Verifying Concurrent C Programs with VCC

Working draft, version 0.0, July 17, 2010

Michał Moskal, Wolfram Schulte
Microsoft Research Redmond

Ernie Cohen, Stephan Tobies
European Microsoft Innovation Center

Abstract

This tutorial provides basic information about developing specifications and annotations for concurrent C programs, so that they can be verified with VCC. [**TODO:** add more]

1 Introduction

This tutorial is an introduction to verifying C code with VCC. Our primary audience is programmers who want to write correct code and verification engineers who want to check code for correctness.[1] The only prerequisite is a working understanding of C.

To use VCC, you must first *annotate* your code to specify how your functions and data structures are meant to be used, and what your functions guarantee to their callers. VCC then takes your program and tries to *prove* (mathematically) that your program meets these specifications. Unlike most program analyzers, VCC doesn't look for bugs, or analyze an abstraction of your program; if VCC certifies that your program is correct, then your program really should be correct [2].

To check your program, VCC generates a number of mathematical statements (called ***verification conditions***) whose proofs suffice to guarantee the program's correctness, and tries to prove these statements using an automatic theorem prover. If any of these proofs fail, VCC reflects these failures back to you in terms of the program itself (as opposed to the formulas used in the

[1] If you're instead looking for a high-level overview of VCC are referred to our system description [1]. If you interested in the VCC concurrency verification methodology (as opposed to verifying concurrent programs with VCC), please see our CAV 2010 paper [2].

[2] In reality, this isn't really a guarantee, because VCC itself might have bugs. But in practice, this is unlikely to cause you to accidentally verify an incorrect program, unless you find and intentionally exploit such a bug. Another source of bugs that could slip into verified software would be the compiler, the operating system, and the hardware. These are generally more likely to introduce bug than VCC.

theorem prover). For example, if your program uses division somewhere, and VCC is unable to prove (from what it thinks holds at the point at which the division is done) that the divisor is nonzero, it will report this to you as a program error at that point in the program. This doesn't mean that your program is necessarily incorrect; most of the time, especially when verifying code that is already well-tested, it is because your specifications aren't strong enough to guarantee that the suspected error doesn't occur. Typically, you fix this "error" by strengthening your specifications. This might lead to other error reports, necessitating the strengthening of other specifications, so verification is in practice an iterative process. Quite often this process will reveal a genuine programming error.

[**TODO:** the following paragraph should be moved out of here] Annotating your program sometimes requires doing some extra programming. For example, your annotations might need to talk about a set of users of a data structure, where the implementation only maintains a count of them. You would then need to include ***ghost data*** to store this set and write small bits of ***ghost code*** to update it. Ghost code is seen by VCC but not by the C compiler, and so introduces no runtime overhead. Part of the VCC philosophy is that programmers would rather do extra programming than drive interactive theorem provers, so ghost code is the preferred way to help VCC understand why your program works. Thus, you interact with VCC entirely at the level of code and program states; usually, you can safely ignore the mathematical reasoning going on "under the hood".

This tutorial covers basics of VCC annotation language. By the time you have finished working through it, you should be able to use VCC to verify some nontrivial programs. It doesn't cover the theoretical background of VCC, implementation details, or advanced topics; information on these can be found on the VCC homepage[3].

You can use VCC either from the command line or from Visual Studio 2008 (VS). The VS interface offers easy access to different components of VCC tool chain and is thus generally recommended, but invoking VCC from command line will be also covered. VCC can be downloaded from its homepage. Make sure to check out installation instructions[4], as they contain important information about installation prerequisites and setting include paths.

Throughout the tutorial, we'll use notes like this one to discuss topics, which can be skipped on the first reading, either because they are somewhat more advanced, arcane, or not so important.

Because this is a tutorial, we will occasionally provide a simplified (and therefore not strictly correct) explanation of what is going on, providing some additional clarification in the footnotes,[5] which can be skipped on first reading.

Definitions will be introduced with ***italic boldface***.

[3] `http://vcc.codeplex.com/`

[4] `http://vcc.codeplex.com/wikipage?title=Install`

[5] Like this one.

1.1 Notes on VCC Invocation

To enable the VCC syntax which is described in this tutorial you need to use the flag `/2`. Additionally, the trigger inference, which we rely upon for the more complex invariant is enabled by the `/it` flag. Both will be on by default soon, but for now, please supply them.

1.2 Top-level To-Do

- fix the `lsearch()` array readability annotations
- explain the default trigger selection algorithm and command line switches for it

2 Assert and assume

Let's begin with a simple example:

```
#include <vcc.h>

int main()
{
  int x,y,z;
  if (x <= y)
    z = x;
  else z = y;
  _(assert z <= x)
  return 0;
}
```

This program (internally) sets `z` to the minimum of `x` and `y`. In addition to the ordinary C code, this program includes an ***annotation***, starting with `_(`, terminating with a closing parenthesis, with balanced parentheses inside. The first identifier after the opening parenthesis (in the program above it's `assert`) is called ***annotation tag*** and identifies the type of annotation provided (and hence its function). (The tag plays this assertion-specific role only at the beginning of an annotation; elsewhere, it would be treated as an ordinary program identifier.)

An annotation of the form `_(assert E)`, called an ***assertion***, asks VCC to prove that `E` is guaranteed to hold (i.e., , evaluate to a value other than `0`) whenever control reaches this assertion[6]. Thus, the line `_(assert z <= x)` says that when control reaches the assertion, `z` is no larger than `x`. If VCC verifies a program, it promises that all such assertions hold in *every* possible execution of the program, considering all possible inputs, all possible ways of scheduling concurrent threads, etc.

[6] This interpretation changes slightly if `E` refers to memory locations that might be concurrently written to by other threads; see Section 7

[**TODO:** should these notes be less prominent?]

> It is instructive to compare `_(assert E)` with the macro `assert(E)` (defined in `<assert.h>`), which evaluates `E` at runtime and aborts execution if `E` doesn't hold. First, `assert(E)` requires runtime overhead (at least in builds where the check is made), whereas `_(assert E)` does not. Second, `assert(E)` will catch failure of the assertion only when it actually fails in an execution, whereas `_(assert E)` is guaranteed to catch the failure if it is possible in *any* execution. Third, because VCC doesn't have to actually execute `_(assert E)`, it allows `E` to include mathematical operations that cannot be executed, such as quantification over infinite domains.

To verify this function using VCC from the command line, save the source in a file called `minimum.c` and run VCC as follows:

```
C:\Somewhere\VCC Tutorial> vcc.exe minimum.c
Verification of main succeeded.
C:\Somewhere\VCC Tutorial>
```

If instead you wish to use VCC Visual Studio plugin, load the solution `VCCTutorial.sln` (distributed with VCC), [**TODO:** This isn't yet true. The samples are available only with VCC sources, available from `http://vcc.codeplex.com/SourceControl/list/changesets`: click Download on the right, get the zip file and navigate to `vcc/Docs/Tutorial/c`.] [**TODO:** we need to package the examples, but only after it is clear which ones are included in the tutorial] locate the file with the example, and right-click on the program text. You should get options to verify the file or just this function (either will work).

> If you right-click within a C source file, several VCC commands are made available, depending on what kind of construction IntelliSense thinks you are in the middle of. The choice of verifying the entire file is always available. If you click within the definition of a struct type, VCC will offer you the choice of checking admissibility for that type (a concept explained in Section 5.5). If you click within the body of a function, VCC should offer you the opportunity to verify just that function. However, IntelliSense often gets confused about the syntactic structure of VCC code, so it might not always present these context-dependent options. However, if you select the name of a function and then right click, it will allow you to verify just that function.

VCC verifies this function successfully, which means that its assertions are indeed guaranteed to hold and that the program cannot crash[7] If VCC is unable to prove an assertion, it reports an error. Try changing the assertion in this program to something that isn't true and see what happens. (You might also want to try coming up with some complex assertion that is true, just to see whether VCC is smart enough to prove it.)

[7] VCC currently doesn't check that your program doesn't run forever or run out of stack space, but future versions will, at least for sequential programs.

Assertions, like all VCC annotations, are surrounded by `_( ... )` to indicate that they are only for VCC, and are not part of the program being verified For the regular C compiler the `<vcc.h>` header file defines:

```
#define _(...) /* nothing */
```

VCC does not use this definition, and instead parses the inside of `_( ... )` annotations.[8]

To understand how VCC works, and to use it successfully, it is useful to think in terms of what VCC "knows" at various program points. In this example, VCC initially knows nothing about the variables (they can initially hold any value). [**TODO:** make sure this makes sense:] Just before the first assignment, VCC knows that `x <= y` (because execution followed that branch of the conditional), and after the assignment, VCC knows additionally that `z` is equal to `x`, so it knows that `z <= x`. Similar reasoning (with `x` and `y` reversed) holds in `else` branch, so VCC knows that the assertion holds when control reaches it. In general, VCC doesn't lose any information when reasoning about assignments and conditionals. We will see in subsequent sections, however, that VCC abstracts over (looses) some information when reasoning about loops and function calls, which means additional annotations are necessary.

When VCC surprises you by failing to verify something that you think it should be able to verify, it is usually because it doesn't know something you think it should know. An assertion provides one way to check whether VCC knows what you think it knows.

You can add to what VCC knows at a particular point by adding a second type of annotation, called an ***assumption***. The assumption `_(assume E)` tells VCC to ignore the rest of an execution if `E` fails to hold (i.e., if `E` evaluates to `0`). If you verify a program with VCC, it guarantees that in any prefix of an execution where all (user-provided) assumptions hold, all assertions will also hold. Reasoning-wise, assumption simply adds `E` to what VCC knows for subsequent reasoning. For example:

```
int x, y;
_(assume x != 0)
y = 100 / x;
```

Without the assumption, VCC would complain about possible division by zero (it checks for division by zero because it would cause the program to crash). With the assumption, this error cannot happen. Since assumptions (like all annotations) are not seen by the compiler, assumption failure won't cause the program to stop, so subsequent assertions might be violated. Thus, your goal should be to eliminate as many assumptions as possible, and preferably remove them altogether.

Although assumptions are to be avoided, they are nevertheless sometimes useful:

[8] You can prevent definition of `_`, and instead surround annotations with `__vcc_spec(...)`. This is controlled with `_VCC_DONT_USE_UNDERSCORE` macro-definition. [**TODO:** This doesn't work at the moment.]

- In an incomplete verification, assumptions can be used to mark the knowledge that VCC is missing, and guide further verification work possibly performed by different people.
- When debugging a failed verification, you can use assumptions to narrow down the failed verification to a more specific failure scenario, perhaps even all the way to a counterexample counterexample.
- Sometimes you want to assume something even though VCC can verify it, just to stop VCC from spending time proving it. For example, assuming `\false` allows VCC to easily prove subsequent assertions, effectively focussing its attention on other parts of the function. Temporarily adding assumptions is a common tactic when developing annotations for complex functions.
- Sometimes you will want to make assumptions about the operating environment of a program. For example, you might want to assume that a 64-bit counter doesn't overflow, but don't want to justify it formally because it depends on extra-logical assumptions (like the speed of the hardware or the lifetime of the software).
- Assumptions provide a useful tool in explaining how VCC reasons about your program. We'll see examples of this throughout this tutorial.

An assertion can also change what VCC knows after the assertion, if the assertion fails; even though VCC will report the failure as an error, it will assume the asserted fact holds afterward. For example, in the following VCC will only report an error for first assumption and no the second:

```
int x;
_(assert x == 0)
_(assert x == 0)
```

When we talk about what VCC knows, we mean what it knows in an ideal sense, where if it knows `E`, it also knows any logical consequence of `E`. In such a world, adding an assertion that succeeds would have no effect on whether later assertions succeed. VCC's ability to deduce consequences is indeed complete for many types of expressions (e.g. expressions that use only equalities, inequalities, addition, subtraction, multiplication by constants, and logical operators), but not for all expressions, so VCC will sometimes fail to prove an assertion, even though it "knows" enough to prove it.

An assertion that succeeds can sometimes cause later assertions that would otherwise fail to succeed, by drawing VCC's attention to a crucial fact it needs to prove the later assertion. This is relatively rare, and typically involves "nonlinear arithmetic" (e.g. where variables are multiplied together), bit-vector reasoning, or quantifiers. We'll say more about these in Section C.

Exercises

1. In the following program fragment, which assertions will fail?

   ```
   int x,y;
   _(assert x > 5)
   _(assert x > 3)
   _(assert x < 2)
   _(assert y < 3)
   ```

2. Is there any difference between

   ```
   _(assume p)
   _(assume q)
   ```

   and

   ```
   _(assume q)
   _(assume p)
   ```

   ? What if the assumptions are replaced by assertions?

3. Is there any difference between

   ```
   _(assume p)
   _(assert q)
   ```

   and

   ```
   _(assert p ==> q)
   ```

   ? [**TODO:** The implication should be introduced earlier.] `p ==> q` means `p` implies `q`, and formally `!p || q`.

