# The VCC Manual

## Working draft, version 0.2, April 7, 2016

VCC User Manual (working draft, ver. 0.2)


## Abstract

This manual presents the annotation language of VCC, a deductive verifier for concurrent C code.


## 1. Introduction

VCC is a deductive verifier for concurrent C code. VCC takes C code, annotated with function contracts, loop invariants, data invariants, ghost data, and ghost code, and tries to verify that the code respects these annotations. This manual describes these annotations in detail, and what constitutes correct verification behavior of VCC. It does not describe in detail the actual implementation of VCC.

### 1.1 Terminology

***VCC*** means either the VCC tool or the language in which the annotations are written (which is an extension of C), depending on context. Text in this typeface is written in this language, with the exception of text enclosed within angle brackets which is used to give an English description of a command or value or values within such a text, e.g. x == <the absolute value of y>. Mathematical formulas are written as VCC expressions, and models of portions of the state are written as VCC types (with occasional liberties as noted in the text).

In this document, we consider the meaning of a fixed, annotated program, which we refer to as ***the program***. ***The unannotated program*** means the annotated program stripped of all VCC annotations (i.e., the program as seen by the C compiler).

Due to an unfortunate historical legacy, the VCC type \object corresponds to what are in this document called pointers, whereas the word "object" means a particular subset of these pointers.

## 2. Overview

The program operates on a fixed set of typed ***objects***; the identity of an object is given by its ***address*** and its ***type***. (For example, for each user-defined struct type there is an object of that type for each address properly aligned for objects of that type.) Each object has a collection of ***fields*** determined by its type. A ***state*** of the program is a function from objects and field names to values. One field of each object is a Boolean indicating whether the object is ***valid***; the valid objects represent those objects that actually exist in a given state, so object creation/destruction corresponds to object validation/invalidation.

A ***transition*** is an ordered pair of states, a ***prestate*** and a ***poststate***. Each object has a two-state ***invariant*** (a predicate on transitions); these invariants can mention arbitrary parts of the state, and are mostly generated directly from the annotations on the program. The invariant of a type/object is the conjunction of all invariants mandated in this manual. An invariant can be interpreted as an invariant on a single state by applying it to the transition from the state to itself (stuttering transition). A state is ***good*** if all object invariants hold in that state; a transition is good if it satisfies the invariant of each object. A sequence of states (finite or infinite) is good iff all of its states are good and the transitions between successive states of the sequence are good.

A transition ***updates*** an object if some field of the object differs between the prestate and poststate of the transition. A transition is ***legal*** if its prestate is not good or if it satisfies the invariants of all updated objects. A sequence of states is an ***legal execution*** iff the initial state is good and the transition from each nonterminal state of the sequence to its successor is legal. The program is good iff every legal execution of the program is good; successful verification of the program shows that it is good.

A type is ***admissible*** iff, for every object o of the type, (1) every legal transition with a good prestate satisfies the invariant of o, and (2) the poststate of every good transition from a good state satisfies the invariant of o. It is easy to prove by induction on legal executions that if all object types are admissible, the program is good. The program is verified by proving that every type of the program is admissible. (There is a type corresponding to each function; admissibility if this type shows that changing the state of the system according to the operational semantics of the function is legal.)

Each valid object has a Boolean ghost field that says whether it is ***closed***, and a ghost field that gives its ***owner*** (which is also an object). User-defined object invariants are guaranteed to hold only when the object is closed. Threads are also modelled as objects; in the context of a thread, an object owned by the thread is said to be ***wrapped*** if it is closed, and ***mutable*** if it is open (not closed). Only threads can own open objects. The owner of an open object has "exclusive" use of the object, and so can operate on it sequentially. An implicit invariant is that nonvolatile fields of objects don't change while the object is closed, so such fields can also be read sequentially if the object is known to be closed. Fields of closed objects can be updated only if they are marked as ***volatile***; such fields can be accessed only within explicitly marked ***atomic action***s.

Each object field is either ***concrete*** or ***ghost***; concrete fields correspond to data present in a running program. Each concrete field of each object has a fixed address and size. Each object has an invariant saying that concrete fields of valid objects don't overlap, and verification of a function is that it accesses only fields of valid objects; thus, every legal execution of good program is simulated by a legal execution in which concrete field accesses are replaced by memory accesses to shared C heap. The program can include ghost code not included in the unannotated program; however, verification guarantees that all such code terminates, and its execution does not change the concrete state. This implies that every legal execution of the unannotated program is the projection of a legal execution of the program, which allows properties of the program to be projected to properties of the unannotated program.

Each thread has a field that gives its "local copy" of the global state. When a thread updates an object, it also updates the local

copy; when it reads an object, it reads from the local copy. Just before each atomic update, the thread updates its local copy to the global state. It is an invariant of each thread that for any object that it owns, the local copy agrees with the actual object on all fields if the object is open and on all nonvolatile fields if the object is closed. However, assertions that mention objects not directly readable by the thread (without using an atomic action) might reference fields where the local and global copies disagree; thus, assertions that appear in the annotation of a thread are guaranteed to correspond to global assertions only at the beginning of an atomic action. This machinery allows us to "pretend" that threads are interrupted by other threads only just before entering explicit atomic actions.

## 3. Preprocessing

The program must #include <vcc.h>; this inclusion must precede any annotations (after expansion of #include directives).

VCC reserves (in both the program and the unannotated program) the preprocessor macro names VERIFY[1], _VCC_H, and the macro _(...)[2]. All VCC annotations occur within this macro. If VERIFY is not set, _() is defined as whitespace, resulting in the unannotated program. VCC assumes that the unannotated program is a legal C program, i.e. one that a conformant C compiler would accept.

VCC uses the C preprocessor, so comments and macro expansion follow the rules of C. Preprocessor commands are permitted inside of annotations.

## 4. Syntax

VCC reserves identifiers starting with \, using them as function names, field names, or operators.

VCC adds the following infix operators. Each has the given associativity. The specified Unicode characters can also be used.

operator unicode associativity description

| | | | |
|---|---|---|---|
| ==> | | right | implies |
| <== | | left | explies |
| <==> | | left | iff (if and only if) |
| \in | $\in$ | left | set membership |
| \is | | left | type test |
| \inter | $\cap$ | left | set intersection |
| \union | $\cup$ | left | set union |
| \diff | | left | set difference |
| \subset | $\subseteq$ | left | subset |

The precedence of operators, in relation to standard C operators in show below:

/ * %
+ −
<< >>
<= => < > \is \inter
\diff
\union
\in
== !=
&
^
|
&&
||
<==
==>
<==>
?:
= += *= etc.

In addition, there is an additional syntactic productions for expressions:

<expression> ::= <quantifier> <variable declarator> ; <triggers>
<expression>
<quantifier> ::= \forall | \exists | \lambda
<triggers> ::= <empty> | <trigger> <triggers>
<trigger> ::= {<terms>}

The scope of the variables declared in the declarators in the first production extends to the end of the expression. The unicode characters $\lambda$, $\forall$, and $\exists$ can be used as well, in place of \lambda, \forall, and \exists respectively.

The syntax of types is extended with the type constructions defined in section 5.

### 4.1 Annotations

VCC annotations are of the form _(tag stuff), where tag is a ***annotation tag*** and stuff has balanced parentheses (after preprocessing). Each VCC annotation is described here with an entry of the form:
_(tag args) (class)
where args is a sequence of 0 or more parameter declarations (including types) and "class" describes the syntactic role of the annotation, which is one of the following:

**statement** an annotation that acts syntactically as a statement;

**cast** an annotation that acts syntactically as a (type) cast in expressions (in concrete or ghost code);

**contract** an annotation that appears either between a function declaration and the following body or semicolon, or between a loop construct and the body of the loop; of a function or a block (in concrete or ghost code);

**specifier** an annotation that appears in the syntactic role of a function specifier (just before a function declaration or definition);

**compound** an annotation that can appear only in the annotation of a compound type definition, just before the struct or union keyword;

**special** an annotation whose role is described in the text.

Several VCC annotations allow a variable number of arguments of the same type. In such cases, we write the argument as a single typed argument followed by an ellipsis; the actual parameters are to be comma separated iff the ellipsis is preceded by a comma.

In addition to annotations, VCC also provides a number of additional types and functions that can be used inside of annotations. These types and functions all have names starting with \. Their syntax is given using conventional C declarations.

Certain annotations create a ***pure context*** where state changes are not permitted, certain constructs are not allowed, and type promotions are somewhat different. Such contexts are marked :pure in this manual; this keyword does not apepar in annotations.

Certain annotations, constants, and functions can appear only in inside of certain contexts:

**pure** can appear only in a pure context;

[1] The explicit use of #ifdef VERIFY is deprecated.

[2] Caveat: VCC currently makes use of a number of other preprocessor macro names, all of which are defined in <vccp.h> in the VCC inclusion directory. This should be fixed.

**thread** can appear only within the body of a function declaration;

**member** can only appear in the syntactic position of a member declaration of a compound data type;

**invariant** can only appear inside an _(invariant) annotation inside the declaration of a compound type definition.

# 5. Types

A type is said to be concrete if it is a type of the unannotated program, and is otherwise said to be ghost. Each type is classified by the number of possible values of that type, which is either finite, small (countable), or large (equinumerous with the reals). Concrete types are all finite; for each kind of ghost type defined below, we give the class of that type. The class of a struct or union type is the class of its largest member (including ghost fields), so while a C compound type without ghost fields is finite, one with ghost fields might be small or large. The class of an array type is the class of its base type.

All types can be classified as either ***value*** types or ***object*** types. Object types are compound types (i.e., struct or union types), array object types, \threads, \claims, and \blobs. All other types are value types. The objects of a program are of object types, while fields of objects are of value types.

## 5.1 Integral Types

\bool (finite)
The Boolean types, with values \true and \false. These values are equal to the \natural numbers 0 and 1, respectively.

\natural (small)
The type of (unbounded) natural numbers.

