Verifying purely `unsafe` Rust programs with VeriFast: a tutorial

Bart Jacobs
KU Leuven, Department of Computer Science, DistriNet Research Group

October 22, 2025

Contents

1 Introduction

VeriFast is a tool for verifying the absence of undefined behavior[1] in single-threaded and multithreaded Rust[2] programs that use **`unsafe`** blocks. The tool reads an `.rs` source code file and reports either "0 errors found" or indicates the location of a potential error. If the tool reports "0 errors found", this means[3], among other things, that the program

- does not perform illegal memory accesses, such as reading or writing a struct instance field after the struct instance has been deallocated, or reading or writing beyond the end of an array (known as a *buffer overflow*, the most common cause of security vulnerabilities in operating systems and internet services) and
- does not include a certain type of concurrency errors known as *data races*, i.e. unsynchronized conflicting accesses of the same field by multiple threads. Accesses are considered conflicting if at least one of them is a write access. And
- complies with function preconditions and postconditions specified by the programmer in the form of special comments (known as *annotations*) in the source code.

Many errors in Rust programs that use **`unsafe`** blocks, such as illegal memory accesses and data races, are generally very difficult to detect by conventional means such as testing or code review, since they are often subtle and typically do not cause a clean crash but have unpredictable effects that are difficult to diagnose. However, even many security-critical and safety-critical Rust programs, such as operating systems, device drivers, web servers (that may serve e-commerce or e-banking applications), embedded software for automobiles, airplanes, space applications, nuclear and chemical plants, etc. use **`unsafe`** blocks, where these programming errors may enable cyber-attacks or cause injuries. For such programs, formal verification approaches such as VeriFast may be the most effective way to achieve the desired level of reliability.

To detect all errors, VeriFast performs *modular symbolic execution* of the program. In particular, VeriFast symbolically executes the body of each function of the program, starting from the symbolic state described by the function's precondition, checking that *permissions* are present in the symbolic state for each memory location accessed by a statement, updating the symbolic state to take into account each statement's effect, and checking, whenever the function returns, that the final symbolic state satisfies the function's postcondition. A symbolic state consists of a *symbolic heap*, containing permissions (known as *chunks*) for accessing certain memory locations, a *symbolic store*, assigning a symbolic value to each local variable, and a *path condition*, which is the set of *assumptions* about the values of the *symbols* used in the symbolic state on the current execution path. Symbolic execution always terminates, because thanks to the use of loop invariants each loop body needs to be symbolically executed only once, and symbolically executing a function call uses only the function's precondition and postcondition, not its body.

We will now proceed to introduce the tool's features for verifying purely **`unsafe`** programs step by step. To try the examples and exercises in this tutorial yourself, please download the release from the VeriFast website at

`https://github.com/verifast/verifast`

. You will find in the `bin` directory a command-line version of the tool (`verifast.exe`), and a version that presents a graphical user interface (`vfide.exe`).

Throughout this tutorial, you will find exercises that ask you to annotate a small Rust program so that it is accepted by VeriFast. You can find solutions to all exercises at the end of this tutorial. Furthermore, un-annotated and annotated versions of each solution are included in the VeriFast distribution in directories `rust_tutorials/purely_unsafe/problems` and `rust_tutorials/purely_unsafe/solutions`, respectively.

Note: In this tutorial, we do not address the following complexities of verifying Rust programs:

[1] `https://doc.rust-lang.org/reference/behavior-considered-undefined.html`

[2] VeriFast also supports C and Java. See *The VeriFast Program Verifier: A Tutorial* and *VeriFast for Java: A Tutorial*.

[3] There are a few known reasons (known as *unsoundnesses*) why the tool may sometimes incorrectly report "0 errors found"; see `https://verifast.github.io/verifast/rust-reference/#soundness-of-verifast`. There may also be unknown unsoundnesses.

- We do not verify that the cleanup performed during stack unwinding after a panic is safe, because VeriFast does not yet have good support for doing so.[4] Therefore, we run VeriFast with the `-ignore_unwind_paths` flag. This is sound only when compiling with `-C panic=abort`.
- We avoid creating shared references, mutable references, or boxes, so that we can ignore Rust's pointer aliasing rules and rules on mutating immutable bytes[5]. So far, VeriFast has only partial support for verifying compliance with these rules.[6]
- We avoid verifying non-**`unsafe`** functions, which involves verifying that they are *sound*[7], i.e. safe for use by code verified only by the Rust type checker. Resources on this topic are available elsewhere[8].

2 Example: illegal_access.rs

To illustrate how VeriFast can be used to detect programming errors that are difficult to spot through testing or code review, we start by applying the tool to a very simple Rust program that contains a subtle error.

Please start `vfide.exe` with the `illegal_access.rs` program that can be downloaded from

`https://www.cs.kuleuven.be/~bartj/verifast/illegal_access.rs`

. The program will be shown in the VeriFast IDE. To verify the program, choose the **Verify program** command in the **Verify** menu, press the **Play** toolbar button, or press F5. You will see something like Fig 1. The program attempts to access the field `balance` of the struct instance `my_account` allocated using `alloc`. However, if there is insufficient memory, `alloc` returns a null pointer and no memory is allocated. VeriFast detects the illegal memory access that happens in this case. Notice the following GUI elements:

- The erroneous program element is displayed in a red color with a double underline.
- The error message states: "No matching heap chunks: Account_balance_(my_account, _)". Indeed, in the scenario where there is insufficient memory, the memory location (or *heap chunk*) that the program attempts to access is not accessible to the program. Account_balance_ is the type of heap chunk that represents the (possibly uninitialized) balance field of the instance of struct Account pointed to by pointer my_account.
- The assignment statement is shown on a yellow background. This is because the assignment statement is the *current step*. VeriFast verifies each function by stepping through it, while keeping track of a symbolic representation of the relevant program state. You can inspect the symbolic state at each step by selecting the step in the Steps pane in the lower left corner of the VeriFast window. The program element corresponding to the current step is shown on a yellow background. The symbolic state consists of the *path condition*, shown in the Assumptions pane; the *symbolic heap*, shown in the Heap chunks pane; and the *symbolic store*, shown in the Locals pane.

To correct the error, uncomment the commented statement. Now press F5 again. We get a green bar: the program now verifies. This means VeriFast has symbolically executed all possible paths of execution through function `main`, and found no errors.

Let's take a closer look at how VeriFast symbolically executed this function. After VeriFast has symbolically executed a program, you can view the *symbolic execution tree* for each function in the Trees pane. The Trees pane is hidden by default, but you can reveal it by dragging the right-hand border of the VeriFast window to the left. At the top of the Trees pane is a drop-down list of all functions that have been symbolically executed. Select the Verifying function '`main`' item to view the symbolic execution tree for function `main`.

A symbolic execution tree has four kinds of nodes:

[4] `https://github.com/verifast/verifast/issues/859`
[5] `https://doc.rust-lang.org/reference/behavior-considered-undefined.html`
[6] `https://github.com/verifast/verifast/issues/685`
[7] `https://doc.rust-lang.org/nomicon/working-with-unsafe.html`
[8] `https://verifast.github.io/verifast/rust-reference/non-unsafe-funcs.html`

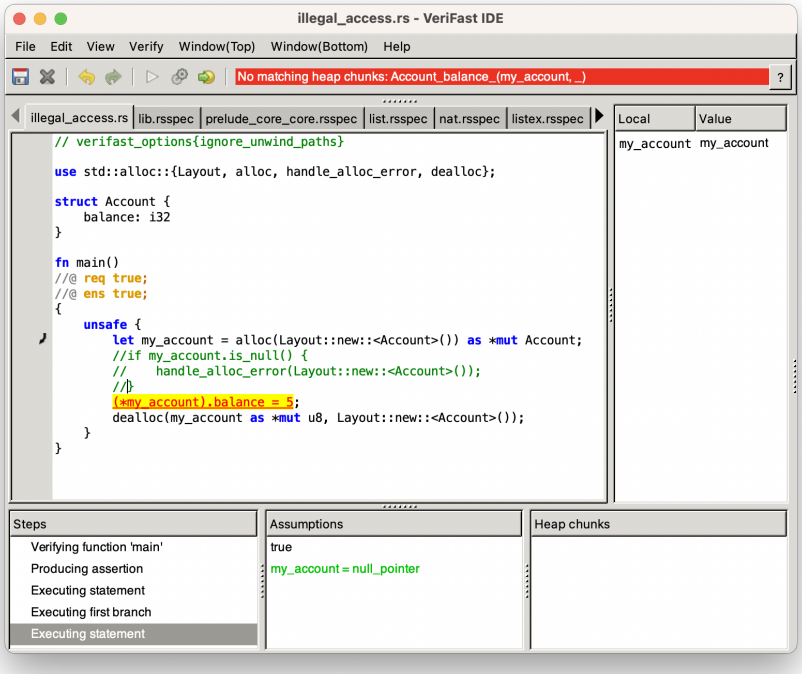

Figure 1: A screenshot of `illegal_access.rs` in the VeriFast IDE

- The *top node* represents the start of the symbolic execution. Click the top node: in the initial symbolic execution state, there are no heap chunks (the Heap chunks pane is empty), there are no local variables (the Locals pane is empty), and there are no assumptions (the Assumptions pane is empty).

- There is one *fork node* at each point where a symbolic execution path forks into two paths. This happens when multiple cases need to be considered in the symbolic execution; it is therefore also called a *case split*. The symbolic execution of function `main` involves three case splits: the first case split is where symbolic execution of the `alloc` call forks into one branch where no memory is available and therefore `alloc` returns a null pointer, and another branch where memory is available and therefore `alloc` allocates the requested amount of memory and returns a pointer to it. A case split also happens on each of these branches when symbolically executing the **`if`** statement.

- A black *final node* is shown for the final symbolic execution state on each path where additional symbolic execution steps were performed after the last fork. This is the case for one of the branches of each of the two fork nodes corresponding to the **`if`** statement.

- There is one green *leaf node* at the end of each complete symbolic execution path through the function. Click the black node preceding any leaf node to see the full corresponding symbolic execution path in the Steps pane. Function `main` has four symbolic execution paths: two paths end when the **`if`** branch is detected to be inconsistent with the preceding `alloc` branch; one path where no memory is available ends when the program ends due to the call of `handle_alloc_error`; and a fourth path, where memory is available, ends when the function returns.

Click the black node preceding the rightmost leaf node to view the (feasible) execution path where `alloc` successfully allocates memory. Notice that VeriFast shows arrows in the left margin next to the code of function `main` to indicate that this path executes the second case of the `alloc` statement and the second case of the **`if`** statement.

To better understand the details of VeriFast's symbolic execution, we will step through this path from the top. Select the first step in the Steps pane. Then, press the Down arrow key. The Verifying function `main` step does not affect the symbolic state. The Producing assertion step adds the assumption true to the Assumptions. We will consider production and consumption of assertions in detail later in this tutorial. We now arrive at the Executing statement step for the `alloc` statement. This statement's effect takes place in the next step, the Executing second branch step; it affects the symbolic state in three ways:

- It adds the heap chunks Account_balance_(my_account, dummy) and alloc_block_Account(my_account) to the symbolic heap (as shown in the Heap chunks pane). Here, my_account and dummy are *symbols* that represent unknown values. Specifically, my_account represents a pointer to the newly allocated struct instance, and dummy represents the initial state of the `balance` field of the struct instance. (The initial state of memory allocated using `alloc` is unspecified, unlike in Java where the fields of a new object are initialized to the default value of their type. In fact, in Rust, uninitialized memory might be in a special *poisoned* state whose behavior is different from any initialized state.[9] VeriFast considers reading from poisoned memory to be an error.)

  VeriFast *freshly picks* these symbols during this symbolic execution step. That is, to represent the location in memory of the new struct instance and the initial state of the `balance` field, VeriFast uses symbols that are not yet being used on this symbolic execution path. To see this, try verifying the function `test` shown below:

```
unsafe fn test()
//@ req true;
//@ ens true;
{
```

[9] See `https://www.ralfj.de/blog/2019/07/14/uninit.html`. We here use the term *poisoned* instead of *uninitialized* to indicate "a memory state that behaves differently from any initialized state" because VeriFast allows for the possibility that memory that has not yet been initialized is in fact in a regular state that behaves just like an initialized state.

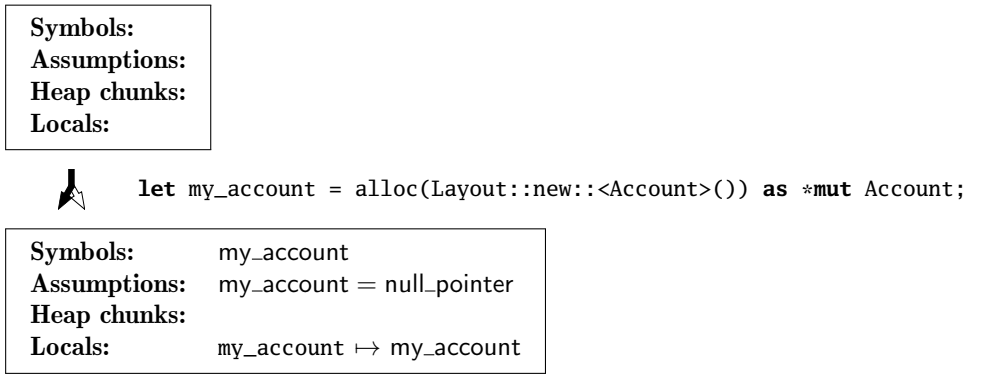

Figure 2: Symbolic execution of an `alloc` statement (first case)

```
    let my_account = alloc(Layout::new::<Account>()) as *mut Account;
    if my_account.is_null() {
        handle_alloc_error(Layout::new::<Account>());
    }
    let your_account = alloc(Layout::new::<Account>()) as *mut Account;
    if your_account.is_null() {
        handle_alloc_error(Layout::new::<Account>());
    }
}
```

Notice that to symbolically execute the first `alloc` statement, VeriFast picked symbols my_account and dummy, and to symbolically execute the second `alloc` statement, VeriFast picked symbols your_account and dummy0.

- It adds the assumption get_pointer_address(my_account) $\neq 0$ (perhaps written differently) to the path condition (as shown in the Assumptions pane). Indeed, if `alloc` succeeds, the returned pointer is not a null pointer.
- It adds a binding to the symbolic store (shown in the Locals pane) that binds local variable `my_account` to symbolic value my_account. Indeed, the program assigns the result of the `alloc` call (represented by the symbol my_account) to the local variable `my_account`. Note that the fact that in this case the local variable and the symbol have the same name is incidental and has no special significance.

Figure 3 summarizes the symbolic execution of `alloc` statements, in the successful case. Figure 2 summarizes the unsuccessful case.

The next step in the symbolic execution trace is the symbolic execution of the **`if`** statement. An **`if`** statement is like an `alloc` statement in the sense that there are two cases to consider; therefore, for **`if`** statements, too, VeriFast performs a case split and forks the symbolic execution path into two branches. On the first branch, VeriFast considers the case where the condition of the **`if`** statement is true. It adds the assumption that this is the case to the path condition and symbolically executes the *then* block of the **`if`** statement. On the second branch, VeriFast considers the case where the condition of the **`if`** statement is false. It adds the corresponding assumption to the path condition and symbolically executes the *else* block, if any. Note that after adding an assumption to the path condition, VeriFast always checks if it can detect an inconsistency in the resulting path condition; if so, the current symbolic execution path does not correspond to any real execution path, so there is no point in continuing the symbolic execution of this path and VeriFast abandons it. This is what happens with the first branch of the **`if`** statement after a successful `alloc`; it is also what happens with the second branch of the **`if`** statement after an unsuccessful `alloc`.

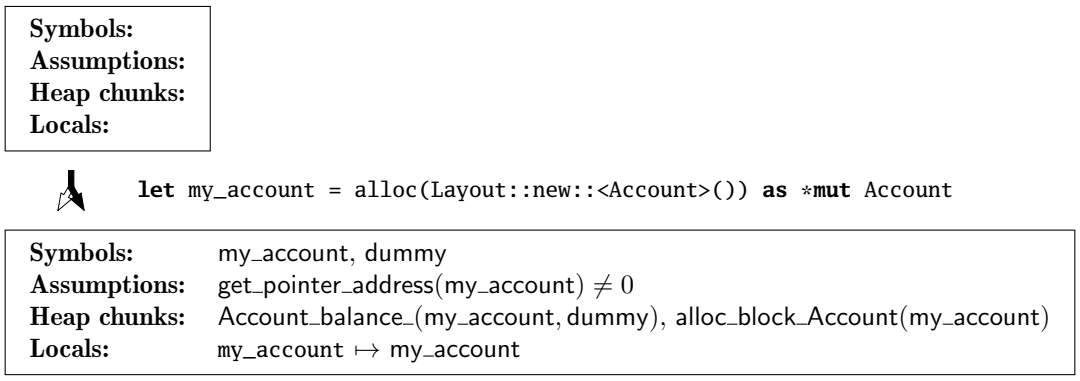

Figure 3: Symbolic execution of an `alloc` statement (second case)

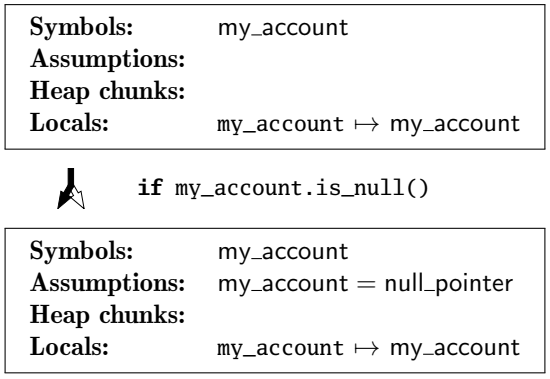

Figure 4: Symbolic execution of an **`if`** statement (first case). Symbolic execution continues with the *then* block of the **`if`** statement.

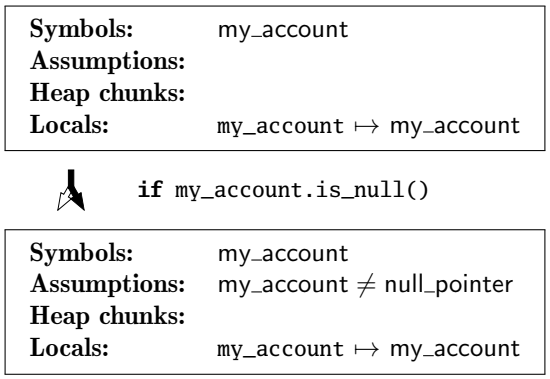

Figure 5: Symbolic execution of an **`if`** statement (second case). Symbolic execution continues with the *else* block of the **`if`** statement, if any.

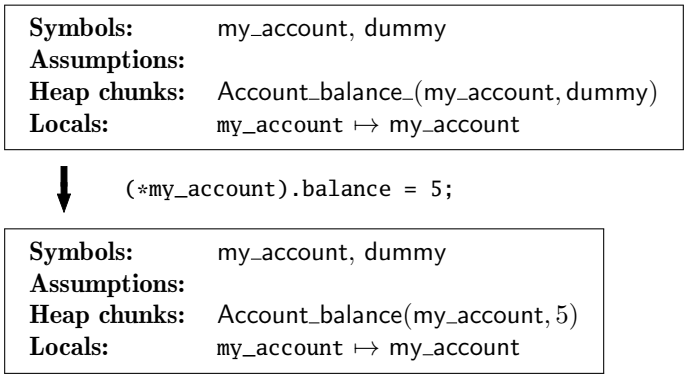

Figure 6: Symbolic execution of a struct field assignment statement

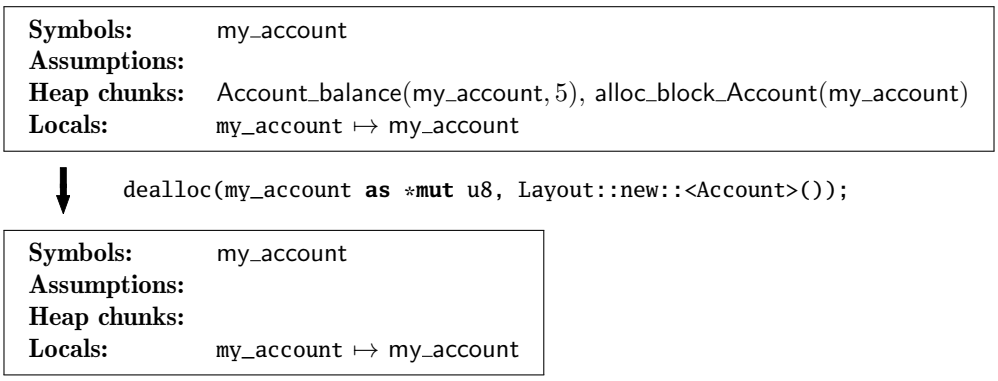

Figure 7: Symbolic execution of a `dealloc` statement

Figures 4 and 5 summarize the two cases of the symbolic execution of an **`if`** statement.

The next step of the symbolic execution path symbolically executes the statement that assigns value 5 to the `balance` field of the newly allocated struct instance. When symbolically executing an assignment to a field of a struct instance, VeriFast first checks that a (possibly uninitialized) heap chunk for that field of that struct instance is present in the symbolic heap. If not, it reports a "No such heap chunk" verification failure. It might mean that the program is trying to access unallocated memory. If the chunk is present, VeriFast replaces the chunk with a (definitely initialized) chunk whose value is the value of the right-hand side of the assignment. This is shown in Figure 6. Notice that for any type of chunk named n (such as Account_balance) that represents definitely initialized memory, there is a corresponding type of chunk named $n\_$ (such as Account_balance_) that represents possibly uninitialized memory.

Finally, symbolic execution of the `dealloc` statement checks that the two heap chunks that were added by the `alloc` statement (the chunk for the `balance` field and the alloc block chunk) are still present in the symbolic heap. If not, VeriFast reports a verification failure; the program might be trying to deallocate a struct instance that has already been deallocated. Otherwise, it removes the chunks, as shown in Figure 7. This ensures that if a program deallocates a struct instance and then attempts to access that struct instance's fields, symbolic execution of the statements accessing the fields will fail (because the heap chunks for the fields will be missing).

A program's *symbolic execution forest* (consisting of one symbolic execution tree for each of the program's functions) constitutes a finite description of the program's (usually infinite-depth and infinite-width) *concrete execution tree*, consisting of the program's *concrete execution paths*.

The concrete execution tree of the corrected version of the program of Figure 1 is shown in Figure 8. Notice that it is in fact (practically) infinite-width. For this example program, the concrete execution tree

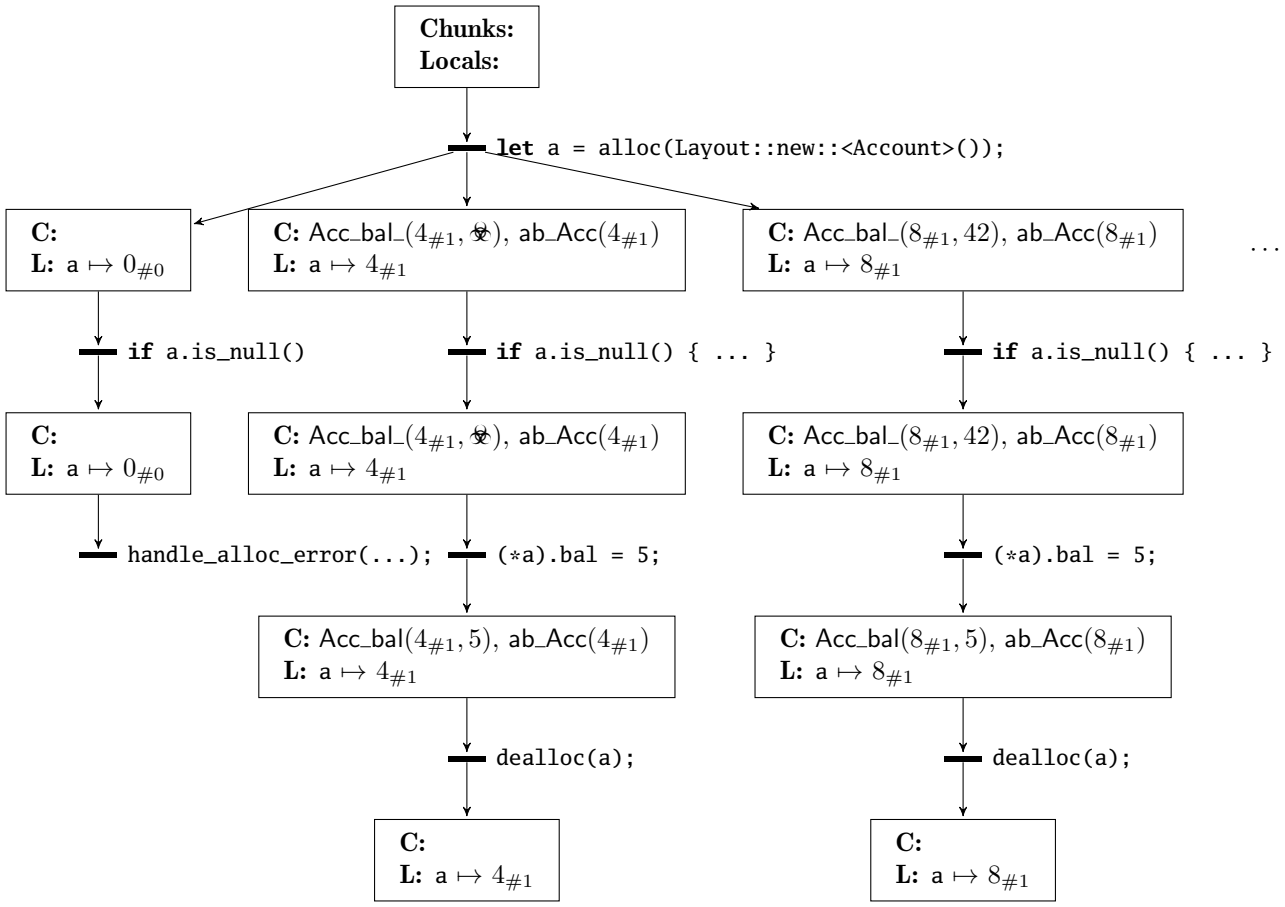

Figure 8: Concrete execution tree of the program of Figure 1 (after uncommenting the **if** statement). We abbreviate **Chunks** as **C**, **Locals** as **L**, `Account` as `Acc`, `balance` as `bal`, `my_account` as `a`, and alloc_block as `ab`. Note that the tree is not shown fully: the `alloc` node has one child node for each combination (ℓ, s) of a memory location ℓ for the new account and an initial state s of its `balance` field. Note that in Rust, a pointer/memory location $\ell = a_p$ consists of an *address* a and a *provenance* p; see `https://www.ralfj.de/blog/2018/07/24/pointers-and-bytes.html`. The null pointer has address 0 and special provenance #0. Pointers to allocated memory blocks have as their provenance the allocation identifier (#1 is the provenance of the first block allocated by the program) and have a nondeterministically selected address; the diagram shows 4 and 8 as examples of such addresses for conciseness. (Note: the provenance model suggested here is just an example; in reality, VeriFast abstracts over the precise provenance model, and in fact, by specifying `-assume_no_provenance` on the command line, you can make VeriFast assume (unsoundly!) that all pointers have the same provenance #0.) The initial state s of a memory location can be the poisoned state (☣) or some value, e.g. 42.

is not infinite-depth, because the program has no loops or recursion. If the program had a loop with an unbounded number of iterations, the tree would have been infinite-depth as well.

