verified programming with VCC

ernie cohen

verified programming

- goal: write code
  - that is mechanically verified to satisfy its specs
  - without compromising on performance
  - (this means that you can change the code, but you shouldn't change the binary substantially)
- approach: verified (sequential, concurrent, hybrid) C code
- why C? because most of the software that matters is in C(++)
  - e.g.: OS kernels, compilers, language runtimes, embedded controllers,…

tool

- we will be programming in C, using VCC (Verified Concurrent C)
  - today, this requires Windows/Visual Studio to run
  - alternately, you can run it batchwise on rise4fun
- VCC is not a production-quality tool, but it has been used to successfully verify highly concurrent code (100KLOC, mostly from products)
- if you don't know C, you should (especially if you want to build tools)
- if you know an imperative programming language (e.g. Java, C#) you should be able to pick it up what you need from the lectures (ask friends for help if you need it)

getting started with VCC

- if you have a windows machine, follow instructions at vcc.codeplex.com to install the software
  - please do this *before* your lab session
- if you want to use VCC over the web, go to rise4fun.com
  - you will miss out on a lot of nice features though
- in either case, the best way to start is by working through the tutorial
- if you are hoping to use VCC to verify something interesting, you should talk to me (or email me at ernie.cohen@acm.org)
- if you get stuck on something for more than 5-10 minutes, ask someone for help; success correlates strongly with having an experienced user to talk to

objects and invariants

- assume fixed
  - a set of objects Ob, each with a unique name
  - a set of field names F
  - a set of Values V
- states: St = Ob->F->V
- an execution is an infinite sequence of states
- transitions: Tr = St x St
  - the transitions of an execution are the consecutive pairs of states
- object invariants: assume a fixed \inv2: Ob -> Tr -> \bool
  - if \inv2(o)(s,t), we say the transition from s to t satisfies the invariant of o
  - \inv(o)(s) = \inv2(o)(s,s)
- a transition (s,t) updates o iff for some field f, s(o)(f) != t(o)(f)
- (s,t) is legal iff (\forall o: (s,t) updates o ==> \inv2(o)(s,t))
- (s,t) is good iff (\forall o: \ inv2(o)(s,t))
  - s is good iff (s,s) is good
  - an execution is good if all of its states and transitions are good
- o is admissible iff

  (\forall s,t: good(s) /\ (s,t) legal =>

  \inv2(o)(s,t)) /\ ((s,t) good => \inv(o)(t))
- thm: if the first state and every transition of an execution is good, and ever object is admissible, then every state and every transition of the execution is good

verification approach

- to prove something about the world,
  - model the world as a collection of objects
  - show that all of the object invariants are admissible
  - show that state changes consistent with natural law (e.g. physics, operational semantics of C, …) are legal
- to verify software, do this while matching software structure
- key enablers:
  - admissibility is normally monotonic (and follows scoping)
  - legality check is local

admissible and inadmissible invariants

- a: \old(a->x) <= a->x
- b: a->x < 5
- c: a->x > 10
- d: d->x == \old(d->x) || a->x == 2
- e: \unchanged(e->x) || \inv2(f)
- f: e->x == 5
- g: g->x > \old(g->x)
- h: \inv2(h)
- i: !\inv2(i)

writing invariants

- \old(e) in an object invariant means e evaluated in the prestate
- \unchanged(e) == (\old(e) == e)
- \inv2() and \inv() can appear in object invariants, but only with positive polarity

approval

```
\approves(o,x) == \unchanged(x) || \inv2(o)
```

```
a: \approves(o,a->x)
```

- this is always admissible
- corresponds to giving o “deny” permission on a->x
- finer-grained permissions can also be expressed, e.g.

```
b: \unchanged(b->x) || b->x != 7 || \inv2(o)
c: \unchanged(c->x) || \old(c->x) <= c->x || \inv2(o)
```

ownership

a: \approves(o,a->x)

ownership

a: \approves(o,a->x)

problem: we want to be able to transfer ownership of a to a new owner

ownership

a: \approves(a->\owner, a->x)

ownership

a: \approves(a->\owner, a->x)

problem: ownership of a can be “stolen” from o

ownership

a: \approves(a->\owner, a->x)

a: \approves(\old(a->\owner), a->\owner)

ownership

a: \approves(a->\owner, a->x)

a: \approves(\old(a->\owner), a->\owner)

problem: what if the new owner doesn’t want ownership?

ownership

a: \approves(a->\owner, a->x)

a: \approves(\old(a->\owner), a->\owner)

a: \approves(a->\owner, a->\owner)

ownership

a: \approves(a->\owner, a->x)

a: \approves(\old(a->\owner), a->\owner)

a: \approves(a->\owner, a->\owner)

problem: we want to transfer ownership of a without having to check the invariant of a

ownership

a: \approves(a_o->\owner, a->x)

a_o: \approves(\old(a_o->\owner), a_o->\owner)

a_o: \approves(a_o->\owner, a_o->\owner)

ownership

a: \approves(a->\owner, a->x)
a_o: \approves(\old(a->\owner), a->\owner)
a_o: \approves(a->\owner, a->\owner)

(to avoid clutter, we declare a_o as a “group” of a and label \owner field of a as really being a field of a_o)

- can generalize this to having multiple “owners” with rights to different actions on a
- can use a level of indirection, so that each right is given to the owner of a corresponding ghost object

abstract data

s: \approves(s->\owner, s->val)

s: coupling_invariant(s->val,s->rep)

o: s->\owner == o

o: p(s->val, o->data)

- o talks about the abstract value of s (s->val)
- the representation of s->val is given by an invariant of s, which can be hidden from o
- o can't change s->val directly (since he doesn't know about the internals of s), but can use functions that know these internals to update s. these functions are spec'd using s->val.
- also possible: o doesn't approve s->val, but depends on other invariants restricting its behavior

devices

```
d: \approves(d->\owner, d->on)
d: \unchanged(d->count)
   || (d->count == d->count+1 && d->on)
```

- owner can turn the device on and off, but doesn’t control the actual counting

forward simulation

abs: beh(abs->val)

abs: \approves(concr, abs->val)

concr: coupling_invariant(abs->val, concr->data)

- operations on concr->data are obliged to maintain
  - the coupling invariant with abs->val
  - the beh invariant of abs->val
- the resulting proof shows that concr “simulates” abs
- (note that, like a device, abs can include other fields not under control of concr)

time

```
t: \unchanged(t->val) || t->val == t->val + 1
t: \unchanged(t->val)
   || \forall \object o; t->timed[o] ==> \inv2(o)
t: \old(t->\timed[o]) ==> timed[o]

plane: t->timed[plane]
plane: \unchanged(t->val) ||
   (plane->pos == \old(plane->pos) + plane->vel))
```

- only “timed” objects can force God to change their state when He moves time forward
- note: this usually depends on the granularity of time being small enough to not miss catastrophies in the middle of a discrete time jump

linearizable operations

```
ob: \unchanged(ob->val) || (\exists op:
        op == ob->curr_op
        && ob->op[op]
        && !\old(op->done)
        && op->done
        && happens(op,ob->val))
…
```

- the operations (which are normally ghost) serve as “tickets” allowing update to ob->val

open and closed objects

- most objects don’t start out initialized
  - e.g. concrete data in C
  - invariants normally hold only after initialization
  - coupling invariants don’t hold in the middle of (sequential) updates
- convention:
  - each object has a Boolean field \closed
  - by default, an object invariant inv in o means

    \old(o->\closed) || o->\closed ==> inv
  - for any nonvolatile field f, there is an implicit invariant

    \unchanged(o->f)

is this admissible?

```
typedef struct S S, *PS;

typedef struct S {
    volatile PS pred;
    volatile PS succ;
    _(invariant pred ==> pred->succ == \this)
    _(invariant succ ==> succ->pred == \this)
} S;
```

▪

?

```
typedef struct S S, *PS;

typedef struct S {
    volatile PS pred;
    volatile PS succ;
    _(invariant \on_unwrap(\this,\false))
    _(invariant pred ==> pred->succ == \this)
    _(invariant succ ==> succ->pred == \this)
} S;
```

▪

?

```
typedef struct S S, *PS;

typedef struct S {
   volatile PS pred;
   volatile PS succ;
   _(invariant \on_unwrap(\this,\false))
   _(invariant pred ==> pred->succ == \this)
   _(invariant succ ==> succ->pred == \this)
   _(invariant \this->\closed && pred ==> pred->\closed)
   _(invariant \this->\closed && succ ==> succ->\closed)
} S;
```

▪

?

```
typedef struct S S, *PS;

typedef struct S {
   volatile PS pred;
   volatile PS succ;
   _(invariant \on_unwrap(\this,\false))
   _(invariant pred ==> pred->succ == \this)
   _(invariant succ ==> succ->pred == \this)
   _(invariant \this->\closed && pred ==> pred->\closed)
   _(invariant \this->\closed && succ ==> succ->\closed)
   _(invariant \unchanged(pred) || !\old(pred) || \inv(\old(pred)))
   _(invariant \unchanged(succ) || !\old(succ) || \inv(\old(succ)))
} S;
```

▪

why verify software?