\integer (small)
Mathematical (unbounded) integers.

C integral types can be cast to \integer, or unsigned integral types cast to \natural, in the obvious way. When a signed value is cast to \natural in an impure context, there is an implicit assertion that the value cast is nonnegative. In a pure context, the casting of a negative value to a \natural is equivalent to applying some unknown (but fixed) function from integer to natural. In any context, casting any integral type to \bool yields \true if the argument is equal to 0 and \false otherwise.

The C arithmetic operators are extended to the types \natural and \integer with the obvious interpretation[3].

An arithmetic operator instance is said to occur in an \unchecked context if it is is a subterm of an expression marked _(unchecked); otherwise, it is said to be checked. On any checked arithmetic operation, VCC asserts that the operation does not overflow or underflow[4]. On any checked division, VCC asserts that the divisor is nonzero. On any checked cast between arithmetic types, VCC asserts that the cast value fits in the range of the target arithmetic type[5]. The result of an expression of arithmetic type is implicitly cast to \integer if (1) the context is pure and checked, or (2) it is subject to a binary arithmetic operator and the other operand is \natural or \integer.

## 5.2 Record Types

_(record)
A record is like a C struct or union, except that it is treated as a single abstract value. Record types are declared using the same syntax as the declaration of a struct type, but with the following differences:

- The keyword struct is immediately followed by _(record).
- The fields of a record type cannot be marked as volatile.
- Record types have no invariants.
- Fields of a record cannot be of struct types or array types (but can be of map type or record type).
- If a field of a record is of union type, it must have a _(backing_member).

If T is a record type, the expression (struct T) produces an arbitrary value of type T.

If e is an expression of record type with fields f1,f2,..., and e1,e2,... are expressions that are assignment compatible with the types of fields f1,f2,... respectively, then

```
e / { .f1 = e1, .f2 = e2, ...}
```

is equal to the record e with the fields f1,f2,... set to the values e1,e2,... respectively.

If v is a variable of a record type with a field f, then v.f = e translates to v = v / {.f = e}; if v.f is itself a record with field g, then v.f.g = e translates to v = v / {.f = v.f / {.g = e}}, and so on.

_(record T { ... }) (macro)
Expands to _(ghost typedef struct _(record)T { ... } T)

## 5.3 Map Types

T1[T2]
The type of maps (i.e., mathematical functions) from type T2 to type T1. T1 and T2 must be value types, and T2 must not be a large type. If T2 is finite, this has the same class as T1. If T2 is small, the resulting type is large.

Map types can also appear in declarations and typedefs, and there the syntax matches the syntax of C arrays, except that a type appears in between the array brackets instead of a size.

\lambda T v; :pure e (expression)
Here, the scope of v continues to the end of e; it introduces v as a variable of type T. If T1 is the type of e, this expression has type T1[T]; the resulting value maps each v to the corresponding value of e, as evaluated in the current state.

If e is a map of type T1[T2] and x is a value of type T2, then e[x] is an expression of type T1, whose value is given by the value of e at the point x. If v is a lvalue of type T1[T2] and e1 is an expression of type T2, then the expression v[e] = e1 is equivalent to the assignment v = (\lambda T2 u; (u == e)? e1 : v[u]) where u is a fresh variable. This is extended to maps of maps in the obvious way.

[3] Caveat: Because of compiler limitations, VCC currently does not allow << or >> operators with the second argument a literal over 63.

[4] VCC should not allow unchecked signed overflows or underflows (at least in concrete code), because the C standard specifies these as resulting in undefined behavior; however currently VCC assumes that these produce some arbitrary but fixed function of the inputs.

[5] VCC currently does not support floating point arithmetic, in that it does not know anything about the semantics of various floating point operations, and does not check for floating point overflow, underflow, or division by zero. However, on platforms where floating point exceptions do not affect control flow, verification remains sound for programs that uses floats or doubles.

### 5.4 Inductive Types

An inductive datatype is introduced by a top-level annotation

```
_(datatype T {
  case c1(<args1>);
  case c2(<args2>);
  ...
})
```

where <argsi> is a comma-separated list, each element of which is a type name optionally followed by a parameter name. This declares each ctor to be a constructor of values of type T. Types can be mutually recursive, but a type cannot be referenced before being declared. The form _(type T) declares T as an abstract type, which can (but doesn't have to) be later redefined as a record or datatype.

The ==, !=, and = operators are extended to T, but it is an error to apply them to compare an argument of type T and an argument of another type, or to assign between such mismatched types.

The C switch statement is extended to inductive types as follows: if v is an expression of type, VCC allows the program statement

```
switch (v) {
  case c1(t11,t12,...): P1;
  case c2(t21,t22,...): P2;
  ...
}
```

where each tij is a variable name not in scope, no name occurs as more than one of the tijs, each constructor of type T occurs exactly once among T1, T2, ..., and each of the cases has proper arity for the given constructor. Is is asserted that there is no fallthrough between cases (i.e., a break or return at the end of each one is mandatory). The switch statement above translates to the following:

```
{
  unsigned x;
  switch (x) {
    case (1) : {
      T11 t11; T12 t12; ...
      _(assume v == c1(t11,t12,...))
      { P1 }
      break;
    } case (2) : {
      T21 t21, T22 t22; ...
      _(assume v == c2(t21,t22,...))
      { P2 }
      break;
    } ...
    default : { _(assume \false) }
  }
}
```

### 5.5 Pointers

\object (type) (small)
The type of pointers.

^T (small)
The type of pointers to ghost objects of type T. (T can be a concrete or ghost type.)

The program determines a fixed set of ***pointers***. Pointers to instaces of object types are called object pointers; pointers to instances of value types are called value pointers. We typically identify an object and the (unique) pointer that points to it, and so when we talk about an object o, o is really an object pointer.

A pointer is characterized by the following (state-independent) parameters:

\natural \addr(\object p)
The address of p. If !\ghost(p), \addr(p)+ \sizeof_object(p)< UINT_PTR_MAX.

\type \typeof(\object p)
The type of p. In C terms, this can be thought of as the type of object to which p points.

\bool \ghost(p)
True if p is a ghost pointer (i.e., points to the ghost heap). If not, we say p is concrete.

\non_primitive_ptr(\object p)
True if p is an object.

\object \embedding(\object p)
The object of which p is a part of. If p is an object, the result is undefined. If p is a concrete value pointer, \embedding(p) is concrete. When the & operator is applied to an expression of the form o−>f, where f is of value type, the aforementioned object is o. When it is applied to a local variable, it is the object containing the variable.

size_t \sizeof_object(\object p)
Equivalent to \sizeof(T), where T is the type of ∗p. This depends only on \typeof(p).

In a pure context, for pointers p and q, p == q iff p and q agree on all the parameters above.

Every pointer of the program has a footprint (a set of addresses) defined as follows. If p is a concrete value pointer, its footprint is the set of addresses [\addr(p),\addr(p)+ \sizeof_object(p)). If p is an object, its footprint is the union of the footprints of all concrete value pointers q such that \embedding(q)== p. Footprints of distinct concrete value pointers with the same embedding are disjoint. If p is a concrete object, its concrete footprint is a subset of [\addr(p),\addr(p)+ \sizeof_object(p)).

p \is T (expr)[6]
p must be a pointer, and T a value type. This is equivalent to (p == (T ∗)p)|| (p == (T ^)p).

_(retype) (cast)
The argument to this case must be a value pointer p. If there is a \valid object o with a field f such that &o−>f and p point have the same base type and the same address, then this operator returns &o−>f. Otherwise, the result of the operation is undefined. This operator is typically used when p points to a value field of struct that is a member of a union, and some other union member is currently \valid.

VCC does not allow C's silent promotion of an expression of type (void ∗) to another pointer type; such a promotion requires an explicit cast[7].

#### 5.5.1 Pointer Sets

typedef \bool \objset[\object];
\objset is the type of sets of pointers. While the definition above will be its eventual semantics, it is currently treated as a distinct type (because it is triggerred differently from other maps).

[6] Deprecated

[7] This is justified on the grounds that it avoids potential errors without changing the compiled code.

```
_(pure)\bool \in(\object o, \objset s) (infix operator)
_(ensures \result <==> s[o])[8]

_(pure)\objset +(\objset s, \object o) (infix operator)
_(ensures \forall \object o1; o1 \in \result <==> o1 == o || o1 \in s)

_(pure)\objset −(\objset s, \object o) (infix operator)
_(ensures \forall \object o1; o1 \in \result <==> o1 != o && o1 \in s)
```
(+= and −= are extended analogously.)

```
_(pure)\objset \universe()
_(ensures \forall \object o; o \in \result)

_(pure)\objset \everything()
_(ensures \result == \universe())

_(pure)\bool \disjoint(\objset o1, \objset o2)) (infix operator)
_(ensures \result <==> \forall \object o; !(o \in s1)|| !(o \in s2))

_(pure)\bool \subset(\objset s1, \objset s2) (infix operator)
_(ensures \result <==> \forall \object o; o \in s1 ==> o \in s2)

_(pure)\objset \diff(\objset s1, \objset s2) (infix operator)
_(ensures \forall \object o; o \in \result <==> o \in s1 && !(o \in s2))

_(pure)\objset \inter(\objset s1, \objset s2) (infix operator)
_(ensures \forall \object o; o \in \result <==> o \in s1 || o \in s2)
```

### 5.5.2 Objects

Each object has a set of named, typed fields, each of which is either ghost or concrete. Each concrete field also has an offset (of type size_t). The fields of an object are determined by its type.

For object o with field f of value type T, &o−>f is a pointer of type T, with embedding o, which is ghost iff o is ghost or f is ghost. If f is a concrete field, \addr(&o−>f)== \addr(o)+ offset, where offset is the offset of f.

If a concrete object type has padding as part of its representation, the padding is essentially treated as concrete (nonvolatile) fields of value types with unknown field names.

