Data Abstraction in VCC

Ernie COHEN [a,1]
[a] *Microsoft Corporation*

Keywords. Software Verification, Data Abstraction, Concurrency

In these notes, we present a methodology for verifying C code, i.e. proving mechanically that it meets its functional specifications. We target C because it is (along with C++) the the language of choice for writing "important" software (operating system kernels, device drivers, compilers, etc.). The methodology has been developed for VCC, a verifier for concurrent C code. VCC and papers about it can be found at `vcc.codeplex.com`. These notes are not intended as a tutorial on VCC, its implementation, use, or mathematical underpinnings; instead, we focus on techniques for reasoning about data abstractions, in both sequential and concurrent contexts.

1. Invariance Reasoning

We focus primarily on proving *safety* properties, i.e. that bad things don't happen. Some examples of bad things we don't want to happen are (unintended) arithmetic overflows, null or wild pointer dereferences, races on nonvolatile data, and wrong results returned from functions. The usual way to prove that bad things don't happen is to define a suitable global ***invariant***: a set of "good" program states such that (1) the system starts out in a good state, and (2) if the system takes a single step from a good state, the resulting state is good. If we can prove that our set of good states is an invariant, and that bad things don't happen in good states, we've proved that bad things don't happen.

Goodness is defined as a big conjunction of properties, defined by ***annotations*** added to the code. Almost all of these properties are of the form "this is true here". For example, here is a small, annotated program that adds two unsigned numbers:

```
unsigned add(unsigned x, unsigned y)
  _(requires x+y <= UINT_MAX)
  _(ensures \result==x+y)
{
  unsigned i=x;
  unsigned j=y;
  while (i>0)
    _(invariant i+j == x+y)
  {
    i--;
    j++;
```

[1]E-mail: Ernie.Cohen@microsoft.com

```
  }
  return j;
}
```

Annotations in VCC are enclosed in the macro `_()`, which is defined to be whitespace to the C compiler. The first identifier after the `_(` (e.g. `requires`) identifies the kind of annotation. Within an annotation, identifiers beginning with `\` (e.g. `\result`) are VCC keywords; the remaining identifiers (e.g. `i`) refer to the surrounding context.

This program has three annotations, each of the form "this is true here":

- `_(requires p)` says that `p` must hold on function entry (after the formals have been bound to arguments); we then say that `p` is a ***precondition*** of the function;
- `_(ensures p)` says that `p` must hold on return from the function (`\result` representing the value returned, if any); we then say that `p` is a ***postcondition*** of the function;
- `_(invariant p)`, when used to annotate a loop, says that `p` must hold every time control reaches the top of the loop (before evaluating the loop test); `p` is what is called a ***loop invariant***.

Of course an appropriate definition of goodness implicitly includes what should be true at other control points as well. For example, in the state after the first statement in the loop, goodness should imply `i+1+j == x+y`, and in the state just after the loop it should imply `i+j == x+y && !(i > 0)`. In addition, there are certain implicit parts of the invariant that can be deduced without explicit annotation; for example, since `x` and `y` are not changed in the function body, VCC can automatically deduce that they always have the value they had on entry to the function. More generally, any variable that isn't updated in the loop has the same value that it had just before loop entry.

We can reason about this function as follows. It's easy to see that the loop invariant holds when control reaches the loop. Inside the loop, We have to check that the operations on `i` and `j` do not overflow or underflow. We know that `i--` doesn't underflow because of the loop test. We know that the increment to `j` doesn't overflow because the loop invariant, combined with the function precondition and the fact that `x` and `y` aren't modified, implies that `i+j <= UINT_MAX` on entry to the loop, so since `i > 0`, `j < UINT_MAX`. Next, we have to show that the loop invariant holds when control transfers back to the top of the loop; this is true because we added to `j` the same amount that we subtracted from `i`. Finally, on loop exit, we know (from the loop invariant and the fact that we exited the loop) that `j == x+y`, so the `return` statement satisfies the function postcondition.

The annotations that appear before the function body constitute the ***specification*** or ***contract*** of the function; in a larger program, these are typically be put on the declaration of the function (in a header file). The function gets to assume that its preconditions hold on function entry, and is obligated to establish the postconditions on every return from the function. Dually, a caller of the function is obligated to establish the preconditions on function entry, and gets to assume the postconditions when the function returns. For example, consider the following function that calls `add`:

```
unsigned add3(unsigned x, unsigned y, unsigned z)
  _(requires x+y+z <= UINT_MAX)
  _(ensures \result == x+y+z)
```

```
{
  unsigned i = add(x,y);
  return  add(i,z);
}
```

We can reason about this function as follows. First, the precondition, along with `z >= 0` (because `z` is unsigned), implies `x+y <= UINT_MAX`, so the precondition of the first call to `add` is satisfied. We can therefore assume the postcondition of the call when it returns, so after the assignment to `i` we know that `i == x+y`. On the second call, since `i==x+y`, we know that `i+z <= UINT_MAX` (thanks to the precondition), so the second call to `add` satisfies its preconditions. Finally, the return from second call to `add` guarantees that `\result == i+z`, and since `i==x+y`, this gives us the postcondition of the function.

Verification tools or methodologies that use contracts as the only interface between functions and their callers are said to be (function-)***modular***. Modular verification has a number of important practical advantages:

- Each function can be verified separately, which means that verification scales well with program size, and can be easily paralellized.
- If you change only the body of a function of a verified program, you only have to reverify that function body; the rest of the program is guaranteed to still verify.
- Irrelevant detail is hidden from the verification engine, making reasoning more efficient.
- The specification of a function provides precise documentation for the function, so that users can just read the specification, without having to read the code. Moreover, verification guarantees that this documentation is up to date.
- You can verify code that calls a function as soon as the specification of the function is written, without having to implement the function first, so a function and its callers can be implemented in parallel. This is particularly useful for functions implemented in hardware; applications can be written and verified before the hardware even exists.

Note that we have not proved that these functions terminate, only that if they return, they give the right answer. While VCC can prove termination, it is not central to the subject of these notes, so we will not go into the details; interested readers are referred to the VCC documentation.

2. Ghost Data and Code

Consider the following function that computes remainder of integer division by repeated subtraction:

```
unsigned mod(unsigned x, unsigned d)
  _(requires d > 0)
{
  unsigned r = x;
  unsigned q = 0;
  while (r >= d)
    _(invariant x == r + q * d)
  {
    r -= d;
    q += 1;
  }
  return r;
}
```

(We will give this a proper postcondition in a moment.) In this function, we don't really care about the quotient `q`, just the remainder `r`. However, `q` is still useful, because it allows us to write a suitable loop invariant. We would like to hide `q` from the compiler, but keep it around for reasoning. We do this by declaring `q` and the code that updates it as `ghost`:

```
unsigned mod(unsigned x, unsigned d)
  _(requires d > 0)
{
  unsigned r = x;
  _(ghost unsigned q = 0;)
  while (r >= d)
    _(invariant x == r + q * d)
  {
    r -= d;
    _(ghost q += 1;)
  }
  return r;
}
```

We reason about ghost code as if it was actually executed, even though it is hidden from the compiler. The reason that this is sound is that the program with the ghost code simulates the program without the ghost code. To make sure that this is the case, we check that each piece of ghost code terminates and that it doesn't update the concrete (non-ghost) state. We can extend `q` for use in the specification by declaring it as an output parameter of the function:

```
unsigned mod(unsigned x, unsigned d _(out unsigned q))
  _(requires d > 0)
  _(ensures x == \result + q * d && \result < d)
```

(A call to `mod` has the form `mod(e0,e1 _(out v))` where `v` is a ghost variable.)

You might wonder why we would use a ghost variable instead of just existentially quantifying over `q` wherever it appears. Using a ghost variable has several advantages:

- It saves the annotational clutter of having to existentially quantify the variable in each assertion.
- It saves the theorem prover from having to find a suitable instantiation for the variable when trying to prove the assertion.
- Existential quantification doesn't distribute through conjunction. This means that if we want to decompose a formula into conjuncts that are maintained by different parts of the program, we have to use a shared ghost variable, rather than existential quantification.

Note that in addition to ordinary C types, ghost data can use other mathematically well-defined types, since the data doesn't have to be implemented at runtime. VCC ghost types include unbounded types like natural numbers and integers, map types, and even a state type (which can be viewed as a mapping from memory addresses to arbitrary values, including states). Similarly, ghost code can use unexecutable program constructs such as unbounded quantification.

3. The Heap

So far, we've considered functions that use only "purely local" variables (local variables and parameters to which the address operator is never applied). These variables have the property that they cannot alias with other variables or be referenced by pointers. Purely local variables are easy to reason about, because in a function with purely local variable `i`, the value of `i` can be changed only by a syntactically identifiable assignment to `i`. However, most C programs have visible side effects through the heap (by which we mean any memory that is not purely local). Moreover, it is only through the heap that concurrent threads can interact. To understand how we specify and verify functions that operate on the heap, we have to describe how the heap is organized in VCC, and how heap memory is shared between functions and threads.