3 Function specifications

Next we turn to the specification of functions. We'll take the example from the previous section, and pull the computation of the minimum of two numbers out into a separate function:

```
#include <vcc.h>

int min(int a, int b)
{
  if (a <= b)
    return a;
  else return b;
}

int main()
{
  int x, y, z;
```

```
  z = min(x, y);
  _(assert z <= x)
  return 0;
}

Verification of min succeeded.
Verification of main failed.
testcase(14,26) : error VC9500: Assertion 'z <= x' did not
    verify.
```

(The listing above presents both the source code and the output of VCC, typeset in a different font and color, and the actual file name of the example is replaced with `testcase`.) VCC failed to prove our assertion, even though it's easy to see that it always holds. This is because the verification is ***modular***: VCC doesn't look inside the body of a function (such as `min()`) when verifying a function that calls it (such as `main()`)[9]. That is, VCC doesn't know anything about the effect of calling `min()`, unless you specify it explicitly. Since the correctness of `main()` clearly depends on what `min()` does, we need to specify `min()` in order to verify `main()`.

> There are at least three good reasons to use modular verification. First, it allows verification to more easily scale to large programs. Second, it allows you to modify the implementation of a function like `min()` without having to worry about breaking the verification of functions that call it (as long as you don't change the specification of `min()`). This is especially important because these callers normally aren't in scope, and the person maintaining `min()` might not even know about them (e.g., if `min()` is in a library). Third, you can verify a function like `main()` even if the implementation of `min()` is unavailable, or perhaps hasn't even been written yet.

The specification of a function is sometimes called a ***contract***, because it gives obligations on both the function and its callers. It is based on two new kinds of annotations:

- The requirements on the caller (sometimes called the ***precondition*** of the function) take the form `_(requires E)`, where `E` is an expression. In verifying the function, VCC implicitly assumes `E` on function entry, and for each call to the function, VCC implicitly asserts `E` (after the function arguments have been evaluated and bound to the parameter names).
- The requirement on the function (sometimes called the ***postcondition*** of the function) takes the form `_(ensures E)`, where `E` is an expression; this says that the function promises that `E` holds when control is returned to the caller. In the code of the caller, VCC implicitly assumes `E` just after the call returns (but before the result is used). In the code of the function,

[9] We will see later how to selectively override this default and perform such inlining during verification.

at each return point of the function (including, implicitly, at the end of the function body), VCC asserts that the postcondition of the function holds (with the VCC keyword `\result` bound to the return value, and the function parameters interpreted according to the values they had on function entry).

For example, we can provide a suitable specification for `min()` as follows:

```
#include <vcc.h>

int min(int a, int b)
  _(requires \true)
  _(ensures \result <= a && \result <= b)
{
  if (a <= b)
    return a;
  else return b;
}
// ... definition of main() unchanged ...

Verification of min succeeded.
Verification of main succeeded.
```

The precondition `_(requires \true)` of `min()` doesn't really say anything (since `\true` holds in every state), but is included to emphasize that the function can be called from any state and with arbitrary parameter values. The postcondition states that the value returned from `min()` is no bigger than either of the inputs. Note that `\true` and `\result` are spelled with a backslash to avoid clash with C identifiers. All VCC keywords start with a backslash. This contrasts with annotation tags (like `requires`), which are only used at the beginning of annotation and therefore cannot be confused with C identifiers (and thus you are still free to have, e.g., a function called `requires` or `assert`).

As described above, VCC translates the program roughly as follows:

```
#include <vcc.h>

int min(int a, int b)
{
  int res;
  _(assume \true)
  if (a <= b)
    res = a;
  else res = b;
  _(assert res <= a && res <= b)
}

int main()
{
  int x, y, z;
```

```
  _(assert \true)
  z = min(x, y);
  _(assume z <= x && z <= y)
  _(assert z <= x)
  return 0;
}
```

```
Verification of min succeeded.
Verification of main succeeded.
```

> The assumptions introduced implicitly by VCC at function entry and after the function return are justified by the assertions introduced before the function call and before the function return, respectively. This is why they do not compromise soundness of the verifier; i.e., if the entire program verifies, then these assumptions will never fail at runtime.

As for call itself, after the desugaring VCC forgets about specification of `min()`, so VCC doesn't know anything about the value the function might possibly return. Therefore, as far as VCC's understanding is concerned, the line:

```
position = min(newPos, LIMIT);
```

could be replaced with:[**TODO:** Should we make `havoc` part of the language?]

```
position = anything();
```

that is assigning an unspecified value to the variable `position`. The only knowledge about it comes from the assumption that follows the assignment.

Exercises

1. Try replacing the `<` in the return statement of `min()` with `>`. Before running the verifier, can you guess which parts of the verification will fail?

2. What is the effect of giving a function the specification `_(requires \false)` ? How does it effect verification of the function itself? What about its callers? Can you think of a good reason to use such a specification?

3. Can you see anything wrong with the above specification of `min()`? Can you give a simpler implementation than the one presented? Is this specification strong enough to be useful? If not, how might it be strengthened to make it more useful?

4. Specify a function that returns the (int) square root of its (int) argument. (You can try writing an implementation for the function, but won't be able to verify it until you've learned about loop invariants.)

5. Can you think of useful functions in a real program that might legitimately guarantee only the trivial postcondition `_(ensures \true)`?

3.1 Reading and Writing

So far, we have considered functions that access only parameters and their own local variables. In an imperative language (particularly one like C that provides only call-by-value), most interesting functions have to read and/or write memory shared with its caller. A function can read a memory object only if the object is known to be valid and not concurrently modified by other threads; such an object is said to be ***thread-local***.

Writing requires something more; by default, VCC assumes that a function call has no side effects on its caller; for example, in verifying `main()` above, VCC got to assume that the call of `min()` didn't change `newPos`. In VCC, possible side effects are specified using another form of annotation, called ***writes clauses***. Intuitively, the clause `_(writes p, q)` says that, of the memory objects that are thread-local to the caller before the call, the function is going to modify only the object pointed to by `p` and the object pointed to by `q`. In other words, it is roughly equivalent to a postcondition that ensures that all other objects local to the caller prior to the call remain unchanged.[10] VCC allows you to specify pointers in separate writes clauses, and implicitly combines them into a single set. If a function spec contains no writes clauses, it is equivalent to specifying a writes clause with empty set of pointers.

At any control point, VCC can "know" that a particular memory object `*p` is writable; in annotations, this is written as `\writeable(p)`, and is defined as a conjunction of two conditions:

- `p` belongs to the current thread and is in a "phase of life" where it can be written; in VCC, this is written as `\unwrapped(p)`. (Note that this also implies `\thread_local(p)`.)
- Either `p` was listed in the writes clause of the function, or it was `\unwrapped` after the function was entered; the latter condition guarantees that either `p` was listed in the writes clause or was not thread-local in the caller.

In particular, formal function parameters and local variables are writable as long as they are in scope and have not been explicitly wrapped (Section 5) or reinterpreted to a different type (Section B.1). VCC asserts `\writable(p)` on each attempt to write to `*p`, as well as on each call to a function that lists `p` in a writes clause.

Here is a simple example of a function that visibly reads and writes memory; it simply copies data from one location to another.

```
#include <vcc.h>

void copy(int *from, int *to)
```

[10] It's not quite the same when considering objects that are wrapped; see Section 5.

```
  _(requires \thread_local(from))
  _(writes to)
  _(ensures *to == \old(*from))
{
  *to = *from;
}

int z;

void main()
_(writes &z)
{
  int x,y;
  copy(&x,&y);
  copy(&y,&z);
  _(assert x==y && y==z)
}
```

```
Verification of copy succeeded.
Verification of main succeeded.
```

In the postcondition the syntax `\old(E)` returns value of expression `E` as if it was evaluated before the function started, in the so-called ***pre-state*** of the call. Thus, our postcondition states that the new value of `*to` equals the value of `*from` before the call. In fact, the postcondition `*to == *from` is equivalent, because the function does not write `*from`.

Whenever a memory object is read, VCC asserts that it is thread-local; whenever a memory object is written or is mentioned in the writes clause of a function call, VCC asserts that it is writable. Try removing each of the function annotations in this example (one at a time) to see what error messages you get.

While thread locality is what is checked on a read, in practice it is rarely used in preconditions, because it could be falsified by the unwrapping of other objects (Section 5). More usual is to provide the stronger precondition of the object being `\unwrapped`; you should use this stronger condition in specifications.

Exercises

1. Specify, write, and verify a function that swaps the values pointed to by two `int` pointers. Hint: use `\old(...)` in the postcondition.

4 Loop Invariants

For the most part, what VCC knows at a control point can be computed from what it knew at the immediately preceding control points. But when the control flow contains a loop, VCC faces a chicken-egg problem, since what it knows at

the top of the loop (i.e., at the beginning of each loop iteration) depends not only on what it knew just before the loop, but also on what it knew just before it jumped back to the top of the loop from the loop body.

Rather than trying to guess what it should know at the top of a loop, VCC lets you tell it what it should know, by providing a ***loop invariant***. To make sure that the loop invariant does indeed hold whenever control reaches the top of the loop, VCC asserts that the invariant holds wherever control jumps to the top of the loop – namely, on loop entry, at the end of the loop body, and at `continue` statements within the loop body. In addition, VCC knows at the top of a loop that any variable that is not modified in the loop has the same value it had on entry to the loop[11]. We simulate this by having VCC forget (at loop entry) everything it knew about variables that are modified in the loop body.

Let's look at an example:

```
#include <vcc.h>

unsigned mod(unsigned a, unsigned b)
  _(ensures \result == a % b)
{
  unsigned res = a;

  for (;;)
    _(invariant a % b == res % b)
  {
    if (res < b) break;
    res -= b;
  }
  return res;
}
```

```
Verification of mod succeeded.
```

[**TODO:** This example currently requires axioms about arithmetic that should actually come by default with VCC. This will be fixed soon.] The `mod()` function computes a reminder of a integer division using iterated subtraction[12] The loop invariant says that the `res % a` stays unchanged, and equal to `a % b`. VCC translates this example roughly as follows:

```
unsigned mod(unsigned a, unsigned b)
{
  unsigned res = a;

  // check that invariant initially holds
  _(assert a % b == res % b)
```

[11] Because of aliasing, it is not always obvious to VCC that a variable is not modified in the body of the loop. However, VCC can check it syntactically for a local variable if you never take the address of that variable.

[12] Of course C has a built-in operator to calculate this, but you can imagine that you are verifying code designed to execute on a machine that doesn't have the operator built-in.

```
  // start an arbitrary loop iteration
  res = anything();
  _(assume a % b == res % b) // assume the invariant
  if (res < b) goto theEnd;
  res -= b;
  _(assert a % b == res % b) // check the invariant
  _(assume \false) // end of an iteration

theEnd:
  ;
  _(assert res == a % b) // translation of ensures
  return res;
}
```

```
Verification of mod succeeded.
```

The invariant is asserted wherever control moves to the top of the loop (here, on entry to the loop and at the end of the loop body). On loop entry, VCC forgets the value of each variable modified in the loop (in this case just `res`), and assumes the invariant (which places some constraints on `res`). VCC doesn't have to consider the actual jump from the end of the loop iteration back to the top of the loop (since it has already checked the loop invariant), so further consideration of that branch is cut off with `_(assume \false)`. The only way to exit this particular loop is through the `break` statement, (which is translated to a `goto`). At this control point, we know the loop invariant holds and that `res < b`, which together imply the postcondition of the function. Since this is the only location from which control jumps to `theEnd`, the postcondition is also known at `theEnd`, so the assertion generated by the postcondition follows.

If the loop had been of the form

```
while (res >= b)
  _( invariant a % b == res % b )
{
  res -= b;
}
```

the assert/assume translation would be exactly the same: the invariant would be assumed before checking the guard, despite the fact that guard syntactically precedes the invariant.

For a more interesting example of a loop, consider the following function that uses linear search to determine if a value occurs within an array: [**TODO:** Change the following to use `thread_local` instead of `wrapped`.]

```
#include <vcc.h>
#include <limits.h>

unsigned lsearch(int elt, int *ar, unsigned sz)
  _(requires \wrapped((int[sz])(ar)))
```

```
  _(ensures \result != UINT_MAX ==> ar[\result] == elt)
{
  unsigned i;
  for (i = 0; i < sz; i++)
  {
    if (ar[i] == elt) return i;
  }

  return UINT_MAX;
}
```

```
Verification of lsearch succeeded.
```

Thus, the postcondition guarantees that when the returned value is not `UINT_MAX`, then `elt` is indeed found at the returned index. This doesn't require an invariant, because the non-`UINT_MAX` index is returned only under that condition. However this specification is not full: what about the case when the result is not found? To express that an element is not in an array we will use a universal quantifier:

```
#include <vcc.h>
#include <limits.h>

unsigned lsearch(int elt, int *ar, unsigned sz)
  _(requires \wrapped((int[sz])(ar)))
  _(ensures \result != UINT_MAX ==> ar[\result] == elt)
  _(ensures \result == UINT_MAX ==>
            \forall unsigned i; i < sz ==> ar[i] != elt)
{
  unsigned i;
  for (i = 0; i < sz; i++)
    _(invariant \forall unsigned j; j < i ==> ar[j] != elt)
  {
    if (ar[i] == elt) return i;
  }

  return UINT_MAX;
}
```

```
Verification of lsearch succeeded.
```

The invariant `\forall unsigned j; j < i ==> ar[j] != elt` means that `ar[j]` is not `elt` for all `j` between `0` and `i - 1`. In other words, take an arbitrary unsigned (and thus non-negative) integer `j`. If it happens to be less than `i` then the value in `ar[j]` is not `elt`. Thus, the invariant says that the element was not found between `0` and `i - 1`. If the loop terminates with `i == sz`, then the postcondition follows.