Every object[9] has the following fields and invariants. Note that these are true invariants, holding in all states, not only when the object is \closed. The fields \version, \volatile_version, and \blobifiable below, and the group \ownerOb, are not allowed in annotations. Also, in annotations, fields beginning with \ can appear only following the −> operator (i.e., p−>\owner is allowed, but (∗p).\owner is not)[10].

```
struct <ObjectType> {
  _(ghost volatile \bool \closed)
  _(ghost volatile \objset \owns)
  _(ghost \bool \valid)
  _(ghost \natural \version)
  _(ghost volatile \natural \volatile_version)
  _(ghost volatile \natural \claim_count)
  _(ghost \bool \blobifiable)

  _(group \ownerOb)
  _(:\ownerOb) volatile \bool \owner;
  _(invariant :\ownerOb \unchanged(\owner)
      || ((\inv2(\old(\owner)) || <\old(\owner) not compound>)
        && (\inv2(\owner) || <\owner not compound>)))
  _(invariant :\ownerOb \owner && \owner−>\closed)
  _(invariant :\ownerOb \this \in \owner−>\owns)
  _(invariant :\ownerOb \closed || ((\thread) \owner == \owner))

  _(invariant \forall \object o1; \this−>\closed && o1 \in \this−>\owns
                                    ==> o1−>\owner == \this)
  _(invariant \closed ==> \valid)
  _(invariant \closed && (\forall \object s; s \in \owns
              ==> s−>\closed && s−>\owner == \this))
  _(invariant (\closed || \old(\this−>\closed)) && <\this compound>
              ==> \inv2(\this))

  _(invariant \old(\version) <= \version)
  _(invariant \unchanged(\version) ==> \unchanged(\closed))
  _(invariant \old(\volatile_version) <= \this−>\volatile_version)
  _(invariant \approves(\owner,\volatile_version))
  _(invariant \forall <field f>; <f nonvolatile> && \closed &&
        \old(\closed)
                                    ==> \unchanged(f))

  _(ghost volatile \bool \used)
  _(ghost volatile \objset \subjects)
  _(invariant \closed ==> \used && \unchanged(\subjects))
  _(invariant \this \is \claim && \closed ==>
      \forall \object o; o \in \subjects ==>
              \claimable(o) && o−>\closed)
  _(invariant \this \is \claim && \old(\used) ==> \used)
  _(invariant !\old(\closed) && \closed ==> !\old(\used))
  _(invariant \claimable(\this) && \claim_count > 0 ==> \closed)
  _(invariant \approves(\owner,\claim_count))
  _(invariant \claimable(\this) && (\closed || \old(\closed)) ==>
     \claim_count ==
     <cardinality of>(\lambda \claim c; c−>\closed && \this \in
          c−>\subjects))
  _(invariant <\this of compound type> ==>
        (<type not marked _(volatile_owns)> && \old(\closed) &&
              \closed
          ==> \unchanged(\owns))
     && (<type not marked _(dynamic_owns)> && \old(!\closed)
           && \closed
         ==> (\forall \object o; o \in \this−>\owns
              <==> <there is an invariant of this type with a
                    top−level
                     conjunct of the form \mine(e), and \old(e) ==
                        o>)
     && (\forall <field f>;
          (<\approves(\this−>\owner,f) is a top−level conjunct
            of an invariant of this type>
           ==> \unchanged(\volatile_version) || !\closed(\this)
               || \unchanged(f))))
  _(invariant \old(\valid) && \valid ==> \unchanged(\blobifiable))
}
```

## 5.6 Special Forms in Invariants

```
\bool \wrapped(\object o) (thread)
_(ensures \result <==> o−>\owner == \me && o−>\closed)

_(dynamic_owns) (compound)
_(volatile_owns) (compound)
```
These cannot both appear on the same type definition. See the invariants above and the definition of _(wrap) for their meaning.

```
\bool \mine(\object o1) (invariant)
_(ensures \result <==> o1 \in \this−>\owns)
```

If the object type is not marked _(dynamic_owns) or _(volatile_owns), this function can only be used as the outermost function of a top-

[8] Caveat: this is not its actual definition, for type reasons, but conveys the meaning.

[9] This does not include the :ownerOb groups defined below.

[10] This restriction should be eliminated.

level conjunct of an invariant.

\bool \approves(\object o, f )) (special)
This function can appear only as a top-level conjunct of an object invariant of a type T, f must be a field of T, and o must be a field of T or the expression \this−>\owner. If o is a field other than \owner, \approves(o,o) must also be a top-level conjunt of an invariant of T. \approves(o,f) translates to \unchanged(f)|| !o || \inv2(f)|| (<o is a thread> && !\unchanged(\volatile_version)).

### 5.7 States

\state (large)
The type of states.

T \at(\state s, T e)
The evaluation of expression e in the state s (with all free variables replaced with their value in the current state).

When \at occurs (implicitly or explicitly) in the declaration of a function, purely local variables are implicitly replaced with their actual values at the state in which the form is evaluated. (For function preconditions and postconditions, this state is function entry.)

\state \now()
The current state.

\bool \unchanged(\object o)
For value pointer o, this translates to ∗o == \old(∗o) For object o, this holds iff, for every field f of o, \unchanged(o−>f).

#### 5.7.1 Threads

\thread (small)
The type of threads.

const \thread \me (thread[11])
This is a pure local variable that gives the current thread.

### 5.8 Compound Types

#### 5.8.1 Ghost Fields

Within a compound type, a field is ghost is it occurs inside the scope of a _(ghost) annotation. Ghost fields may be of any concrete or ghost type.

In annotations of a compound type, the scope of a member name includes the entire type definition.

#### 5.8.2 Invariants

_(pure)\bool \inv2s(:pure \state s1, :pure \state s2, :pure \object o)
If o is not of a user-defined compound type, the result is undefined. Otherwise, it returns \at(s2,p), where p is the conjunction of all explicit type invariants in the definition of the type of o, with \old(e) replaced by \at(s1,e) and with \this replaced with o.

```
_(def \bool \inv2(:pure \object o)
{ \return \inv2s(\old(\now()),\now(),o); })

_(def \bool \inv(\object o)
{ return \inv2s(\now(), \now(), o); })
```

_(invariant \bool :pure e) (member)
This defines an explicit invariant for the type in which it occurs. e can mention any part of the state, including static global variables. However, as there is no thread context, it cannot mention \me.[12]

[11] Caveat: currently, VCC does not check this, and should be fixed.

[12] This not currently checked.

An invariant in the declaration of type T forbids unwrapping if it can be violated in an object o of type T by unwrapping o. An invariant that forbids unwrapping must be written in the form _(invariant \on_unwrap(\this,p)). \on_unwrap(o,p) translates to (\old(\this−>\closed)&& !\this−>\closed ==> p).

Only compound types can have explicit object invariants. Other object types have additional implicit object invariants.

:lemma (special)
If the invariant tag is followed by :lemma, then it is asserted that the invariant given follows from the other (non-lemma) invariants of the type. When wrapping, upwrapping, or updating an object of this type, invariants so marked do not have to be checked. Caveat: currently, :lemma invariants of the form \on_unwrap(...) are ignored.

const \object \this
\this can be used only in object invariants, where it refers to the object whose invariant is being defined.

#### 5.8.3 Structs

The infix . operator (selecting a field of a struct or union) is extended in both arguments, to ghost objects and to records (on the left) and to ghost field names (on the right)[13].

The infix −> operator (selecting a member of a struct or union pointed to by the first argument) is extended to ghost pointers and pointers to ghost types and to pointers to records on the left, and to ghost fields on the right.

Ghost struct declarations need not declare any members (unlike in C).

VCC currently does not allow arrays of size 0 at the end of a struct (these are allowed in C99).

_(inline) (member)
This annotation must come directly before declaration of a member with a struct type. The effect is that the inner object is not considered to be an true object; instead, its fields are semantically fields of the parent struct. Invariants of the inner struct are ignored.

A type definition of structure type defines two types, the usual one and a volatile type. The volatile type differs from the nonvolatile type only in that all of its explicit fields are volatile and it has none of the explicit invariants of the original type. Instead, the type gets an implicit invariant that each of its fields is owner-approved.

When a field of struct type is marked as volatile, and that field is of a struct type, the instance is of the volatile variant of that type. If the type is inlined, the individual fields are inlined as usual, but each of the fields is as if it was declared to be volatile.[14]

***Groups*** A group object is an artificial object that lies inside a concrete struct. The physical address and size of a group are equal to those of the containing struct. The fields of a group need not be contiguous. (This means that the fields left to the containing struct need not be contiguous either.)

[13] Caveat: VCC currently does not support this operator when the name begins with ; the −> operator must be used instead.

[14] This is standard C behavior. It is most often seen when a field is declared as volatile T ∗p, which is a pointer to volatile T, not a volatile pointer to T. This is usually a mistake. To get a volatile field, one needs to declare typedef T ∗PT and then use volatile PT p.

_(group G) (member)
_(group G, <forms>) (member)
These forms declares the identifier G to be the name of a nested ghost struct. The scope of G runs from the point of this declaration to the end of the declaration of the compound type in which it appears. If the second form is used, the <forms> are a comma-separated list of modifiers, each one of _(dynamic_owns), _(volatile_owns), _(record), or _(claimable); these attributes are used as attributes of the newly defined struct type. The type of the declared group is written as T::G, where T is the containing struct type. If e is an expression of type T *, (e :: G *) abbreviates ((T ::G *)e).

_(:G) (member)
It declares that the immediately following fields (until semicolon) are semantically fields of G, rather than fields of the containing struct. The explicit labelling allows the fields of the group to not be contiguous.

_(invariant :G p) (member)
This form declares an invariant for group G defined earlier in the current struct. It follows the same rules as ordinary object invariants; in particular, it can mention members of G that are not yet declared.

#### 5.8.4 Unions

If a union includes a member of value type, exactly one of these members must be be marked _(backing_member). The size of the backing member must be at least as large as the size of the other members of the union. It is a VCC error to apply & to a member of a union with a backing member. However, all concrete members of a union have the same address.