The state is given by the state of a fixed set of ***objects***, each with a number of fields, some of which are ghost. For a given program, this set of objects is fixed; for example, it doesn't change when memory is allocated or deallocated. The objects are disjoint; changing a field of one object doesn't change fields of other objects. We represent an object as a (typed) C pointer to the object. For most purposes, an object can be thought of as an instance of a C `struct` type; primitive types like `int`s, pointers, and arrays of primitives, are fields of objects, but are not themselves objects. For example, for each user-defined `struct` type, there is an object of that type for every properly aligned address where an object of that type could be placed. (There are some artificial objects introduced to contain disembodied bits of memory of primitive types, such as global variables or local variables whose address is taken.)

Each object has a ghost field `\valid` that says whether it is one of the "real" objects of the current state. At each step, we allow fields of invalid objects to change arbitrarily. Moreover, a global invariant maintained by any verified program is that concrete fields of `\valid` objects do not alias (when laid out in physical memory). These properties guarantee that executions of the real program (reading and writing physical memory) simulate executions of the ideal program (operating on objects).

Each object has a Boolean field `\closed` and a field `\owner` (a pointer to an object, which might be a thread). We say an object `o` is said to ***own*** an object `o'` iff `o'->\owner==o`. Verification implies the global invariants that `\closed` objects are `\valid`, and that open (non-closed) objects are owned by threads. In the context of a thread, we say that an object is `\mutable` iff it is owned by the thread and not `\closed`, and `\wrapped` iff it is owned by the thread and `\closed`. A field of an object is said to be `\mutable` iff the object itself is `\mutable`. (A pointer to a primitive implicitly includes the object of which it is a field, so such pointers really represent fields of particular objects rather than just primitive memory locations.) Only threads can own open objects, and only the thread owning an object can ***wrap/unwrap*** the object (make it closed/open).

A memory access can be either ***sequential*** or ***atomic***. A sequential read or write might be broken up into a number of steps, and/or reordered with respect to other sequential memory operations, by a compiler that implicitly assumes that the memory is not being changed by other threads. An atomic read or write is done as part of an action that is guaranteed to be atomic by the underlying platform, and must appear inside an explicitly declared atomic action. (We won't be getting to these for a while.) An atomic read or write can only access mutable fields or fields of objects that are known to be closed, and can only modify a field of a closed object if the field is marked `volatile`. A sequential write is allowed only to a field that is mutable (and satisfies an additional condition). A sequential read is allowed if the field is mutable or is a nonvolatile field of an object that is known to be closed; by default, VCC tries to prove that the field is `\thread_local`, which means that the field is either mutable, a nonvolatile field of an object that is wrapped, or owned by a wrapped object, or owned by an object owned by a wrapped object, etc.

Here's an example of a function that reads sequentially from the heap:

```
size_t lsearch(unsigned *a, size_t len, unsigned v)
  _(requires \thread_local_array(a,len))
  _(ensures \forall size_t i; i < \result ==> a[i] != v)
  _(ensures \result < len ==> a[\result] == v)
{
  for (size_t i=0; i < len; i++)
    _(invariant \forall size_t j; j<i ==> a[j] != v)
  {
    if (a[i] == v) return i;
  }
  return len;
}
```

This example makes use of a few new annotations. `\thread_local_array{a,len}` means that the `unsigned`s `a[0],a[1], ... a[len-1]` are all thread-local; `==>` is logical implication; `p ==> q` is essentially the same as `!p || q`, except that `==>` has lower precedence than the built-in C operators. Similarly, `<==` is reverse implication, and `<==>` is iff and only iff. `\forall T v; p` is true iff `p` is true (i.e., nonzero) for every `v` of type `T`.

Note that the reasoning in this program depends on the fact that the elements of `a` aren't changing. (A change to `a[j]`, where `j<i`, could break the loop invariant.) In fact, VCC assumes that no other threads even run while it executes purely sequential code; we'll see why that pretense is sound later.

We mentioned above that writing a field sequentially requires a stronger condition than mutability. To see why, consider the following function that calls `lsearch`:

```
void test() {
  unsigned a[10];
  a[3] = 0;
  lsearch(a,10,3);
  _(assert a[3] == 0)
}
```

We want this function to verify; that is, in the absence of further annotations, a caller should be able to assume that thread-local data isn't changed across a function call. Note that this doesn't prevent `lsearch` from declaring and updating local variables, or allocating and modifying new memory; it only stops the function from updating data that was thread-local to the caller on function entry, so it does stop `lsearch` from modifying `a[3]`.

To be more precise, if an object is wrapped and not mentioned in the writes clause, then that object remains closed for the duration of the function so the entire sequential domain rooted at that object is unchanged by the call. If a field of a mutable object is not mentioned in the writes clause, it is not changed by the call. To enforce these rules, a function can write to a field of an object only if (1) the field is mentioned in the writes clause of the function, or (2) the object became mutable after entry to the function. Similarly, a function can wrap or unwrap an object only if it is mentioned in a writes clause, or if the object became mutable or wrapped after entry to the function.

4. Data Invariants

Recall that we said that we reason about a program by defining a big invariant for the whole program, and checking that it is preserved by each state change. We've talked about invariants that link conditions to program control points. But other invariants are independent of program control, and are more naturally expressed as invariants on data.

We attach data invariants to objects, in particular, to user-defined `struct` types. We guarantee that closed objects satisfy their invariants by (1) checking the invariant of an object when it is wrapped (i.e., when it goes from open to closed), (2) checking the invariants of a closed object whenever one of its (volatile) fields is modified (in an atomic action), and (3) checking that all object invariants are "admissible", which essentially shows that the invariants of a closed object cannot be broken by updates to other objects.

Here is an example of a data structure that represents a double-precision unsigned number:

```
#define ONE   ((\natural) 1)
#define RADIX (UINT_MAX + ONE)
#define DBL_MAX  (UINT_MAX + UINT_MAX * RADIX)

typedef struct Double {
  // abstract value
  _(ghost \natural val)

  // implementation
  unsigned low;
  unsigned high;

  //coupling invariant
  _(invariant val == low + high * RADIX)
} Double;
```

Each `Double d` functions as an implementation of a natural number (with value no greater than `DBL_MAX`). The field `d.val` gives its abstract value; the coupling invariant connects the abstract value with the concrete representation.

The typical way to "construct" an object is to initialize its fields and wrap the object. For example, here is the code to initialize a new `Double`:

```
void dblNew(Double *d)
  _(writes \extent(d))
  _(ensures \wrapped(d) && d->val == 0)
{
  d->low = 0;
  d->high = 0;
  _(ghost d->val = 0)
  _(wrap d)
}
```

Here, `\extent(d)` is `d` along with all of its fields. When a field is included in a writes clause, it implicitly requires that the field is mutable, so the writes clause implicitly requires that `d` is mutable.) The ghost statement `_(wrap d)` closes `d`, checking its invariant. Note that the postcondition of the function is stated in terms of the abstract value of the object, without mentioning the concrete implementation. This allows swapping the implementation out for another type providing essentially the same contracts, without breaking clients. Note that had we chosen another value for, say, `d->low`, the verification of `_(wrap d)` would fail because the coupling invariant would not be satisfied.

The destructor of a data structure essentially reverses the pre and post-conditions:

```
void dblDestroy(Double *d)
  _(requires \wrapped(d))
  _(writes d)
  _(ensures \extent_mutable(d))
{
  _(unwrap d)
}
```

A typical operation of the data type maintains (i.e., `requires` and `ensures`) that the object is wrapped, writes the object, and has pre/postconditions written in terms of the abstract value of the object:

```
void dblInc(Double *d)
  _(maintains \wrapped(d))
  _(writes d)
  _(requires d->val + 1 < DBL_MAX)
  _(ensures d->val == \old(d->val) + 1)
{
  _(unwrapping d) {
    if (d->low == UINT_MAX) {
      d->high++;
      d->low  = 0;
    } else {
      d->low++;
    }
    _(ghost d->val = d->val + 1)
  }
}
```

5. Model Fields

We've seen that one way to express abstract state and its connection to concrete state is to keep the abstract state as a ghost. An alternative is to define the abstract state as a function of the concrete state; such functions are sometimes called ***abstraction functions***. In some OO methodologies, these functions can be (syntactically) presented as fields; in this case, they are usually called ***model fields***.

