The VeriFast Program Verifier: A Tutorial

Bart Jacobs Jan Smans
Frank Piessens
imec-DistriNet, Department of Computer Science, KU Leuven - University of Leuven, Belgium

February 15, 2017

Contents

1 Introduction

VeriFast is a program verification tool for verifying certain correctness properties of single-threaded and multithreaded C[1] programs. The tool reads a C program consisting of one or more .c source code files (plus any .h header files referenced from these .c files) and reports either "0 errors found" or indicates the location of a potential error. If the tool reports "0 errors found", this means[2] that the program

- does not perform illegal memory accesses, such as reading or writing a struct instance field after the struct instance has been freed, 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 C programs, 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, many security-critical and safety-critical 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. are written in C, 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 step by step. To try the examples and exercises in this tutorial yourself, please download the release from the VeriFast website at

`http://www.cs.kuleuven.be/~bartj/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`).

2 Example: illegal_access.c

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 C program that contains a subtle error.

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

[1] VeriFast also supports Java. See *VeriFast for Java: A Tutorial*.

[2] There are a few known reasons (known as *unsoundnesses*) why the tool may sometimes incorrectly report "0 errors found"; see soundness.md at https://github.com/verifast/verifast. There may also be unknown unsoundnesses.

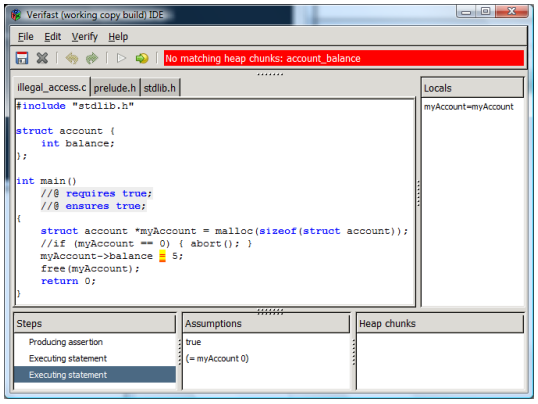

Figure 1: A screenshot of `illegal_access.c` in the VeriFast IDE

`http://www.cs.kuleuven.be/~bartj/verifast/illegal_access.c`

. 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 `myAccount` allocated using `malloc`. However, if there is insufficient memory, `malloc` returns zero 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". 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 balance field of an instance of struct 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 three kinds of nodes:

- 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 *branch 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 one case split: symbolic execution of a `malloc` call forks into one branch where no memory is available and therefore `malloc` returns a null pointer, and another branch where memory is available and therefore `malloc` allocates the requested amount of memory and returns a pointer to it. (A case split also happens when symbolically executing the **`if`** statement, but since the two cases of the **`if`** statement coincide with the two cases of the `malloc` statement, no separate branch node is shown.)
- There is one *leaf node* at the end of each complete symbolic execution path through the function. Click any leaf node to see the full corresponding symbolic execution path in the Steps pane. Function `main` has two symbolic execution paths: the path where no memory is available ends when the program ends due to the call of `abort`; the path where memory is available ends when the function returns.

Click the rightmost leaf node to view the execution path where `malloc` 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 `malloc` 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 `malloc` statement. This statement affects the symbolic state in three ways:

- It adds the heap chunks account_balance(myAccount, value) and malloc_block_account(myAccount) to the symbolic heap (as shown in the Heap chunks pane). Here, myAccount and value are *symbols* that represent unknown values. Specifically, myAccount represents the memory address of the newly allocated struct instance, and value represents the initial value of the `balance` field of the struct instance. (In C, the initial values of the fields of a newly allocated struct instance are unspecified, unlike in Java where the fields of a new object are initialized to the default value of their type.)

  VeriFast *freshly picks* these symbols during this symbolic execution step. That is, to represent the address of the new struct instance and the initial value 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:

```
void test()
   //@ requires true;
   //@ ensures true;
{
   {
      struct account *myAccount = malloc(sizeof(struct account));
      if (myAccount == 0) { abort(); }
   }
   {
      struct account *myAccount = malloc(sizeof(struct account));
      if (myAccount == 0) { abort(); }
   }
}
```

  Notice that to symbolically execute the first `malloc` statement, VeriFast picked symbols myAccount and value, and to symbolically execute the second `malloc` statement, VeriFast picked symbols myAccount0 and value0.

- It adds the assumption myAccount $\neq 0$ (perhaps written differently) to the path condition (as shown in the Assumptions pane). Indeed, if `malloc` succeeds, the returned pointer is not a null pointer.

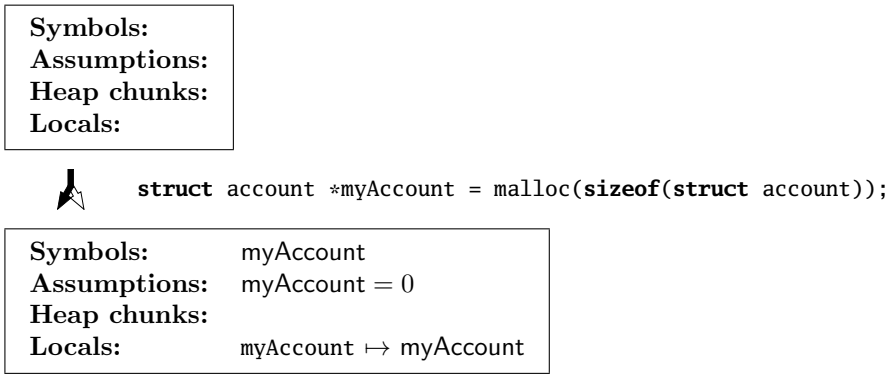

Figure 2: Symbolic execution of a `malloc` statement (first case)

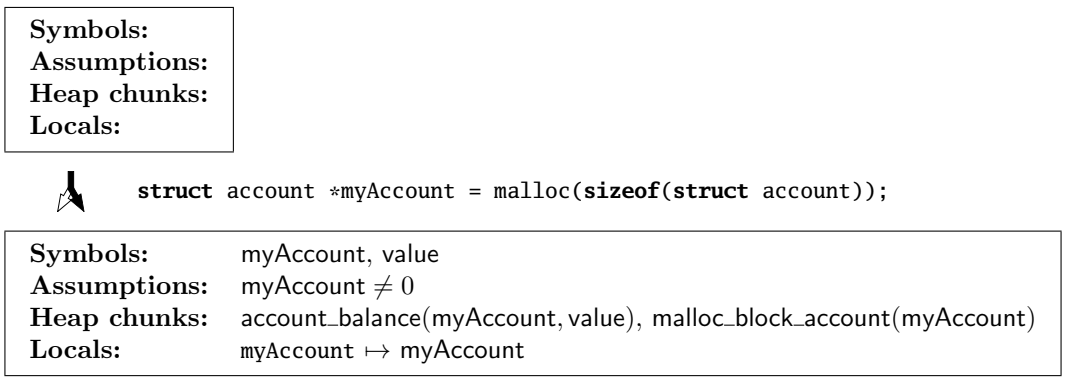

Figure 3: Symbolic execution of a `malloc` statement (second case)

- It adds a binding to the symbolic store (shown in the Locals pane) that binds local variable `myAccount` to symbolic value myAccount. Indeed, the program assigns the result of the `malloc` call (represented by the symbol myAccount) to the local variable `myAccount`. 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 `malloc` 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 a `malloc` 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 `malloc`; it is also what happens with the second branch of the **`if`** statement after an unsuccessful `malloc`.

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

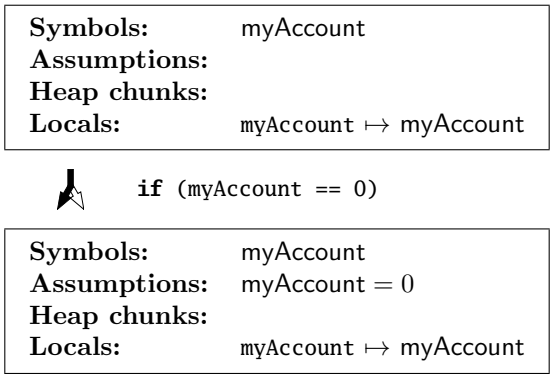

Figure 4: Symbolic execution of an **if** statement (first case). Symbolic execution continues with the *then* block of the **if** statement.

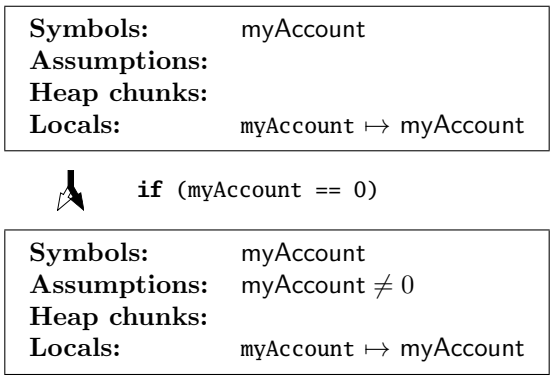

Figure 5: Symbolic execution of an **if** statement (second case). Symbolic execution continues with the *else* block of the **if** statement, if any.

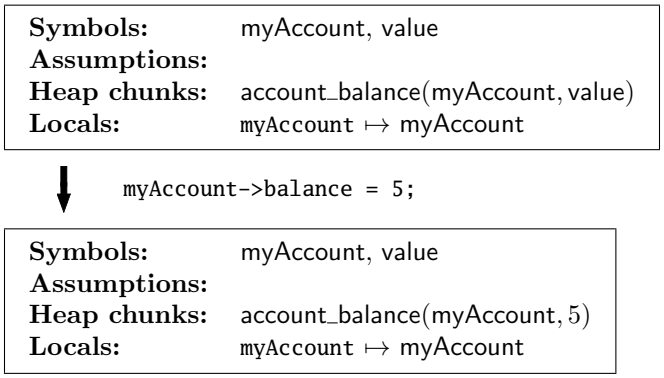

Figure 6: Symbolic execution of a struct field assignment statement

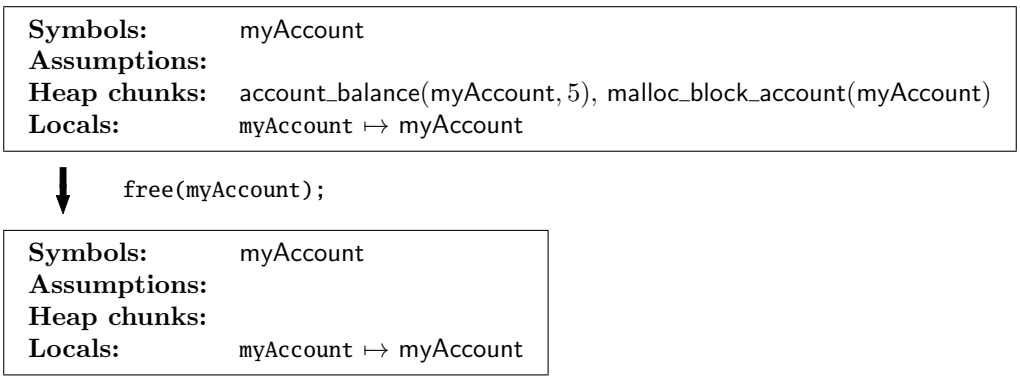

Figure 7: Symbolic execution of a `free` statement

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 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 second argument of the chunk with the value of the right-hand side of the assignment. This is shown in Figure 6.

Finally, the `free` statement removes the two heap chunks from the symbolic heap that were added by the `malloc` statement, as shown in Figure 7.

3 malloc_block Chunks

To better understand why the *malloc* statement generates both an account_balance chunk and a malloc_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:

```
int main()
   //@ requires true;
   //@ ensures true;
{
   struct account myAccountLocal;
   struct account *myAccount = &myAccountLocal;
   myAccount->balance = 5;
```

```
   free(myAccount);
   return 0;
}
```

This program first allocates an instance of struct account on the stack and calls it myAccountLocal. It then assigns the address of this struct instance to pointer variable myAccount. The remainder of the program is as before: the program initializes the balance field to value 5 and then attempts to free the struct instance.

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

No matching heap chunks: malloc_block_account(myAccountLocal_addr)

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

VeriFast detects this error as follows: 1) VeriFast generates a malloc_block chunk only for struct instances allocated using *malloc*, not for struct instances allocated on the stack. 2) When verifying a *free* statement, VeriFast checks that a malloc_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 a function account_set_balance that takes the address of 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:

```
#include "stdlib.h"

struct account {
   int balance;
};

void account_set_balance(struct account *myAccount, int newBalance)
{
   myAccount->balance = newBalance;
}

int main()
   //@ requires true;
   //@ ensures true;
{
   struct account *myAccount = malloc(sizeof(struct account));
   if (myAccount == 0) { abort(); }
   account_set_balance(myAccount, 5);
   free(myAccount);
   return 0;
}
```

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:

```
void account_set_balance(struct account *myAccount, int newBalance)
   //@ requires true;
   //@ ensures true;
```

Notice that contracts, like all VeriFast annotations, are in comments, so that the C 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 at address myAccount. We achieve this simply by mentioning the heap chunk in the precondition:

```
void account_set_balance(struct account *myAccount, int newBalance)
   //@ requires account_balance(myAccount, _);
   //@ ensures true;
```

Notice that we use an underscore in the position where the value of the field belongs. This indicates that we do not care about the old value of the field when the function is called.[3]

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.

```
void account_set_balance(struct account *myAccount, int newBalance)
   //@ requires account_balance(myAccount, _);
   //@ ensures true;
{
   myAccount->balance = newBalance;
   //@ leak account_balance(myAccount, _);
}
```

Function account_set_balance now verifies, and VeriFast attempts to verify function main. It complains that it cannot free the account struct instance because it does not have permission to access the balance field. Indeed, the symbolic heap contains the malloc_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 second step. The malloc statement is about to be executed and the symbolic heap is empty. Select the next step. The malloc statement has added the account_balance chunk and the malloc_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 8). 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

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

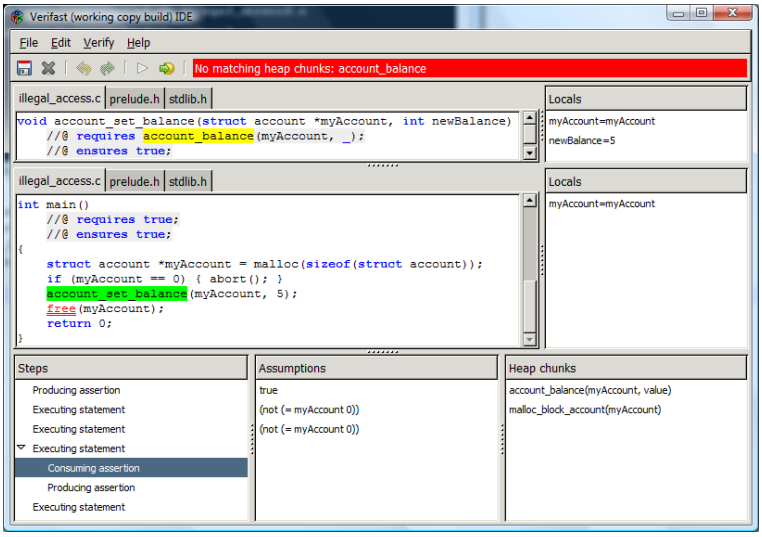

Figure 8: When stepping through a function call, VeriFast shows both the call site (in green, in the lower pane) and the callee's contract (in yellow, in the upper pane).

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.

```
void account_set_balance(struct account *myAccount, int newBalance)
  //@ requires account_balance(myAccount, _);
  //@ ensures account_balance(myAccount, newBalance);
{
  myAccount->balance = newBalance;
}
```

This eliminates the leak error message and the error at the *free* statement. The program now verifies. Notice that we refer to the newBalance 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 1 *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:

```
int account_get_balance(struct account *myAccount)
   //@ requires account_balance(myAccount, _);
   //@ ensures account_balance(myAccount, _);
{
   return myAccount->balance;
}

int main()
   //@ requires true;
   //@ ensures true;
{
   struct account *myAccount = create_account();
   account_set_balance(myAccount, 5);
   int b = account_get_balance(myAccount);
   assert(b == 5);
   account_dispose(myAccount);
   return 0;
}
```

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 `&*&`.

```
int account_get_balance(struct account *myAccount)
   //@ requires account_balance(myAccount, ?theBalance);
   //@ ensures account_balance(myAccount, theBalance) &*& result == theBalance;
{
   return myAccount->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 2 *Add a function that deposits a given amount into an account. Verify the following* `main` *function.*

```
int main()
   //@ requires true;
   //@ ensures true;
{
   struct account *myAccount = create_account();
   account_set_balance(myAccount, 5);
   account_deposit(myAccount, 10);
   int b = account_get_balance(myAccount);
   assert(b == 15);
   account_dispose(myAccount);
   return 0;
}
```

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

Exercise 3 *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 C's conditional expressions* `condition ? value1 : value2`*. Remove function* `account_set_balance`*. Use the shorthand notation for field chunks:* `myAccount->balance |-> value`*. Verify the following* `main` *function.*

```
int main()
   //@ requires true;
   //@ ensures true;
{
   struct account *myAccount = create_account(-100);
   account_deposit(myAccount, 200);
   int w1 = account_withdraw(myAccount, 50);
   assert(w1 == 50);
   int b1 = account_get_balance(myAccount);
   assert(b1 == 150);
   int w2 = account_withdraw(myAccount, 300);
   assert(w2 == 250);
   int b2 = account_get_balance(myAccount);
   assert(b2 == -100);
   account_dispose(myAccount);
   return 0;
}
```

6 Predicates

We continue with the program obtained in Exercise 3. 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.