Exercise 1 *Draw the symbolic execution tree of the* `main` *function of the corrected version of the program of Figure 1. A symbolic execution tree differs from a concrete execution tree in that each state includes a set of used* symbols *and a set of* assumptions *about these symbols, and in that the heap chunks and the local variable bindings are expressed using these symbols. Furthermore, by using these symbols, a single path in a symbolic execution tree can be used to describe infinitely many corresponding paths in the corresponding concrete execution tree.*

3 alloc_block Chunks

To better understand why the *alloc* statement generates both an Account_balance_ chunk and an alloc_block_Account chunk, change the program so that the struct instance is allocated as a local variable on the stack instead of being allocated on the heap:

```
fn main()
//@ req true;
//@ ens true;
{
   unsafe {
      let mut my_account_local = Account { balance: 0 };
      let my_account = &raw mut my_account_local;
      (*my_account).balance = 5;
      dealloc(my_account as *mut u8, Layout::new::<Account>());
   }
}
```

This program first allocates an instance of struct Account on the stack and calls it my_account_local. It then assigns the address of this struct instance to pointer variable my_account. The remainder of the program is as before: the program sets the balance field to value 5 and then attempts to deallocate the struct instance.

If we ask VeriFast to verify this program, VeriFast reports the error

No matching heap chunks: alloc_block_Account(my_account_local_addr)

at the *dealloc* statement. Indeed, the call of *dealloc* is incorrect, since *dealloc* may only be applied to a struct instance allocated on the heap by alloc, not to a struct instance allocated on the stack as a local variable.

VeriFast detects this error as follows: 1) VeriFast generates an alloc_block chunk only for struct instances allocated using *alloc*, not for struct instances allocated on the stack. 2) When verifying a *dealloc* statement, VeriFast checks that an alloc_block chunk exists for the struct instance being freed.

Notice that, in contrast, the account_balance_ chunk is generated in both cases. As a result, the statement that initializes the balance field verifies successfully, regardless of whether the struct instance was allocated on the heap or on the stack.

4 Functions and Contracts

We continue to play with the example of the previous section. The example currently consists of only one function: the main function. Let's add another function. Write an Account associated function set_balance that takes a pointer to an Account struct instance and a integer amount, and assigns this amount to the struct instance's balance field. Then replace the field assignment in the main function with a call to this function. We now have the following program:

```
// verifast_options{ignore_unwind_paths}
use std::alloc::{Layout, alloc, handle_alloc_error, dealloc};

struct Account {
   balance: i32
}

impl Account {

   unsafe fn set_balance(my_account: *mut Account, new_balance: i32) {
      (*my_account).balance = new_balance;
   }

}

fn main()
//@ req true;
//@ ens true;
{
   unsafe {
      let my_account = alloc(Layout::new::<Account>()) as *mut Account;
      if my_account.is_null() {
         handle_alloc_error(Layout::new::<Account>());
      }
      Account::set_balance(my_account, 5);
      dealloc(my_account as *mut u8, Layout::new::<Account>());
   }
}
```

If we try to verify the new program, VeriFast complains that the new function has no contract. Indeed, VeriFast verifies each function separately, so it needs a precondition and a postcondition for each function to describe the initial and final state of calls of the function.

Add the same contract that the main function has:

```
unsafe fn set_balance(my_account: *mut Account, new_balance: i32)
//@ req true;
//@ ens true;
```

Notice that contracts, like all VeriFast annotations, are in comments, so that the Rust compiler ignores them. VeriFast also ignores comments, except the ones that are marked with an at (@) sign.

VeriFast now no longer complains about missing contracts. However, it now complains that the field assignment in the body of Account::set_balance cannot be verified because the symbolic heap does not contain a heap chunk that grants permission to access this field. To fix this, we need to specify in the function's precondition that the function requires permission to access the balance field of the Account struct instance pointed to by my_account. We achieve this simply by mentioning the heap chunk in the precondition:

```
unsafe fn set_balance(my_account: *mut Account, new_balance: i32)
//@ req Account_balance_(my_account, _);
//@ ens true;
```

Notice that we use an underscore in the position where the state of the field belongs. This indicates that we do not care about the old state of the field when the function is called.[10]

[10] VeriFast also supports a more concise syntax for field chunks. For example, `Account_balance_(my_account, _)` can also be written as `(*my_account).balance |-> _`. In fact, the latter (field chunk-specific) syntax is generally recommended over

VeriFast now highlights the brace that closes the body of the function. This means we successfully verified the field assignment. However, VeriFast now complains that the function leaks heap chunks. For now, let's simply work around this error message by inserting a **leak** command, which indicates that we're happy to leak this heap chunk. We will come back to this later.

```
unsafe fn set_balance(my_account: *mut Account, new_balance: i32)
//@ req Account_balance_(my_account, _);
//@ ens true;
{
   (*my_account).balance = new_balance;
   //@ leak Account_balance(my_account, _);
}
```

Function Account :: set_balance now verifies, and VeriFast attempts to verify function main. It complains that it cannot deallocate the Account struct instance because it does not have permission to access the balance field. Indeed, the symbolic heap contains the alloc_block_Account chunk but not the Account_balance chunk. What happened to it? Let's find out by stepping through the symbolic execution path. Select the fourth step. The alloc statement is about to be executed and the symbolic heap is empty. Select the next step. The alloc statement has added the Account_balance_ chunk and the alloc_block_Account chunk.

The if statement has no effect.

We then arrive at the call of Account::set_balance. You will notice that this execution step has two sub-steps, labeled "Consuming assertion" and "Producing assertion". The verification of a function call consists of *consuming* the function's precondition and then *producing* the function's postcondition. The precondition and the postcondition are *assertions*, i.e., expressions that may include heap chunks in addition to ordinary logic. Consuming the precondition means passing the heap chunks required by the function to the function, thus removing them from the symbolic heap. Producing the postcondition means receiving the heap chunks offered by the function when it returns, thus adding them to the symbolic heap.

Selecting the "Consuming assertion" step changes the layout of the VeriFast window (see Figure 9). The source code pane is split into two parts. The upper part is used to display the contract of the function being called, while the lower part is used to display the function being verified. (Since in this example the function being called is so close to the function being verified, it is likely to be shown in the lower part as well.) The call being verified is shown on a green background. The part of the contract being consumed or produced is shown on a yellow background. If you move from the "Consuming assertion" step to the "Producing assertion" step, you notice that the "Consuming assertion" step removes the Account_balance_ chunk from the symbolic heap. Conceptually, it is now in use by the Account::set_balance function while the main function waits for this function to return. Since function Account::set_balance's postcondition does not mention any heap chunks, the "Producing assertion" step does not add anything to the symbolic heap.

It is now clear why VeriFast complained that Account::set_balance leaked heap chunks: since the function did not return the Account_balance chunk to its caller, the chunk was lost and the field could never be accessed again. VeriFast considers this an error since it is usually not the intention of the programmer; furthermore, if too many memory locations are leaked, the program will run out of memory.

It is now also clear how to fix the error: we must specify in the postcondition of function Account::set_balance that the function must hand back the Account_balance chunk to its caller.

```
unsafe fn set_balance(my_account: *mut Account, new_balance: i32)
//@ req Account_balance_(my_account, _);
//@ ens Account_balance(my_account, new_balance);
{
   (*my_account).balance = new_balance;
```

the former (generic chunk) syntax because it causes VeriFast to take into account the field chunk-specific information that there is at most one chunk for a given field, and that the field's value is within the limits of its type. However, for didactical reasons, in this tutorial we initially use the generic chunk syntax so that the chunks written in the annotations and the heap chunks shown in the VeriFast IDE look the same.

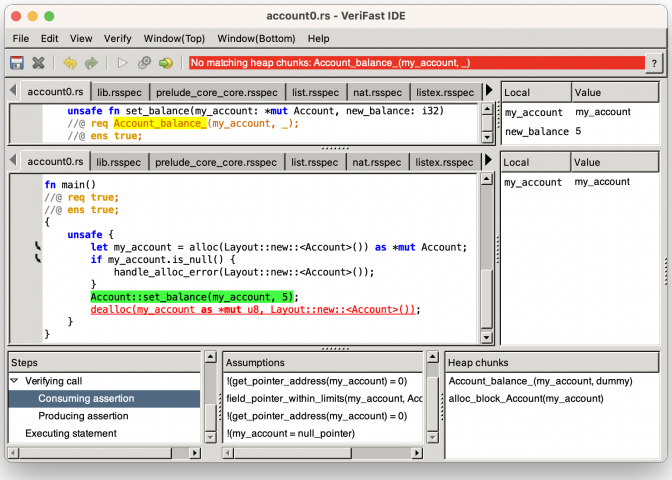

Figure 9: When stepping through a function call, VeriFast shows both the call site (in green, in the lower pane) and the callee's contract (in yellow, in the upper pane).

```
}
```

This eliminates the leak error message and the error at the *dealloc* statement. The program now verifies. Notice that we refer to the new$_{\text{b}}$alance parameter in the position where the value of the field belongs; this means that the value of the field when the function returns must be equal to the value of the parameter.

Exercise 2 *Now factor out the creation and the disposal of the* Account *struct instance into separate functions as well. The creation function should initialize the balance to zero. Note: if you need to mention multiple heap chunks in an assertion, separate them using the* separating conjunction `&*&` *(ampersand-star-ampersand). Also, you can refer to a function's return value in its postcondition by the name* `result`*.*

5 Patterns

Now, let's add a function that returns the current balance, and let's test it in the main function. Here's our first attempt:

```
impl Account {

   unsafe fn get_balance(my_account: *mut Account) -> i32
   //@ req Account_balance(my_account, _);
   //@ ens Account_balance(my_account, _);
   {
      (*my_account).balance
   }

}

fn main()
//@ req true;
//@ ens true;
{
```

```
    unsafe {
        let my_account = Account::create();
        Account::set_balance(my_account, 5);
        let b = Account::get_balance(my_account);
        std::hint::assert_unchecked(b == 5);
        Account::dispose(my_account);
    }
}
```

The new function verifies successfully, but VeriFast complains that it cannot prove the condition `b == 5`. When VeriFast is asked to check a condition, it first translates the condition to a logical formula, by replacing each variable by its symbolic value. We can see in the symbolic store, displayed in the Locals pane, that the symbolic value of variable `b` is the logical symbol `b`. Therefore, the resulting logical formula is $b = 5$. VeriFast then attempts to derive this formula from the *path condition*, i.e., the formulae shown in the Assumptions pane. Since the only assumption in this case is `true`, VeriFast cannot prove the condition.

The problem is that the postcondition of function `Account::get_balance` does not specify the function's return value. It does not state that the return value is equal to the value of the `balance` field when the function is called. To fix this, we need to be able to assign a name to the value of the balance field when the function is called. We can do so by replacing the underscore in the precondition by the *pattern* `?theBalance`. This causes the name `theBalance` to be bound to the value of the `balance` field. We can then use this name in the postcondition to specify the return value using an equality condition. A function's return value is available in the function's postcondition under the name `result`. Logical conditions and heap chunks in an assertion must be separated using the separating conjunction `&*&`.

```
unsafe fn get_balance(my_account: *mut Account) -> i32
//@ req Account_balance(my_account, ?theBalance);
//@ ens Account_balance(my_account, theBalance) &*& result == theBalance;
{
    (*my_account).balance
}
```

Notice that we use the `theBalance` name also to specify that the function does not modify the value of the `balance` field, by using the name again in the field value position in the postcondition.

The program now verifies. Indeed, if we use the **Run to cursor** command to run to the *assert* statement, we see that the assumption `b = 5` has appeared in the Assumptions pane. If we step up, we see that it was added when the equality condition in function `Account::get_balance`'s postcondition was produced. If we step up further, we see that the variable `theBalance` was added to the upper Locals pane when the field chunk assertion was consumed, and bound to value 5. It was bound to value 5 because that was the value found in the symbolic heap. When verifying a function call, the upper Locals pane is used to evaluate the contract of the function being called. It initially contains the bindings of the function's parameters to the arguments specified in the call; additional bindings appear as patterns are encountered in the contract. The assumption `b = 5` is the logical formula obtained by evaluating the equality condition `result == theBalance` under the symbolic store shown in the upper Locals pane.

Exercise 3 *Add a function that deposits a given amount into an account. Verify the following* `main` *function.*

```
fn main()
//@ req true;
//@ ens true;
{
    unsafe {
        let my_account = Account::create();
        Account::set_balance(my_account, 5);
        Account::deposit(my_account, 10);
        let b = Account::get_balance(my_account);
```

```
      std::hint::assert_unchecked(b == 15);
      Account::dispose(my_account);
   }
}
```

Note: VeriFast checks for arithmetic overflow. For now, disable this check in the ***Verify*** *menu.*

Exercise 4 *Add a field* `limit` *to struct* `Account` *that specifies the minimum balance of the account. (It is typically either zero or a negative number.) The limit is specified at creation time. Further add a function to withdraw a given amount from an account. The function must respect the limit; if withdrawing the requested amount would violate the limit, then the largest amount that can be withdrawn without violating the limit is withdrawn. The function returns the amount actually withdrawn as its return value. You will need to use conditional expressions* `if condition { value1 } else { value2 }`*. Remove function* `Account::set_balance`*. Use the shorthand notation for field chunks:* `(*my_account).balance |-> value`*. Verify the following* `main` *function.*

```
fn main()
//@ req true;
//@ ens true;
{
   unsafe {
      let my_account = Account::create(-100);
      Account::deposit(my_account, 200);
      let w1 = Account::withdraw(my_account, 50);
      std::hint::assert_unchecked(w1 == 50);
      let b1 = Account::get_balance(my_account);
      std::hint::assert_unchecked(b1 == 150);
      let w2 = Account::withdraw(my_account, 300);
      std::hint::assert_unchecked(w2 == 250);
      let b2 = Account::get_balance(my_account);
      std::hint::assert_unchecked(b2 == -100);
      Account::dispose(my_account);
   }
}
```

6 Predicates

We continue with the program obtained in Exercise 4. We observe that the contracts are becoming rather long. Furthermore, if we consider the account "class" and the main function to be in different modules, then the internal implementation details of the account module are exposed to the main function. We can achieve more concise contracts as well as information hiding by introducing a *predicate* to describe an `account` struct instance in the function contracts.

```
/*@
pred Account_pred(my_account: *mut Account, theLimit: i32, theBalance: i32) =
   (*my_account).limit |-> theLimit &*& (*my_account).balance |-> theBalance &*&
   alloc_block_Account(my_account);
@*/
```

A predicate is a named, parameterized assertion. Furthermore, it introduces a new type of heap chunk. An `Account_pred` heap chunk bundles an `Account_limit` heap chunk, an `Account_balance` heap chunk, and an `alloc_block_Account` heap chunk into one.

Let's use this predicate to rewrite the contract of the deposit function. Here's a first attempt:

```
unsafe fn deposit(my_account: *mut Account, amount: i32)
//@ req Account_pred(my_account, ?limit, ?balance) &*& 0 <= amount;
//@ ens Account_pred(my_account, limit, balance + amount);
{
   (*my_account).balance += amount;
}
```

This function does not verify. The update of the `balance` field cannot be verified since there is no `Account_balance` heap chunk in the symbolic heap. There is only an `Account_pred` heap chunk. The `Account_pred` heap chunk encapsulates the `Account_balance` heap chunk, but VeriFast does not "un-bundle" the `Account_pred` predicate automatically. We must instruct VeriFast to un-bundle predicate heap chunks by inserting an **open** ghost statement:

```
unsafe fn deposit(my_account: *mut Account, amount: i32)
//@ req Account_pred(my_account, ?limit, ?balance) &*& 0 <= amount;
//@ ens Account_pred(my_account, limit, balance + amount);
{
   //@ open Account_pred(my_account, limit, balance);
   (*my_account).balance += amount;
}
```

The assignment now verifies, but now VeriFast is stuck at the postcondition. It complains that it cannot find the `Account_pred` heap chunk that it is supposed to hand back to the function's caller. The `Account_pred` chunk's constituent chunks are present in the symbolic heap, but VeriFast does not automatically bundle them up into an `Account_pred` chunk. We must instruct VeriFast to do so using a **close** ghost statement:

```
unsafe fn deposit(my_account: *mut Account, amount: i32)
//@ req Account_pred(my_account, ?limit, ?balance) &*& 0 <= amount;
//@ ens Account_pred(my_account, limit, balance + amount);
{
   //@ open Account_pred(my_account, limit, balance);
   (*my_account).balance += amount;
   //@ close Account_pred(my_account, limit, balance + amount);
}
```

The function now verifies. However, the main function does not, since the call of `Account::deposit` expects an `Account_pred` heap chunk.

Exercise 5 *Rewrite the remaining contracts using the* `Account_pred` *predicate. Insert* **`open`** *and* **`close`** *statements as necessary.*

7 Recursive Predicates

In the previous section, we introduced predicates for the sake of conciseness and information hiding. However, there is an even more compelling need for predicates: they are the only way you can describe unbounded-size data structures in VeriFast. Indeed, in the absence of predicates, the number of memory locations described by an assertion is linear in the length of the assertion. This limitation can be overcome through the use of recursive predicates, i.e., predicates that invoke themselves.

Exercise 6 *Implement a stack of integers using a singly linked list data structure: implement* `Stack` *associated functions* `create`*,* `push`*,* `pop`*, and* `dispose`*. In order to be able to specify the precondition of* `pop`*, your predicate will need to have a parameter that specifies the number of elements in the stack. Function* `dispose` *may be called only on an empty stack. Do not attempt to specify the contents of the stack; this is not possible with the annotation elements we have seen. You will need to use conditional assertions:*

`if` condition { assertion1 } `else` { assertion2 }. Note: VeriFast does not allow the use of field dereferences in `open` statements. If you want to use the value of a field in an `open` statement, you must first store the value in a local variable. Verify the following main function:

```
fn main()
//@ req true;
//@ ens true;
{
   unsafe {
      let s = Stack::create();
      Stack::push(s, 10);
      Stack::push(s, 20);
      Stack::pop(s);
      Stack::pop(s);
      Stack::dispose(s);
   }
}
```

Now, let's extend the solution to Exercise 6 on page 70 with a `Stack::is_empty` function. Recall the predicate definitions:

```
pred Nodes(node: *mut Node, count: i32) =
   if node == 0 {
      count == 0
   } else {
      0 < count &*&
      (*node).next |-> ?next &*& (*node).value |-> ?value &*&
      alloc_block_Node(node) &*& Nodes(next, count - 1)
   };

pred Stack(stack: *mut Stack, count: i32) =
   (*stack).head |-> ?head &*& alloc_block_Stack(stack) &*& 0 <= count &*& Nodes(head, count);
```

Here's a first stab at a `Stack::is_empty` function:

```
unsafe fn is_empty(stack: *mut Stack) -> bool
//@ req Stack(stack, ?count);
//@ ens Stack(stack, count) &*& result == (count == 0);
{
   //@ open Stack(stack, count);
   let result = (*stack).head.is_null();
   //@ close Stack(stack, count);
   result
}
```

The function does not verify. VeriFast complains that it cannot prove the condition `result == (count == 0)` in the postcondition. Indeed, if we look at the assumptions in the Assumptions pane, they are insufficient to prove this condition. The problem is that the relationship between the value of the head pointer and the number of nodes is hidden inside the `Nodes` predicate. We need to open the predicate, so that the information is added to the assumptions. Of course, we then need to close it again so that we can close the `Stack` predicate.

```
unsafe fn is_empty(stack: *mut Stack) -> bool
//@ req Stack(stack, ?count);
//@ ens Stack(stack, count) &*& result == (count == 0);
{
```

```
    //@ open Stack(stack, count);
    let head = (*stack).head;
    //@ open Nodes(head, count);
    let result = head.is_null();
    //@ close Nodes(head, count);
    //@ close Stack(stack, count);
    result
}
```

The function now verifies. What happens exactly is the following. When VeriFast executes the open statement, it produces the conditional assertion in the body of the `Nodes` predicate. This causes it to perform a *case split*. This means that the rest of the function is verified twice: once under the assumption that the condition is true, and once under the assumption that the condition is false. In other words, the execution path splits into two execution paths, or two *branches*. On both branches, the postcondition can now be proved easily: on the first branch, we get the assumptions $\mathsf{head} = \mathsf{null\_pointer}$ and $\mathsf{count} = 0$, and on the second branch we get $\mathsf{head} \neq \mathsf{null\_pointer}$ and $0 < \mathsf{count}$.

Exercise 7 *Modify function `Stack::dispose` so that it works even if the stack still contains some elements. Use a recursive helper function.*[11]

Notice that VeriFast performs a case split when verifying an **`if`** statement.

Exercise 8 *Add a function `Stack::get_sum` that returns the sum of the values of the elements on the stack. Use a recursive helper function. The contract need not specify the return value (since we did not see how to do that yet).*

8 Loops

In Exercise 7, we implemented `Stack::dispose` using a recursive function. However, this is not an optimal implementation. If our data structure contains very many elements, we may create too many activation records and overflow the call stack. It is more optimal to implement the function using a loop. Here's a first attempt:[12]

```
unsafe fn dispose(stack: *mut Stack)
//@ req Stack(stack, _);
//@ ens true;
{
    //@ open Stack(stack, _);
    let mut n = (*stack).head;
    loop {
        //@ open Nodes(n, _);
        if n.is_null() {
            break;
        }
        let next = (*n).next;
        dealloc(n as *mut u8, Layout::new::<Node>());
        n = next;
    }
    dealloc(stack as *mut u8, Layout::new::<Stack>());
}
```

[11] Warning: VeriFast does not verify termination; it does not complain about infinite recursion or infinite loops. That is still your own responsibility.

[12] A **`while`** loop would perhaps be more natural here. Unfortunately, however, VeriFast for Rust supports only **`loop`** loops for now.

This function does not verify. VeriFast complains at the loop because the loop does not specify a loop invariant. VeriFast needs a loop invariant so that it can verify an arbitrary sequence of loop iterations by verifying the loop body once, starting from a symbolic state that represents the start of an arbitrary loop iteration (not just the first iteration).

Specifically, VeriFast verifies a loop as follows:

- First, it consumes the loop invariant.
- Then, it removes the remaining heap chunks from the heap (but it remembers them).
- Then, it assigns a fresh logical symbol to each local variable that is modified in the loop body.
- Then, it produces the loop invariant.
- Then, it verifies the loop body. If, during verification of the loop body, an execution path **break**s out of the loop, at that point the heap chunks that were removed from the symbolic heap when the loop was entered are added back and symbolic execution continues after the loop.
- Then, it consumes the loop invariant.
- And then finally it checks for leaks. After this step this execution path is finished.

Notice that this means that the loop can access only those heap chunks that are mentioned in the loop invariant.

The correct loop invariant for the above function is as follows:

```
unsafe fn dispose(stack: *mut Stack)
//@ req Stack(stack, _);
//@ ens true;
{
   //@ open Stack(stack, _);
   let mut n = (*stack).head;
   loop {
      //@ inv Nodes(n, _);
      //@ open Nodes(n, _);
      if n.is_null() {
         break;
      }
      let next = (*n).next;
      dealloc(n as *mut u8, Layout::new::<Node>());
      n = next;
   }
   dealloc(stack as *mut u8, Layout::new::<Stack>());
}
```

You can inspect the execution of the loop by commenting out the first `dealloc` statement. This will cause the leak check at the end of the symbolic execution of the loop to fail. Find the step where the loop invariant is consumed for the first time. Notice that after this step, the value of variable `n` changes from `head0` to `n`, and all chunks are removed from the symbolic heap.