Members of ghost union types do not necessarily alias, so assignment through one member need not be reflected in the value of another member.

In any state, if a union is \valid, exactly one of its members is \valid. When a union is created (by an explicit or implicit \unblobify), this member is the backing member (if there is one), and is otherwise the first member of the union.

### 5.9 Array Object Types

An array object is an object that serves as a container for an array (whose elements are typically of value type). They are typically used to allow ownership transfer of the elements of the array.

An array object type is defined by the base type and length of the array. If the base type of the array is a value type, the elements of the array are effectively fields of the array. If the base type of the array is an object type, the array object has no fields (other than the implicit ones present in all objects), and so is not very useful. Note that array object types are distinct from C array types; array object types cannot be defined via typedef, nor can a variable be declared to have an array object type. Expressions of array object type can only be created by casting: if T is a type and e is an expression of integral type, the expression (T[e])v casts v to a pointer to an array object with base type T and length e (where e is evaluated in the current state)[15]. If the base type of an array is volatile, the array object has an implicit invariant that all changes to the array are owner-approved.

[15] Note that this syntax doesn't create a conflict with ordinary C array types, because C does not allow casts to array types.

_(root_index \type T size_t s)
This macro should be applied only to a pointer o of type T*, and expands to _(T *)_(T[s]). This has the effect of giving back o but, if o is of value type, changing its embedding to the array object _(T[s])o.

\is_array(T *p, size_t s) (expr)
True iff p is a pointer to an array object of size s.

### 5.10 Blobs

\blob (small)
The type of (pointers to) blobs.

A blob is an uninterpreted contiguous portion of the concrete address space, much like a struct of a given size with nothing but padding. A chunk of memory starts out as a blob. This blob can be broken into smaller blobs, joined into larger contiguous blobs, or turned into a collection of objects (essentially, the \extent of the outermost object). The outermost object made from the blob is said to be "blobifiable"; only blobifiable objects can be blobified. (This is to avoid accidentally making multiple objects of the same type with the same address.)

When a new object is allocated (either through a call to malloc or entry to a block with with a local variable that is not purely local), it is implicitly created as a blob of appropriate size and alignment, then implicitly unblobified to the object of specified type.

To change a piece of memory from holding one type of object to holding another, the first object must turned into a blob, this block might be merged with other blobs or split into smaller blocks, and finally a blob is turned into a new object.

_(blob size_t sz)) (cast)
This returns a pointer to a blob of size s with the same address as ptr. It has no effect on the state.

_(blobify \object o) (statement)
This requires \extent_mutable(o)) and o is blobifiable. It has the effect of making all objects in \extent(o) invalid and making valid a blob of size sizeof(o) with the address of o.

_(join_blobs \blob a, \blob b) (statement)
This joins a and b into a single blob. This requires that the address of b is the address of a plus the size of a, and that a and b are mutable.

_(split_blob \blob a, \natural s) (statement)
This requires s is positive, a is \mutable and of size greater than s. The result is to split a into two contiguous blobs, the first of size s and the second of size the original size of a minus s.

_(blob_of) (cast)
The argument must be an object. Applying it to an \object o yields the same result as _(blob \sizeof(o))o.

_(unblobify) (cast)[16]
Applying this to object o requires that there is a \wrapped blob with the same address and size as o. The effect is to make all of the objects in \extent(o) \mutable, marking o as blobifiable and all other objects in \extent(o) as not blobifiable. If the type of o involves unions, the backing member is chosen as the "active" member of

[16] This is provided as a cast rather than a statement, because legacy code often requires unblobification in the middle of an expression.

the union; if the union has no backing member, the first union member is chosen.

_(union_reinterpret \object o)
This requires o to be of the form &(x->f), where x is a pointer to a union type and f is a member of the type. It also requires that the union is mutable, and that the extent of the currently active member of the union (if that memver is of object type) is mutable. The effect is to make f the currently active member of x i.e., to make its extent mutable (choosing union members of nested unions as in _(unblobify).

### 5.11 Claims

\claim (small)
The type of (pointers to) claims.

A compound type (concrete or ghost) can be marked a _(claimable) (just before the struct keyword in its type definition). Claimable objects cannot be unwrapped when their \claim_counts are nonzero.

A \claim has no data beyond those in every object, but can have an additional invariant. For any predicate p that does not use \old(), there is a type of \claims with invariant p, as long as the resulting type is admissible.

Conceptually, when there is an atomic update, the invariants of updated objects are checked first, assuming only the 1-state invariants of all objects in the prestate. (This is the legality check.) Next the invariants of all unupdated non-\claim objects are checked (this is admissibility check, in reality performed per-type, not per-update); this check assumes the 1-state of all objects in the prestate, and the invariants of all updated objects. Finally, the invariants of all claims are checked; this proof gets to additionally assume that all objects (including unmodified ones) satisfy their invariants. This final check is, like the admissibility check, performed once for claim type.

_(pure)\bool \claims(\claim c, :pure p)
This holds iff, in every good state in which c is closed, p holds.

_(pure)\bool \claims_object(\claim c, \object o)
True iff o is one of the claimed objects of c.

_(pure)\bool \active_claim(\claim c)
True iff c is closed. However, it has the effect of also asserting the invariant of c. Asserting or assuming \active_claim(c) in a state (perhaps implicitly) is the only way to trigger instantiation of the invariant of c in that state.

\bool \claims_claim(\claim c, \claim c1)
(Deprecated) Equivalent to \claims(c, c1->\closed && \valid_claim(c1)), but triggers somewhat differently.

\bool \claimable(\object o)
True iff o is of an object type that was declared as _(claimable).

\claim \make_claim(\objset s, :pure \bool p) (special)
The set s is evaluated in the current state, while p is not. p cannot use \old(). This translates roughly to the following:

```
_(assert p)
_(assert <the claim type with invariant (p && \subjects == s) is
      admissible>
\claim c;
_(assume <s is the set of claimed objects of c and \vcc{p} is the
      invariant of s>)
_(assume !c->\used && !c->\closed)
_(ghost_atomic c,s {
    c->\subjects = s;
    _(wrap c)
    foreach \object o \in s;
      _(assert \non_primitive_ptr(o))
      o->\claim_count ++;
      o->claimants += c;
})
```

\claim \upgrade_claim(\claim c1,..., bool p)) (statement)
This creates a new claim that claims the invariants of c1,c2,... conjoined with p, while unwrapping c1,c2,.... (It thus writes c1,..., but does not write the subjects of c1,....) The subjects of the result claim are the sum of the subjects of c1, c2, .... Note that if the input claims claim a common object, the number of objects claiming that object will fall below the \claim_count of the object, making it unable to open again, so this should normally not be the case for concrete objects.

_(pure)\bool \account_claim(:pure \claim c, :pure \object o)
This is semantically equivalent to (\wrapped(c)&& \claims_object(c,o)). It exists to trigger the following axioms:

```
(\forall \claim c1,c2; \forall \object o; {\account_claim(c1,o),
      \account_claim(c2,o)}
    \account_claim(c1,o) && \account_claim(c2,o) && c1 != c2
        ==> o->\claim_count >= 2)
```

and similarly for up to 4 distinct claims.

_(pure)_(by_claim :pure \claim c) (cast)(thread)
The target of the cast must be of the form o->f, where f must be a nonvolatile[17] field of o. It evaluates to o->f, but also asserts \active_claim(c)&& \claims(c,o->\closed).

\bool \always_by_claim(\claim c, \object o)
o must be an identifier. This expression evaluates to \true. Asserting this as a top-level conjunct causes each subsequent occurrence of o->f (until the end of the current block), where f is a nonvolatile field of o, to be replaced with (_(by_claim c)(o->f)).

void \destroy_claim(\claim c, \objset s)
This asserts that c is \wrapped and \writable, and that it claims every object in s (though it might claim others). It has the effect of opening c and decrementing the \claim_cnt of each object in s.

_(pure)T \when_claimed(T :pure e) (special)
This macro can appear only inside of \claims or in the second argument to \make_claim. It translates to \at(s,e), where s is the state in which the surrounding \make_claim is being executed, or the state in which the surrounding \claims is being evaluated. (If inside an _(ensures), s is the state in which the function returned.)

Within a program block, local variables (purely local or not) occurring inside the second argument of \claims or \make_claim are implicitly wrapped in \when_claimed.

## 6. Declarations

In general, declarations follow the C conventions regarding declarations and scoping, with the following exceptions.

Ghost declarations are enclosed in _(ghost), or in ghost code blocks.

[17] Caveat: this should be generalized to volatile fields, where c guarantees that the field has the same value in all states where c is valid, but this extension is not yet supported.

Top-level ghost declarations (global variables, typedefs, and function declarations) can be added anywhere a top-level C declaration can appear. Local ghost variable declarations can appear wherever a C statement can appear. Ghost identifiers use the same namespace as ordinary C identifiers; is an error to shadow one with the other.

The scope of a struct or union declaration, as well as the fields of such declarations, is extended to include all annotations within the entire compilation unit in which it appears. (However, a typedef name declared along with it has the usual scope, from point of declaration forward.) This exception to the scoping rules of C is needed to allow type invariants to mention the fields of types that have not yet been declared (since C provides no mechanism for forward declaration of fields).

If the identifier T is not in use as an annotation tag, the top-level form _(T <decl>) abbreviates _(ghost T <decl>).

### 6.1 Purely Local Variables

A purely local variable is a local variable of a function (or a function parameter) such that neither the variable nor any nested member of the variable (if it is of a compound type) is subject to the & operator[18]. (That is, a variable is purely local if its address is never taken.) Such variables are given special treatment by VCC, because they can be modified by a program fragment only if it mentions the variable by name. (In particular, it cannot be modified by a function call.) Formal parameters of a function (because they are not lvalues) are always purely local.