```
/*@
predicate account_pred(struct account *myAccount, int theLimit, int theBalance) =
   myAccount->limit |-> theLimit &*& myAccount->balance |-> theBalance
   &*& malloc_block_account(myAccount);
@*/
```

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 a malloc_block_account heap chunk into one.

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

```
void account_deposit(struct account *myAccount, int amount)
   //@ requires account_pred(myAccount, ?limit, ?balance) &*& 0 <= amount;
   //@ ensures account_pred(myAccount, limit, balance + amount);
{
   myAccount->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 a 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:

```
void account_deposit(struct account *myAccount, int amount)
   //@ requires account_pred(myAccount, ?limit, ?balance) &*& 0 <= amount;
   //@ ensures account_pred(myAccount, limit, balance + amount);
{
   //@ open account_pred(myAccount, limit, balance);
   myAccount->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:

```
void account_deposit(struct account *myAccount, int amount)
   //@ requires account_pred(myAccount, ?limit, ?balance) &*& 0 <= amount;
   //@ ensures account_pred(myAccount, limit, balance + amount);
{
   //@ open account_pred(myAccount, limit, balance);
   myAccount->balance += amount;
   //@ close account_pred(myAccount, 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 4 *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 5 *Implement a stack of integers using a singly linked list data structure: implement functions* `create_stack`*,* `stack_push`*,* `stack_pop`*, and* `stack_dispose`*. In order to be able to specify the precondition of* `stack_pop`*, your predicate will need to have a parameter that specifies the number of elements in the stack. Function* `stack_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:* `condition ? assertion1 : 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:*

```
int main()
   //@ requires true;
   //@ ensures true;
{
   struct stack *s = create_stack();
   stack_push(s, 10);
   stack_push(s, 20);
   stack_pop(s);
   stack_pop(s);
   stack_dispose(s);
   return 0;
}
```

Now, let's extend the solution to Exercise 5 on page 71 with a `stack_is_empty` function. Recall the predicate definitions:

```
predicate nodes(struct node *node, int count) =
   node == 0 ?
      count == 0
   :
      0 < count
      &*& node->next |-> ?next &*& node->value |-> ?value
      &*& malloc_block_node(node) &*& nodes(next, count - 1);

predicate stack(struct stack *stack, int count) =
   stack->head |-> ?head &*& malloc_block_stack(stack) &*& 0 <= count &*& nodes(head, count);
```

Here's a first stab at a `stack_is_empty` function:

```
bool stack_is_empty(struct stack *stack)
   //@ requires stack(stack, ?count);
   //@ ensures stack(stack, count) &*& result == (count == 0);
{
   //@ open stack(stack, count);
   bool result = stack->head == 0;
   //@ close stack(stack, count);
   return 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.

```
bool stack_is_empty(struct stack *stack)
    //@ requires stack(stack, ?count);
    //@ ensures stack(stack, count) &*& result == (count == 0);
{
    //@ open stack(stack, count);
    struct node *head = stack->head;
    //@ open nodes(head, count);
    bool result = stack->head == 0;
    //@ close nodes(head, count);
    //@ close stack(stack, count);
    return 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 `head == 0` and `count == 0`, and on the second branch we get `head != 0` and `0 < count`.

Exercise 6 *Modify function* `stack_dispose` *so that it works even if the stack still contains some elements. Use a recursive helper function.*[4]

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

Exercise 7 *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 6, 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:

```
void stack_dispose(struct stack *stack)
    //@ requires stack(stack, _);
    //@ ensures true;
{
    //@ open stack(stack, _);
    struct node *n = stack->head;
    while (n != 0)
    {
```

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

```
      //@ open nodes(n, _);
      struct node *next = n->next;
      free(n);
      n = next;
   }
   //@ open nodes(0, _);
   free(stack);
}
```

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 performs a case split on the loop condition:
  - If the condition is true:
    - It verifies the loop body,
    - then it consumes the loop invariant,
    - and then finally it checks for leaks. After this step this execution path is finished.
  - If the condition is false, VeriFast puts the heap chunks that were removed in Step 2 back into the heap, and then verification continues after the loop.

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:

```
void stack_dispose(struct stack *stack)
   //@ requires stack(stack, _);
   //@ ensures true;
{
   //@ open stack(stack, _);
   struct node *n = stack->head;
   while (n != 0)
      //@ invariant nodes(n, _);
   {
      //@ open nodes(n, _);
      struct node *next = n->next;
      free(n);
      n = next;
   }
   //@ open nodes(0, _);
   free(stack);
}
```

You can inspect the first branch of the execution of the loop by placing the cursor at the closing brace of the loop body and choosing the **Run to cursor** command. Find the *Executing statement* step for the `while` statement. Notice that this step is followed by a *Producing assertion* step and a *Consuming assertion* step. Notice that between these two steps, the value of variable `n` changes from `head` to `n`, and all chunks are removed from the symbolic heap. Further notice that in the next step, the assumption `(not (= n 0))` is added to the Assumptions pane.

You can inspect the second branch of the execution of the loop by placing the cursor at the closing brace of the function body and choosing the **Run to cursor** command. Notice that the same things happen as in the first branch, except that the loop body is not executed, and the assumption `(= n 0)` is added to the Assumptions pane.

Exercise 8 *Specify and implement function `stack_popn`, which pops a given number of elements from the stack (and returns `void`). You may call `stack_pop` internally. Use a `while` 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 5). 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:

```
int main()
   //@ requires true;
   //@ ensures true;
{
   struct stack *s = create_stack();
   stack_push(s, 10);
   stack_push(s, 20);
   int result1 = stack_pop(s);
   assert(result1 == 20);
   int result2 = stack_pop(s);
   assert(result2 == 10);
   stack_dispose(s);
   return 0;
}
```

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* `ints`:

```
inductive ints = ints_nil | ints_cons(int, ints);
```

This declaration declares a type `ints` with two *constructors* `ints_nil` and `ints_cons`. `ints_nil` represents the empty sequence. `ints_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

```
ints_cons(1, ints_cons(2, ints_cons(3, ints_nil)))
```

Exercise 9 *Replace the `count` parameter of the `nodes` and `stack` predicates with a `values` parameter of type `ints` and update the predicate bodies. Further update the functions `create_stack`, `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*:

```
fixpoint int ints_head(ints values) {
   switch (values) {
      case ints_nil: return 0;
      case ints_cons(value, values0): return value;
   }
}

fixpoint ints ints_tail(ints values) {
   switch (values) {
      case ints_nil: return ints_nil;
      case ints_cons(value, values0): return values0;
   }
}
```

Notice that we can use switch statements 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 switch statement on one of the function's parameters (called the *inductive parameter*). Furthermore, the body of each case must be a return statement.

In contrast to regular C functions, a fixpoint function may be used in any place where an expression is expected in an annotation. This means we can use the `ints_head` and `ints_tail` functions to adapt function `stack_pop` to the new predicate definitions, and to specify the return value.

Exercise 10 *Do so.*

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

Exercise 11 *Add a C function `stack_get_sum` that returns the sum of the elements of a given stack. Implement it using a recursive helper function `nodes_get_sum`. Specify the new C functions using a recursive fixpoint function `ints_sum`. Remember to turn off arithmetic overflow checking in the **Verify** menu. Verify the following main function:*

```
int main()
   //@ requires true;
   //@ ensures true;
{
   struct stack *s = create_stack();
   stack_push(s, 10);
   stack_push(s, 20);
   int sum = stack_get_sum(s);
   assert(sum == 30);
   int result1 = stack_pop(s);
   assert(result1 == 20);
   int result2 = stack_pop(s);
   assert(result2 == 10);
   stack_dispose(s);
   return 0;
}
```

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 29 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 5). Let's add a `stack_get_count` function that returns the number of elements in the stack, implemented using a loop:

```
int stack_get_count(struct stack *stack)
   //@ requires stack(stack, ?count);
   //@ ensures stack(stack, count) &*& result == count;
{
   //@ open stack(stack, count);
   struct node *head = stack->head;
   struct node *n = head;
   int i = 0;
   while (n != 0)
      //@ invariant true;
   {
      n = n->next;
      i++;
   }
   //@ close stack(stack, count);
   return 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:

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

We can define the `lseg` predicate as follows:

```
predicate lseg(struct node *first, struct node *last, int count) =
   first == last ?
      count == 0
   :
      0 < count &*& first != 0 &*&
      first->value |-> _ &*& first->next |-> ?next &*& malloc_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 C 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 C 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 switch 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 switch 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:

```
lemma void nodes_to_lseg_lemma(struct node *first)
   requires nodes(first, ?count);
   ensures 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 admitted since it performs induction on heap size.

Exercise 12 *We will need the inverse operation as well. Write a lemma function* `lseg_to_nodes_lemma` *that takes an* `lseg` *chunk that ends in 0 and transforms it into a* `nodes` *chunk.*

The `stack_get_count` function now looks as follows:

```
int stack_get_count(struct stack *stack)
   //@ requires stack(stack, ?count);
   //@ ensures stack(stack, count) &*& result == count;
{
   //@ open stack(stack, count);
   struct node *head = stack->head;
   //@ nodes_to_lseg_lemma(head);
   struct node *n = head;
   int i = 0;
   //@ close lseg(head, head, 0);
   while (n != 0)
      //@ invariant lseg(head, n, i) &*& lseg(n, 0, count - i);
   {
      //@ open lseg(n, 0, count - i);
      n = n->next;
      i++;
      // Error!
   }
```

```
   //@ open lseg(0, 0, _);
   //@ lseg_to_nodes_lemma(head);
   //@ close stack(stack, count);
   return 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 |-> _ &*& old_n->next |-> n &*&
malloc_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:

```
lemma void lseg_add_lemma(struct node *first)
   requires
      lseg(first, ?last, ?count) &*& last != 0 &*& last->value |-> _ &*& last->next |-> ?next &*&
      malloc_block_node(last);
   ensures 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:

```
/*@
lemma void lseg_add_lemma(struct node *first)
   requires
      lseg(first, ?last, ?count) &*& last != 0 &*& last->value |-> _ &*& last->next |-> ?next &*&
      malloc_block_node(last) &*& lseg(next, 0, ?count0);
   ensures 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);
}
@*/

int stack_get_count(struct stack *stack)
    //@ requires stack(stack, ?count);
    //@ ensures stack(stack, count) &*& result == count;
{
    //@ open stack(stack, count);
    struct node *head = stack->head;
    //@ nodes_to_lseg_lemma(head);
    struct node *n = head;
    int i = 0;
    //@ close lseg(head, head, 0);
    while (n != 0)
        //@ invariant lseg(head, n, i) &*& lseg(n, 0, count - i);
    {
        //@ open lseg(n, 0, count - i);
        n = n->next;
        i++;
        //@ lseg_add_lemma(head);
    }
    //@ open lseg(0, 0, _);
    //@ lseg_to_nodes_lemma(head);
    //@ close stack(stack, count);
    return i;
}
```

These now verify.

Exercise 13 *Verify the following function. You'll need an extra lemma.*

```
void stack_push_all(struct stack *stack, struct stack *other)
    //@ requires stack(stack, ?count) &*& stack(other, ?count0);
    //@ ensures stack(stack, count0 + count);
{
    struct node *head0 = other->head;
    free(other);
    struct node *n = head0;
    if (n != 0) {
        while (n->next != 0)
        {
            n = n->next;
        }
        n->next = stack->head;
        stack->head = head0;
    }
}
```

Exercise 14 *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:

```
typedef bool int_predicate(int x);

struct node *nodes_filter(struct node *n, int_predicate *p)
   //@ requires nodes(n, _);
   //@ ensures nodes(result, _);
{
   if (n == 0) {
      return 0;
   } else {
      //@ open nodes(n, _);
      bool keep = p(n->value);
      if (keep) {
         struct node *next = nodes_filter(n->next, p);
         //@ open nodes(next, ?count);
         //@ close nodes(next, count);
         n->next = next;
         //@ close nodes(n, count + 1);
         return n;
      } else {
         struct node *next = n->next;
         free(n);
         struct node *result = nodes_filter(next, p);
         return result;
      }
   }
}

void stack_filter(struct stack *stack, int_predicate *p)
   //@ requires stack(stack, _);
   //@ ensures stack(stack, _);
{
   //@ open stack(stack, _);
   struct node *head = nodes_filter(stack->head, p);
   //@ assert nodes(head, ?count);
   stack->head = head;
   //@ open nodes(head, count);
   //@ close nodes(head, count);
   //@ close stack(stack, count);
}

bool neq_20(int x)
   //@ requires true;
   //@ ensures true;
{
   return x != 20;
```

```
}

int main()
   //@ requires true;
   //@ ensures true;
{
   struct stack *s = create_stack();
   stack_push(s, 10);
   stack_push(s, 20);
   stack_push(s, 30);
   stack_filter(s, neq_20);
   stack_dispose(s);
   return 0;
}
```

This program does not verify. VeriFast does not know what contract to use to verify the call of `p` in function `nodes_filter`. VeriFast requires that each function type has a contract:

```
typedef bool int_predicate(int x);
   //@ requires true;
   //@ ensures true;
```

However, this is not sufficient. Even though `p` is of type `int_predicate *`, this does not guarantee that it points to a function that complies with the `int_predicate` function type's signature and contract. Indeed, any integer can be cast to a pointer and any pointer can be cast to a function pointer. Therefore, VeriFast introduces a pure boolean function `is_T` for each function type `T` declared in the program. When verifying a call through a function pointer `p` of type `T *`, VeriFast checks that `is_T(p)` is true. In order to generate an `is_T(f)` fact for a given function `f`, `f`'s header must include a function type implementation clause:

```
bool neq_20(int x) //@ : int_predicate
   //@ requires true;
   //@ ensures true;
```

This further causes VeriFast to check that the contract of `neq_20` subsumes the contract of `int_predicate`.

Finally, we must pass the `is_int_predicate` fact from the client to the call site:

```
struct node *nodes_filter(struct node *n, int_predicate *p)
   //@ requires nodes(n, _) &*& is_int_predicate(p) == true;
   //@ ensures nodes(result, _);

void stack_filter(struct stack *stack, int_predicate *p)
   //@ requires stack(stack, _) &*& is_int_predicate(p) == true;
   //@ ensures stack(stack, _);
```

Notice that we write `is_int_predicate(p) == true` instead of `is_int_predicate(p)`. VeriFast parses the latter form as a predicate assertion, and since there is no such predicate, rejects it.

Exercise 15 *Put all the pieces together.*

The state of our `stack_filter` function is unsatisfactory in two ways: firstly, the `int_predicate` 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 predicate families 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.

```
void nodes_filter(struct node **n, int_predicate *p) {
   if (*n != 0) {
      bool keep = p((*n)->value);
      if (keep) {
         nodes_filter(&(*n)->next, p);
      } else {
         struct node *next = (*n)->next;
         free(*n);
         *n = next;
         nodes_filter(n, p);
      }
   }
}

void stack_filter(struct stack *stack, int_predicate *p) {
   nodes_filter(&stack->head, p);
}
```

In this program, we are effectively passing the pointer to the current node to function `nodes_filter` by reference. Inside function `nodes_filter`, 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 variable chunk; in this case, since the pointer being dereferenced points to a variable that holds a pointer, it looks for a `pointer` chunk. Predicate `pointer` is defined in `prelude.h` as follows:

```
predicate pointer(void **pp; void *p);
```

(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 `nodes_filter`:

```
void nodes_filter(struct node **n, int_predicate *p)
   //@ requires pointer(n, ?node) &*& nodes(node, _) &*& is_int_predicate(p) == true;
   //@ ensures pointer(n, ?node0) &*& nodes(node0, _);
```

In order to be able to call `nodes_filter` from `stack_filter`, we must produce a `pointer` chunk. Specifically, we must transform the `stack_head` chunk into a `pointer` 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:

```
void stack_filter(struct stack *stack, int_predicate *p)
   //@ requires stack(stack, _) &*& is_int_predicate(p) == true;
   //@ ensures stack(stack, _);
{
   //@ open stack(stack, _);
   //@ open stack_head(stack, _);
   nodes_filter(&stack->head, p);
   //@ assert pointer(&stack->head, ?head) &*& nodes(head, ?count);
   //@ close stack_head(stack, head);
   //@ open nodes(head, count);
   //@ close nodes(head, count);
```

```
   //@ close stack(stack, count);
}
```

Notice that, since we cannot use patterns in close statements, we need to bind the value of the head field to a variable `head` before we can close the `stack_head` chunk.

Exercise 16 *Verify function* `nodes_filter`.

Note: VeriFast currently supports the dereference (`*`) operator only for pointers to pointers, pointers to integers, and pointers to characters. In the latter cases, the following predicates are used, also defined in `prelude.h`:

```
predicate integer(int *p; int v);
predicate character(char *p; char v);
```

VeriFast does not yet support taking the address of a local variable.

14 Arithmetic Overflow

Consider the following program:

```
int main()
   //@ requires true;
   //@ ensures true;
{
   int x = 2000000000 + 2000000000;
   assert(0 <= x);
   return 0;
}
```

The behavior of this program is not uniquely defined by the C language. Specifically, the **int** type supports only a finite range of integers, and furthermore the C language does not specify the bounds of this range. The bounds are *implementation-dependent*. The C language does specify that the least and greatest value of type **int** are given by the macros `INT_MIN` and `INT_MAX`, respectively, and that `INT_MIN` is not greater than -32767 and that `INT_MAX` is not less than 32767. Furthermore, the C language does not specify what happens if an arithmetic operation on type **int** yields a value that is not within the limits of type **int**.

VeriFast does not attempt to be sound with respect to arbitrary C implementations (or even with respect to arbitrary standard-compliant ones). Specifically, it assumes that `INT_MIN` equals -2^{31} and `INT_MAX` equals $2^{31} - 1$.

When symbolically evaluating an arithmetic operation in C code (i.e., not in an annotation), VeriFast checks that the result of the operation fits within the bounds of the result type, unless arithmetic overflow checking has been turned off in the **Verify** menu. As a result, VeriFast rejects the above program. It also rejects the following program:

```
void int_add(int *x, int *y)
   //@ requires integer(x, ?vx) &*& integer(y, ?vy);
   //@ ensures integer(x, vx+vy) &*& integer(y, vy);
{
   int x_deref = *x;
   int y_deref = *y;
   *x = x_deref + y_deref;
}
```

Let's try to fix this function so that it works correctly on a 32-bit machine. Here's a first attempt:

```
#include "stdlib.h"
void int_add(int *x_ptr, int *y_ptr)
   //@ requires integer(x_ptr, ?x_value) &*& integer(y_ptr, ?y_value);
   //@ ensures integer(x_ptr, x_value+y_value) &*& integer(y_ptr, y_value);
{
   int x = *x_ptr;
   int y = *y_ptr;
   if (0 <= x) {
      if (INT_MAX - x < y) abort();
   } else {
      if (y < INT_MIN - x) abort();
   }
   *x_ptr = x + y;
}
```

Even though this function works correctly on a 32-bit machine, 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 for verification performance reasons. 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 C local variable (i.e., not a local variable declared in an annotation). The following program verifies.

