Concurrent Featherweight VeriFast Formalization and Soundness Proof

Bart Jacobs
imec-DistriNet, Dept. of Computer Science, KU Leuven, Belgium

March 8, 2017

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; the new 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 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, 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 that no execution of the program will a) perform an illegal memory access such as a null pointer dereference, accessing unallocated memory, or accessing 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.

In [10], 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.

The programming language accepted by Featherweight VeriFast is *sequential*, i.e. it precisely prescribes a total order on the execution steps of a program. However, VeriFast also supports *concurrent* programs, that prescribe only a *partial* order on the execution steps. In these lecture notes, we present Concurrent Featherweight VeriFast, which extends Featherweight VeriFast with support for thread creation, concurrent reading of memory locations, and mutexes. We out-

line a proof of the soundness of the extended approach, i.e. that if verification of a program succeeds, then no concurrent execution of the program accesses unallocated memory or performs a *data race*, where one thread mutates a memory location, and another thread concurrently reads or mutates the same memory location.

As is the case with Featherweight VeriFast, the verification approach of Concurrent Featherweight VeriFast is to *execute* the program *symbolically*. Again, as an intermediate step between concrete execution and symbolic execution, we define *semiconcrete execution*. Compared to concrete execution, semiconcrete execution in Concurrent Featherweight VeriFast eliminates not just nested routine executions and unbounded numbers of loop iterations, but also concurrency. It does so by distributing *ownership* of allocated memory locations among the threads, and by checking that each thread accesses only the memory locations that it owns. This ownership discipline rules out data races, and furthermore it allows each routine to be verified separately, without having to consider which code other threads are executing concurrently. Sharing of memory locations among threads is achieved by allowing mutexes, too, to own memory locations, at least while no thread holds them. When a thread acquires a mutex, ownership of the memory locations owned by that mutex is transfered to the thread; when the thread releases the mutex, ownership of certain memory locations (as described by the mutex's *mutex invariant*) is again transfered from the thread to the mutex.

The step between semiconcrete execution and symbolic execution in Concurrent Featherweight VeriFast is exactly the same as in Featherweight VeriFast: literal values in execution states are replaced by *terms*, demonic choice over the integers is replaced by picking a free *symbol*, a *path condition* is introduced to constrain the interpretation of symbols, and an *SMT solver* is used to make inferences from the path condition. Therefore, we do not further discuss symbolic execution in this document.

Our running example is a producer-consumer program, where the producer threads allocate memory blocks of size zero and insert their address into a shared buffer of size one protected by a mutex; the consumer threads remove these elements from the shared buffer and free the blocks.

The structure of this document is as follows. In Section 1, we define the extended programming language and present the example program. In Section 2, we define concrete execution of programs. In particular, we define a *small-step* semantics, based on *machine configurations* and *machine steps*. In Section 3, we define (Section 3.1) and prove the soundness (Section 3.2) of semiconcrete execution for Concurrent Featherweight VeriFast programs. We provide pointers for further reading in Section 4.

Contents

Contents

1 The Programming Language

In this section, we define the syntax of the programs accepted by Concurrent Featherweight VeriFast, and we discuss an example program.

The Programming Language

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

The programming language of Concurrent Featherweight VeriFast extends the programming language of Featherweight VeriFast with three new commands: the command **fork**(c), which creates a new thread that will execute command c; and the commands **acquire**(e) and **release**(e), which acquire, resp. release the mutex at the address given by expression e. In this programming language, a mutex is simply an allocated memory location whose value is either 0 or 1; value 1 indicates that some thread is holding the mutex, whereas value 0 indicates that no thread is holding the mutex. Command **acquire**(e) blocks the current thread until the memory location at the address given by e has value 0, and then atomically sets it to 1. Command **release**(e) sets the memory location at the given address to zero.

Note: the difference between **release**(e) and $[e] := 0$ (heap mutation) is that these two commands are treated differently in the definition of what constitutes a *data race*: a race between a mutex acquisition command and a heap mutation command is considered a data race, whereas a race between a mutex acquisition

command and a mutex release command is not. Also, executing **release**(e) while the value at e is not 1 is considered a failure. This will be discussed in more depth in the next section.

Example Program

```
routine produce(b) =
  while 0 = 0 do (
    x := malloc(0);
    while ¬(x = 0) do (
      acquire(b);
      e := [b + 1];
      if e = 0 then (
        [b + 1] := x; x := 0
      ) else x := x;
      release(b)
    )
  )

routine consume(b) =
  while 0 = 0 do (
    x := 0;
    while x = 0 do (
      acquire(b);
      x := [b + 1];
      [b + 1] := 0;
      release(b)
    );
    free(x)
  )

b := malloc(2); [b + 1] := 0; [b] := 0;
fork(produce(b)); fork(consume(b));
fork(produce(b)); fork(consume(b))
```

The example program above is a simple instance of the producer-consumer pattern: a producer thread continuously allocates a new zero-size memory block and attempts to insert its address into a one-place shared buffer, while a consumer thread continuously attempts to remove an element from the buffer and free it. The shared buffer is implemented as a memory block of size 2, where the second cell stores the element contained by the buffer, or zero if the buffer is empty, and the first cell serves as the mutex that protects the second cell.

The producer sits in an infinite loop ($0 = 0$ encodes `true` into the syntax of the programming language) that, in each iteration, allocates a zero-size memory block, stores its address into local variable `x`, and inserts it into the shared buffer. The insertion operation is implemented by means of an inner loop that performs a busy wait until the buffer is empty.[1] (The `x := x` command, which has no effect, encodes into the syntax of the programming language that nothing should be done in case the buffer is full. It is necessary because the syntax of the programming language requires that each **if** statement have a **then** branch and an **else** branch.)

Similarly, the consumer sits in an infinite loop that, in each iteration, removes an element from the buffer and frees it.

The main program first allocates the shared buffer and initializes both cells to zero, indicating that the mutex is not held and the buffer is empty. It then

[1] Do not perform busy waiting in real programs! Instead, use blocking synchronization constructs such as condition variables or semaphores. Busy waiting harms performance and can cause starvation, but is used in this program for simplicity.

starts two producer threads and two consumer threads.

We will show how, with Concurrent Featherweight VeriFast, we can prove that this program does not access unallocated memory and does not perform data races.

2 Concrete Execution

In this section, we formally define what it means to execute a program written in the programming language defined in the previous section. The approach we use here for defining concrete execution is very different from the approach we use to define concrete execution in Featherweight VeriFast. In order to easily capture the fact that execution steps by concurrent threads are interleaved arbitrarily, we will define concrete execution by means of *execution traces*, which are sequences of *machine configurations* related by *machine steps*. Starting from an initial configuration, a machine executing a program repeatedly arbitrarily picks an enabled thread (i.e. a thread that is not blocked on an **acquire** operation) and executes one step of that thread's current command.

We first informally introduce the notion of machine configurations and execution traces by means of an example execution trace. Then we formally define the set of machine configurations and the machine step relation. Finally, we define what it means for a program to be safe.

