Verifiable C

Applying the Verified Software Toolchain to C programs

Version 2.11
January 31, 2023

Andrew W. Appel

with Lennart Beringer, Qinxiang Cao, Josiah Dodds

Contents

1 Overview

Verifiable C is a language and program logic for reasoning about the functional correctness of C programs. The *language* is a subset of CompCert C light; it is a dialect of C in which side-effects and loads have been factored out of expressions. The *program logic* is a higher-order separation logic, a kind of Hoare logic with better support for reasoning about pointer data structures, function pointers, and data abstraction.

Verifiable C is *foundationally sound.* That is, it is proved (with a machine-checked proof in the Coq proof assistant) that,

> Whatever observable property about a C program you prove using the Verifiable C program logic, that property will actually hold on the assembly-language program that comes out of the C compiler.

This soundness proof comes in two parts: The program logic is proved sound with respect to the semantics of CompCert C, by a team of researchers primarily at Princeton University; and the C compiler is proved correct with respect to those same semantics, by a team of researchers primarily at INRIA. This chain of proofs from top to bottom, connected in Coq at specification interfaces, is part of the *Verified Software Toolchain*.

Verified
Software
Toolchain

To use Verifiable C, one must have had some experience using Coq, and some familiarity with the basic principles of Hoare logic. These can be obtained by studying Pierce's *Software Foundations* interactive textbook, and doing the exercises all the way to chapter "Hoare2."

It is also useful to read the brief introductions to Hoare Logic and Separation Logic, covered in Appel's *Program Logics for Certified Compilers*, Chapters 2 and 3 (those chapters available free, follow the link).

PROGRAM LOGICS FOR CERTIFIED COMPILERS (Cambridge University Press, 2014) describes *Verifiable C* version 1.1. If you are interested in the semantic model, soundness proof, or memory model of VST, the book is well worth reading. But it is not a reference manual.

More recent VST versions differ in several ways from what the PLCC book describes. • In the LOCAL component of an assertion, one writes temp i v instead of `(eq v) (eval_id i). • In the SEP component of an assertion, backticks are not used (predicates are not lifted). • In general, the backtick notation is rarely needed. • The type-checker now has a more refined view of char and short types. • field_mapsto is now called field_at, and it is dependently typed. • typed_mapsto is renamed data_at, and last two arguments are swapped. • umapsto ("untyped mapsto") no longer exists. • mapsto sh t v w now permits either (w =Vundef) or the value w belongs to type t. This permits describing uninitialized locations, i.e., mapsto_sh t v = mapsto_ sh t v Vundef. For function calls, one uses forward_call instead of forward. • C functions may fall through the end of the function body, and this is (per the C semantics) equivalent to a return statement.

2 Installation

The Verified Software Toolchain runs on Linux, Mac, or Windows. You will need to install:

Coq 8.15, from coq.inria.fr. Follow the standard installation instructions.

CompCert 3.11, from http://compcert.inria.fr/download.html. Build the *clightgen* tool, using these commands: ./configure -clightgen x86_32-linux; make. You might replace x86_32-linux with x86_32-macosx or x86_32-cygwin. Verifiable C should work on other 32-bit architectures as well, but has not been extensively tested. Verifiable C also works (and is regularly tested) on 64-bit architectures.

VST 2.11, from vst.cs.princeton.edu, or else an appropriate version from https://github.com/PrincetonUniversity/VST. After unpacking, read the BUILD_ORGANIZATION file (or simply make -j).

Note on the Windows (cygwin) installation of CompCert: To build CompCert you'll need an up to date version of the menhir parser generator. To work around a cygwin incompatibility in the menhir build, touch src/.versioncheck before doing make.

3 *Workflow, loadpaths*

Within VST, the progs directory contains some sample C programs with their verifications. The workflow is:

- Write a C program F.c.
- Run clightgen -normalize F.c to translate it into a Coq file F.v.
- Write a verification of F.v in a file such as verif_F.v. That latter file must import both F.v and the VST *Floyd*[1] program verification system, VST.floyd.proofauto.

LOAD PATHS. Interactive development environments (CoqIDE or Proof General) will need their load paths properly initialized. Running make in vst creates a file _CoqProject file with the right load paths *for proof development of the VST itself or of its progs/ examples.* From the VST current directory, you can say (for example),
coqide progs/verif_reverse.v &

IN NORMAL USE (if you are not simply browsing the progs examples) your own files (F.c, F.v, verif_F.v) will not be inside the VST directory. You will need to run coqc or coqide (or Proof General) with "coq flags" to access the VST components. For this, use the file _CoqProject-export, created by make in VST.

Example:

```
cd my-own-directory
cp my/path/to/VST/_CoqProject-export _CoqProject
coqide myfile.v &
```

FOR MORE INFORMATION, See the heading USING PROOF GENERAL AND COQIDE in the file BUILD_ORGANIZATION.

[1] Named after Robert W. Floyd (1936–2001), a pioneer in program verification.

4 Verifiable C

Verifiable C is a *program logic* (higher-order impredicative concurrent separation logic) for C programs with these restrictions:

- No casting between integers and pointers.
- No goto statements.
- No bitfields in structs.
- No struct-copying assignments, struct parameters, or struct returns.
- Only structured switch statements (no Duff's device).
- No varargs functions, except limited undocumented support for calling printf and fprintf.

The Verifiable C program logic operates on the CompCert *Clight* language. CompCert's clightgen tool (described in the next chapter) translates C into C light, so that you can use VST to apply the program logic to the program. Clight (and clightgen) does support some of the features listed above (such as goto, bitfields, struct-copying), but programs with those features cannot be proved in Verifiable C.

5 clightgen

CompCert's clightgen tool translates C into abstract syntax trees (ASTs) of CompCert's *Clight* intermediate language. You find clightgen in the root directory of your CompCert installation, after doing make clightgen.

To translate a C source program foo.c into its AST file foo.v, run:

```
clightgen -normalize foo.c
```

Clightgen invokes the standard macro-preprocessor (to handle define and include), parses, type-checks, and produces ASTs.

Although your C programs may have side effects inside subexpressions, and memory dereferences inside subexpressions or if-tests, the *program logic* does not permit this. Therefore, clightgen transforms your programs before you apply the program logic:

- Factors out function calls and assignments from inside subexpressions (by moving them into their own assignment statements).
- Factors && and || operators into **if** statements (to capture short circuiting behavior).
- When the -normalize flag is used, factors each memory dereference into a top level expression, i.e. x=a[b[i]]; becomes t=b[i]; x=a[t];.

SHORT-IDENTS. If you give the -short-idents flag to clightgen, then it will represent identifiers in ASTs in a way that leads to faster processing by VST. But then, if your C program is in several modules (such as x.c y.c z.c) you must process them all together in clightgen,

```
clightgen -normalize -short-idents x.c y.c z.c
```

This produces the files x.v y.v z.v containing Coq representations of ASTs, in a way that the representation of identifiers is consistent across the files. When using clightgen's default canonical-idents mode, this all-at-once processing is not necessary.

6 ASTs: abstract syntax trees

We will introduce Verifiable C by explaining the proof of a simple C program: adding up the elements of an array.

```
unsigned sumarray(unsigned a[], int n) {
  int i; unsigned s;
  i=0;
  s=0;
  while (i<n) {
    s+=a[i];
    i++;
  }
  return s;
}
```

```
unsigned four[4] = {1,2,3,4};

int main(void) {
  unsigned s;
  s = sumarray(four,4);
  return (int)s;
}
```

You can examine this program in VST/progs/sumarray.c. Then look at progs/sumarray.v to find the output of CompCert's *clightgen* utility: it is the abstract syntax tree (AST) of the C program, expressed in Coq. In sumarray.v there are definitions such as,

```
Definition _main : ident := 54%positive.
Definition _s : ident := 50%positive.
...
Definition f_sumarray := {|
  fn_return := tint; ...
  fn_params := ((_a, (tptr tint)) :: (_n, tint) :: nil);
  fn_temps := ((_i, tint) :: (_s, tint) :: (_x, tint) :: nil);
  fn_body :=
(Ssequence
  (Sset _i (Econst_int (Int.repr 0) tint))
  (Ssequence (Sset _s (Econst_int (Int.repr 0) tint)) (Ssequence ... ))) |}.

Definition prog : Clight.program := {| ... f_sumarray ... |}.
```

In general it's never necessary to read the AST file such as sumarray.v. But it's useful to know what kind of thing is in there. C-language identifiers such as main and s are represented in ASTs as positive numbers (for efficiency); the definitions _main and _s are abbreviations for these. The AST for sumarray is in the function-definition f_sumarray.

In the source program sumarray.c, the function sumarray's return type is is int. In the abstract syntax (sumarray.v), the fn_return component of the function definition is tint, or equivalently (by **Definition**) Tint I32 Signed noattr. The Tint constructor is part of the abstract syntax of C type-expressions, defined by CompCert as,

Inductive type : Type :=
  | Tvoid: type
  | Tint: intsize → signedness → attr → type
  | Tpointer: type → attr → type
  | Tstruct: ident → attr → type
  | ... .

See also Chapter 28 (C types).

7 Use the IDE

Chapter 8 through Chapter 22 are meant to be read while you have the file progs/verif_sumarray.v open in a window of your interactive development environment for Coq. You can use Proof General, CoqIDE, or any other IDE that supports Coq.

Reading these chapters will be much less informative if you cannot see the proof state as each chapter discusses it.

Before starting the IDE, review Chapter 3 (Workflow) to see how to set up load paths.

8 Functional model, API spec

A program without a specification cannot be incorrect, *it can only be* surprising. *(Paraphrase of J. J. Horning, 1982)*

The file progs/verif_sumarray.v contains the specification of sumarray.c and the proof of correctness of the C program with respect to that specification. For larger programs, one would typically break this down into three or more files:

1. Functional model (often in the form of a Coq function)
2. API specification
3. Function-body correctness proofs, one per file.

We start verif_sumarray.v with some standard boilerplate:

```
Require Import VST.floyd.proofauto.
Require Import VST.progs.sumarray.
Instance CompSpecs : compspecs. make_compspecs prog. Defined.
Definition Vprog : varspecs. mk_varspecs prog. Defined.
```

The first line imports Verifiable C and its *Floyd* proof-automation library. The second line imports the AST of the program to be proved. Lines 3 and 4 are identical in any verification: see Chapter 29 and Chapter 52.

To prove correctness of sumarray.c, we start by writing a *functional spec* of adding-up-a-sequence, then an *API spec* of adding-up-an-array-in-C.