```
#include "stdlib.h"
void int_add(int *x_ptr, int *y_ptr)
   //@ requires integer(x_ptr, ?x_value) &*& integer(y_ptr, ?y_value);
   //@ ensures integer(x_ptr, x_value+y_value) &*& integer(y_ptr, y_value);
{
   int x = *x_ptr;
   int y = *y_ptr;
   //@ produce_limits(x);
   //@ produce_limits(y);
   if (0 <= x) {
      if (INT_MAX - x < y) abort();
   } else {
      if (y < INT_MIN - x) abort();
   }
   *x_ptr = x + y;
}
```

Note: The above overflow-checked integer addition implementation is probably not the most optimal one in terms of performance. For example, the x86 instruction set includes the INTO (Interrupt on Overflow) instruction, which causes a software interrupt if the overflow flag is set. An implementation that uses this instruction is likely to perform much better.

Note: in case an `integer`, `character` or `pointer` chunk is on the heap, one can also use the lemmas `integer_limits`, `character_limits` and `pointer_limits` respectively. The contracts of these lemmas can be read in the file `prelude.h`.

15 Predicate Families

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:

```
typedef bool int_predicate(void *data, int x);
```

```
struct node *nodes_filter(struct node *n, int_predicate *p, void *data)
{
   if (n == 0) {
      return 0;
   } else {
      bool keep = p(data, n->value);
      if (keep) {
         struct node *next = nodes_filter(n->next, p, data);
         n->next = next;
         return n;
      } else {
         struct node *next = n->next;
         free(n);
         struct node *result = nodes_filter(next, p, data);
         return result;
      }
   }
}

void stack_filter(struct stack *stack, int_predicate *p, void *data)
{
   struct node *head = nodes_filter(stack->head, p, data);
   stack->head = head;
}

struct neq_a_data {
   int a;
};

bool neq_a(struct neq_a_data *data, int x)
{
   bool result = x != data->a;
   return result;
}

int read_int();

int main()
{
   struct stack *s = create_stack();
   stack_push(s, 10);
   stack_push(s, 20);
   stack_push(s, 30);
   int a = read_int();
   struct neq_a_data *data = malloc(sizeof(struct neq_a_data));
   if (data == 0) abort();
   data->a = a;
   stack_filter(s, neq_a, data);
   free(data);
   stack_dispose(s);
   return 0;
}
```

How do we specify the stack module? Here's an attempt:

```
//@ predicate int_predicate_data(void *data) = ??

typedef bool int_predicate(void *data, int x);
   //@ requires int_predicate_data(data);
   //@ ensures int_predicate_data(data);

struct node *nodes_filter(struct node *n, int_predicate *p, void *data)
   //@ requires nodes(n, _) &*& is_int_predicate(p) == true &*& int_predicate_data(data);
   //@ ensures nodes(result, _) &*& int_predicate_data(data);
{ ... }

void stack_filter(struct stack *stack, int_predicate *p, void *data)
   //@ requires stack(stack, _) &*& is_int_predicate(p) == true &*& int_predicate_data(data);
   //@ ensures stack(stack, _) &*& int_predicate_data(data);
{ ... }
```

The problem is in the definition of predicate `int_predicate_data`. There is no way for the stack module to predict what data structure the `data` pointer points to. If we allow the client to choose the definition of the predicate, we can verify the client easily:

```
//@ predicate int_predicate_data(void *data) = neq_a_data_a(data, _);

bool neq_a(struct neq_a_data *data, int x) //@ : int_predicate
   //@ requires int_predicate_data(data);
   //@ ensures int_predicate_data(data);
{
   //@ open int_predicate_data(data);
   bool result = x != data->a;
   //@ close int_predicate_data(data);
   return result;
}

int read_int();
   //@ requires true;
   //@ ensures true;

int main()
   //@ requires true;
   //@ ensures true;
{
   struct stack *s = create_stack();
   stack_push(s, 10);
   stack_push(s, 20);
   stack_push(s, 30);
   int a = read_int();
   struct neq_a_data *data = malloc(sizeof(struct neq_a_data));
   if (data == 0) abort();
   data->a = a;
   //@ close int_predicate_data(data);
   stack_filter(s, neq_a, data);
   //@ open int_predicate_data(data);
   free(data);
   stack_dispose(s);
```

```
   return 0;
}
```

However, this clearly is not feasible. After all, if our stack module has more than one client, we will have multiple definitions for the same predicate. Specifically, we will have one definition of `int_predicate_data` for each function of type `int_predicate`. The problem would be solved if we could refer specifically to the definition of `int_predicate_data` corresponding to a given function of type `int_predicate`. This is exactly what *predicate families* enable. A predicate family is like an ordinary predicate, except that you can have multiple definitions, provided that each definition is associated with a distinct *index*. Predicate family indices must be function pointers.

If we apply predicate families, we get the following specification for the stack module:

```
//@ predicate_family int_predicate_data(void *p)(void *data);

typedef bool int_predicate(void *data, int x);
   //@ requires int_predicate_data(this)(data);
   //@ ensures int_predicate_data(this)(data);

struct node *nodes_filter(struct node *n, int_predicate *p, void *data)
   //@ requires nodes(n, _) &*& is_int_predicate(p) == true &*& int_predicate_data(p)(data);
   //@ ensures nodes(result, _) &*& int_predicate_data(p)(data);

void stack_filter(struct stack *stack, int_predicate *p, void *data)
   //@ requires stack(stack, _) &*& is_int_predicate(p) == true &*& int_predicate_data(p)(data);
   //@ ensures stack(stack, _) &*& int_predicate_data(p)(data);
```

Notice that inside the contract of a function type, you can refer to the function pointer as `this`.

The client can be verified as follows:

```
struct neq_a_data {
   int a;
};

/*@
predicate_family_instance int_predicate_data(neq_a)(void *data) =
   neq_a_data_a(data, _);
@*/

bool neq_a(struct neq_a_data *data, int x) //@ : int_predicate
   //@ requires int_predicate_data(neq_a)(data);
   //@ ensures int_predicate_data(neq_a)(data);
{
   //@ open int_predicate_data(neq_a)(data);
   bool result = x != data->a;
   //@ close int_predicate_data(neq_a)(data);
   return result;
}

int read_int();
   //@ requires true;
   //@ ensures true;

int main()
   //@ requires true;
   //@ ensures true;
```

```
{
   struct stack *s = create_stack();
   stack_push(s, 10);
   stack_push(s, 20);
   stack_push(s, 30);
   int a = read_int();
   struct neq_a_data *data = malloc(sizeof(struct neq_a_data));
   if (data == 0) abort();
   data->a = a;
   //@ close int_predicate_data(neq_a)(data);
   stack_filter(s, neq_a, data);
   //@ open int_predicate_data(neq_a)(data);
   free(data);
   stack_dispose(s);
   return 0;
}
```

Exercise 17 *Add a function* `stack_map` *that takes a function* f *that takes an int and returns an int.* `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 14, where we verified full functional correctness of a `stack_reverse` function for a stack of integers. We used an `ints` inductive datatype, fixpoint functions `append` and `reverse`, and lemmas `append_nil` and `append_assoc`. Suppose, now, that we need the same functionality for a stack of pointers. Clearly, since C does not support generics, we will need to copy-paste the C code and replace `int` with `void *` throughout. However, fortunately, VeriFast does support generics for inductive datatypes, fixpoint functions, lemmas, and predicates, so instead of copy-pasting all of these, we can parameterize them by the element type. Here are parameterized versions of `ints`, `append`, and `append_nil`:

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

fixpoint list<t> append<t>(list<t> xs, list<t> ys) {
   switch (xs) {
      case nil: return ys;
      case cons(x, xs0): return cons<t>(x, append<t>(xs0, ys));
   }
}

lemma void append_nil<t>(list<t> xs)
   requires true;
   ensures append<t>(xs, nil<t>) == xs;
{
   switch (xs) {
      case nil:
      case cons(x, xs0):
         append_nil<t>(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 predicates `nodes` and `stack` and function `stack_reverse` look like for stacks of pointers:

```
predicate nodes(struct node *node, list<void *> values) =
   node == 0 ?
      values == nil<void *>
   :
      node->next |-> ?next &*& node->value |-> ?value &*& malloc_block_node(node) &*&
      nodes(next, ?values0) &*& values == cons<void *>(value, values0);

predicate stack(struct stack *stack, list<void *> values) =
   stack->head |-> ?head &*& malloc_block_stack(stack) &*& nodes(head, values);

void stack_reverse(struct stack *stack)
   //@ requires stack(stack, ?values);
   //@ ensures stack(stack, reverse<void *>(values));
{
   //@ open stack(stack, values);
   struct node *n = stack->head;
   struct node *m = 0;
   //@ close nodes(m, nil<void *>);
   //@ append_nil<void *>(reverse<void *>(values));
   while (n != 0)
      /*@
      invariant
         nodes(m, ?values1) &*& nodes(n, ?values2) &*&
         reverse<void *>(values) == append<void *>(reverse<void *>(values2), values1);
      @*/
   {
      //@ open nodes(n, values2);
      struct node *next = n->next;
      //@ assert nodes(next, ?values2tail) &*& n->value |-> ?value;
      n->next = m;
      m = n;
      n = next;
      //@ close nodes(m, cons<void *>(value, values1));
      //@ append_assoc<void *>(reverse<void *>(values2tail), cons<void *>(value, nil<void *>), values1);
   }
   //@ open nodes(n, _);
   stack->head = m;
   //@ close stack(stack, reverse<void *>(values));
}
```

Thanks to generics, we can now re-use the same datatypes, fixpoints, and lemmas for both stacks of ints and stacks of pointers. However, this approach does seem to introduce a lot of syntactic overhead, considering how we need to insert type argument lists everywhere. Fortunately, VeriFast performs *type argument inference*. 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. Specifically, in the example,

all type argument lists in expressions can be omitted. Note: 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 comes with a header file `list.h` that includes the list datatype, all of the fixpoints and lemmas used in the example, and many more. This header file is implicitly included in each file that is verified by VeriFast, so you do not need to include it explicitly. Note: This also means that you cannot define your own versions of `nil`, `cons`, or other elements provided by `list.h`, since this would result in a name clash.

17 Predicate Values

In the previous section, we obtained a stack of pointers. 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 {
   int x;
   int y;
};

struct vector *create_vector(int x, int y)
{
   struct vector *result = malloc(sizeof(struct vector));
   if (result == 0) abort();
   result->x = x;
   result->y = y;
   return result;
}

int main()
{
   struct stack *s = create_stack();
   while (true)
   {
      char c = input_char();
      if (c == 'p') {
         int x = input_int();
         int y = input_int();
         struct vector *v = create_vector(x, y);
         stack_push(s, v);
      } else if (c == '+') {
         bool empty = stack_is_empty(s);
         if (empty) abort();
         struct vector *v1 = stack_pop(s);
         empty = stack_is_empty(s);
         if (empty) abort();
         struct vector *v2 = stack_pop(s);
         struct vector *sum = create_vector(v1->x + v2->x, v1->y + v2->y);
         free(v1);
         free(v2);
```

```
      stack_push(s, sum);
    } else if (c == '=') {
      bool empty = stack_is_empty(s);
      if (empty) abort();
      struct vector *v = stack_pop(s);
      output_int(v->x);
      output_int(v->y);
      free(v);
    } else {
      abort();
    }
  }
}
```

The specification of `create_vector` is easy:

```
//@ predicate vector(struct vector *v) = v->x |-> _ &*& v->y |-> _ &*& malloc_block_vector(v);

struct vector *create_vector(int x, int y)
  //@ requires true;
  //@ ensures 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:

```
predicate vectors(list<struct vector *> vs) =
  switch (vs) {
    case nil: return true;
    case cons(v, vs0): return 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:

```
predicate foreach(list<void *> vs, predicate(void *) p) =
  switch (vs) {
    case nil: return true;
    case cons(v, vs0): return p(v) &*& foreach(vs0, p);
  };
```

Using generics, we can even generalize this predicate further, to work for lists of arbitrary values:

```
predicate foreach<t>(list<t> vs, predicate(t) p) =
  switch (vs) {
    case nil: return true;
    case cons(v, vs0): return p(v) &*& foreach(vs0, 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.h`; as a result, it is automatically available in each file and you do not need to define it yourself.

Exercise 18 *Verify the program using* `foreach`*.*

18 Predicate Constructors

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

```
struct vector *create_vector(int limit, int x, int y)
{
   if (x * x + y * y > limit * limit) abort();
   struct vector *result = malloc(sizeof(struct vector));
   if (result == 0) abort();
   result->x = x;
   result->y = y;
   return result;
}

int main()
{
   int limit = input_int();
   struct stack *s = create_stack();
   while (true)
   {
      char c = input_char();
      if (c == 'p') {
         int x = input_int();
         int y = input_int();
         struct vector *v = create_vector(limit, x, y);
         stack_push(s, v);
      } else if (c == '+') {
         bool empty = stack_is_empty(s);
         if (empty) abort();
         struct vector *v1 = stack_pop(s);
         empty = stack_is_empty(s);
         if (empty) abort();
         struct vector *v2 = stack_pop(s);
         struct vector *sum = create_vector(limit, v1->x + v2->x, v1->y + v2->y);
         free(v1);
         free(v2);
         stack_push(s, sum);
      } else if (c == '=') {
         bool empty = stack_is_empty(s);
         if (empty) abort();
         struct vector *v = stack_pop(s);
         int x = v->x;
         int y = v->y;
         free(v);
         assert(x * x + y * y <= limit * limit);
         output_int(x);
         output_int(y);
      } else {
         abort();
      }
   }
}
```

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 aborts. When printing a vector, the program asserts that the vector satisfies the size limit.

How do we verify this assert statement?

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:

```
/*@
predicate_ctor vector(int limit)(struct vector *v) =
   v->x |-> ?x &*& v->y |-> ?y &*& malloc_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 19 *Verify the program.*

19 Multithreading

Consider the following program that walks a binary tree of integers, computes the faculty of the value of each node, sums up the results, and prints the sum to the console.

```
int rand();

int fac(int x)
{
   int result = 1;
   while (x > 1)
   {
      result = result * x;
      x = x - 1;
   }
   return result;
}

struct tree {
   struct tree *left;
   struct tree *right;
   int value;
};

struct tree *make_tree(int depth)
{
   if (depth == 0) {
      return 0;
   } else {
```

```
      struct tree *left = make_tree(depth - 1);
      struct tree *right = make_tree(depth - 1);
      int value = rand();
      struct tree *t = malloc(sizeof(struct tree));
      if (t == 0) abort();
      t->left = left;
      t->right = right;
      t->value = value % 2000;
      return t;
   }
}

int tree_compute_sum_facs(struct tree *tree)
{
   if (tree == 0) {
      return 1;
   } else {
      int leftSum = tree_compute_sum_facs(tree->left);
      int rightSum = tree_compute_sum_facs(tree->right);
      int f = fac(tree->value);
      return leftSum + rightSum + f;
   }
}

int main()
{
   struct tree *tree = make_tree(22);
   int sum = tree_compute_sum_facs(tree);
   printf("%i", sum);
   return 0;
}
```

Exercise 20 *Verify the memory safety of this program. You may leak the tree (see Section 4) after computing the sum.*

This program takes 14 seconds on the author's machine. However, the author's machine is a dual core 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.

Unfortunately, the C language does not define a standard mechanism for writing multithreaded programs.[5] Programs for Windows can use the Windows API, and programs for Unix-like operating systems can typically use the POSIX pthreads API. However, the VeriFast distribution includes a wrapper around these APIs that offers a uniform interface across all supported platforms. It is defined in `threading.h` and implemented in `threading.c`; both files are in the VeriFast `bin` directory. VeriFast automatically looks for header files in its `bin` directory, so to use these APIs, simply add the line

```
#include "threading.h"
```

to the top of your file. In this section, we will be using the functions `thread_start_joinable` and `thread_join` from `threading.h`, defined as follows:

```
typedef void thread_run_joinable(void *data);

struct thread;
```

[5]The recent C standard revision C11 introduces support for multithreading, but it is not yet widely supported.

```
struct thread *thread_start_joinable(void *run, void *data);

void thread_join(struct thread *thread);
```

Function `thread_start_joinable` takes a pointer to a `run` function and executes this function in a new thread. Any data required by the run function can be passed via the `data` parameter of `thread_start_joinable`, which is simply passed through to the `run` function. `thread_start_joinable` returns a thread handle, which can be used to join with the created thread using function `thread_join`. The latter waits until the specified thread has finished.

We can use these functions to speed up our example program as follows:

```
struct sum_data {
   struct thread *thread;
   struct tree *tree;
   int sum;
};

void summator(struct sum_data *data)
{
   int sum = tree_compute_sum_facs(data->tree);
   data->sum = sum;
}

struct sum_data *start_sum_thread(struct tree *tree)
{
   struct sum_data *data = malloc(sizeof(struct sum_data));
   struct thread *t = 0;
   if (data == 0) abort();
   data->tree = tree;
   t = thread_start_joinable(summator, data);
   data->thread = t;
   return data;
}

int join_sum_thread(struct sum_data *data)
{
   thread_join(data->thread);
   return data->sum;
}

int main()
{
   struct tree *tree = make_tree(22);
   struct sum_data *leftData = start_sum_thread(tree->left);
   struct sum_data *rightData = start_sum_thread(tree->right);
   int sumLeft = join_sum_thread(leftData);
   int sumRight = join_sum_thread(rightData);
   int f = fac(tree->value);
   printf("%i", sumLeft + sumRight + f);
   return 0;
}
```

The main program creates a tree of depth 22. It then starts two threads. The first thread computes the sum of the faculties of the values of the left subtree, and the second thread computes the sum of the

faculties of the values of the right subtree. The main thread then waits for both threads to finish, and finally sums up the results of both threads with the faculty of the root node's value. On the author's machine, this program takes only 7 seconds; a twofold speedup!

Now, let's verify the memory safety of this program. The verification of this program does not require any new techniques; we will simply apply the techniques we saw in Sections 12 and 15 on function pointers and predicate families. That is, we specify a contract for the `thread_run_joinable` function pointer type and we use predicate families so that each program may concretize this contract according to its needs. The specification of `thread_start_joinable` and `thread_join` in `threading.h` is as follows:

```
/*@

predicate_family thread_run_pre(void *thread_run)(void *data, any info);
predicate_family thread_run_post(void *thread_run)(void *data, any info);

@*/

typedef void thread_run_joinable(void *data);
  //@ requires thread_run_pre(this)(data, ?info);
  //@ ensures thread_run_post(this)(data, info);

struct thread;

/*@ predicate thread(struct thread *thread, void *thread_run, void *data, any info); @*/

struct thread *thread_start_joinable(void *run, void *data);
  //@ requires is_thread_run_joinable(run) == true &*& thread_run_pre(run)(data, ?info);
  //@ ensures thread(result, run, data, info);

void thread_join(struct thread *thread);
  //@ requires thread(thread, ?run, ?data, ?info);
  //@ ensures thread_run_post(run)(data, info);
```

Function `thread_start_joinable` requires that the value of parameter `run` is a function pointer that satisfies the contract of function pointer type `thread_run_joinable`; furthermore, it requires the resources that are required by the `run` function itself, denoted by the `thread_run_pre(run)` predicate family instance. This predicate takes the `data` pointer as an argument, so that the precondition may describe the data structure pointed to by this pointer. The `info` parameter of `thread_run_pre` and `thread_run_post` is used in advanced scenarios and may be ignored for now. Function `thread_start_joinable` returns a `thread` predicate that contains all information required to verify the `thread_join` call. Function `thread_join` takes this `thread` predicate and returns the resources provided by the `run` function when it returns, as described by the `thread_run_post(run)` predicate family instance.

