function Itype_name for these type names that maps each type name to its corresponding type. By using type names and the interpretation function in the definition of sets, we were able to write a Coq function that decides whether an outcome satisfies postcondition True. (For the cases n_Z and n_T this function is not executable, but since symbolic execution uses only the other two cases, it executes properly for the outcomes of symbolic execution.)

Figure 14 shows an example where we run symbolic execution to check validity of a routine that performs in-place reversal of a linked list. Function svalid_routine (the syntax of the example is slightly simplified from the actual Coq development) takes as arguments a list of predicate definitions (in the example, just the definition listDef of the list predicate), a list of routine specifications (in the example, an empty list, since the list reversal routine

```
Definition listDef :=
  predicate list(l) =
    if l = 0 then 0 = 0 else mb(l, 2) ∗ l ↦ _ ∗ l + 1 ↦ ?next ∗ list(next).

Compute svalid_routine [listDef] [] [l] list(l) list(result)
  (
    close list(b);
    while ¬(a = 0) inv list(a) ∗ list(b) do (
      open list(a);
      n := [a + 1]; [a + 1] := b; b := a; a := t;
      close list(b)
    );
    open list(a);
    result := b
  ).
ok
```

Figure 14: Running MFVF on an in-place list reversal routine

does not itself perform any routine calls), a list of parameters (in the example, just a single parameter l, a pointer to the linked list to be reversed), a precondition (in the example, list(l), expressing that the routine expects to find a linked list at address l), a postcondition (in the example, list(result), expressing that after the routine completes, the routine's result will point to a linked list), and the body of the routine to be verified. Coq command `Compute` evaluates a Coq expression and prints the result: in the example, the result is *ok*, indicating that the routine was verified successfully.

MFVF includes an executable definition of symbolic execution and a *semi-executable* definition of concrete execution. Concrete execution is *semi-executable* in the sense that we have been able to write Coq functions that compute, for a given input program and a given sequence of values for demonic choices over the booleans or the integers, if concrete execution, for those choices, ends up in a singleton outcome or in failure.

For example, consider the program that allocates a memory cell and then accesses the memory cell at address 42. If the newly allocated memory cell was allocated at address 42, execution succeeds; otherwise, it fails.

We can easily check that both execution paths do indeed behave as expected, using the Coq functions atZ, isSingle, and isFail shown in Figure 15. As shown in the figure, using Coq's `Compute` command, and by first picking value 2 for the depth of concrete execution (any greater value would do as well) and then 42 for the address of the newly allocated memory block, we can confirm that we end up in a singleton outcome, and that by alternatively picking address 43 we end up in failure.

The definition of function atZ exploits the fact that there is a separate case for type **Z** in type **type_name**, and that concrete execution of **malloc** commands uses this type name in its demonic choice.

```
Definition atZ $z$ $o$ := match $o$ with
  | Some (demonic (set_ n_Z $o'$)) ⇒ Some ($o'$ $z$)
  | _ ⇒ None end.
Definition isSingle $o$ :=
  match $o$ with Some (single _ _) ⇒ true | _ ⇒ false end.
Definition isFail $o$ := match $o$ with
  | Some (angelic (set_ n_Empty_set _)) ⇒ true
  | _ ⇒ false end.

Definition o := cstate0 ▷ exec [] (x := malloc(1); [42] := 123).

Compute Some $o$ ▷ atZ 2 ▷ atZ 42 ▷ isSingle.
true
Compute Some $o$ ▷ atZ 2 ▷ atZ 43 ▷ isFail.
true
```

Figure 15: Testing MFVF concrete execution

5.4. **Soundness.** The Coq statement of the soundness theorem is shown below: if symbolic execution of a program does not fail, then concrete execution of that program does not fail. The proof is accepted by Coq.

```
Theorem soundness rspecs pdefs rdefs c :
  svalid_program rspecs pdefs rdefs c = ok →
  cvalid_program rdefs c.
Proof.
...
Qed.

Print Assumptions soundness.
Coq.Sets.Ensembles.Extensionality_Ensembles
Coq.Logic.Classical_Prop.classic
Coq.Logic.IndefiniteDescription.constructive_indefinite_description
Coq.Logic.FunctionalExtensionality.functional_extensionality_dep
```

We can use Coq's `Print Assumptions` command to check which axioms are used (directly or indirectly) in the proof of the soundness theorem. Only the four listed axioms are used: they are axioms of classical logic, offered by the Coq standard library.