- without verification, you can’t write correct software
- with verification, you can write correct software
- ex: as homework, try to write a correct binary search

```
size_t bsrch(int *p, size_t len, int val)
// return an index i < len s.t. p[i] == val, or len if none exists
```

- hopefully, you learned about binary search in school
- how many of you think you could program a correct binary search?
- how long would it take you to do it?
- how sure would you be that it was correct?
- how much time would it take you to document it? how precise would your documentation be?
- how much work would it be for you to test it thoroughly?

cautionary tale: binary search

- algorithm first published in 1946, but first correct version didn't appear until 1962
- in 1988, a survey of 20 textbooks on algorithms found that at least 15 of them had errors
- Bentley reports giving it as a programming problem to over 100 professional programmers from Bell Labs and IBM, with 2 hours to produce a correct program. At least 90% of the solutions were wrong. Dijkstra reported similar statistics in experiments he performed at many institutions. Bentley reported similar numbers for incoming CMU CS graduate students.
- Bentley published a CACM "programming pearl" on binary search and proving it correct, expanded to 14 pages in "Programming Pearls" (1986).
- Joshua Bloch used Bentley's code as a basis for the binary search implementation in the JDK, in 1997.
- in 2006, a bug was found in the JDK code, the same bug that was in Bentley's code, which nobody had noticed for 20 years. The same bug was in the C code Bentley published for the second edition of his book in 2000.
- these are not exactly your average programmers

Bloch’s conclusion

“…The general lesson that I take away from this bug is humility: It is hard to write even the smallest piece of code correctly, and our whole world runs on big, complex pieces of code.”

(correct)

“We programmers need all the help we can get, and we should never assume otherwise. Careful design is great. Testing is great. Formal methods are great. Code reviews are great. Static analysis is great. But none of these things alone are sufficient to eliminate bugs: They will always be with us. A bug can exist for half a century despite our best efforts to exterminate it. We must program carefully, defensively, and remain ever vigilant.”

(incorrect)

cautionary tale: Chord

- a distributed (ring) hash table algorithm, developed at MIT
- the 4th most cited paper in computer science, according to Citeseer; won SIGCOMM "Test of Time" award in 2011.
- from the paper: "Three features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance."
- the proofs in the paper (and the protocol itself) are buggy; not one of the 7 invariants given in the paper is an invariant
- this is not an isolated example; many published journal concurrent/distributed algorithms are incorrect

cautionary tale: crypto protocols

- in 1995, people finally got around to model-checking and verifying crypto protocols (assuming perfect cryptography)
  - these are basically 2-10 line distributed programs
- more than half of the published authentication protocols were buggy

some takeaways

- people can’t write correct software
- many eyes looking at code doesn’t guarantee correctness
- it’s not good enough to verify algorithms; you have to verify code
- deductive verification is not free, but neither is testing; a typical software shop spends more on trying to eliminate bugs than they spend on writing the code

algorithms vs programs

- when reasoning about an algorithm, you can assume you get to see the whole global state
- when reasoning about a program, you are obliged to follow the scoping rules
  - in particular, you can't see the software that hasn't been written yet
- reasoning about algorithms hasn't changed substantially in 30 years
- reasoning about programs has undergone massive changes in the past decade

invariants

- we’re going to prove things about programs by constructing a big fact F about the program
- we prove F by proving that it is true initially, and that it can never go from being true to being false; we then say F is an “invariant”
- F is the conjunction of many separate statements about the program; these will be of the form “this is true here”:
  - “this is true whenever control reaches this location”
  - “this is always true for each instance of this data structure”
- these annotations will be sprinkled throughout the code

```
unsigned add(unsigned x, unsigned y)



{
    unsigned i = x;
    unsigned j = y;

    while (i > 0)


    {
        i--;
        j++;
    }
    return j;
}
```

```
unsigned add(unsigned x, unsigned y)
    _(requires x + y <= UINT_MAX)


{
    unsigned i = x;
    unsigned j = y;

    while (i > 0)


    {
        i--;
        j++;
    }
    return j;
}
```

```
unsigned add(unsigned x, unsigned y)
    _(requires x + y <= UINT_MAX)


{
    unsigned i = x;
    unsigned j = y;

    while (i > 0)
        _(invariant i + j == x + y)

    {
        i--;
        j++;
    }
    return j;
}
```

```
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++;
    }
    return j;
}
```

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

    while (i > 0)
        _(invariant i + j == x + y)

    {
        i--;
        j++;
    }
    return j;
}
```

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

    while (i > 0)
        _(invariant i + j == x + y)
        _(decreases i)
    {
        i--;
        j++;
    }
    return j;
}
```

```c
_(pure)
unsigned add(unsigned x, unsigned y)
    _(requires x + y <= UINT_MAX)
    _(ensures \result == x + y)
    _(decreases 0)
{
    unsigned i = x;
    unsigned j = y;

    while (i > 0)
        _(invariant i + j == x + y)
        _(decreases i)
    {
        i--;
        j++;
    }
    return j;
}
```

_(requires p)

p holds on entry to the function

(i.e. p is a **precondition** of the function)

_(ensures p)

p holds on return from the function

(i.e., p is a **postcondition** of the function)

\result is the value returned from the function

_(invariant p)

p holds whenever control reaches the top of the loop (before evaluating the loop test)

_(decreases e1,e2, ...)

the function terminates, with any call out to a function that (possibly) calls back to this one having measure smaller than <fn,e1,e2,...>

if on a loop, <e1,e2,...> is decreased by the loop body

\result

the value returned from the function

reasoning with a function spec

```
_(pure)
unsigned add(unsigned x, unsigned y)
   _(requires x + y <= UINT_MAX)
   _(ensures \result == x + y)
;

_(pure)
unsigned add3(unsigned x, unsigned y, unsigned z)
   _(requires x + y + z <= UINT_MAX)
   _(ensures \result == x + y + z)
{
   unsigned i = add(x,y);
   // i == x + y
   return add(i,z);
}
```

modular verification

- the _(requires) and _(ensures) annotations provide the **specification** (or **contract**) for the function add
- when reasoning about a call to a function, we will use only its specification, not its implementation
  - when you call a function, you must prove that its preconditions will be satisfied
  - on return from the function, you can assume its postconditions
  - in a real project, you put the specifications in the header files
- this has several big advantages:
  - it hides irrelevant detail from the reasoner (man or machine)
  - you can verify the functions separately
  - if you change the body of a function without changing its specification, you know the change won't break client code
  - the header can serve as the documentation of the function
  - you can program to the specification of a function that hasn't been written yet

reading the heap

```
int valOf(int *p)
  _(ensures \result == *p)
{
  return *p;
}
```

reading the heap

```
int valOf(int *p)
    _(requires \thread_local(p))
    _(ensures \result == *p)
{
    return *p;
}
```

reading the heap

```
typedef struct Int {
    int val;
} Int;

int valOf(Int *p)
    _(requires \thread_local(p))
    _(ensures \result == p->val)
{
    return p->val;
}
```

reading the heap

```
typedef struct Int {
    int val;
} Int;

int valOf(Int *p)
    _(requires \thread_local(&p->val))
    _(ensures \result == p->val)
{
    return p->val;
}
```

```c
size_t find(int v, int *a, size_t len)




{
    for (size_t i = 0; i < len; i++)

    {
        if (a[i] == v) return i;
    }
    return len;
}
```

```
size_t find(int v, int *a, size_t len)
    _(requires \thread_local_array(a,len))




{
    for (size_t i = 0; i < len; i++)

    {
        if (a[i] == v) return i;
    }
    return len;
}
```

```
size_t find(int v, int *a, size_t len)
    _(requires \thread_local_array(a,len))



{
    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;
}
```

.

```
size_t find(int v, int *a, size_t len)
    _(requires \thread_local_array(a,len))
    _(ensures \result <= 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;
}
```

\thread_local_array(a,len)

a points (at least) len items with type that of *a

these items are all “owned” by this thread

\forall T v; p

\exists T v; p

universal/existential quantification

p ==> q

p <== q

p <==> q

p “only if” / “if” / “iff and only iff” q

framing

```
void test() {
  int a[10];
  _(assume a[3] == 3)
  find(a,10,7);
  _(assert a[3] == 3)
}
```

- should this verify? (presumably yes)
- but how do we know that lsearch doesn't change a[3]?
- rule: a function has to declare (in its spec) anything that might change, if the caller might otherwise "remember" something about it
- the function doesn't have to declare changes to state that the caller either doesn't know about (e.g. memory allocated by the callee) or is obliged to forget about (e.g. state that he doesn't control)

writing the heap

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



{
    *to = *from;
}
```

writing the heap

```
void copy(int *from, int *to)
    _(requires \thread_local(from))


{
    *to = *from;
}
```

writing the heap

```
void copy(int *from, int *to)
    _(requires \thread_local(from))
    _(writes to)

{
    *to = *from;
}
```

writing the heap

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

```
void replace(int  *p, size_t len, int target, int replacement)
    _(writes \array_range(p,len))
    _(ensures \forall size_t i; i < len ==>
        p[i] == ((\old(p[i]) == target) ? replacement : \old(p[i])))
    _(decreases 0)
{
    size_t i = 0;
    for (i = 0; i < len; i++)
        _(invariant \forall size_t j; j < len ==>
            p[j] == ((\old(p[j])== target) && j < i ? replacement : \old(p[j])))
    {
        if (p[i] == target) p[i] = replacement;
    }
}
```

_(assume p)

ignore executions in which p does not hold

(or, more operationally)

wait until p holds

_(assert p)

try to prove that p holds at this point, and assume p afterward

\old(e)

the value that e had on entry to the function

_(writes p,q,…)

requires that the objects/fields pointed to by p,q,… are writable, and that a call to this function is allowed to change these fields/object

\array_range(p,len)

the set of objects {&p[i] s.t. i < len}

(approximate) semantics of a function call

- check that the preconditions hold
- in the poststate of the call,
  - assume that the state agrees on all data owned by the calling thread, except for data listed in the writes clauses of the function
  - assume all postconditions of the call

termination

- to prove that a function terminates, you need to prove two things:
  - no infinite loops
  - no infinite recursion
- you prove absence of an infinite loop by giving a measure that decreases on each iteration through the loop
- you prove absence of an infinite recursion by giving a lexicographic measure that decreases on each function call
  - VCC implicitly adds a highest-order measure of the “rank” of the function in the call graph, for functions whose bodies it sees
  - in practice, this means that you can just write _(decreases 0) for any nonrecursive function
  - mututally recursive functions must be declared so in their specs (see the manual for details)

termination examples