Example Concrete Execution

$$
\begin{array}{l}
// \ \gamma = (\mathbf{0}, \{\!|(\mathbf{0}, \mathsf{pair} := \cdots ; \mathbf{fork}(\cdots); [\mathsf{pair}+1] := 24; \mathbf{done})|\!\}) \\
\mathsf{pair} := \mathbf{malloc}(2); \\
// \ \gamma = (\{\!|\mathsf{mb}(33,2), 33 \mapsto 1, 34 \mapsto -1|\!\}, \\
\qquad\quad \{\!|(\mathbf{0}[\mathsf{pair} := 33], \mathbf{fork}(\cdots); [\mathsf{pair}+1] := 24; \mathbf{done})|\!\}) \\
\mathbf{fork}([\mathsf{pair}] := 42); \\
\quad \gamma = (\{\!|\mathsf{mb}(33,2), 33 \mapsto 1, 34 \mapsto -1|\!\}, \\
// \qquad \{\!|(\mathbf{0}[\mathsf{pair} := 33], [\mathsf{pair}+1] := 24; \mathbf{done}), \\
\qquad\quad (\mathbf{0}[\mathsf{pair} := 33], [\mathsf{pair}] := 42; \mathbf{done})|\!\}) \\
[\mathsf{pair}+1] := 24 \\
// \ \gamma = (\{\!|\mathsf{mb}(33,2), 33 \mapsto 1, 34 \mapsto 24|\!\}, \\
\qquad\quad \{\!|(\mathbf{0}[\mathsf{pair} := 33], [\mathsf{pair}] := 42; \mathbf{done})|\!\}) \\
\qquad [\mathsf{pair}] := 42 \\
// \ \gamma = (\{\!|\mathsf{mb}(33,2), 33 \mapsto 42, 34 \mapsto 24|\!\}, \mathbf{0})
\end{array}
$$

Consider a simple program that allocates a memory block of size 2, forks a thread that sets the first cell to 42, and finally sets the second cell to 24. This program has infinitely many executions, each picking a different address for the memory block or different initial values for the memory cells, or executing the steps of the concurrent threads in a different order. One particular execution is shown above. Here, the memory block is allocated at address 33, the initial

values of the cells are 1 and -1, and the assignment of 24 to the second cell is executed before the assignment of 42 to the first cell.

The machine configurations γ are pairs (h, T) consisting of a concrete heap h and a *thread table* T. Concrete heaps are unchanged from Featherweight VeriFast: they are multisets of points-to chunks and malloc block chunks. A thread table is a multiset of threads. A thread is a pair (s, k) consisting of a concrete store s and a *continuation* k. Concrete stores, too, are unchanged from Featherweight VeriFast: they are functions that associate each local variable name with its current value. A thread's continuation denotes the work that remains to be done by the thread; it corresponds to the thread's instruction pointer in real architectures. A continuation is either **done**, meaning that the thread is finished; a command continuation $c; k$, meaning that the thread's remaining work is to execute command c followed by continuation k; or a return continuation $\mathbf{ret}(s, k)$, meaning that the thread is finished executing a routine and that it should restore the caller's store s and continue executing the caller's continuation k.

The initial machine configuation has an empty heap $\mathbf{0}$ and a single thread with an empty store $\mathbf{0}$ (mapping all variables to zero) and whose continuation encompasses the entire program. Executing the **malloc** command adds the chunks to the heap and binds variable pair in the main thread to the block's address. The **malloc** command is removed from the main thread's continuation. Executing the **fork** command removes it from the main thread's continuation and adds a new thread to the thread table whose continuation is the **fork** command's body. At this point, two threads are enabled and available for scheduling. In this execution, the main thread is scheduled first. It updates the second cell. After this step, the main thread is finished, and its continuation is **done**. (For conciseness, we do not show finished threads in the configurations above.) Finally, the forked thread is scheduled and the first cell is updated. After this step, no threads are left to be scheduled and the program is finished.

Concrete Execution: Machine Configurations

$$
\begin{aligned}
k \in \mathit{CContinuations} ::= \quad & \mathbf{done} \mid c; k \mid \mathbf{ret}(s, k) \\
p \in \mathit{CPredicates} = \quad & \{\mapsto, \mathsf{mb}\} \\
\mathit{CChunks} = \quad & \{p(\ell, v) \mid p \in \mathit{CPredicates}, \ell, v \in \mathbb{Z}\} \\
h \in \mathit{CHeaps} = \quad & \mathit{CChunks} \to \mathbb{N} \\
s \in \mathit{CStores} = \quad & \mathit{Vars} \to \mathbb{Z} \\
\theta \in \mathit{CThreads} = \quad & \mathit{CStores} \times \mathit{CContinuations} \\
T \in \mathit{CThreadTables} = \quad & \mathit{CThreads} \to \mathbb{N} \\
\gamma \in \mathit{CConfigurations} = \quad & \mathit{CHeaps} \times \mathit{CThreadTables}
\end{aligned}
$$

$$\ell \mapsto v \text{ is alternative syntax for } {\mapsto}(\ell, v)$$

We formally define the set of machine configurations above.

Machine Step Judgments

$$\langle s, h \mid k\rangle \rightsquigarrow_{\mathtt{t}} \langle s', h' \mid k'\rangle$$

$$\langle s, h \mid k\rangle \rightsquigarrow_{\mathtt{t}} \mathsf{failure}$$

$$\gamma \rightsquigarrow \tilde{\gamma}$$

$$\gamma \rightsquigarrow^{*} \tilde{\gamma}$$

$$\tilde{\gamma} \in \mathit{CConfigurations} \cup \{\mathsf{failure}\}$$

Next, we define the set of *machine steps*. The machine steps are the successful steps and the failure steps. A successful step, denoted $\gamma \rightsquigarrow \gamma'$, denotes that the machine can transition in one step from machine configuration γ to machine configuration γ'. A failure step, denoted $\gamma \rightsquigarrow \mathsf{failure}$, denotes that γ is an unsafe configuration.

We define the set of machine steps in terms of the set of *thread steps*. A successful thread step $\langle s, h \mid k\rangle \rightsquigarrow_{\mathtt{t}} \langle s', h' \mid k'\rangle$ denotes that, in a machine configuration with heap h, a thread with store s and continuation k can perform a step, resulting in new heap h', new store s', and new continuation k'. A failure thread step $\langle s, h \mid k\rangle \rightsquigarrow_{\mathtt{t}} \mathsf{failure}$ denotes that a machine configuration with thread h, where a thread has store s and continuation k, is an unsafe configuration.

We define the set of thread steps as those that can be obtained by instantiating the rules below.

Thread Step Rules

$$\langle s, h \mid x := e; k\rangle \rightsquigarrow_{\mathtt{t}} \langle s[x := s(e)], h \mid k\rangle \qquad \langle s, h \mid (c; c'); k\rangle \rightsquigarrow_{\mathtt{t}} \langle s, h \mid c; (c'; k)\rangle$$

$$\frac{\textbf{routine } r(\overline{x}) = c}{\langle s, h \mid r(\overline{e}); k\rangle \rightsquigarrow_{\mathtt{t}} \langle \mathbf{0}[\overline{x} := s(\overline{e})], h \mid c; \mathbf{ret}(s, k)\rangle}$$

$$\langle s, h \mid \mathbf{ret}(s', k)\rangle \rightsquigarrow_{\mathtt{t}} \langle s', h \mid k\rangle$$

A local variable assignment $x := e$ updates the binding of x with the value of e. Executing a sequential composition $c; c'$ means first executing c and then executing c'. Executing a routine call means first executing the body of the routine in a fresh store binding the parameters to the values of the argument expressions, and then executing the work remaining after the call in the original store.