We can rewrite the `Double` type using model fields as follows:

```
typedef struct Double {
  unsigned low;
  unsigned high;
} Double;

_(def \natural dblVal(Double *d) {
  return d->low + d->high * RADIX;
})
```

Here, `_(def dblVal(...)...)` defines a pure ghost function whose contract is derived directly from the code for the function. A ***pure*** function is one that doesn't modify the state; only pure functions can be used in assertions, preconditions, postconditions, and invariants.

```
void dblNew(Double *d)
  _(requires \extent_mutable(d))
  _(writes \extent(d))
  _(ensures \wrapped(d) && dblVal(d) == 0)
{
  d->low = 0;
  d->high = 0;
  _(wrap d)
}

void dblDestroy(Double *d)
  _(requires \wrapped(d))
  _(writes d)
  _(ensures \extent_mutable(d))
{
  _(unwrap d)
}

void dblInc(Double *d)
  _(maintains \wrapped(d))
  _(writes d)
  _(requires dblVal(d) < DBL_MAX)
  _(ensures dblVal(d) == \old(dblVal(d)) + 1)
{
  _(unwrapping d) {
    if (d->low == UINT_MAX) {
      d->high++;
      d->low  = 0;
    } else {
      d->low++;
    }
  }
}
```

Note that the only changes are that (1) we omit the ghost field giving the abstract value, as well as updates to that field; and (2) the abstraction function `dblVal` is used in place of the ghost field in writing contracts.

Model fields have some substantial advantages and disadvantages when compared to ghost fields. Some advantages of model fields:

- As we've seen above, using model fields can reduce the amount of annotation needed.
- Updating a piece of data invisibly updates any model fields that depend on it. This is particularly useful in concurrent settings, where the model field might have to be updated in the same atomic action as the representation, but might not be in scope of the code that updates the representation.

Model fields also have some disadvantages:

- They can only substitute for ghost state in those cases where the abstract state can be determined from concrete state, independent of the history. A typical example where this cannot be done is a ring buffer or sliding window, where the abstract value most convenient for reasoning is a sequence of all the values produced, not just those remaining in the buffer.
- The connection between the model field and the concrete state normally holds only when the data is in a consistent state (e.g., when the object containing them is closed). In a concurrent setting, we often want to maintain an abstract state at all times (e.g., by defining it to remain unchanged while the object is open). This generally cannot be done with a model field. (Note, this objection, as well as the last, might be mitigated by defining an extended sort of model field whose value is defined as a recursive function of the state history.)
- Sometimes, the function needed to compute the model field from the data is complex enough that it is hard to reason about. A typical example is the abstract value of an inductive data structure, where the model field typically has a recursive definition. On the other hand, it might be relatively easy to update ghost data in a first-order way; the update itself can be viewed as an inductive proof of a lemma about the model field.
- Because ghost fields are explicitly updated, they are governed by the framing rules (writes clauses), making it easy to detect syntactically the vast majority of cases where the field is known not to change. Conversely, because model fields

need not be explicitly updated or declared in writes clauses, it is sometimes very difficult to guess or prove that a model field has not changed, particularly if it is defined recursively.

- Because ghost fields are explicit object fields, any change to the field causes of check of the object invariant. Thus, an invariant of the object is checked only for those updates that actually occur in the code. Conversely, if the invariant makes use of a model field, one must prove that the invariant cannot be broken by updates that change the model field without changing other fields of the object.

6. Nested Ownership

A good methodology allows user-defined types like `Double` to be used much like built-in types like `unsigned`. Here is an example of a `Quad` type defined on top of `Double`, much as `Double` was defined on top of `unsigned`. The most important difference is that while primitive fields are not objects (and therefore do not have independent owners), fields of compound types are considered independent objects. However, we can get a similar effect by having an object own those "child" objects that serve as part of its representation; this works whether the children are contained within the object struct or not.

```
#define DRADIX (DBL_MAX + ONE)
#define QUAD_MAX (DBL_MAX + DBL_MAX * DRADIX)

typedef struct Quad {
  // abstract value
  _(ghost \natural val)

  Double low;
  Double high;
  _(invariant \mine(&low) && \mine(&high))

  //coupling invariant
  _(invariant val == low.val + high.val * DRADIX)
} Quad;
```

The annotation `\mine(x)` is shorthand for `x->\owner == \this`. The first invariant says that the `Quad` owns the two `Double`s contained inside it. (This invariant has to hold only when the `Quad` is closed; since only threads can own open objects, this implies that the two `\Double`s are also closed.) Note that the couping invariant is stated in terms of the abstract values of the subobjects.

The constructor is much like the constructor for `Double`s, except that its `low` and `high` components have to be explicitly constructed:

```
void quadNew(Quad *q)
  _(requires \extent_mutable(q))
  _(writes \extent(q))
  _(ensures \wrapped(q) && q->val == 0)
{
  dblNew(&q->low);
  dblNew(&q->high);
  _(ghost q->val = 0)
  _(wrap q)
}
```

Every object `o` has a ghost field `o->\owns` that gives the set of objects owned by `o` when it is closed. When a thread wraps `o`, it checks that all objects of `o->\owns` are wrapped, and transfers ownership of these objects to `o`. Conversely, when unwrapping an object, it takes ownership of all objects that were owned by `o`. By default, `o->\owns` is determined by the invariants of the form `\mine(o')` in the type definition of `o`. In the case of a `Quad`, these invariants say that for a `Quad q`, `q->\owns` is the set `{q->low, q->high}`. Thus ownership of these objects transfers to `q` when it is wrapped.

The reason that it is important for `q` to own `q->low` and `q->high` is that without this ownership invariant, the coupling invariant of `q` could be broken by the owner of `q->low` unwrapping it and changing `q->val`. With the ownership invariant, we know that `q->low` and `q->high` must remain closed (since non-thread objects can only own closed objects); since a nonvolatile field of a closed object cannot change, `q->low.val` and `q->high.val` cannot change, so the couppling invariant cannot be broken.

Destruction is analogous:

```
void quadDestroy(Quad *q)
  _(requires \wrapped(q))
  _(writes q)
  _(ensures \extent_mutable(q))
{
  _(unwrap q)
  _(unwrap &q->low)
  _(unwrap &q->high)
}
```

Note that the contracts on construction and destruction are identical to the corresponding contracts from `Double`. In particular, only `q` has to be mentioned in the `writes` clause, because the thread obtained ownership of `q->low` and `q->high` after the function was called.

Operations on `Quad`s look just like operations on `Double`s (except for allowing the larger range represented by `Double`s):

```
void quadInc(Quad *q)
  _(maintains \wrapped(q))
  _(writes q)
  _(requires q->val + 1 < QUAD_MAX)
  _(ensures q->val == \old(q->val) + 1)
{
  _(assert \inv(&d->low))
  _(unwrapping q) {
    if (isDblMax(&q->low)) {
      dblInc(&q->high);
      dblZero(&q->low);
    } else {
      dblInc(&q->low);
    }
    _(ghost q->val = q->val + 1)
  }
}
```

We could likewise implement `Quad`s with model fields instead of ghost fields:

```
typedef  struct Quad {
    Double low;
    Double high;
    _(invariant \mine(&low) && \mine(&high))
} Quad;

_(def \natural qval(Quad *q) {
    return dblVal(&q->low) + dblVal(&q->high) * DRADIX;
})
```

7. Maps

Of course most interesting values cannot be conveniently described in terms of natural numbers or integers; but most can be described naturally using maps. In VCC, maps look much like arrays (except that the entire map constitutes a single primitive value).

Here's a simple example of (small) sets of `unsigned`s implemented using arrays:

```
typedef unsigned Val;

typedef struct Set {
  // abstract value of the set (a function from Val to \bool)
  _(ghost \bool mem[Val])

  // concrete representation
  Val data[SIZE];
  size_t len;
  _(invariant len <= SIZE)
  _(invariant \forall Val v;
      mem[v] <==> \exists size_t j; j < len && data[j] == v)
} Set;

void setNew(Set *s)
  _(requires \mutable(s))
  _(writes \extent(s))
  _(ensures \wrapped(s))
  _(ensures \forall Val v; !s->mem[v])
{
  s->len = 0;
  _(ghost s->mem = \lambda Val v; \false)
  _(wrap s)
}
```

Here, `\lambda T v; e`, where `e` is of type `T'`, is the map from `T` to `T'` that maps `v` to `e`.

```
_(pure) BOOL setMem(Set *s, Val v)
_(requires \wrapped(s))
_(reads s)
_(ensures \result == s->mem[v])
{
  for (size_t i = 0; i < s->len; i++)
    _(invariant \forall size_t j; j < i ==> s->data[j] != v)
  {
    if (s->data[i] == v) return TRUE;
  }
  return FALSE;
}

BOOL setAdd(Set *s, Val v)
  _(maintains \wrapped(s))
  _(writes s)
  _(ensures \forall Val x;
      s->mem[x] == \old(s->mem[x]) || (\result && x == v))
{
  if (s->len == SIZE) return FALSE;
  _(unwrapping s) {
    s->data[s->len] = v;
    s->len++;
    _(ghost s->mem[v] = \true)
  }
  return TRUE;
}
```