In a loop, purely local variables that are not updated are automatically asserted to not change during the loop, without requiring an explicit invariant. Conversely, purely local variables that are updated are implicitly included in the writes clause for the loop, and so do not have to be explicitly listed [19].

Purely local variables are always considered writable, and so do not have to be mentioned in writes clauses for blocks[20].

A purely local variable of struct type is treated as a collection of purely local variables, one per field, and so each field gets separately the treatment described above.

### 6.2 Global Variable Declarations

For every global variable v of value type, there is a dummy object of struct type of which v is the only field (outside those of all objects). If v is declared as volatile, this dummy object has the implicit invariant _(invariant \approves(\this−>\owner,v)) footnoteWithout this invariant, there would be no way to admissibly maintain any information about the value of the variable, making it useless for verified programming. .

\bool \program_entry_point()
True iff all global variables of object types, and the dummy objects owning global variables of value types, are \mutable. This function is allowed only in a thread context[21].

[18] Caveat: a variable should remain purely local if its address is taken only inside of _(writes), _(wrap), and _(unwrap). Finer-grained escape analysis might also be possible, to allow a function spec to declare that its use of a variable ends on function return.

[19] Currently, mentioning the variable explicitly would ruin its pure locality.

[20] This feature is depricated, and will likely be removed in future.

[21] Caveat: currently VCC does not check this. This should be fixed.

### 6.3 Function Declarations

A function declaration, with or without a body, can include a specification. If multiple declarations of the same function include specifications, their specifications must be identical. A specification consists of a nonempty sequence of annotations of the following forms:

_(requires :pure b) (contract)
This puts an obligation on all callers of the function that at the point of the call (after formal parameters have been bound), b should evaluate to a nonzero value (i.e., b is asserted at each call site). This also introduces the assumptionb to be added at the beginning of the function body.

b (or a top-level conjuncts of b) can be of the form _(assume b). The effect is to assume b on every call to the function (i.e., to omit the _(assert b) that would otherwise be generated.

_(ensures :pure b) (contract)
This asserts b at each return point from the function. This also allows callers of the function to assume b when the function returns. Within b, \old(e), where e is an expression, means the value e evaluated to on function entry (after binding of parameters).

_(writes :pure \object o1, o2, ...) (contract)
_(writes :pure \objset s) (contract)
The first form is shorthand for _(writes {o1,o2,...}). If no such forms appear, there is an implicit form _(writes {}). Multiple forms are tacitly combined by rewriting _(writes s1)_(writes s2) to _(writes s1 \union s2).

For a function with exactly one form _(writes s) of this type, the form is, for callers of the function, semantically equivalent to

```
_(requires \forall \object o; o \in s ==> \writable(o))
_(ensures \forall \object o; !(o \in s) ==>
    (\non_primitive_ptr(o) && \old(o−>\owner == \me)
     ==> \unchanged(o−>\version)
         && \unchanged(o−>\volatile_version))
   && (!\non_primitive_ptr(o) && \old(\mutable(o)) ==>
        \unchanged(o)))
```

However, within the body of f, a stronger condition is used: a sequential write to o requires \writable(o).

_(reads :pure \object o1,...) (contract)
This annotation can appear only on _:pure functions. If nultiple reads annotations are on the same function, their lists are concatenated to form a single list. This annotation introduces an additional proof obligation that, for any two states that differ only on the values of fields of o1,..., the function returns the same result. It also allows generation an axiom to that effect (to be used in the verification of other functions), unless the function is also annotated with _(no_frameaxiom) annotation.

_(maintains \bool p) (contract)
Equivalent to _(requires p)_(ensures p).

_(always \claim c, \bool p) (contract)
Equivalent to

```
_(requires \wrapped(c) && \claims(c, cond))
_(requires \assume(\active_claim(c)))
_(ensures \wrapped(c))
_(ensures \assume(\active_claim(c)))
```

_(updates \object o) (contract)
Equivalent to

```
  _(requires \wrapped(o))
  _(ensures \wrapped(o))
  _(writes o)
```

_(out_param(\object p) (contract)
Equivalent to

```
  _(writes p)
  _(maintains \mutable(p))
```

_(returns e)
Equivalent to _(ensures \result == e)

### 6.4 Function Declarations

A function specification consists of a nonempty sequence of annotations with the tag requires, ensures, or writes (or macros that translate to these annotations). A function specification can appear immediately after a function declaration (before the ending semicolon), in a function definition (between the declaration and the body), or immediately before a program block. All specifications on function declarations or definitions of the same function must have the identical specifications.

A typedef of a type that includes a function application can include specifications for the function(s) specified in the defined type; these specifications are given just before the semicolon terminating the typedef. For example:

```
typedef void *Sort(size_t size, int *b)
_(requires \mutable_array(b,size))
_(writes \array_range(b,size))
_(ensures sorted(b,size))
;
```

#### 6.4.1 Ghost Parameters

Before the closing parenthesis of the parameter list, ghost parameters to the function can be declared using the two forms below. Note that such parameters are not comma-separated. These parameters obey the usual rules of function parameters (their names cannot conflict with names of other parameters, their scopes are the entire function declaration, etc.).

_(ghost decl) (special)
This declares a ghost parameter of the function. Such parameters behave just as ordinary function parameters, e.g. they are passed by value.

_(out decl) (special)
Here, decl is an ordinary parameter declaration (i.e., one that could appear as a parameter of a ghost function). This deckares an out parameter of the function. The parameter cannot be mentioned in _(requires) or _(writes) specifications, and is considered writable in the body of the function. On function return, the value of the formal parameter is copied to the variable passed as an actual parameter.

#### 6.4.2 Atomic Inline Functions

_(atomic_inline) (specifier)
If a function is marked _(atomic_inline), (1)calls to the function are replaced by the body (with function parameters assigned as local variables), and (2) the body of the function is treated as an atomic action (which means that it can appear inside an atomic action). Such functions cannot have specifications.

#### 6.4.3 Pure Functions

A pure function is one that doesn't modify the state (as seen my its caller and other threads), and whose return value can be treated as a mathematical function of the state. Only pure functions can be used in specifications and assertions. This mathematical function is defined to be an arbitrary one such that, for any state satisfying its preconditions, the result satisfies its postconditions. For states not satisfying its preconditions, the mathematical function takes on an arbitrary value of the result type[22].

_:pure (specifier)
This declaration can appear only after the return type declaration of a function declaration or definition. A function marked in this way is subject to the following restrictions:

- the function can assign only to local variables;
- its local variables must be purely local;
- it can use _(ghost_atomic) actions, but not other atomic actions;
- it can call only pure functions;
- if it is not a ghost function, its local variables must be initialized before they are read[23].

A warning is issued for a pure function that is not given a body[24].

_(def <decl>)
This is an abbreviation for _(ghost _:pure _(inline)<decl>).

_(pure <decl>)
_(abstract <decl>)[25]
These are abbreviations for _(ghost _:pure <decl>).

#### 6.4.4 Custom Admissibility Checks

_(admissibility) (specifier)
_(admissibility) declares a function as a custom admissibility check. The function function must take a single argument having type a pointer to a struct type; verification of the function is used instead of an admissibility check of the struct type. The function adds an explicit postcondition that _(havoc_others) is called exactly once, and that the function does not otherwise modify the state. It should modify only local variables, and should not take the addresses of any of these variables to assure that they are not heapified. Finally, there is an implicit postcondition that \inv2s(s1,s2,o), where s1 is the state on entry, s2 is the state on exit, and o is the function parameter.

_(havoc_others \object o) (statement)
This statement can appear only in functions marked _(admissibility). It changes the state to an arbitrary \legal state, without changing any fields of o.

#### 6.4.5 Termination

If a function, block, or loop is annotated with a _(decreases) annotation, it is guaranteed to terminate (in any infinite execution that includes an infinite number of steps by the thread entering the function, block, or loop).

_(recursive_with f1,f2,...) (contract)

This annotation can appear only on the specification of a function f; we say that f "publicly calls" each of the functions f1,f2,....

A function f "calls" function g if the body of f includes a call to

[22] Such a value always exists, because all value types are nonempty.

[23] Caveat: this check is not currently done.

[24] VCC currently supports pure functions without function bodies, but with postconditions of particular forms. However, the restrictions on such functions are complex, and their use is deprecated.

[25] Depricated

g; if the body of f is visible to VCC, f "visibly calls" g. Let "apparently calls" by the union of "visibly calls" and "publicly calls". Let "transitively calls" be the transitive closure of the "calls" relation, and let Let "visibly transitively calls" be the transitive closure of the "publicly calls" relation.

For termination verification to be sound, VCC requires, but does not check, the following condition: if function f in the verification unit calls function g that is not in the verification unit, but does not publicly call g, g transitively calls f, and the call to g does not terminate, then g visibly transitively calls f.

_(decreases :pure \natural t1, t2, ...) (contract)

A _(decreases) annotation can appear only in the specification of a function or a block. Within the function body or block, this defines a function from states of the body or block to finite tuples of \naturals, given by the value of the tuple <f1,f2,...>. If the annotation is on a function, this measure is extended to calls of the function by evaluating this tuple after the binding of formal parameters.

Any function declared _:pure that does not have a _(decreases) clause in its specification is given implicitly the annotation _(decreases p1, p2, ...) where the sequence p1,p2,... is obtained from the sequence of function parameters by removing all parameters that are of neither integral nor record type, removing all parameters of type \bool, and applying size() to any arguments of record type.

In the body of a function f with a _(decreases) clause, on any call to a function g that visibly transitively calls f, it is asserted that the measure of the call to g is smaller than the measure of f on entry to f, where measures are ordered as follows:

```
<x1,x2,...> < <y1,y2,...> <==> x1 < y1 || (x1 == y1 && <x2,...> <
      <y2,...>)
<x1,...> = <0,x1,...>
```

the latter equation used to compare tuples of unequal length.