Thread Step Rules: If and While

$$\frac{s(b) = \mathsf{true}}{\langle s, h \mid \textbf{if } b \textbf{ then } c \textbf{ else } c'; k\rangle \rightsquigarrow_{\mathsf{t}} \langle s, h \mid c; k\rangle}$$

$$\frac{s(b) = \mathsf{false}}{\langle s, h \mid \textbf{if } b \textbf{ then } c \textbf{ else } c'; k\rangle \rightsquigarrow_{\mathsf{t}} \langle s, h \mid c'; k\rangle}$$

$$\frac{s(b) = \mathsf{true}}{\langle s, h \mid \textbf{while } b \textbf{ do } c; k\rangle \rightsquigarrow_{\mathsf{t}} \langle s, h \mid c; (\textbf{while } b \textbf{ do } c; k)\rangle}$$

$$\frac{s(b) = \mathsf{false}}{\langle s, h \mid \textbf{while } b \textbf{ do } c; k\rangle \rightsquigarrow_{\mathsf{t}} \langle s, h \mid k\rangle}$$

The execution of **if** and **while** statements depends on the value of the condition, in the obvious way.

Thread Step Rules: Allocating and Freeing Memory

$$\frac{0 < \ell \qquad h' = \{\!|\mathsf{mb}(\ell, n), \ell \mapsto v_1, \ldots, v_n|\!\} \qquad \mathsf{dom}(h) \cap \mathsf{dom}(h') = \emptyset}{\langle s, h \mid x := \textbf{malloc}(n); k\rangle \rightsquigarrow_{\mathsf{t}} \langle s[x := \ell], h \uplus h' \mid k\rangle}$$

$$\frac{s(e) = \ell \qquad h' = \{\!|\mathsf{mb}(\ell, n), \ell \mapsto v_1, \ldots, v_n|\!\}}{\langle s, h \uplus h' \mid \textbf{free}(e); k\rangle \rightsquigarrow_{\mathsf{t}} \langle s, h \mid k\rangle}$$

$$\frac{\neg\exists \ell, n, v_1, \ldots, v_n.\ s(e) = \ell \wedge \{\!|\mathsf{mb}(\ell, n), \ell \mapsto v_1, \ldots, v_n|\!\} \leq h}{\langle s, h \mid \textbf{free}(e); k\rangle \rightsquigarrow_{\mathsf{t}} \mathsf{failure}}$$

Allocating a memory block of size n picks arbitrary initial values $v_1, \ldots, v_n$ for the new cells and an arbitrary positive address ℓ such that the new block does not overlap with existing blocks, binds variable x to this address, and adds the new chunks (one points-to chunk per allocated memory cell, and one malloc block chunk recording the size of the block) to the heap. (Note: $\{\!|\ell \mapsto v_1, v_2|\!\}$ abbreviates $\{\!|\ell \mapsto v_1, \ell + 1 \mapsto v_2|\!\}$.) Freeing a block at a given address fails if there is no malloc block chunk at that address; otherwise, the malloc block chunk and the associated points-to chunks are removed.

Thread Step Rules: Reading and Writing Memory

$$\langle s, h \uplus \{\!\{s(e) \mapsto v\}\!\} \mid x := [e]; k\rangle \rightsquigarrow_{\mathsf{t}} \langle s[x := v], h \uplus \{\!\{s(e) \mapsto v\}\!\} \mid k\rangle$$

$$\frac{\neg\exists v.\ \{\!\{s(e) \mapsto v\}\!\} \leq h}{\langle s, h \mid x := [e]; k\rangle \rightsquigarrow_{\mathsf{t}} \mathsf{failure}}$$

$$\langle s, h \uplus \{\!\{s(e) \mapsto v\}\!\} \mid [e] := e'; k\rangle \rightsquigarrow_{\mathsf{t}} \langle s, h \uplus \{\!\{s(e) \mapsto s(e')\}\!\} \mid k\rangle$$

$$\frac{\neg\exists v.\ \{\!\{s(e) \mapsto v\}\!\} \leq h}{\langle s, h \mid [e] := e'; k\rangle \rightsquigarrow_{\mathsf{t}} \mathsf{failure}}$$

Reading a memory cell at a given address fails if the cell is not allocated; otherwise, the specified variable is bound to the cell's value. Mutating a memory cell at a given address similarly fails if the cell is not allocated; otherwise, the cell's value is updated.

Thread Step Rules: Acquiring and Releasing a Mutex

$$\langle s, h \uplus \{\!\{s(e) \mapsto 0\}\!\} \mid \mathbf{acquire}(e); k\rangle \rightsquigarrow_{\mathsf{t}} \langle s, h \uplus \{\!\{s(e) \mapsto 1\}\!\} \mid k\rangle$$

$$\frac{\neg\exists v.\ \{\!\{s(e) \mapsto v\}\!\} \leq h}{\langle s, h \mid \mathbf{acquire}(e); k\rangle \rightsquigarrow_{\mathsf{t}} \mathsf{failure}}$$

$$\langle s, h \uplus \{\!\{s(e) \mapsto 1\}\!\} \mid \mathbf{release}(e); k\rangle \rightsquigarrow_{\mathsf{t}} \langle s, h \uplus \{\!\{s(e) \mapsto 0\}\!\} \mid k\rangle$$

$$\frac{\{\!\{s(e) \mapsto 1\}\!\} \not\leq h}{\langle s, h \mid \mathbf{release}(e); k\rangle \rightsquigarrow_{\mathsf{t}} \mathsf{failure}}$$

A thread whose continuation is of the form $\mathbf{acquire}(e); k$ fails if the specified address is not allocated; otherwise, it is enabled (can perform a step) only if the value of the specified cell is 0. (Otherwise, no step rule applies.) When executed, it sets the cell to 1. Releasing fails if the specified address is not allocated or if its value is not 1; otherwise, it sets the cell to 0.

Machine Step Rules

$$\frac{\langle s, h \mid k\rangle \rightsquigarrow_{\mathsf{t}} \langle s', h' \mid k'\rangle}{(h, T \uplus \{\!\{(s, k)\}\!\}) \rightsquigarrow (h', T \uplus \{\!\{(s', k')\}\!\})} \qquad \frac{\langle s, h \mid k\rangle \rightsquigarrow_{\mathsf{t}} \mathsf{failure}}{(h, T \uplus \{\!\{(s, k)\}\!\}) \rightsquigarrow \mathsf{failure}}$$

$$(h, T \uplus \{\!\{(s, \mathbf{fork}(c); k)\}\!\}) \rightsquigarrow (h, T \uplus \{\!\{(s, k), (s, c; \mathbf{done})\}\!\})$$

$$(h, T \uplus \{\!\{(s, \mathbf{done})\}\!\}) \rightsquigarrow (h, T)$$

The machine can perform a step if any of its threads can perform a thread step. Furthermore, forking a command adds a new thread to the thread table

whose remaining work is to execute the command; its initial store is the forking thread's current store. Finally, there is a step rule to allow garbage collection of finished threads.