Exercise 9 *Specify and implement function* `Stack::popn`*, which pops a given number of elements from the stack (and does not return a result). You may call* `Stack::pop` *internally. Use a* **`loop`** *loop. Notice that your loop invariant must not only enable verification of the loop body, but must also maintain the relationship between the current state and the initial state, sufficiently to prove the postcondition. This often means that you should not overwrite the function parameter values, since you typically need the original values in the loop invariant.*

9 Inductive Datatypes

Let's return to our initial annotated stack implementation (the solution to Exercise 6). The annotations do not specify full functional correctness. In particular, the contract of function `Stack::pop` does not specify the function's return value. As a result, using these annotations, we cannot verify the following main function:

```
fn main()
//@ req true;
//@ ens true;
{
   unsafe {
      let s = Stack::create();
      Stack::push(s, 10);
      Stack::push(s, 20);
      let result1 = Stack::pop(s);
      //@ assert result1 == 20;
      let result2 = Stack::pop(s);
      //@ assert result2 == 10;
      Stack::dispose(s);
   }
}
```

In order to verify this main function, instead of tracking just the number of elements in the stack, we need to track the values of the elements as well. In other words, we need to track the precise sequence of elements currently stored by the stack. We can represent a sequence of integers using an *inductive datatype* `i32s`:

```
inductive i32s = i32s_nil | i32s_cons(i32, i32s);
```

This declaration declares a type `i32s` with two *constructors* `i32s_nil` and `i32s_cons`. `i32s_nil` represents the empty sequence. `i32s_cons` constructs a nonempty sequence given the *head* (the first element) and the *tail* (the remaining elements). For example, the sequence 1,2,3 can be written as

```
i32s_cons(1, i32s_cons(2, i32s_cons(3, i32s_nil)))
```

Exercise 10 *Replace the* `count` *parameter of the* `Nodes` *and* `Stack` *predicates with a* `values` *parameter of type* `i32s` *and update the predicate bodies. Further update the functions* `Stack::create`*,* `Stack::push`*, and* `Stack::dispose`*. Don't worry about* `Stack::pop` *for now.*

10 Fixpoint Functions

How should we update the annotations for function `Stack::pop`? We need to refer to the tail of the current sequence in the postcondition and in the close statement. Furthermore, in order to specify the return value, we need to refer to the head of the current sequence. We can do so using *fixpoint functions*:

```
fix i32s_head(values: i32s) -> i32 {
   match values {
      i32s_nil => 0,
      i32s_cons(value, values0) => value,
   }
}

fix i32s_tail(values: i32s) -> i32s {
   match values {
```

```
      i32s_nil => i32s_nil,
      i32s_cons(value, values0) => values0,
   }
}
```

Notice that we can use match expressions on inductive datatypes. There must be exactly one case for each constructor. In a case for a constructor that takes parameters, the specified parameter names are bound to the corresponding constructor argument values that were used when the value was constructed. The body of a fixpoint function must be a match expression on one of the function's parameters (called the *inductive parameter*). Furthermore, the body of each case must be an expression (not a block).

In contrast to regular Rust functions, a fixpoint function may be used in any place where an expression is expected in an annotation. This means we can use the `i32s_head` and `i32s_tail` functions to adapt function `Stack::pop` to the new predicate definitions, and to specify the return value.

Exercise 11 *Do so.*

We have now specified full functional correctness of our stack implementation, and VeriFast can now verify the new main function.

Exercise 12 *Add a Rust function `Stack::get_sum` that returns the sum of the elements of a given stack. Implement it using a recursive helper function `get_nodes_sum`. Specify the new Rust functions using a recursive fixpoint function `i32s_sum`. Remember to turn off arithmetic overflow checking in the **Verify** menu. Verify the following main function:*

```
fn main()
//@ req true;
//@ ens true;
{
   unsafe {
      let s = Stack::create();
      Stack::push(s, 10);
      Stack::push(s, 20);
      let sum = Stack::get_sum(s);
      //@ assert sum == 30;
      let result1 = Stack::pop(s);
      //@ assert result1 == 20;
      let result2 = Stack::pop(s);
      //@ assert result2 == 10;
      Stack::dispose(s);
   }
}
```

VeriFast supports recursive fixpoint functions. It enforces that they always terminate by allowing only direct recursion and by requiring that the value of the inductive parameter of a recursive call is a constructor argument of the value of the inductive parameter of the caller.

11 Lemmas

Note: Sections 25 to 28 offer an alternative introduction to recursive predicates. If you are not yet comfortable with recursive predicates, refer to these sections before starting the present section.

Let's return to our initial annotated stack implementation (the solution to Exercise 6). Let's add a `Stack::get_count` function that returns the number of elements in the stack, implemented using a loop:

```
unsafe fn get_count(stack: *mut Stack) -> i32
//@ req Stack(stack, ?count);
//@ ens Stack(stack, count) &*& result == count;
{
   //@ open Stack(stack, count);
   let head = (*stack).head;
   let mut n = head;
   let mut i = 0;
   loop {
      //@ inv true;
      if n.is_null() {
         break;
      }
      n = (*n).next;
      i += 1;
   }
   //@ close Stack(stack, count);
   i
}
```

Clearly, the loop invariant **true** will not do. What should it be? We need to express that `n` is somewhere inside our sequence of nodes. One way to do this is by working with *linked list segments* (or *list segments* for short). We state in the loop invariant that there is a list segment from `head` to `n` and a separate list segment from `n` to 0:

```
//@ inv lseg(head, n, i) &*& lseg(n, 0, count - i);
```

We can define the `lseg` predicate as follows:

```
pred lseg(first: *mut Node, last: *mut Node, count: i32) =
   if first == last {
      count == 0
   } else {
      0 < count &*& first != 0 &*&
      (*first).value |-> ?value &*& (*first).next |-> ?next &*& alloc_block_Node(first) &*&
      lseg(next, last, count - 1)
   };
```

We are not done yet. We need to establish the loop invariant when first entering the loop. That is, we need to establish

```
lseg(head, head, 0) &*& lseg(head, 0, count)
```

The first conjunct is easy: it is empty, so we can just close it. The second conjunct requires that we rewrite the `Nodes(head, count)` chunk into an equivalent `lseg(head, 0, count)` chunk. Notice that we cannot do so using a statically bounded number of open and close operations. We need to use either a loop or a recursive function. Since VeriFast does not allow loops inside annotations, we will use a recursive function. We will not use a regular Rust function, since the function has no purpose at run time; furthermore, we cannot use a fixpoint function, since the latter cannot include open or close statements. VeriFast supports a third kind of function, called *lemma functions*. They are just like regular Rust functions, except that they may not perform field assignments or call regular functions, and they must always terminate. It follows that calling them has no effect at run time. Their only purpose is to transform some heap chunks into an equivalent set of heap chunks, i.e. different heap chunks that represent the same actual memory values. In contrast with fixpoint functions, they may be called only through separate call statements, not from within expressions.

VeriFast checks termination of a lemma function by allowing only direct recursion and by checking each recursive call as follows: first, if, after consuming the precondition of the recursive call, a field chunk

is left in the symbolic heap, then the call is allowed. This is induction on heap size. Otherwise, if the body of the lemma function is a match statement on a parameter whose type is an inductive datatype, then the argument for this parameter in the recursive call must be a constructor argument of the caller's argument for the same parameter. This is induction on an inductive parameter. Finally, if the body of the lemma function is not such a match statement, then the first heap chunk consumed by the precondition of the callee must have been obtained from the first heap chunk consumed by the precondition of the caller through one or more open operations. This is induction on the derivation of the first conjunct of the precondition.

We can transform a `Nodes` chunk into an `lseg` chunk using the following lemma function:

```
lem Nodes_to_lseg_lemma(first: *mut Node)
   req Nodes(first, ?count);
   ens lseg(first, 0, count);
{
   open Nodes(first, count);
   if first != 0 {
      Nodes_to_lseg_lemma((*first).next);
   }
   close lseg(first, 0, count);
}
```

Notice that this lemma is accepted since it performs induction on heap size.

Exercise 13 *We will need the inverse operation as well. Write a lemma function `lseg_to_Nodes_lemma` that takes an `lseg` chunk that ends in a null pointer and transforms it into a `Nodes` chunk.*

The `Stack::get_count` function now looks as follows:

```
unsafe fn get_count(stack: *mut Stack) -> i32
//@ req Stack(stack, ?count);
//@ ens Stack(stack, count) &*& result == count;
{
   //@ open Stack(stack, count);
   let head = (*stack).head;
   //@ Nodes_to_lseg_lemma(head);
   let mut n = head;
   let mut i = 0;
   //@ close lseg(head, head, 0);
   loop {
      //@ inv lseg(head, n, i) &*& lseg(n, 0, count - i);
      if n.is_null() {
         break;
      }
      //@ open lseg(n, 0, count - i);
      n = (*n).next;
      i += 1;
      // Error!
   }
   //@ open lseg(0, 0, _);
   //@ lseg_to_Nodes_lemma(head);
   //@ close Stack(stack, count);
   i
}
```

We are almost done. All that is left to do is to fix the error that we get at the end of the loop body. At that point, we have the following chunks:

```
lseg(head, ?old_n, i - 1) &*& (*old_n).value |-> ?value &*& (*old_n).next |-> n &*&
alloc_block_Node(old_n) &*& lseg(n, 0, count - i)
```

We need to transform these into the following:

```
lseg(head, n, i) &*& lseg(n, 0, count - i)
```

That is, we need to append the node at the old value of `n` to the end of the list segment that starts at `head`. We will again need to use a lemma function for this. Here's a first attempt:

```
lem lseg_add_lemma(first: *mut Node)
   req
      lseg(first, ?last, ?count) &*& last != 0 &*& (*last).value |-> ?value &*&
      (*last).next |-> ?next &*& alloc_block_Node(last);
   ens lseg(first, next, count + 1);
{
   open lseg(first, last, count);
   if first == last {
      close lseg(next, next, 0);
   } else {
      lseg_add_lemma((*first).next);
   }
   close lseg(first, next, count + 1);
}
```

VeriFast complains while trying to perform the final close operation, on the path where `first` equals `last`. It has assumed that `first` equals `next` and it cannot prove that `count + 1` equals zero. In our scenario, `first` never equals `next`, since `first` always points to a node of the stack, and `next` either points to a separate node or equals 0. However, the precondition of our lemma function does not express this. In order to include this information, we need to require the list segment from `next` to 0 as well. By opening and closing this list segment before we perform the final close operation, we obtain the information that `first` and `next` are distinct. Specifically, whenever VeriFast produces a field chunk, and another field chunk for the same field is already in the symbolic heap, it adds an assumption stating that the field chunks belong to distinct struct instances.

We obtain the following `lseg_add_lemma` and `Stack::get_count` functions:

```
/*@
lem lseg_add_lemma(first: *mut Node)
   req lseg(first, ?last, ?count) &*& last != 0 &*& (*last).value |-> ?last_value &*&
      (*last).next |-> ?next &*& alloc_block_Node(last) &*& lseg(next, 0, ?count0);
   ens lseg(first, next, count + 1) &*& lseg(next, 0, count0);
{
   open lseg(first, last, count);
   if first == last {
      close lseg(next, next, 0);
   } else {
      lseg_add_lemma((*first).next);
   }
   open lseg(next, 0, count0);
   close lseg(next, 0, count0);
   close lseg(first, next, count + 1);
}
@*/

impl Stack {
   unsafe fn get_count(stack: *mut Stack) -> i32
```

```
    //@ req Stack(stack, ?count);
    //@ ens Stack(stack, count) &*& result == count;
    {
        //@ open Stack(stack, count);
        let head = (*stack).head;
        //@ Nodes_to_lseg_lemma(head);
        let mut n = head;
        let mut i = 0;
        //@ close lseg(head, head, 0);
        loop {
            //@ inv lseg(head, n, i) &*& lseg(n, 0, count - i);
            if n.is_null() {
                break;
            }
            //@ open lseg(n, 0, count - i);
            n = (*n).next;
            i += 1;
            //@ lseg_add_lemma(head);
        }
        //@ open lseg(0, 0, _);
        //@ lseg_to_Nodes_lemma(head);
        //@ close Stack(stack, count);
        i
    }
}
```

These now verify.

Exercise 14 *Verify the following function. You'll need an extra lemma.*

```
unsafe fn push_all(stack: *mut Stack, other: *mut Stack)
//@ req Stack(stack, ?count) &*& Stack(other, ?count0);
//@ ens Stack(stack, count0 + count);
{
    let head0 = (*other).head;
    dealloc(other as *mut u8, Layout::new::<Stack>());
    let mut n = head0;
    if !n.is_null() {
        loop {
            if (*n).next.is_null() {
                break;
            }
            n = (*n).next;
        }
        (*n).next = (*stack).head;
        (*stack).head = head0;
    }
}
```

Exercise 15 *Implement, specify, and verify a function* `Stack::reverse` *that performs in-place reversal of a stack, i.e., without any memory allocation. Verify full functional correctness (see Section 9). You'll need to define additional fixpoints and lemmas.*

12 Function Pointers

Let's write a function that takes a stack and removes those elements that do not satisfy a given predicate:

```
type I32Predicate = unsafe fn(i32) -> bool;

unsafe fn filter_nodes(n: *mut Node, p: I32Predicate) -> *mut Node
//@ req Nodes(n, _);
//@ ens Nodes(result, _);
{
   if n.is_null() {
      return std::ptr::null_mut();
   } else {
      //@ open Nodes(n, _);
      let keep = p((*n).value);
      let next;
      if keep {
         next = filter_nodes((*n).next, p);
         //@ open Nodes(next, ?count);
         //@ close Nodes(next, count);
         (*n).next = next;
         //@ close Nodes(n, count + 1);
         return n;
      } else {
         next = (*n).next;
         dealloc(n as *mut u8, Layout::new::<Node>());
         let result = filter_nodes(next, p);
         return result;
      }
   }
}

impl Stack {
   unsafe fn filter(stack: *mut Stack, p: I32Predicate)
   //@ req Stack(stack, _);
   //@ ens Stack(stack, _);
   {
      //@ open Stack(stack, _);
      let head = filter_nodes((*stack).head, p);
      //@ assert Nodes(head, ?count);
      (*stack).head = head;
      //@ open Nodes(head, count);
      //@ close Nodes(head, count);
      //@ close Stack(stack, count);
   }
}

unsafe fn neq_20(x: i32) -> bool
//@ req true;
//@ ens true;
{
   x != 20
}
```

```
fn main() {
   unsafe {
      let s = Stack::create();
      Stack::push(s, 10);
      Stack::push(s, 20);
      Stack::filter(s, neq_20);
      Stack::dispose(s);
   }
}
```

This program does not verify. VeriFast does not know what contract to use to verify the call of `p` in function `filter_nodes`. Specifically, when symbolically executing a call of a function pointer `p`, VeriFast looks for a *function type chunk* `is_T(p)` in the symbolic heap that asserts that `p` satisfies *VeriFast function type* `T`.

We therefore need to define a VeriFast function type:

```
/*@
fn_type I32Predicate() = unsafe fn(x: i32) -> bool;
   req true;
   ens true;
@*/
```

It is like a Rust type definition, except that it also associates a precondition and a postcondition with the type.

We must update the precondition of functions `filter_nodes` and `Stack::filter` to assert an appropriate function type chunk:

```
unsafe fn filter_nodes(n: *mut Node, p: I32Predicate) -> *mut Node
//@ req Nodes(n, _) &*& [_]is_I32Predicate(p);
//@ ens Nodes(result, _);

impl Stack {
   unsafe fn filter(stack: *mut Stack, p: I32Predicate)
   //@ req Stack(stack, _) &*& [_]is_I32Predicate(p);
   //@ ens Stack(stack, _);
```

(The `[_]` prefix denotes duplicability of the chunk; it will be explained in Section 24.)

Finally, before calling `Stack::filter` in the main function, we must *produce* the function type chunk, using a **`produce_fn_ptr_chunk`** `T(f)()(params) { proof }` ghost command. This command requires a sequence of ghost commands `proof`, exactly one of which must be a `call();` statement, that proves that the function `f` satisfies the function type's contract. Specifically, VeriFast checks the proof by first producing the function type's precondition, then symbolically executing `proof`, where `call();` is symbolically executed like a call of `f`, and finally consumes the function type's postcondition:

```
fn main()
//@ req true;
//@ ens true;
{
   unsafe {
      let s = Stack::create();
      Stack::push(s, 10);
      Stack::push(s, 20);
      /*@
      produce_fn_ptr_chunk I32Predicate(neq_20)()(x) {
         call();
```

```
      }
      @*/
      Stack::filter(s, neq_20);
      Stack::dispose(s);
   }
}
```

Exercise 16 *Put all the pieces together.*

The state of our `Stack::filter` function is unsatisfactory in two ways: firstly, the `I32Predicate` function cannot read any memory locations, since its precondition does not require any heap chunks; it follows that you cannot filter all elements that are greater than some user-provided value, for example. This will be solved using parameterized function types in Section 15. Secondly, the implementation re-assigns each next pointer, even if only a few elements are removed. This will be solved using by-reference parameters in Section 13.

13 By-Reference Parameters

Here's an alternative implementation of the `Stack::filter` function from Section 12. Instead of re-assigning each next pointer, it passes a pointer to the next pointer and only re-assigns it when it changes.

```
unsafe fn filter_nodes(n: *mut *mut Node, p: I32Predicate) {
   if !(*n).is_null() {
      let keep = p((**n).value);
      if keep {
         filter_nodes(&raw mut (**n).next, p);
      } else {
         let next_ = (**n).next;
         dealloc(*n as *mut u8, Layout::new::<Node>());
         *n = next_;
         filter_nodes(n, p);
      }
   }
}

impl Stack {
   unsafe fn filter(stack: *mut Stack, p: I32Predicate) {
      filter_nodes(&raw mut (*stack).head, p);
   }
}
```

In this program, we are effectively passing the pointer to the current node to function `filter_nodes` by reference. Inside function `filter_nodes`, we dereference `n` to obtain the pointer to the current node. VeriFast treats pointer dereferences in a way similar to field dereferences. However, instead of looking in the symbolic heap for a field chunk, it looks for a generic `points_to` chunk. Predicate `points_to` is defined in `prelude_core.rsspec` as follows:

```
pred points_to<T>(p: *T; v: T);
```

(We will discuss the meaning of the semicolon later; for now, just read it like a comma.) Like in the case of a field chunk, the first argument is the address of the variable, and the second argument is the current value of the variable.

It follows that the following is a valid contract for function `filter_nodes`:

```
unsafe fn filter_nodes(n: *mut *mut Node, p: I32Predicate)
//@ req points_to(n, ?node) &*& Nodes(node, _) &*& [_]is_I32Predicate(p);
//@ ens points_to(n, ?node0) &*& Nodes(node0, _);
```

In order to be able to call `filter_nodes` from `Stack::filter`, we must produce a `points_to` chunk. Specifically, we must transform the `Stack_head` chunk into a `points_to` chunk. We do this simply by opening the `Stack_head` chunk. To turn the pointer chunk back into a `Stack_head` chunk, we simply close the `Stack_head` chunk again:

```
unsafe fn filter(stack: *mut Stack, p: I32Predicate)
//@ req Stack(stack, _) &*& [_]is_I32Predicate(p);
//@ ens Stack(stack, _);
{
   //@ open Stack(stack, _);
   //@ open Stack_head(stack, _);
   filter_nodes(&raw mut (*stack).head, p);
   //@ close Stack_head(stack, ?head);
   //@ open Nodes(head, ?count);
   //@ close Nodes(head, count);
   //@ close Stack(stack, count);
}
```

Exercise 17 *Verify function* `filter_nodes`*.*

Note: in the same way that field chunk assertions can be written more elegantly using a points-to syntax, generic `points_to` chunk assertions can be written using a points-to syntax as well. For example, we can also write the spec of `filter_nodes` as follows:

```
unsafe fn filter_nodes(n: *mut *mut Node, p: I32Predicate)
//@ req *n |-> ?node &*& Nodes(node, _) &*& [_]is_I32Predicate(p);
//@ ens *n |-> ?node0 &*& Nodes(node0, _);
```

14 Arithmetic Overflow

Consider the following program:

```
fn thirty_thousand() -> i16
//@ req true;
//@ ens result == 30000;
{ 30_000 }

fn main()
//@ req true;
//@ ens true;
{
  let a = thirty_thousand();
  let b = thirty_thousand();
  let c: i16 = a + b;
  unsafe { std::hint::assert_unchecked(b < c); }
}
```

The addition in function `main` performs arithmetic overflow: the mathematical result of the addition is outside the limits of the type. Rust considers arithmetic overflow to be an error: in a debug build, the addition panics. (In a release build, the addition wraps around; in this example, this leads to undefined behavior due to the failure of the subsequent unchecked assert.)

VeriFast, too, reports arithmetic overflow as an error by default. Indeed, it does not accept the example program. Sometimes, however, it is useful to postpone worrying about arithmetic overflows; for that reason, VeriFast offers the option to disable the *Check arithmetic overflow* setting in the Verify menu. Be aware, however, that this may lead to unsoundness: with this setting disabled, VeriFast simply assumes that no arithmetic overflow occurs and accepts the example program, even though it has undefined behavior.

Note that for the Rust types `usize` and `isize`, the limits depend on the compiler's target architecture. VeriFast assumes the same limits that the Rust compiler uses: when verifying a program on a 64-bit machine, VeriFast assumes that `usize` and `isize` are 64 bits wide.

Now, consider the following function:

```
unsafe fn i16_add(x: *mut i16, y: *mut i16) -> i16
//@ req points_to(x, ?vx) &*& points_to(y, ?vy);
//@ ens points_to(x, vx) &*& points_to(y, vy) &*& result == vx + vy;
{
   let xv = *x;
   let yv = *y;
   xv + yv
}
```

This function performs arithmetic overflow and is rejected by VeriFast. (Note: inside annotations, arithmetic operations are interpreted mathematically, so the expression `vx + vy` in the `ens` clause has the mathematical meaning regardless of the Rust type of this expression; it follows that the values of expressions inside annotations are not necessarily within the limits of their Rust type.)

Let's try to fix this function. Here's a first attempt:

```
unsafe fn i16_add(x: *mut i16, y: *mut i16) -> i16
//@ req points_to(x, ?vx) &*& points_to(y, ?vy);
//@ ens points_to(x, vx) &*& points_to(y, vy) &*& result == vx + vy;
{
   let xv = *x;
   let yv = *y;
   if 0 <= xv {
      if i16::MAX - xv < yv {
         std::process::abort();
      }
   } else {
      if yv < i16::MIN - xv {
         std::process::abort();
      }
   }
   xv + yv
}
```

Even though this function is now correct, VeriFast does not accept it. This is because VeriFast checks that the results of the arithmetic operations are within the bounds of the type, but it does not assume that the original values are within the bounds of the type. These assumptions are not generated automatically in general. In order to generate these assumptions, we must insert `produce_limits` ghost commands into the code. The argument of a `produce_limits` command must be the name of a Rust local variable (i.e., not a local variable declared in an annotation). The following program verifies.

```
unsafe fn i16_add(x: *mut i16, y: *mut i16) -> i16
//@ req points_to(x, ?vx) &*& points_to(y, ?vy);
//@ ens points_to(x, vx) &*& points_to(y, vy) &*& result == vx + vy;
{
   let xv = *x;
```

```
   let yv = *y;
   //@ produce_limits(xv);
   //@ produce_limits(yv);
   if 0 <= xv {
      if i16::MAX - xv < yv {
         std::process::abort();
      }
   } else {
      if yv < i16::MIN - xv {
         std::process::abort();
      }
   }
   xv + yv
}
```

Note: this is a contrived example; in reality, one should of course use Rust's `i16::checked_add` method instead.

Also, the limits of the values of variables *are* produced automatically if the points-to syntax is used to assert the chunks. The following verifies:

```
unsafe fn i16_add(x: *mut i16, y: *mut i16) -> i16
//@ req *x |-> ?vx &*& *y |-> ?vy;
//@ ens *x |-> vx &*& *y |-> vy &*& result == vx + vy;
{
   let xv = *x;
   let yv = *y;
   if 0 <= xv {
      if i16::MAX - xv < yv {
         std::process::abort();
      }
   } else {
      if yv < i16::MIN - xv {
         std::process::abort();
      }
   }
   xv + yv
}
```

15 Parameterized Function Types

Let's go back to the `Stack::filter` function from Section 12. Suppose we want to remove all occurrences of some user-provided value from the stack:

```
type I32Predicate = unsafe fn(data: *mut u8, i32) -> bool;

unsafe fn filter_nodes(n: *mut Node, p: I32Predicate, data: *mut u8) -> *mut Node {
   if n.is_null() {
      std::ptr::null_mut()
   } else {
      let keep = p(data, (*n).value);
      if keep {
         let next = filter_nodes((*n).next, p, data);
         (*n).next = next;
```

```
            n
        } else {
            let next = (*n).next;
            dealloc(n as *mut u8, Layout::new::<Node>());
            let result = filter_nodes(next, p, data);
            result
        }
    }
}

impl Stack {
    unsafe fn filter(stack: *mut Stack, p: I32Predicate, data: *mut u8) {
        let head = filter_nodes((*stack).head, p, data);
        (*stack).head = head;
    }
}

unsafe fn neq_a(data: *mut u8, x: i32) -> bool
{
    let result = x != *(data as *mut i32);
    result
}

unsafe fn read_i32() -> i32
//@ req true;
//@ ens true;
//@ assume_correct
{
    let mut line = String::new();
    std::io::stdin().read_line(&mut line).unwrap();
    line.parse().unwrap()
}

fn main() {
    unsafe {
        let s = Stack::create();
        Stack::push(s, 10);
        Stack::push(s, 20);
        let mut a = read_i32();
        Stack::filter(s, neq_a, &raw mut a as *mut u8);
        Stack::dispose(s);
    }
}
```

How do we adapt the proof? Here's an attempt:

```
/*@

pred neq_a_data(data: *mut u8) = *(data as *mut i32) |-> ?_;

fn_type I32Predicate() = unsafe fn(data: *mut u8, x: i32) -> bool;
    req neq_a_data(data);
    ens neq_a_data(data);
```

```
@*/

impl Stack {
   unsafe fn filter(stack: *mut Stack, p: I32Predicate, data: *mut u8)
   //@ req Stack(stack, _) &*& [_]is_I32Predicate(p) &*& neq_a_data(data);
   //@ ens Stack(stack, _) &*& neq_a_data(data);
   { ... }
}

unsafe fn neq_a(data: *mut u8, x: i32) -> bool
//@ req neq_a_data(data);
//@ ens neq_a_data(data);
{
   //@ open neq_a_data(data);
   let result = x != *(data as *mut i32);
   //@ close neq_a_data(data);
   result
}

fn main()
//@ req true;
//@ ens true;
{
   unsafe {
      let s = Stack::create();
      Stack::push(s, 10);
      Stack::push(s, 20);
      let mut a = read_i32();
      /*@
      produce_fn_ptr_chunk I32Predicate(neq_a)()(data, x) {
         call();
      }
      @*/
      //@ close neq_a_data(&a as *mut u8);
      Stack::filter(s, neq_a, &raw mut a as *mut u8);
      //@ open neq_a_data(&a as *mut u8);
      Stack::dispose(s);
   }
}
```

This proof goes through, but it is not modular: the proof of the stack module depends on the `neq_a_data` predicate which is specific to the particular client program.