4.1 Review

[**TODO:** Put in some way to get to the relevant info about quantifiers and ghost data?]

[**TODO:** I'm not sure how useful this review is. In particular the parts where we explain the semantics of if statements and assignments in terms of what VCC know just seem confusing. Programmers already know what if statement or assignment does. I think it would be useful just to emphasize what VCC doesn't know, for loops and function calls. –M]

You have now learned enough to verify some nontrivial sequential programs that use only base types and arrays. This is already a very rich domain for programming, and you should take some time using VCC to verify some of the "toy" algorithms you learned in school. It's also a good opportunity to review what we've learned so far.

At each control point within a function, VCC "knows" certain things about the state of the program. Included in this knowledge is what memory locations it can safely read or write. The computation of this knowledge can be summarized as follows:

- On entry to a function, it knows the preconditions of the function.
- A memory object is writable if it is unwrapped and is either listed in the writes clause of the function or was unwrapped after the function was entered.
- After `_(assume E)`, it knows what it knew before the assumption, and in addition knows `E != 0`.
- `_(assert E)` asks VCC to prove that what it knows before the assertion implies `E` (and report an error if it can't). It also assumes `E` afterward.
- [**TODO:** Break this up into variable assignment and memory assignment?] An assignment statement `v = E`, where `E` doesn't have a function call and doesn't mention `v`[13], asserts that the data needed to compute `E` is readable, and that `v` is writeable. After the assignment, it knows everything it knew before the assignment (except for what it knew about the value of `v`), and additionally knows that `v == E`.
- A function call `f(args)`, where `args` is a list of variables[14], asserts that the `args` are readable, asserts that the objects mentioned in the writes clauses of `f` are writeable, asserts the preconditions of `f` (with the actual parameters substituted for the formal parameters), forgets what it knew

[13] If `E` mentions `v`, we can imagine the value of `v` being first copied into a fresh temporary variable, which is used in place of `v` within `E`.

[14] If the arguments to the function call are expressions, we can think of these expressions being evaluated and assigned to temporary variables before the function call.

about `v` and any objects mentioned in the writes clause of `f`, and finally assumes the postconditions of `f`[15]

- For a conditional `if (p)S1 else S2`, it first asserts that `p` is readable. At the beginning of `S1` (resp. `S2`), it knows what it knew before the conditional, and in addition knows `p != 0` (resp. `p == 0`). After the conditional, it knows the disjunction ("or") of what it knew at the end of `S1` and what we know at the end of `S2`.
- For a loop, it knows at the beginning of the loop body just what it knew just before the loop (except that it forgets what it knew about variables modified in the loop), and also knows the loop invariant. Just before the loop, and at any point in the loop body where control jumps back to the top of the loop (including the end of the loop body), it asserts the loop invariant.

4.2 Sorting

The function below implements the bozo-sort algorithm. The algorithm works by swapping two random elements in an array, checking if the resulting array is sorted, and repeating otherwise. We do not recommend using it in production code: it's not stable, and moreover has a fairly bad time complexity. Still, it will serve us to illustrate the use of ghost functions, and later data.

```
#include <vcc.h>
#include <stdlib.h>

_(logic bool sorted(int *buf, unsigned len) =
  \forall unsigned i, j; i < j && j < len ==> buf[i] <= buf[j])

void bozo_sort(int *buf, unsigned len)
  _(writes \array_range(buf, len))
  _(ensures sorted(buf, len))
{
  if (len == 0) return;

  for (;;)
    _(invariant \mutable_array(buf, len))
  {
    int tmp;
    unsigned i = _(unchecked)((unsigned)rand()) % len;
    unsigned j = _(unchecked)((unsigned)rand()) % len;

    tmp = buf[i];
    buf[i] = buf[j];
    buf[j] = tmp;
```

[15] If these postconditions use `\old` to refer to parts of the state before the call, we can think of these parts of the state as copied to temporary variables prior to the call. The result of the function can be viewed as being put into a temporary variable of the caller.

```
    for (i = 0; i < len - 1; ++i)
      _(invariant sorted(buf, i + 1))
    {
      if (buf[i] > buf[i + 1]) break;
    }

    if (i == len - 1) break;
  }
}
```

```
Verification of bozo_sort succeeded.
```

The meaning of `_(logic F(A)= E)` is similar to `#define F(A)E`, except that it prevents name capture and gives better error messages (i.e., ones which mention both `E` and `F(A)`).

The specification that we use is that the output of the sorting routine is sorted. Unfortunately, we do not say that it's actually a permutation of the input. We'll show how to do that in Section **??**.

We use the `rand()` function declared in `<stdlib.h>`. VCC currently does not come with specifications for C standard library functions (there [**TODO:** will be] is an ongoing project, where you can actually contribute specifications!). According to the standard library documentation the specification of `rand()` should be:

```
int rand(void)
  _(ensures 0 <= \result && \result <= RAND_MAX);
```

It is unclear why does it return a signed integer, only to ensure that the return value is never negative. Anyway, we do not actually use this postcondition in our example. In fact we explicitly cast the value returned `rand()` to `unsigned`. VCC does not know the value cannot be negative, and therefore would normally complain, that such a cast might underflow.

[**TODO:** The following paragraph should have it's own section.] To indicate that this underflow behavior is desired we use `_(unchecked)`. This annotation applies to the following expression, and indicates that you expect that there might be overflows in there. For example consider:

```
int a, b;
// ...
a = b + 1;
_(assert a < b)
```

This will either complain about possible overflow of `b + 1` or succeed. However, the following might complain about `a < b`, if VCC does not know that `b + 1` doesn't overflow.

```
int a, b;
// ...
```

```
a = _(unchecked)(b + 1);
_(assert a < b)
```

Exercises

1. Write and verify an iterative program that copies an array if ints from one location to another.

2. Write and verify an iterative program for binary search (a program that checks whether a sorted array contains a given value).

3. Write and verify an iterative program that sorts an array of ints using bubblesort. The specification should be the same as for bozo-sort above.

5 Object invariants

Pre- and postconditions allow for associating consistency conditions with the code. However, fairly often it is also possible to associate such consistency conditions with the data and require all the code operating on such data to obey the conditions. As we will learn in Section 7 this is particularly important for data accessed concurrently by multiple threads, but even for sequential programs enforcing consistency conditions on data reduces annotation clutter and allows for introduction of abstraction boundaries.

In VCC, the mechanisms for enforcing data consistency is ***object invariants***, which are conditions associated with compound C types (`structs` and `unions`). The invariants of a type describe how "proper" objects of of that type behave. In this, and the following, section, we consider only the static aspects of this behavior, namely what the "consistent" states of an object are. Dynamic aspects, i.e., how objects can change, are covered in Section 7. For example, consider the following type definition of `'\0'`-terminated safe strings implemented with statically allocated arrays (we'll see dynamic allocation later).

```
#define SSTR_MAXLEN 100
typedef struct SafeString {
  unsigned len;
  char content[SSTR_MAXLEN + 1];
  _(invariant \this->len <= SSTR_MAXLEN)
  _(invariant content[len] == '\0')
} SafeString;
```

The invariant of `SafeString` states that consistent `SafeStrings` have length not more than `SSTR_MAXLEN` and are `'\0'`-terminated. Within a type invariant, `\this` refers to (the address of) the current instance of the type (as in the first invariant), but fields can also be referred to directly (as in the second invariant).

Because memory in C is allocated without initialization, no nontrivial object invariant could be enforced to hold at all times. Thus, in VCC the invariants are only guaranteed to hold for ***consistent*** objects. Objects, which are consistent

and owned by the current thread are called ***wrapped***. After allocating an object we would usually wrap it to make sure its invariant holds:

```
void sstr_init(struct SafeString *s)
  _(writes \span(s))
  _(ensures \wrapped(s))
{
  s->len = 0;
  s->content[0] = '\0';
  _(wrap s)
}
```

[**TODO:** Should talk about the array rules somewhere.] For a pointer `p` of structured type, `\span(p)` returns the set of pointers to members of `p`. Arrays of base types produce one pointer for each base type component, so in this example, `\span(s)` abbreviates the set

```
{ s, &s->len, &s->content[0], &s->content[1], ...,
    &s->content[SSTR_MAXLEN] }
```

Thus, the writes clause says that the function can write the fields of `s`. The postcondition says that the function returns with `s` wrapped, which implies also that the invariant of `s` holds; this invariant is checked when the object is wrapped.

A function that modifies a wrapped object will first unwrap it, make the necessary updates, and wrap the object again (which causes another check of the object invariant). Unwrapping an object adds all of its members to the writes set of a function, so such a function has to report that it writes the object, but does not have to report writing the fields of the object.

Finally, a function that only reads an object need not unwrap, and so will not list it in its writes clause. For example:

```
void sstr_append_char(struct SafeString *s, char c)
  _(requires \wrapped(s))
  _(requires s->len < SSTR_MAXLEN)
  _(ensures \wrapped(s))
  _(writes s)
{
  _(unwrap s)
  s->content[s->len++] = c;
  s->content[s->len] = '\0';
  _(wrap s)
}

int sstr_index_of(struct SafeString *s, char c)
  _(requires \wrapped(s))
  _(ensures \result >= 0 ==> s->content[\result] == c)
{
  unsigned i;
  for (i = 0; i < s->len; ++i)
    if (s->content[i] == c) return (int)i;
```

```
  return -1;
}
```

The following subsection explains this wrap/unwrap protocol in more details.

5.1 Wrap/unwrap protocol

Because invariants do not always hold, in VCC one needs to explicitly state which objects are consistent, using a ghost field `\consistent` which is defined on every object. The invariants need to hold only when the `\consistent` field is true. When the field is false they need not hold (but of course still can). The field is initially (on newly allocated objects) false, and you need to set it to false before disposing objects.

In addition to the `\consistent` field each object has an ***owner field***. The owner of `p` is `p->\owner`. This field is of pointer (object) type, but VCC provides objects, of `\thread` type, to represent threads of execution, so that threads can also own objects. The idea is that the owner of `p` should have some special rights to `p` that others do not. In particular, the owner of `p` can transfer ownership of `p` to another object (e.g., a thread can transfer ownership of `p` from itself to the memory allocator, in order to dispose of `p`).

When verifying a body of a function VCC assumes that it is being executed by some particular thread. The `\thread` object representing it is referred to as `\me`.

(Some of) the rules of ownership and consistency:

1. on every atomic step of the program the invariants of all the consistent objects have to hold
2. only the owning thread can modify fields of an inconsistent object
3. threads own themselves
4. only threads can own inconsistent objects

Thus, by the first two rules, VCC allows updates of objects in the following two situations:

1. the updated object is consistent, the update is atomic, and the update preserves the invariant of the object,
2. or the updated object is inconsistent and the update is performed by the owning thread.

In the first case to ensure that an update is atomic, VCC requires that the updated field has a `volatile` modifier. There is a lot to be said about atomic updates in VCC, and we shall do that in Section 7, but for now we're only considering sequentially accessed objects, with no `volatile` modifiers on fields. For such objects we can assume that they *do not change* when they are consistent, so the only way to change their fields is to first make them inconsistent, i.e., via method 2 above.

So a thread needs to make the object inconsistent to update it. Because making it inconsistent counts as an update, the thread needs to own it first. This is performed by the unwrap operation, which translates to the following steps:

1. assert that the object is in the writes set
2. assert that the object wrapped (consistent and owned by the current thread),
3. assume the invariant (as a consequence of rule 1, the invariant holds for every consistent object),
4. set the `\consistent` field to false.
5. add the span of the object (i.e., all its fields) to the writes set

The wrap operation does just the reverse:

1. assert that the object is unwrapped (inconsistent and owned by the current thread; note that this does not mean "not wrapped"),
2. assert the invariant,
3. set the `\consistent` field to true (this implicitly prevents further writes to the fields of the object)

Let's then have a look at the definitions of `\wrapped(...)` and `\unwrapped(...)` and at the assert/assume desugaring of `sstr_append_char()` function.

```
logic bool \wrapped(\object o) =
  o->\consistent && o->\owner == \me;
logic bool \unwrapped(\object o) =
  !o->\consistent && o->\owner == \me;

void sstr_append_char(struct SafeString *s, char c)
  _(requires \wrapped(s))
  _(requires s->len < SSTR_MAXLEN)
  _(ensures \wrapped(s))
{
  // _(unwrap s), steps 1-5
  _(assert \writable(s))
  _(assert \wrapped(s))
  _(assume s->len <= SSTR_MAXLEN &&
           s->content[s->len] == '\0')
  _(ghost s->\consistent = \false;)
  _(assume \writable(\span(s)))

  s->content[s->len++] = c;
  s->content[s->len] = '\0';

  // _(wrap s), steps 1-3
```

```
  _(assert \mutable(s))
  _(assert s->len <= SSTR_MAXLEN &&
           s->content[s->len] == '\0')
  _(ghost s->\consistent = \true;)
}
```

The definitions of `\wrapped(...)` and `\unwrapped(...)` use the `\object` type. It is much like `void*`, in the sense that it is a wildcard for any pointer type. However, unlike `void*`, it also carries the dynamic information about the type of the pointer. It can be only used in specifications.