FUNCTIONAL MODEL. A *mathematical model* of this program is the sum of a sequence of integers: $\sum_{i=0}^{n-1} x_i$. It's conventional in Coq to use list to represent a sequence; we can represent the sum with a list-fold:

```
Definition sum_Z : list Z → Z := fold_right Z.add 0.
```

A functional model contains not only definitions; it's also useful to include theorems about this mathematical domain:

```
Lemma sum_Z_app: ∀a b, sum_Z (a++b) = sum_Z a + sum_Z b.
Proof. intros. induction a; simpl; lia. Qed.
```

The data types used in a functional model can be any kind of mathematics at all, as long as we have a way to relate them to the integers, tuples, and sequences used in a C program. But the mathematical integers Z and the 32-bit modular integers Int.int are often relevant. Notice that this functional spec does not depend on sumarray.v or even on anything in the Verifiable C libraries. This is typical, and desirable: the functional model is about mathematics, not about C programming.

THE APPLICATION PROGRAMMER INTERFACE (API) of a C program is expressed in its header file: function prototypes and data-structure definitions that explain how to call upon the modules' functionality. In *Verifiable C*, an *API specification* is written as a series of *function specifications* (funspecs) corresponding to the function prototypes.

```
Definition sumarray_spec : ident * funspec :=
DECLARE _sumarray
 WITH a: val, sh : share, contents : list Z, size: Z
 PRE [ (tptr tuint), tint ]
    PROP(readable_share sh;
          0 ≤ size ≤ Int.max_signed;
          Forall (fun x ⇒ 0 ≤ x ≤ Int.max_unsigned) contents)
    PARAMS(a; Vint (Int.repr size))
    SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a)
  POST [ tuint ]
    PROP()
    RETURN(Vint (Int.repr (sum_Z contents)))
    SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a).
```

The funspec begins, **Definition** f_spec := DECLARE _f ... where f is the name of the C function, and _f : ident is Coq's name for the identifier that denotes f in the AST of the C program (see page 11).

A function is specified by its *precondition* and its *postcondition*. The WITH clause quantifies over Coq values that may appear in both the precondition and the postcondition. The precondition is parameterized by the C-language function parameters, and the postcondition is parameterized by a identifier ret_temp, which is short for, “the temporary variable holding the return value.”

Function preconditions, postconditions, and loop invariants are *assertions* about the state of variables and memory at a particular program point. In an assertion $\mathsf{PROP}(\vec{P})\ \mathsf{LOCAL}(\vec{Q})\ \mathsf{SEP}(\vec{R})$, the propositions in the sequence $\vec{P}$ are all of Coq type Prop, describing facts that are independent of program state. In the precondition above, the $0 \leq \mathsf{size} \leq \mathsf{Int.max\_signed}$ is true *just within the scope of the quantification of the variable size*; it is bound by WITH, and spans the PRE and POST assertions.

If you see a precondition (PRE) with LOCAL instead of PARAMS, it is an old-style funspec; see Chapter 77.

The *local* part of a PROP/LOCAL/SEP assertion takes different forms depending on what kind of local variables it describes: in function preconditions it is written PARAMS() (or sometimes PARAMS()GLOBALS()); in function postconditions it is written RETURN(); and inside a function body it is LOCAL().

Function preconditions are based on nameless, *positional* parameter notation. That is, PRE[$\vec{\tau}$] gives the C-language types (but not the names) of the formal parameters, and PARAMS($\vec{v}$) gives the abstract values (but not the names) of those parameters. As you can see, the abstract values are usually based on variables bound in the WITH clause.

Values of PARAMS and RETURN are C scalar values whose Coq type is val; this type is defined by CompCert as,

```
Inductive val: Type := Vundef: val | Vint: int →val | Vlong: int64 →val
    | Vfloat: float →val | Vsingle: float32 →val | Vptr: block →ptrofs →val.
```

The SEP conjuncts $\vec{R}$ are *spatial assertions* in separation logic. In this case, there's just one, a data_at assertion saying that at address a in memory, there is a data structure of type *array[size] of unsigned integers*, with access-permission sh, and the contents of that array is the sequence map Vint (map Int.repr contents).

THE POSTCONDITION is introduced by POST [tuint], indicating that this function returns a value of type unsigned int. There are no PROP statements in this postcondition—no forever-true facts hold now, that weren't already true on entry to the function. The RETURN clause says what the return value is (or RETURN() for a void function). The SEP clause mentions all the spatial resources from the precondition, minus ones that have been freed (deallocated), plus ones that have been malloc'd (allocated).

So, overall, the specification for sumarray is this: "At any call to sumarray, there exist values a,sh,$contents$,$size$ such that sh gives at least read-permission; $size$ is representable as a nonnegative 32-bit signed integer; the first function-parameter contains value a and the second contains the 32-bit representation of $size$; and there's an array in memory at address a with permission sh containing $contents$. The function returns a value equal to sum_int($contents$), and leaves the array unaltered."

INTEGER OVERFLOW. In Verifiable C's signed integer arithmetic, you must prove (if the system cannot prove automatically) that no overflow occurs. In unsigned integers, arithmetic is treated as modulo-2^n (where n is typically 32 or 64), and overflow is not an issue. See Chapter 25. The function Int.repr: Z $\rightarrow$ int truncates mathematical integers into 32-bit integers by taking the (sign-extended) low-order 32 bits. Int.signed: int $\rightarrow$ Z injects back into the signed integers.

This program uses unsigned arithmetic for the s and the array contents, and uses signed arithmetic for i.

The postcondition guarantees that the value returned is

Int.repr (sum_Z contents). But what if $\sum s \geq 2^{32}$, so the sum doesn't fit in a 32-bit signed integer? Then Int.unsigned(Int.repr (sum_Z contents)) $\neq$ (sum_Z contents). In general, for a claim about Int.repr(x) to be *useful*, one also needs a claim that $0 \leq x \leq$ Int.max_unsigned or Int.min_signed $\leq x \leq$ Int.max_signed. The caller of this function will probably need to prove $0 \leq$ sum_Z contents $\leq$ Int.max_unsigned in order to make much use of the postcondition.

9 Proof of the sumarray program

To prove correctness of a whole program,

1. Collect the function-API specs together into Gprog: list funspec.
2. Prove that each function satisfies its own API spec (with a semax_body proof).
3. Tie everything together with a semax_func proof.

In progs/verif_sumarray.v, the first step is easy:

Definition Gprog := ltac:(with_library prog [sumarray_spec; main_spec]).

The function specs, built using DECLARE, are listed in the argument to with_library. Chapter 73 describes with_library.

In addition to Gprog, the API spec contains Vprog, the list of global-variable type-specs. This is computed automatically by the mk_varspecs tactic, as shown at the beginning of verif_sumarray.v.

Each C function can call any of the other C functions in the API, so each semax_body proof is a client of the entire API spec, that is, Vprog and Gprog. You can see that in the statement of the semax_body lemma for the _sumarray function:

Lemma body_sumarray: semax_body Vprog Gprog f_sumarray sumarray_spec.

Here, f_sumarray is the actual function body (AST of the C code) as parsed by clightgen; you can read it in sumarray.v. You can read body_sumarray as saying, *In the context of Vprog and Gprog, the function body f_sumarray satisfies its specification sumarray_spec.* We need the context in case the sumarray function refers to a global variable (Vprog provides the variable's type) or calls a global function (Gprog provides the function's API spec).

10 start_function

The predicate semax_body states the Hoare triple of the function body, $\Delta \vdash \{Pre\}\, c\, \{Post\}$. *Pre* and *Post* are taken from the funspec for f, c is the body of F, and the type-context Δ is calculated from the global type-context overlaid with the parameter- and local-types of the function.

To prove this, we begin with the tactic start_function, which takes care of some simple bookkeeping and expresses the Hoare triple to be proved.

```
Lemma body_sumarray: semax_body Vprog Gprog f_sumarray sumarray_spec.
Proof.
start_function.
```

The proof goal now looks like this:

```
Espec : OracleKind
a : val
sh : share
contents : list Z
size : Z
Delta_specs := abbreviate : PTree.t funspec
Delta := abbreviate : tycontext
SH : readable_share sh
H : 0 ≤ size ≤ Int.max_signed
H0 : Forall (fun x : Z ⇒ 0 ≤ x ≤ Int.max_unsigned) contents
POSTCONDITION := abbreviate : ret_assert
MORE_COMMANDS := abbreviate : statement
______________________________________(1/1)
semax Delta
  (PROP ()
   LOCAL(temp _a a; temp _n (Vint (Int.repr size)))
   SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a))
  (Ssequence (Sset _i (Econst_int (Int.repr 0) tint)) MORE_COMMANDS)
  POSTCONDITION
```

First we have *Espec*, which you can ignore for now (it characterizes the outside world, but sumarray.c does not do any I/O). Then a,sh,contents,size are exactly the variables of the WITH clause of sumarray_spec.

The two abbreviations Delta_spec, Delta are the type-context in which Floyd's proof tactics will look up information about the types of the program's variables and functions. The hypotheses SH,H,H0 are exactly the PROP clause of sumarray_spec's precondition. The POSTCONDITION is exactly the POST part of sumarray_spec.

To see the contents of an abbreviation, either (1) set your IDE to show implicit arguments, or (2) unfold abbreviate in POSTCONDITION.

Below the line we have one proof goal: the Hoare triple of the function body. In general, any C statement c might satisfy a Hoare-logic judgment $\Delta \vdash \{P\}\, c\, \{R\}$ when, in global context Δ, started in a state satisfying precondition P, statement c is sure not to crash and, if it terminates, the final state will satisfy R. We write the Hoare judgement in Coq as semax (Δ: tycontext) (P: environ→mpred) (c: statement) (R: ret_assert).

- Δ is a *type context*, giving types of function parameters, local variables, and global variables; and *specifications* (funspec) of global functions.
- P is the precondition;
- c is a command in the C language; and
- R is the postcondition. Because a c statement can exit in different ways (fall-through, continue, break, return), a ret_assert has predicates for all of these cases.

Right after start_function, the command c is the entire function body.

Because we do *forward* Hoare-logic proof, we won't care about the postcondition until we get to the end of c, so here we hide it away in an abbreviation. Here, the command c is a long sequence starting with i=0;...*more*, and we hide the *more* in an abbreviation MORE_COMMANDS.