The solution is to *parameterize* the `I32Predicate` VeriFast function type by the predicate that describes the data, by declaring a *function type parameter* `data_pred` between the first set of parentheses:

```
/*@
fn_type I32Predicate(data_pred: pred(*mut u8)) = unsafe fn(data: *mut u8, x: i32) -> bool;
   req data_pred(data);
   ens data_pred(data);
@*/
```

Correspondingly, we need to pass an argument for this function type parameter as an extra argument to the `is_I32Predicate` function type chunk:

```
unsafe fn filter_nodes(n: *mut Node, p: I32Predicate, data: *mut u8) -> *mut Node
```

```
//@ req Nodes(n, _) &*& [_]is_I32Predicate(p, ?data_pred) &*& data_pred(data);
//@ ens Nodes(result, _) &*& data_pred(data);
{ ... }

impl Stack {
   unsafe fn filter(stack: *mut Stack, p: I32Predicate, data: *mut u8)
   //@ req Stack(stack, _) &*& [_]is_I32Predicate(p, ?data_pred) &*& data_pred(data);
   //@ ens Stack(stack, _) &*& data_pred(data);
   { ... }
}
```

And we need to specify an argument for each of a VeriFast function type's parameters between the second set of parentheses of the **produce_fn_ptr_chunk** ghost command:

```
fn main()
//@ req true;
//@ ens true;
{
   unsafe {
      let s = Stack::create();
      Stack::push(s, 10);
      Stack::push(s, 20);
      let mut a = read_i32();
      /*@
      produce_fn_ptr_chunk I32Predicate(neq_a)(neq_a_data)(data, x) {
         call();
      }
      @*/
      //@ close neq_a_data(&a as *mut u8);
      Stack::filter(s, neq_a, &raw mut a as *mut u8);
      //@ open neq_a_data(&a as *mut u8);
      Stack::dispose(s);
   }
}
```

Exercise 18 *Add a function* `Stack::map` *that takes a function* f *that takes an* `i32` *and returns an* `i32`. `Stack::map` *replaces the value of each element of the stack with the result of applying* f *to it.*

Write a client program that creates a stack containing the values 10, 20, 30; then reads a value from the user; and then adds this value to each element of the stack using `Stack::map`. *Verify the memory safety of the resulting program.*

16 Generics

Consider the solution to Exercise 15, where we verified full functional correctness of a `Stack::reverse` function for a stack of integers. We used an `i32s` inductive datatype, fixpoint functions `i32s_append` and `i32s_reverse`, and lemmas `i32s_append_nil_lemma` and `i32s_append_assoc_lemma`. Suppose, now, that we need the same functionality for a stack of pointers. For the Rust structs and functions, the solution is of course to apply Rust's generics support: we make the structs and functions generic in the element type. But what about the VeriFast constructs? Fortunately, VeriFast, too, supports generics: inductive datatypes, fixpoint functions, lemmas, and predicates can all be parameterized. Here are parameterized versions of `i32s`, `i32s_append`, and `i32s_append_nil_lemma`:

```
inductive list<t> = nil | cons(t, list<t>);
```

```
fix append<t>(xs: list<t>, ys: list<t>) -> list<t> {
   match xs {
      nil => ys,
      cons(x, xs0) => cons(x, append(xs0, ys))
   }
}

lem append_nil<t>(xs: list<t>)
   req true;
   ens append(xs, nil) == xs;
{
   match xs {
      nil => {}
      cons(x, xs0) => {
         append_nil(xs0);
      }
   }
}
```

As you can see, an inductive datatype definition, fixpoint function definition, or lemma definition accepts an optional *type parameter list*, which is a list of *type parameters* enclosed in angle brackets. Inside the definition, the type parameters can be used just like other types. Whenever a type-parameterized datatype, fixpoint, lemma, predicate, or constructor of a type-parameterized datatype, is mentioned, a *type argument list* has to be supplied, which is a list of types enclosed in angle brackets.

Here's what generic predicates `Nodes` and `Stack` and function `Stack::reverse` look like:

```
pred Nodes<T>(node: *mut Node<T>, values: list<T>) =
   if node == 0 {
      values == nil
   } else {
      (*node).next |-> ?next &*& (*node).value |-> ?value &*&
      alloc_block_Node(node) &*& Nodes(next, ?values0) &*&
      values == cons(value, values0)
   };

pred Stack<T>(stack: *mut Stack<T>, values: list<T>) =
   (*stack).head |-> ?head &*& alloc_block_Stack(stack) &*& Nodes(head, values);

impl Stack {
   unsafe fn reverse(stack: *mut Stack<T>)
   //@ req Stack(stack, ?values);
   //@ ens Stack(stack, reverse(values));
   {
      //@ open Stack(stack, values);
      let mut n = (*stack).head;
      let mut m = std::ptr::null_mut();
      //@ close Nodes(m, nil);
      //@ append_nil(reverse(values));
      loop {
         //@ inv Nodes(m, ?values1) &*& Nodes(n, ?values2) &*& reverse(values) == append(reverse(values2), value
         //@ open Nodes(n, values2);
         if n.is_null() {
            break;
         }
```

```
            let next = (*n).next;
            //@ assert Nodes(next, ?values2tail) &*& (*n).value |-> ?value;
            (*n).next = m;
            m = n;
            n = next;
            //@ close Nodes(m, cons(value, values1));
            //@ append_assoc(reverse(values2tail), cons(value, nil), values1);
        }
        (*stack).head = m;
        //@ close Stack(stack, reverse(values));
    }
}
```

Just like Rust does, VeriFast performs *type argument inference*, so that most of the time you need not explicitly write type arguments when applying a generic VeriFast construct. If VeriFast encounters an occurrence of a type-parameterized entity in an expression and no type argument list was specified, VeriFast will infer the type argument list. Since VeriFast's type system has no subtyping, a simple unification-based inference approach is sufficient. The result is that you almost never have to supply type argument lists in expressions explicitly. Note: just like Rust, VeriFast does not infer type argument lists in types; that is, when you mention a type-parameterized inductive datatype, you must always supply the type arguments explicitly.

Of course, the `list` datatype is useful much more generally than just for stacks of integers and stacks of pointers. In fact, verification of any non-trivial program will require the use of lists. For this reason, VeriFast's prelude includes a file `list.rsspec` that defines the list datatype, all of the fixpoints and lemmas used in the example, and many more. All definitions in the VeriFast prelude are implicitly available to each file that is verified by VeriFast. Note: This also means that you cannot define your own versions of `nil`, `cons`, or other elements provided by `list.rsspec`, since this would result in a name clash.

17 Predicate Values

In the previous section, we obtained a generic stack. In this section, we will use this stack to keep track of a collection of objects.

Let's try to verify the following program. It is a stack-based calculator for 2D vectors. It uses the stack from the previous section. The user can push a vector onto the stack, replace the top two vectors with their sum, and pop the top vector to print it.

```
struct Vector {
    x: i32,
    y: i32,
}

impl Vector {
    unsafe fn create(x: i32, y: i32) -> *mut Vector {
        let result = alloc(Layout::new::<Vector>()) as *mut Vector;
        if result.is_null() {
            handle_alloc_error(Layout::new::<Vector>());
        }
        (*result).x = x;
        (*result).y = y;
        result
    }
}
```

```
fn main() {
   unsafe {
      let s = Stack::create();
      loop {
         let cmd = input_char();
         match cmd {
            'p' => {
               let x = input_i32();
               let y = input_i32();
               let v = Vector::create(x, y);
               Stack::push(s, v);
            }
            '+' => {
               assert!(!Stack::is_empty(s), "Stack␣underflow");
               let v1 = Stack::pop(s);
               assert!(!Stack::is_empty(s), "Stack␣underflow");
               let v2 = Stack::pop(s);
               let sum = Vector::create((*v1).x + (*v2).x, (*v1).y + (*v2).y);
               dealloc(v1 as *mut u8, Layout::new::<Vector>());
               dealloc(v2 as *mut u8, Layout::new::<Vector>());
               Stack::push(s, sum);
            }
            '=' => {
               assert!(!Stack::is_empty(s), "Stack␣underflow");
               let v_ = Stack::pop(s);
               output_i32((*v_).x);
               output_i32((*v_).y);
               dealloc(v_ as *mut u8, Layout::new::<Vector>());
            }
            _ => panic!("Bad␣command")
         }
      }
   }
}
```

The specification of `Vector::create` is easy:

```
//@ pred Vector(v: *mut Vector) = (*v).x |-> ?x &*& (*v).y |-> ?y &*& alloc_block_Vector(v);

impl Vector {
   unsafe fn create(x: i32, y: i32) -> *mut Vector
   //@ req true;
   //@ ens Vector(result);
   { ... }
}
```

The tricky part is the loop invariant for the loop in `main`. The loop invariant should state that we have a stack at `s`, and furthermore, that for each element of `s`, we have a vector. We can express the first part using the assertion `Stack(s, ?values)`, and we can easily write a recursive predicate to express the second part:

```
pred Vectors(vs: list<*mut Vector>) =
   match vs {
      nil => true,
      cons(v, vs0) => Vector(v) &*& Vectors(vs0)
```

```
};
```

This would work fine. However, it is unfortunate that we have to define a new predicate whenever we wish to express that we have a given predicate for each element of a list. To address this, VeriFast supports *predicate values*. That is, you can pass a predicate as an argument to another predicate. This allows us to generalize the above `Vectors` predicate as follows:

```
pred foreach<t>(xs: list<t>, p: pred(t)) =
   match xs {
      nil => true,
      cons(h, t) => p(h) &*& foreach(t, p)
   };
```

Note that type argument inference allows us to omit the type argument for the recursive `foreach` call.

This predicate is so generally useful that we included it in `list.rsspec`; as a result, it is automatically available in each file and you do not need to define it yourself.

Exercise 19 *Verify the program using* `foreach`*.*

18 Predicate Constructors

Let's add a twist to the program of the previous section:

```
impl Vector {
   unsafe fn create(limit: i32, x: i32, y: i32) -> *mut Vector {
      assert!(x * x + y * y <= limit * limit, "Vector too big");
      let result = alloc(Layout::new::<Vector>()) as *mut Vector;
      if result.is_null() {
         handle_alloc_error(Layout::new::<Vector>());
      }
      (*result).x = x;
      (*result).y = y;
      result
   }
}

fn main() {
   unsafe {
      let limit = input_i32();
      let s = Stack::create();
      loop {
         let cmd = input_char();
         match cmd {
            'p' => {
               let x = input_i32();
               let y = input_i32();
               let v = Vector::create(limit, x, y);
               Stack::push(s, v);
            }
            '+' => {
               assert!(!Stack::is_empty(s), "Stack underflow");
               let v1 = Stack::pop(s);
               assert!(!Stack::is_empty(s), "Stack underflow");
               let v2 = Stack::pop(s);
               let sum = Vector::create(limit, (*v1).x + (*v2).x, (*v1).y + (*v2).y);
```

```
                dealloc(v1 as *mut u8, Layout::new::<Vector>());
                dealloc(v2 as *mut u8, Layout::new::<Vector>());
                Stack::push(s, sum);
            }
            '=' => {
                assert!(!Stack::is_empty(s), "Stack underflow");
                let v_ = Stack::pop(s);
                std::hint::assert_unchecked((*v_).x * (*v_).x + (*v_).y * (*v_).y <= limit * limit);
                output_i32((*v_).x);
                output_i32((*v_).y);
                dealloc(v_ as *mut u8, Layout::new::<Vector>());
            }
            _ => panic!("Bad command")
        }
    }
  }
}
```

The program now starts by asking the user for a number that will serve as a limit on the size of the vectors. When creating a vector, the program checks that its size does not exceed the limit; otherwise it panics. When printing a vector, the program performs an unchecked assert that the vector satisfies the size limit.

How do we verify this assert?

We will need to extend the `Vector` predicate to include the information that the vector's size is within the limit. However, the limit is a local variable and is not in scope in the predicate definition. So we need to pass it as an additional argument. But then we can no longer use the `foreach` predicate, since it expects a predicate that takes just one parameter. To address this, VeriFies supports *partially applied predicates*, in the form of *predicate constructors*. To verify the example program, we can define a predicate constructor `Vector` as follows:

```
/*@
pred_ctor Vector(limit: i32)(v: *mut Vector) =
  (*v).x |-> ?x &*& (*v).y |-> ?y &*& alloc_block_Vector(v) &*& x * x + y * y <= limit * limit;
@*/
```

We can express that for each element of a list `values` we have a vector that satisfies limit `limit` as follows:

```
foreach(values, Vector(limit))
```

That is, wherever we can use a predicate name, we can also use a predicate constructor applied to an argument list. We can do so in open statements, in close statements, and in assertions.

Note: VeriFast currently does not support patterns in predicate constructor argument positions.

Exercise 20 *Verify the program.*

19 Multithreading

Consider the following program that walks a binary tree of nonnegative integers, for each of these integers N computes the N'th Fibonacci number (modulo 2^{64}), sums up the results (again modulo 2^{64}), and prints the sum to the console.

```
#![allow(unsafe_op_in_unsafe_fn)]
use std::{alloc::{alloc, handle_alloc_error, Layout}};

unsafe fn wrapping_fib(n: u16) -> u64 {
  if n <= 1 {
```

```
      1
   } else {
      let mut k: u16 = 2;
      let mut fib_k_minus_1: u64 = 1;
      let mut fib_k: u64 = 1;
      loop {
         if k == n { break; }

         let fib_k_plus_1 = fib_k_minus_1.wrapping_add(fib_k);

         k += 1;
         fib_k_minus_1 = fib_k;
         fib_k = fib_k_plus_1;
      }
      fib_k
   }
}

struct Tree {
   left: *mut Tree,
   right: *mut Tree,
   value: u16,
}

impl Tree {

   unsafe fn make(depth: u8) -> *mut Tree {
      if depth == 0 {
         std::ptr::null_mut()
      } else {
         let left = Self::make(depth - 1);
         let right = Self::make(depth - 1);
         let value = 5000; // TODO: Use a random number here
         let t = alloc(Layout::new::<Tree>()) as *mut Tree;
         if t.is_null() {
            handle_alloc_error(Layout::new::<Tree>());
         }
         (*t).left = left;
         (*t).right = right;
         (*t).value = value;
         t
      }
   }

   unsafe fn compute_sum_fibs(tree: *mut Tree) -> u64 {
      if tree.is_null() {
         0
      } else {
         let left_sum = Self::compute_sum_fibs((*tree).left);
         let f = wrapping_fib((*tree).value);
         let right_sum = Self::compute_sum_fibs((*tree).right);
         left_sum.wrapping_add(f).wrapping_add(right_sum)
      }
```

```
   }

}

unsafe fn print_u64(value: u64)
//@ req true;
//@ ens true;
//@ assume_correct
{
   println!("{}", value);
}

fn main() {
   unsafe {
      let tree = Tree::make(22);
      let sum = Tree::compute_sum_fibs(tree);
      print_u64(sum)
   }
}
```

Exercise 21 *Verify the memory safety of this program. You may leak the tree (see Section 4) after computing the sum.*

This program takes about 5 seconds on the author's machine. However, the author's machine is a multicore machine, so we might be able to get a speedup if we distribute the work amongst two threads, which may then be scheduled by the operating system on both cores simultaneously. Here's a first attempt:

```
struct Sendable<T> { payload: T }
unsafe impl<T> Send for Sendable<T> {}

impl<T> Sendable<T> {
   fn inner(self) -> T { self.payload }
}

fn main() {
   unsafe {
      let tree = Tree::make(22);
      let left = (*tree).left;
      let right = (*tree).right;
      let left_packaged = Sendable { payload: left };
      let left_join_handle = std::thread::spawn(
         move || Tree::compute_sum_fibs(left_packaged.inner())
      );
      let right_packaged = Sendable { payload: right };
      let right_join_handle = std::thread::spawn(
         move || Tree::compute_sum_fibs(right_packaged.inner())
      );
      let root_fib = wrapping_fib((*tree).value);

      let left_sum = left_join_handle.join().unwrap();
      let right_sum = right_join_handle.join().unwrap();
      let sum = left_sum.wrapping_add(root_fib).wrapping_add(right_sum);
```

```
      print_u64(sum)
   }
}
```

The main program creates a tree of depth 22. It then starts two threads. The first thread computes the sum of the Fibonacci numbers of the values of the left subtree, and the second thread computes the sum of the Fibonacci numbers of the values of the right subtree. (We need to package the raw pointers to the subtrees into a `Sendable` struct which we **unsafe**ly declare to be `Send`, because raw pointers are not themselves `Send`.) The main thread then waits for both threads to finish, and finally sums up the results of both threads with the Fibonacci number of the root node's value. On the author's machine, this program takes only about 2.5 seconds; a twofold speedup!

Now, let's verify the memory safety of this program. Unfortunately, VeriFast does not yet support verifying code that uses lambda expressions, so we need to factor these out into separate functions that we mark as `assume_correct` and for which we carefully write a sound contract:

```
/*@
fn_type Spawnee<A, R>(pre: pred(A, pred(R))) = unsafe fn(arg: A) -> R;
   req pre(arg, ?post);
   ens post(result);

pred JoinHandle<R>(h: JoinHandle<Sendable<R>>, post: pred(R));
@*/

type Spawnee<A, R> = unsafe fn(arg: A) -> R;

struct Sendable<T> { payload: T }
unsafe impl<T> Send for Sendable<T> {}

unsafe fn spawn<A, R>(f: Spawnee<A, R>, arg: A) -> JoinHandle<Sendable<R>>
where A: 'static, R: 'static
//@ req [_]is_Spawnee::<A, R>(f, ?pre) &*& pre(arg, ?post);
//@ ens JoinHandle(result, post);
//@ assume_correct
{
   let package = Sendable { payload: arg };
   std::thread::spawn(move || {
      let package_moved = package;
      Sendable { payload: f(package_moved.payload) }
   })
}

unsafe fn join<R>(h: JoinHandle<Sendable<R>>) -> R
//@ req JoinHandle(h, ?post);
//@ ens post(result);
//@ assume_correct
{
   h.join().unwrap().payload
}

fn main() {
   unsafe {
      let tree = Tree::make(22);
      let left = (*tree).left;
      let right = (*tree).right;
```

```
      let left_join_handle = spawn(Tree::compute_sum_fibs, left);
      let right_join_handle = spawn(Tree::compute_sum_fibs, right);
      let root_fib = wrapping_fib((*tree).value);

      let left_sum = join(left_join_handle);
      let right_sum = join(right_join_handle);
      let sum = left_sum.wrapping_add(root_fib).wrapping_add(right_sum);

      print_u64(sum)
   }
}
```

The verification of this program does not require any new techniques; we will simply apply the techniques we saw in Sections 12, 15, and 18 on function pointers, parameterized function types, and predicate constructors.

Exercise 22 *Verify the memory safety of the program. Don't worry about memory deallocation; simply leak chunks that you no longer need.*

20 Fractional Permissions

Suppose, now, that we want to adapt the program of the previous section as follows: instead of computing just the sum of the Fibonacci numbers, we want to compute both their sum and their product. Since our machine has two cores, we want to have two threads: one that computes the sum of the Fibonacci numbers of the values of the nodes of the tree, and one that computes the product of the Fibonacci numbers of the values of the nodes of the tree:

```
impl Tree {
   unsafe fn compute_product_fibs(tree: *mut Tree) -> u64 {
      if tree.is_null() {
         1
      } else {
         let left_product = Self::compute_product_fibs((*tree).left);
         let f = wrapping_fib((*tree).value);
         let right_product = Self::compute_product_fibs((*tree).right);
         left_product.wrapping_mul(f).wrapping_mul(right_product)
      }
   }
}

fn main() {
   unsafe {
      let tree = Tree::make(22);

      let sum_join_handle = spawn(Tree::compute_sum_fibs, tree);
      let product_join_handle = spawn(Tree::compute_product_fibs, tree);

      let sum = join(sum_join_handle);
      let product = join(product_join_handle);

      print_u64(sum);
      print_u64(product);
   }
}
```

However, this program cannot be verified using the techniques that we have seen. The first `spawn` call consumes the `Tree` chunk, and therefore VeriFast will complain that the second call's precondition is not satisfied. In the system, as explained, only one thread may own a particular chunk of memory at any one time. Therefore, if the sum thread owns the tree, the product thread cannot access it simultaneously. However, this is in fact safe, so long as both threads only read and do not modify the tree.

VeriFast supports the read-only sharing of chunks of memory using *fractional permissions*. Each chunk in the VeriFast symbolic heap has a *coefficient*, which is a real number between zero, exclusive, and one, inclusive. The default coefficient is 1 and is not shown. If a chunk's coefficient is not 1, it is shown to the left of the chunk enclosed in square brackets. A chunk with coefficient 1 represents a *full permission*, that is, a *read-write permission*. A chunk with a coefficient less than 1 represents a *fractional permission*, that is, a *read-only permission*.

Since the `Tree::compute_sum_fibs` and `Tree::compute_product_fibs` functions do not modify the tree, they require only a fraction of the `Tree` chunk:

```
impl Tree {
   unsafe fn compute_sum_fibs(tree: *mut Tree) -> u64
   //@ req [?frac]Tree(tree, ?depth);
   //@ ens [frac]Tree(tree, depth);
   {
      if tree.is_null() {
         0
      } else {
         //@ open [frac]Tree(tree, depth);
         let left_sum = Self::compute_sum_fibs((*tree).left);
         let f = wrapping_fib((*tree).value);
         let right_sum = Self::compute_sum_fibs((*tree).right);
         //@ close [frac]Tree(tree, depth);
         left_sum.wrapping_add(f).wrapping_add(right_sum)
      }
   }
}
```

Notice that we use a pattern in the coefficient position, so that the function may be applied to an arbitrarily small fraction of the `Tree` chunk. Notice also that we mention the fraction in the **`open`** and **`close`** statements. This is not necessary for the **`open`** statement, since it will by default open whatever fraction is present in the symbolic heap, but it is necessary for the **`close`** statement, since it will by default attempt to close a chunk with coefficient 1.

Exercise 23 *Verify the memory safety of the program. Adapt the spawnee pre and post predicates so that they require only a 1/2 fraction of the tree chunk. Note: decimal notation is not supported; use fractional notation instead.*

21 Precise Predicates

The program of the previous section leaks large amounts of memory. This is fine, since the leaks occur only just before the program ends anyway. However, suppose now that this program was part of a larger, long-running program; in that case it would be important to eliminate the leaks. Therefore, in this section, let's attempt to eliminate the **`leak`** commands from the program of the previous section.

First of all, we need to update the Rust program so that it properly deallocates all dynamically allocated memory. We need to introduce a function `Tree::dispose` and update function `main`:

```
impl Tree {
   unsafe fn dispose(tree: *mut Tree) {
      if !tree.is_null() {
```

```
        Self::dispose((*tree).left);
        Self::dispose((*tree).right);
        dealloc(tree as *mut u8, Layout::new::<Tree>());
      }
   }
}

fn main() {
   unsafe {
      let tree = Tree::make(22);

      let sum_join_handle = spawn(Tree::compute_sum_fibs, tree);
      let product_join_handle = spawn(Tree::compute_product_fibs, tree);

      let sum = join(sum_join_handle);
      let product = join(product_join_handle);

      Tree::dispose(tree);

      print_u64(sum);
      print_u64(product);
   }
}
```

Updating the annotations is slightly trickier. Consider the `Tree::dispose` call in function `main`. It needs full permission for the tree, i.e. it needs a `Tree(tree, _)` chunk. Notice that each `join` call yields a `[1/2]Tree(tree, _)` chunk. We have two half chunks, and we need one full chunk; why does VeriFast not simply merge the two halves?

The reason is that merging two fractional chunks into a single chunk is not always a sound (i.e., safe) thing to do. For example, the following program verifies:

```
/*@
pred foo(b: bool) = true;

pred bar(x: *mut i32, y: *mut i32) = foo(?b) &*& if b { *x |-> 0 } else { *y |-> 0 };

lem merge_bar() // This lemma is false!!
   req [?f1]bar(?x, ?y) &*& [?f2]bar(x, y);
   ens [f1 + f2]bar(x, y);
{
   assume(false);
}
@*/

fn main()
//@ req true;
//@ ens true;
{
   unsafe {
      let mut x = 0;
      let mut y = 0;
      //@ close [1/2]foo(true);
      //@ close [1/2]bar(&x, &y);
      //@ close [1/2]foo(false);
```

```
      //@ close [1/2]bar(&x, &y);
      //@ merge_bar();
      //@ open bar(&x, &y);
      //@ assert foo(?b);
      //@ if b { points_to_limits(&x); } else { points_to_limits(&y); }
      std::hint::unreachable_unchecked();
   }
}
```

At the end of the `main` function, if `b` is true, we have a fraction 1.5 of the points-to chunk for `x`, and otherwise we have a fraction 1.5 of the points-to chunk for `y`.

This program demonstrates that assuming that any two fractions of a given chunk can be merged into a single chunk, leads to unsoundness (i.e. accepting a program that has undefined behavior). It uses the following lemma from `prelude_core.rsspec`:

```
lem points_to_limits<T>(p: *T);
   req [?f]points_to(p, ?v);
   ens [f]points_to(p, v) &*& f <= 1 &*& ...;
```

The problem is that the two `[1/2]bar(&x, &y)` chunks do not represent the same memory region: one represents `[1/2]x |-> 0` and the other one represents `[1/2]y |-> 0`. Combining them should yield neither `x |-> 0` nor `y |-> 0`.