5.2 Ownership trees

Objects often stand for abstractions that are implemented with more than just one physical object. As a simple example, consider our `SafeString`, changed to have dynamically allocated buffer. The logical string object consists of the control object holding the length and the array of bytes holding the content. In real programs such abstraction become hierarchical, e.g., an address book might consists of a few hash tables, each of which consists of a control object, an array of buckets, and the attached linked lists.

```
struct SafeString {
  unsigned capacity, len;
  char *content;
  _(invariant len < capacity)
  _(invariant content[len] == '\0')
  _(invariant \mine((char[capacity])content))
};
```

In C the type `char[10]` denotes an array with exactly 10 elements. VCC extends that location to allow the type `char[capacity]` denoting an array with `capacity` elements (where `capacity` is a variable). Such types can be only used in casts, as in `(char[capacity])content`which means: take the pointer `content` and interpret it as an array of `capacity` elements of type `char`. This notation is used so we can think of arrays as objects (of some weird type). The other way to think about it is that `content` represents just one object of type `char`, whereas `(char[capacity])content` is an object representing the array.

The invariant of `SafeString` specifies that it ***owns*** the array object. The syntax `\mine(o1, ..., oN)` is roughly equivalent (we'll get into details later) to:

```
o1->\owner == \this && ... && oN->\owner == \this
```

Conceptually there isn't much difference between having the `char` array embedded and owning a pointer to it. In particular, the functions operating on some `s` of type `SafeString` should still list only `s` in their writes clauses, and not also `(char[s->capacity])s->content`, or any other objects the string might comprise of. To achieve that VCC performs ***ownership transfers***, i.e., assignments to the `\owner` field, when unwrapping and wrapping objects.

- When object is unwrapped, in addition to five steps described in previous section, the `\owner` field of all objects owned by unwrapped object is set to `\me`. Additionally, these objects are added to the writes set.
- When object `p` is wrapped, in addition to previously described steps, for every object `q` that needs to be owned by `p` VCC asserts that `q` is wrapped and writable, and sets `q->\owner` to `p`. What `p` needs to own is is determined by `p`'s invariant, as you'll see in the next section.

Let's then have a look at an example:

```
void sstr_append_char(struct SafeString *s, char c)
  _(requires \wrapped(s))
  _(requires s->len < s->capacity - 1)
  _(ensures \wrapped(s))
  _(writes s)
{
  _(unwrapping s) {
    _(unwrapping (char[s->capacity])(s->content)) {
      s->content[s->len++] = c;
      s->content[s->len] = '\0';
    }
  }
}
```

First, let's explain the syntax:

```
_(unwrapping o) { ... }
```

is equivalent to:

```
_(unwrap o) { ... } _(wrap o)
```

Thus, at the beginning of the function the string is owned by the current thread and consistent (i.e., wrapped), whereas, by the string invariant, the content is owned by the string and consistent. After unwrapping the string, the ownership of the content goes to the current thread, but the content remains consistent. Thus, unwrapping the string makes the string unwrapped, and the content wrapped. Then we unwrap the content (which doesn't own anything, so the thread gets no new wrapped objects), perform the changes, and wrap the content. Finally, we wrap the string. This transfers ownership of the content from the current thread to the string, so the content is no longer wrapped (but still consistent). Let's then have a look at the assert/assume translation:

```
void sstr_append_char(struct SafeString *s, char c)
  _(requires \wrapped(s))
  _(requires s->len < s->capacity - 1)
  _(ensures \wrapped(s))
  _(writes s)
{
  _(ghost \object cont = (char[s->capacity]) s->content; )
  // _(unwrap s) steps 1-5
```

```
  _(assert \writable(s) && \wrapped(s))
  _(assume \writable(\span(s)) && \inv(s))
  _(ghost s->\consistent = \false; )
  // and the transfer:
  _(ghost cont->\owner = \me; )
  _(assume \writable(cont))
  // _(unwrap cont) steps 1-5
  _(assert \writable(cont) && \wrapped(cont))
  _(ghost cont->\consistent = \false; )
  _(assume \writable(\span(cont)) && \inv(cont))
  // no transfer here

  s->content[s->len++] = c;
  s->content[s->len] = '\0';

  // _(wrap cont) steps 1-3
  _(assert \mutable(cont) && \inv(cont))
  _(ghost cont->\consistent = \true; )
  // _(wrap s) steps 1-3, with transfer in the middle
  _(assert \mutable(s))
  _(ghost cont->\owner = s; )
  _(assert \inv(s))
  _(ghost s->\consistent = \true; )
}
```

To make it easier to read, we made it store the `s->content` pointer casted to an array into a temporary variable. Also, an invariant of `p` is referred to as `\inv(p)`. As you can see there are two ownership transfers of `cont` to and from `\me`. This happens because `s` owns `cont` beforehand, as specified in its invariant. However, let's say we had an invariant like:

```
struct S {
  struct T *a, *b;
  _(invariant \mine(a) || \mine(b))
};
```

When wrapping an instance of `struct S`, should we transfer ownership of `a`, `b`, or both? By default VCC will reject such invariants, and only allow `\mine(...)` as a top-level conjunct in an invariant. Invariants like the ones above are supported, but need additional annotation and manual ownership transfer when wrapping, see Section 5.3.

5.3 Dynamic ownership

When a struct is annotated with `_(dynamic_owns)` the ownership transfers during wrapping need to performed explicitly, but `\mine(...)` can be freely used in its invariant, including using it under a universal quantifier.

```
_(dynamic_owns) struct SafeContainer {
  struct SafeString **strings;
```

```
  unsigned len;

  _(invariant \mine((struct SafeString *[len])strings))
  _(invariant \forall unsigned i; i < len ==>
      \mine(strings[i]))
  _(invariant \forall unsigned i, j; i < len && j < len ==>
      i != j ==> strings[i] != strings[j])
};
```

The invariant of `struct SafeContainer` states that it owns its underlying array, as well as all elements pointed to from it. It also states that there are no duplicates in that array. Let's now say we want to change a pointer in that array, from `x` to `y`. After such operation, the container should own whatever it used to own minus `x` plus `y`. To facilitate such transfers VCC introduces the ***owns set***. It is essentially the inverse of the owner field. It is defined on every object `p` and referred to as `p->\owns`. VCC maintains that:

```
\forall \object p, q; p->\consistent ==>
  (q \in p->\owns <==> p->\owner == q)
```

The operator `<==>` reads "if and only if", and is simply boolean equality (or implication both ways), with a binding priority lower than implication. That is, for consistent `p`, the set `p->\owns` contains exactly the objects owned by `p`. Additionally, the unwrap operation does not touch the owns set, that is after unwrapping `p`, the `p->\owns` still contains all that objects that `p` used to own. Finally, the wrap operation will attempt to transfer ownership of everything in the owns set to the object being wrapped. This requires that the current thread has write access to these objects and that they are wrapped.

Thus, the usual pattern is to unwrap the object, potentially modify the owns set, and wrap the object. Note that when no ownership transfers are needed, one can just unwrap and wrap the object, without worrying about ownership. Let's have a look at an example, which does perform an ownership transfer:

```
void sc_set(struct SafeContainer *c,
            struct SafeString *s, unsigned idx)
  _(requires \wrapped(c) && \wrapped(s))
  _(requires idx < c->len)
  _(ensures \wrapped(c))
  _(ensures c->strings[idx] == s)
  _(ensures \wrapped(\old(c->strings[idx])))
  _(ensures \fresh(\old(c->strings[idx])))
  _(ensures c->len == \old(c->len))
  _(writes c, s)
{
  _(unwrapping c) {
    _(unwrapping (struct SafeString *[c->len])(c->strings)) {
      c->strings[idx] = s;
    }
    _(ghost {
      c->\owns -= \old(c->strings[idx]);
```

```
      c->\owns += s;
    })
  }
}
```

The `sc_set()` function transfers ownership of `s` to `c`, and additionally leaves object initially pointed to by `s->strings[idx]` wrapped, i.e., owned by the current thread. Moreover, it promises that this object is ***fresh***, i.e., the thread did not own it directly before. This can be used in the call site:

```
void use_case(struct SafeContainer *c, struct SafeString *s)
  _(requires \wrapped(c) && \wrapped(s))
  _(requires c->len > 10)
  _(writes c, s)
{
  struct SafeString *o;
  o = c->strings[5];
  _(assert \wrapped(c)) // OK
  _(assert \wrapped(s)) // OK
  _(assert \wrapped(o)) // error
  sc_set(c, s, 5);
  _(assert o != s) // OK
  _(assert \wrapped(c)) // OK
  _(assert \wrapped(s)) // error
  _(assert \wrapped(o)) // OK
}
```

In the contract of `sc_add` the string `s` is mentioned in the writes clause, but in the postcondition we do not say it's wrapped. Thus, asserting `\wrapped(s)` after the call fails. On the other hand, asserting `\wrapped(o)` fails before the call, but succeeds afterwards. Additionally, `\wrapped(c)` holds before and after as expected.

How is the write set updated?
Before allowing a write to `*p` VCC will assert `\unwrapped(p)`. Additionally, it will assert that either `p` is in the writes clause, or the consistency or ownership of `p` was updated after the current function started executing. Thus, after you unwrap an object, you modify consistency of all its fields, which provides the write access to them. Also, you modify ownership of all the objects that it used to own, providing write access to unwrap these objects. In case a write clause is specified on a loop, think of an implicit function definition around the loop.

5.4 Ownership domains

An ***ownership domain*** of an object `p` is the set of objects that it owns and their ownership domains, plus `p` itself. In other words, it's the set of objects that are transitively owned by a given object.

[**TODO:** this remark might be confusing, and possibly no longer true] In

general there can be cycles in the ownership graph, and so the definition above should be understood in the least fix point sense. However, every object has exactly one owner and threads own themselves, and thus anything that is owned by a thread will have a ownership domain that is a tree. It is most useful to think about ownership as trees, and disregard the degenerate cycle case.

Let's then take a look at an ownership graph: we will have a bunch of threads as roots. In particular, the current thread will own a number of objects, some of them unwrapped (inconsistent and thus not owning anything), but other wrapped (consistent and so with possibly large ownership trees (domains) hanging off them). The ownership domains of the wrapped objects are disjoint (in the grand ownership tree of the thread they are all at the same level).

Mentioning a wrapped object in the writes clause gives the function a right to unwrap it, and then unwrap everything it owns. Thus, it effectively gives write access to its entire ownership domain. [**TODO:** make it a real example that does something useful] Consider the following piece of code:

```
void f(T *p)
  _(writes p) { ... }
...
T *p, *q, *r;
_(assert \wrapped(p) && \wrapped(q) && p != q);
_(assert q \in \domain(q))
_(assert r \in \domain(q))
_(assert q->f == 1 && r->f == 2);
f(p);
_(assert q->f == 1 && r->f == 2);
```

The function `\domain(o)` returns the ownership domain of `o`. We have three objects, two of them are wrapped. We call a function that will update one of them. We now want to know if the values of the other two are preserved. Clearly, because `p != q` and both are wrapped, then `q` is not in the ownership domain of `p`, so value of `q->f` should be preserved by the call. The value of `r->f` will be preserved unless `r \in \domain(p)`, because `f(p)` could have written everything in `\domain(p)` (according to its writes clause). Unfortunately, the underlying logic used by VCC is not strong enough to show this directly (technically: the transitive closure of a relation, ownership in this case, is not expressible in first-order logic). However, VCC knows that there is no way to change anything in an ownership domain without writing its root. This is because we always enforce that ownership domains are disjoint. For example, the only way for `f(p)` to write something in `\domain(q)` would be to list `q` in `f()`'s writes clause, which is not the case. Thus, if VCC knows that `r \in \domain(q)`, and it knows that `f(p)` couldn't have written `q`, then it also knows that `r->f` is unchanged. Unfortunately, we need to explicitly tell VCC in which ownership domain `r` is to make use of that. This is what the assertion `r \in \domain(q)` is doing. We currently also need to assert `q \in \domain(q)` to help VCC with reasoning. This is because it treats the fields of `q` similarly to objects owned by `q`. We plan to fix that in future.

5.5 Simple sequential admissibility

Until now we've been skimming on the issue of what you can actually mention in an invariant. Intuitively the invariant of an object should talk about consistent states of that very object, not some other objects. However, for example the invariant of the `struct SafeString` talks about the values stored in its underlying array. This also seems natural: one should be able to mention things from the ownership domain of `p` in `p`'s invariant.

This is important, because VCC checks only invariants of objects that you actually modify, and as we recall the most important verification property we want to enforce (and which we rely on in our verifications) is that all invariant of all (consistent) objects are always preserved (Section 5.1). For example, consider:

```
struct A { int x; };
struct A *a; // global variable
struct B {
  int y;
  _(invariant a->x == y)
};
void foo()
  _(requires \wrapped(a))
  _(writes a)
{
  _(unwrapping a) { a->x = 7; }
}
```

In `foo()`, when wrapping `a` we would only check invariant of `a`, not all `struct Bs` that could possibly depend on it. Thus, an action which preserves invariant of modified object breaks invariant of another object. For this reason VCC makes the invariant of `struct B` inadmissible. In fact, for all invariants VCC will check that they are admissible. Admissibility of type `T` is checked by verifying VCC-generated function called `T#adm`. You can see messages about them when you verify files with type invariants.