Again, we could instead write the same type using model fields instead:

```
_(typedef \bool valSet[Val])

typedef struct Set {
  Val data[SIZE];
  size_t len;
  _(invariant len <= SIZE)
} Set;

_(def valSet setMem(Set *s) {
  return \lambda Val v;
    \exists size_t i; i < s->len && s->data[i] == v;
})
```

8. Existential Quantification and Explicit Witnesses

The coupling invariant (or model field) for `Set` in the last example used existential quantification. We discussed earlier some of the advantages of replacing existential quantification with explicit witnesses. In this case, we can eliminate the existential quantification by keeping track, for each element of the set, an index where that element appears in the array of elements.

```
typedef struct Set {
  _(ghost \bool mem[Val])  // abstract value of the set
  Val data[SIZE];
  size_t len;
  _(invariant len <= SIZE)

  // explicit witness
  _(ghost size_t idx[Val])

  _(invariant \forall size_t i; i < len ==> mem[data[i]])

  // witness for each abstract member
  _(invariant \forall Val v;
      mem[v] ==> idx[v] < len && data[idx[v]] == v)
} Set;

BOOL setAdd(Set *s, Val v)
  _(maintains \wrapped(s))
  _(writes s)
  _(ensures \forall Val x;
      s->mem[x] == \old(s->mem[x]) || (\result && x == v))
{
  if (s->len == SIZE) return FALSE;
  _(unwrapping s) {
    s->data[s->len] = v;
    _(ghost s->mem[v] = \true)
    // update the witness
    _(ghost s->idx[v] = s->len)
    s->len++;
  }
  return TRUE;
}
```

Here, `_(unwrapping s)B` is syntactic sugar for `_(unwrap s)B _(wrap s)`.

9. Inductive types

.

Sometimes, what you really want to do is functional programming. VCC lets you define inductive datatypes in ghost code, much like modern functional languages like Haskell or ML (forgive the C-like syntax):

```
_(datatype Tree {
  case Leaf(unsigned val);
  case Node(Tree left, Tree right);
})
```

This defines a the type if binary trees with `unsigned`s on the leaves; for example, `Node(Node(Leaf(1),Leaf(2)), Node(Leaf(3),Leaf(4)))` is a balanced binary tree with 4 leaves. Note that inductive types are primitive ghost types.

We can write recursive functions on inductive types using pattern matching:

```
// functional programming
_(def Tree reverse(Tree t)
{
  switch (t) {
    case Leaf(val) : return t;
    case Node(l, r): return Node(reverse(r),reverse(l));
  }
})
```

Note that `reverse`, like all ghost functions, is implicitly checked for termination; we won't go into the details here.

Functional program reasoning follows a somewhat different style from imperative program reasoning. In imperative programming, you try to abstract data to first-order abstractions, and write function specifications in terms of first-order updates on these abstractions. In functional programming, you normally keep things in terms of recursive functions, exposing their recursive definitions rather than a first-order abstraction. To reason about such functions, you reason by induction over data, rather than inducting over time (with invariants).

You can prove theorems by induction just by writing pure functions with suitable postconditions. For example:

```
_(def void revRev(Tree t)
  _(ensures reverse(reverse(t)) == t)
{
  switch (t) {
    case Leaf(v): return;
    case Node(l, r): revRev(l); revRev(r); return;
  }
})
```

Note that in the recursive case, we make explicit calls to the lemma being proved. These calls provide the needed inductive instances needed to prove the theorem (along with the recursive definition of `Reverse`).

The main use of inductive types is for those cases where you are really doing functional programming; you reason in terms of the inductive types (using induction, as above), and show that the concrete implementation simulates the functional computation. For example, we could implement trees as follows:

```
typedef _(dynamic_owns) struct Tr {
  _(ghost Tree val)
  BOOL isLeaf;
  Tr *l,*r;
  unsigned v;
  _(invariant isLeaf  ==> val == Leaf(v))
  _(invariant !isLeaf ==> \mine(l) && \mine(r))
  _(invariant !isLeaf ==> val == Node(l->val, r->val))
} Tr;
```

The `_(dynamic_owns)` annotation on the type `Tr` says that the owns set is managed explicitly; this is needed because the set of objects owned by a `Tr` can change and must be managed explicitly; for example,

```
void trCons(Tr *t, Tr *l, Tr *r)
  _(requires \extent_mutable(t) && \wrapped(l) && \wrapped(r))
  _(writes \full_extent(t), l, r)
  _(ensures \wrapped(t) && t->val == Node(l->val, r->val))
{
  t->l = l;
  t->r = r;
  t->isLeaf = FALSE;
  _(ghost t->val = Node(l->val, r->val);)
  _(ghost t->\owns = {l, r})
  _(wrap t)
}
```

The obvious concrete implementation of `reverse` below fails to verify; can you find the bug?

```
void rev(Tr *t)
  _(maintains \wrapped(t))
  _(writes t)
  _(ensures t->val == reverse(\old(t->val)))
{
  if (t->isLeaf) return;
  _(unwrapping t) {
    Tr *tmp =t->l;
    t->l = t->r;
    t->r = tmp;
    rev(t->l);
    rev(t->r);
    _(ghost t->val = reverse(t->val))
  }
}
```

The bug is that if `t->l` and `t->r` happen to point to the same tree, the sole child will be reversed twice, resulting in the original tree being returned. There are two natural ways to fix this. First, you can add an invariant to `Tr` that says that children don't alias:

```
  _(invariant isLeaf || l != r)
```

Second, you can fix the code so that only one reversal is performed if `l==r`. These are left as exercises.

10. Inductive Data Structures

In the last section, we saw one approach to handling inductive data structures: there is a concrete node type for each constructor (or, in the example above, one type representing all of the constructors), a constructor object owns the child data structures, and each instance of a constructor stores the abstract value of the tree beneath it. This works fine for sequential programs where the computation recurses down the structure (`Rev` is a good example) , but it is a problem for programs that iterate down the structure, and especially for programs that mutate in the middle of the structure. This is because the structure has to be unwrapped from the outside-in, and rewrapped in the opposite direction. Moreover, this structure is even more problematic in the concurrent setting, where the whole structure has to remain wrapped; updating part of the structure requires simultaneously updating all of its ancestors.

A more flexible design is to make the constructor nodes purely structural (with essentially no invariants), and to put both the abstract value and all structural invariants in a ghost "master" object that owns all of the nodes of the particular recursive structure. The downside of this arrangement is that the invariants of the master have to quantify over all of the nodes, which makes the reasoning a bit more involved. The upside is that all of the nodes can be updated with a single map update.

There are several possible options for representing the abstract value in the master. If the structure is essentially linear, then one attractive representation is to represent the structure as a map from naturals to nodes. For example, here is an implementation of a linked list:

```
typedef struct Node Node, *PNode;

typedef struct Node {
  PNode nxt;
} List;

typedef _(dynamic_owns) struct List {
  PNode hd;
  _(ghost \natural len)
  _(ghost PNode val[\natural])
  _(invariant val[0] == hd)
  _(invariant val[len] == (PNode) NULL)
  _(invariant \forall \natural i; {val[i]}
      i < len ==> \mine(val[i]))
  _(invariant \forall \natural i;
      i < len ==> val[i] && val[i]->nxt == val[i+1])
} List;
```

(In the next-to-last invariant, `{val[i]}` is a hint to the deductive engine telling it when to instantiate the universally quantified variables of the invariant; this is outside the scope of this paper.)

With this representation, some operations are easily specified, such as finding the n'th element of the list, reversing the list, adding or removing individual elements, appending lists, and so on. Note, however, that some functions are not so easily specified; for example, a function that deletes all nodes satisfying some condition has to be specified with a recursive function, because there is no first-order expression that gives the n'th element of the result.

For treelike structures, it is possible to generalize the abstract value to a map from paths to nodes, though in practice this is not worthwhile (mainly because tools like VCC use an SMT solver as a reasoning engine, and SMT solvers are very good at reasoning about numbers but not more general sorts of partial orders).

A second possibility is to express the abstract value as a partial order giving the reachability relation on its nodes. (This can be done for arbitrary acyclic data structures.) The reason to keep the reachability relation is that it allows the statement of global structural properties, e.g. that every node of the structure can be reached by walking the list starting from the head.

11. Admissible Invariants

Let's step back for a moment and consider how we are reasoning about programs. Recall that the state is partitioned into objects. (In addition to the objects we've talked about, you can imagine that every function activation is also an object, with fields that give the values of the program counter and local variables.) A state of the world is given by the state of each of these objects. An execution is a sequence of states. A transition is an ordered pair of states; we can think of each consecutive pair of states of an execution as a transition from the first state to the second.