```
void test(unsigned x)
    _(decreases x)
{
    for (unsigned i = 0; i<x; i++)
        _(decreases x-i)
    {
        test(i);
    }
}

_(\natural Ackermann(\natural m, \natural n)
    _(decreases m, n)
{
    if (m == 0) return n + 1;
    else if (n == 0) return Ackermann(m - 1, 1);
    else return Ackermann(m - 1, Ackermann(m, n - 1));
})
```

objects and pointers

- a program text defines a fixed set of **objects**
- each object o == <\addr(o),\typeof(o),\ghost(o)>
  - the type of an object determines its **fields** and their types
  - each field is either **concrete** or **ghost**
  - each concrete field occupies a set of byte addresses in memory
- \state == Objects→Fieldname→Values
- note: the objects are logically disjoint
- a **pointer** is either an object or a pair <o,f> where f is a field name
  - \embedding(<o,f>) == o
  - \is_primitive_ptr(<o,f>) == \true;
  - \is_primitive_ptr(o) == \false
  - &(o->f) == <o,f>; *<o,f> in state S == S(o)(f)
- \object is (for the moment) the type of pointers, rather than the type of objects ☹

validity and aliasing

- each object has a ghost \bool field \valid, which determines whether it is one of the “current” objects
- two objects overlap iff they have overlapping concrete fields
- VCC forces programs to maintain the invariant that \valid objects don’t alias
  - you can only make an object o valid if you simultaneously make invalid a set of objects whose concrete fields cover the concrete fields of ob
- proof obligations guarantee that all reads and writes are of fields of \valid objects
- these conditions allow reads and writes of concrete fields to be implemented by reads and writes to shared memory
  - maintain the global invariant that concrete fields of \valid objects agree with their corresponding bytes in memory
- so these conditions immediately eliminate all “crazy” aliasing in C

closed objects and ownership

- each object has a \bool field \closed
  - only valid objects are \closed
- each object has an \object field \owner (which must be an object)
  - only threads can own open objects
  - only the owner of an object can open or close it
- in the context of a thread,

  \wrapped(o)

  means o is a closed object owned by \me

  \mutable(o)

  means o is open object owned by \me, or o == <o’,f> and \mutable(o’)

  \thread_local(o)

  means o is transitively owned by \me, or o==<o’,f> and \thread_local(o’)

  _(wrap o)

  closes o

  _(unwrap o)

  opens o

ghost data

- we make heavy use of ghost data
  - ghost variables in functions
  - ghost fields in objects
  - ghost parameters to functions
- ghost data is used to facilitate verification, but is not part of the compiled program
- ghost data can have some additional types
  - \natural, \integer, \object, \state
  - maps
  - records (structs, but pure values without identity)
  - inductive data types
  - (a few others)

```
#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;

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

_(ghost ...)

ghost code or data declaration

\natural

type of natural numbers

\span(d)

union of d and pointers to all d's primitive fields

\extent(d)

union of \span(d) and the extents of d's nonprimitive fields

\mutable(d)

d is open and owned by \me

\extent_mutable(d)

\extent(d) is open and owned by \me

\wrapped(d)

d is closed and owned by \me

_(wrap d)

set d->\closed to true

_(unwrap d)

set d->\closed to false

_(unwrapping d1, d2,) { ... }

sugar for _(unwrap d1) _(unwrap d2) … _(wrap d2) _(wrap d1)

objects owning objects

- each object has a field \owns (a set of objects)
  - when o->\closed, o->\owns gives the objects owned by o
- _(unwrap o) transfers ownership of all objects in o->\owns from o->\owner to \me
  - this involves a check that they are all \wrapped and writable
- _(wrap o) transfers ownership of all objects in o->\owns from \me to o->\owner
- by default, o->\owns is static and is computed from the invariant of o
- if a type is marked _(dynamic_owns), o->\owns is maintained manually (in ghost code)
- if a type is marked _(volatile_owns), o->\owns can change even while the object is closed (subject to o's invariants)

```
#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;

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)
}
```

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

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)
   }
}
```

objects are not fields

- a struct or union nested inside another struct or union is logically a completely separate object, one that just happens to have an arithmetically related address
- therefore, the low and high members of a Quad are not actually fields of the Quad
- in particular, if the invariants of the Quad type had allowed it, low and high could be owned by another object, and could change while the Quad is closed

admissibility

- the invariant of Quad talks about low.val and high.val…
- …but these are fields of completely separate objects!
- what stops someone from modifying these fields and falsifying the invariant?
- answer: the Quad has an invariant that it owns &low and &high
- for every type, VCC checks that the invariant of an object o of that type cannot be broken (while o is closed) by a legal change to the state
  - if Quad o is closed and the state changes without changing o, then by the invariant of o, o.low is owned by o. Since o is not a thread, o.low is closed. A legal update cannot change a nonvolatile field of a closed object, so o.low.val doesn’t change.

framing

- on a function call, the caller forgets everything he knew about the state, except for the “version” of those objects that he (\me) owns
  - the values in the fields of o, and the versions of the objects owned by o, are a (fixed) function of the version of o
  - if o is not written, the values of all objects transitively owned by o are also a function of the version of o, so they are also unchanged
- the _(writes) specifications in a function contract are there to tell the caller what additional information he has to forget
- therefore, the _(writes) clauses don’t have to mention any objects that were not owned by \me, or fields of objects that were not \mutable, when the function is called
- conversely, if you want to remember more about the state, you need to get ownership of additional objects whose invariants give the information you want to move “into the future” (perhaps by creating them yourself); we’ll see a lot of this starting in the next lecture

model fields

- the alternative to representing the abstract state with ghost fields is to use a function of the concrete state
- these are sometimes called model fields, because they are materialized as fields in some systems
- advantages vs. ghost fields:
  - you don't have to manually update the ghost state
  - in a concurrent setting, the model fields automatically change instantaneously when the concrete state changes, so other objects that
- disadvantages:
  - sometimes the abstract state is related only relationally to the concrete state
  - if the abstract state is a function of the state of other objects, and is subject to further invariants, admissibility has to be proved for every possible update to the other objects, rather than just the ones actually invoked in code. (this isn't a problem if the concrete state is all owned by the object with the invariants)
  - with a ghost field, the verifier can more easily take advantage of parts of the abstract value that doesn't change (since it is reflected syntactically rather than semantically)
  - it is harder to prove admissibility for invariants that use model fields
  - reads clauses don't take into account model fields

```c
#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;

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

```c
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 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)
    }
}
```

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

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

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

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++;
   }
}
```

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

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;
})

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

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

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

maps and records

```
_(typedef \bool UnsSet[unsigned])
_(typedef \bool NatSet[\natural])
_(typedef \bool UnsSetSet[UnsSet])

_(typedef _(record) struct FinNatSetSeq {
   NatSet vals[\natural];
   \natural len;
} FinNatSeq;)

_(void test() {
   \natural squares[\natural];
   squares = (\lambda \natural n; n*n);
   FinNatSeq s;
   s.len = 1000;
   s.vals = (\lambda \natural n;
                  (\lambda \natural m; m < n));
})
```

```
typedef int Val;

typedef struct Set {
    // abstract value of the set
    _(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)
}
```

```
_(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;
}
```

```
typedef int Val;

_(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;
})

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

```
_(pure) BOOL setMem(Set *s, Val v)
    _(requires \wrapped(s))
    _(reads s)
    _(ensures \result == setMem(s)[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; setMem(s)[x] ==
      \old(setMem(s)[x]) || (\result && x == v))
{
    if (s->len == SIZE) return FALSE;
    _(unwrapping s) {
        s->data[s->len] = v;
        s->len++;
    }
    return TRUE;
}
```

existential quantification

- proving an existential quantification requires searching for an appropriate instance
- resolution provers are pretty good at this, but SMT solvers are not (we'll see why later)
- in proof checking, you would construct a suitable witness as part of building the proof
- you can do it in a program annotation by maintaining the witness as ghost data

```
typedef int Val;

typedef struct Set {
  // abstract value of the set
  _(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;
```

```
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;
}
```

```
typedef int Val;

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

   // concrete representation
   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;
}

.
```

recursive data structures

- ex: lists, trees, sorted lists, binary search trees, …
- what should the ownership structure be?
  - each node owns its children?
  - a master object owns all of the nodes?
- how do we make sure that recursive functions terminate?
- how do we make sure traversal doesn't miss nodes?
- how should the abstract value be defined?
  - ghost field for each node?
  - recursively defined model field?
  - inductive datatype?
  - first-order abstraction?
- other issues
  - destructive updates in the middle of data structures
  - controlling aliasing within the structure

ownership options

- there are two basic approaches to ownership in linked data structures
  - you can to keep ownership local; e.g. a list can own its successor
    - we saw an example of this with trees
    - this works well if you are always operating top-down through the structure
  - you can have a ghost object own all of the nodes of the structure
    - this is usually mandatory if you are going to use fine-grained atomic operations on the structure
    - it is also convenient if you want to destructively update the structure in the middle
    - finally, it allows you to use “generic” nodes within the structure, without having to define a separate type for each kind of structure

inductive datatypes

- if you are writing functional programs (but implementing them using concrete data), you can work as follows:
  - define inductive data types
  - define recursive functions on these types, and prove properties of them using pure functions with postconditions
  - show that your concrete data structures implement these data types
  - show that your concrete code simulates the recursive functions