The Coq development can be browsed in HTML and PDF form and the full sources can be downloaded at http://www.cs.kuleuven.be/~bartj/fvf/.

6. Related work

6.1. **Hoare logic, separation logic.** A more abstract, higher-level approach for reasoning about imperative pointer-manipulating programs is given by separation logic [39, 42, 38], which is an extension of Hoare logic [24].

$$\textsc{Assign}\quad \{b[e/x]\}\ x := e\ \{b\}$$

$$\textsc{If}\quad \frac{\{b \wedge b'\}\ c\ \{b''\} \qquad \{b \wedge \neg b'\}\ c'\ \{b''\}}{\{b\}\ \mathbf{if}\ b'\ \mathbf{then}\ c\ \mathbf{else}\ c'\ \{b''\}}$$

$$\textsc{While}\quad \frac{\{b \wedge b'\}\ c\ \{b\}}{\{b\}\ \mathbf{while}\ b'\ \mathbf{do}\ c\ \{b \wedge \neg b'\}}$$

$$\textsc{Seq}\quad \frac{\{b\}\ c\ \{b'\} \qquad \{b'\}\ c'\ \{b''\}}{\{b\}\ c; c'\ \{b''\}}$$

$$\textsc{Conseq}\quad \frac{b \Rightarrow b' \qquad \{b'\}\ c\ \{b''\} \qquad b'' \Rightarrow b'''}{\{b\}\ c\ \{b'''\}}$$

$$\textsc{Exists}\quad \frac{\forall v.\ \{b[v/x]\}\ c\ \{b'[v/x]\}}{\{\exists x.\ b\}\ c\ \{\exists x.\ b'\}}$$

Figure 16: The main axioms and inference rules of Hoare logic

Hoare logic deals with *program correctness judgments* (also known as *Hoare triples*) of the form $\{b\}\ c\ \{b'\}$, where b, the *precondition*, and b', the *postcondition*, are boolean expressions (as in Definition 1.1, except that they may also contain additional logical operators such as conjunction and quantification), and c is a command that does not involve the heap (i.e., it does not allocate, deallocate, or access heap cells); the judgment means that c, when started with a store that satisfies precondition b, if it terminates, terminates with a store that satisfies postcondition b':

$$\forall s.\ [\![b]\!]_s = \mathsf{true} \Rightarrow s \triangleright \mathsf{exec}(c)\ \{s'.\ [\![b']\!]_{s'} = \mathsf{true}\}$$

Hoare logic defines a number of *axioms* and *inference rules* for deriving correctness judgments; the main ones are shown in Figure 16. Here, $b[e/x]$ denotes the boolean expression obtained by substituting expression e for variable x in b, and $b \Rightarrow b'$ denotes that b implies b' in all stores, *i.e.* $\forall s.\ [\![b]\!]_s \Rightarrow [\![b']\!]_s$.

For example, we can derive the judgment $\{0 \leq n\}\ i := 0; \mathbf{while}\ i < n\ \mathbf{do}\ i := i+1\ \{i = n\}$ using the proof tree in Figure 17.

A more convenient representation of this proof tree is in the form of the *proof outline* of Figure 18, where assertions inserted between components of a sequential composition indicate applications of the Seq rule, and multiple consecutive assertions indicate applications of the Conseq rule.