Exercise 21 *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 faculties, we want to compute both the sum of the faculties and the product of the faculties. Since our machine has two cores, we want to have two threads: one that computes the sum of the faculties of the values of the nodes of the tree, and one that computes the product of the faculties of the values of the nodes of the tree.

To achieve this, first we generalize our `tree_compute_sum_facs` function to a `tree_fold` function that takes the function that it should apply to combine the node values as an argument:

```
typedef int fold_function(int acc, int x);

int tree_fold(struct tree *tree, fold_function *f, int acc)
{
   if (tree == 0) {
      return acc;
   } else {
      acc = tree_fold(tree->left, f, acc);
      acc = tree_fold(tree->right, f, acc);
      acc = f(acc, tree->value);
      return acc;
   }
}
```

If the elements of the tree t in postfix order are e_1, e_2, e_3, e_4, then `tree_fold`(t, f, a) returns

$$f(f(f(f(a, e_1), e_2), e_3), e_4)$$

Likewise, we generalize our summator thread to a folder thread:

```
struct fold_data {
   struct thread *thread;
   struct tree *tree;
   fold_function *f;
   int acc;
};

void folder(struct fold_data *data)
{
   int acc = tree_fold(data->tree, data->f, data->acc);
   data->acc = acc;
}

struct fold_data *start_fold_thread(struct tree *tree, fold_function *f, int acc)
{
   struct fold_data *data = malloc(sizeof(struct fold_data));
   struct thread *t = 0;
   if (data == 0) abort();
   data->tree = tree;
   data->f = f;
   data->acc = acc;
   t = thread_start_joinable(folder, data);
   data->thread = t;
   return data;
}

int join_fold_thread(struct fold_data *data)
{
   thread_join(data->thread);
   return data->acc;
}
```

We can re-implement the program of the previous section using folder threads as follows:

```
int sum_function(int acc, int x)
{
```

```
   int f = fac(x);
   return acc + f;
}

int main()
   //@ requires true;
   //@ ensures true;
{
   struct tree *tree = make_tree(22);
   //@ open tree(tree, _);
   struct fold_data *leftData = start_fold_thread(tree->left, sum_function, 0);
   struct fold_data *rightData = start_fold_thread(tree->right, sum_function, 0);
   int sumLeft = join_fold_thread(leftData);
   int sumRight = join_fold_thread(rightData);
   int f = fac(tree->value);
   //@ leak tree->left |-> _ &*& tree->right |-> _ &*& tree->value |-> _ &*& malloc_block_tree(tree);
   printf("%i", sumLeft + sumRight + f);
   return 0;
}
```

Exercise 22 *Verify the memory safety of this program.*

It is now easy to write the program that computes the sums in one thread and the products in another, and returns the difference between the product and the sum:

```
int sum_function(int acc, int x)
{
   int f = fac(x);
   return acc + f;
}

int product_function(int acc, int x)
{
   int f = fac(x);
   return acc * f;
}

int main()
{
   struct tree *tree = make_tree(21);
   struct fold_data *sumData = start_fold_thread(tree, sum_function, 0);
   struct fold_data *productData = start_fold_thread(tree, product_function, 1);
   int sum = join_fold_thread(sumData);
   int product = join_fold_thread(productData);
   printf("%i", product - sum);
   return 0;
}
```

However, this program cannot be verified using the techniques that we have seen. The first `start_fold_thread` 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_fold` function does not modify the tree, it requires only a fraction of the `tree` chunk:

```
int tree_fold(struct tree *tree, fold_function *f, int acc)
   //@ requires [?frac]tree(tree, ?depth) &*& is_fold_function(f) == true;
   //@ ensures [frac]tree(tree, depth);
{
   if (tree == 0) {
      return acc;
   } else {
      //@ open [frac]tree(tree, depth);
      acc = tree_fold(tree->left, f, acc);
      acc = tree_fold(tree->right, f, acc);
      acc = f(acc, tree->value);
      return acc;
      //@ close [frac]tree(tree, depth);
   }
}
```

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 function* `start_fold_thread` *so that it requires 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 C program so that it properly deallocates all dynamically allocated memory. We need to introduce a function `dispose_tree` and update functions `join_fold_thread` and `main`:

```
void dispose_tree(struct tree *tree)
{
   if (tree != 0) {
      dispose_tree(tree->left);
      dispose_tree(tree->right);
      free(tree);
   }
}

int join_fold_thread(struct fold_data *data)
{
```

```
   thread_join(data->thread);
   int result = data->acc;
   free(data);
   return result;
}

int main()
{
   struct tree *tree = make_tree(21);
   struct fold_data *sumData = start_fold_thread(tree, sum_function, 0);
   struct fold_data *productData = start_fold_thread(tree, product_function, 1);
   int sum = join_fold_thread(sumData);
   int product = join_fold_thread(productData);
   dispose_tree(tree);
   printf("%i", product - sum);
   return 0;
}
```

Updating the annotations is slightly trickier. First, let's do the easy part: verifying the `free` statement in function `join_fold_thread`. It already has almost all required permissions; only the `malloc_block_fold_data` chunk is missing. This chunk is leaked in function `start_fold_thread`. Removing that **`leak`** statement and adding the chunk to the **`ensures`** clause of `start_fold_thread` and the **`requires`** clause of `join_fold_thread` solves that problem; the `free` statement now verifies.

All that is left now is to verify the `dispose_tree` call in function `main`. It needs full permission for the tree, i.e. it needs a `tree(tree, _)` chunk. Notice that each `join_fold_thread` call leaks a `[1/2]tree(tree, _)` chunk. We need to update the annotations of function `join_fold_thread` so that it returns this chunk to its caller instead of leaking it. However, we currently have no way to identify the tree in the contract of `join_fold_thread`, and a `[1/2]tree(_, _)` chunk would not help us, since we would not know in `main` that the chunk pertains to the specific tree that we are disposing.

To solve this problem, notice that the tree is pointed to by field `data->tree` in the pre-state of function `join_fold_thread`. However, we cannot mention this field in function `join_fold_thread`'s precondition since we have passed full permission for this field to the folder thread.

The solution is to pass only a fraction of the permission for field `data->tree` to the fold thread, and to retain the remaining fraction in the main thread. To do so, update the `thread_run_pre` and `thread_run_post` predicate family instances, as well as the contracts of functions `start_fold_thread` and `join_fold_thread`:

```
predicate_family_instance thread_run_pre(folder)(struct fold_data *data, any info) =
   [1/2]data->tree |-> ?tree &*& [1/2]tree(tree, _) &*&
   data->f |-> ?f &*& is_fold_function(f) == true &*& data->acc |-> _;
predicate_family_instance thread_run_post(folder)(struct fold_data *data, any info) =
   [1/2]data->tree |-> ?tree &*& [1/2]tree(tree, _) &*&
   data->f |-> ?f &*& is_fold_function(f) == true &*& data->acc |-> _;

struct fold_data *start_fold_thread(struct tree *tree, fold_function *f, int acc)
   //@ requires [1/2]tree(tree, _) &*& is_fold_function(f) == true;
   /*@
   ensures
      [1/2]result->tree |-> tree &*& result->thread |-> ?t &*&
      thread(t, folder, result, _) &*& malloc_block_fold_data(result);
   @*/
int join_fold_thread(struct fold_data *data)
   /*@
   requires
```

```
      [1/2]data->tree |-> ?tree &*& data->thread |-> ?t &*&
      thread(t, folder, data, _) &*& malloc_block_fold_data(data);
   @*/
   //@ ensures [1/2]tree(tree, _);
```

We can now specify the `[1/2]tree(tree, _)` chunk in the postcondition of function `join_fold_thread`.

Notice that we now have two `[1/2]tree(tree, _)` chunks in function `main` at the `dispose_tree` call. 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:

```
/*@

predicate foo(bool b) = true;

predicate bar(int *x, int *y) = foo(?b) &*& b ? integer(x, _) : integer(y, _);

lemma void merge_bar() // This lemma is false!!
   requires [?f1]bar(?x, ?y) &*& [?f2]bar(x, y);
   ensures [f1 + f2]bar(x, y);
{
   assume(false);
}

@*/

int main()
   //@ requires true;
   //@ ensures true;
{
   int x, y;
   //@ 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) integer_unique(&x); else integer_unique(&y);
   assert(false);
}
```

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. a program that raises an assertion failure at run time verifies successfully). It uses the following lemma from `prelude.h`:

```
lemma void integer_unique(int *p);
   requires [?f]integer(p, ?v);
   ensures [f]integer(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]integer(&x, _)` and the other one represents `[1/2]integer(&y, _)`. Combining them yields neither `integer(&x, _)` nor `integer(&y, _)`.

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 `integer` itself is a precise predicate; it is declared in `prelude.h` as follows:

```
predicate integer(int *p; int v);
```

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

```
predicate tree(struct tree *t; int depth) =
   t == 0 ?
      depth == 0
   :
      t->left |-> ?left &*& t->right |-> ?right &*& t->value |-> _ &*& malloc_block_tree(t) &*&
      tree(left, ?childDepth) &*& tree(right, childDepth) &*& depth == childDepth + 1;
```

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 **`switch`** 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. 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. 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.

The C language does not itself offer any synchronization constructs; rather, the operating system does. For example, the Windows API offers a mutex construct called critical sections, and the Linux API offers a mutex construct called mutexes. To enable a uniform interface across all platforms, VeriFast includes a module `threading.c` that offers two synchronization constructs: mutexes and locks. The only difference between the two is that mutexes are non-re-entrant and locks are re-entrant. Since mutexes are easier to use, we will use those in this example.

Here's the source code of the example program:

```
#include "stdlib.h"
#include "threading.h"

void wait_for_pulse(int source);
```

```
void sleep(int millis);
void print_int(int n);

struct counter {
   int count;
   struct mutex *mutex;
};

struct count_pulses_data {
   struct counter *counter;
   int source;
};

void count_pulses(struct count_pulses_data *data) {
   struct counter *counter = data->counter;
   int source = data->source;
   free(data);

   struct mutex *mutex = counter->mutex;

   while (true) {
      wait_for_pulse(source);
      mutex_acquire(mutex);
      counter->count++;
      mutex_release(mutex);
   }
}

void count_pulses_async(struct counter *counter, int source) {
   struct count_pulses_data *data = malloc(sizeof(struct count_pulses_data));
   if (data == 0) abort();
   data->counter = counter;
   data->source = source;
   thread_start(count_pulses, data);
}

int main() {
   struct counter *counter = malloc(sizeof(struct counter));
   if (counter == 0) abort();
   counter->count = 0;
   struct mutex *mutex = create_mutex();
   counter->mutex = mutex;

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

   while (true) {
      sleep(1000);
      mutex_acquire(mutex);
      print_int(counter->count);
      mutex_release(mutex);
   }
}
```

For this program we use the threading API function `thread_start` instead of `thread_start_joinable`, since we will not be joining the thread. The specification of function `thread_start` is analogous to the specification of function `thread_start_joinable` discussed in Section 19; it is given in header file `threading.h` as follows:

```
//@ predicate_family thread_run_data(void *thread_run)(void *data);

typedef void thread_run(void *data);
  //@ requires thread_run_data(this)(data);
  //@ ensures true;

void thread_start(void *run, void *data);
  //@ requires is_thread_run(run) == true &*& thread_run_data(run)(data);
  //@ ensures true;
```

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.

The mutex functions offered by `threading.c` are specified in `threading.h` as follows:

```
struct mutex;

/*@
predicate mutex(struct mutex *mutex; predicate() p);
predicate mutex_held(struct mutex *mutex, predicate() p, int threadId, real frac);
predicate create_mutex_ghost_arg(predicate() p) = true;
@*/

struct mutex *create_mutex();
  //@ requires create_mutex_ghost_arg(?p) &*& p();
  //@ ensures mutex(result, p);

void mutex_acquire(struct mutex *mutex);
  //@ requires [?f]mutex(mutex, ?p);
  //@ ensures mutex_held(mutex, p, currentThread, f) &*& p();

void mutex_release(struct mutex *mutex);
  //@ requires mutex_held(mutex, ?p, currentThread, ?f) &*& p();
  //@ ensures [f]mutex(mutex, p);

void mutex_dispose(struct mutex *mutex);
  //@ requires mutex(mutex, ?p);
  //@ ensures p();
```

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 `create_mutex_ghost_arg` predicate, which exists only for this purpose. That is, before calling `create_mutex`, you need to close a `create_mutex_ghost_arg` 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 `mutex_held` 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 `mutex_acquire` call additionally produces the lock invariant, giving the thread access to the memory locations that were owned by the mutex.

The `mutex_release` call consumes a `mutex_held` 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:

```
#include "stdlib.h"
#include "threading.h"

int wait_for_source();
bool wait_for_pulse(int source); // true means the sensor has been removed.
void sleep(int millis);
void print_int(int n);

struct counter {
   int count;
   struct mutex *mutex;
};

struct count_pulses_data {
   struct counter *counter;
   int source;
};

void count_pulses(struct count_pulses_data *data) {
   struct counter *counter = data->counter;
   int source = data->source;
   free(data);

   struct mutex *mutex = counter->mutex;
```

```
    bool done = false;
    while (!done) {
        done = wait_for_pulse(source);
        if (!done) {
            mutex_acquire(mutex);
            counter->count++;
            mutex_release(mutex);
        }
    }
}

void count_pulses_async(struct counter *counter, int source) {
    struct count_pulses_data *data = malloc(sizeof(struct count_pulses_data));
    if (data == 0) abort();
    data->counter = counter;
    data->source = source;
    thread_start(count_pulses, data);
}

void print_count(struct counter *counter) {
    struct mutex *mutex = counter->mutex;
    while (true) {
        sleep(1000);
        mutex_acquire(mutex);
        print_int(counter->count);
        mutex_release(mutex);
    }
}

int main() {
    struct counter *counter = malloc(sizeof(struct counter));
    if (counter == 0) abort();
    counter->count = 0;
    struct mutex *mutex = create_mutex();
    counter->mutex = mutex;

    thread_start(print_count, counter);

    while (true) {
        int source = wait_for_source();
        count_pulses_async(counter, 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 Character Arrays

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

```
char getchar();
void putchar(char c);

int main()
{
   char c1 = getchar();
   char c2 = getchar();
   char c3 = getchar();
   char c4 = getchar();
   char c5 = getchar();
   for (int i = 0; i < 2; i++) {
      putchar(c1);
      putchar(c2);
      putchar(c3);
      putchar(c4);
      putchar(c5);
   }
   return 0;
}
```

This program works.[6] If we run the program and enter `Hello`, we get back `HelloHello`.

However, clearly this approach is not practical for large numbers of characters. One approach that works for large numbers of characters is to use an array of characters. We allocate the array using the standard C function `malloc` and then read and write the characters 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 characters.

```
char getchar();
void putchar(char c);
```

[6] We simplified the return type of `getchar` a little bit; the real function returns an int. But this version should compile and run just fine.

```
char *malloc(int count);

void getchars(char *start, int count) {
   if (count > 0) {
      char c = getchar();
      *start = c;
      getchars(start + 1, count - 1);
   }
}

void putchars(char *start, int count) {
   if (count > 0) {
      char c = *start;
      putchar(c);
      putchars(start + 1, count - 1);
   }
}

int main() {
   char *array = malloc(5);
   getchars(array, 5);
   putchars(array, 5);
   putchars(array, 5);
   return 0;
}
```

This program works.[7] 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 each function the simplest possible contract:

```
//@ requires true;
//@ ensures true;
```

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

```
*start = c;
```

This statement writes character `c` into the memory location at address `start`. Whenever VeriFast sees such a statement, it checks that the function has permission to write to this location. Specifically, it checks that a chunk that matches `character(start, _)` is present in the symbolic heap. In this case, there is no such chunk in the symbolic heap, since the requires clause of `getchars` 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 `getchars` that whoever calls the function must give it permission to write to the location at address `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:

```
//@ requires count > 0 ? character(start, _) : true;
//@ ensures count > 0 ? character(start, _) : 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 `getchars`. Indeed, if `count` is greater than one, then the recursive call needs permission to access the location at address `start + 1`. Let's extend our contract to reflect this:

[7] We ignore `malloc` failures for now.