The precondition of this semax has LOCAL and SEP parts taken directly from the funspec (the PROP clauses have been moved above the line). The statement (Sset _i (Econst_int (Int.repr 0) tint)) is the AST generated by clightgen from the C statement i=0;.

11 forward

We do Hoare logic proof by forward symbolic execution. On page 20 we show the proof goal at the beginning of the sumarray function body. In a forward Hoare logic proof of $\{P\}\, i = 0; \mathit{more}\, \{R\}$ we might first apply the sequence rule,

$$\frac{\{P\}\, i = 0; \{Q\} \quad \{Q\}\, \mathit{more}\, \{R\}}{\{P\}\, i = 0; \mathit{more}\, \{R\}}$$

assuming we could derive some appropriate assertion Q. For many kinds of statements (assignments, return, break, continue) this is done automatically by the forward tactic, which applies a strongest-postcondition style of proof rule to derive Q. When we execute forward here, the resulting proof goal is,

```
Espec, a, sh, contents, size, Delta_spec, SH, H, H0 as before
Delta := abbreviate : tycontext
POSTCONDITION := abbreviate : ret_assert
MORE_COMMANDS := abbreviate : statement
-----------------------------------------------------------------(1/1)
semax Delta
  (PROP ()
   LOCAL(temp _i (Vint (Int.repr 0)); temp _a a;
   temp _n (Vint (Int.repr size)))
   SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a))
  (Ssequence (Sset _s (Econst_int (Int.repr 0) tuint)) MORE_COMMANDS)
  POSTCONDITION
```

Notice that the precondition of this semax is really the *postcondition* Q of the i=0; statement; it is the precondition of the *next* statement, s=0;. It's much like the precondition of i=0; what has changed?

- The LOCAL part contains temp _i (Vint (Int.repr 0)) in addition to what it had before; this says that the local variable i contains integer value zero.

- the command is now s=0;*more*, where MORE_COMMANDS no longer contains s=0;.
- Delta has changed; it now records the information that i is initialized.

Applying the forward again will go through s=0; to yield a proof goal with a LOCAL binding for the _s variable.

Forward works on several kinds of C commands. In each of the following cases, x must be a nonaddressable local variable, a temp.

c_1; c_2 Sequence of commands. The forward tactic will work on c_1 first.

(c_1; c_2); c_3 In this case, forward will re-associate the commands using the seq_assoc axiom, and work on c_1; (c_2; c_3).

x=E; Assignment statement. Expression E must not contain memory dereferences (loads or stores using ∗prefix, suffix[], or -> operators). No restrictions on the form of the precondition (except that it must be in canonical form, PROP/LOCAL/SEP). The expression &p→next is permitted, since it does not actually load or store (it just computes an address).

x= *E; Memory load.

x= a[E]; Array load.

x= E→*fld*; Field load.

x= E→f_1.f_2; Nested field load; see Chapter 33.

x= E→f_1[i].f_2; Fields and subscripts; see Chapter 33.

E_1 = E_2; Memory store. Expression E_2 must not dereference memory. Expression E_1 must be equivalent to a single memory store via some access *path* (see Chapter 33), and the precondition must contain an appropriate storable data_at or field_at.

if (E) C_1 **else** C_2 For an if-statement, use forward_**if** and (perhaps) provide a postcondition.

while (E) C For a while-loop, use the forward_while tactic (page 28) and provide a loop invariant.

break; The forward tactic works.

continue; The forward tactic works.

return E**;** Expression E must not dereference memory, and the presence/absence of E must match the nonvoid/void return type of the function. The proof goal left by forward is to show that the precondition (with appropriate substitution for the abstract variable ret_var) entails the function's postcondition.

$x = f(a_1,\ldots,a_n)$; For a function call, use forward_call (see Chapter 21).

12 *Hint*

In any VST proof state, running the `hint` tactic will print a suggestion (if it can) that will help you make progress in the proof. In stepping through the case studies described in this reference manual, insert `hint.` at any point to see what it says.

13 If, While, For

To do forward proof through if-statements, while-loops, and for-loops, you need to provide additional information: join-postconditions, loop invariants, etc. The tactics are forward_**if**, forward_while, forward_for, forward_for_simple_bound.

If you're not sure which tactic to use, and with how many arguments, just use forward, and the error message will make a suggestion.

- **if** e **then** s_1 **else** s_2; $s_3 \ldots$
  Use forward_if Q, where Q is the *join postcondition*, the precondition of statement s_3. Q may be a full assertion (environ→mpred), or it may be just a Prop, in which case it will be *added* to the current precondition.
- **if** e **then** s_1 **else** s_2;}...
  When the if-statement appears at the end of a block, so the postcondition is already known, you can do forward_**if**. That is, you don't need to supply a join postcondition if POSTCONDITION is fully instantiated, without any unification variables. You can unfold abbreviate **in** POSTCONDITION to see.

  When one (or both) of your then/else branches exits by break, continue, or return then you don't need to supply a join postcondition.
- **while** $(e)s; \ldots$ (no break statements in s)
  You write forward_while Q, where Q is a loop invariant. See Chapter 14.
- **while** $(e)s; \ldots$ (with break statements in s)
  You must treat this as if it were **for**$(;e;)$ s; see below.
- **for**$(e_1;e_2;e_3)$ s
  Use a tactic for for-loops:
  forward_for_simple_bound (Chapter 53),
  forward_for (Chapter 54), or
  forward_loop (Chapter 55).

14 While loops

To prove a *while* loop by forward symbolic execution, you use the tactic forward_while, and you must supply a loop invariant. Take the example of the forward_while in progs/verif_sumarray.v. The proof goal is,

```
Espec, Delta_specs, Delta
a : val, sh : share, contents : list Z, size : Z
SH : readable_share sh
H : 0 ≤size ≤Int.max_signed
H0 : Forall (fun x : Z ⇒ 0 ≤x ≤Int.max_unsigned) contents
POSTCONDITION := abbreviate : ret_assert
MORE_COMMANDS, LOOP_BODY := abbreviate : statement
------------------------------------------------------------(1/1)
semax Delta
  (PROP ()
   LOCAL(temp _s (Vint (Int.repr 0)); temp _i (Vint (Int.repr 0));
           temp _a a; temp _n (Vint (Int.repr size)))
   SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a))
  (Ssequence
     (Swhile (Ebinop Olt (Etempvar _i tint) (Etempvar _n tint) tint)
        LOOP_BODY)
   MORE_COMMANDS)
  POSTCONDITION
```

A loop invariant is an assertion, almost always in the form of an existential EX...PROP(...)LOCAL(...)SEP(...). Each iteration of the loop has a state characterized by a different value of some iteration variable(s), the EX binds that value. The invariant for the sumarray loop is,

```
EX i: Z,
 PROP(0 ≤i ≤size)
 LOCAL(temp _a a; temp _i (Vint (Int.repr i)); temp _n (Vint (Int.repr size));
          temp _s (Vint (Int.repr (sum_Z (sublist 0 i contents)))))
 SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a).
```

The existential binds i, the iteration-dependent value of the local variable named _i. In general, there may be any number of EX quantifiers.

The forward_while tactic will generate four subgoals to be proved:

1. the precondition (of the whole loop) implies the loop invariant;
2. the loop-condition expression type-checks (i.e., guarantees to evaluate successfully);
3. the postcondition of the loop body implies the loop invariant;
4. the loop invariant (and negation of the loop condition) is a strong enough precondition to prove the MORE_COMMANDS after the loop.

Let's take a look at that first subgoal:

```
    (above-the-line hypotheses elided)
______________________________________________________1/4
ENTAIL Delta,
      PROP()
      LOCAL(temp _s (Vint (Int.repr 0)); temp _i (Vint (Int.repr 0));
            temp _a a; temp _n (Vint (Int.repr size)))
      SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a)
⊢EX i : Z,
      PROP(0 ≤ i ≤ size)
      LOCAL(temp _a a; temp _i (Vint (Int.repr i));
            temp _n (Vint (Int.repr size));
            temp _s (Vint (Int.repr (sum_Z (sublist 0 i contents)))))
      SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a)
```

This is an *entailment* goal; Chapter 16 shows how to prove such goals.

15 PROP( ) LOCAL( ) SEP( )

Each element of a SEP clause is a *spatial predicate*, that is, a predicate on some part of the memory. The Coq type for a spatial predicate is mpred; it can be thought of as mem→Prop (but is not quite the same, for quite technical semantic reasons).

The SEP represents the *separating conjunction* of its spatial predicates. When we write spatial predicates outside of a PROP/LOCAL/SEP, we use $*$ instead of semicolon to indicate separating conjunction.

The LOCAL part of an assertion describes the values of local variables.

A program assertion (precondition, postcondition, loop invariant, etc.) is a predicate *both* on its local-var *environ* and its memory. Its Coq type is environ→mpred. If you do the Coq command, Check (PROP()LOCAL()SEP()) then Coq replies, environ→mpred. We call assertions of this type *lifted predicates*.

The *canonical form* of a lifted assertion is PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP($\vec{R}$), where $\vec{P}$ is a list of propositions (Prop), where $\vec{Q}$ is a list of local-variable definitions (localdef), and $\vec{R}$ is a list of base-level assertions (mpred). Each list is semicolon-separated.

The existential quantifier EX can also be used on canonical forms, e.g., EX x:T, PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP($\vec{R}$).

16 Entailments

An *entailment* in separation logic, $P \vdash Q$, says that any state satisfying P must also satisfy Q. In Verifiable C, if P and Q are mpreds, then any mem satisfying P must also satisfy Q. If P and Q are *lifted predicates*, then any environ×mem satisfying P must also satisfy Q.

Usually we write lifted entailments as ENTAIL $\Delta, P \vdash Q$ in which Δ is the global type context, providing additional constraints on the state.

Verifiable C's *rule of consequence* is,

$$\frac{\mathsf{ENTAIL}\ \Delta, P \vdash P' \quad \mathsf{semax}\ \Delta\ P'\ c\ Q' \quad \mathsf{ENTAIL}\ \Delta, Q' \vdash Q}{\mathsf{semax}\ \Delta\ P\ c\ Q}$$

Using this axiom (called semax_pre_post) on a proof goal semax $\Delta\ P\ c\ Q$ yields three subgoals: another semax and two (lifted) entailments, ENTAIL $\Delta, P \vdash P'$ and ENTAIL $\Delta, Q \vdash Q'$. P and Q are typically in the form PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP($\vec{R}$), perhaps with some EX quantifiers in the front. The turnstile $\vdash$ is written in Coq as |--.