Separation logic extends Hoare logic with additional assertion logic constructs and proof rules for reasoning conveniently about heap-manipulating programs. The syntax of separation logic assertions extends the syntax of logical formulae with constructs for specifying the heap: the assertion **emp** states that the heap is empty; the points-to assertion $e \mapsto e'$ states that the heap consists of exactly one heap cell, mapping address e to value e', and the separating conjunction $P * Q$ states that the heap can be split into two disjoint parts such that P holds for one part and Q holds for the other. Instead of Featherweight VeriFast's $?x$ syntax, separation logic uses regular existential quantification. Formally:

$$\begin{array}{ll}
s, h \vDash \mathbf{emp} & \Leftrightarrow h = \emptyset \\
s, h \vDash e \mapsto e' & \Leftrightarrow h = \{([\![e]\!]_s, [\![e']\!]_s)\} \\
s, h \vDash P * Q & \Leftrightarrow \exists h_1, h_2.\ h = h_1 \uplus h_2 \wedge s, h_1 \vDash P \wedge s, h_2 \vDash Q \\
s, h \vDash \exists x.\ P & \Leftrightarrow \exists v.\ s[x := v], h \vDash P \\
s, h \vDash b & \Leftrightarrow [\![b]\!]_s = \mathsf{true}
\end{array}$$

$$\dfrac{\dfrac{}{(b)}\textsc{Assign} \qquad \dfrac{(d) \qquad \dfrac{\dfrac{(h) \quad \dfrac{}{(i)}\textsc{Assign} \quad (j)}{(g)}\textsc{Conseq}}{(e)}\textsc{While} \qquad (f)}{(c)}\textsc{Conseq}}{(a)}\textsc{Seq}$$

$$\begin{array}{ll}
(a) & \{0 \leq n\}\ i := 0; \textbf{while}\ i < n\ \textbf{do}\ i := i + 1\ \{i = n\} \\
(b) & \{0 \leq n\}\ i := 0\ \{i \leq n\} \\
(c) & \{i \leq n\}\ \textbf{while}\ i < n\ \textbf{do}\ i := i + 1\ \{i = n\} \\
(d) & i \leq n \Rightarrow i \leq n \\
(e) & \{i \leq n\}\ \textbf{while}\ i < n\ \textbf{do}\ i := i + 1\ \{i \leq n \land \neg(i < n)\} \\
(f) & i \leq n \land \neg(i < n) \Rightarrow i = n \\
(g) & \{i \leq n \land i < n\}\ i := i + 1\ \{i \leq n\} \\
(h) & i \leq n \land i < n \Rightarrow i + 1 \leq n \\
(i) & \{i + 1 \leq n\}\ i := i + 1\ \{i \leq n\} \\
(j) & i \leq n \Rightarrow i \leq n
\end{array}$$

Figure 17: Proof tree in Hoare logic for a simple example program

$$\begin{array}{l}
\{0 \leq n\} \\
i := 0; \\
\{i \leq n\} \\
\textbf{while}\ i < n\ \textbf{do} \\
\quad \{i \leq n \land i < n\} \\
\quad \{i + 1 \leq n\} \\
\quad i := i + 1 \\
\quad \{i \leq n\} \\
\{i \leq n \land \neg(i < n)\} \\
\{i \leq n\}
\end{array}$$

Figure 18: Proof outline in Hoare logic for a simple example program

In separation logic, predicates are typically treated like inductive definitions, i.e. their meaning is taken to be the smallest interpretation (i.e. set of heaps) that satisfies the definition; such an interpretation always exists (by the Knaster-Tarski theorem) provided that predicates are used inside of predicate definitions only in positive positions, i.e. not under negations or on the left-hand side of implications [40]. The typical example of such a predicate is the predicate $\mathsf{lseg}(\ell, \ell')$ denoting a linked list segment from a starting node ℓ (inclusive) to a limiting node ℓ' (exclusive):

$$\mathsf{lseg}(\ell, \ell') \stackrel{\text{def}}{=} \ell = \ell' \land \mathbf{emp} \lor \exists v, n.\ \ell \mapsto v * \ell + 1 \mapsto n * \mathsf{lseg}(n, \ell')$$