We shall refrain now from giving a full definition of admissibility, as it only makes full sense after we learn about two-state object invariants (see Section 7.3), but for sequential programs the useful approximation is that invariants that only talk about their ownership domains are admissible.

5.6 Type Safety

Throughout this section we have been talking about typed memory "objects" as if this were a meaningful concept. This typed view of memory is supported by most modern programming languages, like Java, C#, and ML, where memory consists of a collection of typed objects. Programs in these languages don't allocate memory (on the stack or on the heap), they allocate objects, and the type of an object remains fixed until the object is destroyed. Moreover, a non-null reference to an object is guaranteed to point to a "valid" object. But in C,

a type simply provides a way to interpret a sequence of bytes; nothing prevents a program from having multiple pointers of different types pointing into the same memory, or even having two instances of the same `struct` type partially overlapping. Moreover, a non-null pointer might still point into an invalid region of memory.

That said, most C functions really do access memory using a strict type discipline and tacitly assume that their callers do so also. For example, if the parameters of a function are a pointer to an `int` and a pointer to a `char`, we shouldn't have to worry about crazy possibilities like the `char` aliasing with the second half of the `int`. (Without such assumptions, we would have to provide explicit preconditions to this effect.) On the other hand, if the second parameter is a pointer to an `int`, we do consider the possibility of aliasing (as we would in a strongly typed language). Moreover, since in C objects of structured types literally contain objects of other types, if the second argument were a struct that had a member of type `int`, we would have to consider the possibility of the first parameter aliasing that member.

To support this, VCC essentially maintains a typed view of memory; in any state, `p->\valid` means that `p` points to memory that is currently "has" type `p`. The rules governing validity guarantee that in any state, the valid pointers constitute a typesafe view of memory. In particular, if two valid pointers point to overlapping portions of memory, one of them is properly contained in the other; if a `struct` is typed, then each of its members is typed; and if a `union` is typed, then exactly one of its members is typed. The definition of validity is folded into the definitions of `\thread_local` and `\unwrapped`; these definitions check not only that the memory pointed to exists, but that the pointer to it is valid.

There are rare situations where a program needs to change type assignment of a pointer. The most common is in the memory allocator, which needs to create and destroy objects of arbitrary types from arrays of bytes in its memory pool. Therefore, VCC includes annotations (explained in Section B.1) that explicitly change the typestate. Thus, while your program can access memory using pretty much arbitrary types and typecasting, doing so will require additional annotations. But for most programs, checking type safety is completely transparent, so you don't have to worry about it.

6 Abstraction

Usually there are many ways of implementing a given data structure. For example, a set might be implemented as a linked list, an array, or a hash table.

When reasoning about a program which uses a data structure we don't want to be concerned with implementation details of the data structure. We should reason at somewhat higher, abstract level. For example, when we use a linked list as a representation of a set, we should not be concerned with how the list nodes are laid out in memory.

In VCC we abstract over such details using ghost data: data-structures

contain ghost fields with mathematical abstractions of what they represent. It is only rarely the case that a simple C type, say `unsigned int`, would be suitable to store such abstraction (how would one store an unbounded set in a primitive C type?). To that end, VCC provides ***map types***. The syntax is similar to syntax of array types, `int m[T*]` defines a map `m` from `T*` to `int`. That is the type of expression `m[p]` is `int`, provided that `p` is a pointer of type `T*`. A map `T* a[unsigned]` is similar to an array of pointers of length 2^{32}.[16] A map `bool s[int]` can be thought of as a set of `int`s: the operation `s[k]` will return true if and only if the element `k` is in the set `s`.

Let's then have a look at an example of a list abstracted as a set:

```
struct Node {
  struct Node *next;
  int data;
};

_(dynamic_owns) struct List {
  struct Node *head;
  _(ghost bool val[int];)
  _(invariant head != NULL ==> \mine(head))
  _(invariant \forall struct Node *n;
                \mine(n) ==> n->next == NULL || \mine(n->next))
  _(invariant \forall struct Node *n;
                \mine(n) ==> val[n->data])
};
```

The expressions inside `{...}` after the quantified variables are hints for the theorem prover called triggers. Ignore them for a moment. The invariant states that:

- the list owns the head node (if it's non-null)
- if the list owns a node, it also owns the next node (provided it's non-null)
- if the list owns a node, then its data is present in the set `val`; this binds the values stored in the implementation to the abstract representation

You may note that the set `val` is under-specified: it might be that it has some elements not stored in the list. We'll get back to this issue later. Now let's have a look at the specification of a function adding a node to the list:

```
int add(struct List *l, int k)
  _(requires \wrapped(l))
  _(ensures \wrapped(l))
  _(ensures \result != 0 ==> l->val == \old(l->val))
  _(ensures \result == 0 ==>
       \forall int p; l->val[p] == (\old(l->val)[p] || p == k))
  _(writes l)
```

[16] Because a map can be used only in ghost code, the issue of runtime memory consumption does not apply to it.

The writes-clause and contracts about the list being wrapped are similar to what we've seen before. Then, there are the contracts talking about the result value. This function might fail because there is not enough memory to allocate list node, in such case it will return a non-zero value (an error code perhaps), and the contracts guarantee that the set represented by the list will not be changed. However, if the function succeeds (and thus returns zero), the contract specifies that if we take an arbitrary integer `p`, then it is a member of the new abstract value if and only if it was already a member before or it is `k`.

In other words, the new value of `l->val` will be the union of the old abstract value and the element `k`. Ideally, this contract is all that the caller will need to know about that function: what kind of effect does it have on the abstract state. It doesn't specify if the node will be appended at the beginning, or in the middle of the list. It doesn't talk about possible duplicates or memory allocation. Everything about implementation is completely abstracted away. Still, we need a concrete implementation, and here it goes:

```
{
  struct Node *n = malloc(sizeof(*n));
  if (n == NULL) return -1;
  _(unwrapping l) {
    n->next = l->head;
    n->data = k;
    _(wrap n)
    l->head = n;
    _(ghost {
      l->\owns += n;
      l->val = (\lambda int z; z == k || l->val[z]);
    })
  }
  return 0;
}
```

We allocate the node, unwrap the list, initialize the new node and wrap it (we want to wrap the list, and thus everything it is going to own will need to be wrapped beforehand; the list is wrapped at the end of the unwrapping block), and prepend the node at the beginning of the list. Then we update the owns set (we've also already seen that). Finally, we update the abstract value using a ***lambda expression***. The expression `\lambda T x; E` returns a map, which for any `x` returns the value of expression `E`, which can reference `x`. If the type of `E` is `S`, then the type of map, returned by the lambda expression, is `S[T]`. An assignment `m = \lambda T x; E` has a similar effect to the following assumption (note that `E` will most likely reference `x`):

```
_(assume \forall T x; m[x] == E)
```

Unlike assumptions, lambda expressions do not compromise soundness of the verifier. Just like for assumptions, the expression is always evaluated in the state as it was when the lambda was first defined, for example:

```
int x = 1;
```

```
int m[int] = \lambda int y; y + x;
_(assert m[0] == 1) // succeeds
x = 2;
_(assert m[0] == 1) // still succeeds
```

One can imagine, that when this lambda expression is defined, VCC will iterate over all possible values of `x`, and store the value of `E` in `m[x]`. Lambda expressions are much like function values in functional languages or delegates in C#.

The body of our lambda expression shows similarity to the body of the quantifier we have used in specification. It doesn't, however, have to be the same:

```
int[int] foo(int v)
  _(ensures \forall int x; x >= 7 ==> \result[x] >= v)
{
  return \lambda int y; (y&1) == 0 ? INTMAX : v;
}
```

Thus, the specifications for lambda expressions can hide information.

6.1 More ghost state

The following subsection, till the beginning of Section 7 (which is about concurrency), might somewhat difficult upon first reading of this tutorial. It deals with the concept of the reachable set bookkeeping. It is not required to understand Section 7.

At minimum the list should support adding elements and checking for membership. For example, we would expect:

```
int member(struct List *l, int k)
  _(requires \wrapped(l))
  _(ensures \result != 0 <==> l->val[k])
```

Our current list invariant is strong enough only to show `\result != 0 ==> l->val[k]`, because it only says that if the list owns something, then it's in the `val`. It also says that if something can be reached by following the `next` field from the `head`, then it is owned. What we want to additionally say is that if something is in the `val` set, then it can be reached from the `head`. Unfortunately, such property is not directly expressible in first-order logic (which is the underlying logic of VCC specifications). To workaround this problem we associate with each node the set of values stored in all the following nodes and the current node. Additionally we say that the set for `NULL` node is empty. This way, as we walk down the `next` pointers we can keep track of all the elements that can be still reached. Once we reach the `NULL` pointer, we know that nothing more can be reached. The set of reachable nodes are stored as maps from `int` to `bool`. We need one such map per each node, so we just put a ghost map from `struct Node*` to the sets. Alternatively, we could store these sets as a

field inside of each node, but maps gives more flexibility in updating it using lambda expressions.

```
_(dynamic_owns) struct List {
  _(ghost bool val[int];)
  struct Node *head;
  _(ghost bool followers[struct Node *][int];)
  _(invariant val == followers[head])
  _(invariant head != NULL ==> \mine(head))
  _(invariant followers[NULL] == \lambda int k; \false)
  _(invariant \forall struct Node *n;
                \mine(n) ==> n->next == NULL || \mine(n->next))
  _(invariant \forall struct Node *n;
                \mine(n) ==>
                   \forall int e;
                      followers[n][e] <==>
                      followers[n->next][e] || e == n->data)
};
```

All these changes in the invariant do not affect the contract of `add()` function, and the only change in the body is that we need to replace the update of `l->val` with the following:

```
      l->followers[n] =
        (\lambda int z; l->followers[n->next][z] || z == k);
      l->val = l->followers[n];
```

That is adding a node at the head only affect the followers set of the new head, and the followers sets of all the other nodes remain unchanged. Now let us have a look at the `member()` function:

```
int member(struct List *l, int k)
  _(requires \wrapped(l))
  _(ensures \result != 0 <==> l->val[k])/*{endspec}*/
{
  struct Node *n;

  for (n = l->head; n; n = n->next)
    _(invariant n != NULL ==> n \in l->\owns)
    _(invariant l->val[k] <==> l->followers[n][k])
  {
    if (n->data == k)
      return 1;
  }
  return 0;
}
```

The invariants of the `for` loop state that we only iterate over nodes owned by the list, and that at each iteration `k` is in the set of values represented by the list if and only if it is in the followers set of the current node. Both are trivially true for the head of the list, for the first iteration of the loop. For each next iteration, the invariant of the list tells us that by following the `next` pointer

we stay in the owns set. It also tells us, that the `followers[n->next]` differs from `followers[n]` only by `n->data`. Thus, if `n->data` is not `val`, then the element, if it's in `followers[n]` must be also in `followers[n->next]`.

6.2 Sorting revisited

In Section 4.2 we have considered the bozo-sort algorithm. We have verified that the array after it returns is sorted. But we would also like to know that it's a permutation of the input array. To do that we will return a ghost map, which states the exact permutation that the sorting algorithm produced.

```
#include <vcc.h>
#include <stdlib.h>

_(logic bool sorted(int *buf, unsigned len) =
  \forall unsigned i, j; i < j && j < len ==> buf[i] <= buf[j])

_(typedef unsigned perm_t[unsigned]; )

_(logic bool is_permutation(perm_t perm, unsigned len) =
  (\forall unsigned i, j;
    i < j && j < len ==> perm[i] != perm[j]))

_(logic bool is_permuted(\state s, int *buf, unsigned len,
                         perm_t perm) =
  \forall unsigned i; i < len ==>
      perm[i] < len && \at(s, buf[ perm[i] ]) == buf[i])

_(logic perm_t swap(perm_t p, unsigned i, unsigned j) =
      \lambda unsigned k; k == i ? p[j] : k == j ? p[i] : p[k])

void bozo_sort(int *buf, unsigned len _(out perm_t perm))
  _(writes \array_range(buf, len))
  _(ensures sorted(buf, len))
  _(ensures is_permutation(perm, len))
  _(ensures is_permuted(\old(\now()), buf, len, perm))
{
  _(ghost \state s0 = \now() )

  _(ghost perm = \lambda unsigned i; i)

  if (len == 0) return;

  for (;;)
    _(invariant \mutable_array(buf, len))
    _(invariant is_permutation(perm, len))
    _(invariant is_permuted(s0, buf, len, perm))
  {
    int tmp;
    unsigned i = _(unchecked)((unsigned)rand()) % len;
```

```
    unsigned j = _(unchecked)((unsigned)rand()) % len;

    tmp = buf[i];
    buf[i] = buf[j];
    buf[j] = tmp;
    _(ghost perm = swap(perm, i, j) )

    for (i = 0; i < len - 1; ++i)
      _(invariant sorted(buf, i + 1))
    {
      if (buf[i] > buf[i + 1]) break;
    }

    if (i == len - 1) break;
  }
}
```

```
Verification of bozo_sort succeeded.
```

This sample introduces two new features. First is the output ghost parameter `_(out perm_t perm)`. We use it when we need a function to return something in addition to what it normally returns. To call `bozo_sort()` you need to supply a local variable to hold the permutation when the function exits, as in:

```
void f(int *buf, unsigned len)
  // ...
{
  _(ghost perm_t myperm; )
  // ...
  bozo_sort(buf, len _(out myperm));
}
```

The value is only copied on exit of `bozo_sort()`, thought its execution it has its own copy. It is thus different than passing a pointer to the local. It is also more efficient for the verifier.