If a _(decreases) clause is put on a loop, VCC asserts on any return to the head of the loop from within the loop that the measure on loop is smaller than the value of that measure on loop entry. If a while or for loop without a _(decreases) clause appears in the body of a function or loop with a _(decreases) clause, and the loop test is of the form e1 < e2, e1 <= e2, e1 > e2, e1 >= e2, or e1 != e2, the loop is given a default clause _(decreases abs(e1 − e2)), where abs is the absolute value function (on \integer).

# 7. Expressions

This section covers only expression forms not in C and not covered earlier in section 5.

## 7.1 Logical Operators

\bool <==>(T1 op1, T2 op2) (infix operator)
\bool ==>(T1 op1, T2 op2) (infix operator)
\bool <==(T1 op1, T2 op2) (infix operator)
T1 and T2 must be integral types. (x ==> y) is \true iff x == \false or y != \false
(x <== y) is equivalent to (y ==> x). (x <==> y) is \true iff (x ==> y) and (y ==> x).

### 7.1.1 Quantifications

\forall <decl>; :pure \bool p (expression)
\exists <decl>; :pure \bool p (expression)

A quantification has one of the forms where <decl> is a variable declaration. In each variable declarations, the scope of the declared variable is to the end of the expression. Optional patterns may appear after the semicolon (§ **??**). The forall/exists evaluates to \true iff p evaluates to a non-\false value for each/some possible instantiation of the quantified variables.

## 7.2 Function Calls

In a call to a function with declared _(ghost) or _(out) parameters, the corresponding arguments of a call must be filled with parameters of the form _(ghost e) or _(out v), where e/v is an expression/-variable of suitable type. As in the declaration, these parameters must not be comma-separated.

# 8. Statements

## 8.1 Wrapping and Unwrapping

_(wrap :pure \object o1, ...)(statement)
This translates roughly as follows[26]:

```
foreach (\object o \in o1,...) {
  _(assert \mutable(o) && \writable(o))
  _(assert \forall \object o1; o1 \in o−>\owns ==> \wrapped(o1) &&
        \writable(o1))
  if <the type of o is not marked _(dynamic_owns) or
        _(volatile_owns_)>
    o−>\owns = {};
    foreach <top level conjunct \mine(t) in the invariant of o>
      o−>\owns += {t};
}
_(ghost_atomic o1, ... {
    foreach (\object o \in o1, ...) {
      foreach (\object x \in o−>\owns) x−>\owner = o;
      o−>\closed = \true;
      _(bump_volatile_version o)
    }
})
```

_(unwrap :pure \object o, ...) (statement)
This translates roughly to

```
  _(assert \wrapped(o) && \writable(o))
  _(ghost_atomic o {
    o−>\closed = \false;
    _(bump_volatile_version o)
    foreach (\object o1; o1 \in o−>\owns)
      o1−>\owner = \me;
  })
```

### 8.1.1 Unwrapping Blocks

An unwrapping block has the form _(unwrapping \object o1, \object o2, ...)W B (statement)
where writes is a sequence of _(writes) annotations and block is a program block without a specification. Let S be the set of objects listed in writes, interpreted at the entry to the unwrapping block. The block then translates as follows:

```
_(ghost \state st = \now())
_(ghost \objset W = \at(st, <set of objects listed in writes clauses>))
_(ghost \object o1 = \at(st,o1))
_(ghost \object o2 = \at(st,o2))
...
_(unwrap o1)
_(unwrap o2)
...
block
...
_(wrap \at(st,o2)) _(assert \domain(o2) == \at(st,\domain(o2)))
_(wrap \at(st,o1)) _(assert \domain(o1) == \at(st,\domain(o1)))
```

[26] Caveat: currently, _(wrap) and _(unwrap) cannot be used in atomic actions (except before _(begin_update). This restriction should be lifted.

```
_(assert \forall \object s,x; s \in {o1,o2,...} && x \in \domain(s)
  ==> (*x == \at(st,*x) || x \in W))
```

where o1,o2,... are fresh variable names. If the the block has no writes clauses, the last assersion is omitted. If the _(unwrapping) is not followed by a program block, it is equivalent to a program block without writes clauses.

\bool \domain_updated_at(\object d, \objset s)
This macro translates to

```
_(unchanged \domain(d))
&& (\forall \object o;
      \old(o \in \domain(d) && (\forall \object c; c \in s ==> !(o \in
            \domain(s))))
      ==> \unchanged(o))
```

Currently, VCC will verify such a postcondition only under some very specific conditions. First, s must be syntactically presented as an explicit set of explicit paths through the domain of d (e.g., d−>a, d−>b−>c−>e). Second, the verification will fail if d is unwrapped with _(unwrap d). The only modifications allowed to \domain(d) are through

- a direct assignment
- a function call or block with a contract that writes a, such that the function or block specification implies (under the conditions in which it is called) _(ensures domain_updated_at(d, s1)) where \subset(s1,\at(st,s)))and where \vcc{st is the state at entry to the containing function body
- an unwrapping block whose writes within \domain(d)

### 8.2 Expression Statements

As in C, an expression, followed by a semicolon, is a statement; it is executed by evaluating the expression. In a ghost context, VCC does not allow expression statements without side effects; an expression statement must either contain an assignment or a call to a non-\pure function.

If the expression statement reads a field of an object, there is an implicit assertion that the object is \thread_local. If the expression writes a field of an object, VCC asserts that the field is \writable.

For each read of a variable v, VCC asserts that v is either \thread_local or is a volatile field of an object in the read set of an enclosing atomic action.

For each write of a variable v, VCC asserts that v is either mutable and \writable, or is a volatile field of an object in the writes set of an enclosing atomic action.

### 8.3 Block Contracts

If a program block immediately preceded by one or more function annotations with tag requires, ensures, writes, or pure, these annotations are used as specifications for the block, with \old(e) meaning the value of e at entry to the block. Any postconditions are enforced for the block, even if control exits via a goto to an outside location.

If a block has among its preconditions _(requires \full_context()), the block is verified using full context from the preceding code of the function. Otherwise, the block is verified using only those preconditions listed explicitly in the block contract.

If a block has among its postconditions _(ensures \full_context()), then the full context of the block is available on exit from the block. Otherwise, only those postconditions explicitly given to the block are exposed on exit. In this case, if the block is marked _:pure, then the block is treated as a pure function call; otherwise, it is treated as an impure function call (wrt. the havocing of state).

### 8.4 Assertions and assumptions

_(assume :pure b)
This is equivalent to a call to a pure function with the contract

_(ensures b)

[27]

_(assert :pure b) (statement)
This is equivalent to a call to a pure function with the contract _(requires b).

_(assert {:bv} :pure b)
b cannot include free variables or reference the heap, and its quantifications must be over finite types. This is equivalent to _(assert b), but requires b to be proved using bitwise reasoning.

### 8.5 Ghost Statements

_(ghost stuff) (statement)
This introduces a ghost statement. Here, stuff is a declaration or a statement (which might be a block statement). If a declaration, it does not have to precede all statements in the current block. stuff follows the syntax of C statements, but can use the operators and functions described in this document. stuff does not have to be terminateed by a semicolon.

### 8.6 Iterative Statements

The keywords while, do, and for) can be succeeded by a sequence of annotations of the form _(invariant e), _(writes) clauses, or _(ensures e). Such annotations belong to the loop itself, and not to the program block (if any) that follows the keyword; the following block cannot have a contract, though blocks nested inside can. If the keyword is not annotated by any _(writes) annotations, it is implicitly so annotated with the _(writes) set of the immediately surrounding loop if it is nested inside another loop, and otherwise with the _(writes) set of the function. For every purely local variable v that is not explicitly update in the loop, there is an implicit loop invariant _(invariant v == \at(s,v)), where s is the state at loop entry.

VCC does not allow control transfer into the body of a loop, but does allow a goto to transfer control out of loops. Gotos within ghost code cannot use a target outside of ghost block in which the goto occurs.

VCC requires that the control flow graph of a function body is reducible. This allows it to translate the function into properly nested blocks, with only forward gotos. In doing this, a label within the body may become a head of a loop of the form of while (1){...}. If this is done, the invariant associated with that loop is given by any explicit assertions that immediately follow the label. However, there is no way to put an explicit writes clause on such a loop.

### 8.7 Atomic Actions

#### 8.7.1 Closed Object Lists

The atomic actions all make use of ***closed object lists***. A closed object list is a nonempty, comma-separated list of objects each possibly preceded by _(read_only). Those objects in the list that are not marked _(read_only) are called the write set of the list.

[27] A common idiom is to use _(assume 0) to prevent verifications of assertions that follow. It should be noted that VCC can often tell that the following code is unreachable, causing later jumps to be ignored when constructing the control flow graph of the function. In a loop body, this has the effect of causing VCC to verify the body only for the first iteration through the loop, rather than an arbitrary one. This can be avoided by assuming something like x!= x.

Intuitively, a closed object list represents a set of objects that are known by this thread to be closed, and are guaranteed to remain closed despite legal actions by other threads. Each object of the closed object list must either be \wrapped, or its closedness must follow from the invariants of objects earlier in the list (along with thread-local information). The list included as part of an atomic action is required to include all objects that are read or written in the atomic action (other than those that are \mutable).

To validate an object o in the list is to _(assert \active_claim(o)) if o is a \claim and _(assert o−>\closed && \inv(o)) otherwise. To validate an object list is to validate its elements in left-to-right order. A validation of the list consists of this sequence of assertions performed in a particular state. The read objects of a validation is the set of objects ocurring in the list, along with the union of sets ocurring in the list, as evaluated in the validation state. The set of write objects of a validation is defined the same way, but omitting those elements annotated as _(read_only).