Separation logic's program logic extends Hoare logic's inference system with axioms for the heap manipulation commands and the *frame axiom* (see Figure 19). The former are *small axioms*: they mention only the heap cells required for the command to succeed. The frame axiom allows the small axioms to be lifted to larger heaps. Similarly, one can write small

$$\textsc{Cons} \quad \{\mathbf{emp}\}\ x := \mathbf{cons}(v, v')\ \{x \mapsto v * x + 1 \mapsto v'\} \qquad \textsc{Dispose} \quad \{\ell \mapsto v\}\ \mathbf{dispose}(\ell)\ \{\mathbf{emp}\}$$

$$\textsc{Read} \quad \{\ell \mapsto v\}\ x := [\ell]\ \{\ell \mapsto v \land x = v\} \qquad \textsc{Write} \quad \{\ell \mapsto v\}\ [\ell] := v'\ \{\ell \mapsto v'\}$$

$$\textsc{Frame} \quad \frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}\ \text{if } \mathsf{targets}(c) \cap \mathsf{freevars}(R) = \emptyset$$

Figure 19: The additional proof rules of separation logic

```
{$\mathsf{lseg}(i, 0)$}
$j := 0$;
{$\mathsf{lseg}(i, 0) * \mathsf{lseg}(j, 0)$}
while $i \neq 0$ do (
  {$\mathsf{lseg}(i, 0) * \mathsf{lseg}(j, 0) \land i \neq 0$}
  {$\exists v, n.\ i + 1 \mapsto n * i \mapsto v * \mathsf{lseg}(n, 0) * \mathsf{lseg}(j, 0)$}
    {$i + 1 \mapsto n * i \mapsto v * \mathsf{lseg}(n, 0) * \mathsf{lseg}(j, 0)$}  Rule EXISTS. Fix $v, n$.
      {$i + 1 \mapsto n$}  Rule FRAME.
      $k := [i + 1]$;
      {$i + 1 \mapsto n \land k = n$}
    {$(i + 1 \mapsto n \land k = n) * i \mapsto v * \mathsf{lseg}(n, 0) * \mathsf{lseg}(j, 0)$}
    {$i + 1 \mapsto k * i \mapsto v * \mathsf{lseg}(k, 0) * \mathsf{lseg}(j, 0)$}
      {$i + 1 \mapsto k$}  Rule FRAME.
      $[i + 1] := j$;
      {$i + 1 \mapsto j$}
    {$i + 1 \mapsto j * i \mapsto v * \mathsf{lseg}(k, 0) * \mathsf{lseg}(j, 0)$}
    {$\mathsf{lseg}(k, 0) * \mathsf{lseg}(i, 0)$}
    $j := i$;
    {$\mathsf{lseg}(k, 0) * \mathsf{lseg}(j, 0)$}
    $i := k$
    {$\mathsf{lseg}(i, 0) * \mathsf{lseg}(j, 0)$}
  {$\mathsf{lseg}(i, 0) * \mathsf{lseg}(j, 0)$}
)
{$\mathsf{lseg}(i, 0) * \mathsf{lseg}(j, 0) \land i = 0$}
{$\mathsf{lseg}(j, 0)$}
```

Figure 20: A proof outline in separation logic of a program that performs an in-place reversal of a linked list

specifications for routines and use the frame axiom to lift those to the larger heap present in a given calling context.

Figure 20 shows a proof outline in separation logic for a program that performs an in-place reversal of a linked list.

VeriFast could be considered to be a type of "separation logic theorem prover", by interpreting the input files as a separation logic Hoare triple that serves as the proof goal and the annotations as hints to direct the construction of the proof. From this point of view, VeriFast applies the separation logic frame rule when verifying loops and routine calls.