However, merging two fractions is sound if both fractions represent the same memory region. To support this, VeriFast supports the notion of *precise predicates*: you can declare a predicate as *precise* by writing a semicolon instead of a comma between the *input parameters* and the *output parameters* in the predicate's parameter list. For such a predicate, VeriFast checks that two chunks with the same input arguments represent the same memory region and always have the same output arguments. VeriFast automatically merges fractions of precise predicates that have the same input arguments.

For example, predicate `points_to` itself is a precise predicate; it is declared in `prelude_core.rsspec` as follows:

```
pred points_to<t>(p: *t; v: t);
```

This declaration specifies that predicate `points_to` is precise and has one input parameter `p` and one output parameter `v`. When VeriFast detects two chunks `[f1]points_to::<t>(p1, v1)` and `[f2]points_to::<t>(p2, v2)` and it can prove that `p1` equals `p2`, then it merges the chunks into a single chunk `[f1 + f2]points_to::<t>(p1, v1)` and it produces the assumption that `v1` equals `v2`.

In conclusion, to verify our example program, we need to declare predicate `Tree` as precise:

```
pred Tree(t: *mut Tree; depth: u8) =
   if t == 0 {
      depth == 0
   } else {
      (*t).left |-> ?left &*& Tree(left, ?depth0) &*& depth == depth0 + 1 &*&
      (*t).right |-> ?right &*& Tree(right, depth0) &*&
      (*t).value |-> ?value &*&
      alloc_block_Tree(t)
   };
```

Notice that we rephrased the body of the predicate slightly so that it is accepted by VeriFast's preciseness analysis. The program now verifies.

The preciseness analysis checks that the body of the predicate is precise under the input parameters and fixes the output parameters. The rules for the various forms of assertions being precise under a set of fixed variables X are as follows:

- A predicate assertion is precise under X if the predicate is precise and the input arguments are fixed by X; it fixes any variables that appear as output arguments. For example, for assertion

`p(x + y, z)` to be considered precise, `p` must be a precise predicate. Furthermore, suppose it has one input parameter. Then `x` and `y` must be in X, and the assertion fixes `z`.

- A fractional assertion is precise under X if the coefficient is fixed by X or it is a dummy pattern and the body is precise under X; it fixes any variables fixed by its body.
- A boolean assertion is always precise under any X. It does not fix any variables, unless it is an equality and the left-hand side is a variable and the right-hand side is fixed by X; in that case, it fixes the variable on the left-hand side.
- A conditional assertion is precise under X if the condition is fixed by X and each branch is precise under X; the fixed variables are the variables that are fixed by all branches.
- Similarly, a **`match`** assertion is precise under X if its operand is fixed by X and each branch is precise under the union of X and the constructor arguments; the fixed variables are the variables that are fixed by all branches.
- A separating conjunction is precise under X if its first operand is precise under X, and its second operand is precise under the union of X and the variables fixed by the first operand. It fixes the variables fixed by either operand.

That is, the analysis traverses the assertion from left to right, tracking the set of fixed variables. Input arguments of nested predicate assertions and branch conditions must depend only on variables that are already fixed; variables that appear as output arguments of nested predicate assertions or on the left-hand side of equalities whose right-hand side is fixed are added to the set of fixed variables.

22 Auto-open/close

In the preceding section, we introduced VeriFast's support for precise predicates so that we could merge fractions of the same chunk. However, declaring a predicate as precise has another advantage: it enables VeriFast's logic for automatically opening and closing the predicate. This cuts down on the number of **`open`** and **`close`** commands that you need to write explicitly.

Specifically, when VeriFast is consuming a predicate assertion, and no matching chunk exists in the symbolic heap, but the predicate is precise and all input arguments are specified in the assertion, then auto-open or auto-close is attempted. Auto-close is performed if a chunk that appears in the body of the predicate is found in the symbolic heap; auto-open is performed if the desired chunk appears in the body of some chunk that appears in the symbolic heap.

For example, in the program of the preceding section this allows us to remove all ghost commands for opening or closing `Tree` chunks.

23 Mutexes

Suppose we want to write a program that monitors two sensors and that prints the total number of pulses detected by both sensors, once every second. Suppose we can wait for a pulse from a given sensor using the API function `wait_for_pulse`. Since pulses come in at both sensors simultaneously, we need to wait for pulses from each sensor in a separate thread. Whenever a pulse comes in at a sensor, the corresponding thread increments a counter that is shared between the threads. The main thread prints the value of the counter once every second.

When multiple threads access the same variable concurrently, such as the counter in this example, then they need to synchronize their accesses; otherwise, two concurrent accesses may interfere, causing the result to be different from the result that would be obtained if the accesses were performed one after the other. Unsynchronized conflicting concurrent accesses of the same variable are known as *data races*. In some programming languages, including C, C++, and Rust, data races cause *undefined behavior*, which means

the compiler can miscompile programs that perform data races in arbitrary ways. Most programming platforms provide various synchronization constructs to synchronize threads in various ways.

Perhaps the most common synchronization construct is the mutual exclusion lock, also known as a lock or a mutex. At any point, a mutex is in one of two states: either it is free, or it is held by some thread. A mutex can never be held by more than one thread. Mutexes provide two operations: acquire and release. When a thread attempts to acquire a mutex, there are two cases.

- if the mutex is free, the attempt succeeds and the thread holds the mutex until it releases it.
- if the mutex is held by some other thread, the thread that performed the attempt blocks until the mutex is free.
- if the mutex is already held by the thread that attempts to acquire it, then the result depends on whether the mutex is re-entrant or non-re-entrant. If the mutex is re-entrant, the attempt succeeds and the thread still holds the lock. If the mutex is non-re-entrant, the attempt fails. This either means that the thread blocks forever, or that an error occurs and the program is terminated.

Here's the source code of the example program:

```
use std::alloc::{Layout, alloc, handle_alloc_error};

unsafe fn wait_for_pulse(_source: i32) { ... }
unsafe fn print_u32(n: u32) { println!("{}", n); }

struct CountPulsesData {
   counter: *mut u32,
   mutex: *mut Mutex,
   source: i32,
}

unsafe fn count_pulses(data: CountPulsesData) {
   let CountPulsesData {counter, mutex, source} = data;

   loop {
      wait_for_pulse(source);
      let guard = acquire(mutex);
      *counter = (*counter).checked_add(1).unwrap();
      release(guard);
   }
}

unsafe fn count_pulses_async(counter: *mut u32, mutex: *mut Mutex, source: i32) {
   let data = CountPulsesData { counter, mutex, source };
   spawn(count_pulses, data);
}

fn main() {
   unsafe {
      let counter = alloc(Layout::new::<u32>()) as *mut u32;
      if counter.is_null() { handle_alloc_error(Layout::new::<u32>()); }
      *counter = 0;
      let mutex = create_mutex();

      count_pulses_async(counter, mutex, 1);
      count_pulses_async(counter, mutex, 2);
```

```
      loop {
         std::thread::sleep(std::time::Duration::from_millis(1000));
         let guard = acquire(mutex);
         let count = *counter;
         release(guard);
         print_u32(count);
      }
   }
}
```

This program uses the following API for spawning a thread (similar to the `spawn` function used in earlier examples, but simpler since we don't need to join the spawned thread):

```
/*@
fn_type Spawnee<T>(pre: pred(T)) = unsafe fn(arg: T);
   req pre(arg);
   ens true;
@*/

unsafe fn spawn<T: 'static>(f: unsafe fn(arg: T), arg: T)
//@ req [_]is_Spawnee(f, ?pre) &*& pre(arg);
//@ ens true;
```

For mutual exclusion, it uses the following simple API offering non-re-entrant mutexes:[13]

```
unsafe fn create_mutex() -> *mut Mutex { ... }
unsafe fn acquire(mutex: *mut Mutex) -> MutexGuard { ... }
unsafe fn release(guard: MutexGuard) { ... }
unsafe fn dispose_mutex(mutex: *mut Mutex) { ... }
```

It's easy to make mistakes when programming using mutexes. The worst problem is if the programmer forgets to acquire the mutex before accessing the data structure that it protects. The consequence will be incorrect results, but this may be difficult to diagnose.

Fortunately, VeriFast can help us catch these tricky bugs. Remember from the preceding sections that as a result of VeriFast's checks, each thread can access only the memory locations that it owns, and no two threads can (fully) own the same memory location at the same time. This prevents interference from concurrent accesses. But how, then, can threads share a mutable variable? The answer is, of course, using mutexes. When a mutex is created, it takes ownership of a certain set of memory locations, as specified by its *lock invariant*. When a thread acquires a mutex, the memory locations that were owned by the mutex now become owned by the thread, until the thread releases the mutex, at which point the memory locations again become the property of the mutex. This way, by sharing a mutex, threads can indirectly share arbitrary memory locations in a well-synchronized way.

Here is a specification for the mutex API:

```
//@ pred Mutex(mutex: *mut Mutex; inv_: pred());
//@ pred MutexGuard(guard: MutexGuard, mutex: *mut Mutex, inv_: pred(), frac: real, t: thread_id_t);

unsafe fn create_mutex() -> *mut Mutex
//@ req exists::<pred()>(?inv_) &*& inv_();
//@ ens Mutex(result, inv_);

unsafe fn acquire(mutex: *mut Mutex) -> MutexGuard
//@ req [?frac]Mutex(mutex, ?inv_);
```

[13] This API can be implemented straightforwardly on top of Rust's `std::sync::Mutex<()>`.

```
//@ ens MutexGuard(result, mutex, inv_, frac, currentThread) &*& inv_();

unsafe fn release(guard: MutexGuard)
//@ req MutexGuard(guard, ?mutex, ?inv_, ?frac, currentThread) &*& inv_();
//@ ens [frac]Mutex(mutex, inv_);

unsafe fn dispose_mutex(mutex: *mut Mutex) { ... }
//@ req Mutex(mutex, ?inv_);
//@ ens inv_();
```

When you create a mutex, you need to specify the *lock invariant* that specifies the memory locations that will be owned by the mutex. You do so by specifying a *predicate value* (see Section 17). Since the real function `create_mutex` cannot take the predicate value as a real argument, it takes it as a ghost argument, in the form of the argument of the `exists` predicate, which exists only for this purpose. It is defined in `prelude_core.rsspec` as

```
pred exists<t>(x: t;) = true;
```

Before calling `create_mutex`, you need to close an `exists` chunk, whose argument is the name of the predicate that specifies the lock invariant for the new mutex. The `create_mutex` call consumes the ghost argument chunk and the lock invariant chunk itself, and produces a `Mutex` chunk that represents the mutex. This chunk specifies the lock invariant as its second argument.

To allow multiple threads to share a mutex, a mutex chunk can be split into multiple fractions. Only a fraction of a mutex chunk is required to acquire the mutex. When the mutex is acquired, the `Mutex` chunk fraction is transformed into a `MutexGuard` chunk, that specifies not only the mutex and the lock invariant, but also the thread that acquired the mutex and the coefficient of the mutex chunk fraction that was used to acquire the mutex. The `acquire` call additionally produces the lock invariant, giving the thread access to the memory locations that were owned by the mutex.

The `release` call consumes a `MutexGuard` chunk for the current thread, as well as the lock invariant, and produces the original `Mutex` chunk that was used to acquire the lock.

If necessary, after a program is done using a mutex, it can reassemble all mutex chunk fractions and dispose the mutex; this returns ownership of the lock invariant to the thread that disposes the mutex.

Exercise 24 *Verify the example program.*

24 Leaking and Dummy Fractions

We start from the example program of the previous section. In that program, the number of pulse sources is fixed. Suppose now that sources are connected and disconnected dynamically. Initially, there are no sources. Suppose there is an API function `wait_for_source` that waits for a new source to be connected and returns its identifier. Suppose further that API function `wait_for_pulse` now returns a boolean. If the result is false, it means a new pulse was detected; if it is true, it means the source has been disconnected. The goal remains to count the total number of pulses detected across all sources, and print it out once a second. The following program achieves this:

```
use std::alloc::{alloc, handle_alloc_error, Layout};

unsafe fn wait_for_source() -> i32 { ... }
unsafe fn wait_for_pulse(_source: i32) -> bool { ... }
unsafe fn print_u32(n: u32) { ... }

struct CountPulsesData {
    counter: *mut u32,
    mutex: *mut Mutex,
    source: i32,
```

```
}

unsafe fn count_pulses(data: CountPulsesData) {
   let CountPulsesData {counter, mutex, source} = data;

   loop {
      let done = wait_for_pulse(source);
      if done { break }
      let guard = acquire(mutex);
      *counter = (*counter).checked_add(1).unwrap();
      release(guard);
   }
}

unsafe fn count_pulses_async(counter: *mut u32, mutex: *mut Mutex, source: i32) {
   let data = CountPulsesData { counter, mutex, source };
   spawn(count_pulses, data);
}

struct PrintCountData {
   counter: *mut u32,
   mutex: *mut Mutex,
}

unsafe fn print_count(data: PrintCountData) {
   let PrintCountData {counter, mutex} = data;
   loop {
      std::thread::sleep(std::time::Duration::from_millis(1000));
      let guard = acquire(mutex);
      print_u32(*counter);
      release(guard);
   }
}

unsafe fn print_count_async(counter: *mut u32, mutex: *mut Mutex) {
   let data = PrintCountData { counter, mutex };
   spawn(print_count, data);
}

fn main() {
   unsafe {
      let counter = alloc(Layout::new::<u32>()) as *mut u32;
      if counter.is_null() {
         handle_alloc_error(Layout::new::<u32>());
      }
      *counter = 0;
      let mutex = create_mutex();

      print_count_async(counter, mutex);

      loop {
         let source = wait_for_source();
         count_pulses_async(counter, mutex, source);
```

```
        }
    }
}
```

Notice that in the example program of the previous section, the mutex was shared amongst three threads, whereas now it may at any time be shared amongst arbitrarily many threads. This has consequences for verification: whereas in the previous section we could get away with giving each thread one third of the mutex chunk, splitting up the fractions in the current example is more complicated.

Furthermore, there is another problem: when a `count_pulses` thread terminates, it still owns a fraction of the mutex. VeriFast complains about this as part of its leak check. Indeed, in general, leaking mutexes can cause the program to eventually run out of memory. However, in the case of this program, leaking the one mutex is not a problem. And since the mutex chunk fractions will be leaked anyway and will never be reassembled to dispose the mutex, there is not really any point in carefully keeping track of the coefficients of the various fractions.

VeriFast has a feature called *dummy fractions* that makes it easy to deal with chunks that are shared among many threads and that will never be reassembled. Specifically, applying the **`leak`** ghost command to a chunk does not remove the chunk but simply replaces the chunk's coefficient with a *dummy fraction coefficient symbol*. A chunk whose coefficient is a dummy fraction coefficient symbol is called a *dummy fraction*. When VeriFast performs the leak check at the end of each function, it complains only about chunks that are not dummy fractions.

Dummy fractions are denoted in assertions using dummy patterns, as in `[_]chunk(args)`. When consuming an assertion with a dummy coefficient, the matching chunk must be a dummy fraction, or VeriFast reports an error since matching the chunk would implicitly leak it. When producing an assertion with a dummy coefficient, the produced chunk is a dummy fraction.

To make it easy to share dummy fractions arbitrarily, when consuming a dummy fraction assertion, VeriFast automatically splits the matching chunk in two: one part is consumed, and the other part remains in the symbolic heap.

Exercise 25 *Verify the example program. Leak the mutex chunk directly after creating it. Use dummy patterns to denote the mutex chunk fractions' coefficients in assertions.*

25 Byte Arrays

Let's verify a program that reads a fixed number of bytes from standard input and then writes them out twice. Here's a version that reads 5 bytes:

```
use std::io::{Read, Write, stdin, stdout};

unsafe fn read_byte() -> u8 {
    let mut buf = [0u8];
    stdin().read_exact(&mut buf[..]).unwrap();
    buf[0]
}

unsafe fn write_byte(value: u8) {
    let buf = [value];
    stdout().write(&buf[..]).unwrap();
}

fn main() {
    unsafe {
        let b1 = read_byte();
        let b2 = read_byte();
        let b3 = read_byte();
```

```
      let b4 = read_byte();
      let b5 = read_byte();
      for _ in 0..2 {
         write_byte(b1);
         write_byte(b2);
         write_byte(b3);
         write_byte(b4);
         write_byte(b5);
      }
   }
}
```

This program works. If we run the program and enter `Hello`, we get back `HelloHello`.

However, clearly this approach is not practical for large numbers of bytes. One approach that works for large numbers of bytes is to use an array of bytes. We allocate the array using an `alloc` function and then read and write the bytes using a recursive function. For now, we don't worry about freeing the array at the end of the program. Let's first again do a version that reads 5 bytes.

```
use std::alloc::{Layout, alloc, handle_alloc_error};
unsafe fn read_byte() -> u8 { ... }
unsafe fn write_byte(value: u8) { ... }

unsafe fn alloc(count: usize) -> *mut u8 {
   let layout = Layout::from_size_align(count, 1).unwrap();
   let result = alloc(layout);
   if result.is_null() {
      handle_alloc_error(layout);
   }
   result
}

unsafe fn read_bytes(start: *mut u8, count: usize) {
   if count > 0 {
      let b = read_byte();
      *start = b;
      read_bytes(start.add(1), count - 1);
   }
}

unsafe fn write_bytes(start: *mut u8, count: usize) {
   if count > 0 {
      let b = *start;
      write_byte(b);
      write_bytes(start.add(1), count - 1);
   }
}

fn main() {
   unsafe {
      let array = alloc(5);
      read_bytes(array, 5);
      write_bytes(array, 5);
      write_bytes(array, 5);
   }
```

```
}
```

This program works. Now let's verify it.

It's always a good approach to simply run VeriFast on the program and see where it complains. If you verify the above program, VeriFast complains that the functions have no contract. Let's start by giving function `read_bytes` the simplest possible contract:

```
//@ req true;
//@ ens true;
```

VeriFast now complains at the following line in function `read_bytes`:

```
*start = b;
```

This statement writes byte `b` at the place in memory pointed to by pointer `start`. Whenever VeriFast sees such a statement, it checks that the function has permission to write to this place. Specifically, it checks that a chunk that matches `points_to_::<u8>(start, _)` is present in the symbolic heap. In this case, there is no such chunk in the symbolic heap, since the requires clause of `read_bytes` does not mention it and the function also does not get it from anywhere else.

To solve this problem, we need to specify in the requires clause of function `read_bytes` that whoever calls the function must give it permission to write to the place pointed to by `start`, at least if parameter `count` is greater than zero. As a good citizen, we also give the permission back to the caller when we are done with it, as specified in the ensures clause:

```
//@ req if count > 0 { points_to_(start, _) } else { true };
//@ ens if count > 0 { points_to(start, _) } else { true };
```

Notice that in the postcondition, we assert a `points_to` chunk (notice the missing trailing underscore). It asserts that the memory has been *initialized*. Function `write_bytes` will need to assert this chunk in its precondition, because reading uninitialized memory is an error.

By the way, we can write `points_to_` and `points_to` assertions more concisely using special *points-to notation*:

```
//@ req if count > 0 { *start |-> _ } else { true };
//@ ens if count > 0 { *start |-> ?_ } else { true };
```

If we run VeriFast again, we see that VeriFast now accepts the assignment to `*start`. However, it now complains at the recursive call of `read_bytes`. Indeed, if `count` is greater than one, then the recursive call needs permission to access the place pointed to by `start + 1`. Let's extend our contract to reflect this:

```
//@ req if count > 0 { *start |-> _ &*& if count > 1 { *(start + 1) |-> _ } else { true } } else { true };
//@ ens if count > 0 { *start |-> ?_ &*& if count > 1 { *(start + 1) |-> ?_ } else { true } } else { true };
```

Alas, if we run VeriFast again, we notice VeriFast complains again at the recursive call.[14] Indeed, if `count` is greater than 2, the recursive call now needs to permission to access the place pointed to by `start + 2`.

We can patch up the contract again, but will it ever end? If we step back and think about what `read_bytes` does, we realize that a call `read_bytes(start, count)` requires permission to access the locations in the range `start` through `start + count - 1`. How can we express this? If we support only up to 5 characters, we can use the following precondition:

```
/*@
req if count == 0 { true } else {
      *start |-> _ &*&
      if count - 1 == 0 { true } else {
         *(start + 1) |-> _ &*&
         if count - 2 == 0 { true } else {
            *(start + 2) |-> _ &*&
            if count - 3 == 0 { true } else {
```

[14] Remember to disable arithmetic overflow checking in the Verify menu.

```
                *(start + 3) |-> _ &*&
                if count - 4 == 0 { true } else {
                   *(start + 4) |-> _ &*&
                   if count - 5 == 0 { true } else {
                      false}}}}};
@*/
```

If we use the `?_` version of this assertion as the postcondition as well, function `read_bytes` verifies. And if we use the `?_` version of the assertion as the precondition and postcondition for `write_bytes` as well, that function verifies as well. And we can use the `_` version as the postcondition of `alloc` as well, provided that we replace the variable name `start` by `result`. The only problem left now is that VeriFast complains at the end of function `main` that we leak five memory locations. We can tell VeriFast that we are happy to do so by inserting the following ghost statement at the end of the function:

```
/*@
leak
   *array |-> ?_ &*& *(array + 1) |-> ?_ &*& *(array + 2) |-> ?_ &*&
   *(array + 3) |-> ?_ &*& *(array + 4) |-> ?_;
@*/
```

The program now verifies. Great!

Exercise 26 *Modify the program so that it reads 100 characters.*

Did you finish the exercise? No? I don't blame you. Of course, writing these contracts for large counts is totally impractical.

The solution is to use *recursive predicates*. Notice that the precondition of `read_bytes` has a recursive structure. By giving a name to the assertion, and then using this name in the assertion itself, we can make the assertion go on forever:

```
pred bytes_(start: *mut u8, count: usize) =
   if count == 0 { true } else { *start |-> _ &*& bytes_(start + 1, count - 1) };

pred bytes(start: *mut u8, count: usize) =
   if count == 0 { true } else { *start |-> ?_ &*& bytes_(start + 1, count - 1) };
```

You can obtain something very close to the precondition of `read_bytes` by unrolling predicate `bytes_` five times, i.e. by replacing the occurrence of `bytes_` with its definition five times. The only difference is that instead of **`false`**, we get `bytes_(start + 5, count - 5)`. That is, `bytes_` doesn't stop at 5 bytes; it goes on forever.

So, let's solve Exercise 26 by using `bytes_` and `bytes` in the contracts:

```
// verifast_options{ignore_unwind_paths disable_overflow_check}
use std::io::{Read, Write, stdin, stdout};

unsafe fn read_byte() -> u8 /*@ req true; @*/ /*@ ens true; @*/ /*@ assume_correct @*/ { ... }
unsafe fn write_byte(value: u8) /*@ req true; @*/ /*@ ens true; @*/ /*@ assume_correct @*/ { ... }

/*@
pred bytes_(start: *mut u8, count: usize) =
   if count == 0 { true } else { *start |-> _ &*& bytes_(start + 1, count - 1) };
pred bytes(start: *mut u8, count: usize) =
   if count == 0 { true } else { *start |-> ?_ &*& bytes(start + 1, count - 1) };
@*/

unsafe fn alloc(count: usize) -> *mut u8
//@ req 1 <= count;
```

```
//@ ens bytes_(result, count);
//@ assume_correct
{ ... }

unsafe fn read_bytes(start: *mut u8, count: usize)
//@ req bytes_(start, count);
//@ ens bytes(start, count);
{
   if count > 0 {
      let b = read_byte();
      *start = b;
      read_bytes(start.add(1), count - 1);
   }
}

unsafe fn write_bytes(start: *mut u8, count: usize)
//@ req bytes(start, count);
//@ ens bytes(start, count);
{
   if count > 0 {
      let b = *start;
      write_byte(b);
      write_bytes(start.add(1), count - 1);
   }
}

fn main()
//@ req true;
//@ ens true;
{
   unsafe {
      let array = alloc(100);
      read_bytes(array, 100);
      write_bytes(array, 100);
      write_bytes(array, 100);
      //@ leak bytes(array, 100);
   }
}
```

This doesn't quite work yet. The problem is that VeriFast does not automatically replace a `bytes_` or `bytes` chunk with its definition, or vice versa. You need to do this explicitly by inserting **`open`** and **`close`** ghost commands. For example, when you run VeriFast on the above program, VeriFast complains at the assignment to `*start` in function `read_bytes` because it cannot find a chunk matching `points_to_(start, _)`. This chunk is in fact present; it's just that it is hidden inside the `bytes_` chunk. We need to open up the `bytes_` chunk so that VeriFast can see the `points_to_` chunk that is inside of it. The following version of function `read_bytes` verifies:

```
unsafe fn read_bytes(start: *mut u8, count: usize)
//@ req bytes_(start, count);
//@ ens bytes(start, count);
{
   //@ open bytes_(start, count);
   if count > 0 {
      let b = read_byte();
```

```
        *start = b;
        read_bytes(start.add(1), count - 1);
    }
    //@ close bytes(start, count);
}
```

Inserting analogous commands in function `write_bytes` yields a correct solution of Exercise 26.

Exercise 27 *Implement and verify a simple encryption/decryption program. The program should read two arrays of 10 bytes from standard input. It should then replace each byte in the first array with the XOR of the byte and the corresponding byte in the second array. Write a recursive function to do this. It should then write the first array to standard output. The XOR of two bytes `b1` and `b2` can be written in Rust as `b1 ^ b2`).*

26 Looping over an Array

The example program of the previous section, that reads a sequence of 100 bytes from standard input and then writes it twice to standard output, is correct. However, it would probably not work if we attempted to read ten million bytes. This is because functions `read_bytes` and `write_bytes` would in that scenario perform ten million nested recursive calls. This would most likely exhaust the call stack.[15] Therefore, it is safer to rewrite these functions so that they use a loop instead of recursion. Let's rewrite function `read_bytes`:[16]

```
unsafe fn read_bytes(start: *mut u8, count: usize) {
    let mut i = 0;
    loop {
        if i == count { break; }
        let b = read_byte();
        *start.add(i) = b;
        i += 1;
    }
}
```

If we attempt to verify the program now, VeriFast complains that it needs a loop invariant. It needs a loop invariant so that it can verify the loop body only once, starting from a symbolic state that represents the start of an arbitrary iteration of the loop. The loop invariant describes this symbolic state. Specifically, it describes the contents of the symbolic heap, as well as any required information about the value of the local variables that are modified by the loop. (See Section 8 for more information about loops.)