Machine Step Rules: Data Races

$$\frac{\theta_1 \text{ writes } \ell \qquad \theta_2 \text{ accesses } \ell \qquad \neg(\theta_1 \text{ syncs\_on } \ell \wedge \theta_2 \text{ syncs\_on } \ell)}{(h, T \uplus \{\!\{\theta_1, \theta_2\}\!\}) \rightsquigarrow \text{failure}}$$

$$(s, \mathbf{acquire}(e); k) \text{ syncs\_on } s(e) \qquad (s, \mathbf{release}(e); k) \text{ syncs\_on } s(e)$$

$$(s, [e] := e; k) \text{ writes } s(e) \qquad \frac{\theta \text{ syncs\_on } \ell}{\theta \text{ writes } \ell}$$

$$(s, x := [e]; k) \text{ accesses } s(e) \qquad \frac{\theta \text{ writes } \ell}{\theta \text{ accesses } \ell}$$

A machine configuration exhibits a data race if two threads are simultaneously attempting to access the same memory location, and one of the accesses is a write, and the accesses are not both synchronisation operations. The synchronisation operations are the acquire and release operations. The write operations are the heap mutations and the synchronisation operations. The accesses are the read operations and the write operations.

We consider data races to be unsafe, because if a program exhibits data races, then executing it on a real machine may exhibit behaviors that it cannot exhibit according to the definitions given here. For example, consider the following program:

A Racy Program

$$\begin{array}{l}
\mathsf{x} := \mathbf{malloc}(1); \mathsf{y} := \mathbf{malloc}(1); [\mathsf{y}] := 0; \\
\mathbf{fork}([\mathsf{y}] := 1; [\mathsf{x}] := 1; [\mathsf{y}] := 2); \\
[\mathsf{x}] := 0; \mathsf{r1} := [\mathsf{y}]; \mathsf{r2} := [\mathsf{y}]; \mathsf{r3} := [\mathsf{x}]; \\
\mathbf{if}\ \mathsf{r1} = 0 \wedge \mathsf{r2} = 2 \wedge \mathsf{r3} = 0\ \mathbf{then}\ [0] := 0\ \mathbf{else}\ \mathsf{x} := \mathsf{x}
\end{array}$$

According to our definition of machine steps above, this program never executes the null pointer access: if $\mathsf{r1} = 0$ and $\mathsf{r2} = 2$, then the mutation of [x] in the forked thread occurs between the assignment to r1 and the assignment to r2. Therefore, we must have $\mathsf{r3} = 1$. However, on most real architectures, including x86 and x64, this program can perform the null pointer access. This is because most real architectures perform write buffering, and therefore an execution is possible where the mutation $[\mathsf{x}] := 0$ is stored in the main thread's write buffer, and the operation $\mathsf{r3} := [\mathsf{x}]$ is satisfied from the write buffer instead of main memory, and therefore still yields 0, even though the value in main memory is 1.

The same behavior could also be introduced by the compiler. In particular, consider a program that performs a write to some memory location x, and, at some later point, a read from that memory location. If a compiler can tell that no other writes to x and no synchronisation constructs intervene in the program between this write and this read, then, for performance reasons, in most cases it will assume that the read will yield the value written by the write, and it may optimize the read by using a value from a register instead of emitting a memory access. That is, most compilers perform optimizations that are correct only in the absence of data races.

As a result, if we verified the absence of failures in a program's executions according to our definitions, but ignoring the data race rule, then successful verification would not imply that we could safely execute the program on real platforms.

Concrete Semantics: Program Safety

$$\gamma \rightsquigarrow^* \gamma \qquad\qquad \frac{\gamma \rightsquigarrow^* \gamma' \qquad \gamma' \rightsquigarrow \gamma''}{\gamma \rightsquigarrow^* \gamma''}$$

Definition 1 (Safe Program)**.**

$$\mathsf{safe\_program}(c) \quad \Leftrightarrow \quad (\mathbf{0}, \{\!|(\mathbf{0}, c; \mathbf{done})|\!\}) \not\rightsquigarrow^* \mathsf{failure}$$

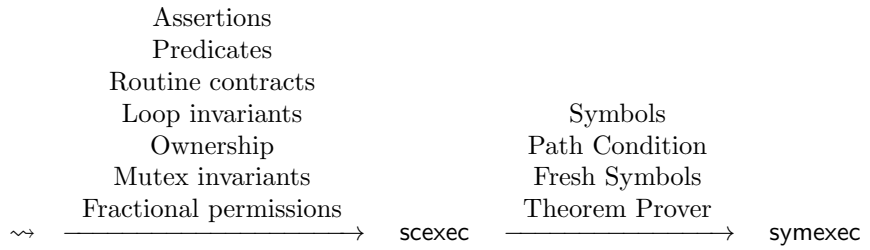
We say a configuration γ' is *reachable* from a configuration γ, denoted $\gamma \rightsquigarrow^* \gamma'$, if there is a sequence of one or more configurations, related by machine steps, that starts in γ and ends in γ'. In particular, any configuration is reachable from itself, and if γ' is reachable from γ and there is a machine step from γ' to γ'', then γ'' is reachable from γ.

We say a program with main command c is *safe* if failure is not reachable from the initial configuration (where the heap is empty and there is a single thread whose store maps all variables to zero and whose continuation is the program's main command).

3 Semiconcrete Execution

Solving the Verification Problem

	$\rightsquigarrow$	scexec	symexec
Recursion	Yes	No	No
Looping	Yes	No	No
Branching	Infinite	Infinite	Finite
Concurrency	Yes	No	No
Is Algorithm	No	No	Yes



In this section, we present semiconcrete execution for CFVF. Like in FVF, in semiconcrete execution, each routine is executed in isolation. Unlike in FVF, this means not just in isolation from other code running in the same thread, but also in isolation from code running in other threads. To achieve this, CFVF semiconcrete execution adds to the assertions, user-defined predicates, routine contracts, and loop invariants of FVF semiconcrete execution, the following new ingredients: *ownership* of chunks by threads and mutexes, **create_mutex** *ghost commands*, *mutex invariants*, mutex *and* mutex_held *chunks*, and *fractional chunks*. In the same way that routine contracts and loop invariants describe the interface between different pieces of code running in the same thread, thus allowing them to be verified in isolation, these new ingredients describe the interface between different threads, thus allowing each thread to be verified in isolation.

In particular, semiconcrete execution assigns *ownership* of each chunk to some thread, or to a mutex. A thread may access only the resources described by the chunks it owns. This rules out data races, and it allows us to reason about the code being executed by a thread without having to worry about what particular code other threads are executing concurrently. When a thread forks a new thread, it transfers ownership of some of the chunks it owns to the new thread.

Furthermore, the **create_mutex** ghost command indicates the point where a program starts using an allocated memory cell as a mutex to share among threads a particular set of resources. At this point, a *mutex invariant* is associated with the mutex that describes which resources (chunks) are protected by the mutex (or, equivalently, shared among threads through the mutex). Once a program starts using a memory cell as a mutex, the memory cell is no longer avaiable for regular reading and writing; its points-to chunk is replaced by a mutex chunk that indicates that this memory cell is now used as a mutex, with a particular associated mutex invariant. Furthermore, ownership of the chunks described by the mutex's mutex invariant is transfered from the thread to the mutex.

Semiconcrete execution checks at an **acquire**(ℓ) command that the memory cell at address ℓ is being used as a mutex; in particular, it checks that a mutex chunk exists for this address. To allow multiple threads to execute an **acquire**(ℓ) command concurrently, mutex chunks can be *split* into *fractions*, which may then be distributed among multiple threads. An **acquire**(ℓ) command requires only a fraction of the mutex chunk for ℓ. After an **acquire**(ℓ) command succeeds,

a mutex_held chunk is produced to indicate that the mutex is now held and may be released. Furthermore, ownership of the chunks protected by the mutex is transfered from the mutex to the thread. When the mutex is released, the mutex_held chunk is consumed and ownership of the chunks described by the mutex invariant is transfered again from the thread to the mutex.