6.2. **Separation logic tools.**

6.2.1. *Smallfoot.* Smallfoot [9] was a breakthrough in program verification tool development; it was successful in its goal of showcasing for the first time the power of separation logic for automated program verification and analysis. Like FVF, it takes as input an annotated program and checks each procedure against its contract. The programming language is very similar: like FVF's, it is a simple while language with procedures. The main difference is that it includes concurrency constructs (resource declarations, parallel procedure calls, and conditional critical regions). The annotation language is very similar as well: a precondition and postcondition must be specified for each procedure, and a loop invariant must be specified for each loop; these are not inferred. (If one of these is omitted, it defaults to **emp**.) The main difference is that besides the points-to assertion, Smallfoot has built-in predicates for trees, list segments, doubly-linked lists, and xor lists, does not support user-defined predicates, and does not require **open** or **close** commands or any other kinds of proof hints (other than the procedure and loop annotations mentioned above). Another difference is that it does not support (even FVF's very restricted form of) existential quantification.

The main difference in Smallfoot's functional behavior is that it is automatic: thanks to a complete, decidable proof theory for the supported assertion language, Smallfoot never requires proof hints. In particular, not only does it automatically fold and unfold the definitions of the inductive predicates (which in FVF requires **open** and **close** ghost commands), it also has sufficient rules built in to reason automatically about inductive properties such as appending two list segments. In FVF, this would require defining and calling a recursive "lemma" routine that establishes the property.

While Smallfoot's algorithm is in many ways more powerful and more interesting than FVF's, FVF's goal is educational, and we believe its presentation in this article succeeds better at clearly conveying the essence of VeriFast's operation, especially to an audience that is new to formal methods, than the presentation of Smallfoot's operation [9, 8] does.

6.2.2. *Other tools.* Smallfoot's algorithm has been used as a basis for *shape analysis* algorithms that automatically infer loop invariants and postconditions [21], and even preconditions [13]. These algorithms have been implemented in a tool called Infer [12] that has successfully been exploited commercially. Another tool based on these ideas, called SLAyer [10], is being used inside Microsoft to verify Windows device drivers.

These techniques have been extended to a concurrent setting, e.g. to infer invariants for shared resources [14]. Integration of separation logic and rely-guarantee reasoning [31] has led to tools SmallfootRG [15] for verifying safety properties and Cave [47] for verifying linearizability of fine-grained concurrent modules.

Extensions of separation logic for dealing with object-oriented programming patterns such as dynamic binding have been implemented in the tool jStar [22] that takes as input a Java program, a precondition and postcondition for each method, and a set of inference and abstraction rules, and attempts to automatically apply these rules to verify each method body against its specification. jStar does not require (or support) annotations inside method bodies.

The HIP/SLEEK toolstack [16] uses separation logic-based symbolic execution to automatically verify shape, size, and bag properties of programs. Like VeriFast, it supports user-defined recursive predicates to express the shape of data structures.

6.2.3. *Proof assistant-based approaches.* Like VeriFast, the tools mentioned above take as input annotated programs and then run without further user interaction. Another approach is to see program verification as a special case of interactive proof development, and to extend proof assistants like Isabelle/HOL and Coq with theories defining program syntax and semantics and specification formalisms, as well as lemmas and tactics (reusable proof scripts) for aiding users in discharging proof obligations.

Holfoot [46] is an implementation of Smallfoot inside the HOL 4 theorem prover. In addition to the features supported by Smallfoot it can handle data and supports interactive proofs. Moreover, it can handle arrays. Simple specifications with data like copying a list can be handled automatically. More complicated ones like fully functional specifications of filtering a list, mergesort, quicksort or an implementation of red-black trees require user interaction. During this interaction all the features of the HOL 4 theorem prover can be used, including the interface to external SMT solvers like Yices.

Ynot [18] is a library for the Coq proof assistant which turns it into a full-fledged environment for writing and verifying imperative programs. In the tradition of the Haskell IO monad, Ynot axiomatizes a parameterized monad of imperative computations, where the type of a computation specifies not only what type of data it returns, but also what Hoare-logic-style precondition and postcondition it satisfies. On top of the simple axiomatic base, the library defines a separation logic. Specialized automation tactics are able to discharge automatically most proof goals about separation-style formulas that describe heaps, meaning that building a certified Ynot program is often not much harder than writing that program in Haskell.