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

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

// lemma written as a function
_(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;
    }
})
```

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

// concrete in-place Reverse
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))
    }
}
```

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

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

// lemma written as a function
_(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;
   }
})
```

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

// concrete in-place Reverse
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);
      if (t->l != t->r) Rev(t->r);
     _(ghost t->val = Reverse(t->val))
   }
}
```

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

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

// lemma written as a function
_(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;
    }
})
```

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

// concrete in-place Reverse
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))
    }
}
```

functional programming warning

- resist the temptation to reduce imperative programming to functional programming
  - use inductive data types as an abstraction only when that is really the abstraction you want to expose
- example: what is the right abstraction of a binary search tree?
  - you can encode these as trees (with a recursive function to test for well-formedness)…
  - …but that just forces the abstraction to expose more information than necessary
  - proving that mutations preserve this abstraction just makes your job harder
  - much simpler: use a set abstraction for each subtree; this makes it easy to state the local correctness of the data structure

single owner approach

- the basic goals of invariants on the structure are
  - make sure that searches don’t miss items
  - make sure that searches terminate
- reachability approach: maintain the binary reachability relation between nodes of the structure
  - this allows first-order updates for arbitrary DAG data structures
  - also allows many items to be deleted from linear structures in one step
  - formulating these invariants is often complex
- indexed approach: make structures ordered
  - usually easiest for linear or tree-like structures
  - for structures with ordered keys, this comes for free
  - otherwise, indices can be maintained with a separate map
  - this approach is usually easier to verify

```
typedef struct Node Node, *PNode;
  struct Node {
  PNode nxt;
};

typedef _(dynamic_owns) struct Queue {
  // abstract value
  _(ghost \natural len)
  _(ghost PNode seq[\natural])

  // implementation
  Node *head;
  Node *tail;
  // idx is the inverse of seq
  _(ghost \natural idx[PNode])
  // coupling invariant
  _(invariant tail ==
    (len == 0 ? NULL : seq[len-1]))
  _(invariant head ==
    (len == 0 ? NULL : seq[0]))
  _(invariant \forall \natural i; {seq[i]}
    i < len ==>
      idx[seq[i]] == i &&
      \mine(seq[i]) &&
      seq[i]->nxt ==
        (i + 1< len ? seq[i+1] : (PNode) 0))
} Queue, *PQ;
```

```
void qInit(PQ q)
_(requires \extent_mutable(q))
_(writes \extent(q))
_(ensures \wrapped(q) && q->len == 0)
{
  q->head = NULL;
  q->tail = NULL;
  _(ghost q->len = 0)
  _(ghost q->\owns = {})
  _(wrap q)
}

_(pure) BOOL qEmpty(PQ)
  _(requires \wrapped(q))
  _(reads q)
  _(ensures \result == (q->len == 0))
{
  return q->head == NULL;
}
```

```
typedef struct Node Node, *PNode;
    struct Node {
    PNode nxt;
};

typedef _(dynamic_owns) struct Queue {
    // abstract value
    _(ghost \natural len)
    _(ghost PNode seq[\natural])

    // implementation
    Node *head;
    Node *tail;
    // idx is the inverse of seq
    _(ghost \natural idx[PNode])
    // coupling invariant
    _(invariant tail ==
        (len == 0 ? NULL : seq[len-1]))
    _(invariant head ==
        (len == 0 ? NULL : seq[0]))
    _(invariant \forall \natural i; {seq[i]}
        i < len ==>
            idx[seq[i]] == i &&
            \mine(seq[i]) &&
            seq[i]->nxt ==
                (i + 1< len ? seq[i+1] : (PNode) 0))
} Queue, *PQ;
```

```
void qEnqueue(PQ q, PNode n)
    _(maintains \wrapped(q))
    _(requires \extent_mutable(n))
    _(writes \extent(n), q)
    _(ensures q->len == \old(q->len + 1))
    _(ensures \forall \natural i; i < q->len ==>
        q->seq[i] == (i == \old(q->len) ? n
                                        : \old(q->seq[i])))
{
    n->nxt = NULL;
    _(wrap n)
    _(unwrapping q) {
        if (!q->head) q->head = n;
        else _(unwrapping q->tail)
            q->tail->nxt = n;
        q->tail = n;
        _(ghost {
            q->seq[q->len] = n;
            q->idx[n] = q->len;
            q->\owns += n;
            q->len = q->len + 1;
        })
    }
}
```

```
typedef struct Node Node, *PNode;
    struct Node {
    PNode nxt;
};

typedef _(dynamic_owns) struct Queue {
    // abstract value
    _(ghost \natural len)
    _(ghost PNode seq[\natural])

    // implementation
    Node *head;
    Node *tail;
    // idx is the inverse of seq
    _(ghost \natural idx[PNode])
    // coupling invariant
    _(invariant tail ==
       (len == 0 ? NULL : seq[len-1]))
    _(invariant head ==
       (len == 0 ? NULL : seq[0]))
    _(invariant \forall \natural i; {seq[i]}
       i < len ==>
          idx[seq[i]] == i &&
          \mine(seq[i]) &&
          seq[i]->nxt ==
           (i + 1< len ? seq[i+1] : (PNode) 0))
} Queue, *PQ;
```

```
PNode qDeque(PQ q)
   _(maintains \wrapped(q))
   _(requires  !qEmpty(q))
   _(writes q)
   _(ensures \result == \old(q->seq[0]))
   _(ensures \extent_mutable(\result))
   _(ensures q->len == \old(q->len) - 1)
   _(ensures \forall \natural i; i <q->len ==>
          q->seq[i] == \old(q->seq[i+1]))
{
    PNode res = q->head;
    _(unwrapping q) {
       _(ghost {
          q->\owns -= q->head;
          q->len = q->len – 1;
          q->seq =
             \lambda \natural i; q->seq[i+1];
          q->idx = \lambda PNode n;
             (q->idx[n] > 0) ? q->idx[n] - 1 : 0;
       })
       q->head = q->head->nxt;
    }
    _(unwrap res)
    return res;
}
```

```
typedef struct Node Node, *PNode;
  struct Node {
  PNode nxt;
};

typedef _(dynamic_owns) struct Queue {
  // abstract value
  _(ghost \natural len)
  _(ghost PNode seq[\natural])

  // implementation
  Node *head;
  Node *tail;
  // idx is the inverse of seq
  _(ghost \natural idx[PNode])
  // coupling invariant
  _(invariant tail ==
    (len == 0 ? NULL : seq[len-1]))
  _(invariant head ==
    (len == 0 ? NULL : seq[0]))
  _(invariant \forall \natural i; {seq[i]}
    i < len ==>
      idx[seq[i]] == i &&
      \mine(seq[i]) &&
      seq[i]->nxt ==
       (i + 1< len ? seq[i+1] : (PNode) 0))
} Queue, *PQ;
```

```
PNode qDeque(PQ q)
  _(maintains \wrapped(q))
  _(requires  !qEmpty(q))
  _(writes q)
  _(ensures \result == \old(q->seq[0]))
  _(ensures \extent_mutable(\result))
  _(ensures q->len == \old(q->len) - 1)
  _(ensures \forall \natural i; i <q->len ==>
         q->seq[i] == \old(q->seq[i+1]))
{
   PNode res = q->head;
   _(unwrapping q) {
      _(ghost {
         q->\owns -= q->head;
         q->len = q->len – 1;
         q->seq =
            \lambda \natural i; q->seq[i+1];
         q->idx = \lambda PNode n;
            (q->idx[n] > 0) ? q->idx[n] - 1 : 0;
      })
      q->head = q->head->nxt;
      if (!q->head) q->tail = NULL;
   }
   _(unwrap res)
   return res;
}
```

how about a model field?

- we could have written the abstract value as a model field…
- …but the resulting function would be recursive
  - it’s very hard to tell how an update to a data structure changes the value of a recursive function on that data structure (it typically requires a separate proof)
  - once you start going down the road of recursive functions, you quickly end up having to prove lots of inductive lemmas
- we strongly prefer to reason with first-order formulas instead of recursive functions
  - first-order == local
- in particular, don’t try to replace quantification with recursion

non-hierarchical data structures

- consider a graph; there is no natural internal ownership structure
- sequentially, we could just put everything we know about it into a big global invariant
  - but this is unlikely to scale as the graph gets more heterogeneous
- we'd like nodes to have local information about their neighbors, but then how do we change the nodes without opening up the whole structure?
- ultimately, the problem becomes one of sharing information without an ownership relationship
- this is exactly the problem we will address when we study…

concurrency

what does concurrency have to do with sequential programming?