We've given some examples of object invariants. In fact, every object has a "2-state" invariant, an invariant evaluated over a pair of states. (When we talk about an object having multiple invariants, we mean they are all conjoined to form "the" invariant of the object.) We can treat this invariant as a single-state invariant by applying it to the stuttering transition that remains in the prestate. A state is ***good*** if it satisfies the 1-state invariant of every object; a transition is good iff it satisfies every 2-state invariant. An execution is good if each of its states is good and each of its transitions is good. The goal of verification is to show that every execution of the program (starting from a good state) is good.

A transition is ***legal*** if the prestate is not good or the transition satisfies the invariants of all updated objects (i.e., all objects whose state differs in the prestate and poststate of the transition). The invariant of object `o` is ***admissible*** iff, for every legal transition from a good prestate, the transition and the poststate both satisfy the invariant of `o`. We verify a program by showing that every transition of any of its executions is legal, and every object invariant is admissible; it's easy to show by induction on execution length that these imply that every execution is good.

To bring this approach into our C verification context, when we say that object `o` has the invariant `p`, we really mean that it has the invariant `\old(o->\closed)||` `o->\closed ==> p`, i.e. the invariant holds if `o` is `\closed` in either the prestate or the poststate of a transition. In addition, each object has implicit invariants that its nonvolatile fields do not change while the object is closed, as well as invariants related to ownership (e.g., `o->\closed && o′␣\in␣o->owns␣==>␣o′->\closed` `&& o′->\owner␣==␣o`).

As an example, let's next consider the problem of designing a suitable invariant for a doubly-linked list. We want the invariants to say that the predecessor and successor pointers of successive nodes match:

```
typedef struct Node Node, *PNode;

typedef struct Node{
  volatile PNode pred;
  volatile PNode succ;
  _(invariant pred && succ)
  _(invariant pred ==> pred->succ == \this)
  _(invariant succ ==> succ->pred == \this)
} Node;
```

The first invariant, which says that neither the predecessor nor successor fields are `NULL`, is trivially admissible, since it only mentions fields of the object itself; the only way to break such an invariant is to change a field of the object, so a legal transition necessarily satisfies the object's invariant.

However, the second and third invariants are not admissible, because there is nothing that guarantees that the neighboring nodes are closed. For example, if `o->succ == o1` and `o1` is not closed, then the owner of `o1` can change `o1->pred` without checking any invariants, which would break the second invariant of `o`. We can fix this problem by adding invariants that links point to closed nodes:

```
  _(invariant pred ==> pred->\closed)
  _(invariant succ ==> succ->\closed)
```

However, the new invariants are inadmissible, since nothing prevents the owner of `succ` from opening it up. There simplest way to prevent this is by just not allowing closed nodes to be opened:

```
  _(invariant \on_unwrap(\this,\false))
```

Of course this is rather unrealistic - well-designed objects should allow eventual graceful destruction - but we will use this for now.

But this is still not strong enough to make the original invariants admissible. To see why, suppose we are in a state where `o` and `o1` are each other's predecessor and successor, and `o2` is its own predecessor and successor. Consider an atomic action that makes `o1` and `o2` each other's predecessor and successor. This action preserves the invariants of `o1` and `o2`, which are the only objects that are updated. However, it breaks the invariant of `o`, since `o` still lists `o1` as its predecessor and successor.

What we want to say in our invariant is that any time you swing a pointer away from a node, you must make sure to check that the invariants you used to point to. We can write this as an invariant itself:

```
  _(invariant \unchanged(pred) || \inv(\old(pred)))
  _(invariant \unchanged(succ) || \inv(\old(succ)))
} S;
```

These invariants say that if `pred` or `succ` is changed, then we must check that the invariant of the old predecessor/successor in the new state. Here, `\inv(o)` is a special built-in function that holds in a state if the invariant of `o` holds in that state. Since the `\inv` function can be used in object invariants, we have to be careful not to introduce inconsistencies; for example, we don't want to allow a type to have an invariant like `_(invariant !\inv(\this))`. The simplest way to maintain logical consistency (i.e., give a consistent interpretation of the `\inv` function) is to allow `\inv` to occur only with positive polarity in object invariants and in function contracts.

These invariant are themselves necessarily admissible. In fact, any object invariant of the form `\unchanged{f} || p`, where `f` is a field of the object, is necessarily admissible: the invariant necessarily holds for any transition that doesn't change `f`, and a legal transition that changes `f` must satisfy the object invariant anyway.

12. Atomic Actions

We now consider reasoning about concurrent programs. It turns out that concurrent programming techniques are useful even if you are doing sequential programming, because it is sometimes useful to update ghost fields "atomically". Indeed, concurrent programming is not really about atomic updates to the state, which are easy to arrange; it is about safely sharing the state between simultaneous observers. This is just as much a problem for large-scale sequential programming.

At its core, the concurrent programming methodology is quite simple. An atomic action has to be explicitly marked as atomic. Within the action, in addition to sequential reads and writes, there can be reads of fields of closed objects and writes of volatile fields of closed objects; these objects must be explicitly listed in the atomic action annotation. The atomic action must be legal, i.e. it must preserve the invariants of any updated objects.

As a simple example, consider a monotonic counter:

```
typedef struct Counter {
  volatile unsigned val;
  _(invariant \old(val) <= val)
  _(invariant \on_unwrap(\this,\false))
} Counter;
```

A volatile field like `val` can change at any time while its containing object is closed, subject to the condition that such a change preserves the invariant of the counter. (It is useful to imagine that the such updates can be initiated not only by the program itself, but by the environment also.) It is unsafe to access such a field sequentially; for example, a compiler might optimize a sequential read to read the high and low order bits of the counter separately. Thus, such a field should only be accessed using operations for which the platform and compiler guarantee atomicity. For example:

```
void inc(Counter *cnt)
  _(requires \wrapped(cnt))
{
  _(atomic _(read_only) cnt) {
    unsigned x = cnt->val;
  }
  if (x == UINT_MAX) return;
  _(atomic cnt) {
    if (cnt->val == x) cnt->val = x+1;
  }
}
```

This function contains two atomic actions, one to read the counter, the other to update it. In each case, we are accessing a field of a closed object, which must be listed in the atomic action annotation preceding the action itself. In the first action, fields of `cnt` are being read but not updated, so the `_(read_only)` annotation allows us to skip checking the invariant of `cnt`. In the second action, we have to check that the whole atomic action preserves the invariant of `cnt`.

Note that `cnt` is not listed in a writes clause of `inc`. Atomic actions in general do not count as writes. The reason for this is that the atomic update done by this thread could just as well have been done by another thread (or the environment), and we certainly cannot account for their actions in a writes clause. Since the caller is doing something that the environment is allowed to do anyway, there is no point in restricting it via writes clauses.

While this function verifies, VCC will also issue a warning that the atomic action contains more than one access to volatile physical fields. (In fact, such an action is unlikely to be atomic on most platforms.) It is good programming practice to isolate any operation on physical memory that is implemented atomically as a separate function (and indeed, most compilers would require the explicit use of a compiler intrinsic function for any atomic beyond a single volatile access that fits within a machine word). In VCC, you can break out the physically atomic action into a separate inline function, as follows:

```
_(atomic_inline) unsigned cmpXchg(unsigned *loc, unsigned cmp,
    unsigned xchg)
{
  if (*loc == cmp) {
    *loc = xchg;
    return cmp;
  }
  else return *loc;
}

void inc(Counter *cnt)
  _(requires \wrapped(cnt))
{
  _(atomic _(read_only) cnt) {
    unsigned x = cnt->val;
  }
  if (x == UINT_MAX) return;
  _(atomic cnt) {
    cmpXchg(&cnt->val, x, x+1);
  }
}
```

This eliminates the warning.

Finally, we note in passing that the first atomic action can be written using some friendly syntactic sugar:

```
  unsigned x = _(atomic_read cnt) cnt->val;
```

13. Collecting Volatile Information

Suppose we did two consecutive reads of `cnt->val` (in separate atomic actions), returning values `x` and `y`, respectively. We would expect to be able to prove afterwards that `x <= y`, thanks to the invariant of `cnt`. However, in general such reasoning is much like guessing a loop invariant. Thus, just as with loop invariants, VCC makes you spell this reasoning out. To understand how, we have to look at how VCC deals with interference from other threads.