Bedrock [17] is a Coq library for mostly-automated verification of low-level programs in computational separation logic; a major difference from Ynot is that it has improved support for reasoning about code pointers.

Charge! [7] is a set of tactics for working with a shallow embedding of a higher-order separation logic for a subset of Java in Coq.

The Verified Software Toolchain project [5, 6, 3] has produced a separation logic for C, called Verified C, in the form of a Coq library, as well as a Smallfoot implementation in Coq, extractable to OCaml, called VeriSmall [4], both proven sound in Coq with respect to the operational semantics of C against with the CompCert project [37] verified the correctness of their C compiler, thus obtaining that the compiled program satisfies the verified properties.

6.3. **Non-separation logic tools.** Another approach for extending Hoare logic to reason about programs with pointers (or other kinds of aliasing, such as Java's object references) is to simply treat the heap as a program variable whose value is a function that maps addresses to values, and to retain regular classical logic as the assertion language. The following tools are based on this approach.

In this approach, the separation logic frame rule and small axioms that allow a simple syntactic treatment of heap mutation and procedure effect framing are generally not available, but other approaches to procedure effect framing may be used. Most alternative approaches are variants of *dynamic frames* [32], where a module uses abstract variables of type "set of memory locations" to abstractly specify which memory locations are modified by a procedure as well as which memory locations may influence the value of abstract variables.

VCC [19] is a verifier for concurrent C programs annotated with contracts expressed in classical logic. For each C function, VCC generates a set of verification conditions (using a variant of *weakest preconditions* [20]) to be discharged by an SMT solver. For modularity, it uses the *admissible invariants* approach: a *two-state invariant* may be associated with each C struct instance s, which may mention the fields of s as well as those of other struct instances s', provided it is *admissible*: any update of $s'.f$ that satisfies the invariant of s' must preserve the invariant of s. By encoding an ownership system on top of this approach, it can be used both for precise reasoning about fine-grained concurrency and for reasoning in a dynamic frames-like style about sequential code. VCC has been used to verify a large part of the Microsoft Hyper-V hypervisor.

Other important non-separation logic tools include Chalice [36] (a verifier for concurrent Java-like programs based on *implicit dynamic frames* [43]), Dafny [35], KeY [1], and KIV [45].

As in the case of separation logic-based approaches, some non-separation logic-based verification efforts have been carried out in a general-purpose proof assistant rather than a specialized tool. Notable in this category are the L4.verified project [34], which verified an OS microkernel consisting of 8KLOC of C code in Isabelle/HOL, and the Verisoft project [2], which performed large parts of the pervasive verification, also in Isabelle/HOL, of the complete software stack (plus parts of the hardware), including microkernel, kernel, and applications, of a secure e-mail system and an embedded automotive system.

6.4. **Semantic framework: Outcomes.** In our formalization, to express and relate the semantics of the programming language and the verification algorithm, via the intermediary of semiconcrete execution, we developed the semantic framework based on *outcomes*, with the important derived concepts of *mutators*, *postcondition satisfaction*, and *coverage*. This enabled us to deal conveniently with failure, nontermination, and both demonic and angelic nondeterminism.

This framework is essentially nothing more than the predicate transformer semantics proposed by Dijkstra [20]:

$$\mathsf{exec}(c)\ \{Q\} \equiv \mathsf{wp}(c, Q)$$

Also, mutators with answers are essentially a combination of a state monad and a continuation monad.

Our choice of defining the set of outcomes as an inductive datatype, rather than a predicate over postconditions (i.e. a function from postconditions to bool, such that mutators would be predicate transformers or functions from predicates to predicates) or, equivalently, a state-continuation monad, has two advantages: firstly, we immediately have that all outcomes are *monotonic* (postcondition satisfaction is preserved by weakening of the postcondition); and secondly, our Coq encoding of concrete execution yields not an unexecutable function to bool but an (infinite-branching) execution tree which we can explore, as shown in Section 5.