The other, somewhat more advanced, feature is explicit state manipulation. The function `\now()` returns the current state of the heap (i.e., dynamically allocated memory; in future it will also work for locals, but for now it only applied to memory location, address of which was taken). The state is encapsulated in a value of type `\state`. The expression `\at(s, E)` returns value of expression `E` as evaluated in state `s`. You can see `\old(...)` as a special case of this.

Thus, the algorithm maintains the map containing the current permutation of the data, with respect to the initial data (we store the initial state in `s0`). The initial permutation is just identity, and whenever we swap elements of the array, we also swap elements of the permutation.

Exercises

1. Write and verify an iterative program that sorts an array of ints using bubblesort. The specification should be the same as for bozo-sort above.

7 Atomics

Writing concurrent programs is generally considered to be harder than writing sequential programs. Similar opinions are held about verification. Surprisingly, in VCC the leap from verifying sequential programs to verifying fancy lock-free stuff is not that big. This is because the verification in VCC is inherently based on invariants: conditions that attached to data and need hold *no matter which thread* accesses it.

But let us move from words to actions, and verify a canonical example of a lock-free algorithm, which is the implementation of a spin-lock itself. The spin-lock data-structure is really simple – it contains just a single field, meant to be interpreted as a boolean stating whether the spin-lock is currently acquired. However, in VCC we would like to attach some formal meaning to this boolean. We do that through ownership – the spin-lock will protect some object, and will own it whenever it is not acquired. Thus, the following invariant should come as no surprise:

```
_(volatile_owns) struct Lock {
  volatile int locked;
  _(ghost \object protected_obj;)
  _(invariant locked == 0 ==> \mine(protected_obj))
};
```

We use a ghost field to hold a reference to the object meant to be protected by this lock. If you wish to protect multiple objects with a single lock, you can make the object referenced by `protected_obj` own them all. The `locked` field is annotated with `volatile`. It has the usual meaning for the regular C compiler (i.e., it makes the compiler assume that the environment might write to that field, outside the knowledge of the compiler). For VCC it means that the field can be written also when the object is consistent (that is after wrapping it). The idea is that we will not unwrap the object, but write it atomically, while preserving its invariant. The attribute `_(volatile_owns)` means that we want the `\owns` set to be treated as a volatile field (i.e., we want to be able to write it while to object is consistent; normally this is not possible).

First, let's have a look at lock initialization:

```
void InitializeLock(struct Lock *l _(ghost \object obj))
  _(writes \span(l), obj)
  _(requires \wrapped(obj))
  _(ensures \wrapped(l) && l->protected_obj == obj)
{
  l->locked = 0;
  _(ghost {
```

```
    l->protected_obj = obj;
    l->\owns = {obj};
    _(wrap l)
  })
}
```

One new thing there is the use of ***ghost parameter***. The regular lock initialization function prototype does not say which object the lock is supposed to protect, but our lock invariant requires it. Thus, we introduce additional parameter for the purpose of verification. A call to the initialization will look like `InitiliazeLock(&l _(ghost o))`.

Second, we require that the object to be protected is wrapped (recall that wrapped means consistent and owned by the current thread). We need it to be consistent because we will want to make the lock own it, and lock can only own consistent objects. We need the current thread to own it, because ownership transfer can only happen between the current thread and an object, and not for example some other thread and an object. Third, we say we're going to write the protected object. This allows for the transfer, and prevents the calling function from assuming that the object stays wrapped after the call. Note that this contract is much like the contract of the function adding an object to a container data-structure, like `sc_add()` from Section 5.3.

Now we can see how we operate on volatile fields. We shall start with the function releasing the lock, as it is simpler, than the one acquiring it.

```
void Release(struct Lock *l)
  _(requires \wrapped(l))
  _(requires \wrapped(l->protected_obj))
  _(writes l->protected_obj)
{
  _(atomic l) {
    l->locked = 0;
    _(ghost l->\owns += l->protected_obj)
  }
}
```

First, let's have a look at the contract. `Release()` requires the lock to be wrapped.[17] The preconditions on the protected object are very similar to the preconditions on the `InitiliazeLock()`. Note that the `Release()` does not mention the lock it its writes clause, this is because the write it performs is volatile. Intuitively, VCC needs to assume such writes can happen at any time, so one additional write from this function doesn't make a difference.

The `atomic` block is similar in spirit to the `unwrapping` block — it allows for modifications of listed objects and checks if their invariants are preserved. The difference is that the entire update happens instantaneously from the point of view of other threads. We needed the unwrapping operation because we wanted to mark that we temporarily break the object invariants. Here, there is no point

[17] You might wonder how multiple threads can all own the lock (to have it wrapped), we will fix that later.

in time where other threads can observe that the invariants are broken. Invariants hold before the beginning of the atomic block (by our principal reasoning rule, Section 5.1), and we check the invariant at the end of the atomic block.

The question arises, what guarantees that other threads won't interfere with the atomic action? VCC allows only one physical memory operation inside of an atomic block, which is indeed atomic from the point of view of the hardware. Here, that operation is writing to the `l->locked`. Other possibilities include reading from a volatile field, or a performing a primitive operation supported by the hardware, like interlocked compare-and-exchange. However, inside our atomic block we can also see the update of the owns set. This is fine, because the ghost code is not executed by the actual hardware.

> [**TODO:** rewrite this:] If we were to imagine a VCC machine, which would actually execute the ghost code, we could say that it blocks other threads when the ghost code is executing. Because the real machine doesn't execute any ghost code, it doesn't need to do any such blocking.

It is not particularly difficult to see that this atomic operation preserves the invariant of the lock. But this isn't the only condition imposed by VCC here. To transfer ownership of `l->protected_obj` to the lock, we also need write permission to the object being transferred, and we need to know its consistent. For example, should we forget to mention `l->protected_obj` in the writes clause VCC will complain about:

```
Verification of Lock#adm succeeded.
Verification of Release failed.
testcase(16,27) : error VC8510: Assertion 'l->protected_obj is
    writable in call to l->\owns += l->protected_obj' did not
    verify.
```

Should we forget to perform the ownership transfer inside of `Release()`, we'll get complain about the invariant of lock.

```
Verification of Lock#adm succeeded.
Verification of Release failed.
testcase(15,26) : error VC8524: Assertion 'chunk locked == 0
    ==> \mine(protected_obj) of invariant of l holds after
    atomic' did not verify.
```

Let's then move to `Acquire()`. The specification is not very surprising: it requires the lock to be wrapped, and ensures that after the call the thread will own the protected object, and moreover, that the thread didn't directly own it before. This is much like the postcondition on `sc_add()` function from Section 5.3.

```
void Acquire(struct Lock *l)
```

```
  _(requires \wrapped(l))
  _(ensures \wrapped(l->protected_obj) &&
      \fresh(l->protected_obj))
{
  int stop = 0;

  do {
    _(atomic l) {
      stop = InterlockedCompareExchange(&l->locked, 1, 0) == 0;
      _(ghost if (stop) l->\owns -= l->protected_obj)
    }
  } while (!stop);
}
```

The `InterlockedCompareAndExchange()` function is a compiler built-in, which on the x86/x64 hardware translates to `cmpxchg` assembly instruction. It takes a memory location and two values. If the memory location contains the first value, then it is replaced with the second. It returns the old value. The entire operation is performed atomically (and is also a write barrier).

VCC doesn't have all the primitives of all the C compilers predefined. One can define them by suppling a body. It is presented only to the VCC compiler (it is enclosed in `_(atomic_inline ...)`) so that the normal compiler doesn't get confused about it.

```
_(atomic_inline) int InterlockedCompareExchange(volatile
    int *Destination, int Exchange, int Comparand) {
  if (*Destination == Comparand) {
    *Destination = Exchange;
    return Comparand;
  } else {
    return *Destination;
  }
}
```

This is one of the places where one needs to be very careful, as there is no way for VCC to know if the definition you provided matches the semantics of your regular C compiler. Make sure to check with the regular C compiler manual for exact semantics of its built-in functions.

The header files coming with VCC provide a handful of popular operations, you can just rename them to fit your compiler. [**TODO:** we should actually do that]

7.1 Using claims

The contracts of functions operating on the lock require that the lock is wrapped. This is because one can only perform atomic operations on objects that are consistent. If object is inconsistent, then the owning thread is in full control of it. However, wrapped means not only consistent, but also owned by the

current thread, which defeats the purpose of the lock — it should be possible for multiple threads to compete for the lock. Let's then say, there is a thread which owns the lock. Assume some other thread `t` got to know that the lock is consistent. How would `t` know that the owning thread won't unwrap (or worse yet, deallocate) the lock, just before `t` tries an atomic operation on the lock? The owning thread thus needs to somehow promise `t` that lock will stay consistent. In VCC such a promise takes form of a ***claim***. Later we'll see that claims are more powerful, but for now consider the following to be the definition of a claim:

```
_(ghost
typedef struct {
  \ptrset claimed;
  _(invariant \forall \object o; o \in claimed ==>
      o->\consistent)
} \claim_struct, *\claim;
)
```

Thus, a claim is an object, with an invariant stating that a number of other objects (we call them ***claimed objects***) are consistent. As this is stated in the invariant of the claim, it only needs to be true as long as the claim itself stays consistent.

Recall that what can be written in invariants is subject to the admissibility condition, which we have seen partially explained in Section 5.5. There we said that an invariant should talk only about things the object owns. But here the claim doesn't own the claimed objects, so how should the claim know the object will stay consistent? In general, an admissible invariant can depend on other objects invariants always being preserved (we'll see the precise rule in Section 7.3). So VCC adds an implicit invariant to all types marked with `_(claimable)` attribute. This invariant states that the object cannot be unwrapped when when there are consistent claims on it. More precisely, each claimable object keeps track of the count of outstanding claims. The number of outstanding claim on an object is stored in `\claim_count` field.

Now, getting back to our lock example, the trick is that there can be multiple claims claiming the lock (note that this is orthogonal to the fact that a single claim can claim multiple objects). The thread that owns the lock will need to keep track of who's using the lock. The owner won't be able to destroy the lock (which requires unwrapping it), before it makes sure there is no one using the lock. Thus, we need to add `_(claimable)` attribute to our lock definition, and change the contract on the functions operating on the lock. As the changes are vary similar we'll only show `Release()`.

```
void Release(struct Lock *l _(ghost \claim c))
  _(requires \wrapped(c) && \claims_object(c, l))
  _(requires l->protected_obj != c)
  _(requires \wrapped(l->protected_obj))
  _(ensures \wrapped(c))
  _(writes l->protected_obj)
{
```

```
  _(atomic c, l) {
    _(assert \by_claim(c, l->protected_obj) != c) // why do we
        need it?
    l->locked = 0;
    _(ghost l->\owns += l->protected_obj)
  }
}
```

We pass a ghost parameter holding a claim. The claim should be wrapped. The function `\claims_obj(c, l)` is defined to be `l \in c->claimed`, i.e., that the claim claims the lock. We also need to know that the claim is not the protected object, otherwise we couldn't ensure that the claim is wrapped after the call. This is the kind of weird corner case that VCC is very good catching (even if it's bogus in this context). Other than the contract, the only change is that we list the claim as parameter to the atomic block. [**TODO:** our current syntax is horrible, we need something different] Listing a normal object as parameter to the atomic makes VCC know you're going to modify the object. For claims, it is just a hint, that it should use this claim when trying to prove that the object is consistent.

Additionally, the `InitiliazeLock()` needs to ensure `l->\claim_count == 0` (i.e., no claims on freshly initialized lock). VCC even provides a syntax to say something is wrapped and has no claims: `\wrapped0(l)`.

7.2 Creating claims

When creating (or destroying) a claim one needs to list the claimed objects. Let's have a look at an example.

```
void create_claim(struct Data *d)
  _(requires \wrapped(d))
  _(writes d)
{
  _(ghost \claim c;)
  struct Lock l;
  InitializeLock(&l _(ghost d));
  _(ghost c = \make_claim({&l}, \true);)
  Acquire(&l _(ghost c));
  Release(&l _(ghost c));
  _(ghost \destroy_claim(c, {&l}));
  _(unwrap &l)
}
```

This function tests that we can actually create a lock, create a claim on it, use the lock, and then destroy it. The `InitiliazeLock()` leaves the lock wrapped and writable by the current thread. This allows for creation of claim, which is then passed to `Acquire()` and `Release()`. Finally, we destroy the claim, which allows for unwrapping of the lock, and finally deallocating it when the function activation record is popped off the stack.