In the presence of interference, a thread knows that a field of an object cannot change if the object is mutable, or if it is a nonvolatile field of an object that is known to remain closed. Moreover, actions of other threads are guaranteed to keep the state good, so all object invariants continue to hold. We can use these properties to capture what we need to know about the two consecutive reads. Suppose we define the following "helper" type:

```
_(typedef struct O {
  Counter *c;
  unsigned x;
  _(invariant c->\closed && x <= c->val)
} O;)
```

The key thing to note is that the invariant of `O` is admissible, thanks to the invariants of `Counter`. We can now use one of these objects as follows:

```
void test(Counter *cnt)
  _(requires \wrapped(cnt))
{
  unsigned x = _(atomic_read cnt) cnt->val;
  _(ghost O o)
  _(ghost o.c = cnt)
  _(ghost o.x = x)
  _(wrap &o)

  unsigned y = _(atomic_read cnt) cnt->val;
  _(assert \inv(&o))
  _(assert x <= y)

  _(unwrap &o)
}
```

To understand why this works, it's important to realize that VCC pretends that a thread takes a scheduler boundary only just before performing an atomic action. (This pretense is justified, but we won't discuss why here.) Thus, other threads (or the environment) can only change the state just before the two `atomic_read`s. Thus, between the first read and just before the second read, `cnt->val` isn't changing; this is why we can safely wrap `&o` (since at that point we know that `o.x == cnt->x`.

Just before the second read, we forget everything we knew about `cnt->val`. However, since `&o` is wrapped, we know that its invariant still holds, and that `o.x` and `x` haven't changed. Thus, after the second read, we know that `x==o.x`, `o.x <= cnt->val`, and `cnt->val == y`, from which we can deduce the asserted `x <= y`. Afterwards, `o` has served its purpose, so we can destroy it.

Ghost objects like o are so useful that VCC provides syntactic sugar for constructing ghost objects like o in the example above, without having to explicitly declare types like O. These special ghost objects are called ***claims***.

```
void test(Counter *cnt)
  _(requires \wrapped(cnt))
{
  unsigned x = _(atomic_read cnt) cnt->val;
  _(ghost \claim c =
      \make_claim({}, cnt->\closed && x <= cnt->val))

  unsigned y = _(atomic_read c) cnt->val;
  _(assert x <= y)
}
```

The claim c can be thought of as an object whose invariant is given in the second argument of \make_claim. The claim object has a local variable recording a copy of the state at the time the claim was created, and free variables in the claim invariant (like x and cnt) are interpreted in that saved state.

Of course, to have sharing, threads must be able to operate on objects that they don't own. This is okay, because concurrent access only requires the object to be closed, not wrapped. The easiest way to do this is for the thread to have a claim that claims that the object is closed. We could modify the function to take such a claim as an argument as follows:

```
void test(Counter *cnt _(ghost \claim cl))
  _(always cl, cnt->\closed)
{
  unsigned x = _(atomic_read c, cnt) cnt->val;
  _(ghost c = \make_claim({}, cnt->\closed && x <= cnt->val))
  ...
```

The precondition `_(always cl, p)` is a macro that translates to requiring and ensuring that c is wrapped, and that cl claims p (i.e., that in any state where cl is closed and the invariant of cl holds, p also holds). Note that the atomic read now lists two objects: the first, c is included because its invariant is used to prove that the second object, cnt, is closed.

Of course the invariant that counters cannot ever be unwrapped is rather unrealistic for real objects that must be cleaned up (eventually). In that case, the invariant of claim c is no longer admissible. However, we can keep cnt closed by keeping cl closed, so it would suffice for c to keep cl closed. To implement this idea, each claim has an additional field, a set of "claimed objects" that are guaranteed to remain closed as long as the claim is closed; the first argument to \make_claim gives the set of claimed objects for the claim being created. Claimed objects must be of types declared as `_(claimable)`; objects of such types have an additional field \claim_count that keeps track of how many claims claim the object, and unwrapping such an object requires proving that its claim count is 0. In particular, claims are themselves claimable.

The claim count of an object o obviously has to be volatile (since it can change while o is closed). Because o can be unwrapped only when `o->\claim_count == 0`, `o->\owner` must somehow control this count in order to ever be able to safely unwrap the object. This means that the `o->\claim_count` behaves much like the field `o->\owner`, in that it can be changed only with the "approval" of `o->\owner`. We will see how this is done in general a bit later, but the important thing for now is that to change the claim count of a wrapped object, the object must be writable.

Using dependent claims, we can rewrite the counter update example to make the new claim depend on the old one:

```
typedef struct Counter {
  volatile unsigned x;
  _(invariant \old(x) <= x)
} Counter;

void test(Counter *cnt _(ghost \claim cl))
  _(always cl, cnt->\closed)
  _(writes cl)
  _(ensures \unchanged(cl->\claim_count))
{
  unsigned x = _(atomic_read cl,cnt) cnt->x;
  _(ghost \claim c =
      \make_claim({cl}, cnt->\closed && x <= cnt->x))
  unsigned y = _(atomic_read c,cnt) cnt->x;
  _(assert x <= y)
  _(ghost \destroy_claim(c,{cl}))
}
```

The first argument of `\make_claim` lists the claimed objects of the new claim (i.e., the set of objects whose claim count is being incremented), while the second argument to `\destroy_claim` lists the same set (i.e., the set of objects whose counts are being decremented). Note that we have to report `cl` in the writes clause because the function changes `cl->\claim_count`.

14. Approval

Consider an object o with a nonvolatile field f. Invariants of o can freely talk about o.f since o.f doesn't change while o is closed. If we then replace the field f with an object fOb with a nonvolatile field val that gives its value, and add to o an invariant that it owns fOb, we have seen that we can still use val freely in the invariant of o. Moreover, unwrapping o transfers to the owner of o similar control over f and val.

Now consider the situation where f is a volatile field. Invariants of o can still freely talk about f, because any change to f requires a check of the invariant of o. However, if we replace f with an object fOb with a volatile field val, where o owns fOb, we cannot in general freely use fOb.val in invariants of o because such invariants might be inadmissible. This is because ownership of an object does not convey any special rights over its volatile fields. If we want the invariants of the owner of fOb to be checked on every update to val, we can make this an explicit invariant of fOb of the form

```
  _(invariant \unchanged(val} || \inv2(\this->\owner))
```

Here, `\inv2(o)` means the two-state invariant of object `o`. (There is similarly a function `\inv(o)` that gives the single-state invariant of `o`.) `\inv2` (and `\inv`) can be used in assertions and object invariants, but can appear only with positive logical polarity (to avoid introducing logical inconsistency). The invariant above says that on any change to `val`, the invariant of the owner must be checked. Note that this invariant is trivially admissible.

Now, suppose the owner of `fOb` happens to be a thread. What does it mean to check the invariant of a thread? When reasoning about a function, VCC implicitly constructs from the annotations on the program a suitable invariant capturing what a thread executing the function should "know" at each control point; we can think of the invariant of a thread as being the conjunction of the corresponding invariants for each of its active function activations. As long as modifications to `val` do not break these invariants, they won't affect the ability to verify the functions. Thus, we would expect that it should appear to the owner of `fOb` that no other thread is concurrently modifying `val`. This amounts to saying that `val` is just like a mutable variable, except that (1) it has to be modified using atomic actions, and (2) other threads can concurrently read it (with atomic actions).

Unfortunately, we have forgotten about one further important aspect of mutable variables, namely framing. A sequential field of a mutable object can only be modified in a function call only if it (or the object) is explicitly mentioned in a writes clause. But this restriction doesn't hold for volatile fields of closed objects. Thus, to allow the proper framing of volatile fields, we have to somehow force the owner of `fOb` to treat the update of `val` as a sequential write. To enable this behavior, VCC provides a special annotation:

```
  _(invariant \approves(\this->\owner, val))
```

If `\this->\owner` is not a thread, `\approves(\this->\owner, val)` is equivalent to `\unchanged(val)|| \inv2(\this->\owner)`. However, if the object is owned by a thread, an update to `val` is allowed only by the owning thread, and only if the object is writable. We say that `val` is "owner-approved".

Lock-free data structures designed for general use in other data structures typically are defined with a ghost abstract value that is owner-approved. A function that updates the abstract value then needs to know that the object is wrapped, or needs to know the type of its owner.