6.5. Machine-checked tools.

An effort similar to our executable machine-checked encoding into Coq of Featherweight VeriFast is the executable machine-checked encoding into Coq of Smallfoot, called VeriSmall [4].

Whereas VeriSmall's primary purpose is to serve as the basis for a certified program verification tool chain, MFVF's primary purpose is to serve as evidence for the correctness of the presentation of FVF and its soundness proof in this article. Therefore, MFVF mirrors the presentation very closely, and is more optimized for reading than VeriSmall.

7. Conclusion

We presented a formal definition and outlined a soundness proof of Featherweight VeriFast, thus hopefully achieving a clear and precise exposition of a core subset of the VeriFast approach for sound modular verification of imperative programs. We also described our executable definition and machine-checked soundness proof of Mechanised Featherweight VeriFast, a slight variant of Featherweight VeriFast, in the Coq proof system.

Future work includes: extending Featherweight VeriFast to include additional features of VeriFast, such as lemma functions, inductive datatypes and fixpoint functions, concurrency, fractional permissions, function pointers, lemma function pointers, predicate families, and higher-order predicates[9]; extending the executable definition of Mechanised Featherweight VeriFast so that it can be used as a higher-assurance drop-in replacement for VeriFast to verify annotated C source code files; and linking the resulting tool to existing formalisations of C semantics, such as CompCert [37].

Acknowledgements. We would like to thank the anonymous reviewers for their helpful comments. This work was supported in part by the Research Fund KU Leuven and by EU project ADVENT.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. Key: A formal method for object-oriented systems. In Marcello M. Bonsangue and Einar Broch Johnsen, editors, *FMOODS*, volume 4468 of *Lecture Notes in Computer Science*, pages 32–43. Springer, 2007.

[2] Eyad Alkassar, Wolfgang Paul, Artem Starostin, and Alexandra Tsyban. Pervasive verification of an OS microkernel: Inline assembly, memory consumption, concurrent devices. In Peter O'Hearn, Gary T. Leavens, and Sriram Rajamani, editors, *Verified Software: Theories, Tools, Experiments (VSTTE 2010)*, volume 6217 of *Lecture Notes in Computer Science*, pages 71–85, Edinburgh, UK, August 2010. Springer.

[3] Andrew W. Appel. Verified software toolchain. In *ESOP*, 2011.

[4] Andrew W. Appel. VeriSmall: Verified Smallfoot shape analysis. In *CPP*, 2011.

[5] Andrew W. Appel. Verification of a cryptographic primitive: Sha-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7:1–7:31, April 2015.

[6] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA, 2014.

[7] Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! a framework for higher-order separation logic in Coq. In *ITP*, 2012.

[8] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.

[9] Note that these advanced features have already been formalized, with machine-checked soundness proofs, separately [29, 26].

[9] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2006.

[10] Josh Berdine, Byron Cook, and Samin Ishtiaq. SLAyer: Memory safety for systems-level code. In *CAV*, 2011.

[11] Michael Butler and Wolfram Schulte, editors. *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *Lecture Notes in Computer Science*. Springer, 2011.

[12] Cristiano Calcagno and Dino Distefano. Infer: an automatic program verifier for memory safety of C programs. In *Proc. 3rd NASA Formal Methods Symposium*, number 6671 in Lecture Notes in Computer Science. Springer, 2011.

[13] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 289–300. ACM, 2009.

[14] Cristiano Calcagno, Dino Distefano, and Viktor Vafeiadis. Bi-abductive resource invariant synthesis. In *APLAS*, number 5904 in LNCS, pages 259–274. Springer, 2009.

[15] Cristiano Calcagno, Matthew Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS*, number 4634 in LNCS, pages 233–238. Springer, August 2007.

[16] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.

[17] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In Mary W. Hall and David A. Padua, editors, *PLDI*, pages 234–245. ACM, 2011.

[18] Adam Chlipala, J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In Graham Hutton and Andrew P. Tolmach, editors, *ICFP*, pages 79–90. ACM, 2009.