To allow threads to perform read-only sharing of memory locations not protected by a mutex, semiconcrete execution allows not just mutex chunks, but points-to chunks as well to be split into fractions, which may then be passed to different threads. Writing a memory cell requires a full points-to chunk, but reading requires only a fractional points-to chunk. This rule allows read-only sharing while preventing data races.

The remainder of this section is structured as follows. In Sec. 3.1, we define semiconcrete execution. In Sec. 3.2, we prove that if a program is safe for semiconcrete execution, then it is safe for concrete execution.

3.1 Definition

In this subsection, we first illustrate semiconcrete execution and the program annotations it requires by means of the producer-consumer example. We then formally define the syntax of annotations, semiconcrete execution states, semiconcrete production and consumption of assertions, the semiconcrete execution mutator, and safety of a program under semiconcrete execution.

3.1.1 Example Program

First, we illustrate semiconcrete execution by means of the example program.

Example Program

```
mutex_invariant 1(b) =
  b + 1 ↦ ?e ∗ if e = 0 then 0 = 0 else mb(e, 0)

routine produce(b)
  req [_]mutex(b, 1)
  ens ¬(0 = 0)

routine consume(b)
  req [_]mutex(b, 1)
  ens ¬(0 = 0)

b := malloc(2); [b + 1] := 0; [b] := 0;
create_mutex(b, 1);
fork(produce(b));
consume(b)
```

Here we show routine contracts for routines produce and consume, as well as the annotated version of the main command. Both routines assert in their precondition that the memory cell at location b is being used as a mutex. The associated mutex invariant is denoted by the number 1, which is associated with an assertion by the **mutex_invariant** declaration. The parameter b of the **mutex_invariant** declaration can be used in the mutex invariant to denote the

address of the mutex. In this example, the mutex invariant expresses that the mutex protects the memory cell at address $\mathsf{b}+1$, as well as a malloc block chunk $\mathsf{mb}(\mathsf{e}, 0)$, where e is the value of the memory cell at address $\mathsf{b}+1$, provided that e is nonzero.

The preconditions of the routines assert only a *fraction* of the mutex chunk, as indicated by the $[\_]$ prefix.

The postconditions of the routines encode false into the assertion language of CFVF, indicating that the routines never return.

The **create_mutex** ghost command in the annotated version of the main command indicates that the memory cell at address b is henceforth used as a mutex with mutex invariant 1. The effect on the semiconcrete state is illustrated below.

Example Program

$$
\begin{array}{l}
\textbf{mutex\_invariant}\ 1(\mathsf{b}) = \\
\quad \mathsf{b}+1 \mapsto ?\mathsf{e} * \textbf{if}\ \mathsf{e} = 0\ \textbf{then}\ 0 = 0\ \textbf{else}\ \mathbf{mb}(\mathsf{e}, 0)
\end{array}
$$

routine produce(b)	**routine** consume(b)
req $[_]$mutex(b, 1)	**req** $[_]$mutex(b, 1)
ens $\neg(0 = 0)$	**ens** $\neg(0 = 0)$

$$
\begin{array}{l}
\mathsf{b} := \mathbf{malloc}(2); [\mathsf{b}+1] := 0; [\mathsf{b}] := 0; \\
//\ s = \mathbf{0}[\mathsf{b} := 10], h = \{\!|\mathsf{mb}(10, 2), 10 \mapsto 0, 11 \mapsto 0|\!\} \\
\mathbf{create\_mutex}(\mathsf{b}, 1); \\
//\ s = \mathbf{0}[\mathsf{b} := 10], h = \{\!|\mathsf{mb}(10, 2), \mathsf{mutex}(10, 1)|\!\} \\
\mathbf{fork}(\mathsf{produce}(\mathsf{b})); \\
//\ s = \mathbf{0}[\mathsf{b} := 10], h = \{\!|\mathsf{mb}(10, 2), [1/2]\mathsf{mutex}(10, 1)|\!\} \\
\mathsf{consume}(\mathsf{b})
\end{array}
$$

In particular, the **create_mutex** command turns the points-to chunk $10 \mapsto 0$ into a $\mathsf{mutex}(10, 1)$ chunk; furthermore, it consumes the chunk $11 \mapsto 0$ described by mutex invariant 1. Indeed, ownership of this chunk is transfered to the mutex; from now on, to access this memory cell, the mutex must first be acquired.

Forking the producer thread transfers to the new thread the resources described by the precondition of routine produce, which asserts some fraction of the mutex chunk. In this example execution trace, one half of the chunk is consumed, and one half remains in the main thread, to be consumed by the precondition of routine consume.

Example Program

```
routine consume(b)
  req [_]mutex(b, 1) ens ¬(0 = 0)
=
  while 0 = 0 inv [_]mutex(b, 1) do (
    x := 0;
    while x = 0
    inv [_]mutex(b, 1) ∗ if x = 0 then 0 = 0 else mb(x, 0)
    do (
      acquire(b);
      x := [b + 1];
      [b + 1] := 0;
      release(b)
    );
    free(x)
  )
```

Here we show the annotated version of routine consume. The outer loop's loop invariant is straightforward. The inner loop's loop invariant expresses that if variable x is nonzero, it points to an empty malloc block, which can then be freed by the **free**(x) command.

The body of the inner loop accesses the memory cell protected by the mutex. Semiconcrete execution succeeds because acquisition of the mutex transfers ownership of the resources described by the mutex invariant from the mutex to the thread. An example semiconcrete execution state directly after the mutex acquisition is shown below.

Example Program

```
routine consume(b)
  req [_]mutex(b, 1) ens ¬(0 = 0)
=
  while 0 = 0 inv [_]mutex(b, 1) do (
    x := 0;
    while x = 0
    inv [_]mutex(b, 1) ∗ if x = 0 then 0 = 0 else mb(x, 0)
    do (
      acquire(b);
      // s = 0[b := 10], h = {|mutex_held(10, 1), 11 ↦ 22, mb(22, 0)|}
      x := [b + 1];
      [b + 1] := 0;
      release(b)
    );
    free(x)
  )
```

In the case shown here, the memory cell holds nonzero value 22 when the mutex is acquired. This means that the resources transfered to the thread include a malloc block at address 22. Since routine consume clears the memory cell before releasing the mutex, ownership of this malloc block is not transfered back to the mutex when the mutex is released, since it is no longer considered to be part of the resources protected by the mutex, as described by the mutex invariant.