15. A Lock-Free Set

To bring together most of the ideas we've seen so far, let's consider a monotonically growing set of `unsigned`s, implemented using an array:

```
typedef unsigned Val, volatile VVal;

typedef struct Set {
 _(ghost volatile \bool mem[Val])  // abstract value of the set

  // abstract behavior of the set: the set only grows
  _(invariant \forall Val v;
      \old(mem[v]) && \this->\closed ==> mem[v])

  // concrete representation
  VVal data[SIZE];
  volatile size_t len;

  // explicit witness: idx[v] gives an index at which v appears
  _(ghost volatile size_t idx[Val])

  _(invariant len <= SIZE)
  _(invariant \forall size_t i; i < len ==> mem[data[i]])
  _(invariant \forall Val v;
      mem[v] <==> idx[v] < len && data[idx[v]] == v)
  _(invariant \old(len) <= len)
  _(invariant \forall size_t i; {data[i]}
      i < \old(len) ==> \unchanged(data[i]))
} Set;
```

The invariants say that the length of the used part of the array grows monotonically, and that values in the used part of the array never change. The constructor for the set looks just like it does in the sequential case:

```
void setNew(Set *s)
  _(requires \mutable(s))
  _(writes \extent(s))
  _(ensures \wrapped(s))
  _(ensures \forall Val v; !s->mem[v])
{
  s->len = 0;
  _(ghost s->mem = \lambda Val v; \false)
  _(wrap s)
}
```

Adding an element to the set has a different specification than in the sequential case, because other threads might be concurrently adding elements to the set. What we can guarantee is that the element we are adding to the set is in the set afterwards (unless we run out of room). Note that a caller of the function could use a claim to show that other elements that were in the set before the function call are in the set afterward.

```
BOOL setAdd(Set *s, Val v _(ghost \claim c))
  _(always c, s->\closed)
  _(writes c)
  _(ensures \unchanged(c->\claim_count))
  _(ensures \result ==> s->mem[v])
{
  BOOL result;
  _(atomic c,s) {
    result = (s->len != SIZE);
    if (result) {
      s->data[s->len] = v;
      _(ghost s->idx[v] = s->len)
      s->len++;
      _(ghost s->mem[v] = \true)
    }
  }
  return result;
}
```

Note that the updates to the set all have to happen within the atomic action. Note also that our atomic action contains several physical memory accesses, which will produce warnings (for good reason). We will just ignore these for now.

When testing membership in the set, we have a similar problem in taking into account the actions of other threads. If the function says the element is in the set, we can trust that it is in the set afterward (thanks to monotonicity); if it is not in the set, we can only be sure that it wasn't in the set when the call started (but might be in the set now).

```
BOOL setMem(Set *s, Val v
  _(ghost \claim c))
  _(requires v)
  _(maintains \wrapped0(c))
  _(always c, s->\closed)
  _(writes c)
  _(ensures \result ==> s->mem[v])
  _(ensures !\result ==> !\old(s->mem[v]))
{
  _(ghost size_t idx = s->idx[v])
  _(ghost \bool isMem = s->mem[v])
  _(ghost \claim cl = \make_claim({c}, s->\closed &&
      (isMem ==> idx < s->len && s->data[idx] == v)))

  size_t len = _(atomic_read cl,s) s->len;
  _(ghost \destroy_claim(cl,{c}))
  _(ghost cl = \make_claim({c}, s->\closed &&  len <= s->len &&
      (isMem ==> idx < len && s->data[idx] == v)))

  for (size_t i = 0; i < len; i++)
    _(writes cl,c)
    _(invariant \wrapped0(cl))
    _(invariant idx < i ==> !isMem)
    _(invariant \wrapped(c) && c->\claim_count == 1)
  {
    if (_(atomic_read cl,s) s->data[i] == v) {
      _(ghost \destroy_claim(cl,{c}))
      return TRUE;
    }
  }
  _(ghost \destroy_claim(cl,{c}))
  return FALSE;
}
```

This function first records (in ghost state) an index at which v occurs in the array (if any), and whether it occurs in the array. It then claims that if v was in the set when the call was made, then it is still in the set (at the recorded index). After we read the length of the line, we strengthen this claim so that it also claims that if v was in the set, then its index is less than the read length. Since this hypothetical index is remaining fixed, the loop is doing what amounts to a linear search for it.

16. Locks

We now dive a bit deeper, and consider how typical synchronization primitives are constructed. We consider here the most basic one, a spinlock.

There are various ways one might try to formalize what a spinlock provides to the client. For example, one might imagine adding to threads a ghost variable that tracks whether they hold the lock, and maintaining a global invariant that at most one of these is true at any time. But we already have a mechanism that does essentially this, namely ownership of some "protected object" associated with the lock. Moreover, ownership provides a nice way to package up exactly what you are allowed to do when you hold a lock: you are allowed to enjoy precisely those privileges that come with ownership, such as the ability to unwrap the object. Moreover, by defining a lock that takes an arbitrary protected object, we effectively make locks polymorphic in the data and its invariant. (This is a common idiom for capturing polymorphism.)

```
typedef _(volatile_owns) struct Lock {
  volatile BOOL locked;
  _(ghost \object ob)
  _(invariant locked || \mine(ob))
} Lock;
```

Note that \object is the ghost type of (typed) pointers, so ob is really a pointer to a (real or ghost) object. The invariant of the lock says that whenever the lock is free (i.e., whenever it is not locked), the lock owns the protected object ob. Note that while the field locked is volatile, the field ob is not, so changing the object protected by the lock requires opening up the lock. The _(volatile_owns) annotation says that the set of objects owned by a Lock can change while the Lock is closed; this means that we have to manage the owns set of locks explicitly (just like we had to do for objects of types marked _(dynamic_owns)).

```
void lockCreate(Lock *l _(ghost \object ob))
  _(requires \extent_mutable(l))
  _(requires \wrapped(ob))
  _(writes ob, \extent(l))
  _(ensures \wrapped(l) && l->ob == ob)
{
  l->locked = FALSE;
  _(ghost l->ob = ob)
  _(ghost l->\owns = {ob})
  _(wrap l)
}
```

Creating a lock requires that the protected object is already wrapped (we can't expect this function to do it because we can't wrap an object without knowing its specific type). Lock destruction is routine (it just unwraps the lock).

```
void lockAcquire(Lock *l _(ghost \claim c))
_(always c, l->\closed)
_(ensures \wrapped(l->ob) && \fresh(l->ob))
{
  BOOL done;
  do
  {
    _(atomic c,l) {
      done = !cmpxchg(&l->locked, 0, 1);
      _(ghost if (done) l->\owns -= l->ob)
    }
  } while (!done);
}
```

Acquiring the lock is more interesting. Within an atomic action, we try to change the lock state from unlocked to locked. If this succeeds, we transfer ownership of the protected object from the lock to the caller. Note that if the change succeeds, the object was unlocked in the prestate, so we are actually guaranteed to get ownership of the protected object when the update succeeds.

```
void lockRelease(Lock *l _(ghost \claim c))
  _(always c, l->\closed)
  _(requires c != l->ob)
  _(requires \wrapped(l->ob))
  _(writes l->ob)
{
  _(atomic c,l) {
    l->locked = FALSE;
    _(ghost l->\owns += l->ob)
  }
}
```

Releasing the lock involves transferring ownership of the protected object back to the lock. Note that the thread releasing the lock need not be the thread that acquired the lock; the protected object might have been transferred through some other means (such as another lock). The requirement that the protected object be wrapped prevents a thread from double-freeing. On the other hand, there is nothing to prevent us from double-locking; this will simply deadlock.

A typical use of a lock is to allow threads to share a piece of data where consistency-preserving updates to the data cannot be done atomically in hardware. We make a lock-protected object using a raw object and a lock; the lock-protected object owns the lock, of which the lock-protected object is the raw object:

```
// a type requiring lock protection
typedef struct Counter {
  unsigned val;
} Counter;

// a lock-protected S
typedef struct AtomicCounter {
  Lock l;
  Counter c;
  _(invariant \mine(&l) && l.ob == &c)
} AtomicCounter;
```

To do an atomic operation on the lock protected data, we first acquire the lock, which gives ownership of the raw data. We can then unwrap the data and update it sequentially; after restoring its consistency, we wrap it up again and unlock the lock:

```
// do a locked update on ac
void inc(AtomicCounter *ac _(ghost \claim c))
  _(always c, ac->\closed)
{
  lockAcquire(&ac->l _(ghost c));
  _(unwrapping &ac->c) {
    if (ac->c.val < UINT_MAX)
      ac->c.val++;
  }
  lockRelease(&ac->l _(ghost c));
}
```

17. Locked Atomics

While this presentation of the last data sufficed to show concurrency safety (e.g., it guarantees that you lock shared data before using it), it is not quite sufficient for reasoning about the programs that use the data. For example, we cannot talk about a counter implemented this way as monotonically increasing; because it is being unwrapped all the time, a thread knows nothing about what other threads might do to the data (other than being sure that its invariant is restored). In order to do such reasoning, we need to represent the abstract value as the field of an object that stays closed; this will allow a thread to hold a claim that claims that the containing object is closed and the abstract value is increasing. But at the same time, we want the holder of a lock on the object to control updates to this abstract value.

A natural way to do this is to replace the abstract value field with an abstract value object that always stays closed, with the abstract value stored as an owner-approved field. This abstract value object is owned by the concrete object when it is closed, and is owned by the thread modifying the concrete object when the concrete object is open.