- concurrency is not about parallelization of activity
  - changing the state safely is easy
- concurrency is about sharing information
  - maintaining accurate knowledge about the state is hard
- message: keep paying attention, even if you only care about sequential programming

invariants and updates

- invariant admissibility is independent of how the program updates the state
- admissibility depends only on the invariants of the objects
- thus, admissibility can be checked based only on the type definitions, without looking at the function bodies
- in VCC, admissibility checking obeys C scoping rules (except for textual ordering)

reading and writing

- to read data sequentially, you must prove that it is not changing
  - normally you prove this by proving it is a nonvolatile field of a closed object
- to write data sequentially, it must be mutable (owned by you and open)
- to read data atomically, you must prove that it is a field of a closed object
  - you prove this using invariants from objects in your sequential domain
- to write data atomically, you must prove that it is a volatile field of an object that is closed, and the action must be legal

volatile fields

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

```
void test(Counter *cnt)
   _(requires \wrapped(cnt))
{
   int x = cnt->val;
}
```

▪

atomic actions

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

```
void test(Counter *cnt)
    _(requires \wrapped(cnt))
{
    int x = _(atomic_read cnt) cnt->val;
}
```

▪

atomic actions

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

```
void test(Counter *cnt)
    _(requires \wrapped(cnt))
{
    int x = _(atomic_read cnt) cnt->val;
    int y = _(atomic_read cnt) cnt->val;
    _(assert x <= y)
}
```

▪

atomic actions

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

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

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

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

    _(unwrap &o)
}
```

implicit reduction

- the only time other threads seem to run is just before a non-ghost atomic action
- when other threads run, you lose all information about the state, except for the versions of the objects you own
- because non-pure functions can engage in atomic actions without reporting them, function calls also lose this information

this works, but is a bit verbose…

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

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

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

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

    _(unwrap &o)
}
```

using claims

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

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

    int y = _(atomic_read c) cnt->val;
    _(assert x <= y)
    _(unwrap &o)
}
```

▪

claims

- a \claim is a ghost object with no data, created only for its invariant
- a \claim c is characterized by
  - the objects it claims
  - its invariant
- the objects claimed by a claim claims must be of types marked _(claimable)
  - claimable objects keep a (ghost) count \claim_count of the number of claims that claim it, and have an invariant that they cannot be unwrapped while this count is nonzero
  - \wrapped0(o) == \wrapped(o) && o->\claim_count == 0
- the invariant of the claim must hold at the time it is formed, and be admissible
  - the invariant includes implicitly that all of the claimed objects are closed
  - this check is done inline where the claim is formed, rather than in a separate type definition
- claims serve as first-class chunks of knowledge; they can be assigned to variables, stored in data structures, passed in and out as parameters

owning the subject is a bit unrealistic…

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

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

   int y = _(atomic_read c) cnt->val;
   _(assert x <= y)
   _(unwrap &o)
}
```

.

claim what you know

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

void test(Counter *cnt)
    _(requires cnt->\closed)
{
    _(ghost \claim c = \make_claim({}, cnt->\closed))

    int x = _(atomic_read c) cnt->val;
    _(ghost c = \make_claim({}, cnt->\closed && x <= cnt->val))

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

▪

what if the subject can go away?

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

void test(Counter *cnt)
    _(requires cnt->\closed)
{
    _(ghost \claim c = \make_claim({}, cnt->\closed))    // no longer admissible

    int x = _(atomic_read cnt) cnt->val;
    _(ghost c = \make_claim({}, cnt->\closed && x <= cnt->val))

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

.

passing knowledge through a parameter

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

void test(Counter *cnt _(ghost \claim c))
    _(always c, cnt->\closed)
    _(maintains \wrapped0(c))
    _(writes c)
{
    int x = _(atomic_read c, cnt) cnt->val;
    _(ghost \claim c1 = \make_claim({c}, cnt->\closed && x <= cnt->val))

    int y = _(atomic_read c1, cnt) cnt->val;
    _(assert x <= y)
    _(ghost \destroy_claim(c1,{c}))
}
```

writing

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

void test(Counter *cnt _(ghost \claim c))
    _(always c, cnt->\closed)
    _(maintains \wrapped0(c))
    _(writes c)
{
    int x = _(atomic_read c, cnt) cnt->val;
    if (x == INT_MAX) return;
    _(atomic c, cnt) {
        if (cnt->val == x) cnt->val = x+1;
    }
}
```

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

void test(Counter *cnt _(ghost \claim c))
    _(always c, cnt->\closed)
    _(maintains \wrapped0(c))
    _(writes c)
{
    int x = _(atomic_read c, cnt) cnt->val;
    if (x == INT_MAX) return;
    _(atomic c, cnt) {
        cmpXchg(&cnt->val, x,x+1);
    }
}

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

atomic actions

- an atomic action has the form

  _(atomic l) stmt

  where l is a closed object list, such that
  - every field read in stmt is either \thread_local or a field of an object of l
  - every field written in stmt is either writable or a volatile field of an object of l not marked _(read_only)
  - the entire atomic statement preserves the invariants of all of the objects listed in l and not marked _(read_only)
- VCC will warn you if there is more than one access that is neither ghost nor \thread_local, but it is up to you to make sure that compiler treats these accesses as atomic.
- you can define _(atomic_inline) functions giving the semantics of atomic compiler intrinsics

ghost atomic actions

_(ghost_atomic o1, o2, … {stmt})

- this is just like an ordinary atomic action, except
  - there is no scheduler boundary
  - only ghost fields can be modified

ex: a lock-free set

```
typedef struct Set {
  // abstract value of the set
  _(ghost volatile \bool mem[Val])
  // abstract behavior of the set
  _(invariant \forall Val v;
     \old(mem[v]) && \this->\closed
     ==> mem[v])

  // concrete representation
  VVal data[SIZE];
  volatile size_t len;
  _(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;
     i < \old(len) ==> \unchanged(data[i]))
} 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)
}
```

.

.

ex: a lock-free set

```
typedef struct Set {
  // abstract value of the set
  _(ghost volatile \bool mem[Val])
  // abstract behavior of the set
  _(invariant \forall Val v;
    \old(mem[v]) && \this->\closed
    ==> mem[v])

  // concrete representation
  VVal data[SIZE];
  volatile size_t len;
  _(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;
    i < \old(len) ==> \unchanged(data[i]))
} Set;
```

.

```
BOOL setAdd(Set *s, Val v
                      _(ghost \claim c))
  _(always c, s->\closed)
  _(maintains \wrapped0(c))
  _(writes c)
  _(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;
}
```

.

```
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;
}
```

approval

```
struct S {
   volatile int x;
   _(ghost \object o)
   _(invariant \unchanged(x) || \inv2(o))
} s;
```

- changes to s->x are guaranteed to not break the invariant of of s->o
- thus, s->o can freely talk about s->x
- as far as s->o can observe, s->x never changes except when “he” changes it
- this is like s->o having a read permission on s->x
- since there are no other invariants restricting change to s->x, this is almost like s->o owning s->x; the only difference is that s->o cannot give away his rights to another owner without opening up s
- in a lock free data structure, the abstract value is typically owner-approved

exercise: break up element insertion

- use 0 as a “not yet filled” value
- maintain a ghost table of values to be filled in (values don’t change below len)
- use a cmpXchg to increase the len field
  - if it succeeds, assign to the ghost table the value you are inserting
  - use the ghost table to prove that you are not overwriting a nonzero value in the real data array

locking

- coarse-grained locking looks a lot like sequential programming
  - a lock is just like a container
  - its exclusivity comes from the exclusivity of ownership
- the only differences between reasoning with locks and reasoning about ordinary containers are
  - you have to share the container with other objects, so instead of owning it, you have evidence that it is closed (typically a claim)
  - instead of unwrapping a container to get its contents out, you call functions to get the contents out and put it back
- because a lock is just a container, any “real” synchronization depends on what you do with what you take out of the lock

```
typedef _(volatile_owns) struct Lock {
    _(ghost \object ob)
    ....

} Lock
;
void lockCreate(Lock *l _(ghost \object ob))
    _(requires \extent_mutable(l))
    _(requires \wrapped(ob))
    _(writes ob, \extent(l))
    _(ensures \wrapped(l) && l->ob == ob)
;
void lockDestroy(Lock *l _(ghost \object ob))
    _(requires \wrapped(l))
    _(writes l)
    _(ensures \extent_mutable(l))
;
void lockAcquire(Lock *l _(ghost \claim c))
    _(always c, l->\closed)
    _(ensures \wrapped(l->ob) && \fresh(l->ob))
;
void lockRelease(Lock *l _(ghost \claim c))
    _(always c, l->\closed)
    _(requires \wrapped(l->ob))
    _(writes l->ob)
;
```

```
typedef _(volatile_owns) struct Lock {
    _(ghost \object ob)
    ....

} Lock
;
void lockCreate(Lock *l _(ghost \object ob))
    _(requires \extent_mutable(l))
    _(requires \wrapped(ob))
    _(writes ob, \extent(l))
    _(ensures \wrapped(l) && l->ob == ob)
;
void lockDestroy(Lock *l _(ghost \object ob))
    _(requires \wrapped(l))
    _(writes l)
    _(ensures \extent_mutable(l))
;
void lockAcquire(Lock *l _(ghost \claim c))
    _(always c, l->\closed)
    _(ensures \wrapped(l->ob) && \fresh(l->ob))
;
void lockRelease(Lock *l _(ghost \claim c))
    _(always c, l->\closed)
    _(requires \wrapped(l->ob))
    _(writes l->ob)
;
```

```
// a type requiring lock protection
typedef struct S {
    int x, y;
    _(invariant x==y)
} S;

// a lock-protected S
typedef struct AtomicS {
    Lock l;
    S s;
    _(invariant \mine(&l) && l.ob == &s)
} AtomicS;

// do an atomic update on s
void sOp(AtomicS *s _(ghost \claim c))
    _(always c, s->\closed)
{
    lockAcquire(&s->l _(ghost c));
    _(unwrapping &s->s) {
        s->s.x = 0;
        s->s.y = 0;
    }
    lockRelease(&s->l _(ghost c));
}
```

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

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)
}

void lockDestroy(Lock *l)
    _(requires \wrapped(l))
    _(writes l)
    _(ensures \extent_mutable(l))
{
    _(unwrap l)
}
```