In the example, at the start of each loop iteration, the first `i` bytes have been initialized, and the remaining `count - i` bytes are yet to be initialized. We encode this as follows:

```
unsafe fn read_bytes(start: *mut u8, count: usize)
//@ req bytes_(start, count);
//@ ens bytes(start, count);
{
    let mut i = 0;
    loop {
        //@ inv bytes(start, i) &*& bytes_(start + i, count - i);
        if i == count { break; }
        let b = read_byte();
        *start.add(i) = b;
        i += 1;
```

[15] The Rust compiler might in some cases perform tail call optimization, but we cannot rely on it doing so.

[16] A **for** loop would have been more idiomatic here, but unfortunately VeriFast only supports **loop** loops for now.

```
    }
}
```

VeriFast now complains that it cannot find the `bytes(start, 0)` chunk, while trying to consume the loop invariant upon entry to the loop. We need to explicitly close it. Similarly, we need to open the empty `bytes_(start + count, 0)` chunk before the function returns, to avoid a leak error, and we need to open the `bytes_(start + i, count - i)` chunk before writing to `*start.add(i)`:

```
unsafe fn read_bytes(start: *mut u8, count: usize)
//@ req bytes_(start, count);
//@ ens bytes(start, count);
{
    //@ close bytes(start, 0);
    let mut i = 0;
    loop {
        //@ inv bytes(start, i) &*& bytes_(start + i, count - i);
        if i == count { break; }
        let b = read_byte();
        //@ open bytes_(start + i, count - i);
        *start.add(i) = b;
        i += 1;
    }
    //@ open bytes_(start + count, 0);
}
```

VeriFast now complains that it cannot find the `bytes(start, i + 1)` chunk while trying to consume the loop invariant after symbolically executing the loop body. We do have a `bytes(start, i)` chunk and a `points_to(start + i, _)` chunk, so we do have all memory permissions asserted by assertion `bytes(start, i + 1)`, but they are not in the form expected by VeriFast.

To put the chunks in the expected form, we need to perform `i + 1` **open** operations (including the opening of `bytes(start + i, 0)`), followed by `i + 2` **close** operations (including the closing of `bytes(start + i + 1, 0)`).

To do so, we can write a helper function. A helper function that serves only to perform ghost operations is best written as a *lemma function*. A lemma function is like an ordinary Rust function, except that it starts with the **lem** keyword and it is written inside an annotation:

```
/*@
lem bytes_add_byte(start: *mut u8)
    req bytes(start, ?count) &*& *(start + count) |-> ?_;
    ens bytes(start, count + 1);
{
    open bytes(start, count);
    if count == 0 {
        close bytes(start + 1, 0);
    } else {
        bytes_add_byte(start + 1);
    }
    close bytes(start, count + 1);
}
@*/
```

Notice that this is a recursive lemma function. VeriFast checks that lemma functions terminate. In this case, VeriFast can tell that the recursion terminates because the *derivation depth* of the first chunk asserted by the precondition (i.e. the number of **close** operations that one would need to perform to construct this chunk) decreases at each recursive call.

If we try to verify this lemma function, VeriFast complains that it leaks heap chunks, on a path where `count + 1` equals 0. But this is an infeasible path, because the `count` parameter of a `bytes` chunk is never

negative. However, VeriFast is not automatically aware of that. To make it aware, we need to add an *invariant lemma* for predicate `bytes`:

```
/*@
lem_auto bytes_count_nonneg()
   req bytes(?start, ?count);
   ens bytes(start, count) &*& 0 <= count;
{
   assume(false); // TODO: implement this lemma
}
@*/
```

An invariant lemma is a particular type of *autolemma*, a lemma that will be applied automatically by VeriFast in certain circumstances. Specifically, an invariant lemma for a predicate p is applied automatically whenever a p assertion is produced. We postpone the implementation of this lemma and instead simply write `assume(false);`, for now. Make sure to insert this lemma *before* lemma `bytes_add_byte`, so that the former is available while verifying the latter.

Lemma `bytes_add_byte` now verifies. To complete the verification of function `read_bytes`, we need to insert a call of the lemma at the end of the loop body:

```
unsafe fn read_bytes(start: *mut u8, count: usize)
//@ req bytes_(start, count);
//@ ens bytes(start, count);
{
   //@ close bytes(start, 0);
   let mut i = 0;
   loop {
      //@ inv bytes(start, i) &*& bytes_(start + i, count - i);
      if i == count { break; }
      let b = read_byte();
      //@ open bytes_(start + i, count - i);
      *start.add(i) = b;
      //@ bytes_add_byte(start);
      i += 1;
   }
   //@ open bytes_(start + count, 0);
}
```

Exercise 28 *Implement lemma* `bytes_count_nonneg`*. Also, rewrite function* `write_bytes` *as well to use a loop instead of recursion, and verify it.*

27 Recursive Loop Proofs

An important observation that we can make after the previous section, is that verifying the recursive version of function `read_bytes` is much easier than verifying the version that uses a loop. Indeed, verifying the loop requires us to maintain a loop invariant which describes all of the permissions (i.e. heap chunks) used by the loop. In contrast, the contract of a recursive function describes only the permissions used by a specific call of the function; at the point of a recursive call, if some of the permissions used by the caller are not required by the callee, then these permissions simply sit in the caller's symbolic heap for the duration of the recursive call.

Does this mean that we have to make a choice between run-time performance and verification-time convenience? Fortunately, we do not! A researcher called Thomas Tuerk proposed an approach for verifying loops that is as convenient as verifying the corresponding recursive function. Specifically, Tuerk noticed that any loop can be rewritten into an equivalent recursive function. Consider the following loop:

```
loop {
   if condition { break; }
   ... body ...
}
```

This loop is equivalent to the call

```
iter();
```

where function `iter` is defined as

```
fn iter() {
   if condition { return; }
   ... body ...
   iter();
}
```

As a concrete example, consider function `read_bytes` from the previous section. We can apply the above translation scheme to replace the loop in that function with an equivalent local recursive function:

```
unsafe fn read_bytes(start: *mut u8, count: usize) {
   let mut i = 0;
   fn iter() {
      if i == count { return; }
      let b = read_byte();
      *start.add(i) = b;
      i += 1;
      iter();
   }
   iter();
}
```

Note that this translation scheme uses local functions that access local variables from the enclosing scope. Rust has local functions, but they cannot access variables from the enclosing scope. If they could, however, then this program would work.

After noticing that each loop can be rewritten into an equivalent recursive function, Tuerk realized that this meant that a loop can be verified in the same way that a recursive function is verified: by providing a contract for the recursive function.

VeriFast supports this approach. When verifying a loop, instead of specifying a loop invariant, you can specify a loop contract consisting of a precondition and a postcondition. VeriFast will then verify the loop as if it were written using a local recursive function. To see how this works, let's first verify the version of function `read_bytes` that uses a local recursive function `iter`:

```
unsafe fn read_bytes(start: *mut u8, count: usize)
//@ req bytes_(start, count);
//@ ens bytes(start, count);
{
   let mut i = 0;
   fn iter()
   //@ req bytes_(start + i, count - i);
   //@ ens bytes(start + old_i, count - old_i);
   {
      //@ open bytes_(start + i, count - i);
      if i == count {
         //@ close bytes(start + i, 0);
         return;
      }
```

```
        let b = read_byte();
        *start.add(i) = b;
        i += 1;
        iter();
        //@ close bytes(start + old_i, count - old_i);
    }
    iter();
}
```

Notice that we use variable names prefixed by `old_` to refer to the value of the variable at the start of the function call. VeriFast currently does not support local functions, but if it did, this function would verify successfully.

All that now remains to be done in order to verify the version of `read_bytes` that uses a loop, is to transplant the annotations that we inserted for the recursive function, back into the loop version:

```
unsafe fn read_bytes(start: *mut u8, count: usize)
//@ req bytes_(start, count);
//@ ens bytes(start, count);
{
    let mut i = 0;
    loop {
        /*@
        req bytes_(start + i, count - i);
        ens bytes(start + old_i, count - old_i);
        @*/
        //@ open bytes_(start + i, count - i);
        if i == count {
            //@ close bytes(start + i, 0);
            break;
        }
        let b = read_byte();
        *start.add(i) = b;
        i += 1;
        //@ recursive_call();
        //@ close bytes(start + old_i, count - old_i);
    }
}
```

Notice that we inserted a `recursive_call();` ghost statement to indicate where the imaginary recursive call occurs, and that in contrast to function contracts, loop contracts must be written as a single annotation. VeriFast verifies this function successfully.

Exercise 29 *Write a version of function* `write_bytes` *that uses a loop and verify it using a loop contract.*

Exercise 30 *Verify function* `Stack::get_count` *from Section 11 using a loop contract. You do not need the* `lseg` *predicate or any lemmas for this proof!*

28 Tracking Array Contents

Suppose we wish to verify functional correctness of the following implementation of the `copy_nonoverlapping` function, which copies the contents of one byte array into another one.

```
unsafe fn copy_nonoverlapping(src: *const u8, dst: *mut u8, count: usize) {
    for i in 0..count {
        *dst.add(i) = *src.add(i);
```

```
    }
}
```

To specify functional correctness, the contract for `copy_nonoverlapping` must express that when the function returns, the contents of the `dst` array are equal to the contents of the `src` array. This means that we cannot use the `bytes` predicate from the preceding sections, since that predicate specifies only the size of a byte array, and not its contents. We must add a predicate parameter that specifies the list of bytes held by the array:

```
pred array<T>(p: *T, count: usize; values: list<T>) =
    if count == 0 {
        values == nil
    } else {
        *p |-> ?value &*& array(p + 1, count - 1, ?values0) &*& values == cons(value, values0)
    };
```

(We went ahead and also made the predicate generic in the element type.) We can use this predicate to fully specify the behavior of the `copy_nonoverlapping` function as follows:

```
unsafe fn copy_nonoverlapping(src: *const u8, dst: *mut u8, count: usize)
//@ req [?f]array(src, count, ?bs) &*& array_(dst, count, _);
//@ ens [f]array(src, count, bs) &*& array(dst, count, bs);
```

This contract also uses an `array_` predicate, asserting a possibly uninitialized array, defined as follows:

```
pred array_<T>(p: *T, count: usize; values: list<option<T>>) =
    if count == 0 {
        values == nil
    } else {
        points_to_(p, ?value) &*& array_(p + 1, count - 1, ?values0) &*& values == cons(value, values0)
    };
```

These predicates are in fact (almost[17]) the "official" array predicates in VeriFast: they are declared in `prelude_core.rsspec`, along with a number of useful lemmas, and they are used by VeriFast for certain purposes. For example, the following function verifies given the above contract for `copy_nonoverlapping`:

```
fn main()
//@ req true;
//@ ens true;
{
    unsafe {
        let buffer1: [u8; _] = [10, 20, 30];
        let buffer2 = alloc(Layout::from_size_align_unchecked(3, 1));
        if buffer2.is_null() {
            handle_alloc_error(Layout::from_size_align_unchecked(3, 1));
        }
        copy_nonoverlapping(&raw const buffer1 as *const u8, buffer2, 3);
        //@ open array(buffer2, 3, _);
        std::hint::assert_unchecked(*buffer2.add(1) == 20);
        //@ close array(buffer2, 3, _);
        //@ array_to_array_(buffer2);
        dealloc(buffer2, Layout::from_size_align_unchecked(3, 1));
        //@ array_to_array_(&buffer1 as *u8);
    }
}
```

[17] The official predicates also assert that the pointer is *within the limits of its provenance*, even if the count is zero.

Indeed, declaring a local array causes VeriFast to produce an `array` chunk, and function `alloc` is specified in `rust/std/lib.rsspec` to produce an `array_` chunk.

Notice the semicolon in the definition of these predicates: this means they are declared as *precise* with two input parameters (`p` and `count`) and one output parameter (`values`). As explained in Sections 21 and 22, it follows that VeriFast will merge fractions of these predicates and automatically **`open`** and **`close`** them in certain circumstances.

VeriFast supports *array slice syntax* to improve the readability of assertions about array chunks. The notation `a[i..n] |-> ?vs` is equivalent to `array(a + i, n - i, ?vs)`, and `a[i..n] |-> _` is equivalent to `array_(a + i, n - i, _)`. Using array slice syntax, we can write the contract of function `copy_nonoverlapping` somewhat more readably as follows:

```
unsafe fn copy_nonoverlapping(src: *const u8, dst: *mut u8, count: usize)
//@ req [?f]src[..count] |-> ?bs &*& dst[..count] |-> _;
//@ ens [f]src[..count] |-> bs &*& dst[..count] |-> bs;
```

We can now verify the implementation. As so often when working with arrays, it pays off to use loop contracts instead of loop invariants (see Section 27).

```
unsafe fn copy_nonoverlapping(src: *const u8, dst: *mut u8, count: usize)
//@ req [?f]src[..count] |-> ?bs &*& dst[..count] |-> _;
//@ ens [f]src[..count] |-> bs &*& dst[..count] |-> bs;
{
   let mut i = 0;
   loop {
      /*@
      req [?f1]src[i..count] |-> ?bs1 &*& dst[i..count] |-> _;
      ens [f1]src[old_i..count] |-> bs1 &*& dst[old_i..count] |-> bs1;
      @*/
      //@ open array(src + i, count - i, _);
      //@ open array_(dst + i, count - i, _);
      if i == count { break; }
      *dst.add(i) = *src.add(i);
      i += 1;
   }
}
```

We replaced the **`for`** loop by a **`loop`** loop because VeriFast supports only **`loop`** loops for now.

Exercise 31 *Specify and verify that the following implementation of function* `memcmp` *returns zero if and only if the two arrays have the same contents.*

```
unsafe fn memcmp(p1: *const u8, p2: *const u8, count: usize) -> i32 {
   let mut result = 0;
   let mut i = 0;
   loop {
      if i == count {
         break;
      }
      if *p1.add(i) < *p2.add(i) {
         result = -1;
         break;
      }
      if *p1.add(i) > *p2.add(i) {
         result = 1;
         break;
      }
```

```
      i += 1;
   }
   result
}
```

29 Solutions to Exercises

Note: all of the solutions are also available in directory `rust_tutorials` of the VeriFast distribution.

29.1 Exercise 1

See Figure 10. Note: there are many alternative ways to express this symbolic execution tree that are equivalent. In particular, replacing a symbolic state by an equivalent one yields an equivalent symbolic execution tree. Two symbolic states are equivalent if they denote the same set of concrete states. For example, if there is only one value v for some symbol ς that satisfies the assumptions, then replacing any occurrence of ς by v yields an equivalent symbolic state. For another example, if a symbol is not mentioned by the assumptions, the heap chunks or the local variable bindings, then removing it from the set of used symbols yields an equivalent symbolic state.

29.2 Exercise 2: account.rs

```
// verifast_options{ignore_unwind_paths}
use std::alloc::{Layout, alloc, handle_alloc_error, dealloc};

struct Account {
   balance: i32,
}

impl Account {

   unsafe fn create() -> *mut Account
   //@ req true;
   //@ ens Account_balance(result, 0) &*& alloc_block_Account(result);
   {
      let my_account = alloc(Layout::new::<Account>()) as *mut Account;
      if my_account.is_null() {
         handle_alloc_error(Layout::new::<Account>());
      }
      (*my_account).balance = 0;
      my_account
   }

   unsafe fn set_balance(my_account: *mut Account, new_balance: i32)
   //@ req Account_balance(my_account, _);
   //@ ens Account_balance(my_account, new_balance);
   {
      (*my_account).balance = new_balance;
   }

   unsafe fn dispose(my_account: *mut Account)
   //@ req Account_balance(my_account, _) &*& alloc_block_Account(my_account);
   //@ ens true;
```

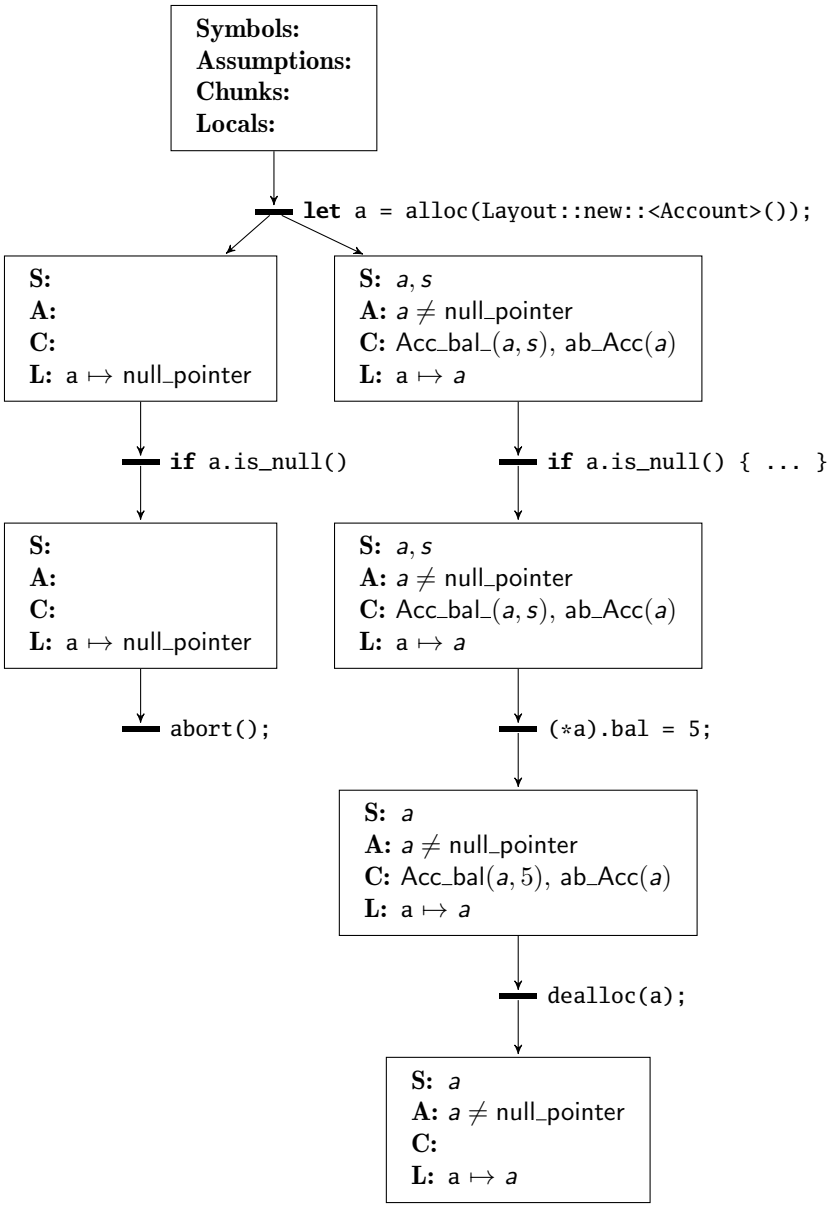

Figure 10: Symbolic execution tree of the program of Figure 1 (after uncommenting the **if** statement). We abbreviate **Chunks** as **C**, **Locals** as **L**, `Account` as `Acc`, `balance` as `bal`, `my_account` as `a`, and alloc_block as `ab`. Note that the tree is shown fully: the `alloc` node has only two child nodes: the second child node summarizes (practically) infinitely many corresponding child nodes from the concrete execution tree. We print symbols in slanted font (*a*) to avoid confusion with similarly named local variables (`a`).

```
    {
        dealloc(my_account as *mut u8, Layout::new::<Account>());
    }

}

fn main()
//@ req true;
//@ ens true;
{
    unsafe {
        let my_account = Account::create();
        Account::set_balance(my_account, 5);
        Account::dispose(my_account);
    }
}
```

29.3 Exercise 3: deposit.rs

```
unsafe fn deposit(my_account: *mut Account, amount: i32)
//@ req Account_balance(my_account, ?theBalance);
//@ ens Account_balance(my_account, theBalance + amount);
{
    (*my_account).balance += amount;
}
```

29.4 Exercise 4: limit.rs

```
// verifast_options{ignore_unwind_paths disable_overflow_check}
use std::alloc::{Layout, alloc, handle_alloc_error, dealloc};
//@ use std::alloc::{alloc_block, Layout};

struct Account {
    limit: i32,
    balance: i32,
}

impl Account {

    unsafe fn create(limit: i32) -> *mut Account
    //@ req limit <= 0;
    //@ ens (*result).limit |-> limit &*& (*result).balance |-> 0 &*& alloc_block_Account(result);
    {
        let my_account = alloc(Layout::new::<Account>()) as *mut Account;
        if my_account.is_null() {
            handle_alloc_error(Layout::new::<Account>());
        }
        (*my_account).limit = limit;
        (*my_account).balance = 0;
        my_account
    }
```

```
    unsafe fn get_balance(my_account: *mut Account) -> i32
    //@ req (*my_account).balance |-> ?theBalance;
    //@ ens (*my_account).balance |-> theBalance &*& result == theBalance;
    {
        (*my_account).balance
    }

    unsafe fn deposit(my_account: *mut Account, amount: i32)
    //@ req (*my_account).balance |-> ?theBalance;
    //@ ens (*my_account).balance |-> theBalance + amount;
    {
        (*my_account).balance += amount;
    }

    unsafe fn withdraw(my_account: *mut Account, amount: i32) -> i32
    //@ req (*my_account).limit |-> ?limit &*& (*my_account).balance |-> ?balance &*& 0 <= amount;
    /*@
    ens (*my_account).limit |-> limit &*& (*my_account).balance |-> balance - result &*&
        result == if balance - amount < limit { balance - limit } else { amount };
    @*/
    {
        let result =
            if (*my_account).balance - amount < (*my_account).limit {
                (*my_account).balance - (*my_account).limit
            } else {
                amount
            };
        (*my_account).balance -= result;
        result
    }

    unsafe fn dispose(my_account: *mut Account)
    //@ req (*my_account).limit |-> _ &*& (*my_account).balance |-> _ &*& alloc_block_Account(my_account);
    //@ ens true;
    {
        dealloc(my_account as *mut u8, Layout::new::<Account>());
    }

}
```

29.5 Exercise 5: pred.rs

```
// verifast_options{ignore_unwind_paths disable_overflow_check}
use std::alloc::{Layout, alloc, handle_alloc_error, dealloc};
//@ use std::alloc::{alloc_block, Layout};

struct Account {
    limit: i32,
    balance: i32,
}
```

```
/*@

pred Account_pred(my_account: *mut Account, theLimit: i32, theBalance: i32) =
   (*my_account).limit |-> theLimit &*& (*my_account).balance |-> theBalance &*&
   alloc_block_Account(my_account);

@*/

impl Account {

   unsafe fn create(limit: i32) -> *mut Account
   //@ req limit <= 0;
   //@ ens Account_pred(result, limit, 0);
   {
      let my_account = alloc(Layout::new::<Account>()) as *mut Account;
      if my_account.is_null() {
         handle_alloc_error(Layout::new::<Account>());
      }
      (*my_account).limit = limit;
      (*my_account).balance = 0;
      //@ close Account_pred(my_account, limit, 0);
      my_account
   }

   unsafe fn get_balance(my_account: *mut Account) -> i32
   //@ req Account_pred(my_account, ?limit, ?balance);
   //@ ens Account_pred(my_account, limit, balance) &*& result == balance;
   {
      //@ open Account_pred(my_account, limit, balance);
      let result = (*my_account).balance;
      //@ close Account_pred(my_account, limit, balance);
      result
   }

   unsafe fn deposit(my_account: *mut Account, amount: i32)
   //@ req Account_pred(my_account, ?limit, ?balance) &*& 0 <= amount;
   //@ ens Account_pred(my_account, limit, balance + amount);
   {
      //@ open Account_pred(my_account, limit, balance);
      (*my_account).balance += amount;
      //@ close Account_pred(my_account, limit, balance + amount);
   }

   unsafe fn withdraw(my_account: *mut Account, amount: i32) -> i32
   //@ req Account_pred(my_account, ?limit, ?balance) &*& 0 <= amount;
   /*@
   ens Account_pred(my_account, limit, balance - result) &*&
      result == if balance - amount < limit { balance - limit } else { amount };
   @*/
   {
      //@ open Account_pred(my_account, limit, balance);
      let result =
         if (*my_account).balance - amount < (*my_account).limit {
```

```
            (*my_account).balance - (*my_account).limit
        } else {
            amount
        };
    (*my_account).balance -= result;
    //@ close Account_pred(my_account, limit, balance - result);
    result
}

unsafe fn dispose(my_account: *mut Account)
//@ req Account_pred(my_account, _, _);
//@ ens true;
{
    //@ open Account_pred(my_account, _, _);
    dealloc(my_account as *mut u8, Layout::new::<Account>());
}

}

fn main()
//@ req true;
//@ ens true;
{
    unsafe {
        let my_account = Account::create(-100);
        Account::deposit(my_account, 200);
        let w1 = Account::withdraw(my_account, 50);
        std::hint::assert_unchecked(w1 == 50);
        let b1 = Account::get_balance(my_account);
        std::hint::assert_unchecked(b1 == 150);
        let w2 = Account::withdraw(my_account, 300);
        std::hint::assert_unchecked(w2 == 250);
        let b2 = Account::get_balance(my_account);
        std::hint::assert_unchecked(b2 == -100);
        Account::dispose(my_account);
    }
}
```