Let's consider the entailment arising from forward_while in the progs/verif_sumarray.v example:

```
H : 0 ≤ size ≤ Int.max_signed
   (other above-the-line hypotheses elided)
______________________________________________1/4
ENTAIL Delta,
      PROP()
      LOCAL(temp _s (Vint (Int.repr 0)); temp _i (Vint (Int.repr 0));
              temp _a a; temp _n (Vint (Int.repr size)))
      SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a)
⊢ EX i : Z,
      PROP(0 ≤ i ≤ size)
      LOCAL(temp _a a; temp _i (Vint (Int.repr i));
              temp _n (Vint (Int.repr size));
              temp _s (Vint (Int.repr (sum_Z (sublist 0 i contents)))))
      SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a)
```

We instantiate the existential with the only value that works here, zero: **Exists** 0. Chapter 24 explains how to handle existentials with **Intros** and **Exists**.

Now we use the entailer! tactic to solve as much of this goal as possible (see Chapter 42). In this case, the goal solves entirely automatically. In particular, $0 \le i \le$ size solves by lia; sublist 0 0 contents rewrites to nil; and sum_Z nil simplifies to 0.

THE SECOND SUBGOAL of forward_while in progs/verif_sumarray.v is a *type-checking entailment*, of the form ENTAIL Δ, PQR $\vdash$ tc_expr Δ e where e is (the abstract syntax of) a C expression; in the particular case of a *while* loop, e is the negation of the loop-test expression. The assertion tc_expr Δ e says that executing e won't crash: all the variables it references exist and are initialized; and it doesn't divide by zero, et cetera.

In this case, the entailment concerns the expression $\neg(i < n)$,

```
ENTAIL Delta, PROP(...) LOCAL(...) SEP(...)
⊢tc_expr Delta
    (Eunop Onotbool (Ebinop Olt (Etempvar _i tint) (Etempvar _n tint) tint)
       tint)
```

This solves completely via the entailer! tactic. To see why that is, instead of doing entailer!, do unfold tc_expr; simpl. You'll see that the right-hand side of the entailment simplifies down to !!True, (equivalent to TT, the "true" mpred). That's because the typechecker is *calculational*, as Chapter 25 of *Program Logics for Certified Compilers* explains.

17 Array subscripts

THE THIRD SUBGOAL of forward_while in progs/verif_sumarray.v is the *body* of the while loop: {x=a[i]; s+=x; i++;}.

This can be handled by three forward commands, but the first one needs a bit of extra help. To see why, try doing forward just *before* the assert_PROP instead of after. You'll see an error message saying that it can't prove 0 ≤ i < Zlength contents. Indeed, the command x=a[i]; is safe only if i is in-bounds of the array a.

Let's examine the proof goal:

```
SH : readable_share sh
H : 0 ≤size ≤Int.max_signed
H0 : Forall (fun x : Z ⇒ 0 ≤x ≤Int.max_unsigned) contents
i : Z
HRE : i < size
H1 : 0 ≤i ≤size
______________________________________(1/1)
semax Delta
  (PROP ()
   LOCAL(temp _a a; temp _i (Vint (Int.repr i));
   temp _n (Vint (Int.repr size));
   temp _s (Vint (Int.repr (sum_Z (sublist 0 i contents)))))
   SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a))
  (Ssequence
     (Sset _x
        (Ederef
           (Ebinop Oadd (Etempvar _a (tptr tuint)) (Etempvar _i tint)
              (tptr tuint)) tuint)) MORE_COMMANDS) POSTCONDITION
```

The Coq variable i was introduced automatically by forward_while from the existential variable, the EX i:Z of the loop invariant.

Going forward through x=a[i]; will be enabled by the data_at in the precondition, as long as the subscript value is less than the length of contents. One important property of data_at π (tarray τ n) σ p is that $n = \mathsf{Zlength}(\sigma)$. If we had that fact above the line, then (using assumptions HRE and H) it would be easy to prove $0 \leq \mathsf{i} < \mathsf{Zlength}$ contents.

Therefore, we write,

```
assert_PROP (Zlength contents = size). {
  entailer!. do 2 rewrite Zlength_map. reflexivity.
}
```

Chapter 44 describes assert_PROP, which (like Coq's standard assert) will put Zlength contents=size above the line. The *first* subgoal of assert_PROP requires us to prove the proposition, *using facts from the current Hoare precondition* (which would not be accessible to Coq's standard assert). The reason this one is so easily provable is that entailer! extracts the $n = \mathsf{Zlength}(\sigma)$ fact from data_at and puts it above the line.

The *second* subgoal is just like the subgoal we had before doing assert_PROP, but with the new proposition above the line. Now that H_2: Zlength contents = size is above the line, forward succeeds on the array subscript.

Two more forward commands take us to the end of the loop body.

18 At the end of the loop body

In progs/verif_sumarray.v, at the comment “Now we have reached the end of the loop body,” it is time to prove that the *current* precondition (which is the postcondition of the loop body) entails the loop invariant. This is the proof goal:

```
H : 0 ≤ size ≤ Int.max_signed
H0 : Forall (fun x : Z ⇒ 0 ≤ x ≤ Int.max_unsigned) contents
HRE : i < size
H1 : 0 ≤ i ≤ size
  (other above-the-line hypotheses elided)
--------------------------------------------------------
ENTAIL Delta,
  PROP()
  LOCAL(temp _i (Vint (Int.add (Int.repr i) (Int.repr 1)));
    temp _s
     (force_val
       (sem_add_default tint tint
        (Vint (Int.repr (sum_Z (sublist 0 i contents))))
        (Znth i (map Vint (map Int.repr contents)))));
    temp _x (Znth i (map Vint (map Int.repr contents)));
    temp _a a; temp _n (Vint (Int.repr size)))
  SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a)
⊢ EX a0 : Z,
    PROP(0 ≤ a0 ≤ size)
    LOCAL(temp _a a; temp _i (Vint (Int.repr a0));
    temp _n (Vint (Int.repr size));
    temp _s (Vint (Int.repr (sum_Z (sublist 0 a0 contents)))))
    SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a)
```

The right-hand side of this entailment is just the loop invariant. As usual at the end of a loop body, there is an existentially quantified variable that must be instantiated with an iteration-dependent value. In this case it’s obvious: the quantified variable represents the contents of C local variable _i, so we do, **Exists** (i+1).

The resulting entailment has many trivial parts and a nontrivial residue. The usual way to get to the hard part is to run entailer!, which we do now. After clearing away the irrelevant hypotheses, we have:

H : 0 ≤ Zlength contents ≤ Int.max_signed
HRE : i < Zlength contents
H1 : 0 ≤ i ≤ Zlength contents
---(1/1)
Vint (Int.repr (sum_Z (sublist 0 (i + 1) contents))) =
Vint (Int.repr (sum_Z (sublist 0 i contents) + Znth i contents))

Applying f_equal twice, leaves the goal,

sum_Z (sublist 0 (i + 1) contents) =
sum_Z (sublist 0 i contents) + Znth i contents

Now the lemma sublist_split is helpful here:

sublist_split: $\forall l\, m\, h$ al, $0 \leq l \leq m \leq h \leq$ |al| →
sublist l h al = sublist l m al ++ sublist m h al

So we do, rewrite (sublist_split 0 i (i+1)) by lia. A bit more rewriting with the theory of sum_Z and sublist finishes the proof.

See also: Chapter 63 (sublist).

19 Returning from a function

In progs/verif_sumarray.v, at the comment "After the loop," we have reached the return statement. The forward tactic works here, leaving a proof goal that the precondition of the return entails the postcondition of the function-spec. (Sometimes the entailment solves automatically, leaving no proof goal at all.) The goal is a *lowered* entailment (on mpred assertions).

```
H4 : Forall (value_fits tuint) (map Vint (map Int.repr contents))
H2 : field_compatible (Tarray tuint (Zlength ...) noattr) [] a
   (other above-the-line hypotheses elided)
------------------------------------------------------------
data_at sh (tarray tuint (Zlength ...)) (map Vint (map Int.repr contents)) a
⊢ !!(Vint (Int.repr (sum_Z contents)) =
        Vint (Int.repr (sum_Z (sublist 0 i contents))))
```

The left-hand side of this entailment is a spatial predicate (data_at). Purely nonspatial facts (H4 and H2) derivable from it have already been inferred and moved above the line by saturate_local (see Chapter 38).

In general the right-hand side of a lowered entailment is $!!P\ \&\&\ R$, where P is a conjunction of propositions (Prop) and R is a separating conjunction of spatial predicates. The !! operator converts a Prop into an mpred.

This entailment's right-hand side has no spatial predicates. That's because, in the sumarray function, the SEP clause of the funspec's postcondition had exactly the same data_at clause as we see here in the entailment precondition, and the entailment-solver called by forward has already cleared it away.

We can proceed by using entailer! The remaining subgoal solves easily in the theory of sublists. The proof of the function sumarray is now complete.

20 *Global variables and* main()

C programs may have "extern" global variables, either with explicit initializers or initialized by default. Any function that accesses a global variable must have the appropriate spatial assertions in its funspec's precondition (and postcondition). But the main function is special: it has spatial assertions for *all* the global variables. Then it may pass these on, piecemeal, to the functions it calls on an as-needed basis.

The function-spec for the sumarray program's main is,

```
Definition main_spec :=
 DECLARE _main
   WITH gv : globals
   PRE [ ] main_pre prog gv
   POST [ tint ]
     (* application-specific postcondition *)
     PROP()
     RETURN(Vint (Int.repr (1+2+3+4)))
     SEP(TT).
```

The first four lines are always the same for any program. main_pre calculates the precondition automatically from the list of extern global variables and initializers of the program.

Now, when we prove that main satisfies its funspec,

```
Lemma body_main: semax_body Vprog Gprog f_main main_spec.
Proof.
start_function.
```

the start_function tactic "unpacks" main_pre into an assertion:

```
gv: globals
------------------------------------(1/1)
semax Delta
 (PROP () LOCAL(gvars gv)
  SEP(data_at Ews (tarray tuint 4)
      (map Vint [Int.repr 1; Int.repr 2; Int.repr 3; Int.repr 4]) (gv _four)))
 (... function body...)
 POSTCONDITION
```

The LOCAL clause expresses the function-precondition's GLOBALS(gv), constraining the global-variable map gv to the link-time environment. See Chapter 36.

The SEP clause means that there's data of type "array of 4 integers" at address (gv _four), with access permission Ews and contents [1;2;3;4]. Ews stands for "external write share," the standard access permission of extern global writable variables. See Chapter 48.