The `\make_claim(...)` function takes the set of objects to be claimed and a property (an invariant of the claim, we'll get to that in the next section). Let us give desugaring of `\make_claim(...)` for a single object in terms of the `\claim_struct` defined in the previous section.

```
// c = \make_claim({o}, true) expands to
o->\claim_count += 1;
c = malloc(sizeof(\claim_struct));
c->claimed = {o};
_(wrap c);

// \destroy_claim(c, {o}) expands to
assert(o \in c->claimed);
o->\claim_count -= 1;
_(unwrap c);
free(c);
```

Because creating or destroying a claim on `c` assigns to `c->\claim_count`, it requires write access to that memory location. One way to obtain such access is getting sequential write access to `c` itself: in our example the lock is created on the stack and thus sequentially writable. We can thus create a claim and immediately use it. A more realistic claim management scenario is described in Section 7.5.

The `\true` in `\make_claim(...)` is the claimed property (an invariant of the claim), which will be explained in the next section.

The destruction can possibly leak claim counts, i.e., one could say:

```
\destroy_claim(c, {});
```

and it would verify just fine. This avoids the need to have write access to `p`, but on the other hand prevents `p` from unwrapping forever (which might be actually fine if `p` is a ghost object).

7.3 Two-state invariants

Sometimes it is not only important what are the valid states of objects, but also what are the allowed *changes* to objects. For example, let's take a counter keeping track of certain operations since the beginning of the program.

```
_(claimable) struct Counter {
  volatile unsigned v;
  _(invariant v > 0)
  _(invariant v == \old(v) || v == \old(v) + 1)
};
```

Its first invariant is plain single-state invariant – for some reason we decided to exclude zero as the valid count. The second invariant says that for any atomic update of (consistent) counter, `v` can either stay unchanged or increment by exactly one. The syntax `\old(v)` is used to refer to value of `v` before an atomic

update, and plain `v` is used for the value of `v` after the update. That is, when checking that an atomic update preserves the invariant of a counter, we will take the state of the program right before the update, the state right after the update, and check that the invariant holds for that pair of states.

> In fact, it would be easy to prevent any changes to some field `f`, by saying `_(invariant \old(f)== f)`. This is roughly what happens under the hood when a field is declared without the `volatile` modifier.

As we can see the single- and two-state invariants are both defined using the `_(invariant ...)` syntax. The single-state invariants are just two-state invariants, which do not use `\old(...)`. However, we often need an interpretation of an object invariant in a single state `S`. For that we use the ***stuttering*** transition from `S` to `S` itself. VCC enforces that all invariants are ***reflexive*** that is if they hold over a transition `S0, S1`, then they should hold in just `S1` (i.e., over `S1, S1`). In practice, this means that `\old(...)` should be only used to describe how objects change, and not what are their proper values. In particular, all invariants which do not use `\old(...)` are reflexive, and so are all invariants of the form `\old(E)== E || P`, for any expression `E` and condition `P`. On the other hand, the invariants `\old(f)< 7` and `x == \old(x)+ 1` are not reflexive.

Let's now discuss where can you actually rely on invariants being preserved.

```
void foo(struct Counter *n)
  _(requires \wrapped(n))
{
  int x, y;
  atomic(n) { x = n->v; }
  atomic(n) { y = n->v; }
}
```

The question is what do we know about `x` and `y` at the end of `foo()`. If we knew that nobody is updating `n->v` while `foo()` is running we would know `x == y`. This would be the case if `n` was unwrapped, but it is wrapped. In our case, because `n` is consistent, other threads can update it, while `foo()` is running, but they will need to adhere to `n`'s invariant. So we might guess that at end of `foo()` we know `y == x || y == x + 1`. But this is incorrect: `n->v` might get incremented by more than one, in several steps. The correct answer is thus `x <= y`. Unfortunately, in general, such properties are very difficult to deduce automatically, which is why we use plain object invariants and admissibility check to express such properties in VCC.

> An invariant is ***transitive*** if it holds over states `S0, S2`, provided that it holds over `S0, S1` and `S1, S2`. Transitive invariants could be assumed over arbitrary pairs of states, provided that the object stays consistent in between them. VCC does not require invariants to be transitive though.
>
> Some invariants are naturally transitive (e.g., we could say `_(invariant \old(x)<= x)` in `struct Counter`, and it would be almost as good our

current invariant). Some other invariants, especially the more complicated ones, are more difficult to make transitive. For example, an invariant on a reader-writer lock might say

```
_(invariant writer_waiting ==> old(readers) >= readers)
```

To make it transitive one needs to introduce version numbers. Some invariants describing hardware (e.g., a step of physical CPU) are impossible to make transitive.

Consider the following structure definition:

```
struct Reading {
  struct Counter *n;
  volatile unsigned r;
  _(ghost \claim c;)
  _(invariant \mine(c) && \claims_object(c, n))
  _(invariant n->v >= r)
};
```

It is meant to represent a reading from a counter. Let's consider its admissibility. It has a pointer to the counter, and a owns a claim, which claims the counter. So far, so good. It also states that the current value of the counter is no less than `r`. Clearly, the `Reading` doesn't own the counter, so our previous rule from Section 5.5, which states that you can mention in your invariant everything that you own, doesn't apply. It would be tempting to extend that rule to say "everything that you own or have a claim on", but VCC actually uses a more general rule. In a nutshell, the rule says that every invariant should be preserved under changes to other objects, provided that these other objects change according to their invariants. When we look at our `struct Reading`, its invariant cannot be broken when its counter increments, which is the only change allowed by counters invariant. On the other hand, an invariant like `r == n->v` or `r >= n->v` could be broken by such a change. But let us proceed with somewhat more precise definitions.

First, assume that every object invariant holds when the object is inconsistent. This might sound counter-intuitive, but remember that consistency is controlled by a field. When that field is set to false, we want to *effectively* disable the invariant, which is the same as just forcing it to be true in that case. Alternatively, you might try to think of all objects as being consistent for a while.

An atomic action, which updates state `S0` into `S1`, is ***legal*** if and only if the invariants of objects that have changed between `S0` and `S1` hold over `S0, S1`. In other words, a legal action preservers invariants of updated objects. This should not come as a surprise: this is exactly what VCC checks for in atomic blocks.

An invariant is ***stable*** if and only if it cannot be broken by legal updates. More precisely, to prove that an invariant of `p` is stable, VCC needs to "simulate" an arbitrary legal update:

- Take two arbitrary states `S0` and `S1`.

- Assume that all invariants (including p's) hold over S0, S0.
- Assume that for all objects, some fields of which are not the same in S0 and S1, their invariants hold over S0, S1.
- Assume that all fields of p are the same in S0 and S1.
- Check that invariant of p holds over S0, S1.

The first assumption comes from the fact that all invariants are reflexive. The second assumption is legality. The third assumption follows from the second (if p did change, its invariant would automatically hold).

An invariant is ***admissible*** if and only if its stable and reflexive.

First, let's see how our previous notion of admissibility relates to this one. If p owns q, then q \in p->\owns. By the third admissibility assumption, after the simulated action p still owns q. By the rules of ownership (Section 5.1), only threads can own inconsistent objects, so we know that q is consistent in both S0 and S1. Therefore non-volatile fields of q do not change between S0 and S1, and thus the invariant of p can freely talk about their values: whatever property of them was true in S0, will also be true in S1. Additionally, if q owned r before the atomic action, and the q->\owns is non-volatile, it will keep owning r, and thus non-volatile fields of r will stay unchanged. Thus our previous notion of admissibility is a special case of this one.

Getting back to our foo() example, to deduce that x <= y, after the first read we could create a ghost Reading object, and use its invariant in the second action. While we need to say that x <= y is what's required, using a full-fledged object might seem like an overkill. Luckily, definitions of claims themselves can specify additional invariants.

The admissibility condition above is semantic: it will be checked by the theorem prover. This allows construction of the derived concepts like claims and ownership, and also escaping their limitations if needed. It is therefore the most central concept of VCC verification methodology, even if it doesn't look like much at the first sight.

7.4 Guaranteed properties in claims

When constructing a claim, you can specify additional invariants to put on the imaginary definition of the claim structure. Let's have a look at annotated version of our previous foo() function.

```
void readtwice(struct Counter *n)
  _(requires \wrapped(n))
  _(writes n)
{
  unsigned int x, y;
  _(ghost \claim r;)
```

```
  _(atomic n) {
    x = n->v;
    _(ghost r = \make_claim({n}, x <= n->v);)
  }

  _(atomic n) {
    y = n->v;
    _(assert \active_claim(r))
    _(assert x <= y)
  }
}
```

First, a high-level description of what's going on. Just after reading `n->v` we create a claim `r`, which guarantees that in every state, where `r` is consistent, the current value of `n->v` is no less than the value of `x` at the time when `r` was created. Then, after reading `n->v` for the second time, we tell VCC to make use of `r`'s guaranteed property, by asserting that it is "active". This makes VCC know `x <= n->v` in the current state, where also `y == n->v`. From these two facts VCC can conclude that `x <= y`.

The general syntax for constructing a claim is:

```
_(ghost c = \make_claim(S, P))
```

We already explained, that this requires that `s->\claim_count` is writable for `s \in S`. As for the property `P`, we pretend it forms the invariant of the claim. Because we're just constructing the claim, just like during regular object initialization, the invariant has to hold initially (i.e., at the moment when the claim is created, that is wrapped). Moreover, the invariant has to be admissible, under the condition that all objects in `S` stay consistent as long as the claim itself stays consistent. The claimed property cannot use `\old(...)`, and therefore it's automatically reflexive, thus it only needs to be stable to guarantee admissibility.

But what about locals? Normally, object invariants are not allowed to reference locals. The idea is that when the claim is constructed, all the locals that the claim references are copied into imaginary fields of the claim. The fields of the claim never change, once it is created. Therefore an assignment `x = MAX_UINT;` in between the atomic blocks would not invalidate the claim — the claim would still refer to the old value of `x`. Of course, it would invalidate the final `x <= y` assert.

> For any expression `E` you can use `\at(\now(), E)` in `P` in order to have the value of `E` be evaluated in the state when the claim is created, and stored in the field of the claim.

This copying business doesn't affect initial checking of the `P`, `P` should just hold at the point when the claim is created. It does however affect the admissibility check for `P`:

- Consider an arbitrary legal action, from `S0` to `S1`.
- Assume that all invariants hold over `S0, S0`, including assuming `P` in `S0`.

- Assume that fields of `c` didn't change between `S0` and `S1` (in particular locals referenced by the claim are the same as at the moment of its creation).
- Assume all objects in `S` are consistent in both `S0` and `S1`.
- Assume that for all objects, fields of which are not the same in `S0` and `S1`, their invariants hold over `S0, S1`.
- Check that `P` holds in `S1`.

To prove `\active_claim(c)` one needs to prove `c->\consistent` and that the current state is a ***full-stop*** state, i.e., state where all invariants are guaranteed to hold. Any execution state outside of an atomic block is full-stop. The state right at the beginning of an atomic block is also full-stop. The states in the middle of it (i.e., after some state updates) might not be.

> Such middle-of-the-atomic states are not observable by other threads, and therefore the fact that the invariants don't hold there does not create soundness problems.

The fact that `P` follows from `c`'s invariant after the construction is expressed using `\claims(c, P)`. It is roughly equivalent to saying:

```
\forall \state s {\at(s, \active_claim(c))};
  \at(s, \active_claim(c)) ==> \at(s, P)
```

Thus, after asserting `\active_claim(c)` in some state `s`, `\at(s, P)` will be assumed, which means VCC will assume `P`, where all heap references are replaced by their values in `s`, and all locals are replaced by the values at the point when the claim was created.

[**TODO:** I think we need more examples about that at() business, claim admissibility checks and so forth]

7.5 Dynamic claim management

So far we have only considered the case of creating claims to wrapped objects. In real systems some resources are managed dynamically: threads ask for "handles" to resources, operate on them, and give the handles back. These handles are usually purely virtual — asking for a handle amounts to incrementing some counter. Only after all handles are given back the resource can be disposed. This is pretty much how claims work in VCC, and indeed they were modeled after this real-world scenario. Below we have an example of prototypical reference counter.

```
struct RefCnt {
  volatile unsigned cnt;
  _(ghost \object resource;)
  _(invariant \mine(resource))
  _(invariant \claimable(resource))
```

```
  _(invariant resource->\claim_count == cnt >> 1)
  _(invariant \old(cnt & 1) ==> \old(cnt) >= cnt)
};
```

Thus, a `struct RefCnt` owns a resource, and makes sure that the number of outstanding claims on the resource matches the physical counter stored in it. `\claimable(p)` means that the type of object pointed to by `p` was marked with `_(claimable)`. The lowest bit is used to disable giving out of new references (this is expressed in the last invariant).

```
void init(struct RefCnt *r _(ghost \object rsc))
  _(writes \span(r), rsc)
  _(requires \wrapped0(rsc) && \claimable(rsc))
  _(ensures \wrapped(r) && r->resource == rsc)
{
  r->cnt = 0;
  _(ghost r->resource = rsc;)
  _(wrap r)
}
```

Initialization shouldn't be very surprising: `\wrapped0(o)` means `\wrapped(o)&& o->\claim_count == 0`, and thus on initialization we require a resource without any outstanding claims.

```
int try_incr(struct RefCnt *r _(ghost \claim c)
             _( out \claim ret))
  _(always c, r->\consistent)
  _(ensures \result == 0 ==>
     \claims_object(ret, r->resource) && \wrapped0(ret) &&
         \fresh(ret))
{
  unsigned v, n;

  for (;;) {
    _(atomic c, r) { v = r->cnt; }
    if (v & 1) return -1;

    _(assume v <= UINT_MAX - 2)
    _(atomic c, r) {
      n = InterlockedCompareExchange(&r->cnt, v + 2, v);
      _(ghost
        if (v == n) ret = \make_claim({r->resource}, \true);)
    }

    if (v == n) return 0;
  }
}
```

First, let's have a look at the function contract. The syntax `_(always c, P)` is equivalent to:

```
  _(requires \wrapped(c) && \claims(c, P))
```

```
  _(ensures \wrapped(c))
```

Thus, instead of requiring `\claims_obj(c, r)`, we require that the claim guarantees `r->\consistent`. One way of doing this is claiming `r`, but another is claiming the owner of `r`, as we will see shortly.