29.6 Exercise 6: stack.rs

```
// verifast_options{ignore_unwind_paths}
use std::alloc::{Layout, alloc, handle_alloc_error, dealloc};
//@ use std::alloc::{Layout, alloc_block};

struct Node {
    next: *mut Node,
    value: i32,
}

struct Stack {
    head: *mut Node,
}
```

```
/*@

pred Nodes(node: *mut Node, count: i32) =
   if node == 0 {
      count == 0
   } else {
      0 < count &*&
      (*node).next |-> ?next &*&
      (*node).value |-> ?value &*&
      alloc_block_Node(node) &*&
      Nodes(next, count - 1)
   };

pred Stack(stack: *mut Stack, count: i32) =
   (*stack).head |-> ?head &*&
   alloc_block_Stack(stack) &*&
   0 <= count &*&
   Nodes(head, count);

@*/

impl Stack {

   unsafe fn create() -> *mut Stack
   //@ req true;
   //@ ens Stack(result, 0);
   {
      let stack = alloc(Layout::new::<Stack>()) as *mut Stack;
      if stack.is_null() {
         handle_alloc_error(Layout::new::<Stack>());
      }
      (*stack).head = std::ptr::null_mut();
      //@ close Nodes(0, 0);
      //@ close Stack(stack, 0);
      stack
   }

   unsafe fn push(stack: *mut Stack, value: i32)
   //@ req Stack(stack, ?count);
   //@ ens Stack(stack, count + 1);
   {
      //@ open Stack(stack, count);
      let n = alloc(Layout::new::<Node>()) as *mut Node;
      if n.is_null() {
         handle_alloc_error(Layout::new::<Node>());
      }
      (*n).next = (*stack).head;
      (*n).value = value;
      (*stack).head = n;
      //@ close Nodes(n, count + 1);
      //@ close Stack(stack, count + 1);
   }
```

```
    unsafe fn pop(stack: *mut Stack) -> i32
    //@ req Stack(stack, ?count) &*& 0 < count;
    //@ ens Stack(stack, count - 1);
    {
        //@ open Stack(stack, count);
        let head = (*stack).head;
        //@ open Nodes(head, count);
        let result = (*head).value;
        (*stack).head = (*head).next;
        dealloc(head as *mut u8, Layout::new::<Node>());
        //@ close Stack(stack, count - 1);
        result
    }

    unsafe fn dispose(stack: *mut Stack)
    //@ req Stack(stack, 0);
    //@ ens true;
    {
        //@ open Stack(stack, 0);
        //@ open Nodes(_, _);
        dealloc(stack as *mut u8, Layout::new::<Stack>());
    }

}

fn main()
//@ req true;
//@ ens true;
{
    unsafe {
        let s = Stack::create();
        Stack::push(s, 10);
        Stack::push(s, 20);
        Stack::pop(s);
        Stack::pop(s);
        Stack::dispose(s);
    }
}
```

29.7 Exercise 7: dispose.rs

```
unsafe fn dispose_nodes(n: *mut Node)
//@ req Nodes(n, _);
//@ ens true;
{
    //@ open Nodes(n, _);
    if !n.is_null() {
        dispose_nodes((*n).next);
        dealloc(n as *mut u8, Layout::new::<Node>());
    }
}
```

```
impl Stack {

   unsafe fn dispose(stack: *mut Stack)
   //@ req Stack(stack, _);
   //@ ens true;
   {
      //@ open Stack(stack, _);
      dispose_nodes((*stack).head);
      dealloc(stack as *mut u8, Layout::new::<Stack>());
   }

}
```

29.8 Exercise 8: sum.rs

```
unsafe fn get_nodes_sum(nodes: *mut Node) -> i32
//@ req Nodes(nodes, ?count);
//@ ens Nodes(nodes, count);
{
   let mut result = 0;
   //@ open Nodes(nodes, count);
   if !nodes.is_null() {
      result = get_nodes_sum((*nodes).next);
      result += (*nodes).value;
   }
   //@ close Nodes(nodes, count);
   result
}

impl Stack {

   unsafe fn get_sum(stack: *mut Stack) -> i32
   //@ req Stack(stack, ?count);
   //@ ens Stack(stack, count);
   {
      //@ open Stack(stack, count);
      let result = get_nodes_sum((*stack).head);
      //@ close Stack(stack, count);
      result
   }

}
```

29.9 Exercise 9: popn.rs

```
unsafe fn popn(stack: *mut Stack, n: i32)
//@ req Stack(stack, ?count) &*& 0 <= n &*& n <= count;
//@ ens Stack(stack, count - n);
{
   let mut i = 0;
```

```
    loop {
        //@ inv Stack(stack, count - i) &*& i <= n;
        if i == n {
            break;
        }
        Stack::pop(stack);
        i += 1;
    }
}
```

29.10 Exercise 10: values.rs

```
// verifast_options{ignore_unwind_paths}
use std::alloc::{Layout, alloc, handle_alloc_error, dealloc};
//@ use std::alloc::{Layout, alloc_block};

struct Node {
    next: *mut Node,
    value: i32,
}

struct Stack {
    head: *mut Node,
}

/*@

inductive i32s = i32s_nil | i32s_cons(i32, i32s);

pred Nodes(node: *mut Node, values: i32s) =
    if node == 0 {
        values == i32s_nil
    } else {
        (*node).next |-> ?next &*& (*node).value |-> ?value &*&
        alloc_block_Node(node) &*& Nodes(next, ?values0) &*&
        values == i32s_cons(value, values0)
    };

pred Stack(stack: *mut Stack, values: i32s) =
    (*stack).head |-> ?head &*& alloc_block_Stack(stack) &*& Nodes(head, values);

@*/

impl Stack {

    unsafe fn create() -> *mut Stack
    //@ req true;
    //@ ens Stack(result, i32s_nil);
    {
        let stack = alloc(Layout::new::<Stack>()) as *mut Stack;
        if stack.is_null() {
            handle_alloc_error(Layout::new::<Stack>());
```

```
        }
        (*stack).head = std::ptr::null_mut();
        //@ close Nodes(0, i32s_nil);
        //@ close Stack(stack, i32s_nil);
        stack
    }

    unsafe fn push(stack: *mut Stack, value: i32)
    //@ req Stack(stack, ?values);
    //@ ens Stack(stack, i32s_cons(value, values));
    {
        //@ open Stack(stack, values);
        let n = alloc(Layout::new::<Node>()) as *mut Node;
        if n.is_null() {
            handle_alloc_error(Layout::new::<Node>());
        }
        (*n).next = (*stack).head;
        (*n).value = value;
        (*stack).head = n;
        //@ close Nodes(n, i32s_cons(value, values));
        //@ close Stack(stack, i32s_cons(value, values));
    }

    unsafe fn dispose(stack: *mut Stack)
    //@ req Stack(stack, i32s_nil);
    //@ ens true;
    {
        //@ open Stack(stack, i32s_nil);
        //@ open Nodes(_, _);
        dealloc(stack as *mut u8, Layout::new::<Stack>());
    }

}
```

29.11 Exercise 11: fixpoints.rs

```
unsafe fn pop(stack: *mut Stack) -> i32
//@ req Stack(stack, ?values) &*& values != i32s_nil;
//@ ens Stack(stack, i32s_tail(values)) &*& result == i32s_head(values);
{
    //@ open Stack(stack, values);
    let head = (*stack).head;
    //@ open Nodes(head, values);
    let result = (*head).value;
    (*stack).head = (*head).next;
    dealloc(head as *mut u8, Layout::new::<Node>());
    //@ close Stack(stack, i32s_tail(values));
    result
}
```

29.12 Exercise 12: sum_full.rs

```
/*@

fix i32s_sum(values: i32s) -> i32 {
   match values {
      i32s_nil => 0,
      i32s_cons(value, values0) => value + i32s_sum(values0),
   }
}

@*/

unsafe fn get_nodes_sum(node: *mut Node) -> i32
//@ req Nodes(node, ?values);
//@ ens Nodes(node, values) &*& result == i32s_sum(values);
{
   //@ open Nodes(node, values);
   let mut result = 0;
   if !node.is_null() {
      let tail_sum = get_nodes_sum((*node).next);
      result = (*node).value + tail_sum;
   }
   //@ close Nodes(node, values);
   result
}

impl Stack {

   unsafe fn get_sum(stack: *mut Stack) -> i32
   //@ req Stack(stack, ?values);
   //@ ens Stack(stack, values) &*& result == i32s_sum(values);
   {
      //@ open Stack(stack, values);
      let result = get_nodes_sum((*stack).head);
      //@ close Stack(stack, values);
      result
   }

}
```

29.13 Exercise 13: lemma.rs

```
lem lseg_to_Nodes_lemma(first: *mut Node)
   req lseg(first, 0, ?count);
   ens Nodes(first, count);
{
   open lseg(first, 0, count);
   if first != 0 {
      lseg_to_Nodes_lemma((*first).next);
   }
   close Nodes(first, count);
}
```

29.14 Exercise 14: push_all.rs

```
/*@

lem lseg_append_lemma(first: *mut Node)
   req lseg(first, ?n, ?count) &*& lseg(n, 0, ?count0);
   ens lseg(first, 0, count + count0);
{
   open lseg(first, n, count);
   if first != n {
      open lseg(n, 0, count0);
      close lseg(n, 0, count0);
      lseg_append_lemma((*first).next);
      close lseg(first, 0, count + count0);
   }
}

@*/

impl Stack {
   unsafe fn push_all(stack: *mut Stack, other: *mut Stack)
   //@ req Stack(stack, ?count) &*& Stack(other, ?count0);
   //@ ens Stack(stack, count0 + count);
   {
      //@ open Stack(stack, count);
      //@ Nodes_to_lseg_lemma((*stack).head);
      //@ open Stack(other, count0);
      //@ Nodes_to_lseg_lemma((*other).head);
      let head0 = (*other).head;
      dealloc(other as *mut u8, Layout::new::<Stack>());
      let mut n = head0;
      //@ open lseg(head0, 0, count0);
      if !n.is_null() {
         //@ close lseg(head0, head0, 0);
         loop {
            /*@
            inv lseg(head0, n, ?count1) &*& n != 0 &*& (*n).value |-> ?n_value &*&
               (*n).next |-> ?next &*&
               alloc_block_Node(n) &*&
               lseg(next, 0, count0 - count1 - 1);
            @*/
            if (*n).next.is_null() {
               break;
            }
            n = (*n).next;
            //@ lseg_add_lemma(head0);
            //@ open lseg(next, 0, count0 - count1 - 1);
         }
         //@ open lseg(0, 0, _);
         (*n).next = (*stack).head;
         //@ lseg_add_lemma(head0);
         //@ lseg_append_lemma(head0);
         (*stack).head = head0;
```

```
        }
        //@ lseg_to_Nodes_lemma((*stack).head);
        //@ close Stack(stack, count0 + count);
    }
}
```

29.15 Exercise 15: reverse.rs

```
fix i32s_append(values1: i32s, values2: i32s) -> i32s {
    match values1 {
        i32s_nil => values2,
        i32s_cons(h, t) => i32s_cons(h, i32s_append(t, values2)),
    }
}

lem i32s_append_nil_lemma(values: i32s)
    req true;
    ens i32s_append(values, i32s_nil) == values;
{
    match values {
        i32s_nil => {}
        i32s_cons(h, t) => {
            i32s_append_nil_lemma(t);
        }
    }
}

lem i32s_append_assoc_lemma(values1: i32s, values2: i32s, values3: i32s)
    req true;
    ens i32s_append(i32s_append(values1, values2), values3)
        == i32s_append(values1, i32s_append(values2, values3));
{
    match values1 {
        i32s_nil => {}
        i32s_cons(h, t) => {
            i32s_append_assoc_lemma(t, values2, values3);
        }
    }
}

fix i32s_reverse(values: i32s) -> i32s {
    match values {
        i32s_nil => i32s_nil,
        i32s_cons(h, t) => i32s_append(i32s_reverse(t), i32s_cons(h, i32s_nil))
    }
}

@*/

impl Stack {
    unsafe fn reverse(stack: *mut Stack)
    //@ req Stack(stack, ?values);
```

```
    //@ ens Stack(stack, i32s_reverse(values));
    {
        //@ open Stack(stack, values);
        let mut n = (*stack).head;
        let mut m = std::ptr::null_mut();
        //@ close Nodes(m, i32s_nil);
        //@ i32s_append_nil_lemma(i32s_reverse(values));
        loop {
            /*@
            inv Nodes(m, ?values1) &*& Nodes(n, ?values2) &*&
                i32s_reverse(values) == i32s_append(i32s_reverse(values2), values1);
            @*/
            if n.is_null() {
                break;
            }
            //@ open Nodes(n, values2);
            let next = (*n).next;
            //@ assert Nodes(next, ?values2_tail) &*& (*n).value |-> ?value;
            (*n).next = m;
            m = n;
            n = next;
            //@ close Nodes(m, i32s_cons(value, values1));
            //@ i32s_append_assoc_lemma(i32s_reverse(values2_tail), i32s_cons(value, i32s_nil), values1);
        }
        //@ open Nodes(n, _);
        (*stack).head = m;
        //@ close Stack(stack, i32s_reverse(values));
    }
}
```

29.16 Exercise 16: filter.rs

```
// verifast_options{ignore_unwind_paths}
use std::alloc::{Layout, alloc, handle_alloc_error, dealloc};
//@ use std::alloc::{Layout, alloc_block};

struct Node {
    next: *mut Node,
    value: i32,
}

struct Stack {
    head: *mut Node,
}

/*@

pred Nodes(node: *mut Node, count: i32) =
    if node == 0 {
        count == 0
    } else {
        0 < count &*&
```

```
      (*node).next |-> ?next &*& (*node).value |-> ?value &*&
      alloc_block_Node(node) &*& Nodes(next, count - 1)
   };

pred Stack(stack: *mut Stack, count: i32) =
   (*stack).head |-> ?head &*& alloc_block_Stack(stack) &*& 0 <= count &*& Nodes(head, count);

fn_type I32Predicate() = unsafe fn(x: i32) -> bool;
   req true;
   ens true;

@*/

type I32Predicate = unsafe fn(i32) -> bool;

unsafe fn filter_nodes(n: *mut Node, p: I32Predicate) -> *mut Node
//@ req Nodes(n, _) &*& [_]is_I32Predicate(p);
//@ ens Nodes(result, _);
{
   if n.is_null() {
      std::ptr::null_mut()
   } else {
      //@ open Nodes(n, _);
      let keep = p((*n).value);
      let next;
      if keep {
         next = filter_nodes((*n).next, p);
         //@ open Nodes(next, ?count);
         //@ close Nodes(next, count);
         (*n).next = next;
         //@ close Nodes(n, count + 1);
         n
      } else {
         next = (*n).next;
         dealloc(n as *mut u8, Layout::new::<Node>());
         let result = filter_nodes(next, p);
         result
      }
   }
}

unsafe fn dispose_nodes(n: *mut Node)
//@ req Nodes(n, _);
//@ ens true;
{
   //@ open Nodes(n, _);
   if !n.is_null() {
      dispose_nodes((*n).next);
      dealloc(n as *mut u8, Layout::new::<Node>());
   }
}

impl Stack {
```

```
unsafe fn create() -> *mut Stack
//@ req true;
//@ ens Stack(result, 0);
{
   let stack = alloc(Layout::new::<Stack>()) as *mut Stack;
   if stack.is_null() {
      handle_alloc_error(Layout::new::<Stack>());
   }
   (*stack).head = std::ptr::null_mut();
   //@ close Nodes(0, 0);
   //@ close Stack(stack, 0);
   stack
}

unsafe fn push(stack: *mut Stack, value: i32)
//@ req Stack(stack, ?count);
//@ ens Stack(stack, count + 1);
{
   //@ open Stack(stack, count);
   let n = alloc(Layout::new::<Node>()) as *mut Node;
   if n.is_null() {
      handle_alloc_error(Layout::new::<Node>());
   }
   (*n).next = (*stack).head;
   (*n).value = value;
   (*stack).head = n;
   //@ close Nodes(n, count + 1);
   //@ close Stack(stack, count + 1);
}

unsafe fn pop(stack: *mut Stack) -> i32
//@ req Stack(stack, ?count) &*& 0 < count;
//@ ens Stack(stack, count - 1);
{
   //@ open Stack(stack, count);
   let head = (*stack).head;
   //@ open Nodes(head, count);
   let result = (*head).value;
   (*stack).head = (*head).next;
   dealloc(head as *mut u8, Layout::new::<Node>());
   //@ close Stack(stack, count - 1);
   result
}

unsafe fn filter(stack: *mut Stack, p: I32Predicate)
//@ req Stack(stack, _) &*& [_]is_I32Predicate(p);
//@ ens Stack(stack, _);
{
   //@ open Stack(stack, _);
   let head = filter_nodes((*stack).head, p);
   //@ assert Nodes(head, ?count);
   (*stack).head = head;
```

```
        //@ open Nodes(head, count);
        //@ close Nodes(head, count);
        //@ close Stack(stack, count);
    }

    unsafe fn dispose(stack: *mut Stack)
    //@ req Stack(stack, _);
    //@ ens true;
    {
        //@ open Stack(stack, _);
        dispose_nodes((*stack).head);
        dealloc(stack as *mut u8, Layout::new::<Stack>());
    }

}

unsafe fn neq_20(x: i32) -> bool
//@ req true;
//@ ens true;
{
    x != 20
}

fn main()
//@ req true;
//@ ens true;
{
    unsafe {
        let s = Stack::create();
        Stack::push(s, 10);
        Stack::push(s, 20);
        /*@
        produce_fn_ptr_chunk I32Predicate(neq_20)()(x) {
            call();
        }
        @*/
        Stack::filter(s, neq_20);
        Stack::dispose(s);
    }
}
```

29.17 Exercise 17: byref.rs

```
unsafe fn filter_nodes(n: *mut *mut Node, p: I32Predicate)
//@ req *n |-> ?node &*& Nodes(node, _) &*& [_]is_I32Predicate(p);
//@ ens *n |-> ?node0 &*& Nodes(node0, _);
{
    if !(*n).is_null() {
        //@ open Nodes(node, _);
        let keep = p((**n).value);
        if keep {
            //@ open Node_next(node, _);
```

```
            filter_nodes(&raw mut (**n).next, p);
            //@ close Node_next(node, ?next);
            //@ open Nodes(next, ?count);
            //@ close Nodes(next, count);
            //@ close Nodes(node, count + 1);
        } else {
            let next_ = (**n).next;
            dealloc(*n as *mut u8, Layout::new::<Node>());
            *n = next_;
            filter_nodes(n, p);
        }
    }
}
```

29.18 Exercise 18: map.rs

```
/*@
fn_type I32Func(data_pred: pred(*mut u8)) = unsafe fn(data: *mut u8, x: i32) -> i32;
    req data_pred(data);
    ens data_pred(data);
@*/

type I32Func = unsafe fn(*mut u8, i32) -> i32;

unsafe fn map_nodes(n: *mut Node, f: I32Func, data: *mut u8)
//@ req Nodes(n, ?count) &*& [_]is_I32Func(f, ?data_pred) &*& data_pred(data);
//@ ens Nodes(n, count) &*& data_pred(data);
{
    //@ open Nodes(n, _);
    if !n.is_null() {
        let y = f(data, (*n).value);
        (*n).value = y;
        map_nodes((*n).next, f, data);
    }
    //@ close Nodes(n, count);
}

impl Stack {
    unsafe fn map(stack: *mut Stack, f: I32Func, data: *mut u8)
    //@ req Stack(stack, ?count) &*& [_]is_I32Func(f, ?data_pred) &*& data_pred(data);
    //@ ens Stack(stack, count) &*& data_pred(data);
    {
        //@ open Stack(stack, _);
        map_nodes((*stack).head, f, data);
        //@ close Stack(stack, count);
    }
}
```

29.19 Exercise 19: foreach.rs

```
unsafe fn input_char() -> char
```

```
//@ req true;
//@ ens true;
//@ assume_correct
{
   let mut line = String::new();
   std::io::stdin().read_line(&mut line).unwrap();
   line.chars().next().unwrap()
}

unsafe fn input_i32() -> i32
//@ req true;
//@ ens true;
//@ assume_correct
{
   let mut line = String::new();
   std::io::stdin().read_line(&mut line).unwrap();
   line.trim().parse().unwrap()
}

unsafe fn output_i32(value: i32)
//@ req true;
//@ ens true;
//@ assume_correct
{
   println!("{}", value);
}

struct Vector {
   x: i32,
   y: i32,
}

//@ pred Vector(v: *mut Vector) = (*v).x |-> ?x &*& (*v).y |-> ?y &*& alloc_block_Vector(v);

impl Vector {

   unsafe fn create(x: i32, y: i32) -> *mut Vector
   //@ req true;
   //@ ens Vector(result);
   {
      let result = alloc(Layout::new::<Vector>()) as *mut Vector;
      if result.is_null() {
         handle_alloc_error(Layout::new::<Vector>());
      }
      (*result).x = x;
      (*result).y = y;
      //@ close Vector(result);
      result
   }

}

fn main()
```

```
//@ req true;
//@ ens true;
{
   unsafe {
      let s = Stack::create();
      //@ close foreach(nil, Vector);
      loop {
         //@ inv Stack(s, ?values) &*& foreach(values, Vector);
         let cmd = input_char();
         match cmd {
            'p' => {
               let x = input_i32();
               let y = input_i32();
               let v = Vector::create(x, y);
               Stack::push(s, v);
               //@ close foreach(cons(v, values), Vector);
            }
            '+' => {
               assert!(!Stack::is_empty(s), "Stack␣underflow");
               let v1 = Stack::pop(s);
               //@ open foreach(values, Vector);
               //@ open Vector(v1);
               assert!(!Stack::is_empty(s), "Stack␣underflow");
               let v2 = Stack::pop(s);
               //@ open foreach(tail(values), Vector);
               //@ open Vector(v2);
               let sum = Vector::create((*v1).x + (*v2).x, (*v1).y + (*v2).y);
               dealloc(v1 as *mut u8, Layout::new::<Vector>());
               dealloc(v2 as *mut u8, Layout::new::<Vector>());
               Stack::push(s, sum);
               //@ close foreach(cons(sum, tail(tail(values))), Vector);
            }
            '=' => {
               assert!(!Stack::is_empty(s), "Stack␣underflow");
               let v_ = Stack::pop(s);
               //@ open foreach(values, Vector);
               //@ open Vector(v_);
               output_i32((*v_).x);
               output_i32((*v_).y);
               dealloc(v_ as *mut u8, Layout::new::<Vector>());
            }
            _ => panic!("Bad␣command")
         }
      }
   }
}
```

29.20 Exercise 20: predctors.rs

```
struct Vector {
   x: i32,
   y: i32,
```

```
}

/*@

pred_ctor Vector(limit: i32)(v: *mut Vector) =
   (*v).x |-> ?x &*& (*v).y |-> ?y &*& alloc_block_Vector(v) &*& x * x + y * y <= limit * limit;

@*/

impl Vector {

   unsafe fn create(limit: i32, x: i32, y: i32) -> *mut Vector
   //@ req true;
   //@ ens Vector(limit)(result);
   {
      assert!(x * x + y * y <= limit * limit, "Vector too big");
      let result = alloc(Layout::new::<Vector>()) as *mut Vector;
      if result.is_null() {
         handle_alloc_error(Layout::new::<Vector>());
      }
      (*result).x = x;
      (*result).y = y;
      //@ close Vector(limit)(result);
      result
   }

}

fn main()
//@ req true;
//@ ens true;
{
   unsafe {
      let limit = input_i32();
      let s = Stack::create();
      //@ close foreach(nil, Vector(limit));
      loop {
         //@ inv Stack(s, ?values) &*& foreach(values, Vector(limit));
         let cmd = input_char();
         match cmd {
            'p' => {
               let x = input_i32();
               let y = input_i32();
               let v = Vector::create(limit, x, y);
               Stack::push(s, v);
               //@ close foreach(cons(v, values), Vector(limit));
            }
            '+' => {
               assert!(!Stack::is_empty(s), "Stack underflow");
               let v1 = Stack::pop(s);
               //@ open foreach(values, Vector(limit));
               //@ open Vector(limit)(v1);
               assert!(!Stack::is_empty(s), "Stack underflow");
```

```
                let v2 = Stack::pop(s);
                //@ open foreach(tail(values), Vector(limit));
                //@ open Vector(limit)(v2);
                let sum = Vector::create(limit, (*v1).x + (*v2).x, (*v1).y + (*v2).y);
                dealloc(v1 as *mut u8, Layout::new::<Vector>());
                dealloc(v2 as *mut u8, Layout::new::<Vector>());
                Stack::push(s, sum);
                //@ close foreach(cons(sum, tail(tail(values))), Vector(limit));
            }
            '=' => {
                assert!(!Stack::is_empty(s), "Stack underflow");
                let v_ = Stack::pop(s);
                //@ open foreach(values, Vector(limit));
                //@ open Vector(limit)(v_);
                std::hint::assert_unchecked((*v_).x * (*v_).x + (*v_).y * (*v_).y <= limit * limit);
                output_i32((*v_).x);
                output_i32((*v_).y);
                dealloc(v_ as *mut u8, Layout::new::<Vector>());
            }
            _ => panic!("Bad command")
        }
    }
  }
}
```