The sumarray program's main_spec *postcondition* is specific to this program: we say that main returns the value $1+2+3+4$.

The postcondition's SEP clause says TT; we cannot say simply SEP() because that is equivalent to emp in separation logic, enforcing the empty resource. But memory is not empty: it still contains all the initialized extern global variable four. So we give a looser spatial postcondition, TT (equivalent to True in separation logic).

21 Function calls

Continuing our example, the **Lemma** body_main in verif_sumarray.v:

Now it's time to prove the function-call statement, s = sumarray(four,4). When proving a function call, one must supply a *witness* for the WITH clause of the function-spec. The _sumarray function's WITH clause (page 15) starts,

```
Definition sumarray_spec :=
DECLARE _sumarray
 WITH a: val, sh : share, contents : list Z, size: Z
```

so the type of the witness will be (val∗(share∗(list Z ∗ Z))). To choose the witness, examine your actual parameter values (along with the precondition of the funspec) to see what witness would be consistent; here, we use (v_four,Ews,four_contents,4) as follows:

```
forward_call (v_four,Ews,four_contents,4).
```

The forward_call tactic (usually) leaves subgoals: you must prove that your current precondition implies the funspec's precondition. Here, these solve easily, as shown in the proof script.

Finally, we are at the return statement. See Chapter 19. In this case, the forward tactic is able to prove (using a form of the entailer tactic) that the current assertion implies the postcondition of _main.

22 Tying all the functions together

We build a whole-program proof by composing together the proofs of all the function bodies. Consider Gprog, the list of all the function-specifications:

Definition Gprog : funspecs := sumarray_spec :: main_spec :: nil.

Each semax_body proof says, assuming that *all the functions I might call* behave as specified, then *my own function-body* indeed behaves as specified:

Lemma body_sumarray: semax_body Vprog Gprog f_sumarray sumarray_spec.

Note that *all the functions I might call* might even include "myself," in the case of a recursive or mutually recursive function.

This might seem like circular reasoning, but (for partial correctness) it is actually sound—by the miracle of step-indexed semantic models, as explained in Chapters 18 and 39 of *Program Logics for Certified Compilers*.

The rule for tying the functions together is called semax_func, and its use is illustrated in this theorem, the main proof-of-correctness theorem for the program sumarray.c:

Lemma prog_correct: semax_prog prog Vprog Gprog.
Proof.
prove_semax_prog.
semax_func_cons body_sumarray.
semax_func_cons body_main.
Qed.

The calls to semax_func_cons must appear in the same order as the functions appear in prog.(prog_defs).

23 *Separation logic:* EX, $*$, emp, !!

These are the operators and primitives of *spatial predicates*, that is, the kind that can appear as conjuncts of a SEP.

R ::=	emp	empty
	TT	True
	FF	False
	$R_1 * R_2$	separating conjunction
	R_1 && R_2	ordinary conjunction
	field_at $\pi\ \tau\ \vec{fld}\ v\ p$	"field maps-to"
	data_at $\pi\ \tau\ v\ p$	"maps-to"
	array_at $\tau\ \pi\ v\ lo\ hi$	array slice
	!!P	pure proposition
	EX $x : T,\ R$	existential quantification
	ALL $x : T,\ R$	universal quantification
	$R_1 \| R_2$	disjunction
	wand $R\ R'$	magic wand $R \mathrel{-\!\!*} R'$
	...	other operators, including user definitions

24 EX, Intros, Exists

In a canonical-form lifted assertion, existentials can occur at the outside, or in one of the base-level conjuncts within the SEP clause. The left-hand side of this assertion has both:

```
ENTAIL Δ,        (* this example in progs/tutorial1.v *)
   EX x:Z,
    PROP(0≤x) LOCAL(temp _i (Vint (Int.repr x)))
    SEP(EX y:Z, !!(x < y) && data_at π tint (Vint (Int.repr y)) p)
⊢EX u: Z,
     PROP(0<u) LOCAL()
     SEP(data_at π tint (Vint (Int.repr u)) p)
```

To prove this entailment, one can first move x and y "above the line" by the tactic **Intros** a b:

```
  a: Z
  b: Z
  H: 0 ≤a
  H0: a < b
--------------------------------------------------------------------------
ENTAIL Δ,
    PROP() LOCAL(temp _i (Vint (Int.repr a)))
    SEP(data_at π tint (Vint (Int.repr b)) p)
⊢EX u: Z,
     PROP(0 < u) LOCAL()
     SEP(data_at π tint (Vint (Int.repr u)) p)
```

One might just as well say **Intros** x y to use those names instead of a b, or **Intros** ? ? to use x y automatically. Note that the propositions (previously hidden inside existential quantifiers) have been moved above the line by **Intros**. Also, if there had been any separating-conjunction operators $*$ within the SEP clause, those will be "flattened" into semicolon-separated conjuncts within SEP.

Sometimes, even when there are no existentials to introduce, one wants to move PROP propositions above the line and flatten the $*$ operators into semicolons. One can just say **Intros** with no arguments to do that. One can also say **Intros** $*$ to introduce all existentials, without specifying names, as well as moving PROP propositions and flattening $*$ operators.

If you want to Intro an existential *without* PROP-introduction and $*$-flattening, you can just use **Intro** a, instead of **Intros** a.

Then, instantiate u by **Exists** b.

a: Z
b: Z
H: $0 \le a$
H0: $a < b$

ENTAIL Δ,
PROP() LOCAL(temp _i (Vint (Int.repr a)))
SEP(data_at π tint (Vint (Int.repr b)) p)
$\vdash$ PROP($0 < b$) LOCAL()
SEP(data_at π tint (Vint (Int.repr b)) p)

This entailment proves straightforwardly by entailer!.

The EExists tactic takes no argument; it instantiates the existential with a unification variable (much like Coq's eexists versus exists).

Exists ? uses the binder name to instantiate the existential with the variable of the same name, if it exists. So using **Exists** ? instead of **Exists** b would've looked for a variable named u.

25 *Integers:* nat, Z, int

Coq's standard library has the natural numbers nat and the integers Z.

C-language integer values are represented by the type Int.int (or just int for short), which are 32-bit two's complement signed or unsigned integers with mod-2^{32} arithmetic. Chapter 59 describes the operations on the int type.

For most purposes, specifications and proofs of C programs should use Z instead of int or nat. Subtraction doesn't work well on naturals, and that screws up many other kinds of arithmetic reasoning. *Only when you are doing direct natural-number induction* is it natural to use nat, and so you might then convert using Z.to_nat to do that induction.

Conversions between Z and int are done as follows:

Int.repr: Z $\rightarrow$ int.
Int.unsigned: int $\rightarrow$ Z.
Int.signed: int $\rightarrow$ Z.

with the following lemmas:

$$\text{Int.repr\_unsigned}\ \frac{}{\mathsf{Int.repr}(\mathsf{Int.unsigned}\ z) \ = \ z}$$

$$\text{Int.unsigned\_repr}\ \frac{0 \le z \le \mathsf{Int.max\_unsigned}}{\mathsf{Int.unsigned}(\mathsf{Int.repr}\ z) \ = \ z}$$

$$\text{Int.repr\_signed}\ \frac{}{\mathsf{Int.repr}(\mathsf{Int.signed}\ z) \ = \ z}$$

$$\text{Int.signed\_repr}\ \frac{\mathsf{Int.min\_signed} \le z \le \mathsf{Int.max\_signed}}{\mathsf{Int.signed}(\mathsf{Int.repr}\ z) \ = \ z}$$

Int.repr truncates to a 32-bit twos-complement representation (losing information if the input is out of range). Int.signed and Int.unsigned are different injections back to Z that never lose information.

When doing proofs about signed integers, you must prove that your integers never overflow; when doing proofs about unsigned integers, it's still a good idea to prove that you avoid overflow. That is, if the C variable _x contains the value Vint (Int.repr x), then make sure x is in the appropriate range. Let's assume that _x is a signed integer, i.e. declared in C as int x; then the hypothesis is,

H: Int.min_signed ≤ x ≤ Int.max_signed *(* this example in progs/tutorial1.v *)*

If you maintain this hypothesis "above the line", then Floyd's tactical proof automation can solve goals such as Int.signed (Int.repr x) $= x$. Also, to solve goals such as,

```
...
H2 : 0 ≤ n ≤ Int.max_signed (* this example in progs/tutorial1.v *)
...
------------------------
Int.min_signed ≤ 0 ≤ n
```

you can use the rep_lia tactic (see Chapter 66), which is basically just lia with knowledge of the values of Int.min_signed, Int.max_signed, and Int.max_unsigned.

To take advantage of this, put conjuncts into the PROP part of your function precondition such as $0 \leq i < n$; $n \leq$ Int.max_signed. Then the start_function tactic will move them above the line, and the other tactics mentioned above will make use of them.

To see an example in action, look at progs/verif_sumarray.v. The funspec's precondition contains,

```
PROP(...   0 ≤size ≤Int.max_signed;
        Forall (fun x ⇒ 0 ≤x ≤Int.max_unsigned) contents)
```

to ensure that size is representable as a nonnegative signed integer, and each element of contents is representable as an unsigned.

26 *Int, Int8, Int16, Int64, Ptrofs*

C programs use signed and unsigned integers of various sizes: 8-bit (signed char, unsigned char), 16-bit (signed short, unsigned short), 32-bit (int, unsigned int), 64-bit (long, unsigned long).

A C compiler may be "32-bit" in which case sizeof(void∗)=4 or "64-bit" in which case sizeof(void∗)=8. The macro size_t is defined in the C standard library as a typedef for the appropriate signed integer, typically unsigned int on a 32-bit system and unsigned long on a 64-bit system.

To talk about integer *values* in all of these sizes, which have n-bit modular arithmetic (if unsigned) or n-bit twos-complement arithmetic (if signed), CompCert has several instantiations of the Integers module:

Int8 for char (signed or unsigned)
Int16 for short (signed or unsigned)
Int for int (signed or unsigned)
Int64 for long (signed or unsigned)
Ptrofs for size_t

where Ptrofs is isomorphic to the Int module (in 32-bit systems) and to the Int64 module (in 64-bit systems). You pronounce "Ptrofs" as "pointer offset" because it is frequently used to indicate the distance between two pointers into the same object.

The following definitions are used for shorthand:

Definition int = Int.int.
Definition int64 = Int64.int.
Definition ptrofs = Ptrofs.int.