Execution of an atomic block translates approximately as follows:

```
\state s = \now();
<set the current state to an arbitrary good state>
_(assert \me == \at(s,\me))
_(assert \forall \object o; \at(s, \wrapped(o) || \mutable(o))
     ==> \unchanged(o−>\version) &&
           \unchanged(o−>\volatile_version))
s = \stutter(\now());
<translation code block before _(begin_update)>
<validate obs>
s = \now();
<translation of code block after _(begin_update)>
foreach (\object o \in obs not marked _(read_only))
  _(assert \inv2s(o, s,\now()))
```

#### 8.7.2 Ghost Atomic Actions

_(ghost_atomic obs block) (statement)
Here, obs is a closed object list and block is a code block. The block is considered to be in a ghost context. Operationally, this executes block atomically.

The ghost block is subject to the following restructions (beyond those normally in effect for ghost code). Only pure function calls are allowed, and all such calls must occur before the first state update (i.e., an update not to a purely local variable). block cannot include atomic actions, ghost atomic actions, _(wrap) or _(unwrap). Each update object must be \mutable or in the write set of the closed object list obs.

Ghost atomic actions do not introduce an implicit scheduler boundary, unlike non-ghost atomic actions. This is because the scheduling of ghost actions can be viewed as angelic, rather than demonic.

#### 8.7.3 Atomic Blocks

_(atomic obs)<statement> (statement)
_(begin_update)
obs is a closed object list. If the statement following the _(atomic) is not a block, or if it does not include an occurrance of _(begin_update), the annotation says that the following statement is executed atomically. Otherwise, it says that that part of the block that follows the _(begin_update) is to be executed atomically.

An atomic block can contain at most one occurrance of _(begin_update); if it does occur, it must occur as a top-level statement of the block (i.e., not inside a conditional or loop).

Before the _(begin_update), the only operations allowed are _(wrap), _(unwrap), memory operations on mutable data, _(ghost_atomic) operations.

Inside an atomic block, on any atomic access to a field of an object, VCC asserts that the object is either \mutable or is one of the read objects of the validation. If the access is a write, it asserts that it is one of the write objects of the validation.

If the body contains more than one memory access of a concrete object other than stack variables, at at least one such access along with a call to a _(atomic_inline) function, VCC will warn that the block might not appear atomic. Beyond this, it is up to the user to make sure that the compiler and hardware platform on which the program executes guarantees that the accesses to volatile concrete memory within the atomic action in a way that appears to be atomic.

In addition to _(atomic_inline) functions, the atomic block can include calls to _:pure functions. (Note, however, that such a function call after a write is likely to fail to establish the precondition that the state is full_stop().) _(wrap) and _(unwrap) cannot be used after the _(begin_update).

#### 8.7.4 Atomic Operations

_(atomic_op obs, e) (cast)
This cast cannot occur in a pure context or within an atomic action. It has the effect of turning the evaluation of the following expression exp into approximately the following:

```
_(atomic obs) {
  T \result = exp;
  e;
}
```

with \result being the result returned from the execution[28].

***Atomic Reads*** _(a_atomic_read obs) (cast)
An atomic read acts just like an atomic action, with all of its objects implicitly marked _(read_only), except that it applies to the implicit sequentialization of the compuation of the term to which it is applied.

## 9. Verifying Functions

For this section, consider a fixed function body, where all local variables of the function have been lifted to topmost scope. Define a ***predicate*** to be a boolean expression whose free variables are local variables of the function or parameters of the function. Define a ***label function*** to be a function from program labels to predicates. Given two label functions l1 and l2, define or(l1,l2)(l)== l1(l)|| l2(l).

Let S and S' be two states, let w be a pointer set, and let t be a thread. Define

```
agree(S,S',E) == (\at(S,E) == \at(S',E))

differOnlyAt(S,S',w,t) <==>
\forall \object o; !(o \in w) && \non_primitive_ptr(o) &&
      \at(S,o−>\owner == t)
  ==> (\at(S,o−>\closed) ==>
          agree(S,S',o−>\version) &&
      agree(S,S',o−>\volatile_version))
```

[28] It is unfortunate that e can only be an expression, rather than an arbitrary code block; this restriction is in place because C does not allow code blocks to appear inside expressions.

␣␣␣␣␣&&␣(\forall␣<field>␣f;␣!(&(o−>f)␣\in␣w)␣==>␣
agree(S,S',o−>f))

havocOnlyUnowned(S,S',t)␣<==>
\forall␣\object␣o;␣\non_primitive_ptr(o)␣&&␣\at(S,o−>\owner␣==␣t)
␣␣==>␣agree(S,S',o−>\version) &&
\agree(S,S',o−>\volatile_version)

Each function body is implicitly converted to one of the following restricted forms, where each statement or block below is also in one of these restricted forms, and all local variable declarations are at the outermost level. We define three functions:

- Oblig(B,P,L), where B is a restricted statement, P is a predicate, and L is a label function. This gives a proof obligation sufficient to verify that executing B from a state satisfying P (or from a jump to label l satisfying L(l)) generates only legal state transitions and only returns satisfying the postcondition of the function.
- Post(B,P,L), which gives what is known after normal exit from B starting from a state satisfying P (or from a jump to label l satisfying L(l)).
- Exit(B,P,L), which gives a label function which, when applied to a given label, gives the condition known to hold if B exits with a goto to the given label.

Succesful verification that the function meets its specification follows from Oblig(B,P,(\lambda l; \false)), where B is the body of the function (not including the local variable declarations) P is the precondition of the function, conjoined with the condition that all object invariants hold. We define these functions as follows. In each case, Exit(B,P,L) is L unless defined otherwise.

- Oblig(B;C, P, L) ==
  (Oblig(B,P,L) && Oblig(C,Post(B,P,L), or(L,Exit(B,P,L))))
  Post(B;C, P, L) ==
  Post(C,Post(B,P,L), or(L,Exit(B,P,L)))
  Exit(B;C,P,L) ==
  Exit(C,Post(B,P,L), or(L,Exit(B,P,L)))

- If B is x = e, where e is the application of a single operator (other than ++ or −−) to a suitable number of purely local variables other than x, or application of & to a local variable, then

  Oblig(x = y op z,P) == (P ==> <y op z doesn't␣overflow␣or␣
  underflow>)
  Oblig(x␣=␣*y,P)␣==
  ␣␣(P␣==>␣\non_primitive_ptr(y)␣?␣\mutable(y)␣:␣
  \mutable(\embedding(y))
  Post(B,P,L)␣==␣((\exists␣x;␣P)␣&&␣x␣==␣e)

- If B is x = f(<args>) where args is a list of purely local variables not including x, and f has the contract _(requires pre)_(writes w)_(ensures post):

  Oblig(B,P,L) ==
  <pre after parameter substitution>
  && (\forall \object o; o \in w ==> \writable(o))
  Post(B,P,L) ==
  \exists \state S; \at(s,\me) == \me &&
  differOnlyAt(s,\now(),w,\me) && post

- If B is l: (declaring a label), then

  Oblig(B,P,L) == \true
  Post(B,P,L) == P || L(l)
  Exit(B,P,L) == \lambda \label x; x == l ? \false : L(x)

- If B is goto l where l is a label that occurs syntactically later in the function body, then

  Oblig(B,P,L) == \true;
  Post(B,P,L) == \false;
  Exit(B,P,L) == \lambda \label x; x == l ? P : l(x)

- If B is if (v)C, where v is a purely local variable and C is a simplified block, then

  Oblig(B,P,L) == Oblig(C,P && v, L)
  Post(B,P,L) == v ? Post(C,P && v, L) : P
  Exit(B,P,L) == \lambda \label l; v ? Exit(C,P && v, L)(l) : L(l)

- If B is while (v)_(invariant p)_(writes s)C where v is a purely local variable and C is a simplified block, then

  q == \exists \state S; \at(s,\me) == \me && \at(s, P) && p
  && differOnlyAt(S,\now(),s,\me)
  Oblig(B,P,L) == [P ==> p]
  && Oblig(C,q,\lambda \label l; \false)
  && [Post(C,q,\lambda \label l; \false) ==> p]
  Post(B,P,L) == q && !v
  Exit(B,P,L) == or(L,Exit(C,q,\lambda \label l; \false))

- If B is _(assert p), then

  Oblig(B,P,L) == [P ==> p]
  Post(B,P,L) == P && p

- Uf B is _(assume p), then

  Oblig(B,P,L) == \true
  Post(B,P,L) == P && p

Conversion into this normalized form can involve changing the semantics of the C program, because of C's liberality in order of evaluation. Additional assertions are added to allow the program to be converted into such a form, e.g. that the order in which parameters to a function are evaluated doesn't matter.

## 10. Utility Functions

_(bump_volatile_version o)} (statement)
{
\natural \n;
_(assume n > o−>\volatile_version)
_(assert \wrapped(o) ==> \writable(o))
_(assert o−>\owner == \me)
o−>\volatile_version = n;
}

This is the only operation that is allowed to update o−>\volatile_version while o−>\closed [29].

\bool \fresh(\object o)
This predicate can appear only in a thread context or in a function or block _(ensures) annotation; it means that the object was not owned by this thread when the function was called.

\bool \nested(\object o)
True iff o−>\owner is not a thread.

\objset \span(\object o)
The set consisting of o, along with a pointer to each primative field of o.

\objset \extent(\object o)
This returns \span(o), along with \extent of any struct fields of o.

\objset \full_extent(\object)

[29] Caveat: Volatile versions should be exposed directly.

The set consisting of \extent(o), unioned with the \full_extent of any members of any union field of o.

\bool \extent_mutable(\object o)
Translates to \forall \object o1; o1 \in \extent(o)==> \mutable(o).

\bool \extent_zero(\object)
True if every all bytes in the value fields of o or in padding between fields of o are 0.

\bool \extent_fresh(\object)
True if every object in the extent of \extent(o) is \fresh.

\bool \thread_local(\object o)
True iff it is in the \domain of some object o1 that is \wrapped or \mutable.

\bool \thread_local_array(\object o, size_t s)
True iff \forall size_t i; i < s ==> \thread_local(o + s).

\bool \mutable(\object o)
True iff o−>\closed && o−>\owner == \me.