```
//@ requires count > 0 ? character(start, _) &*& (count > 1 ? character(start + 1, _) : true) : true;
//@ ensures count > 0 ? character(start, _) &*& (count > 1 ? character(start + 1, _) : true) : true;
```

Alas, if we run VeriFast again, we notice VeriFast complains again at the recursive call.[8] Indeed, if `count` is greater than 2, the recursive call now needs to permission to access the location at address `start + 2`.

We can patch up the contract again, but will it ever end? If we step back and think about what `getchars` does, we realize that a call `getchars(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:

```
/*@
requires
   count <= 0 ? true :
      character(start, _) &*&
      (count - 1 <= 0 ? true :
         character(start + 1, _) &*&
         (count - 2 <= 0 ? true :
            character(start + 2, _) &*&
            (count - 3 <= 0 ? true :
               character(start + 3, _) &*&
               (count - 4 <= 0 ? true :
                  character(start + 4, _) &*&
                  (count - 5 <= 0 ? true :
                     false)))));
@*/
```

If we use this same assertion as the postcondition as well, function `getchars` verifies. And if we use the same contract for `putchars`, that function verifies as well. And we can use the same assertion as the postcondition of `malloc`, 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
   character(array, _) &*& character(array + 1, _) &*& character(array + 2, _) &*&
   character(array + 3, _) &*& character(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 `getchars` 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:

```
predicate characters(char *start, int count) =
   count <= 0 ? true : character(start, _) &*& characters(start + 1, count - 1);
```

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

So, let's solve Exercise 26 by using `characters` in the contracts:

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

```
char getchar(); /*@ requires true; @*/ /*@ ensures true; @*/
void putchar(char c); /*@ requires true; @*/ /*@ ensures true; @*/

/*@
predicate characters(char *start, int count) =
   count <= 0 ? true : character(start, _) &*& characters(start + 1, count - 1);
@*/

char *malloc(int count);
   //@ requires true;
   //@ ensures characters(result, count);

void getchars(char *start, int count)
   //@ requires characters(start, count);
   //@ ensures characters(start, count);
{
   if (count > 0) {
      char c = getchar();
      *start = c;
      getchars(start + 1, count - 1);
   }
}

void putchars(char *start, int count)
   //@ requires characters(start, count);
   //@ ensures characters(start, count);
{
   if (count > 0) {
      char c = *start;
      putchar(c);
      putchars(start + 1, count - 1);
   }
}

int main() /*@ requires true; @*/ /*@ ensures true; @*/
{
   char *array = malloc(100);
   getchars(array, 100);
   putchars(array, 100);
   putchars(array, 100);
   //@ leak characters(array, 100);
   return 0;
}
```

This doesn't quite work yet. The problem is that VeriFast does not automatically replace a `characters` 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 `getchars` because it cannot find a chunk matching `character(start, _)`. This chunk is in fact present; it's just that it is hidden inside the `characters` chunk. We need to open up the `characters` chunk so that VeriFast can see the `character` chunk that is inside of it. The following version of function `getchars` verifies:

```
void getchars(char *start, int count)
   //@ requires characters(start, count);
```

```
   //@ ensures characters(start, count);
{
   if (count > 0) {
      //@ open characters(start, count);
      char c = getchar();
      *start = c;
      getchars(start + 1, count - 1);
      //@ close characters(start, count);
   }
}
```

Inserting the same commands in function `putchars` 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 characters from standard input. It should then replace each character in the first array with the XOR of the character and the corresponding character 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 characters* `c1` *and* `c2` *can be written in C as* `(char)(c1 ^ c2)`*.*

26 Looping over an Array

The example program of the previous section, that reads a sequence of 100 characters 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 characters. This is because functions `getchars` and `putchars` would in that scenario perform ten million nested recursive calls. This would most likely exhaust the call stack.[9] Therefore, it is safer to rewrite these functions so that they use a loop instead of recursion. Let's rewrite function `getchars`:

```
void getchars(char *start, int count)
   //@ requires characters(start, count);
   //@ ensures characters(start, count);
{
   for (int i = 0; i < count; i++) {
      char c = getchar();
      *(start + i) = c;
   }
}
```

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 symbolic heap contains the `characters` chunk, and the value of variable `i` is nonnegative. We encode this as follows:

```
void getchars(char *start, int count)
   //@ requires characters(start, count);
   //@ ensures characters(start, count);
{
   for (int i = 0; i < count; i++)
      //@ invariant characters(start, count) &*& 0 <= i;
   {
```

[9] Unless the C compiler performs tail call optimization. But the C standard does not require compilers to do so.

```
      char c = getchar();
      *(start + i) = c;
   }
}
```

VeriFast now complains that it cannot find the permission to write the character at address `start + i` in the symbolic heap. This permission is in fact present, but it is hidden inside the `characters(start, count)` chunk. In the easy case, it is sufficient to simply open a chunk in order to reveal the permissions that are hidden inside of it. However, in this case, the permission is hidden below multiple layers of the `characters` predicate. In fact, it is hidden below exactly `i + 1` layers. It would require `i + 1` open operations to reveal the permission. We cannot write these operations directly in the program text, since we do not know how many operations to write.

The solution is to first rewrite the `characters(start, count)` chunk into an equivalent set of chunks in such a way that the permission that VeriFast is looking for comes to the surface. Notice that the `characters(start, count)` chunk describes the same set of memory permissions as the following pair of chunks:

```
characters(start, i) &*& characters(start + i, count - i)
```

If we can rewrite the chunk into this form, we can then simply open the `characters(start + i, count - i)` chunk to obtain our `character(start + i, _)` permission.

We can perform this rewrite by first performing `i` open operations, to lay bare the first `i` characters, and then performing `i + 1` close operations, to combine these `i` characters into a `characters(start, i)` chunk. To perform these operations, 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 C function, except that it starts with the **`lemma`** keyword and it is written inside an annotation:

```
/*@
lemma void split_characters_chunk(char *start, int i)
   requires characters(start, ?count) &*& 0 <= i &*& i <= count;
   ensures characters(start, i) &*& characters(start + i, count - i);
{
   if (i == 0) {
      close characters(start, 0);
   } else {
      open characters(start, count);
      split_characters_chunk(start + 1, i - 1);
      close characters(start, i);
   }
}
@*/
```

Like an ordinary function, a lemma function has a contract and a body. The contract of the above lemma function, `split_characters_chunk`, states that the function requires a single characters chunk, as well as a value `i` that lies between zero and `count`, and gives back two chunks: one that contains the first `i` characters, and one that contains the remaining `count - i` characters.

The function implementation first checks if `i` equals zero. If it does, the incoming chunk corresponds exactly to the second chunk that needs to be returned. The function only needs to generate the first chunk, but since `i` equals zero, this is an empty chunk and can be created simply by closing it.

In case `i` is not equal to zero, the function first opens the incoming chunk. This lays bare the first character (the one at `start`) as well as the `characters` chunk that goes from `start + 1` to the end. The function then splits the latter chunk into a part that contains the first `i - 1` characters and a part that contains the remaining `count - i` characters. This latter chunk is the second chunk that needs to be returned. Finally, it bundles the `character` chunk at `start` up with the `characters(start + 1, i - 1)` chunk that was returned by the recursive call, to obtain the first chunk that needs to be returned.

To verify the assignment to `*(start + i)` in function `getchars`, we now simply need to call the lemma function and then open chunk `characters(start + i, count - i)`:

```
void getchars(char *start, int count)
   //@ requires characters(start, count);
   //@ ensures characters(start, count);
{
   for (int i = 0; i < count; i++)
      //@ invariant characters(start, count) &*& 0 <= i;
   {
      char c = getchar();
      //@ split_characters_chunk(start, i);
      //@ open characters(start + i, count - i);
      *(start + i) = c;
   }
}
```

VeriFast now accepts the assignment. It now complains when checking the loop invariant at the end of the loop body. It complains that it cannot find chunk `characters(start, count)`. Notice that this is again a matter of rewriting the symbolic heap: all memory permissions described by `characters(start, count)` are in fact in the symbolic heap, just not in the packaging in which VeriFast expects them. To satisfy VeriFast, we need to first close the `characters(start + i, count - i)` chunk again and then call a lemma function that merges the `characters(start, i)` and `characters(start + i, count - i)` chunks back into a single `characters(start, count)` chunk.

Exercise 28 *Write the lemma function `merge_characters_chunks`. Then insert a call to this function into `getchars` so that `getchars` verifies. Then also rewrite `putchars` to use a loop instead of recursion and verify the resulting program.*

27 Recursive Loop Proofs

An important observation that we can make after the previous section, is that verifying the recursive version of function `getchars` 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 recently 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:

```
while (condition) {
   ... body ...
}
```

This loop is equivalent to the call

```
iter();
```

where function `iter` is defined as

```
void iter() {
   if (condition) {
```

```
      ... body ...
      iter();
   }
}
```

As a concrete example, consider function `getchars` from the previous section. If we rewrite the **`for`** loop as a **`while`** loop, we get the following code:

```
void getchars(char *start, int count) {
   int i = 0;
   while (i < count) {
      char c = getchar();
      *(start + i) = c;
      i++;
   }
}
```

We can now apply the above translation scheme to replace the loop with an equivalent local recursive function:

```
void getchars(char *start, int count) {
   int i = 0;
   void iter() {
      if (i < count) {
         char c = getchar();
         *(start + i) = c;
         i++;
         iter();
      }
   }
   iter();
}
```

Note that this translation scheme uses local functions. Standard C does not have local functions, but the GNU C compiler, gcc, does. gcc compiles and runs the above function correctly.

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 `getchars` that uses a local recursive function `iter`:

```
void getchars(char *start, int count)
   //@ requires characters(start, count);
   //@ ensures characters(start, count);
{
   int i = 0;
   void iter()
      //@ requires characters(start + i, count - i);
      //@ ensures characters(start + old_i, count - old_i);
   {
      if (i < count) {
         //@ open characters(start + i, count - i);
         char c = getchar();
         *(start + i) = c;
```

```
        i++;
        iter();
        //@ close characters(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 `getchars` that uses a **`while`** loop, is to transplant the annotations that we inserted for the recursive function, back into the loop version:

```
void getchars(char *start, int count)
   //@ requires characters(start, count);
   //@ ensures characters(start, count);
{
   int i = 0;
   while (i < count)
      //@ requires characters(start + i, count - i);
      //@ ensures characters(start + old_i, count - old_i);
   {
      //@ open characters(start + i, count - i);
      char c = getchar();
      *(start + i) = c;
      i++;
      //@ recursive_call();
      //@ close characters(start + old_i, count - old_i);
   }
}
```

Notice that we inserted a `recursive_call();` ghost statement to indicate where the imaginary recursive call occurs. VeriFast verifies this function successfully.

Exercise 29 *Write a version of function* `putchars` *that uses a* **`while`** *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! Note, however, that you will need to rewrite the* **`while`** *loop into a* **`for (;;)`** *loop (or, equivalently, a* **`while (true)`** *loop) because you need to execute a ghost command before exiting the loop.*

28 Tracking Array Contents

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

```
void memcpy(char *dest, char *src, int count) {
   for (int i = 0; i < count; i++) {
      dest[i] = src[i];
   }
}
```

To specify functional correctness, the contract for `memcpy` must express that when the function returns, the contents of the `dest` array are equal to the contents of the `src` array. This means that we cannot

use the `characters` predicate from the preceding sections, since that predicate specifies only the size of a character array, and not its contents. We must add a predicate parameter that specifies the list of characters held by the array:

```
predicate chars(char *array, int count; list<char> cs) =
   count == 0 ?
      cs == nil
   :
      character(array, ?c) &*& chars(array + 1, count - 1, ?cs0) &*& cs == cons(c, cs0);
```

We can use this predicate to fully specify the behavior of the `memcpy` function as follows:

```
void memcpy(char *dest, char *src, int count)
   //@ requires chars(dest, count, _) &*& [?f]chars(src, count, ?cs);
   //@ ensures chars(dest, count, cs) &*& [f]chars(src, count, cs);
```

This predicate is in fact the "official" character array predicate in VeriFast: it is declared in `prelude.h`, along with a number of useful lemmas, and it is used by VeriFast for certain purposes. For example, the following function verifies given the above contract for `memcpy`:

```
void test()
   //@ requires true;
   //@ ensures true;
{
   char buffer1[7] = "Hello!";
   char buffer2[7];
   memcpy(buffer2, buffer1, 7);
   assert(buffer2[5] == '!');
}
```

Indeed, declaring a local character array causes VeriFast to produce a `chars` chunk.

Notice the semicolon in the definition of predicate `chars`: this means it is declared as precise with two input parameters (`array` and `count`) and one output parameter (`cs`). As explained in Sections 21 and 22, it follows that VeriFast will merge `chars` fractions and automatically **`open`** and **`close`** `chars` chunks in certain circumstances.

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

```
void memcpy(char *dest, char *src, int count)
   //@ requires dest[0..count] |-> _ &*& [?f]src[0..count] |-> ?cs;
   //@ ensures dest[0..count] |-> cs &*& [f]src[0..count] |-> cs;
```

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).

```
void memcpy(char *dest, char *src, int count)
   //@ requires dest[0..count] |-> _ &*& [?f]src[0..count] |-> ?cs;
   //@ ensures dest[0..count] |-> cs &*& [f]src[0..count] |-> cs;
{
   for (int i = 0; ; i++)
      //@ requires dest[i..count] |-> _ &*& [f]src[i..count] |-> ?cs0;
      //@ ensures dest[old_i..count] |-> cs0 &*& [f]src[old_i..count] |-> cs0;
   {
      //@ open chars(dest + i, _, _);
      //@ open chars(src + i, _, _);
      if (i == count) {
         break;
```

```
      }
      dest[i] = src[i];
   }
}
```

Notice that, as usual when using loop contracts, we had to move the loop condition into the loop body so that we could insert ghost commands before the loop exit (i.e., the **break** statement).

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. Note: if a boolean expression in an assertion starts with a parenthesis, you must prefix the boolean expression with* **true** `==`*, e.g.* **true** `== ((a == b) == (c == d));` *this is a limitation of the VeriFast parser.*

```
int memcmp(char *p1, char *p2, int count) {
   int result = 0;
   for (int i = 0; ; i++) {
      if (i == count) {
         break;
      }
      if (p1[i] < p2[i]) {
         result = -1; break;
      }
      if (p1[i] > p2[i]) {
         result = 1; break;
      }
   }
   return result;
}
```

29 Strings

In C programs, strings are conventionally stored in memory as *zero-terminated* character sequences. For example, the string `"Hello"` is stored as `'H'`,`'e'`,`'l'`, `'l'`, `'o'`, `0`. This means a function that computes the length of a string could be written as follows:

```
int strlen(char *s) {
   int i = 0;
   for (; s[i] != 0; i++);
   return i;
}
```

Here is a simple main program that uses this function:

```
int main() {
   int n = strlen("Hello, world!");
   assert(n == 13);
   return 0;
}
```

How do we verify this function and this main program?

The precondition of function `strlen` must specify that the caller must supply sufficient permissions to access all locations starting at address `s`, up to and including the first location after `s` whose value is zero. We could do this using the `chars` predicate introduced in the previous section, but it is more elegant to define a new predicate for this specific purpose:

```
predicate string(char *s; list<char> cs) =
   character(s, ?c) &*&
   c == 0 ?
      cs == nil
   :
      string(s + 1, ?cs0) &*& cs == cons(c, cs0);
```

Using this predicate, we can specify function `strlen` as follows:

```
int strlen(char *s)
   //@ requires string(s, ?cs);
   //@ ensures string(s, cs) &*& result == length(cs);
```

Given this specification, the function is easy to verify.

In fact, the `string` predicate is the "official" predicate for zero-terminated strings in VeriFast: it is declared in `prelude.h` along with a number of useful lemmas, and VeriFast produces a `string` chunk when a string literal expression is evaluated.

Let's now attempt to verify the main program. Unfortunately, this fails. The reason is that the above precondition of function `strlen` requires full permission for the string, i.e. permission to read and modify the string's characters. However, a C string literal is *immutable*; the program must not modify it. VeriFast reflects this by generating only a *fractional permission* for the string literal (see Section 20).

However, fortunately, function `strlen` does not actually modify the string that it operates on, so we can weaken its contract so that it requires only a fractional permission:

```
int strlen(char *s)
   //@ requires [?f]string(s, ?cs);
   //@ ensures [f]string(s, cs) &*& result == length(cs);
```

Exercise 32 *Verify function* `strlen` *and the main program. Use a loop contract (see Section 27) to verify the loop. Note that you'll need to move the loop condition into the body of the loop.*

30 Arrays of Pointers

The previous section introduced VeriFast's support for arrays of characters. Similar support exists for arrays of unsigned characters, integers, unsigned integers, and pointers. In this section, we will use VeriFast's support for arrays of pointers to verify memory safety of a very simple application for keeping track of the names of students in a class. The following program reads a list of student names, and then allows the user to look up the name of the kth student in constant time. When the user enters an invalid student number, the program deallocates all allocated memory and terminates.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int read_int() {
   int x;
   int scanf_result = scanf("%i", &x);
   if (scanf_result < 1) abort();
   return x;
}

char *read_line() {
   char buffer[100];
   int scanf_result = scanf("␣%99[^\n]", buffer);
   if (scanf_result < 1) abort();
```

```
   char *result = strdup(buffer);
   if (result == 0) abort();
   return result;
}

int main() {
   printf("How many students do you have? ");
   int n = read_int();
   if (n < 0 || 0x20000000 <= n) abort();
   char **names = malloc(n * sizeof(char **));
   if (names == 0) abort();
   for (int i = 0; i != n; i++) {
      printf("Please enter the name of student number %d: ", i + 1);
      char *name = read_line();
      printf("Adding '%s'...\n", name);
      names[i] = name;
   }

   for (;;) {
      printf("Please enter a student number: ");
      int k = read_int();
      if (k < 1 || n < k) {
         printf("Student number out of range. Terminating...\n");
         break;
      } else {
         char *name = names[k - 1];
         printf("Student number %d is called %s.\n", k, name);
      }
   }

   for (int i = 0; i != n; i++) {
      free(names[i]);
   }
   free(names);

   return 0;
}
```

Helper function `read_int` uses the standard C library function `scanf` (declared in header file `stdio.h`) to read an integer value (in decimal or hexadecimal notation) from standard input and store it in variable `x`. Similarly, helper function `read_line` uses `scanf` to read a sequence of up to 99 characters that are not newline characters, i.e., a single line of text of at most 99 characters, and store it in character array `buffer` (allocated on the stack). The return value of `scanf` indicates the number of items successfully read. Then, `read_line` uses library function `strdup` (from the POSIX standard, declared in header file `string.h`) to copy the zero-terminated string in `buffer` to a newly heap-allocated memory block. `strdup` returns 0 if the operation fails due to insufficient memory.

Verification of function `read_int` is trivial. Verification of function `read_line` involves one difficulty: after the `scanf` call, array `buffer` is described by a `chars` chunk, but function `strdup` requires a `string` chunk. Fortunately, we can extract a `string` chunk out of any character array that contains a zero character using lemma `chars_separate_string` declared in `prelude.h`. Since the array must again be available as a `chars` chunk when function `read_line` returns, we must call the companion lemma `chars_unseparate_string` after the `strdup` call to merge the `string` chunk back into the `chars` chunk.

In the main function, verification requires that we annotate each of the three loops. Let's focus on the first loop. Since it traverses the `names` array front to back, we will use a loop contract. The loop requires access to the array; how do we specify this? How does VeriFast represent the array allocated by the `malloc` statement? In fact, if a `malloc` call is of the form `T **x = malloc(n * sizeof(T *))`, VeriFast produces a `pointers(x, n, _)` chunk. Predicate `pointers` is defined in `prelude.h` and is analogous to predicate `chars` (see Section 28):

```
predicate pointers(void **pp, int count; list<void *> ps) =
   count == 0 ?
      ps == nil
   :
      pointer(pp, ?p) &*& pointers(pp + 1, count - 1, ?ps0) &*& ps == cons(p, ps0);
```

Like for `chars` chunks, VeriFast supports array slice syntax for `pointers` chunks: the notation `a[i..n] |-> ?ps` where `a` is of type `T **` is equivalent to `pointers(a + i, n - i, ?ps)`. A suitable precondition for the first loop in function `main` is therefore as follows:

```
//@ requires names[i..n] |-> _;
```

This loop stores in each element of array `names` a pointer to a heap-allocated memory block containing a zero-terminated string. The postcondition of the loop should therefore specify not just the array itself but these memory blocks as well. As discussed in Section 17, we can specify the presence of a chunk for each element of a list using predicate `foreach`. A reasonable postcondition for the loop would therefore be:

```
//@ ensures names[old_i..n] |-> ?ps &*& foreach(ps, student);
```

where predicate `student` is defined as

```
predicate student(char *name) = string(name, ?cs) &*& malloc_block_chars(name, length(cs) + 1);
```

(Note: allocating an array `a` of `n` characters produces a `malloc_block_chars(a, n)` chunk; similarly, allocating an array `a` of `n` pointers produces a `malloc_block_pointers(a, n)` chunk.) The above postcondition would work, but we can reduce the number of **open** and **close** statements required to work with these chunks by using precise variants (see Section 22). We can declare predicate `student` as precise as follows:

```
predicate student(char *name;) = string(name, ?cs) &*& malloc_block_chars(name, length(cs) + 1);
```

VeriFast comes with a *ghost header file* `listex.gh` (to be included using a ghost include directive `//@ #include <listex.gh>`) that declares a precise variant of predicate `foreach`, called `foreachp`:

```
predicate foreachp<t>(list<t> xs, predicate(t;) p;) =
   xs == nil ?
      emp
   :
      xs == cons(head(xs), tail(xs)) &*& p(head(xs)) &*& foreachp(tail(xs), p);
```

Using these precise predicates, we get the following postcondition for the first loop:

```
//@ ensures names[old_i..n] |-> ?ps &*& foreachp(ps, student);
```

The body of the first loop now verifies without any annotations... almost. VeriFast does not realize that if `i == n`, then `ps == nil`. This causes verification of the path that exits the loop to fail. We need to force a case split by moving the loop condition into the loop body and inserting an **open** statement at the top of the body:

```
for (int i = 0; ; i++)
   //@ requires names[i..n] |-> _;
   //@ ensures names[old_i..n] |-> ?ps &*& foreachp(ps, student);
{
   //@ open pointers(_, _, _);
```

```
    if (i == n) {
        break;
    }
    printf("Please␣enter␣the␣name␣of␣student␣number␣%d:␣", i + 1);
    char *name = read_line();
    printf("Adding␣'%s'...\n", name);
    names[i] = name;
}
```

The first loop now verifies.

The second loop differs from the first one in that it does not traverse the loop linearly; rather, it performs random access. Therefore, we use an ordinary loop invariant:

```
//@ invariant names[0..n] |-> ?ps &*& foreachp(ps, student);
```

In the first loop, the array element access `names[i]` caused an auto-open of the `pointers(names + i, n - i, _)` chunk, laying bare the `pointer(names + i, _)` chunk required by the element access (see Section 13). In the second loop, the `names[k - 1]` access cannot be verified in this way, since the required `pointer` chunk is in the middle of the `pointers` chunk, rather than in the front; it would require `k` **open** operations to lay it bare, and the auto-open feature cannot handle this. However, the access verifies anyway, because VeriFast recognizes this access as a random access and treats it specially. In particular, if, when evaluating an array access of the form `a[i]`, VeriFast finds a chunk `pointers(a, n, ps)` such that `0 <= i < n`, it considers the access to be valid and returns `nth(i, ps)` as the result of the access. The fixpoint function `nth` is declared in header file `list.h`; `nth(i, ps)` returns the `i`'th element of the list `ps`.

This deals with the array access itself, but there is another problem: the `printf` call requires the `string` chunk for the string pointed to by element `k - 1`. This chunk is inside the `foreachp` chunk. We can extract it using lemma `foreachp_remove_nth` declared in ghost header `listex.gh`:

```
lemma void foreachp_remove_nth<t>(int n);
    requires foreachp<t>(?xs, ?p) &*& 0 <= n &*& n < length(xs);
    ensures foreachp<t>(remove_nth(n, xs), p) &*& p(nth(n, xs));
```

It uses the fixpoint function `remove_nth` declared in `list.h`.

Once the `student` chunk for element `k - 1` is available, it is auto-opened and the `printf` call verifies. After the `printf` call, we need to merge the `student` chunk back into the `foreachp` chunk using the companion lemma `foreachp_unremove_nth`:

```
lemma void foreachp_unremove_nth<t>(list<t> xs, int n);
    requires foreachp<t>(remove_nth(n, xs), ?p) &*& 0 <= n &*& n < length(xs) &*& p(nth(n, xs));
    ensures foreachp<t>(xs, p);
```

These two lemma calls are the only annotations required to verify the body of the second loop.

The third loop again traverses the loop front to back; a loop spec is indicated. Since the loop deallocates the memory blocks holding the student names, the `foreachp` chunk disappears from the postcondition:

```
//@ requires names[i..n] |-> ?ps &*& foreachp(ps, student);
//@ ensures names[old_i..n] |-> _;
```

Verifying the body of the loop requires a few annotations. Firstly, on the path that exits the loop, we get a leak error, again because VeriFast does not realize that `i == n` means that `ps == nil`. Like in the first loop, we need to move the loop condition into the loop body and insert an explicit **open** of the `pointers` chunk.

The second problem is that the `free` call is not satisfied: a `free(a)` call where `a` is of type **char** `*` looks for a `malloc_block_chars(a, n)` chunk and a `chars(a, n, _)` chunk. The `malloc_block_chars` chunk is available inside the `foreachp` chunk, but the `chars` chunk is not; the memory block is instead described by a `string` chunk. Therefore, we need to transform the `string` chunk into a `chars` chunk. A lemma `string_to_chars` is available for this purpose in header `prelude.h`. We insert a call of this lemma before the `free` call.

The third problem is that the `string_to_chars` call is not satisfied. The `string` chunk that it looks for can be obtained by opening the `foreachp` chunk and then opening the resulting `student` chunk, but the auto-open feature does not see this because the `student` chunk is not *statically* inside the `foreachp` chunk; rather, it is inside the `foreachp` chunk only if its second argument is the name of the `student` predicate. The auto-open feature may support this scenario in the future, but for now we need to explicitly open the `foreachp` chunk. The final proof of the third loop looks as follows:

```
for (int i = 0; ; i++)
   //@ requires names[i..n] |-> ?ps &*& foreachp(ps, student);
   //@ ensures names[old_i..n] |-> _;
{
   //@ open pointers(_, _, _);
   if (i == n) {
      break;
   }
   //@ open foreachp(_, _);
   //@ string_to_chars(names[i]);
   free(names[i]);
}
```

The program now verifies.

Exercise 33 *Verify the program (available as* `students.c` *in directory* `tutorial` *of the VeriFast distribution).*

31 Solutions to Exercises

31.1 Exercise 1: account.c

```
#include "stdlib.h"

struct account {
   int balance;
};

struct account *create_account()
   //@ requires true;
   //@ ensures account_balance(result, 0) &*& malloc_block_account(result);
{
   struct account *myAccount = malloc(sizeof(struct account));
   if (myAccount == 0) { abort(); }
   myAccount->balance = 0;
   return myAccount;
}

void account_set_balance(struct account *myAccount, int newBalance)
   //@ requires account_balance(myAccount, _);
   //@ ensures account_balance(myAccount, newBalance);
{
   myAccount->balance = newBalance;
}

void account_dispose(struct account *myAccount)
```

```
   //@ requires account_balance(myAccount, _) &*& malloc_block_account(myAccount);
   //@ ensures true;
{
   free(myAccount);
}

int main()
   //@ requires true;
   //@ ensures true;
{
   struct account *myAccount = create_account();
   account_set_balance(myAccount, 5);
   account_dispose(myAccount);
   return 0;
}
```

31.2 Exercise 2: deposit.c

```
void account_deposit(struct account *myAccount, int amount)
   //@ requires account_balance(myAccount, ?theBalance) &*& 0 <= amount;
   //@ ensures account_balance(myAccount, theBalance + amount);
{
   myAccount->balance += amount;
}
```

31.3 Exercise 3: limit.c

```
#include "stdlib.h"

struct account {
   int limit;
   int balance;
};

struct account *create_account(int limit)
   //@ requires limit <= 0;
   //@ ensures result->limit |-> limit &*& result->balance |-> 0 &*& malloc_block_account(result);
{
   struct account *myAccount = malloc(sizeof(struct account));
   if (myAccount == 0) { abort(); }
   myAccount->limit = limit;
   myAccount->balance = 0;
   return myAccount;
}

int account_get_balance(struct account *myAccount)
   //@ requires myAccount->balance |-> ?theBalance;
   //@ ensures myAccount->balance |-> theBalance &*& result == theBalance;
{
   return myAccount->balance;
}

void account_deposit(struct account *myAccount, int amount)
   //@ requires myAccount->balance |-> ?theBalance;
```

```
   //@ ensures myAccount->balance |-> theBalance + amount;
{
   myAccount->balance += amount;
}

int account_withdraw(struct account *myAccount, int amount)
   //@ requires myAccount->limit |-> ?limit &*& myAccount->balance |-> ?balance &*& 0 <= amount;
   /*@ ensures myAccount->limit |-> limit &*& myAccount->balance |-> balance - result &*&
         result == (balance - amount < limit ? balance - limit : amount); @*/
{
   int result = myAccount->balance - amount < myAccount->limit ?
      myAccount->balance - myAccount->limit : amount;
   myAccount->balance -= result;
   return result;
}

void account_dispose(struct account *myAccount)
   //@ requires myAccount->limit |-> _ &*& myAccount->balance |-> _ &*& malloc_block_account(myAccount);
   //@ ensures true;
{
   free(myAccount);
}
```

31.4 Exercise 4: pred.c

```
#include "stdlib.h"

struct account {
   int limit;
   int balance;
};

/*@
predicate account_pred(struct account *myAccount, int theLimit, int theBalance) =
   myAccount->limit |-> theLimit &*& myAccount->balance |-> theBalance
   &*& malloc_block_account(myAccount);
@*/

struct account *create_account(int limit)
   //@ requires limit <= 0;
   //@ ensures account_pred(result, limit, 0);
{
   struct account *myAccount = malloc(sizeof(struct account));
   if (myAccount == 0) { abort(); }
   myAccount->limit = limit;
   myAccount->balance = 0;
   //@ close account_pred(myAccount, limit, 0);
   return myAccount;
}

int account_get_balance(struct account *myAccount)
   //@ requires account_pred(myAccount, ?limit, ?balance);
   //@ ensures account_pred(myAccount, limit, balance) &*& result == balance;
```

```
{
  //@ open account_pred(myAccount, limit, balance);
  int result = myAccount->balance;
  //@ close account_pred(myAccount, limit, balance);
  return result;
}

void account_deposit(struct account *myAccount, int amount)
  //@ requires account_pred(myAccount, ?limit, ?balance) &*& 0 <= amount;
  //@ ensures account_pred(myAccount, limit, balance + amount);
{
  //@ open account_pred(myAccount, limit, balance);
  myAccount->balance += amount;
  //@ close account_pred(myAccount, limit, balance + amount);
}

int account_withdraw(struct account *myAccount, int amount)
  //@ requires account_pred(myAccount, ?limit, ?balance) &*& 0 <= amount;
  /*@ ensures account_pred(myAccount, limit, balance - result)
        &*& result == (balance - amount < limit ? balance - limit : amount); @*/
{
  //@ open account_pred(myAccount, limit, balance);
  int result = myAccount->balance - amount < myAccount->limit ?
    myAccount->balance - myAccount->limit : amount;
  myAccount->balance -= result;
  //@ close account_pred(myAccount, limit, balance - result);
  return result;
}

void account_dispose(struct account *myAccount)
  //@ requires account_pred(myAccount, _, _);
  //@ ensures true;
{
  //@ open account_pred(myAccount, _, _);
  free(myAccount);
}

int main()
  //@ requires true;
  //@ ensures true;
{
  struct account *myAccount = create_account(-100);
  account_deposit(myAccount, 200);
  int w1 = account_withdraw(myAccount, 50);
  assert(w1 == 50);
  int b1 = account_get_balance(myAccount);
  assert(b1 == 150);
  int w2 = account_withdraw(myAccount, 300);
  assert(w2 == 250);
  int b2 = account_get_balance(myAccount);
  assert(b2 == -100);
  account_dispose(myAccount);
  return 0;
```

```
}
```

31.5 Exercise 5: stack.c

```
#include "stdlib.h"

struct node {
   struct node *next;
   int value;
};

struct stack {
   struct node *head;
};

/*@

predicate nodes(struct node *node, int count) =
   node == 0 ?
      count == 0
   :
      0 < count
      &*& node->next |-> ?next &*& node->value |-> ?value
      &*& malloc_block_node(node) &*& nodes(next, count - 1);

predicate stack(struct stack *stack, int count) =
   stack->head |-> ?head &*& malloc_block_stack(stack) &*& 0 <= count &*& nodes(head, count);

@*/

struct stack *create_stack()
   //@ requires true;
   //@ ensures stack(result, 0);
{
   struct stack *stack = malloc(sizeof(struct stack));
   if (stack == 0) { abort(); }
   stack->head = 0;
   //@ close nodes(0, 0);
   //@ close stack(stack, 0);
   return stack;
}

void stack_push(struct stack *stack, int value)
   //@ requires stack(stack, ?count);
   //@ ensures stack(stack, count + 1);
{
   //@ open stack(stack, count);
   struct node *n = malloc(sizeof(struct node));
   if (n == 0) { abort(); }
   n->next = stack->head;
   n->value = value;
   stack->head = n;
   //@ close nodes(n, count + 1);
```

```
   //@ close stack(stack, count + 1);
}

int stack_pop(struct stack *stack)
   //@ requires stack(stack, ?count) &*& 0 < count;
   //@ ensures stack(stack, count - 1);
{
   //@ open stack(stack, count);
   struct node *head = stack->head;
   //@ open nodes(head, count);
   int result = head->value;
   stack->head = head->next;
   free(head);
   //@ close stack(stack, count - 1);
   return result;
}

void stack_dispose(struct stack *stack)
   //@ requires stack(stack, 0);
   //@ ensures true;
{
   //@ open stack(stack, 0);
   //@ open nodes(_, _);
   free(stack);
}

int main()
   //@ requires true;
   //@ ensures true;
{
   struct stack *s = create_stack();
   stack_push(s, 10);
   stack_push(s, 20);
   stack_pop(s);
   stack_pop(s);
   stack_dispose(s);
   return 0;
}
```

31.6 Exercise 6: dispose.c

```
void nodes_dispose(struct node *n)
   //@ requires nodes(n, _);
   //@ ensures true;
{
   //@ open nodes(n, _);
   if (n != 0) {
      nodes_dispose(n->next);
      free(n);
   }
}

void stack_dispose(struct stack *stack)
```

```
   //@ requires stack(stack, _);
   //@ ensures true;
{
   //@ open stack(stack, _);
   nodes_dispose(stack->head);
   free(stack);
}
```

31.7 Exercise 7: sum.c

```
int nodes_get_sum(struct node *nodes)
   //@ requires nodes(nodes, ?count);
   //@ ensures nodes(nodes, count);
{
   int result = 0;
   //@ open nodes(nodes, count);
   if (nodes != 0) {
      result = nodes_get_sum(nodes->next);
      result += nodes->value;
   }
   //@ close nodes(nodes, count);
   return result;
}

int stack_get_sum(struct stack *stack)
   //@ requires stack(stack, ?count);
   //@ ensures stack(stack, count);
{
   //@ open stack(stack, count);
   int result = nodes_get_sum(stack->head);
   //@ close stack(stack, count);
   return result;
}
```

31.8 Exercise 8: popn.c

```
void stack_popn(struct stack *stack, int n)
   //@ requires stack(stack, ?count) &*& 0 <= n &*& n <= count;
   //@ ensures stack(stack, count - n);
{
   int i = 0;
   while (i < n)
      //@ invariant stack(stack, count - i) &*& i <= n;
   {
      stack_pop(stack);
      i++;
   }
}
```

31.9 Exercise 9: values.c

```
#include "stdlib.h"

struct node {
```

```
   struct node *next;
   int value;
};

struct stack {
   struct node *head;
};

/*@

inductive ints = ints_nil | ints_cons(int, ints);

predicate nodes(struct node *node, ints values) =
   node == 0 ?
      values == ints_nil
   :
      node->next |-> ?next &*& node->value |-> ?value &*& malloc_block_node(node) &*&
      nodes(next, ?values0) &*& values == ints_cons(value, values0);

predicate stack(struct stack *stack, ints values) =
   stack->head |-> ?head &*& malloc_block_stack(stack) &*& nodes(head, values);

@*/

struct stack *create_stack()
   //@ requires true;
   //@ ensures stack(result, ints_nil);
{
   struct stack *stack = malloc(sizeof(struct stack));
   if (stack == 0) { abort(); }
   stack->head = 0;
   //@ close nodes(0, ints_nil);
   //@ close stack(stack, ints_nil);
   return stack;
}

void stack_push(struct stack *stack, int value)
   //@ requires stack(stack, ?values);
   //@ ensures stack(stack, ints_cons(value, values));
{
   //@ open stack(stack, values);
   struct node *n = malloc(sizeof(struct node));
   if (n == 0) { abort(); }
   n->next = stack->head;
   n->value = value;
   stack->head = n;
   //@ close nodes(n, ints_cons(value, values));
   //@ close stack(stack, ints_cons(value, values));
}

void stack_dispose(struct stack *stack)
   //@ requires stack(stack, ints_nil);
   //@ ensures true;
```

```
{
   //@ open stack(stack, ints_nil);
   //@ open nodes(_, _);
   free(stack);
}
```

31.10 Exercise 10: fixpoints.c

```
int stack_pop(struct stack *stack)
   //@ requires stack(stack, ?values) &*& values != ints_nil;
   //@ ensures stack(stack, ints_tail(values)) &*& result == ints_head(values);
{
   //@ open stack(stack, values);
   struct node *head = stack->head;
   //@ open nodes(head, values);
   int result = head->value;
   stack->head = head->next;
   free(head);
   //@ close stack(stack, ints_tail(values));
   return result;
}
```

31.11 Exercise 11: sum_full.c

```
/*@

fixpoint int ints_sum(ints values) {
   switch (values) {
      case ints_nil: return 0;
      case ints_cons(value, values0): return value + ints_sum(values0);
   }
}