27 *Values:* Vint, Vptr

```
Definition block : Type := positive.

Inductive val: Type :=
  | Vundef: val
  | Vint: int → val
  | Vlong: int64 → val
  | Vfloat: float → val
  | Vsingle: float32 → val
  | Vptr: block → ptrofs → val.
```

Vundef is the *undefined* value—found, for example, in an uninitialized local variable.

Vint(i) is an integer value, where i is a CompCert 32-bit integer. These 32-bit integers can also represent short (16-bit) and char (8-bit) values.

Vfloat(f) is a 64-bit floating-point value.
Vsingle(f) is a 32-bit floating-point value.

Vptr b z is a pointer value, where b is an abstract block number and z is an offset within that block. Different *malloc* operations, or different extern global variables, or stack-memory-resident local variables, will have different abstract block numbers. Pointer arithmetic must be done within the same abstract block, with $(\mathsf{Vptr}\, b\, z) + (\mathsf{Vint}\ i) \;=\; \mathsf{Vptr}\, b\, (z+i)$. Of course, the C-language + operator first multiplies i by the size of the array-element that $\mathsf{Vptr}\, b\, z$ points to.

Vundef is not always treated as distinct from a defined value. For example, $p \mapsto \mathsf{Vint}\, 5 \;\vdash\; p \mapsto \mathsf{Vundef}$, where $\mapsto$ is the data_at operator (Chapter 32). That is, $p \mapsto \mathsf{Vundef}$ really means $\exists v, p \mapsto v$. Vundef could mean "truly uninitialized" or it could mean "initialized but arbitrary."

28 C types

CompCert C describes C's type system with inductive data types.

```
Inductive signedness := Signed | Unsigned.
Inductive intsize := I8 | I16 | I32 | IBool.
Inductive floatsize := F32 | F64.

Record attr : Type := mk_attr {
  attr_volatile: bool; attr_alignas: option N
}.
Definition noattr := {| attr_volatile := false; attr_alignas := None |}.

Inductive type : Type :=
  | Tvoid: type
  | Tint: intsize → signedness → attr → type
  | Tlong: signedness → attr → type
  | Tfloat: floatsize → attr → type
  | Tpointer: type → attr → type
  | Tarray: type → Z → attr → type
  | Tfunction: typelist → type → calling_convention → type
  | Tstruct: ident → attr → type
  | Tunion: ident → attr → type

with typelist : Type :=
  | Tnil: typelist
  | Tcons: type → typelist → typelist.
```

We have abbreviations for commonly used types:

```
Definition tint = Tint I32 Signed noattr.
Definition tuint = Tint I32 Unsigned noattr.
Definition tschar = Tint I8 Signed noattr.
Definition tuchar = Tint I8 Unsigned noattr.
Definition tarray (t: type) (n: Z) = Tarray t n noattr.
Definition tptr (t: type) := Tpointer t noattr.
```

29 CompSpecs

The C language has a namespace for struct- and union-identifiers, that is, *composite types*. In this example, struct foo {int value; struct foo *tail} a,b; the "global variables" namespace contains a,b, and the "struct and union" namespace contains foo.

When you use CompCert clightgen to parse myprogram.c into myprogram.v, the main definition it produces is prog, the AST of the entire C program:

Definition prog : Clight.program := {| prog_types := composites; ... |}.

To interpret the meaning of a type expression, we need to look up the names of its struct identifiers in a *composite* environment. This environment, along with various well-formedness theorems about it, is built from prog as follows:

Require Import VST.floyd.proofauto. *(* Import Verifiable C library *)*
Require Import myprogram. *(* AST of my program *)*
Instance CompSpecs : compspecs. **Proof**. make_compspecs prog. **Defined**.

The make_compspecs tactic automatically constructs the *composite specifications* from the program. As a typeclass Instance, CompSpecs is supplied automatically as an implicit argument to the functions and predicates that interpret the meaning of types:

Definition sizeof {env: composite_env} (t: type) : Z := ...
Definition data_at_ {cs: compspecs} (sh: share) (t: type) (v: val) := ...

@sizeof (@cenv_cs CompSpecs) (Tint I32 Signed noattr) = 4.
sizeof (Tint I32 Signed noattr) = 4.
sizeof (Tstruct _foo noattr) = 8.
@data_at_ CompSpecs sh t v ⊢ data_at_ sh t v

When you have two separately compiled .c files, each will have its own prog and its own compspecs. See Chapter 82.

30 reptype

For each C-language data type, we define a *representation type*, the Type of Coq values that represent the contents of a C variable of that type.

```
Definition reptype {cs: compspecs} (t: type) : Type := ... .

Lemma reptype_ind: ∀(t: type),
 reptype t =
        match t with
        | Tvoid ⇒ unit
        | Tint _ _ _ ⇒ val
        | Tlong _ _ ⇒ val
        | Tfloat _ _ ⇒ val
        | Tpointer _ _ ⇒ val
        | Tarray t0 _ _ ⇒ list (reptype t0)
        | Tfunction _ _ _ ⇒ unit
        | Tstruct id _ ⇒ reptype_structlist (co_members (get_co id))
        | Tunion id _ ⇒ reptype_unionlist (co_members (get_co id))
        end
```

reptype_structlist is the right-associative cartesian product of all the (reptypes of) the fields of the struct. For example,

```
struct list {int hd; struct list *tl;};
struct one {struct list *p};
struct three {int a; struct list *p; double x;};

reptype (Tstruct _list noattr) = (val*val)
reptype (Tstruct _one noattr) = val
reptype (Tstruct _three noattr) = (val*(val*val))
```

We use val instead of int for the reptype of an integer variable, because the variable might be uninitialized, in which case its value will be Vundef.

31 *Uninitialized data,* default_val

CompCert represents uninitialized atomic (integer, pointer, float) values as Vundef : val.

The dependently typed function default_val calculates the undefined value for any C type:

default_val: ∀ {cs: compspecs} (t: type), reptype t.

For any C type t, the default value for variables of type t will have Coq type (reptype t).

For example:

struct list {int hd; struct list *tl;};

default_val tint = Vundef
default_val (tptr tint) = Vundef
default_val (tarray tint 4) = [Vundef; Vundef; Vundef; Vundef]
default_val (tarray t n) = list_repeat (Z.to_nat n) (default_val t)
default_val (Tstruct _list noattr) = (Vundef, Vundef)

32 data_at

Consider a C program with these declarations:

```
struct list {int hd; struct list *tl;} L;
int f(struct list a[5], struct list *p) { ... }
```

Assume these definitions in Coq:

```
Definition t_list := Tstruct _list noattr.
Definition t_arr := Tarray t_list 5 noattr.
```

Somewhere inside f, we might have the assertion,

PROP() LOCAL(temp _a a, temp _p p, gvars gv)
SEP(data_at Ews t_list (Vint (Int.repr 0), nullval) (gv _L);
    data_at π t_arr (list_repeat (Z.to_nat 5) (Vint (Int.repr 1), p)) a;
    data_at π t_list (default_val t_list) p)

This assertion says, "Local variable _a contains address a, _p contains address p, global variable _L is at address (gv _L). There is a struct list at (gv _L) with permission-share Ews ("extern writable share"), whose hd field contains 0 and whose tl contains a null pointer. At address a there is an array of 5 list structs, each with hd=1 and tl=p, with permission π; and at address p there is a single list cell that is uninitialized[1], with permission π."

In pencil-and-paper separation logic, we write $q \mapsto i$ to mean data_at Tsh tint (Vint (Int.repr i)) q. We write (gv _L)$\mapsto(0, \text{NULL})$ to mean data_at Tsh t_list (Vint (Int.repr 0), nullval) (gv _L). We write $p \mapsto (\_, \_)$ to mean data_at π t_list (default_val t_list) p.

In fact, the definition data_at_ is useful for the situation $p \mapsto \_$:

```
Definition data_at_ {cs: compspecs} sh t p := data_at sh t (default_val t) p.
```

[1] Uninitialized, or initialized but we don't know or don't care what its value is

33 field_at

Consider the example in progs/nest2.c

```
struct a {double x1; int x2;};
struct b {int y1; struct a y2;};
struct b p;
```

The command i = p.y2.x2; does a nested field load. We call y2.x2 the *field path*. The precondition for this command might include the assertion,

LOCAL(gvars gv)   SEP(data_at *sh* t_struct_b (u,(v,w)) (gv _pb))

The postcondition (after the load) would include the new LOCAL fact, temp _i w.

The tactic (unfold_data_at (data_at _ _ _(gv _p))) changes the SEP part of the assertion as follows:

SEP(field_at Ews t_struct_b (DOT _y1) (Vint u) (gv _pb);
    field_at Ews t_struct_b (DOT _y2) (Vfloat v, Vint w) (gv _pb))

and then doing (unfold_field_at 2%nat) unfolds the second field_at,

SEP(field_at Ews t_struct_b (DOT _y1) (Vint u) (gv _pb);
    field_at Ews t_struct_b (DOT _y2 DOT _x1) (Vfloat v) (gv _pb);
    field_at Ews t_struct_b (DOT _y2 DOT _x2) (Vint w) (gv _pb))

The third argument of field_at represents the *path* of structure-fields that leads to a given substructure. The empty path (nil) works too; it "leads" to the entire structure. In fact, data_at π τ v p is just short for field_at π τ nil v p.

Arrays and structs may be nested together, in which case the field path may also contain array subscripts at the appropriate places, using the notation SUB i along with DOT *field*.

34 data_at_, field_at_

An uninitialized data structure of type t, or a data structure with don't-care values, is said to contain the default value for t, default_val(t).

data_at sh t (default_val t) p

We abbreviate this with the definition data_at_:

data_at_ sh t p = data_at sh t (default_val t) p

Similarly, field_at_ sh t gfs p = field_at sh t gfs (default_val t) p.