Ownership

γ_0 t_{main}:**0**
$\downarrow t_{\text{main}}$: **malloc**$(2) \to 10; [11] := 0; [10] := 0$
γ_1 t_{main}:{mb$(10, 2), 10 \mapsto 0, 11 \mapsto 0$}
$\downarrow t_{\text{main}}$: **create_mutex**$(10, 1)$
γ_2 t_{main}:{mutex$(10, 1)$}, m_{10}:{$11 \mapsto 0$}
$\downarrow t_{\text{main}}$: **fork** t_{prod}
γ_3 t_{main}:{$[1/2]$mutex$(10, 1)$}, t_{prod}:{$[1/2]$mutex$(10, 1)$}, m_{10}:{$11 \mapsto 0$}
$\downarrow t_{\text{prod}}$: **malloc**$(0) \to 22$
γ_4 t_{main}:{$[1/2]$mutex$(10, 1)$}, t_{prod}:{$[1/2]$mutex$(10, 1)$, mb$(22, 0)$}, m_{10}:{$11 \mapsto 0$}
$\downarrow t_{\text{prod}}$: **acquire**(10)
γ_5 t_{main}:{$[1/2]$mutex$(10, 1)$}, t_{prod}:{mutex_held$(10, 1)$, mb$(22, 0), 11 \mapsto 0$}, m_{10}:**0**
$\downarrow t_{\text{prod}}$: $[11] := 22$
γ_6 t_{main}:{$[1/2]$mutex$(10, 1)$}, t_{prod}:{mutex_held$(10, 1)$, mb$(22, 0), 11 \mapsto 22$}, m_{10}:**0**
$\downarrow t_{\text{prod}}$: **release**(10)
γ_7 t_{main}:{$[1/2]$mutex$(10, 1)$}, t_{prod}:{$[1/2]$mutex$(10, 1)$}, m_{10}:{$11 \mapsto 22$, mb$(22, 0)$}
$\downarrow t_{\text{main}}$: **acquire**(10)
γ_8 t_{main}:{mutex_held$(10, 1), 11 \mapsto 22$, mb$(22, 0)$}, t_{prod}:{$[1/2]$mutex$(10, 1)$}, m_{10}:**0**
$\downarrow t_{\text{main}}$: $[11] := 0$; **release**(10)
γ_9 t_{main}:{$[1/2]$mutex$(10, 1)$, mb$(22, 0)$}, t_{prod}:{$[1/2]$mutex$(10, 1)$}, m_{10}:{$11 \mapsto 0$}

The figure above shows an example execution trace of the producer-consumer program, and a possible ownership distribution at each point of the trace.[2]

Initially, there are no resources to be owned: the main thread owns nothing. After the main thread allocates a memory block of size 2 at address 10, it owns the mb$(10, 2)$ chunk and the two points-to chunks. Since the mb$(10, 2)$ chunk is not used in the remainder of this trace, for conciseness we do not mention it in the remainder of the figure, even though it remains owned by the main thread throughout the execution. The **create_mutex** ghost command turns the memory cell at location 10 into a mutex whose mutex invariant is the one associated with label 1 by the **mutex_invariant** annotation shown earlier. Correspondingly, the $10 \mapsto 0$ chunk is replaced by a mutex$(10, 1)$ chunk. Furthermore, the ghost command transfers ownership of the resources described by this mutex invariant to the mutex: after this ghost step, the mutex at address 10 owns the memory cell at address 11.

[2]For this execution trace, the ownership distributions shown are the only possible ones, except that when the main thread forks the producer thread, instead of transfering ownership of a $1/2$ fraction of the mutex$(10, 1)$ chunk, it could transfer any other fraction q with $0 < q < 1$.

The main thread then forks the producer thread. As part of a fork operation, ownership of a bundle of resources that satisfies the forked thread's precondition is transfered to the forked thread. In the example trace, this means that some fraction of the $\mathsf{mutex}(10, 1)$ chunk is transfered to the producer thread. In the example distribution, a $1/2$ fraction is transfered, but any other fraction between 0 and 1, exclusive, would be possible as well.

The producer thread then allocates an empty memory block at address 22; it receives ownership of the new $\mathsf{mb}(22, 0)$ chunk. It then acquires the mutex at address 10. This turns the $[1/2]\mathsf{mutex}(10, 1)$ chunk into a $\mathsf{mutex\_held}(10, 1)$ chunk; furthermore, this transfers ownership of the resources owned by the mutex to the thread. As a result, after this step the producer thread owns the memory cell at address 11, and the mutex owns nothing.

Now that the producer thread owns the memory cell at address 11, it can update it to point to the newly allocated memory block at address 22. It then releases the mutex, which turns the $\mathsf{mutex\_held}(10, 1)$ chunk back into a $[1/2]\mathsf{mutex}(10, 1)$ chunk, and transfers a bundle of resources that satisfies the mutex invariant to the mutex. This means the producer thread loses ownership of the memory cell at address 11, as well as the $\mathsf{mb}(22, 0)$ chunk, and the mutex receives ownership of these chunks.

In the example trace, at this point the main thread is again scheduled, and it acquires the mutex. Therefore ownership of both chunks owned by the mutex is transfered to the main thread. It then clears the memory cell at address 11 and releases the mutex. Again, the release operation transfers ownership of a bundle of resources that satisfies the mutex invariant to the mutex. However, since the memory cell now no longer points to the memory block at address 22, this bundle does not include the $\mathsf{mb}(22, 0)$ chunk, so it remains owned by the main thread.

Notice that in this example, ownership of the $\mathsf{mb}(22, 0)$ chunk was transfered from the producer thread to the main thread, via the mutex.

3.1.2 Formal definitions

Annotations

$$
\begin{array}{rcl}
 & & q \in \mathit{UserDefinedPredicates} \\
p & ::= & \mapsto \mid \mathsf{mb} \mid q \mid \mathsf{mutex} \mid \mathsf{mutex\_held} \\
a & ::= & b \mid p(\overline{e}, \overline{?x}) \mid [\_]p(\overline{e}, \overline{?x}) \mid a * a \mid \textbf{if } b \textbf{ then } a \textbf{ else } a \\
\mathit{preddef} & ::= & \textbf{predicate } q(\overline{x}) = a \\
\mathit{midef} & ::= & \textbf{mutex\_invariant } z(x) = a \\
c & ::= & \cdots \mid \textbf{while } b \textbf{ inv } a \textbf{ do } c \mid \textbf{open } q(\overline{e}) \mid \textbf{close } q(\overline{e}) \\
 & & \mid \textbf{create\_mutex}(e, z) \mid \textbf{fork}(r(\overline{e})) \\
\mathit{rspec} & ::= & \textbf{routine } r(\overline{x}) \textbf{ req } a \textbf{ ens } a
\end{array}
$$

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

The syntax of annotated programs is as follows. The predicates p are the points-to predicate $\mapsto$, the malloc block predicate mb, the user-defined predicates q, the mutex predicate and the $\mathsf{mutex\_held}$ predicate. The assertions a are the boolean expressions b, the predicate assertions $p(\overline{e}, \overline{?x})$, the *fractional predicate assertions* $[\_]p(\overline{e}, \overline{?x})$, the separating conjunctions $a * a$, and the conditional assertions **if** b **then** a **else** a. The predicate definitions are unchanged from FVF: they associate a parameter list $\overline{x}$ and a body assertion a with a user-defined predicate q. A **mutex_invariant** definition associates a parameter x and a body assertion a with an integer mutex invariant identifier z. As in FVF, loops are annotated with loop invariants, and **open** and **close** ghost commands may appear. Additionally, **create_mutex**(e, z) ghost commands may appear, which turn the memory cell at address e into a mutex with mutex invariant identifier z. For semiconcrete execution, we restrict the syntax of **fork** commands to be of the form **fork**$(r(\overline{e}))$, so that routine r's precondition describes for which resources ownership should be transfered to the new thread.