```
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);
}

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)
    }
}
```

lock destruction

- when destroying a lock, there is no guarantee that you will find the protected object inside
  - the last person to acquire the lock might have not bothered to give it back
- how should this evil be detected?
  - the person who used their right to use the lock shouldn’t have been able to “give back” that right without unlocking
- solution: force lock acquirers to place a claim that the lock is closed “on deposit” in the lock
- the lock invariant is changed so that when the lock is locked, it owns a claim that claims it is closed
- thus, if you every open a lock, the lock invariant guarantees you get the protected object back!

reader-writer locks

- a reader lock on an object is essentially a claim that it is closed
- since the object itself might not be claimable, you need a separate, claimable dummy object that owns the protected object when its claim count is nonzero
- the lock keeps a concrete volatile count that is equal to the claim count on the dummy object
- to acquire a writer lock, check that the (concrete) claim count is zero; if it is, take ownership of the protected object from the dummy object
- when a reader lock is released, you need to return the claim on the dummy object to decrease its claim count
- to prevent a reader from giving back a lesser claim, the lock maintains the set of outstanding claims it has given out, and requires that a thread releasing a reader lock give up one of these claims

```
_(typedef struct Token {} Token)
_(typedef _(claimable) struct Dmy {} Dmy)

typedef _(volatile_owns) struct RwLock {
   volatile unsigned cnt;

   _(ghost \object ob)
   _(ghost Token token)
   _(ghost Dmy claimCnt)

   _(invariant ob != &claimCnt)
   _(invariant ob != &token)
   _(invariant \mine(&claimCnt))

   _(invariant (&claimCnt)->\claim_count
                  == cnt >> 1)
   _(invariant \on_unwrap(\this, !(cnt>>1)))

   _(invariant \mine(&token) || \mine(ob))
   _(invariant (cnt & 1) || \mine(&token))
   _(invariant (cnt == 1) || \mine(ob))
} RwLock;
```

```
void init(RwLock *r _(ghost \object ob))
   _(writes \extent(r), ob)
   _(requires \wrapped(ob))
   _(ensures \wrapped(r) && r->ob == ob)
{
   r->cnt = 0; // no readers or writers
   _(ghost r->ob = ob;)
   _(wrap &r->token)
   _(wrap &r->claimCnt)
   _(ghost r->\owns =
            { &r->token, ob, &r->claimCnt })
   _(wrap r)
}
```

```
_(typedef struct Token {} Token)
_(typedef _(claimable) struct Dmy {} Dmy)

typedef _(volatile_owns) struct RwLock {
   volatile unsigned cnt;

   _(ghost \object ob)
   _(ghost Token token)
   _(ghost Dmy claimCnt)

   _(invariant ob != &claimCnt)
   _(invariant ob != &token)
   _(invariant \mine(&claimCnt))

   _(invariant (&claimCnt)->\claim_count
                    == cnt >> 1)
   _(invariant \on_unwrap(\this, !(cnt>>1)))

   _(invariant \mine(&token) || \mine(ob))
   _(invariant (cnt & 1) || \mine(&token))
   _(invariant (cnt == 1) || \mine(ob))
} RwLock;
```

```
void acquire_read(
            RwLock *r
            _(ghost \claim c)
            _(out \claim ret)
)
   _(always c, r->\closed)
   _(ensures \claims_object(ret, &r->claimCnt))
   _(ensures \claims(ret, r->ob->\closed))
   _(ensures \wrapped0(ret) && \fresh(ret))
{
   unsigned v, n;
   for (;;) {
      _v = (atomic_read c, r) r->cnt;

      // if writer or too many readers, spin
      if (v & 1 || v > UINT_MAX-2) continue;

      // try to bump the reader count
      _(atomic c, r) {
         n = cmpXchg(&r->cnt, v + 2, v);
         _(ghost if (v == n)
            ret = \make_claim({&r->claimCnt},
               r->\closed && r->cnt >> 1 > 0
               && r->ob->\closed))
      }
      if (v == n) return;
   }
}
```

```
_(typedef struct Token {} Token)
_(typedef _(claimable) struct Dmy {} Dmy)

typedef _(volatile_owns) struct RwLock {
   volatile unsigned cnt;

   _(ghost \object ob)
   _(ghost Token token)
   _(ghost Dmy claimCnt)

   _(invariant ob != &claimCnt)
   _(invariant ob != &token)
   _(invariant \mine(&claimCnt))

   _(invariant (&claimCnt)->\claim_count
                   == cnt >> 1)
   _(invariant \on_unwrap(\this, !(cnt>>1)))

   _(invariant \mine(&token) || \mine(ob))
   _(invariant (cnt & 1) || \mine(&token))
   _(invariant (cnt == 1) || \mine(ob))
} RwLock;
```

```
void release_read(RwLock *r
      _(ghost \claim c) _(ghost \claim handle))
   _(always c, r->\closed)
   _(requires \claims_object(handle, &r->claimCnt)
   _(requires \wrapped0(handle))
   _(requires c != handle)
   _(writes handle)
{
   unsigned v, n;
   for (;;)
      _(writes handle)
      _(invariant \wrapped0(handle))
   {
      v = _(atomic_read c, r) r->cnt;
      _(assert \active_claim(handle) && v >= 2)

      _(atomic c, r) {
         n = cmpXchg(&r->cnt, v - 2, v);
         _(ghost if (v == n)
            \destroy_claim(handle, {&r->claimCnt}))
      }
      if (v == n) break;
   }
}
```

```
_(typedef struct Token {} Token)
_(typedef _(claimable) struct Dmy {} Dmy)

typedef _(volatile_owns) struct RwLock {
    volatile unsigned cnt;

    _(ghost \object ob)
    _(ghost Token token)
    _(ghost Dmy claimCnt)

    _(invariant ob != &claimCnt)
    _(invariant ob != &token)
    _(invariant \mine(&claimCnt))

    _(invariant (&claimCnt)->\claim_count
                    == cnt >> 1)
    _(invariant \on_unwrap(\this, !(cnt>>1)))

    _(invariant \mine(&token) || \mine(ob))
    _(invariant (cnt & 1) || \mine(&token))
    _(invariant (cnt == 1) || \mine(ob))
} RwLock;
```

```
void acquire_write(RwLock *r _(ghost \claim c))
    _(always c, r->\closed)
    _(ensures \wrapped(r->ob))
{
    // grab the token
    for (;;) {
        unsigned v,n;
        v = _(atomic_read c, r) r->cnt;
        // if there is already a read lock, spin
        if (v & 1) continue;
        // try to set the writer bit
        _(atomic c, r) {
            n = cmpXchg(&r->cnt, v|1, v);
            // if succesful, grab the token
            _(ghost if (v == n) r->\owns -= &r->token)
        }
        if (v == n) break;
    }

    // wait for the readers to leave
    while (1 != _(atomic_read c, r) r->cnt)
        _(writes {})
    {}
    // exchange the token for the object
    _(ghost_atomic c,r,&r->token {
        r->\owns += &r->token;
        r->\owns -= r->ob;
    })
}
```

```
_(typedef struct Token {} Token)
_(typedef _(claimable) struct Dmy {} Dmy)

typedef _(volatile_owns) struct RwLock {
   volatile unsigned cnt;

   _(ghost \object ob)
   _(ghost Token token)
   _(ghost Dmy claimCnt)

   _(invariant ob != &claimCnt)
   _(invariant ob != &token)
   _(invariant \mine(&claimCnt))

   _(invariant (&claimCnt)->\claim_count
                    == cnt >> 1)
   _(invariant \on_unwrap(\this, !(cnt>>1)))

   _(invariant \mine(&token) || \mine(ob))
   _(invariant (cnt & 1) || \mine(&token))
   _(invariant (cnt == 1) || \mine(ob))
} RwLock;
```

```
void release_write(RwLock *r _(ghost \claim c))
   _(always c, r->\closed)
   _(requires c != r->ob)
   _(requires \wrapped(r->ob))
   _(writes r->ob)
{
   _(atomic c, r) {
      r->cnt = 0;
      _(ghost r->\owns += r->ob)
   }
}
```

.

approval

- an owner-approved field is almost like a mutable variable
  - i.e. to the approver, it looks like nobody else can update it
- differences:
  - updates to the field must be atomic
  - other threads can read it (atomically)
- it would be nice to frame it like a mutable variable
  - i.e., force function calls to declare if they update it
- to make this happen, objects with owner-approved fields have a volatile version (in addition to their sequential version)
  - wrapped objects whose volatile versions change must be listed in a writes clause
  - updating an owner-approved field requires bumping the volatile version (currently)

```
typedef struct Ob {
    volatile int x;
    volatile int y;
    _(invariant \approves(\this->\owner, x))
} Ob;

void op1(Ob *o)
    _(maintains \wrapped(o))
{
    _(atomic o) o->y = 1;
    _(atomic o) o->x = 1; // fails
}

void op2(Ob *o)
    _(maintains \wrapped(o))
    _(writes o)
{
    _(atomic o) {
        o->x = 1;
        _(bump_volatile_version o)
    }
}
```

```
void test(Ob *o)
    _(requires \wrapped(o))
    _(writes o)
{
    _(assume o->x == 2 && o->y==2)
    op1(o);
    _(assert o->x ==2)
    _(assert o->y == 2) // fails
}
```

automata

- invariants describe generalized automata
  - the invariants on closing an object capture the “initial states”
  - the invariants on opening an object capture the “final states”
  - the invariant controlling transitions between closed states represent the transition relation
- this means that you can take your favorite automata models (for safety) and use them inside a program
- because VCC invariants can mention the states of other parts of the system, you can also use these automata to capture synchronous models like CSP and IO automata

simulation

- in most formalisms, simulation is a relation on automata
- in VCC, (forward) simulation is just an invariant
- just as function calls are spec’d in terms of the effect on abstract state, the behavior of an object can be spec’d in terms of the behavior of its abstraction
- usual pattern:
  - abstract object is described as ghost automaton, with its state changes owner-approved
  - a concrete object owns a volatile abstract object, with a coupling invariant relating the two
  - because of the coupling invariant, some changes to the concrete state force update of the abstract state, which requires a check of the abstraction behavior
- the proof obligations match those of forward simulation, but the abstraction remains available for use in other invariants
- the code looks just like our previous code for sequential programming abstractions, except that the updates are atomic and don’t open the object