[19] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.

[20] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18(8):453–457, August 1975.

[21] Dino Distefano, Peter O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.

[22] Dino Distefano and Matthew Parkinson. jStar: Towards practical verification for java. In *OOPSLA*, 2008.

[23] Jean-Christophe Filliâtre, Andrei Paskevich, and Aaron Stump. The 2nd verified software competition: Experience report. In Vladimir Klebanov and Sarah Grebing, editors, *COMPARE2012: 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems*, Manchester, UK, June 2012. EasyChair.

[24] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[25] Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan. VerifyThis Verification Competition 2012 - organizer's report. Technical Report Karlsruhe Reports in Informatics 2013, 1, Karlsruhe Institute of Technology, Faculty of Informatics, 2013.

[26] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 271–282. ACM, 2011.

[27] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. Invited paper. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

[28] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In *APLAS 2010*, volume 6461 of *LNCS*, pages 304–311, Heidelberg, 2010. Springer.

[29] Bart Jacobs, Jan Smans, and Frank Piessens. Verification of unloadable modules. In Butler and Schulte [11], pages 402–416.

[30] Bart Jacobs, Jan Smans, and Frank Piessens. The VeriFast program verifier: a tutorial. Available from `http://www.cs.kuleuven.be/~bartj/verifast/`, 2014.

[31] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[32] Ioannis T. Kassios. Dynamic frames: support for framing, dependencies and sharing without restrictions. In *FM*, 2006.

[33] Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark A. Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st verified software competition: Experience report. In Butler and Schulte [11], pages 154–168.

[34] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

[35] K. Rustan M. Leino. Developing verified programs with dafny. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *VSTTE*, volume 7152 of *Lecture Notes in Computer Science*, page 82. Springer, 2012.

[36] K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In *ESOP*, 2009.

[37] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[38] Peter W. O'Hearn. A primer on separation logic (and automatic program verification and analysis). In Tobias Nipkow, Orna Grumberg, and Benedikt 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.

[39] Peter W. O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.

[40] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL 2005*, 2005.

[41] Pieter Philippaerts, Jan Tobias Mühlberg, Willem Penninckx, Jan Smans, Bart Jacobs, and Frank Piessens. Software verification with VeriFast: Industrial case studies. *Science of Computer Programming*, 2013.

[42] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, 2002.

[43] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2, 2012.

[44] Jan Smans, Bart Jacobs, and Frank Piessens. Verifast for java: A tutorial. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming*, volume 7850 of *Lecture Notes in Computer Science*, pages 407–442. Springer, 2013.

[45] B. Tofan, G. Schellhorn, and W. Reif. A compositional proof method for linearizability applied to a wait-free multiset. In *iFM*, 2014.

[46] Thomas Tuerk. A separation logic framework for HOL. Technical Report UCAM-CL-TR-799, University of Cambridge, Computer Laboratory, June 2011.

[47] Viktor Vafeiadis. Automatically proving linearizability. In *CAV*, 2010.

[48] Gijs Vanspauwen and Bart Jacobs. Sound symbolic linking in the presence of preprocessing. In Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti, editors, *SEFM*, volume 8137 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 2013.

[49] Frédéric Vogels, Bart Jacobs, Frank Piessens, and Jan Smans. Annotation inference for separation logic based verifiers. In Roberto Bruni and Jürgen Dingel, editors, *FMOODS/FORTE*, volume 6722 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2011.

Notes

[a]Results of the VeriFast team:

Competition	Conference	Result
1st Verified Software Competition [33]	VSTTE 2010	roughly tied with all other teams
2nd Verified Software Competition [23]	VSTTE 2012	score 570/600, rank 8
VerifyThis [25]	FM 2012	sole winner

[b]by Jesper Bengtson (at ITU Copenhagen), Alexey Gotsman (at ENS Lyon), Dilian Gurov (at KTH Stockholm), and Stephan van Staden (at ETH Zurich)