Semiconcrete execution

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

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

The only difference between semiconcrete states in CFVF, compared to FVF, apart from the potential presence of the new built-in predicates mutex and $\mathsf{mutex\_held}$, is that semiconcrete heaps are *fractional multisets*: they contain each chunk some fractional number of times, where $\mathbb{Q}^+$ denotes the nonnegative rational numbers.

Producing and Consuming Assertions

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

$$\mathsf{produce}([\_]p(\overline{e}, \overline{?x})) = \\ \quad \overline{v} \leftarrow \mathsf{eval}(\overline{e}); \bigotimes \pi, \overline{v}'.\ \mathsf{produce}([\pi]p(\overline{v}, \overline{v}')); \overline{x} := \overline{v}'$$

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

$$\mathsf{consume}([\_]p(\overline{e}, \overline{?x})) = \\ \quad \overline{v} \leftarrow \mathsf{eval}(\overline{e}); \bigoplus \pi, \overline{v}'.\ \mathsf{consume}([\pi]p(\overline{v}, \overline{v}')); \overline{x} := \overline{v}'$$

Semiconcrete production and consumption of assertions is unchanged from FVF, except that rules are added for producing and consuming fractional predicate assertions. Producing a fractional predicate assertion produces a demonically picked fraction of the chunk, where π ranges over the positive rational numbers, and $[\pi]p(\overline{v})$ denotes the fractional multiset $\mathbf{0}[p(\overline{v}) := \pi]$. Similarly, consuming a fractional predicate assertion consumes an angelically picked fraction of the chunk.

Semiconcrete Execution of Commands

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

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

$$\mathsf{scexec}(\mathbf{create\_mutex}(e, z)) = \ell \leftarrow \mathsf{eval}(e); \\ \quad \mathsf{c}(\ell \mapsto 0); \mathsf{with}(\mathbf{0}[x := \ell], \mathsf{c}(a)); \mathsf{p}(\mathsf{mutex}(\ell, z)) \\ \quad \text{where } \mathbf{mutex\_invariant}\ z(x) = a$$

$$\mathsf{scexec}(\mathbf{acquire}(e)) = \ell \leftarrow \mathsf{eval}(e); \bigoplus \pi, z. \\ \quad \mathsf{c}([\pi]\mathsf{mutex}(\ell, z)); \mathsf{p}(\mathsf{mutex\_held}(\ell, z)); \mathsf{with}(\mathbf{0}[x := \ell], \mathsf{p}(a)) \\ \quad \text{where } \mathbf{mutex\_invariant}\ z(x) = a$$

$$\mathsf{scexec}(\mathbf{release}(e)) = \ell \leftarrow \mathsf{eval}(e); \bigoplus z. \\ \quad \mathsf{c}(\mathsf{mutex\_held}(\ell, z)); \mathsf{with}(\mathbf{0}[x := \ell], \mathsf{c}(a)); \bigotimes \pi.\ \mathsf{p}([\pi]\mathsf{mutex}(\ell, z)) \\ \quad \text{where } \mathbf{mutex\_invariant}\ z(x) = a$$

Semiconcrete execution of commands is mostly unchanged from FVF, except that the rule for memory lookup commands is modified, and new rules are added for the new commands. We here use c and p to abbreviate consume and produce. Reading a memory cell consumes an angelically picked fraction of the points-to chunk ($\ell \overset{\pi}{\mapsto} v$ is alternative syntax for $[\pi]{\mapsto}(\ell, v)$) and then produces the same fraction. Forking a routine call consumes the routine's precondition.

Creating a mutex consumes the memory cell at the specified address, whose value must be zero, as well as the mutex invariant corresponding to the specified identifier, and produces a mutex chunk. Acquiring a mutex at a given address consumes an angelically picked fraction of the mutex chunk, and produces both a mutex_held chunk and the mutex invariant identified by the consumed mutex chunk. Releasing a mutex consumes a mutex_held chunk as well as the mutex invariant, and produces a demonically picked fraction of the mutex chunk.

Semiconcrete Execution

$$\mathsf{sc\text{-}safe\_program}(c) \quad \Leftrightarrow \quad ((\mathbf{0}, \mathbf{0}) \triangleright \mathsf{scexec}(c)\ \{\mathrm{true}\} \wedge \forall r.\ \mathsf{valid}(r))$$

As in FVF, a program is safe under semiconcrete execution if semiconcrete execution of the main command succeeds, starting from an empty heap and store, and all routines are valid. Routine validity is unchanged from FVF.

3.2 Soundness

In this section, we prove soundness of semiconcrete execution with respect to concrete execution, which means that if a program is safe under semiconcrete execution, then it is safe under concrete execution.

We first define semiconcrete execution of a continuation, and safety of a thread and of a thread table under a given semiconcrete heap. We then define the notion of a concrete heap concretizing a semiconcrete heap. Based on these notions, we can define safety of a machine configuration, and we can prove that the initial configuration is safe, and that machine steps preserve safety. Soundness then follows.

Soundness: Thread Safety

$$\begin{aligned}
&\mathsf{scexec}(\mathbf{done}) = \lambda s.\ \top \\
&\mathsf{scexec}(c; k) = \mathsf{scexec}(c); \mathsf{scexec}(k) \\
&\mathsf{scexec}(\mathbf{ret}(s, k)) = \mathsf{store} := s; \mathsf{scexec}(k)
\end{aligned}$$

$$\mathsf{sc\text{-}safe\_thread}((s, k)) = \lambda h.\ (s, h) \triangleright \mathsf{scexec}(k)\ \{\mathrm{true}\}$$

$$\mathsf{sc\text{-}safe\_thread\_table}(T) = \lambda h.\ h \vDash \circledast_{\theta \in T}\ \mathsf{sc\text{-}safe\_thread}(\theta)$$

$$\mathbf{0} \vDash \circledast_{x \in \mathbf{0}}\ P(x) \qquad \frac{h \vDash \circledast_{x \in X}\ P(x) \qquad P(x')(h')}{h \uplus h' \vDash \circledast_{x \in X \uplus \{x'\}}\ P(x)}$$

Semiconcrete execution of a continuation is defined straightforwardly in terms of semiconcrete execution of commands. Safety of a thread under a given heap means that semiconcrete execution of the thread's continuation under the given heap and the thread's store does not fail. Safety of a thread table under a given heap means that the heap can be split into subheaps, one for each thread, such that each thread is safe under its subheap.