```
_(typedef struct AbsClock {
    volatile \natural t;
    _(invariant \unchanged(t) || t == \old(t) + 1)
    _(invariant \approves(\this->\owner,t))
})

typedef struct Clock {
    _(ghost AbsClock val)

    volatile unsigned  low;
    _(ghost volatile \natural high)
    _(invariant \mine(&val))
    _(invariant val.t == low + RADIX*high)
} Clock;

void tick(Clock *c _(ghost \claim cl))
    _(always cl, c->\closed)
{
    _(atomic cl,c,&c->val) {
        if (c->low == UINT_MAX) {
            _(ghost c->high=c->high+1)
            c->low = 0;
        }
        else c->low++;
        _(ghost c->val.t = c->val.t+1)
    }
}
```

making locked updates appear atomic

- locks have nothing to do with atomicity; they are just a mechanism to move ownership around
- often we use locks to implement atomic data types
- if we want to make updates to locked data to appear atomic to other threads, we have to couple the protected data with its abstract value
  - this is the obligation of the client, and it's type-dependent, so it belongs in the invariant of the protected object
- the abstract value itself will remain closed, so that clients can have claims on it

locked atomics

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

typedef struct Counter {
    int val;
} Counter;

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

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

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

- .

locked atomics

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

typedef struct Counter {
    int val;
} Counter;

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

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

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

non-hierarchical invariants

- when operating on a large linked data structure, we often want to operate sequentially on a small part of the structure
- this often breaks invariants on the boundry of the updated part (admissibility usually requires that your neighbors are closed)
- the obvious solution is to unwrap the whole structure, but this requires knowledge of all the different node types
- instead, we can add a volatile ghost Boolean to each edge in each node indicating whether the party on the other side is potentially inconsistent (and hence not necessarily closed)
  - these ghost Booleans are approved by the graph, so that it can keep track of which parts of the graph have to be "cleaned up"
- this lets you unwrap only those nodes you have to actually operate on, and then clean up the marked edges (by checking the invariants of the nodes on the boundary)
- this is a typical example of how invariants on volatile ghost field are useful for sequential programming

linearizability

- making an operation linearizable means identifying an external point at which the operation appears to occur
  - in particular, we have to identify which operation occurs when, to avoid attributing the same update to more than one operation
- in a concurrent setting, we can't do this with pre/post
- instead, we use an explicit ghost operation object with a flag that is set exactly when the operation seems to occur
  - an invariant of the operation object says how the atomic object must change state when he goes from not done to done
  - the atomic object has a pointer to the "current" op, to prevent multiple ops from simultaneously getting credit for the same update
- the use of explicit ops allows fancy effects, like one thread helping another by pushing his atomic operation forward (to avoid blocking)
- the owner of the object can either control creation of new ops or pprove changes to the abstract state; each has advantages and disadvantages

polymorphism

- there are two principle ways to write polymorphic functions/data structures in VCC
- you can take an object, making the function polymorphic in the object type (in particular, in its invariant)
  - we used this for locks
- you can express a type as a characteristic function over a fixed supertype, e.g. \object
  - this works for all object types
  - it even works for tuples of object types, e.g. if you have a function from objects to objects, you can express its specification as a characteristic function on pairs of objects

rights

- in VCC, the right to modify o.f is just the knowledge that the modification won’t break any invariants
- approval of o.f is one way to convey a right to a thread/object
  - must be o->\owner
  - must approve all changes to o.f
- another is to give the thread/object control of another object r with an owner-approved volatile field r.f that must change when o.f changes in some particular way
  - fine-grained control
  - can have many different rights
  - downside: have to frame with an object that owns r

```
_(typedef struct Right {
   volatile \bool dummy;
   _(invariant \approves(\this->\owner, dummy))
})

typedef struct S {
   volatile int v;
   _(ghost Right up)
   _(invariant \unchanged(up.dummy) ==> v <= \old(v)))
} S;

void test(S *s _(ghost \claim c))
   _(always c, s->\closed)
   _(maintains \wrapped(&s->up))
   _(writes &s->up)
{
   _(atomic c,s,&s->up) {
      s->v = 0;
      _(ghost s->up.dummy ^= 1)
      _(bump_volatile_version &s->up)
   }
}
```

protectors

- approval is usually the preferred way to stop x from changing
- an alternative is to own a dummy object d (“the protector”), and to allow the x to change in some way only when !d->\closed
  - you can then form a claim that talks about x by claiming d
- downside:
  - you have to use claims (or object invariants) to frame x
  - the code is ugly: you open up the protector while atomically updating x, then close it again after the atomic but before a scheduler boundary
- upsides:
  - you can form a claim on the protector, rather than having to transfer ownership of a Right

```
typedef struct Counter {
   volatile int val;
    _(ghost _(claimable) struct {} up,down)
   _(invariant \on_unwrap(\this,\false))
   _(invariant \old(val) <= val
                || !(&down)->\closed)
   _(invariant val <= \old(val)
                || !(&up)->\closed)
} Counter;
```

```
void counterUp(Counter *c
                   _(ghost \claim cl))
   _(always cl, c->\closed)
   _(writes &c->up)
   _(maintains \wrapped0(&c->up))
{
   int v = _(atomic_read cl,c) c->val;
   if (v < INT_MAX) {
      _(ghost \claim c1 =
         \make_claim({&c->up},
                 c->\closed && c->val <= v))
      _(atomic cl,c) {
         _(assert \active_claim(c1))
         _(ghost \destroy_claim(c1,{&c->up}))
         _(unwrap &c->up)
         _(begin_update)
         c->val = v+1;
      }
      _(wrap &c->up)
   }
}
```

devices

- the environment can be viewed as concurrent threads
- there are two ways to model the behavior of the world
- as a piece of (concurrently running) code
  - “verifying” the device proves that its actions don’t break your invariants
  - necessary for “devices” like MMUs that scribble all over memory
- as an abstract object with a transition relation
  - usually more convenient
  - can have concrete fields (for a memory-mapped device)
  - can use a functional interface to read ghost fields
  - (“hybrid” fields another option)
  - use approval to distinguish program-controlled volatiles from environment-controlled volatiles
  - in many practical examples, you can only use the weak form of approval
  - you can validate this model with an explicit thread model (perhaps with big atomic actions)
- note: you need something more for devices with side-effects from reading, or from writes that don’t change the value (e.g. make these intrinsics)

```c
typedef struct Dev {
    volatile int in;
    volatile int output;
} Dev;

void DevDriver(Dev *dev)
{
    int in = dev->in;

    while (1)
    {
        int output = dev->output;
        if (output == in) {
            in = !in;
            dev->in = in;
        }
    }
}
```

```
typedef struct Dev {
    volatile int in;
    volatile int output;
} Dev;

void DevDriver(Dev *dev)
{
    int in = dev->in;

    while (1)
    {
        int output = dev->output;
        if (output == in) {
            in = !in;
            dev->in = in;
        }
    }
}
```

```
typedef struct Dev {
    volatile int in;
    volatile int output;

    _(invariant \approves(\this->\owner, in))

    _(invariant in==0 || in == 1)
    _(invariant output == 0 || output == 1)
    _(invariant \unchanged(in) || in != output)
    _(invariant \unchanged(output) || in == output)
    _(invariant \on_unwrap(\this,\false))
} Dev;
```

device methodology

- define a protocol that captures what you need to maintain about the device to reason about it sequentially
  - typically will have owner-approved volatiles mirroring aspects of the device
  - can use the same ideas as in managing rights
  - can have multiple protocols managing the same object if the rights can be separated appropriately
- a driver will typically wrap the device inside a protocol object, and use the protocol to mediate his access to the device

```
typedef struct Dev {
    volatile int in;
    volatile int output;
} Dev;

void DevDriver(Dev *dev)
{
    int in = dev->in;

    while (1)
    {
        int output = dev->output;
        if (output == in) {
            in = !in;
            dev->in = in;
        }
    }
}
```

```
typedef struct Dev {
    volatile int in;
    volatile int output;

      (invariant \unchanged(in) || \inv2(\this->\owner))

    _(invariant in==0 || in == 1)
    _(invariant output == 0 || output == 1)
    _(invariant \unchanged(in) || in != output)
      (invariant \unchanged(output) || in == output)
      (invariant \on_unwrap(\this,\false))
} Dev;
```

```
typedef struct Dev {
  volatile int in;
  volatile int output;

   (invariant \unchanged(in) || \inv2(\this-
>\owner))

  _(invariant in==0 || in == 1)
  _(invariant output == 0 || output == 1)
  _(invariant \unchanged(in) || in != output)
  _(invariant \unchanged(output) || in == output)
  _(invariant \on_unwrap(\this,\false))
} Dev;

void DevDriver(Dev *dev)
{
  int in = dev->in;

  while (1)
  {
    int output = dev->output;
    if (output == in) {
      in = !in;
      dev->in = in;
    }
  }
}
```

```
_(typedef struct Protocol {
  Dev *dev;
  volatile int in;
  volatile \bool ready;

  _(invariant \mine(dev))
  _(invariant \approves(\this->\owner,in))
  _(invariant \approves(\this->\owner, ready))

  _(invariant dev->in == in)
  _(invariant ready ==> dev->in == dev->output)
} Protocol)
```

```
typedef struct Dev {
    volatile int in;
    volatile int output;
} Dev;

void DevDriver(Dev *dev)
{
    int in = dev->in;

    while (1)
    {
        int output = dev->output;
        if (output == in) {
            in = !in;
            dev->in = in;
        }
    }
}