\bool \mutable_array(\object o, size_t s)
True iff \forall size_t i; i < s ==> \mutable(o + s).

\bool \in_array(\object o, \object o1, size_t s))
True iff \exists size_t i; i < s && o == o1 + s.

\objset \array_range(\object o, size_t s)
The set of all objects o+i where size_t i < s.

\objset \domain(\object o)
_(\objset \vdomain(\object o))
VCC includes the following axioms:

```
  _(axiom \forall \object o; \forall \state s; {\at(s, o \in \domain(o))}
            o−>\closed && !\nested(o) && \non_primitive_ptr(o)
            ==> \at(s, o \in \domain(o)))

  _(axiom \forall \object o,x,y; \forall \state s;
            {\at(s, x \in \domain(o)), \at(s,y \in \domain(o))}
            \at(s, x \in \domain(o)) && \at(s, y−>\owner == x)
            && <x is of a type that is not declared as
                  _(volatile_owns)>
            ==> \at(s, y \in \domain(o)))

  _(axiom \forall \object o,x,y; \forall \state s;
            {\at(s, x \in \domain(o)), \at(s,y \in \vdomain(o))}
            (\at(s,x \in \domain(o)) && \at(s, y−>\owner == x)
               && (\forall \state t; \at(s,x−>\version) ==
                     \at(t,x−>\version)
                                 ==> \at(t, y−>\owner == x)))
            ==> y \in \vdomain{o} && y \in \domain(o))

  _(axiom \forall \object o,x; \forall \state s,t;
              {\at(s,x \in \domain(o))}
              \at(s, x \in \domain(o)) && \at(t,o−>version) ==
                    \at(s,o−>version)
              ==> \at(t,x−>version) == \at(s,x−>version)
```

T ^\alloc_array<T>(\size_t s))
T ^\alloc<T>()
Here, T is a type name. \alloc<T>() allocates a fresh object of type T on the ghost heap, and returns a pointer to the new object. \alloc_array<T>(s) allocates an array of s such ghost objects.

_(bool \not_shared(\object o))
True iff o−>\claim_cnt == 0 or if o is of an unclaimable type.

\bool \malloc_root(\object o))
True iff o is the root object of a memory allocation.
\bool \object_root(\object o))
True iff o was directly created from a blob. In particular, if o is a stack variable, \object_root(&o).
\bool \union_active(\object o)
True if o is the currently active member of a valid union, i.e., if o is of union type, \valid(o)==> (\valid(&o−>f)<==> \union_active(o−>f))

\bool \addr_eq(\object o1,\object o2))
True iff \addr(o1)== \addr(o2).

\bool \arrays_disjoint(\object o1, size_t s1, \object o2, \size_t s2)
o1 and o2 must be arrays with the same basetype. True iff (\forall size_t i1,i2; i1 < s1 && i2 < s2 ==> o1[i1] != o2[i2]).

\bool \wrapped_with_deep_domain(\object o)
This is semantically equivalent to \wrapped(o). However, it causes VCC to assert that, for any object o1 in the sequential domain of o, o1 \in \domain(o).

\object \domain_root(\object o)
If there is an \object o1 such that o is transitively owned by o1, o1 is not a thread, and o1−>\owner is a thread, then this function returns o1. Otherwise, it returns an arbitrary object.

\integer \index_within(\object o1,\object o2)
Semantically equivalent to \addr(o1)− \addr(o2)/ sizeof(\type(o2)), but uses a more efficient representation internally.

\bool \writable(\object o)
This predicate is allowed only in function bodies, not in function contracts. Let w be the set of objects in the writes clause of the function (instantiated according to the call of the function). Then

```
\writable(o) <==>
  (\non_primitive_ptr(o) && (o \in w || (o−>\owner == \me &&
    <o−>\valid or o−>\closed changed after the current function
          was called>)))
  || (!\non_primitive_ptr(o) && (o \in w || \writable(\embedding(o))))
```

\natural \size(e)
e must be an expression of integral or record type, all of whose fields are of such type. If e is of integral type, this returns the absolute value of e. If e is a record type, this returns 1 plus the sum of the sizes of the fields of e.

\bool \shallow_eq(T s, T t)
s and t must be of the same structured type. This returns \true iff s and t have the same values for all explicit fields of non-structured type.

\bool \deep_eq(T s, T t)
s and t must be of the same structured type. This returns \true iff \shallow_eq(s,t) and, for each explicit structured field f of type T,

\bool \wrapped0(\object o) Equivalent to \wrapped(o)&& o−>\claim_count == 0).

T \old(T e)
Within an \ensures clause or the body of a function, \old(e) gives the value of e at function entry, after replacing any ocurrances of local variables in e with their current values. Within an object invariant, this expands to the prestate of the transition over which the invariant is evaluated.
\bool \unchanged(e)

\bool \same(e)
Equivalent to (\old(e)== e). Note that \old has different meanings depending on context.

## 11. Macros

A new annotation tag can be defined with an annotation of the form

```
_(\bool \macro_<Name>(args) {
  <body>
}
```

where <Name> is the name of the new annotation tag, and <args> gives the remaining arguments that occur within the annotation. This defines the annotation _(<Name> args) to expand to <body>.

## 12. Inference

If \wrapped(e) is a top-level conjunct of a precondition of a function, then _(requires e _(assume)\in \domain(e)) is implicitly added to the function contract. If e is of \claim type, then _(requires _(assume)\valid_claim(e)) is implicitly added to the contract.

If \claims(c, \closed(o)) is a top-level conjunct of a precondition of a function, then the preconditions _(requires _(assume)\always_by_claim(c,o)) and _(requires _(assume)\inv(o) are added to the function.

_(isolate_proof) (function attribute)
This causes VCC to verify this function in a prover session that is isolated from the rest of the verification session. (That is, the function should verify if and only if it was the only function being verified.)

_(frameaxiom) (specifier)
This forms can only be used as a specifier for a pure function with a reads clause. It tells VCC to generate an axiom that says that for any two states that agree on the values of the objects in the _(reads) clause of the function, the funciton returns the same value.

_(no_frameaxiom)
This annotation can appear only on a _:pure function with a _(reads) clause. It says not to generate a frame axiom for the function.

### 12.1 Triggering

VCC uses an SMT solver as its reasoning backend. SMT solvers typically work at the level of ground terms, and handle quantifiers by means of triggers - patterns that, when matched by terms arising in deduction, cause instances of quantified formulas to be generated. In each annotation containing a quantification, VCC allows optionally the inclusion of explicit triggers. If these are not included, VCC generates triggers automatically. In most cases, it is best to allow VCC to generate triggers on its own, since it avoids the possibility of instantiation loops. However, on rare occasions, you may want to provide your own. See the section on triggers in the tutorial for more information.

_(bool _:pure \match_long(__int64 )_(ensures \result == \true);)
_(bool _:pure \match_ulong(unsigned __int64)_(ensures \result == \true);)
These functions are declared with the contracts given above. They are provided as a convenience for writing triggers.

### 12.2 Debugging

The features of this section are included only to aid in the debugging of verifications; they should never appear in a final program. Use of any of these annotation may render verification unsound.

\bool \start_here()
This predicate can appear only in the annotation _(assume \start_here()); it tells VCC to not verify any of the assertions in the current function that lexically precede this statement. At most one such annotation should appear within any function.

_(assume_correct) (specifier)
This annotation can appear only immediately preceding a function definition. It tells VCC to not verify this function, i.e. to consider all of the assertions generated inside the body of the function definition to be assumptions.

_(no_reads_check) (specifier)
This can only be used as an attribute of a pure function with a reads clause. It causes VCC to not perform the reads check for the function, i.e. to not check that the function depends only on the objects listed in the reads clause.

_(no_admissibility) (specifier)
This annotation can appear only immediately before the definition of a struct, union, or _(record) type. It tells VCC to omit checking the admissibility of the following type.

### 12.3 Smoke Tests

Smoke testing (which is not done by default) causes VCC to check that no control location of the program is provably unreachable (provable in the sense of VCC's treatment of what is known where). This amounts to checking that an \assert(\false) added to any location would fail (with some reasonably small timeout). Because deduction is incomplete, smoke testing is not sound (though it does not interfere with the soundness of the proofs of other properties being verified).

An explicit _(assert \false) at the beginning of a block causes smoke testing to be omitted for that block.

_(known \bool val) (cast)
val must be the literal \true or the literal \false. It asserts that the cast expression has value val. If the expression is the test of a conditional or loop, or the first argument to || or && or ?:, it disables smoke testing for the branch that is known not to be taken.

_(skip_smoke) (specifier)
This annotation causes VCC to skip smoke testing on the following function.

### 12.4 Verification Switches

VCC has a number of parameters that effect how verification is performed. These can be adjusted using arguments to the statement used to run VCC (either on the command line or through the Visual Studio pluggin).

Some of these can be overridden for individual functions by means of attributes put on individual function definitions. These attributes are currently expressed with the special top-level form vcc_attr(<property>, <value>) just before a function definition. Here, <property> is a string giving the name of the property and <value> is a string giving the value for that property.

## 13. Caveats

VCC is intended to be *sound*, which is to say that if VCC verifies your program without giving any errors or warnings, the program is correct (the meaning of correctness is defined in section ??). It is not intended to be complete, which is to say that there are programs that are correct that you will not be able to verify with VCC, perhaps

even some programs that might be verified with other tools. There are also some programs that require minor syntactic changes to be verified; while we can (and have) extend the annotation syntax to handle such cases, such activity eventually meets with diminishing returns.

When reasoning about concurrency, VCC assumes that programs are running under the standard model of shared variable concurrency, namely sequentially consistent memory. Most architectures, including x86 and ARM, provide somewhat weaker memory models. VCC as described here is not sound for these models in general. However, it is sound if further conditions on the program are met. One simple restriction that suffices (for x86 and other TSO architectures) is that all writes are interlocked (i.e., all volatile atomic updates flush the store buffers). VCC will hopefully support checking weaker conditions in the future.