We begin with an abstract value type, which can be constrained with invariants, and whose components are owner-approved:

```
_(typedef _(claimable) struct AbsCounter {
  _(ghost volatile unsigned val)
  _(invariant val >= \old(val))
  _(invariant \approves(\this->\owner,val))
} AbsCounter)
```

Note that the abstract counter is claimable. The reason for this is that clients who want to reason about the counter are going to reason about the abstract counter (which will stay closed), and doing this requires forming a claim that guarantees that the abstract counter stays closed. Since we are heading toward a solution in which the thread updating the counter will own the abstract counter, the only way for another thread to keep the abstract counter closed is to have a claim on it.

A concrete value contains an abstract value object (which it owns), and a coupling invariant linking the abstract and concrete values:

```
typedef struct Counter {
  unsigned val;
  _(ghost AbsCounter abs)
  _(invariant \mine(&abs) && abs.val == val)
} Counter;
```

An atomic counter consists of a concrete counter and a lock protecting the counter:

```
typedef struct AtomicCounter {
  Counter c;
  Lock l;
  _(invariant \mine(&l) && (&l)->ob == &c)
} AtomicCounter;

void counterNew(AtomicCounter *s)
  _(requires \extent_mutable(s))
  _(writes \extent(s))
  _(ensures \wrapped(s))
{
  s->c.val = 0;
  _(ghost s->c.abs.val = 0;)
  _(wrap &s->c.abs)
  _(wrap &s->c)
  lockNew(&s->l _(ghost &s->c));
  _(wrap s)
}
```

Now, to update the counter, we acquire the lock, unwrap the counter, update the concrete data, and update the abstract counter atomically while the counter is open. (We need to do this to restore the coupling invariant, which is required to rewrap the counter.)

```
void counterUpdate(AtomicCounter *s _(ghost \claim cl))
  _(always cl, s->\closed)
{
  lockAcquire(&s->l _(ghost cl));
  Counter *c = &s->c;
  _(ghost AbsCounter ^abs = &c->abs;)
  _(unwrapping c) {
    if (c->val < UINT_MAX) {
      c->val++;
      _(ghost_atomic abs {
        abs->val++;
        _(bump_volatile_version abs)
      })
    }
  }
  lockRelease(&s->l _(ghost cl));
}
```

(The `_(bump_volatile_version abs)` is needed for implementation reasons that go beyond the scope of this article.)

Finally, we can replicate the example of using claims to link two successive observations of the counter. We are doing just what we did before, but forming the claim on the abstract counter rather than on the counter

18. Transactions

We often want lock-free data structures that provide linearizability - the illusion that an operation (such as a read of or update to the abstract value of a data structure) appears to happen atomically sometime between the invocation and return. It's easy to validate visually that the code for a function body behaves this way - we can simply look for where the abstract value is updated, and prove that the update happens exactly once. However, it's not so easy to write a specification for the function that does this update.

One way to capture the intuition of what we want is to introduce a ghost object representing each ghost operation. This operation object has a boolean, owner-approved field `done` that goes from `\false` to `\true` exactly when the operation appears to "happen". The operation object has an invariant that says that when the operation happens, the abstract value changes according to the semantics of the operation. The function "performing" such an operation takes as precondition that the operation is wrapped but not done, and ensures that the operation is done when it returns.

This sounds good, but is not quite good enough, for several reasons. First, this specification allows the function to do other updates to the data structure that do not correspond to the operation. Second, it allows multiple operations to be marked done in a single atomic step, with the corresponding update to the data structure "satisfying" all of these operations. We can prevent these problems by requiring that exactly one operation is marked done on any update to the abstract value. (Note that it would be safe to allow multiple atomic reading operations to be marked done simultaneously, but we won't treat read operations separately here.) Third, nothing prevents anyone from making new operations, which can then be used to operate on the object. We can prevent this by having the object owner approve all new operation creation.

This last point might seem counterintuitive. Why have the object owner approve the creation of operations, rather than their execution? The reason is that we want to write the code that implements operations without knowing anything about the owner of the data (and hence, what his invariant might be). All the operation implementation needs to know is that it has a suitable operation object available. Thus, the operation object serves as a one-time "permission" to perform the specific operation on the data.

Let's look at some concrete code. We'll do a toy example of a counter, with a single type of operation that increments the counter.

```
typedef struct Counter Counter, *PCounter;
_(typedef struct Op Op, ^POp)

typedef struct Counter {
  volatile unsigned val;
  _(ghost volatile POp turn)
  _(invariant \on_unwrap(\this,\false))
  _(invariant \unchanged(val) ||
      (turn && turn->\closed && turn->c == \this && \inv2(turn)))
} Counter;
```

(`^T` means a pointer to a ghost object of type `T`; we have to keep these separate from ordinary pointers because we have to make sure to not accidentally update a concrete object from ghost code by dereferencing a ghost pointer that happens to point to it.) The field `turn` witnesses the (unique) choice of operation that is being "executed" whenever the value of the counter changes. Note that this type definition is generic with respect to what the particular operations are.

The invariants of an operation are somewhat more complex:

```
_(typedef struct Op {
  _(invariant \on_unwrap(\this, \false))

  PCounter c;
  _(invariant c->\closed)
  _(invariant \old(!\this->\closed) ==> \inv2(c->\owner))
```

To keep things simple, operations are never destroyed. (This is not a problem, because we don't necessarily have to clean up ghost objects.) Each operation is specific to a particular counter. The last invariant says that the owner of the counter has to approve creation of new operations on the counter.

```
  volatile \bool done;
  _(invariant \old(done) ==> done)
  _(invariant \approves(\this->\owner, done))
  _(invariant \unchanged(done) && c->turn == \this
      ==> \unchanged(c->val))
```

The owner-approved field `done` identifies when the operation has completed. The last invariant says that if this operation is the selected one, but it's not happening, then the abstract data value doesn't change.

```
  \claim cl;
  _(invariant \mine(cl))
  _(invariant \claims(cl, !done ==> c->val < UINT_MAX))

  _(invariant \unchanged(done) ||
      (c->turn == \this && c->val == \old(c->val) + 1))
} Op, ^POp)
```

In order to safely increment the counter, we need to know that it's not going to overflow. However, we cannot simply make this an invariant, since the invariant would not be admissible. This is because the justification for its invariance is specific to how the counter is being used, i.e., it depends on the owner of the counter. Therefore, we need an object of some unknown type whose invariant implies the necessary safety property. The natural object to use for this is a claim. The actual claimed property of the claim is hidden at this level of the code.

We can now write a function to perform the increment operation on the counter:

```
void counterInc(PCounter c _(ghost POp op))
  _(requires \wrapped(op) && !op->done && op->c == c)
  _(writes op)
  _(ensures op->done)
{
  _(atomic op, c,  _(read_only) op->cl) {
    c->val++;
    _(ghost op->done = 1)
    _(ghost c->turn = op)
    _(bump_volatile_version op)
  }
}
```

Note that we don't need a separate claim that `c` is closed, because this is already part of the invariant of the operation `op`.

What we have given so far can be viewed as the implementation of a linearizable counter. Here is an example of a toy protocol using the counter, that increases the counter by 2:

```
#define rem(op) ((!op || !op->done) ? 1 : 0)
_(typedef struct Inc2 {
  PCounter c;
  volatile POp op1, op2;
  _(invariant \mine(c))
  _(invariant op1 ==> op1 != op2)
  _(invariant op1 ==> op1->\closed && op1->c == c)
  _(invariant op2 ==> op2->\closed && op2->c == c)
  _(invariant \forall POp op;
      op->\closed && op->c == c ==> op == op1 || op == op2)
  _(invariant \inv2(c) && c->val + rem(op1) + rem(op2) == 2)
} Inc2;)
```

This protocol allows the construction of two (possibly concurrent) operations on the counter. Note that the protocol must own the counter, since it must control the creation of new operations (in order for the next-to-last invariant to be admissible). The last invariant says that the number of pending operations plus the current count is 2. Thus, when all operations have completed, the counter is necessarily 2.

While we have focussed on linearizability, this approach works for arbitrarily complex operation automata. For example, we could write an operation automata describing an abstract algorithm or protocol, and require a function to follow that protocol (returning in a state where the protocol object is in a final state).

19. Conclusion

We have described an approach to the verification C programs, and in particular, hierarchical data structures built as abstract data types, that allows modular verification while respecting the scoping found in real implementations. In contrast to some alternative approaches, the reasoning was based completely on ordinary program invariants, without the need for extra-logical constructions like separating conjunction or permissions.

While this wasn't difficult for sequential code, doing this for concurrent code was rather complicated. It's quite possible that there are much more economical verification constructions for linearizable data objects than the ones we have presented.