Soundness: Mutex Tables

$$
\begin{array}{l}
\mathit{MutexTables} = \mathcal{P}(\mathbb{Z} \times \mathbb{Z} \times \{0,1\} \times [0,1] \times \mathit{SCHeaps}) \\
\\
\mathsf{consistent}(M) = \\
\quad (\forall(\ell, z, 0, 0, h_\mathsf{M}) \in M.\ (\mathbf{0}[x := \ell], h_\mathsf{M}) \triangleright \mathsf{c}(a); \mathsf{leakcheck}\ \{\mathrm{true}\} \\
\qquad \text{where } \mathbf{mutex\_invariant}\ z(x) = a) \\
\quad \wedge\ (\forall(\ell, z, v, \pi, h_\mathsf{M}) \in M.\ v = 0 \Leftrightarrow \pi = 0) \\
\\
\mathit{CMutexTables} = \{M \in \mathit{MutexTables} \mid \mathsf{consistent}(M)\} \\
\\
\mathsf{owned}(M) = \biguplus_{(\ell,z,0,0,h_\mathsf{M}) \in M} h_\mathsf{M} \\
\mathsf{mcells}(M) = \{\!|\ell \mapsto v \mid (\ell, z, v, \pi, h_\mathsf{M}) \in M|\!\} \\
\mathsf{mchunks}(M) = \{\!|[1-\pi]\mathsf{mutex}(\ell, z) \mid (\ell, z, v, \pi, h_\mathsf{M}) \in M, \pi < 1|\!\} \\
\qquad\qquad \uplus\ \{\!|\mathsf{mutex\_held}(\ell, z) \mid (\ell, z, 1, \pi, h_\mathsf{M}) \in M|\!\}
\end{array}
$$

We define the notion of a *mutex table* M; we will use it to describe the state of the mutexes that exist at some point. For each such mutex, it contains an element of the form $(\ell, z, v, \pi, h_\mathsf{M})$, where ℓ is the address of the mutex, z is the mutex invariant identifier, v is the value (0 indicating that the mutex is currently not held by any thread, and 1 indicating that some thread currently holds the mutex), π is the fraction of the mutex chunk that was consumed by the most recent **acquire**(ℓ) command, or 0 if the mutex is not currently held by any thread, and h_M denotes the semiconcrete chunks currently owned by the mutex.

We say a mutex table is *consistent* if for each mutex whose value is zero, its owned chunks satisfy its mutex invariant and its fraction π is zero, and for each mutex whose value is nonzero, its fraction is nonzero. We denote by *CMutexTables* the set of consistent mutex tables.

We define a number of properties of a mutex table M. The multiset of *owned* chunks of M is the multiset union of the owned chunks of the mutexes whose value is zero. The *mutex cells* of M are the points-to chunks corresponding to the cells currently being used as mutexes. The *mutex chunks* of M are the mutex and mutex_held chunks corresponding to the mutexes in M.

Soundness: Concurrent Heap Concretization

$$
\begin{array}{l}
h_\mathsf{c} \sim h \Leftrightarrow \\
\quad \exists M \in \mathit{CMutexTables}, h_{\mathsf{c}0}, h_\mathsf{tot}. \\
\qquad h_\mathsf{c} = h_{\mathsf{c}0} \uplus \mathsf{mcells}(M) \\
\qquad \wedge\ h_\mathsf{tot} \leq h_{\mathsf{c}0} \uplus \mathsf{mchunks}(M) \\
\qquad \wedge\ h_\mathsf{tot} \triangleleft h \uplus \mathsf{owned}(M)
\end{array}
$$

Semiconcrete heaps contain mutex and mutex_held chunks; concrete heaps do not. A mutex or mutex_held chunk in a semiconcrete heap implies the existence

in the corresponding concrete heap of the memory cell that is used to implement the mutex, but it also implies the existence of the resources protected by the mutex, provided the mutex is not currently held by some thread. Note, however, that mutexes may protect "semiconcrete resources" such as mutex and mutex_held chunks, in addition to concrete resources such as points-to and malloc block chunks.

Consider a semiconcrete heap h which represents the semiconcrete resources owned by the threads at some point during execution of a program. We say that h is *concretized* by some concrete heap h_c, denoted $h_\mathsf{c} \sim h$, if there exists a consistent mutex table M such that h_c is the multiset union of the mutex cells of M and some leftover concrete chunks $h_{\mathsf{c}0}$, and there exists some sub-multiset h_tot of the multiset of available semiconcrete resources (the leftover concrete chunks and the mutex chunks of M) that refines the multiset of resources owned by the threads and the mutexes.

(h_tot does not necessarily comprise all available semiconcrete resources because of garbage collection of threads that are done.)

Soundness: Machine Configuration Safety

$$\mathsf{sc\text{-}safe\_config}((h_c, T)) = \exists h.\ h_c \sim h \wedge \mathsf{sc\text{-}safe\_thread\_table}(T)(h)$$

Theorem 2 (Soundness)**.** *If* $\forall r.\ \mathsf{valid}(r)$ *then*

$$\begin{array}{rcl} \mathsf{sc\text{-}safe\_program}(c) & \Rightarrow & \mathsf{sc\text{-}safe\_config}(\mathbf{0}, \{\!|(\mathbf{0}, c; \mathbf{done})|\!\}) \\ \mathsf{sc\text{-}safe\_config}(\gamma) & \Rightarrow & \gamma \not\rightsquigarrow \mathsf{failure} \\ \mathsf{sc\text{-}safe\_config}(\gamma) \wedge \gamma \rightsquigarrow \gamma' & \Rightarrow & \mathsf{sc\text{-}safe\_config}(\gamma') \\ \\ \mathsf{sc\text{-}safe\_program}(c) & \Rightarrow & \mathsf{safe\_program}(c) \end{array}$$

Semiconcrete safety of a machine configuration means that the heap concretizes some semiconcrete heap for which the thread table is safe. If a program is safe under semiconcrete execution, then its initial machine configuration is safe, and safe configurations do not step to failure, and safe configurations step to safe configurations. The latter two lemmas can be proven by case analysis on the step rule. It follows that the program is safe under concrete execution.

4 Further Reading

CFVF, like FVF, is based on *separation logic*, which is itself based on *Hoare logic*. See [10] for general references on these topics; here, we cite work on verification of concurrent programs specifically.

A Hoare logic for programs that use *monitors* (similar to mutexes) was proposed by Hoare [4]. He proposed the notion of *monitor invariants*, similar to mutex invariants.

O'Hearn proposed a separation logic for concurrency [9]. It was implemented in the Smallfoot tool [1].

Concurrent separation logic was extended with fractional permissions by Bornat *et al.* [2].

In the work mentioned above, mutexes were statically allocated. A logic for dynamically allocated mutexes was proposed by Gotsman *et al.* [3].

Unlike CFVF, VeriFast supports the specification and verification of functional correctness of concurrent modules. It uses *higher-order ghost code* for this [6], specified using a form of *nested Hoare triples* [7] and whose termination is verified modularly [5]. A more abstract alternative is the *Iris* logic [8].

References

[1] J. Berdine, C. Calcagno, and P.W. O'Hearn (2006). Smallfoot: modular automatic assertion checking with separation logic. In *FMCO'06*.

[2] R. Bornat, C. Calcagno, P.W. O'Hearn and M. Parkinson (2005). Permission accounting in separation logic. In *POPL'05*.

[3] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv (2007). Local reasoning for storable locks and threads. In *APLAS'07*.

[4] C.A.R. Hoare (1974). Monitors: An operating system structuring concept. *Communications of the ACM* 17(10).

[5] B. Jacobs, D. Bosnacki, and R. Kuiper. Modular termination verification. In *ECOOP'15*.

[6] B. Jacobs and F. Piessens (2011). Expressive modular fine-grained concurrency specification. In *POPL'11*.

[7] B. Jacobs, J. Smans, and F. Piessens (2011). Verification of unloadable modules. In *FM'11*.

[8] R. Jung, R. Krebbers, L. Birkedal, and D. Dreyer (2016). Higher-order ghost state. In *ICFP'16*.

[9] P.W. O'Hearn (2004). Resources, concurrency and local reasoning. In *CONCUR'04*.

[10] F. Vogels, B. Jacobs, and F. Piessens (2015). Featherweight VeriFast. *Logical Methods in Computer Science* 11(3:19).