29.21 Exercise 21: threads0.rs

```
// verifast_options{ignore_unwind_paths}
#![allow(unsafe_op_in_unsafe_fn)]
use std::{alloc::{alloc, handle_alloc_error, Layout}};

unsafe fn wrapping_fib(n: u16) -> u64
//@ req true;
//@ ens true;
{
   if n <= 1 {
      1
   } else {
      let mut k: u16 = 2;
      let mut fib_k_minus_1: u64 = 1;
      let mut fib_k: u64 = 1;
      loop {
         //@ inv k <= n;

         if k == n { break; }

         let fib_k_plus_1 = fib_k_minus_1.wrapping_add(fib_k);

         k += 1;
         fib_k_minus_1 = fib_k;
         fib_k = fib_k_plus_1;
      }
```

```
      fib_k
   }
}

struct Tree {
   left: *mut Tree,
   right: *mut Tree,
   value: u16,
}

/*@

pred Tree(t: *mut Tree, depth: u8) =
   if t == 0 {
      depth == 0
   } else {
      (*t).left |-> ?left &*& Tree(left, depth - 1) &*&
      (*t).right |-> ?right &*& Tree(right, depth - 1) &*&
      (*t).value |-> ?value &*&
      alloc_block_Tree(t)
   };

@*/

impl Tree {

   unsafe fn make(depth: u8) -> *mut Tree
   //@ req true;
   //@ ens Tree(result, depth);
   {
      if depth == 0 {
         //@ close Tree(0, 0);
         std::ptr::null_mut()
      } else {
         let left = Self::make(depth - 1);
         let right = Self::make(depth - 1);
         let value = 5000; // TODO: Use a random number here
         let t = alloc(Layout::new::<Tree>()) as *mut Tree;
         if t.is_null() {
            handle_alloc_error(Layout::new::<Tree>());
         }
         (*t).left = left;
         (*t).right = right;
         (*t).value = value;
         //@ close Tree(t, depth);
         t
      }
   }

   unsafe fn compute_sum_fibs(tree: *mut Tree) -> u64
   //@ req Tree(tree, ?depth);
   //@ ens Tree(tree, depth);
   {
```

```
      if tree.is_null() {
         0
      } else {
         //@ open Tree(tree, depth);
         let left_sum = Self::compute_sum_fibs((*tree).left);
         let f = wrapping_fib((*tree).value);
         let right_sum = Self::compute_sum_fibs((*tree).right);
         //@ close Tree(tree, depth);
         left_sum.wrapping_add(f).wrapping_add(right_sum)
      }
   }

}

unsafe fn print_u64(value: u64)
//@ req true;
//@ ens true;
//@ assume_correct
{
   println!("{}", value);
}

fn main()
//@ req true;
//@ ens true;
{
   unsafe {
      let tree = Tree::make(22);
      let sum = Tree::compute_sum_fibs(tree);
      //@ leak Tree(tree, 22);
      print_u64(sum)
   }
}
```

29.22 Exercise 22: threads.rs

```
/*@
pred_ctor compute_sum_fibs_post(tree: *mut Tree, depth: i32)(result: u64) = Tree(tree, depth);
pred compute_sum_fibs_pre(tree: *mut Tree, post: pred(u64)) =
   Tree(tree, ?depth) &*& post == compute_sum_fibs_post(tree, depth);
@*/

fn main()
//@ req true;
//@ ens true;
{
   unsafe {
      let tree = Tree::make(22);
      //@ open Tree(tree, 22);
      let left = (*tree).left;
      let right = (*tree).right;
      /*@
```

```
        produce_fn_ptr_chunk Spawnee<*mut Tree, u64>(Tree::compute_sum_fibs)
            (compute_sum_fibs_pre)(arg) {
            open compute_sum_fibs_pre(arg, _);
            assert Tree(arg, ?depth);
            let result = call();
            close compute_sum_fibs_post(arg, depth)(result);
        }
        @*/
        //@ close compute_sum_fibs_pre(left, _);
        let left_join_handle = spawn(Tree::compute_sum_fibs, left);
        //@ close compute_sum_fibs_pre(right, _);
        let right_join_handle = spawn(Tree::compute_sum_fibs, right);
        let root_fib = wrapping_fib((*tree).value);

        let left_sum = join(left_join_handle);
        //@ open compute_sum_fibs_post(left, 21)(_);
        let right_sum = join(right_join_handle);
        //@ open compute_sum_fibs_post(right, 21)(_);
        let sum = left_sum.wrapping_add(root_fib).wrapping_add(right_sum);

        //@ close Tree(tree, 22);
        //@ leak Tree(tree, 22);
        print_u64(sum)
    }
}
```

29.23 Exercise 23: fractions.rs

```
/*@
pred_ctor inspect_tree_post(tree: *mut Tree, depth: i32)(result: u64) =
    [1/2]Tree(tree, depth);
pred inspect_tree_pre(tree: *mut Tree, post: pred(u64)) =
    [1/2]Tree(tree, ?depth) &*& post == inspect_tree_post(tree, depth);
@*/

fn main()
//@ req true;
//@ ens true;
{
    unsafe {
        let tree = Tree::make(22);

        /*@
        produce_fn_ptr_chunk Spawnee<*mut Tree, u64>(Tree::compute_sum_fibs)
            (inspect_tree_pre)(arg) {
            open inspect_tree_pre(arg, _);
            assert [1/2]Tree(arg, ?depth);
            let result = call();
            close inspect_tree_post(arg, depth)(result);
        }
        @*/
        //@ close inspect_tree_pre(tree, _);
```

```
    let sum_join_handle = spawn(Tree::compute_sum_fibs, tree);

    /*@
    produce_fn_ptr_chunk Spawnee<*mut Tree, u64>(Tree::compute_product_fibs)
       (inspect_tree_pre)(arg) {
       open inspect_tree_pre(arg, _);
       assert [1/2]Tree(arg, ?depth);
       let result = call();
       close inspect_tree_post(arg, depth)(result);
    }
    @*/
    //@ close inspect_tree_pre(tree, _);
    let product_join_handle = spawn(Tree::compute_product_fibs, tree);

    let sum = join(sum_join_handle);
    //@ open inspect_tree_post(tree, 22)(_);
    //@ leak [1/2]Tree(tree, 22);

    let product = join(product_join_handle);
    //@ open inspect_tree_post(tree, 22)(_);
    //@ leak [1/2]Tree(tree, 22);

    print_u64(sum);
    print_u64(product);
  }
}
```

29.24 Exercise 24: mutexes.rs

```
// verifast_options{ignore_unwind_paths}
use std::alloc::{Layout, alloc, handle_alloc_error};

/*@
fn_type Spawnee<T>(pre: pred(T)) = unsafe fn(arg: T);
  req pre(arg);
  ens true;
@*/

struct Sendable<T> { payload: T }
unsafe impl<T> Send for Sendable<T> {}

unsafe fn spawn<T: 'static>(f: unsafe fn(arg: T), arg: T)
//@ req [_]is_Spawnee(f, ?pre) &*& pre(arg);
//@ ens true;
//@ assume_correct
{
  let package = Sendable { payload: arg };
  std::thread::spawn(move || {
    let package_moved = package;
    f(package_moved.payload)
  });
}
```

```
type Mutex = std::sync::Mutex<()>;
type MutexGuard = std::sync::MutexGuard<'static, ()>;

//@ pred Mutex(mutex: *mut Mutex; inv_: pred());
//@ pred MutexGuard(guard: MutexGuard, mutex: *mut Mutex, inv_: pred(), frac: real, t: thread_id_t);

unsafe fn create_mutex() -> *mut Mutex
//@ req exists::<pred()>(?inv_) &*& inv_();
//@ ens Mutex(result, inv_);
//@ assume_correct
{
    let mutex = alloc(Layout::new::<Mutex>()) as *mut Mutex;
    if mutex.is_null() { handle_alloc_error(Layout::new::<Mutex>()); }
    mutex.write(Mutex::new(()));
    mutex
}

unsafe fn acquire(mutex: *mut Mutex) -> MutexGuard
//@ req [?frac]Mutex(mutex, ?inv_);
//@ ens MutexGuard(result, mutex, inv_, frac, currentThread) &*& inv_();
//@ assume_correct
{
    (*mutex).lock().unwrap()
}

unsafe fn release(guard: MutexGuard)
//@ req MutexGuard(guard, ?mutex, ?inv_, ?frac, currentThread) &*& inv_();
//@ ens [frac]Mutex(mutex, inv_);
//@ assume_correct
{
    drop(guard);
}

unsafe fn wait_for_pulse(_source: i32)
//@ req true;
//@ ens true;
{
    // For simplicity, instead of actually waiting for input
    // from a sensor, we just sleep for 500ms.
    std::thread::sleep(std::time::Duration::from_millis(500));
}

unsafe fn print_u32(n: u32)
//@ req true;
//@ ens true;
//@ assume_correct
{
    println!("{}", n);
}

//@ pred_ctor counter_inv(counter: *mut u32)() = *counter |-> ?count;
```

```
struct CountPulsesData {
   counter: *mut u32,
   mutex: *mut Mutex,
   source: i32,
}

//@ pred count_pulses_pre(data: CountPulsesData) = [1/3]Mutex(data.mutex, counter_inv(data.counter));

unsafe fn count_pulses(data: CountPulsesData)
//@ req count_pulses_pre(data);
//@ ens true;
{
   //@ open count_pulses_pre(data);
   let CountPulsesData {counter, mutex, source} = data;

   loop {
      //@ inv [1/3]Mutex(mutex, counter_inv(counter));
      wait_for_pulse(source);
      let guard = acquire(mutex);
      //@ open counter_inv(counter)();
      *counter = (*counter).checked_add(1).unwrap();
      //@ close counter_inv(counter)();
      release(guard);
   }
}

unsafe fn count_pulses_async(counter: *mut u32, mutex: *mut Mutex, source: i32)
//@ req [1/3]Mutex(mutex, counter_inv(counter));
//@ ens true;
{
   let data = CountPulsesData { counter, mutex, source };
   //@ close count_pulses_pre(data);
   //@ produce_fn_ptr_chunk Spawnee<CountPulsesData>(count_pulses)(count_pulses_pre)(arg) { call(); }
   spawn(count_pulses, data);
}

fn main()
//@ req true;
//@ ens true;
{
   unsafe {
      let counter = alloc(Layout::new::<u32>()) as *mut u32;
      if counter.is_null() { handle_alloc_error(Layout::new::<u32>()); }
      *counter = 0;
      //@ close counter_inv(counter)();
      //@ close exists(counter_inv(counter));
      let mutex = create_mutex();

      count_pulses_async(counter, mutex, 1);
      count_pulses_async(counter, mutex, 2);

      loop {
         //@ inv [1/3]Mutex(mutex, counter_inv(counter));
```

```
            std::thread::sleep(std::time::Duration::from_millis(1000));
            let guard = acquire(mutex);
            //@ open counter_inv(counter)();
            let count = *counter;
            //@ close counter_inv(counter)();
            release(guard);
            print_u32(count);
        }
    }
}
```

29.25 Exercise 25: leaks.rs

```
// verifast_options{ignore_unwind_paths}
#![allow(unsafe_op_in_unsafe_fn)]
use std::alloc::{alloc, handle_alloc_error, Layout};

/*@
fn_type Spawnee<T>(pre: pred(T)) = unsafe fn(arg: T);
    req pre(arg);
    ens true;
@*/

struct Sendable<T> { payload: T }
unsafe impl<T> Send for Sendable<T> {}

unsafe fn spawn<T: 'static>(f: unsafe fn(arg: T), arg: T)
//@ req [_]is_Spawnee(f, ?pre) &*& pre(arg);
//@ ens true;
//@ assume_correct
{
    let package = Sendable { payload: arg };
    std::thread::spawn(move || {
        let package_moved = package;
        f(package_moved.payload)
    });
}

type Mutex = std::sync::Mutex<()>;
type MutexGuard = std::sync::MutexGuard<'static, ()>;

//@ pred Mutex(mutex: *mut Mutex; inv_: pred());
//@ pred MutexGuard(guard: MutexGuard, mutex: *mut Mutex, inv_: pred(), frac: real, t: thread_id_t);

unsafe fn create_mutex() -> *mut Mutex
//@ req exists::<pred()>(?inv_) &*& inv_();
//@ ens Mutex(result, inv_);
//@ assume_correct
{
    let mutex = alloc(Layout::new::<Mutex>()) as *mut Mutex;
    if mutex.is_null() { handle_alloc_error(Layout::new::<Mutex>()); }
    mutex.write(Mutex::new(()));
```

```
    mutex
}

unsafe fn acquire(mutex: *mut Mutex) -> MutexGuard
//@ req [?frac]Mutex(mutex, ?inv_);
//@ ens MutexGuard(result, mutex, inv_, frac, currentThread) &*& inv_();
//@ assume_correct
{
    (*mutex).lock().unwrap()
}

unsafe fn release(guard: MutexGuard)
//@ req MutexGuard(guard, ?mutex, ?inv_, ?frac, currentThread) &*& inv_();
//@ ens [frac]Mutex(mutex, inv_);
//@ assume_correct
{
    drop(guard);
}

unsafe fn wait_for_source() -> i32
//@ req true;
//@ ens true;
{
    std::thread::sleep(std::time::Duration::from_millis(500));
    42
}

/// `true` means the sensor has been removed.
unsafe fn wait_for_pulse(_source: i32) -> bool
//@ req true;
//@ ens true;
{
    std::thread::sleep(std::time::Duration::from_millis(500));
    false
}

unsafe fn print_u32(n: u32)
//@ req true;
//@ ens true;
//@ assume_correct
{
    println!("{}", n);
}

//@ pred_ctor Counter(counter: *mut u32)() = *counter |-> ?count;

struct CountPulsesData {
    counter: *mut u32,
    mutex: *mut Mutex,
    source: i32,
}

//@ pred count_pulses_pre(data: CountPulsesData) = [_]Mutex(data.mutex, Counter(data.counter));
```

```
unsafe fn count_pulses(data: CountPulsesData)
//@ req count_pulses_pre(data);
//@ ens true;
{
   //@ open count_pulses_pre(data);
   let CountPulsesData {counter, mutex, source} = data;

   loop {
      //@ inv [_]Mutex(mutex, Counter(counter));
      let done = wait_for_pulse(source);
      if done { break }
      let guard = acquire(mutex);
      //@ open Counter(counter)();
      *counter = (*counter).checked_add(1).unwrap();
      //@ close Counter(counter)();
      release(guard);
   }
}

unsafe fn count_pulses_async(counter: *mut u32, mutex: *mut Mutex, source: i32)
//@ req [_]Mutex(mutex, Counter(counter));
//@ ens true;
{
   let data = CountPulsesData { counter, mutex, source };
   //@ close count_pulses_pre(data);
   //@ produce_fn_ptr_chunk Spawnee<CountPulsesData>(count_pulses)(count_pulses_pre)(data_) { call(); }
   spawn(count_pulses, data);
}

struct PrintCountData {
   counter: *mut u32,
   mutex: *mut Mutex,
}

//@ pred print_count_pre(data: PrintCountData) = [_]Mutex(data.mutex, Counter(data.counter));

unsafe fn print_count(data: PrintCountData)
//@ req print_count_pre(data);
//@ ens true;
{
   //@ open print_count_pre(data);
   let PrintCountData {counter, mutex} = data;
   loop {
      //@ inv [_]Mutex(mutex, Counter(counter));
      std::thread::sleep(std::time::Duration::from_millis(1000));
      let guard = acquire(mutex);
      //@ open Counter(counter)();
      print_u32(*counter);
      //@ close Counter(counter)();
      release(guard);
   }
}
```

```
unsafe fn print_count_async(counter: *mut u32, mutex: *mut Mutex)
//@ req [_]Mutex(mutex, Counter(counter));
//@ ens true;
{
   let data = PrintCountData { counter, mutex };
   //@ close print_count_pre(data);
   //@ produce_fn_ptr_chunk Spawnee<PrintCountData>(print_count)(print_count_pre)(data_) { call(); }
   spawn(print_count, data);
}

fn main()
//@ req true;
//@ ens true;
{
   unsafe {
      let counter = alloc(Layout::new::<u32>()) as *mut u32;
      if counter.is_null() {
         handle_alloc_error(Layout::new::<u32>());
      }
      *counter = 0;
      //@ close Counter(counter)();
      //@ close exists(Counter(counter));
      let mutex = create_mutex();
      //@ leak Mutex(mutex, _);

      print_count_async(counter, mutex);

      loop {
         //@ inv [_]Mutex(mutex, Counter(counter));
         let source = wait_for_source();
         count_pulses_async(counter, mutex, source);
      }
   }
}
```

29.26 Exercise 26: bytes.rs

```
// verifast_options{ignore_unwind_paths disable_overflow_check}
use std::io::{Read, Write, stdin, stdout};

unsafe fn read_byte() -> u8
//@ req true;
//@ ens true;
//@ assume_correct
{
   let mut buf = [0u8];
   stdin().read_exact(&mut buf[..]).unwrap();
   buf[0]
}

unsafe fn write_byte(value: u8)
```

```
//@ req true;
//@ ens true;
//@ assume_correct
{
   let buf = [value];
   stdout().write(&buf[..]).unwrap();
}

/*@

pred bytes_(start: *mut u8, count: usize) =
   if count == 0 { true } else { *start |-> _ &*& bytes_(start + 1, count - 1) };

pred bytes(start: *mut u8, count: usize) =
   if count == 0 { true } else { *start |-> ?_ &*& bytes(start + 1, count - 1) };

@*/

unsafe fn alloc(count: usize) -> *mut u8
//@ req 1 <= count;
//@ ens bytes_(result, count);
//@ assume_correct
{
   let layout = std::alloc::Layout::from_size_align(count, 1).unwrap();
   let result = std::alloc::alloc(layout);
   if result.is_null() {
      std::alloc::handle_alloc_error(layout);
   }
   result
}

unsafe fn read_bytes(start: *mut u8, count: usize)
//@ req bytes_(start, count);
//@ ens bytes(start, count);
{
   //@ open bytes_(start, count);
   if count > 0 {
      let b = read_byte();
      *start = b;
      read_bytes(start.add(1), count - 1);
   }
   //@ close bytes(start, count);
}

unsafe fn write_bytes(start: *mut u8, count: usize)
//@ req bytes(start, count);
//@ ens bytes(start, count);
{
   if count > 0 {
      //@ open bytes(start, count);
      let b = *start;
      write_byte(b);
      write_bytes(start.add(1), count - 1);
```

```
      //@ close bytes(start, count);
   }
}

fn main()
//@ req true;
//@ ens true;
{
   unsafe {
      let array = alloc(100);
      read_bytes(array, 100);
      write_bytes(array, 100);
      write_bytes(array, 100);
      //@ leak bytes(array, 100);
   }
}
```

29.27 Exercise 27: xor.rs

```
// verifast_options{ignore_unwind_paths disable_overflow_check}
use std::io::{Read, Write, stdin, stdout};

unsafe fn read_byte() -> u8
//@ req true;
//@ ens true;
//@ assume_correct
{
   let mut buf = [0u8];
   stdin().read_exact(&mut buf[..]).unwrap();
   buf[0]
}

unsafe fn write_byte(value: u8)
//@ req true;
//@ ens true;
//@ assume_correct
{
   let buf = [value];
   stdout().write(&buf[..]).unwrap();
}

/*@

pred bytes_(start: *mut u8, count: usize) =
   if count == 0 { true } else { *start |-> _ &*& bytes_(start + 1, count - 1) };

pred bytes(start: *mut u8, count: usize) =
   if count == 0 { true } else { *start |-> ?_ &*& bytes(start + 1, count - 1) };

@*/

unsafe fn alloc(count: usize) -> *mut u8
```

```
//@ req 1 <= count;
//@ ens bytes_(result, count);
//@ assume_correct
{
   let layout = std::alloc::Layout::from_size_align(count, 1).unwrap();
   let result = std::alloc::alloc(layout);
   if result.is_null() {
      std::alloc::handle_alloc_error(layout);
   }
   result
}

unsafe fn read_bytes(start: *mut u8, count: usize)
//@ req bytes_(start, count);
//@ ens bytes(start, count);
{
   //@ open bytes_(start, count);
   if count > 0 {
      let b = read_byte();
      *start = b;
      read_bytes(start.add(1), count - 1);
   }
   //@ close bytes(start, count);
}

unsafe fn xor_bytes(text: *mut u8, key: *mut u8, count: usize)
//@ req bytes(text, count) &*& bytes(key, count);
//@ ens bytes(text, count) &*& bytes(key, count);
{
   if count > 0 {
      //@ open bytes(text, count);
      //@ open bytes(key, count);
      let t = *text;
      let k = *key;
      *text = t ^ k;
      xor_bytes(text.add(1), key.add(1), count - 1);
      //@ close bytes(text, count);
      //@ close bytes(key, count);
   }
}

unsafe fn write_bytes(start: *mut u8, count: usize)
//@ req bytes(start, count);
//@ ens bytes(start, count);
{
   if count > 0 {
      //@ open bytes(start, count);
      let b = *start;
      write_byte(b);
      write_bytes(start.add(1), count - 1);
      //@ close bytes(start, count);
   }
}
```

```
fn main()
//@ req true;
//@ ens true;
{
   unsafe {
      let text = alloc(10);
      let key = alloc(10);
      read_bytes(text, 10);
      read_bytes(key, 10);
      xor_bytes(text, key, 10);
      write_bytes(text, 10);
      //@ leak bytes(text, 10) &*& bytes(key, 10);
   }
}
```

29.28 Exercise 28: bytes_loop.c

```
// verifast_options{ignore_unwind_paths disable_overflow_check}
use std::io::{Read, Write, stdin, stdout};

unsafe fn read_byte() -> u8
//@ req true;
//@ ens true;
//@ assume_correct
{
   let mut buf = [0u8];
   stdin().read_exact(&mut buf[..]).unwrap();
   buf[0]
}

unsafe fn write_byte(value: u8)
//@ req true;
//@ ens true;
//@ assume_correct
{
   let buf = [value];
   stdout().write(&buf[..]).unwrap();
}

/*@

pred bytes_(start: *mut u8, count: usize) =
   if count == 0 { true } else { *start |-> _ &*& bytes_(start + 1, count - 1) };

pred bytes(start: *mut u8, count: usize) =
   if count == 0 { true } else { *start |-> ?_ &*& bytes(start + 1, count - 1) };

@*/

unsafe fn alloc(count: usize) -> *mut u8
//@ req 1 <= count;
```

```
//@ ens bytes_(result, count);
//@ assume_correct
{
   let layout = std::alloc::Layout::from_size_align(count, 1).unwrap();
   let result = std::alloc::alloc(layout);
   if result.is_null() {
      std::alloc::handle_alloc_error(layout);
   }
   result
}

/*@

lem_auto bytes_count_nonneg()
   req bytes(?start, ?count);
   ens bytes(start, count) &*& 0 <= count;
{
   open bytes(start, count);
   if count != 0 {
      bytes_count_nonneg();
   }
   close bytes(start, count);
}

lem bytes_add_byte(start: *mut u8)
   req bytes(start, ?count) &*& *(start + count) |-> ?_;
   ens bytes(start, count + 1);
{
   open bytes(start, count);
   if count == 0 {
      close bytes(start + 1, 0);
   } else {
      bytes_add_byte(start + 1);
   }
   close bytes(start, count + 1);
}

@*/

unsafe fn read_bytes(start: *mut u8, count: usize)
//@ req bytes_(start, count);
//@ ens bytes(start, count);
{
   //@ close bytes(start, 0);
   let mut i = 0;
   loop {
      //@ inv bytes(start, i) &*& bytes_(start + i, count - i);
      if i == count { break; }
      let b = read_byte();
      //@ open bytes_(start + i, count - i);
      *start.add(i) = b;
      //@ bytes_add_byte(start);
      i += 1;
```

```
    }
    //@ open bytes_(start + count, 0);
}

unsafe fn write_bytes(start: *mut u8, count: usize)
//@ req bytes(start, count);
//@ ens bytes(start, count);
{
    //@ close bytes(start, 0);
    let mut i = 0;
    loop {
        //@ inv bytes(start, i) &*& bytes(start + i, count - i);
        if i == count { break; }
        //@ open bytes(start + i, count - i);
        let b = *start.add(i);
        //@ bytes_add_byte(start);
        write_byte(b);
        i += 1;
    }
    //@ open bytes(start + count, 0);
}

fn main()
//@ req true;
//@ ens true;
{
    unsafe {
        let array = alloc(100);
        read_bytes(array, 100);
        write_bytes(array, 100);
        write_bytes(array, 100);
        //@ leak bytes(array, 100);
    }
}
```

29.29 Exercise 29: tuerk.rs

```
unsafe fn write_bytes(start: *mut u8, count: usize)
//@ req bytes(start, count);
//@ ens bytes(start, count);
{
    let mut i = 0;
    loop {
        /*@
        req bytes(start + i, count - i);
        ens bytes(start + old_i, count - old_i);
        @*/
        if i == count { break; }
        //@ open bytes(start + i, count - i);
        write_byte(*start.add(i));
        i += 1;
        //@ recursive_call();
```

```
        //@ close bytes(start + old_i, count - old_i);
    }
}
```

29.30 Exercise 30: stack_tuerk.rs

```
unsafe fn stack_get_count(stack: *mut Stack) -> i32
//@ req Stack(stack, ?count);
//@ ens Stack(stack, count) &*& result == count;
{
    //@ open Stack(stack, count);
    let mut n = (*stack).head;
    let mut i = 0;
    loop {
        /*@
        req Nodes(n, ?count1);
        ens Nodes(old_n, count1) &*& i == old_i + count1;
        @*/
        //@ open Nodes(n, count1);
        if n.is_null() {
            //@ close Nodes(n, count1);
            break;
        }
        n = (*n).next;
        i += 1;
        //@ recursive_call();
        //@ close Nodes(old_n, count1);
    }
    //@ close Stack(stack, count);
    i
}
```

29.31 Exercise 31: memcmp.rs

```
unsafe fn memcmp(p1: *const u8, p2: *const u8, count: usize) -> i32
//@ req [?f1]p1[0..count] |-> ?cs1 &*& [?f2]p2[0..count] |-> ?cs2;
//@ ens [f1]p1[0..count] |-> cs1 &*& [f2]p2[0..count] |-> cs2 &*& (result == 0) == (cs1 == cs2);
{
    let mut result = 0;
    let mut i = 0;
    loop {
        /*@
        req [f1]p1[i..count] |-> ?xs1 &*& [f2]p2[i..count] |-> ?xs2 &*& result == 0;
        ens [f1]p1[old_i..count] |-> xs1 &*& [f2]p2[old_i..count] |-> xs2 &*& (result == 0) == (xs1 == xs2);
        @*/
        //@ open array(p1 + i, _, _);
        //@ open array(p2 + i, _, _);
        if i == count {
            break;
        }
        if *p1.add(i) < *p2.add(i) {
```

```
            result = -1;
            break;
        }
        if *p1.add(i) > *p2.add(i) {
            result = 1;
            break;
        }
        i += 1;
    }
    result
}
```