As for the body, we assume our reference counter will never overflow. This clearly depends on the running time of the system and usage patterns, but in general it would be difficult to specify this, and thus we just hand-wave it.

The new thing about the body is that we make a claim on the resource, even though it's not wrapped. There are two ways of obtaining write access to `p->\claim_count`: either having `p` writable sequentially and wrapped, or in case `p->\owner` is a non-thread object, checking invariant of `p->\owner`. Thus, inside an atomic update on `p->\owner` (which will check the invariant of `p->\owner`) one can create claims on `p`. The same rule applies to claim destruction:

```
void decr(struct RefCnt *r _(ghost \claim c) _(ghost \claim
    handle))
  _(always c, r->\consistent)
  _(requires \claims_object(handle, r->resource) &&
      \wrapped0(handle))
  _(requires c != handle)
  _(writes handle)
{
  unsigned v, n;

  for (;;)
    _(invariant \wrapped(c) && \wrapped0(handle))
  {
    _(atomic c, r) {
      v = r->cnt;
      _(assert \active_claim(handle))
      _(assert v >= 2)
    }

    _(atomic c, r) {
      n = InterlockedCompareExchange(&r->cnt, v - 2, v);
      _(ghost
        if (v == n) {
          _(ghost \destroy_claim(handle, {r->resource}));
        })
    }

    if (v == n) break;
  }
}
```

A little tricky thing here, is that we need to make use of the `handle` claim right after reading `r->cnt`. Because this claim is valid, we know that the claim count on the resource is positive and therefore (by reference counter invariant)

`v >= 2`. Without using the `handle` claim to deduce it we would get a complaint about overflow in `v - 2` in the second atomic block.

Finally, let's have a look at a possible use scenario of our reference counter.

```
_(claimable) struct A {
  volatile int x;
};

struct B {
  struct RefCnt rc;
  struct A a;
  _(invariant \mine(&rc))
  _(invariant rc.resource == &a)
};

void useb(struct B *b _(ghost \claim c))
  _(always c, b->\consistent)
{
  _(ghost \claim ac;)
  if (try_incr(&b->rc _(ghost c) _(out ac)) == 0) {
    _(atomic &b->a, ac) {
      b->a.x = 10;
    }
    decr(&b->rc _(ghost c) _(ghost ac));
  }
}

void initb(struct B *b)
  _(writes \extent(b))
  _(ensures \wrapped(b))
{
  b->a.x = 7;
  _(wrap &b->a)
  init(&b->rc _(ghost &b->a));
  _(wrap b)
}
```

The `struct B` contains a `struct A` governed by a reference counter. It owns the reference counter, but not `struct A` (which is owned by the reference counter). A claim guaranteeing consistency of `struct B` also guarantees consistency of its counter, so we can pass it to `try_incr()`, which gives us a handle on `struct A`.

Of course a question arises where one does get a claim on `struct B` from? In real systems the top-level claims come either from global objects that are always consistent, or from data passed when the thread is created.

A Triggers

The triggers are likely the most difficult part of this tutorial. As of recently, the trigger inference algorithm in VCC has been improved, so triggers need to be

used less often. Thus, we didn't need any trigger annotations for the examples in the tutorial. Still, you'll need them to deal with more complex VCC verification tasks.

SMT solvers, which are the underlying VCC theorem proving technology, prove that the program is correct by looking for possible counterexamples, or ***models***, where your program goes wrong (e.g., by violating an assertion). Once the solver goes through *all* possible counterexamples, and finds them all to be inconsistent (i.e., impossible), it considers the program to be correct. Normally, it would take virtually forever, for there is very large number of possible counterexamples, one for every input to the function (values stored in the heap also count as input). To workaround this problem, the SMT solver considers ***partial models***, i.e., sets of statements about the state of the program. For example, the model description may say `x == 7`, `y > x` and `*p == 12`, which describes all the concrete models, where these statements hold. There is great many such models, for example one for each different values `y` and other program variables, not even mentioned in the model.

It is thus useful to think of the SMT solver as sitting there with a possible model, and trying to find out whether the model is consistent or not. For example, if the description of the model says that `x > 7` and `x < 3`, then the solver can apply rules of arithmetic, conclude this is impossible, and move on to a next model. The SMT solvers are usually very good in finding inconsistencies in models where the statements describing them do not involve universal quantifiers. With quantifiers things tend to get a bit tricky.

For example, let's say the model description states that the two following facts are true:

```
\forall unsigned i; i < 10 ==> a[i] > 7
a[4] == 3
```

The meaning of the universal quantifier is that it should hold not matter what we substitute for `i`, for example the universal quantifier above implies the following facts (which are called ***instances*** of the quantifier):

```
 4 < 10 ==>  a[4] > 7  // for i == 4
```

which happens to be the one needed to refute our model,

```
11 < 10 ==> a[11] > 7  // for i == 11
```

which is trivially true, because false implies everything, and

```
 k < 10 ==>  a[k] > 7  // for i == k
```

where `k` is some program variable of type `unsigned`.

However, there is potentially infinitely many such instances, and certainly too many to enumerate them all. Still, to prove that our model candidate is indeed contradictory we only need the first one, not the other two. Once the solver adds it to the model description, it will simplify `4 < 10` to true, and then see that `a[4] > 7` and `a[4] == 3` cannot hold at the same time.

The question remains: how does the SMT solver decide that the first instance is useful, and the other two are not? This is done through so called ***triggers***.

Triggers are either specified by the user or inferred automatically by the SMT solver or the verification tool. In all the examples before we just relied on the automatic trigger inference, but as we go to more complex examples, we'll need to consider explicit trigger specification.

A trigger for a quantified formula is usually some subexpression of that formula, which contains all the variables that the formula quantifies over. For example, in the following formula:

```
\forall int i; int p[int]; is_pos(p, i) ==> f(i, p[i]) && g(i)
```

possible triggers include the following expressions `is_pos(p, i)`, `p[i]`, and also `f(i, p[i])`, whereas `g(i)` would not be a valid trigger, because it does not contain `p`.

Let's assume that `is_pos(p, i)` is the trigger. The basic idea is that when the SMT solvers considers a model, which mentions `is_pos(q, 7)` (where `q` is, e.g., a local variable), then the formula should be instantiated with `q` and `7` substituted for `p` and `i` respectively.

Note that the trigger `f(i, p[i])` is ***more restrictive*** than `p[i]`: if the model contains `f(k, q[k])` it also contains `q[k]`. Thus, a "bigger" trigger will cause the formula to be instantiated less often, generally leading to better proof performance, but also possibly preventing VCC from proving some assertions.

Triggers cannot contain boolean operators or the equality operator. As of the current release, arithmetic operators are allowed, but cause warnings and work unreliably, so you should avoid them.

A formula can have more than one trigger. It is enough for one trigger to match in order for the formula to be instantiated.

The explicit triggers are listed in `{...}`, after the quantified variables. They don't have to be subexpressions of the formula. We'll see some examples of that later. However, the implicit, solver-selected, triggers always are subexpressions of the formula. To select default triggers the SMT solver first considers all valid subexpression triggers, in the example above that would be: `is_pos(p, i)`, `p[i]`, and `f(i, p[i])`. Then, it looks if any of these is a subexpression of another. If it is so, the bigger of the two is discarded. This leads to discarding `f(i, p[i])`. Note that this is the *least* restrictive set of triggers possible, that is one that will trigger *most* often. Thus, specifying explicit triggers will usually make them more restrictive.[18] Also, adding additional explicit triggers, will make the quantified formula trigger more often. In future, it will likely be VCC (which has a lot more information available), not the solver, who will decide on default triggers, but for now the SMT solver choices might not be what you want.

Multi-triggers: Consider the following formula:

```
\forall int a, b, c; P(a, b) && Q(b, c) ==> R(a, c)
```

There is no subexpression here, which would contain all the variables and not contain boolean operators. In such case we need to use a ***multi-trigger***,

[18] Unless you use a trigger which is not a subexpression of the formula.

which is a set of expressions which together cover all variables. An example trigger here would be `{P(a, b), Q(b, c)}`. It means that for any model, which has both `P(a, b)` and `Q(b, c)` (for the same `b`!), the quantifier will be instantiated. In case a formula has multiple multi-triggers, *all* expressions in at least *one* of multi-triggers must match for the formula to be instantiated.

If it is impossible to select any single-triggers in the formula, and none are specified explicitly, Z3 will select *some* multi-trigger, which is usually not something that you want.

A.0.1 Typical triggers

[**TODO:** maybe we want a section like that?]

```
\forall unsigned i; {a[i]} i < 100 ==> a[i] > 0

\forall unsigned i, j; {f(i, j)} f(i, j) == i + j * 2
```

A.0.2 Matching loops

Consider a model description

```
\forall struct Node *n; {\mine(n)} \mine(n) ==> \mine(n->next)
\mine(a)
```

Let's assume the SMT solver will instantiate the quantifier with `a`, yielding:

```
\mine(a) ==> \mine(a->next)
```

It will now add `\mine(a->next)` to the set of facts describing the model. This however will lead to instantiating the quantifier again, this time with `a->next`, and in turn again with `a->next->next` and so forth. Such situation is called a ***matching loop***. The SMT solver would usually cut such loop at a certain depth, but it might make the solver run out of time, memory, or both.

Note that, if you skip the explicit trigger annotation, by the definition above, `{\mine(n)}` and `{n->next}` are still going to be default triggers of the formula above, still leading to the matching loop.

To avoid matching loop we need to use a different, more restrictive, trigger, for example `{\mine(n->next)}`. This will cause the quantifier to be instantiated only when the current model talks about `n->next` already. It is thus more "goal-oriented". This is a typical pattern when specifying properties of recursive data-structures using quantified formulas in VCC. This is what our example does, however instead of using the shorthand notation, it uses explicit owns set membership. [**TODO:** we need to fix that]

So the upside of the more restrictive explicit trigger is that it avoids excessive quantifier instantiations. The downside is that if the SMT solver is missing some instantiations, it might fail to rule out a model and report a bogus counterexample. This doesn't happen in for our list, but we'll see such problems (but with solutions!) later.

B Memory model

In most situations in C the type of a pointer is statically known: while at the machine code level the pointer is passed around as a type-less word, at the C level, in places where it is used, we know its type. VCC memory model makes this explicit: pointers are understood as pairs of their type and address (an word or integer representing location in memory understood as an array of bytes). For any state of program execution, VCC maintains the set of ***proper pointers***. [**TODO:** we might want a better name] Only proper pointers can be accessed (read or written). There are rules on changing the proper pointer set — e.g., one can remove a pointer `(T*)a`, and add pointers `(char*)a`, `(char*)(a+1)`, ..., `(char*)(a+sizeof(T))`, or *vice versa*. These rules make sure that at any given time, representations of two unrelated proper pointers do not overlap, which greatly simplifies reasoning. Note that given a `struct SafeString *p`, when `\proper(p)` we will also expect `\proper(&p->len)`. That is, when a structure is proper, and thus safe to access, so should be all its fields. This is what "unrelated" means in the sentence above: the representations overlap if and only if they pointer refer to a struct and fields of that struct. It is OK that fields overlap with their containing struct, but that structs overlap each other.

B.1 Reinterpretation

C Bitvector reasoning

Remember our first `min()` example? Surprisingly it can get more involved. For example the one below does not use a branch.

```
#include <vcc.h>

int min(int a, int b)
  _(requires \true)
  _(ensures \result <= a && \result <= b)
{
  _(assert {:bv} \forall int x; (x & (-1)) == x)
  _(assert {:bv} \forall int a,b; (a - (a - b)) == b)
  return _(unchecked)(a - ((a - b) & -(a > b)));
}
```

```
Verification of min succeeded.
```

The syntax:

```
_( assert {bv} \forall int x; (x & -1) == x )
```

VCC to prove the assertion using the fixed-length bit vector theory, a.k.a. machine integers. This is then used as a lemma to prove the postcondition.

References

[1] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009. Invited paper.

[2] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In Byron Cook, Paul Jackson, and Tayssir Touili, editors, *Computer Aided Verification (CAV 2010)*, volume 6174 of *Lecture Notes in Computer Science*, pages 480–494, Edinburgh, UK, July 2010. Springer.