35 reptype', repinj

This chapter is advanced material, describing a feature that is sometimes convenient but never necessary. You can skip this chapter.

```
struct a {double x1; int x2;};
struct b {int y1; struct a y2;} p;
repinj: ∀t: type, reptype' t → reptype t
reptype t_struct_b = (val*(val*val))
reptype' t_struct_b = (int*(float*int))
repinj t_struct_b (i,(x,j)) = (Vint i,(Vfloat x, Vint j))
```

The reptype function maps C types to the the corresponding Coq types of (possibly uninitialized) values. When we know a variable is definitely initialized, it may be more natural to use int instead of val for integer variables, and float instead of val for double variables. The reptype' function maps C types to the Coq types of (definitely initialized) values.

```
Definition reptype' {cs: compspecs} (t: type) : Type := ... .
Lemma reptype'_ind: ∀(t: type),
 reptype t =
       match t with
       | Tvoid ⇒ unit
       | Tint _ _ _ ⇒ int
       | Tlong _ _ ⇒ Int64.int
       | Tfloat _ _ ⇒ float
       | Tpointer _ _ ⇒ pointer_val
       | Tarray t0 _ _ ⇒ list (reptype' t0)
       | Tfunction _ _ _ ⇒ unit
       | Tstruct id _ ⇒ reptype'_structlist (co_members (get_co id))
       | Tunion id _ ⇒ reptype'_unionlist (co_members (get_co id))
       end
```

The function repinj maps an initialized value to the type of possibly uninitialized values:

```
Definition repinj {cs: compspecs} (t: type) : reptype' t → reptype t    := ...
```

The program progs/nest2.c (verified in progs/verif_nest2.v) illustrates the use of reptype' and repinj.

```
struct a {double x1; int x2;};
struct b {int y1; struct a y2;} p;

int get(void) { int i; i = p.y2.x2; return i; }
void set(int i) { p.y2.x2 = i; }
```

Our API spec for get reads as,

```
Definition get_spec :=
 DECLARE _get
  WITH v : reptype' t_struct_b, gv : globals
  PRE []
    PROP() LOCAL(gvars gv)
    SEP(data_at Ews t_struct_b (repinj _ v) (gv _p))
  POST [ tint ]
    PROP() RETURN(Vint (snd (snd v)))
    SEP(data_at Ews t_struct_b (repinj _ v) (gv _p)).
```

In this program, reptype' t_struct_b = (int∗(float∗int)), and repinj t_struct_b $(i,(x,j))$ = (Vint i, (Vfloat x, Vint j)).

One could also have specified get without reptype' at all:

```
Definition get_spec :=
 DECLARE _get
  WITH i: Z, x: float, j: int, gv : globals
  PRE []
    PROP() LOCAL(gvars gv)
    SEP(data_at Ews t_struct_b (Vint (Int.repr i), (Vfloat x, Vint j)) (gv _p))
  POST [ tint ]
    PROP() RETURN(Vint j)
    SEP(data_at Ews t_struct_b (Vint (Int.repr i), (Vfloat x, Vint j)) (gv _p)).
```

36 LOCAL *defs:* temp, lvar, gvars

The LOCAL part of a PROP()LOCAL()SEP() assertion is a list of localdefs that bind variables to their values or addresses.

```
Inductive localdef : Type :=
 | temp: ident → val → localdef
 | lvar: ident → type → val → localdef
 | gvars: globals → localdef.
```

temp i v binds a nonaddressable local variable i to its value v.
lvar i t v binds an *addressable* local variable i (of type t) to its *address* v.
gvars G describes the *addresses* of all global variables. Here, G maps global variable identifiers to their addresses (globals is just (ident → val)).

The *contents* of an addressable (local or global) variable is on the heap, and can be described in the SEP clause.

```
int g=2;
int f(void) { int g; int *p = &g; g=6; return g; }
```

In this program, the global variable g is shadowed by the local variable g. In an assertion inside the function body, one could still write

```
PROP() LOCAL(temp _p q; lvar _g tint q; gvars G}
SEP(data_at Ews tint (Vint (Int.repr 2)) (G _g);
      data_at Tsh tint (Vint (Int.repr 6)) q)
```

to describe a shadowed global variable _g that is still there in memory but (temporarily) cannot be referred to by its name in the C program.

37 go_lower

Normally one does not use this tactic directly, it is invoked as the first step of entailer or entailer!

Given a lifted entailment ENTAIL Δ, PROP($\vec{P}$) LOCAL($\vec{Q}$) SEP($\vec{R}$) $\vdash S$, one often wants to prove it at the base level: that is, with all of $\vec{P}$ moved above the line, with all of $\vec{Q}$ out of the way, just considering the base-level separation-logic conjuncts $\vec{R}$.

When Δ,$\vec{P}$,$\vec{Q}$,$\vec{R}$ are *concrete*, the go_lower tactic does this. Concrete means that the $\vec{P}$,$\vec{Q}$ are nil-terminated lists (not Coq variables) that every element of $\vec{Q}$ is manifestly a localdef (not hidden in Coq abstractions), the identifiers in $\vec{Q}$ are (computable to) ground terms, and the analogous (tree) property for Δ. It is not necessary that Δ,$\vec{P}$,$\vec{Q}$,$\vec{R}$ be fully *ground terms*: Coq variables (and other Coq abstractions) can appear anywhere in $\vec{P}$ and $\vec{R}$ and in the *value* parts of Δ and $\vec{Q}$. When the entailment is not fully concrete, or when there existential quantifiers outside PROP, the tactic old_go_lower can still be useful.

go_lower moves the propositions $\vec{P}$ above the line; when a proposition is an equality on a Coq variable, it substitutes the variable.

For each localdef in $\vec{Q}$ (such as temp i v), go_lower looks up i in Δ to derive a type-checking fact (such as tc_val t v), then introduces it above the line and simplifies it. For example, if t is tptr tint, then the typechecking fact simplifies to is_pointer_or_null v.

Then it proves the localdefs in S, if possible. If there are still some local-environment dependencies remaining in S, it introduces a variable rho to stand for the run-time environment.

The remaining goal will be of the form $\vec{R} \vdash S'$, with the semicolons in $\vec{R}$ replaced by the separating conjunction $*$. S' is the residue of S after lowering to the base separation logic and deleting its (provable) localdefs.

38 saturate_local

Normally one does not use this tactic directly, it is invoked by entailer or entailer!

To prove an entailment $R_1 * R_2 * \ldots * R_n \vdash !!(P'_1 \wedge \ldots P'_n) \&\& R'_1 * \ldots * R'_m$, first extract all the *local (nonspatial)* facts from $R_1 * R_2 * \ldots * R_n$, use them (along with other propositions above the line) to prove $P'_1 \wedge \ldots P'_n$, and then work on the separation-logic (spatial) conjuncts $R_1 * \ldots * R_n \vdash R'_1 * \ldots * R'_m$.

An example local fact: $\mathsf{data\_at}\ \mathsf{Ews}\ (\mathsf{tarray}\ \mathsf{tint}\ n)\ v\ p \vdash !!\ (\mathsf{Zlength}\ v = n)$. That is, the value v in an array "fits" the length of the array.

The Hint database saturate_local contains all the local facts that can be extracted from *individual* spatial conjuncts:

field_at_local_facts:

$$\begin{aligned}
&\mathsf{field_at}\ \pi\ t\ path\ v\ p\ \vdash !!(\mathsf{field_compatible}\ t\ path\ p \\
&\qquad\qquad\qquad \wedge\ \mathsf{value_fits}\ (\mathsf{nested_field_type}\ t\ path)\ v) \\
&\mathsf{data_at}\ \pi\ t\ v\ p\ \vdash !!(\mathsf{field_compatible}\ t\ \mathsf{nil}\ p \wedge \mathsf{value_fits}\ t\ v)
\end{aligned}$$

memory_block_local_facts:

$$\mathsf{memory\_block}\ \pi\ n\ p\ \vdash !!\ \mathsf{isptr}\ p$$

The assertion $(\mathsf{Zlength}\ v = n)$ is actually a consequence of value_fits when t is an array type. See Chapter 40.

If you create user-defined spatial terms (perhaps using EX, data_at, etc.), you can add hints to the saturate_local database as well.

The tactic saturate_local takes a proof goal of the form $R_1 * R_2 * \ldots * R_n \vdash S$ and adds saturate-local facts for *each* of the R_i, though it avoids adding duplicate hypotheses above the line.

39 *field_compatible, field_address*

CompCert C light comes with an "address calculus." Consider this example:

```
struct a {double x1; int x2;};
struct b {int y1; struct a y2;};
struct a *pa; int *q = &(pa→y2.x2);
```

Suppose the value of _pa is p. Then the value of _q is $p+\delta$; how can we reason about δ?

Given type t such as Tstruct _b noattr, and *path* such as (DOT _y2 DOT _x2), then (nested_field_type t *path*) is the type of the field accessed by that path, in this case tint; (nested_field_offset t *path*) is the distance (in bytes) from the base of t to the address of the field, in this case (on a 32-bit machine) 12 or 16, depending on the field-alignment conventions of the target machine (and the compiler).

On the Intel x86 architecture, where doubles need not be 8-byte-aligned, we have,

$$\begin{array}{l}\mathsf{data\_at}\ \pi\ \mathsf{t\_struct\_b}\ (i,(f,j))\ p\ \vdash \\ \qquad \mathsf{data\_at}\ \pi\ \mathsf{tint}\ i\ p\ *\ \mathsf{data\_at}\ \pi\ \mathsf{t\_struct\_a}\ (f,j)\ (\mathsf{offset\_val}\ p\ 12)\end{array}$$

but the converse is not valid:

$$\begin{array}{l}\mathsf{data\_at}\ \pi\ \mathsf{tint}\ i\ p\ *\ \mathsf{data\_at}\ \pi\ \mathsf{t\_struct\_a}\ (f,j)\ (\mathsf{offset\_val}\ p\ 12) \\ \qquad \nvdash\ \mathsf{data\_at}\ \pi\ \mathsf{t\_struct\_b}\ (i,(f,j))\ p\end{array}$$

The reasons: we don't know that $p+12$ satisfies the alignment requirements for struct b; we don't know whether $p+12$ crosses the end-of-memory boundary. That entailment *would* be valid in the presence of this hypothesis: field_compatible t_struct_b nil p : Prop.
which says that an entire struct b value *can* fit at address p. Note that

this is a *nonspatial* assertion about *addresses*, independent of the *contents* of memory.

In order to assist with reasoning about reassembly of data structures, saturate_local (and therefore entailer) puts field_compatible assertions above the line; see Chapter 38.

Sometimes one needs to name the address of an internal field—for example, to pass just that field to a function. In that case, one *could* use field_offset, but it is better to use field_address:

Definition field_address (t: type) ($path$: list gfield) (p: val) : val :=
  if field_compatible_dec t $path$ p
  then offset_val (Int.repr (nested_field_offset t $path$)) p
  else Vundef

That is, field_address has "baked in" the fact that the offset is "compatible" with the base address (is aligned, has not crossed the end-of-memory boundary). Therefore we get a valid converse for the example above:

data_at π tint i p
 $*$ data_at π t_struct_a (f,j) (field_address t_struct_b (DOT _y2 DOT _x2) p)
$\vdash$ data_at π t_struct_b $(i,(f,j))$ p

FIELD_ADDRESS VS FIELD_ADDRESS0. You use field_address t $path$ p to indicate that p points to **at least one** thing of the appropriate field type for $t.path$, that is, the type nested_field_type t $path$.