@*/

int nodes_get_sum(struct node *node)
   //@ requires nodes(node, ?values);
   //@ ensures nodes(node, values) &*& result == ints_sum(values);
{
   //@ open nodes(node, values);
   int result = 0;
   if (node != 0) {
      int tailSum = nodes_get_sum(node->next);
      result = node->value + tailSum;
   }
   //@ close nodes(node, values);
   return result;
}

int stack_get_sum(struct stack *stack)
   //@ requires stack(stack, ?values);
   //@ ensures stack(stack, values) &*& result == ints_sum(values);
{
   //@ open stack(stack, values);
```

```
   int result = nodes_get_sum(stack->head);
   //@ close stack(stack, values);
   return result;
}
```

31.12 Exercise 12: lemma.c

```
lemma void lseg_to_nodes_lemma(struct node *first)
   requires lseg(first, 0, ?count);
   ensures nodes(first, count);
{
   open lseg(first, 0, count);
   if (first != 0) {
      lseg_to_nodes_lemma(first->next);
   }
   close nodes(first, count);
}
```

31.13 Exercise 13: push_all.c

```
/*@

lemma void lseg_append_lemma(struct node *first)
   requires lseg(first, ?n, ?count) &*& lseg(n, 0, ?count0);
   ensures 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);
   }
}

@*/

void stack_push_all(struct stack *stack, struct stack *other)
   //@ requires stack(stack, ?count) &*& stack(other, ?count0);
   //@ ensures stack(stack, count0 + count);
{
   //@ open stack(stack, count);
   //@ nodes_to_lseg_lemma(stack->head);
   //@ open stack(other, count0);
   //@ nodes_to_lseg_lemma(other->head);
   struct node *head0 = other->head;
   free(other);
   struct node *n = head0;
   //@ open lseg(head0, 0, count0);
   if (n != 0) {
      //@ close lseg(head0, head0, 0);
      while (n->next != 0)
         /*@
         invariant
```

```
            lseg(head0, n, ?count1) &*&
            n != 0 &*& n->value |-> _ &*& n->next |-> ?next &*& malloc_block_node(n) &*&
            lseg(next, 0, count0 - count1 - 1);
         @*/
      {
         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);
}
```

31.14 Exercise 14: reverse.c

```
fixpoint ints ints_append(ints values1, ints values2) {
   switch (values1) {
      case ints_nil: return values2;
      case ints_cons(h, t): return ints_cons(h, ints_append(t, values2));
   }
}

lemma void ints_append_nil_lemma(ints values)
   requires true;
   ensures ints_append(values, ints_nil) == values;
{
   switch (values) {
      case ints_nil:
      case ints_cons(h, t):
         ints_append_nil_lemma(t);
   }
}

lemma void ints_append_assoc_lemma(ints values1, ints values2, ints values3)
   requires true;
   ensures
      ints_append(ints_append(values1, values2), values3) ==
      ints_append(values1, ints_append(values2, values3));
{
   switch (values1) {
      case ints_nil:
      case ints_cons(h, t):
         ints_append_assoc_lemma(t, values2, values3);
   }
}

fixpoint ints ints_reverse(ints values) {
```

```
   switch (values) {
      case ints_nil: return ints_nil;
      case ints_cons(h, t): return ints_append(ints_reverse(t), ints_cons(h, ints_nil));
   }
}

@*/

void stack_reverse(struct stack *stack)
   //@ requires stack(stack, ?values);
   //@ ensures stack(stack, ints_reverse(values));
{
   //@ open stack(stack, values);
   struct node *n = stack->head;
   struct node *m = 0;
   //@ close nodes(m, ints_nil);
   //@ ints_append_nil_lemma(ints_reverse(values));
   while (n != 0)
      /*@
      invariant
         nodes(m, ?values1) &*& nodes(n, ?values2) &*&
         ints_reverse(values) == ints_append(ints_reverse(values2), values1);
      @*/
   {
      //@ open nodes(n, values2);
      struct node *next = n->next;
      //@ assert nodes(next, ?values2tail) &*& n->value |-> ?value;
      n->next = m;
      m = n;
      n = next;
      //@ close nodes(m, ints_cons(value, values1));
      //@ ints_append_assoc_lemma(ints_reverse(values2tail), ints_cons(value, ints_nil), values1);
   }
   //@ open nodes(n, _);
   stack->head = m;
   //@ close stack(stack, ints_reverse(values));
}
```

31.15 Exercise 15: filter.c

```
#include "stdlib.h"

struct node {
   struct node *next;
   int value;
};

struct stack {
   struct node *head;
};

/*@
```

```
predicate nodes(struct node *node, int count) =
   node == 0 ?
      count == 0
   :
      0 < count &*& node->next |-> ?next &*& node->value |-> ?value &*&
      malloc_block_node(node) &*& nodes(next, count - 1);

predicate stack(struct stack *stack, int count) =
   stack->head |-> ?head &*& malloc_block_stack(stack) &*& 0 <= count &*& nodes(head, count);

@*/

struct stack *create_stack()
   //@ requires true;
   //@ ensures stack(result, 0);
{
   struct stack *stack = malloc(sizeof(struct stack));
   if (stack == 0) { abort(); }
   stack->head = 0;
   //@ close nodes(0, 0);
   //@ close stack(stack, 0);
   return stack;
}

void stack_push(struct stack *stack, int value)
   //@ requires stack(stack, ?count);
   //@ ensures stack(stack, count + 1);
{
   //@ open stack(stack, count);
   struct node *n = malloc(sizeof(struct node));
   if (n == 0) { abort(); }
   n->next = stack->head;
   n->value = value;
   stack->head = n;
   //@ close nodes(n, count + 1);
   //@ close stack(stack, count + 1);
}

int stack_pop(struct stack *stack)
   //@ requires stack(stack, ?count) &*& 0 < count;
   //@ ensures stack(stack, count - 1);
{
   //@ open stack(stack, count);
   struct node *head = stack->head;
   //@ open nodes(head, count);
   int result = head->value;
   stack->head = head->next;
   free(head);
   //@ close stack(stack, count - 1);
   return result;
}

typedef bool int_predicate(int x);
```

```
   //@ requires true;
   //@ ensures true;

struct node *nodes_filter(struct node *n, int_predicate *p)
   //@ requires nodes(n, _) &*& is_int_predicate(p) == true;
   //@ ensures nodes(result, _);
{
   if (n == 0) {
      return 0;
   } else {
      //@ open nodes(n, _);
      bool keep = p(n->value);
      if (keep) {
         struct node *next = nodes_filter(n->next, p);
         //@ open nodes(next, ?count);
         //@ close nodes(next, count);
         n->next = next;
         //@ close nodes(n, count + 1);
         return n;
      } else {
         struct node *next = n->next;
         free(n);
         struct node *result = nodes_filter(next, p);
         return result;
      }
   }
}

void stack_filter(struct stack *stack, int_predicate *p)
   //@ requires stack(stack, _) &*& is_int_predicate(p) == true;
   //@ ensures stack(stack, _);
{
   //@ open stack(stack, _);
   struct node *head = nodes_filter(stack->head, p);
   //@ assert nodes(head, ?count);
   stack->head = head;
   //@ open nodes(head, count);
   //@ close nodes(head, count);
   //@ close stack(stack, count);
}

void nodes_dispose(struct node *n)
   //@ requires nodes(n, _);
   //@ ensures true;
{
   //@ open nodes(n, _);
   if (n != 0) {
      nodes_dispose(n->next);
      free(n);
   }
}

void stack_dispose(struct stack *stack)
```

```
   //@ requires stack(stack, _);
   //@ ensures true;
{
   //@ open stack(stack, _);
   nodes_dispose(stack->head);
   free(stack);
}

bool neq_20(int x) //@ : int_predicate
   //@ requires true;
   //@ ensures true;
{
   return x != 20;
}

int main()
   //@ requires true;
   //@ ensures true;
{
   struct stack *s = create_stack();
   stack_push(s, 10);
   stack_push(s, 20);
   stack_push(s, 30);
   stack_filter(s, neq_20);
   stack_dispose(s);
   return 0;
}
```

31.16 Exercise 16: byref.c

```
void nodes_filter(struct node **n, int_predicate *p)
   //@ requires pointer(n, ?node) &*& nodes(node, _) &*& is_int_predicate(p) == true;
   //@ ensures pointer(n, ?node0) &*& nodes(node0, _);
{
   if (*n != 0) {
      //@ open nodes(node, _);
      bool keep = p((*n)->value);
      if (keep) {
         //@ open node_next(node, _);
         nodes_filter(&(*n)->next, p);
         //@ assert pointer(&((struct node *)node)->next, ?next) &*& nodes(next, ?count);
         //@ close node_next(node, next);
         //@ open nodes(next, count);
         //@ close nodes(next, count);
         //@ close nodes(node, count + 1);
      } else {
         struct node *next = (*n)->next;
         free(*n);
         *n = next;
         nodes_filter(n, p);
      }
   }
}
```

31.17 Exercise 17: map.c

```
//@ predicate_family int_func_data(void *f)(void *data);

typedef int int_func(void *data, int x);
   //@ requires int_func_data(this)(data);
   //@ ensures int_func_data(this)(data);

void nodes_map(struct node *n, int_func *f, void *data)
   //@ requires nodes(n, ?count) &*& is_int_func(f) == true &*& int_func_data(f)(data);
   //@ ensures nodes(n, count) &*& is_int_func(f) == true &*& int_func_data(f)(data);
{
   //@ open nodes(n, _);
   if (n != 0) {
      int y = f(data, n->value);
      n->value = y;
      nodes_map(n->next, f, data);
   }
   //@ close nodes(n, count);
}

void stack_map(struct stack *stack, int_func *f, void *data)
   //@ requires stack(stack, ?count) &*& is_int_func(f) == true &*& int_func_data(f)(data);
   //@ ensures stack(stack, count) &*& is_int_func(f) == true &*& int_func_data(f)(data);
{
   //@ open stack(stack, _);
   nodes_map(stack->head, f, data);
   //@ close stack(stack, count);
}
```

31.18 Exercise 18: foreach.c

```
char input_char();
   //@ requires true;
   //@ ensures true;

int input_int();
   //@ requires true;
   //@ ensures true;

void output_int(int x);
   //@ requires true;
   //@ ensures true;

struct vector {
   int x;
   int y;
};

//@ predicate vector(struct vector *v) = v->x |-> _ &*& v->y |-> _ &*& malloc_block_vector(v);

struct vector *create_vector(int x, int y)
   //@ requires true;
```

```
   //@ ensures vector(result);
{
   struct vector *result = malloc(sizeof(struct vector));
   if (result == 0) abort();
   result->x = x;
   result->y = y;
   //@ close vector(result);
   return result;
}

int main()
   //@ requires true;
   //@ ensures true;
{
   struct stack *s = create_stack();
   //@ close foreach(nil, vector);
   while (true)
      //@ invariant stack(s, ?values) &*& foreach(values, vector);
   {
      char c = input_char();
      if (c == 'p') {
         int x = input_int();
         int y = input_int();
         struct vector *v = create_vector(x, y);
         stack_push(s, v);
         //@ close foreach(cons(v, values), vector);
      } else if (c == '+') {
         bool empty = stack_is_empty(s);
         if (empty) abort();
         struct vector *v1 = stack_pop(s);
         //@ open foreach(values, vector);
         //@ open vector(head(values));
         empty = stack_is_empty(s);
         if (empty) abort();
         struct vector *v2 = stack_pop(s);
         //@ open foreach(tail(values), vector);
         //@ open vector(head(tail(values)));
         struct vector *sum = create_vector(v1->x + v2->x, v1->y + v2->y);
         free(v1);
         free(v2);
         stack_push(s, sum);
         //@ close foreach(cons(sum, tail(tail(values))), vector);
      } else if (c == '=') {
         bool empty = stack_is_empty(s);
         if (empty) abort();
         struct vector *v = stack_pop(s);
         //@ open foreach(values, vector);
         //@ open vector(head(values));
         output_int(v->x);
         output_int(v->y);
         free(v);
      } else {
         abort();
```

```
      }
   }
}
```

31.19 Exercise 19: predctors.c

```
struct vector {
   int x;
   int y;
};

/*@
predicate_ctor vector(int limit)(struct vector *v) =
   v->x |-> ?x &*& v->y |-> ?y &*& malloc_block_vector(v) &*& x * x + y * y <= limit * limit;
@*/

struct vector *create_vector(int limit, int x, int y)
   //@ requires true;
   //@ ensures vector(limit)(result);
{
   if (x * x + y * y > limit * limit) abort();
   struct vector *result = malloc(sizeof(struct vector));
   if (result == 0) abort();
   result->x = x;
   result->y = y;
   //@ close vector(limit)(result);
   return result;
}

int main()
   //@ requires true;
   //@ ensures true;
{
   int limit = input_int();
   struct stack *s = create_stack();
   //@ close foreach(nil, vector(limit));
   while (true)
      //@ invariant stack(s, ?values) &*& foreach(values, vector(limit));
   {
      char c = input_char();
      if (c == 'p') {
         int x = input_int();
         int y = input_int();
         struct vector *v = create_vector(limit, x, y);
         stack_push(s, v);
         //@ close foreach(cons(v, values), vector(limit));
      } else if (c == '+') {
         bool empty = stack_is_empty(s);
         if (empty) abort();
         struct vector *v1 = stack_pop(s);
         //@ open foreach(values, vector(limit));
         //@ open vector(limit)(head(values));
         empty = stack_is_empty(s);
```

```
         if (empty) abort();
         struct vector *v2 = stack_pop(s);
         //@ open foreach(tail(values), vector(limit));
         //@ open vector(limit)(head(tail(values)));
         struct vector *sum = create_vector(limit, v1->x + v2->x, v1->y + v2->y);
         free(v1);
         free(v2);
         stack_push(s, sum);
         //@ close foreach(cons(sum, tail(tail(values))), vector(limit));
      } else if (c == '=') {
         bool empty = stack_is_empty(s);
         if (empty) abort();
         struct vector *v = stack_pop(s);
         //@ open foreach(values, vector(limit));
         //@ open vector(limit)(head(values));
         int x = v->x;
         int y = v->y;
         free(v);
         assert(x * x + y * y <= limit * limit);
         output_int(x);
         output_int(y);
      } else {
         abort();
      }
   }
}
```

31.20 Exercise 20: threads0.c

```
#include "stdlib.h"
#include "malloc.h"

int rand();
   //@ requires true;
   //@ ensures true;

int fac(int x)
   //@ requires true;
   //@ ensures true;
{
   int result = 1;
   while (x > 1)
      //@ invariant true;
   {
      result = result * x;
      x = x - 1;
   }
   return result;
}

struct tree {
   struct tree *left;
   struct tree *right;
```

```
   int value;
};

/*@ predicate tree(struct tree *t, int depth) =
   t == 0 ?
      depth == 0
   :
      t->left |-> ?left &*& t->right |-> ?right &*& t->value |-> _ &*& malloc_block_tree(t) &*&
      tree(left, depth - 1) &*& tree(right, depth - 1);
@*/

struct tree *make_tree(int depth)
   //@ requires true;
   //@ ensures tree(result, depth);
{
   if (depth == 0) {
      //@ close tree(0, 0);
      return 0;
   } else {
      struct tree *left = make_tree(depth - 1);
      struct tree *right = make_tree(depth - 1);
      int value = rand();
      struct tree *t = malloc(sizeof(struct tree));
      if (t == 0) abort();
      t->left = left;
      t->right = right;
      t->value = value % 2000;
      //@ close tree(t, depth);
      return t;
   }
}

int tree_compute_sum_facs(struct tree *tree)
   //@ requires tree(tree, ?depth);
   //@ ensures tree(tree, depth);
{
   if (tree == 0) {
      return 1;
   } else {
      //@ open tree(tree, depth);
      int leftSum = tree_compute_sum_facs(tree->left);
      int rightSum = tree_compute_sum_facs(tree->right);
      int f = fac(tree->value);
      return leftSum + rightSum + f;
      //@ close tree(tree, depth);
   }
}

int main()
   //@ requires true;
   //@ ensures true;
{
   struct tree *tree = make_tree(22);
```

```
   int sum = tree_compute_sum_facs(tree);
   //@ leak tree(tree, _);
   return sum;
}
```

31.21 Exercise 21: threads.c

```
struct sum_data {
   struct thread *thread;
   struct tree *tree;
   int sum;
};

/*@

predicate_family_instance thread_run_pre(summator)(struct sum_data *data, any info) =
   data->tree |-> ?tree &*& tree(tree, _) &*& data->sum |-> _;
predicate_family_instance thread_run_post(summator)(struct sum_data *data, any info) =
   data->tree |-> ?tree &*& tree(tree, _) &*& data->sum |-> _;

@*/

void summator(struct sum_data *data) //@ : thread_run_joinable
   //@ requires thread_run_pre(summator)(data, ?info);
   //@ ensures thread_run_post(summator)(data, info);
{
   //@ open thread_run_pre(summator)(data, info);
   int sum = tree_compute_sum_facs(data->tree);
   data->sum = sum;
   //@ close thread_run_post(summator)(data, info);
}

struct sum_data *start_sum_thread(struct tree *tree)
   //@ requires tree(tree, _);
   //@ ensures result->thread |-> ?t &*& thread(t, summator, result, _);
{
   struct sum_data *data = malloc(sizeof(struct sum_data));
   struct thread *t = 0;
   if (data == 0) abort();
   //@ leak malloc_block_sum_data(data);
   data->tree = tree;
   //@ close thread_run_pre(summator)(data, unit);
   t = thread_start_joinable(summator, data);
   data->thread = t;
   return data;
}

int join_sum_thread(struct sum_data *data)
   //@ requires data->thread |-> ?t &*& thread(t, summator, data, _);
   //@ ensures true;
{
   thread_join(data->thread);
   //@ open thread_run_post(summator)(data, _);
```

```
   return data->sum;
   //@ leak data->tree |-> ?tree &*& tree(tree, _) &*& data->sum |-> _ &*& data->thread |-> _;
}

int main()
   //@ requires true;
   //@ ensures true;
{
   struct tree *tree = make_tree(22);
   //@ open tree(tree, _);
   struct sum_data *leftData = start_sum_thread(tree->left);
   struct sum_data *rightData = start_sum_thread(tree->right);
   int sumLeft = join_sum_thread(leftData);
   int sumRight = join_sum_thread(rightData);
   int f = fac(tree->value);
   //@ leak tree->left |-> _ &*& tree->right |-> _ &*& tree->value |-> _ &*& malloc_block_tree(tree);
   return sumLeft + sumRight + f;
}
```

31.22 Exercise 22: fractions0.c