_(typedef struct Protocol {
    Dev *dev;
    volatile int in;
    volatile \bool ready;

    _(invariant \mine(dev))
    _(invariant \approves(\this->\owner,in))
    _(invariant \approves(\this->\owner, ready))

     (invariant dev->in == in)
    _(invariant ready ==> dev->in == dev->output)
} Protocol)
```

```
void DevDriver(Dev *dev)
    _(maintains \wrapped(dev))
    _(writes dev)
{
    int in = _(atomic_read dev) dev->in;

    _(ghost Protocol prot)
    _(ghost prot.in = in)
    _(ghost prot.dev = dev)
    _(ghost prot.ready = \false)
    _(wrap &prot)

    while (1)
        _(invariant \wrapped(&prot) && prot.dev == dev)
        _(invariant in == prot.in)
        _(writes &prot)
    {
        int output = _(atomic_read &prot, dev) dev->output;
        if (output == in) {
            _(ghost_atomic &prot {
                prot.ready = \true;
                _(bump_volatile_version &prot)
            })
            in = !in;
            _(atomic &prot,dev) {
                dev->in = in;
                _(ghost prot.in = in)
                _(ghost prot.ready = \false)
                _(bump_volatile_version &prot)
            }
        }
    }
}
```

exercise

- define a toy MMIO disk controller
  - address, data, read/write, one character at a time
- define a (single-threaded) driver that manages the disk as a big virtual map, providing calls to read and write a given address (with suitable abstract semantics)
- define on top of the driver a toy file system, where a file is just a sequence of characters
  - define a block type, sequences of blocks, etc.
  - define directories on top of files

hybrid systems

- introduce time as a ghost object with time moving forward
- God keeps track of which objects in the world are “timed”; when moving time forward, He is obliged to preserve the invariants of timed objects
- timed objects specify their continuous behavior by invariants expressing what they require when time changes (discontinuously)
  - e.g., a physical quantity typically is specified to not change without time changing, and change according to some function of the previous state when time moves forward
- a deadline is a timed object that prevents time from moving past a specified moment
  - once a deadline time is reached, time is frozen and the deadline can never be destroyed
- this allows deadlines to be used to prove safety properties of timed and hybrid systems
- soundness of the use of deadlines depends on proving that every deadline object is successfully destroyed
  - one solution: allow deadlines to be created only
- to prove that deadlines are not reached, you need assumptions that bound how long it can take certain sequential pieces of code to execute
- to use discrete time jumps, keep jumps small enough to not miss catastrophes

```
_(ghost struct Time {
    volatile \natural cur;   // the current time
    volatile \bool timed[\object];  // the set of timed objects
    // timed objects remain timed
    _(invariant \forall \object o; \old(timed[o]) ==> timed[o])
    _(invariant  \old(cur) <= cur && cur <= \old(cur) + 1)
    _(invariant \on_unwrap(\this,\false))  // time never ends
    // timed objects approve changes to time
    _(invariant  \unchanged(cur) || \forall \object o;
        timed[o] && o->\closed ==>  \inv2(o))
} time;)

_(axiom (&time)->\closed);

// timed objects are listed in time.timed
#define TIMED _(invariant  time.timed[\this]) \
    _(invariant (&time)->\closed && \inv2(&time))

#define T (time.cur)

// dT == how far time just moved forward (atomically)
#define dT (time.cur - \old(time.cur))

_(typedef struct Deadline {
    TIMED
    volatile \integer t;
    // keep delta so that we can read or reset the alarm
    // without having to be atomic it time
    volatile \integer delta;
    _(invariant delta + T == t)
    _(invariant T <= t)
    _(invariant \unchanged(t) || \old(T < t)) // stop time on expiration
    _(invariant \on_unwrap(\this, \old(T < t)))
    _(invariant \approves(\this->\owner,t))
} Deadline, ^PDeadline;)
```

```
_(void DeadlineNew(_(ghost Deadline ^d) _(ghost \natural delta))
    _(requires \extent_mutable(d))
    _(writes \extent(d))
    _(ensures \wrapped(d) && d->delta == delta)
    _(ensures \unchanged(T))
    _(decreases 0)
{
    _(ghost_atomic &time {
        d->t = T + delta;
        d->delta = delta;
        time.timed[d] = \true;})
    _(wrap d)
})

#define DeadlineReset(deadline,newDelta) _(ghost {\
  deadline->t += newDelta - deadline->delta; \
  deadline->delta = newDelta; \
})

#define TIMER(name) _(ghost \integer name) _(ghost name = T)
#define READ_TIMER(name) (T - name)
```

```
typedef struct Boiler {
  TIMED
  volatile int level;
  volatile BOOL on;
  _(invariant level ==
    \old(level) + (\old(on) ? (int) dT : 0-(int) dT))
  _(invariant \approves(\this->\owner,on))
} Boiler;

_(typedef struct BoilerCtrl {
  Boiler *b;
  _(invariant \mine(b))
  Deadline ^d;
  _(invariant \mine(d))

  volatile \integer deadline;
  _(invariant deadline == d->t)
  _(invariant \approves(\this->\owner,deadline))

  _(invariant b->level <= 70 && b->level >= 30)
  _(invariant b->on ==> b->level + d->t - T <= 70)
  _(invariant !b->on ==> b->level - d->t + T >= 30)
} BoilerCtrl;)
```

```
void boilerDriver(Boiler *b)
  _(maintains \wrapped(b))
  _(writes b)
  _(decreases 0)
{
  _(ghost Deadline ctrlDeadline)
  _(ghost BoilerCtrl ctrl)
  _(ghost ctrl.d = &ctrlDeadline)
  _(ghost ctrl.b = b)
  TIMER(t0);
  _(ghost DeadlineNew(&ctrlDeadline, 15);)
  _(assert ctrl.d \in \domain(ctrl.d))
  _(ghost_atomic ctrl.d {
    ctrl.deadline = ctrl.d->t;
  })
  // in real code, we would initialize the boilder instead
  _(assume b->level == 50)
  _(ghost (&ctrl)->\owns = {ctrl.d, b})
  _(wrap &ctrl)
  for (unsigned i = 0; i < 10000000; i++)
    _(writes &ctrl)
    _(invariant ctrl.d == &ctrlDeadline)
    _(invariant \wrapped(&ctrl) && ctrl.d->delta > 10 && b==ctrl.b)
  {
    TIMER(t1);
    _(assert &ctrl \in \domain(&ctrl))
    _(assert ctrl.d \in \domain(&ctrl))
    _(atomic &ctrl, ctrl.b, ctrl.d) {
      _(assume READ_TIMER(t1) < 5)
      b->on = (b->level < 50);
      DeadlineReset((ctrl.d), 15);
      _(ghost ctrl.deadline = ctrl.d->t)
      _(bump_volatile_version &ctrl)
    }
  }
  _(unwrap &ctrl)
  _(unwrap &ctrlDeadline)}
```

assembly code

- embedded code (e.g., hypervisors) typically includes some assembly
- assembly instructions are treated as function calls (i.e., given contracts or expressed with inline code)
  - these registers are made of special “hybrid” memory that doesn’t have an address but from which information is allowed to flow into real memory
- when you enter assembly code, the registers satisfy certain conditions specified by the platform ABI
- in practice, reasoning about most assembly code is very easy
- this simple view of assembly code works only for code that doesn’t stomp on control flow (e.g., thread switch by switching stacks)
  - handling nasty control flow within VCC, or any C verifier, is an open problem

progress

- VCC doesn't provide a notion of global progress
- a thread can guarantee its own local progress (in the form of termination)
- a thread cannot depend on progress from other threads (because there is no place to express such progress in shared object invariants)
  - ex: if you put <>p in a type spec, who is responsible for making this happen?
  - can't use the trick from real-time verification because we lack the ordering on time
- a decent framework for "modular progress" is a nice research challenge

encoding other disciplines

- many disciplines for concurrency control can be coded up using admissible invariants and ghost data, e.g.
  - ownership
  - CSL
  - counting permissions
  - fractional permissions
  - deny-guarantee
  - concurrent abstract predicates
- this means that you can use these disparate mechanisms in a single program, or even in a single function/object
- the downside is that you have to explicitly manipulate things in ghost code, rather than depending on a fancy logic to do automatic programming for you
  - doing this in a reasonable way for ghost code might be a good project

refinement

- you can develop your code top-down, refinement-wise in several ways
  - give functions to contracts, but omit giving implementations
  - define types with their abstractions and public invariants, but omit their concrete implementations
  - use block contracts to specify chunks of code without having to fill in implementations

some applications of VCC to real code

- hypervisors
- OS kernel code
- efficient bignum arithmetic
- crypto code
- lock-free optimistic multiversion concurrency control
- lock-free resizable hash tables

conclusion

- admissible invariants and ghost code provide a relatively simple foundation for
  - programmers to write verified code
  - encoding new programming disciplines
- deductive verification != proof checking
  - verification = ghost programming + automatic deduction
- the most important challenges are on the boundary of research and engineering
  - ex: more predictable and scalable deduction
- verified programming is practical now for experts
- verified programming is on the cusp of being practical for ordinary programmers

thanks

- Michal Moskal and Stephan Tobies (the primary developers of VCC); Wolfram Schulte also helped with the early development
- Wolfgang Paul and the Verisoft 2 team: Mark Hillebrand, Norbert Schirmer, Eyad Alkassar, Vladimir Boyarinov, Ulan Degenbaev, Bruno Langenstein, Dirk Leinenbach, Hristo Pentchev, Elena Petrova, Sabine Schmaltz, Andrey Shadrin, Alexandra Tsyban, Sergey Tverdyshev
- Thomas Santen and Markus Dahlweid from MSR ATLE