```
typedef int fold_function(int acc, int x);
   //@ requires true;
   //@ ensures true;

int tree_fold(struct tree *tree, fold_function *f, int acc)
   //@ requires tree(tree, ?depth) &*& is_fold_function(f) == true;
   //@ ensures tree(tree, depth);
{
   if (tree == 0) {
      return acc;
   } else {
      //@ open tree(tree, depth);
      acc = tree_fold(tree->left, f, acc);
      acc = tree_fold(tree->right, f, acc);
      acc = f(acc, tree->value);
      return acc;
      //@ close tree(tree, depth);
   }
}

struct fold_data {
   struct thread *thread;
   struct tree *tree;
   fold_function *f;
   int acc;
};

/*@

predicate_family_instance thread_run_pre(folder)(struct fold_data *data, any info) =
   data->tree |-> ?tree &*& tree(tree, _) &*&
   data->f |-> ?f &*& is_fold_function(f) == true &*& data->acc |-> _;
```

```
predicate_family_instance thread_run_post(folder)(struct fold_data *data, any info) =
   data->tree |-> ?tree &*& tree(tree, _) &*&
   data->f |-> ?f &*& is_fold_function(f) == true &*& data->acc |-> _;

@*/

void folder(struct fold_data *data) //@ : thread_run_joinable
   //@ requires thread_run_pre(folder)(data, ?info);
   //@ ensures thread_run_post(folder)(data, info);
{
   //@ open thread_run_pre(folder)(data, info);
   int acc = tree_fold(data->tree, data->f, data->acc);
   data->acc = acc;
   //@ close thread_run_post(folder)(data, info);
}

struct fold_data *start_fold_thread(struct tree *tree, fold_function *f, int acc)
   //@ requires tree(tree, _) &*& is_fold_function(f) == true;
   //@ ensures result->thread |-> ?t &*& thread(t, folder, result, _);
{
   struct fold_data *data = malloc(sizeof(struct fold_data));
   struct thread *t = 0;
   if (data == 0) abort();
   //@ leak malloc_block_fold_data(data);
   data->tree = tree;
   data->f = f;
   data->acc = acc;
   //@ close thread_run_pre(folder)(data, unit);
   t = thread_start_joinable(folder, data);
   data->thread = t;
   return data;
}

int join_fold_thread(struct fold_data *data)
   //@ requires data->thread |-> ?t &*& thread(t, folder, data, _);
   //@ ensures true;
{
   thread_join(data->thread);
   //@ open thread_run_post(folder)(data, _);
   return data->acc;
   //@ leak data->tree |-> ?tree &*& [_]tree(tree, _);
   //@ leak data->f |-> _ &*& data->acc |-> _ &*& data->thread |-> _;
}

int sum_function(int acc, int x) //@ : fold_function
   //@ requires true;
   //@ ensures true;
{
   int f = fac(x);
   return acc + f;
}

int product_function(int acc, int x) //@ : fold_function
```

```
   //@ requires true;
   //@ ensures true;
{
   int f = fac(x);
   return acc * f;
}

int main()
   //@ requires true;
   //@ ensures true;
{
   struct tree *tree = make_tree(22);
   //@ open tree(tree, _);
   struct fold_data *leftData = start_fold_thread(tree->left, sum_function, 0);
   struct fold_data *rightData = start_fold_thread(tree->right, sum_function, 0);
   int sumLeft = join_fold_thread(leftData);
   int sumRight = join_fold_thread(rightData);
   int f = fac(tree->value);
   //@ leak tree->left |-> _ &*& tree->right |-> _ &*& tree->value |-> _ &*& malloc_block_tree(tree);
   return sumLeft + sumRight + f;
}
```

31.23 Exercise 23: fractions.c

```
/*@

predicate_family_instance thread_run_pre(folder)(struct fold_data *data, any info) =
   data->tree |-> ?tree &*& [1/2]tree(tree, _) &*&
   data->f |-> ?f &*& is_fold_function(f) == true &*& data->acc |-> _;
predicate_family_instance thread_run_post(folder)(struct fold_data *data, any info) =
   data->tree |-> ?tree &*& [1/2]tree(tree, _) &*&
   data->f |-> ?f &*& is_fold_function(f) == true &*& data->acc |-> _;

@*/

void folder(struct fold_data *data) //@ : thread_run_joinable
   //@ requires thread_run_pre(folder)(data, ?info);
   //@ ensures thread_run_post(folder)(data, info);
{
   //@ open thread_run_pre(folder)(data, info);
   int acc = tree_fold(data->tree, data->f, data->acc);
   data->acc = acc;
   //@ close thread_run_post(folder)(data, info);
}

struct fold_data *start_fold_thread(struct tree *tree, fold_function *f, int acc)
   //@ requires [1/2]tree(tree, _) &*& is_fold_function(f) == true;
   //@ ensures result->thread |-> ?t &*& thread(t, folder, result, _);
{
   struct fold_data *data = malloc(sizeof(struct fold_data));
   struct thread *t = 0;
   if (data == 0) abort();
   //@ leak malloc_block_fold_data(data);
```

```
    data->tree = tree;
    data->f = f;
    data->acc = acc;
    //@ close thread_run_pre(folder)(data, unit);
    t = thread_start_joinable(folder, data);
    data->thread = t;
    return data;
}

int join_fold_thread(struct fold_data *data)
    //@ requires data->thread |-> ?t &*& thread(t, folder, data, _);
    //@ ensures true;
{
    thread_join(data->thread);
    //@ open thread_run_post(folder)(data, _);
    return data->acc;
    //@ leak data->tree |-> ?tree &*& [_]tree(tree, _);
    //@ leak data->f |-> _ &*& data->acc |-> _ &*& data->thread |-> _;
}
```

31.24 Exercise 24: mutexes.c

```
#include "stdlib.h"
#include "threading.h"

void wait_for_pulse(int source);
    //@ requires true;
    //@ ensures true;

void sleep(int millis);
    //@ requires true;
    //@ ensures true;

void print_int(int n);
    //@ requires true;
    //@ ensures true;

struct counter {
    int count;
    struct mutex *mutex;
};

//@ predicate_ctor counter(struct counter *counter)() = counter->count |-> _;

struct count_pulses_data {
    struct counter *counter;
    int source;
};

/*@

predicate_family_instance thread_run_data(count_pulses)(struct count_pulses_data *data) =
    data->counter |-> ?counter &*& data->source |-> _ &*& malloc_block_count_pulses_data(data) &*&
```

```
   [1/2]counter->mutex |-> ?mutex &*& [1/3]mutex(mutex, counter(counter));

@*/

void count_pulses(struct count_pulses_data *data) //@ : thread_run
   //@ requires thread_run_data(count_pulses)(data);
   //@ ensures true;
{
   //@ open thread_run_data(count_pulses)(data);
   struct counter *counter = data->counter;
   int source = data->source;
   free(data);

   struct mutex *mutex = counter->mutex;

   while (true)
      //@ invariant [1/3]mutex(mutex, counter(counter));
   {
      wait_for_pulse(source);
      mutex_acquire(mutex);
      //@ open counter(counter)();
      counter->count++;
      //@ close counter(counter)();
      mutex_release(mutex);
   }
}

void count_pulses_async(struct counter *counter, int source)
   //@ requires [1/2]counter->mutex |-> ?mutex &*& [1/3]mutex(mutex, counter(counter));
   //@ ensures true;
{
   struct count_pulses_data *data = malloc(sizeof(struct count_pulses_data));
   if (data == 0) abort();
   data->counter = counter;
   data->source = source;
   //@ close thread_run_data(count_pulses)(data);
   thread_start(count_pulses, data);
}

int main() //@ : main
   //@ requires true;
   //@ ensures true;
{
   struct counter *counter = malloc(sizeof(struct counter));
   if (counter == 0) abort();
   counter->count = 0;
   //@ close counter(counter)();
   //@ close create_mutex_ghost_arg(counter(counter));
   struct mutex *mutex = create_mutex();
   counter->mutex = mutex;

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

```
   while (true)
      //@ invariant [1/3]mutex(mutex, counter(counter));
   {
      sleep(1000);
      mutex_acquire(mutex);
      //@ open counter(counter)();
      print_int(counter->count);
      //@ close counter(counter)();
      mutex_release(mutex);
   }
}
```

31.25 Exercise 25: leaks.c

```
#include "stdlib.h"
#include "threading.h"

int wait_for_source();
   //@ requires true;
   //@ ensures true;

bool wait_for_pulse(int source); // true means the sensor has been removed.
   //@ requires true;
   //@ ensures true;

void sleep(int millis);
   //@ requires true;
   //@ ensures true;

void print_int(int n);
   //@ requires true;
   //@ ensures true;

struct counter {
   int count;
   struct mutex *mutex;
};

//@ predicate_ctor counter(struct counter *counter)() = counter->count |-> _;

struct count_pulses_data {
   struct counter *counter;
   int source;
};

/*@

predicate_family_instance thread_run_data(count_pulses)(struct count_pulses_data *data) =
   data->counter |-> ?counter &*& data->source |-> _ &*& malloc_block_count_pulses_data(data) &*&
   [_]counter->mutex |-> ?mutex &*& [_]mutex(mutex, counter(counter));

@*/
```

```
void count_pulses(struct count_pulses_data *data) //@ : thread_run
   //@ requires thread_run_data(count_pulses)(data);
   //@ ensures true;
{
   //@ open thread_run_data(count_pulses)(data);
   struct counter *counter = data->counter;
   int source = data->source;
   free(data);

   struct mutex *mutex = counter->mutex;
   bool done = false;
   while (!done)
      //@ invariant [_]mutex(mutex, counter(counter));
   {
      done = wait_for_pulse(source);
      if (!done) {
         mutex_acquire(mutex);
         //@ open counter(counter)();
         counter->count++;
         //@ close counter(counter)();
         mutex_release(mutex);
      }
   }
}

void count_pulses_async(struct counter *counter, int source)
   //@ requires [_]counter->mutex |-> ?mutex &*& [_]mutex(mutex, counter(counter));
   //@ ensures true;
{
   struct count_pulses_data *data = malloc(sizeof(struct count_pulses_data));
   if (data == 0) abort();
   data->counter = counter;
   data->source = source;
   //@ close thread_run_data(count_pulses)(data);
   thread_start(count_pulses, data);
}

/*@

predicate_family_instance thread_run_data(print_count)(struct counter *counter) =
   [_]counter->mutex |-> ?mutex &*& [_]mutex(mutex, counter(counter));

@*/

void print_count(struct counter *counter) //@ : thread_run
   //@ requires thread_run_data(print_count)(counter);
   //@ ensures true;
{
   //@ open thread_run_data(print_count)(counter);
   struct mutex *mutex = counter->mutex;
   while (true)
      //@ invariant [_]mutex(mutex, counter(counter));
```

```
    {
        sleep(1000);
        mutex_acquire(mutex);
        //@ open counter(counter)();
        print_int(counter->count);
        //@ close counter(counter)();
        mutex_release(mutex);
    }
}

int main() //@ : main
    //@ requires true;
    //@ ensures true;
{
    struct counter *counter = malloc(sizeof(struct counter));
    if (counter == 0) abort();
    counter->count = 0;
    //@ close counter(counter)();
    //@ close create_mutex_ghost_arg(counter(counter));
    struct mutex *mutex = create_mutex();
    counter->mutex = mutex;
    //@ leak counter->mutex |-> mutex &*& mutex(mutex, _);

    //@ close thread_run_data(print_count)(counter);
    thread_start(print_count, counter);

    while (true)
        //@ invariant [_]counter->mutex |-> mutex &*& [_]mutex(mutex, counter(counter));
    {
        int source = wait_for_source();
        count_pulses_async(counter, source);
    }
}
```

31.26 Exercise 26: characters.c

The solution is given in the text of Section 25.

31.27 Exercise 27: xor.c

```
char getchar(); /*@ requires true; @*/ /*@ ensures true; @*/
void putchar(char c); /*@ requires true; @*/ /*@ ensures true; @*/

/*@
predicate characters(char *start, int count) =
    count <= 0 ? true : character(start, _) &*& characters(start + 1, count - 1);
@*/

char *malloc(int count);
    //@ requires true;
    //@ ensures characters(result, count);

void getchars(char *start, int count)
```

```
   //@ requires characters(start, count);
   //@ ensures characters(start, count);
{
   if (count > 0) {
      //@ open characters(start, count);
      char c = getchar();
      *start = c;
      getchars(start + 1, count - 1);
      //@ close characters(start, count);
   }
}

void xorchars(char *text, char *key, int count)
   //@ requires characters(text, count) &*& characters(key, count);
   //@ ensures characters(text, count) &*& characters(key, count);
{
   if (count > 0) {
      //@ open characters(text, count);
      //@ open characters(key, count);
      *text = (char)(*text ^ *key);
      xorchars(text + 1, key + 1, count - 1);
      //@ close characters(text, count);
      //@ close characters(key, count);
   }
}

void putchars(char *start, int count)
   //@ requires characters(start, count);
   //@ ensures characters(start, count);
{
   if (count > 0) {
      //@ open characters(start, count);
      char c = *start;
      putchar(c);
      putchars(start + 1, count - 1);
      //@ close characters(start, count);
   }
}

int main() /*@ requires true; @*/ /*@ ensures true; @*/
{
   char *text = malloc(10);
   char *key = malloc(10);
   getchars(text, 10);
   getchars(key, 10);
   xorchars(text, key, 10);
   putchars(text, 10);
   //@ leak characters(text, 10) &*& characters(key, 10);
   return 0;
}
```

31.28 Exercise 28: characters_loop.c

```
char getchar(); /*@ requires true; @*/ /*@ ensures true; @*/
void putchar(char c); /*@ requires true; @*/ /*@ ensures true; @*/

/*@
predicate characters(char *start, int count) =
   count <= 0 ? true : character(start, _) &*& characters(start + 1, count - 1);
@*/

char *malloc(int count);
   //@ requires true;
   //@ ensures characters(result, count);

/*@

lemma void split_characters_chunk(char *start, int i)
   requires characters(start, ?count) &*& 0 <= i &*& i <= count;
   ensures characters(start, i) &*& characters(start + i, count - i);
{
   if (i == 0) {
      close characters(start, 0);
   } else {
      open characters(start, count);
      split_characters_chunk(start + 1, i - 1);
      close characters(start, i);
   }
}

lemma void merge_characters_chunk(char *start)
   requires characters(start, ?i) &*& characters(start + i, ?count) &*& 0 <= i &*& 0 <= count;
   ensures characters(start, i + count);
{
   open characters(start, i);
   if (i != 0) {
      merge_characters_chunk(start + 1);
      close characters(start, i + count);
   }
}

@*/

void getchars(char *start, int count)
   //@ requires characters(start, count);
   //@ ensures characters(start, count);
{
   for (int i = 0; i < count; i++)
      //@ invariant characters(start, count) &*& 0 <= i;
   {
      char c = getchar();
      //@ split_characters_chunk(start, i);
      //@ open characters(start + i, count - i);
      *(start + i) = c;
      //@ close characters(start + i, count - i);
      //@ merge_characters_chunk(start);
```

```
   }
}

void putchars(char *start, int count)
   //@ requires characters(start, count);
   //@ ensures characters(start, count);
{
   for (int i = 0; i < count; i++)
      //@ invariant characters(start, count) &*& 0 <= i;
   {
      //@ split_characters_chunk(start, i);
      //@ open characters(start + i, count - i);
      char c = *(start + i);
      //@ close characters(start + i, count - i);
      //@ merge_characters_chunk(start);
      putchar(c);
   }
}

int main() /*@ requires true; @*/ /*@ ensures true; @*/
{
   char *array = malloc(100);
   getchars(array, 100);
   putchars(array, 100);
   putchars(array, 100);
   //@ leak characters(array, 100);
   return 0;
}
```

31.29 Exercise 29: tuerk.c

```
void putchars(char *start, int count)
   //@ requires characters(start, count);
   //@ ensures characters(start, count);
{
   int i = 0;
   while (i < count)
      //@ requires characters(start + i, count - i);
      //@ ensures characters(start + old_i, count - old_i);
   {
      //@ open characters(start + i, count - i);
      putchar(*(start + i));
      i++;
      //@ recursive_call();
      //@ close characters(start + old_i, count - old_i);
   }
}
```

31.30 Exercise 30: stack_tuerk.c

```
int stack_get_count(struct stack *stack)
   //@ requires stack(stack, ?count);
   //@ ensures stack(stack, count) &*& result == count;
{
```

```
   //@ open stack(stack, count);
   struct node *n = stack->head;
   int i = 0;
   for (;;)
      //@ requires nodes(n, ?count1);
      //@ ensures nodes(old_n, count1) &*& i == old_i + count1;
   {
      //@ open nodes(n, count1);
      if (n == 0) {
         //@ close nodes(n, count1);
         break;
      }
      n = n->next;
      i++;
      //@ recursive_call();
      //@ close nodes(old_n, count1);
   }
   //@ close stack(stack, count);
   return i;
}
```

31.31 Exercise 31: memcmp.c

```
int memcmp(char *p1, char *p2, int count)
   //@ requires [?f1]p1[0..count] |-> ?cs1 &*& [?f2]p2[0..count] |-> ?cs2;
   /*@
   ensures
      [f1]p1[0..count] |-> cs1 &*& [f2]p2[0..count] |-> cs2 &*&
      true == ((result == 0) == (cs1 == cs2));
   @*/
{
   int result = 0;
   for (int i = 0; ; i++)
      //@ requires [f1]p1[i..count] |-> ?xs1 &*& [f2]p2[i..count] |-> ?xs2 &*& result == 0;
      /*@
      ensures
         [f1]p1[old_i..count] |-> xs1 &*& [f2]p2[old_i..count] |-> xs2 &*&
         true == ((result == 0) == (xs1 == xs2));
      @*/
   {
      //@ open chars(p1 + i, _, _);
      //@ open chars(p2 + i, _, _);
      if (i == count) {
         break;
      }
      if (p1[i] < p2[i]) {
         result = -1; break;
      }
      if (p1[i] > p2[i]) {
         result = 1; break;
      }
   }
   return result;
```

```
}
```

31.32 Exercise 32: strlen.c

```
int strlen(char *s)
   //@ requires [?f]string(s, ?cs);
   //@ ensures [f]string(s, cs) &*& result == length(cs);
{
   int i = 0;
   for (;; i++)
      //@ requires [f]string(s + i, ?cs1);
      //@ ensures [f]string(s + old_i, cs1) &*& i == old_i + length(cs1);
   {
      //@ open [f]string(s + i, cs1);
      if (s[i] == 0) {
         break;
      }
   }
   return i;
}

int main()
   //@ requires true;
   //@ ensures true;
{
   int n = strlen("Hello, world!");
   assert(n == 13);
   return 0;
}
```