Sometimes when dealing with arrays, you want a pointer that might possibly point just one past the end of the array; that is, points to **at least zero** things. In this case, use field_address0 t $path$ p, which is built from field_compatible0. It has slightly looser requirements for how close p can be to the end of memory.

40 *value_fits*

The spatial maps-to assertion, data_at $\pi\ t\ v\ p$, says that there's a value v in memory at address p, filling the data structure whose C type is t (with permission π). A corollary is value_fits $t\ v$: v is a value that actually *can* reside in such a C data structure.

Value_fits is a recursive, dependently typed relation that is easier described by its induction relation; here, we present a simplified version that assumes that all types t are not volatile:

value_fits $t\ v$ = tc_val' $t\ v$ *(when t is an integer, float, or pointer type)*
value_fits (tarray $t'\ n$) v = (Zlength v = Z.max 0 n) ∧ Forall (value_fits t') v
value_fits (Tstruct i noattr) $(v_1,(v_2,(\ldots,v_n)))$ =
  value_fits (field_type $f_1\ v_1$) ∧ ... ∧ value_fits (field_type $f_n\ v_n$)
  (when the fields of struct i are $f_1,\ldots,f_n$)

The predicate tc_val' says,

Definition tc_val' (t: type) (v: val) := $v \neq$Vundef → tc_val $t\ v$.

```
Definition tc_val (t: type) : val → Prop :=
  match t with
  | Tvoid ⇒ False
  | Tint sz sg _ ⇒ is_int sz sg
  | Tlong _ _ ⇒ is_long
  | Tfloat F32 _ ⇒ is_single
  | Tfloat F64 _ ⇒ is_float
  | Tpointer _ _ | Tarray _ _ _ | Tfunction _ _ _ ⇒ is_pointer_or_null
  | Tstruct _ _ | Tunion _ _ ⇒ isptr
end
```

So, an atomic value (int, float, pointer) fits *either* when it is Vundef or when it type-checks. We permit Vundef to "fit," in order to accommodate partially initialized data structures in C.

Since τ is usually concrete, tc_val τ v immediately unfolds to something like,

TC0: is_int I32 Signed (Vint i)
TC1: is_int I8 Unsigned (Vint c)
TC2: is_int I8 Signed (Vint d)
TC3: is_pointer_or_null p
TC4: isptr q

TC0 says that i is a 32-bit signed integer; this is a tautology, so it will be automatically deleted by go_lower.

TC1 says that c is a 32-bit signed integer whose value is in the range of unsigned 8-bit integers (unsigned char). TC2 says that d is a 32-bit signed integer whose value is in the range of signed 8-bit integers (signed char). These hypotheses simplify to,

TC1: $0 \leq$ Int.unsigned c $\leq$ Byte.max_unsigned
TC2: Byte.min_signed $\leq$ Int.signed c $\leq$ Byte.max_signed

41 *cancel*

The cancel tactic proves associative-commutative rearrangement goals such as $(A_1 * A_2) * ((A_3 * A_4) * A_5) \vdash A_4 * (A_5 * A_1) * (A_3 * A_2)$.

If the goal has the form $(A_1 * A_2) * ((A_3 * A_4) * A_5) \vdash (A_4 * B_1 * A_1) * B_2$ where there is only a partial match, then cancel will remove the matching conjuncts and leave a subgoal such as $A_2 * A_3 * A_5 \vdash B_1 * B_2$.

cancel solves $(A_1 * A_2) * ((A_3 * A_4) * A_5) \vdash A_4 * \mathsf{TT} * A_1$ by absorbing $A_2 * A_3 * A_5$ into TT. If the goal has the form

$$\frac{F := ?224 : \mathsf{list(environ} \to \mathsf{mpred)}}{(A_1 * A_2) * ((A_3 * A_4) * A_5) \vdash A_4 * (\mathsf{fold\_right\ sepcon\ emp}\ F) * A_1}$$

where F is a *frame* that is an abbreviation for an uninstantiated logical variable of type list(environ→mpred), then the cancel tactic will perform *frame inference*: it will unfold the definition of F, instantiate the variable (in this case, to $A_2 :: A_3 :: A_5 :: \mathsf{nil}$), and solve the goal. The frame may have been created by evar(F: list(environ→mpred)), as part of forward symbolic execution through a function call.

WARNING: cancel can turn a provable entailment into an unprovable entailment. Consider this:

$$\frac{A * C \vdash B * C}{A * D * C \vdash C * B * D}$$

This goal is provable by first rearranging to $(A * C) * D \vdash (B * C) * D$. But cancel may aggressively cancel C and D, leaving $A \vdash B$, which is not provable. You might wonder, what kind of crazy hypothesis is $A*C \vdash B*C$; but indeed such "context-dependent" cancellations do occur in the theory of linked lists; see PLCC Chapter 19.

CANCEL DOES *not* USE $\beta\eta$ equality, as that could be slow in some cases. That means sometimes cancel leaves a residual subgoal $A \vdash A'$ where $A =_\beta A'$; sometimes the only differences are in (invisible) implicit arguments. You can apply derives_refl to solve such residual goals.

UNIFICATION VARIABLES. cancel does not instantiate unification variables, other than the Frame as described above. The ecancel tactic does instantiate evars (much like the difference between assumption and eassumption).

42 entailer!

The entailer, entailer!, and entailer!! tactics simplify (or solve entirely) entailments in either the lifted or base-level separation logic. The entailer never turns a provable entailment into an unprovable one; entailer! is more aggressive and more efficient, but sometimes (rarely) turns a provable entailment into an unprovable one. We recommend trying entailer! first.

When go_lower is applicable, the entailers start by applying it (see Chapter 37)—but entailer!! uses a streamlined version of go-lower, see below.

Then: saturate_local (see Chapter 38)—but entailer!! leaves this out, see below.

NEXT: on each side of the entailment, gather the propositions to the left: $R_1 * (!!P_1 \&\& (!!P_2 \&\& R_2))$ becomes $!!(P_1 \wedge P_2) \&\& (R_1 * R_2)$.

Move all left-hand-side propositions above the line; substitute variables. Autorewrite with entailer_rewrite, a *modest* hint database. If the r.h.s. or its first conjunct is a "valid_pointer" goal (or one of its variants), try to solve it.

At this point, entailer tries normalize and (if progress) back to NEXT; entailer! applies cancel to the spatial terms and prove_it_now to each propositional conjunct.

The result is that either the goal is entirely solved, or a residual entailment or proposition is left for the user to prove.

entailer!! The entailer! tactic extracts propositional information from LOCAL and SEP clauses, and puts it above the line. Examples: from LOCAL(lvar $i\ t\ p$) or SEP(data_at sh $t\ v\ p$), entailer! asserts that p is a pointer (isptr p). If you don't need all this extra information above the line when proving your goal, use entailer!! instead.

43 normalize

The normalize tactic performs autorewrite **with** norm and several other transformations. **Normalize can be slow: use Intros and entailer if they can do the job.**

The norm rewrite-hint database uses several sets of rules.

Generic separation-logic simplifications.

$$P * \mathsf{emp} = P \qquad \mathsf{emp} * P = P \qquad P \mathbin{\&\&} \mathsf{TT} = P \qquad \mathsf{TT} \mathbin{\&\&} P = P$$

$$P \mathbin{\&\&} \mathsf{FF} = \mathsf{FF} \qquad \mathsf{FF} \mathbin{\&\&} P = \mathsf{FF} \qquad P * \mathsf{FF} = \mathsf{FF} \qquad \mathsf{FF} * P = \mathsf{FF}$$

$$P \mathbin{\&\&} P = P \qquad (\mathsf{EX}\ \_ : A,\ P) = P \qquad \mathsf{local}\ \text{`}\mathsf{True} = \mathsf{TT}$$

Pull EX **and !! out of *-conjunctions.**

$$(\mathsf{EX}\ x : A,\ P) * Q = \mathsf{EX}\ x : A,\ P * Q \qquad (\mathsf{EX}\ x : A,\ P) \mathbin{\&\&} Q = \mathsf{EX}\ x : A,\ P \mathbin{\&\&} Q$$

$$P * (\mathsf{EX}\ x : A,\ Q) = \mathsf{EX}\ x : A,\ P * Q \qquad P \mathbin{\&\&} (\mathsf{EX}\ x : A,\ Q) = \mathsf{EX}\ x : A,\ P \mathbin{\&\&} Q$$

$$P * (!!Q \mathbin{\&\&} R) = !!Q \mathbin{\&\&} (P * R) \qquad (!!Q \mathbin{\&\&} P) * R = !!Q \mathbin{\&\&} (P * R)$$

Delete auto-provable propositions.

$$P \to (!!P \mathbin{\&\&} Q = Q) \qquad P \to (!!P = \mathsf{TT})$$

Integer arithmetic.

$$n + 0 = n \qquad 0 + n = n \qquad n * 1 = n \qquad 1 * n = n \qquad \mathsf{sizeof\ tuchar} = 1$$

$$\mathsf{align}\ n\ 1 = n \qquad (z > 0) \to (\mathsf{align}\ 0\ z = 0) \qquad (z \geq 0) \to (\mathsf{Z.max}\ 0\ z = z)$$

Int32 arithmetic.

$$\mathsf{Int.sub}\ x\ x\ =\ \mathsf{Int.zero} \qquad \mathsf{Int.sub}\ x\ \mathsf{Int.zero}\ =\ x$$

$$\mathsf{Int.add}\ x\ (\mathsf{Int.neg}\, x)\ =\ \mathsf{Int.zero} \qquad \mathsf{Int.add}\ x\ \mathsf{Int.zero}\ =\ x$$

$$\mathsf{Int.add}\ \mathsf{Int.zero}\ x\ =\ x$$

$$x \neq y \to \mathsf{offset\_val}(\mathsf{offset\_val}\ v\ i)\ j = \mathsf{offset\_val}\ v\ (\mathsf{Int.add}\ i\ j)$$

$$\mathsf{Int.add}(\mathsf{Int.repr}\ i)(\mathsf{Int.repr}\ j) = \mathsf{Int.repr}(i+j)$$

$$\mathsf{Int.add}(\mathsf{Int.add}\ z\ (\mathsf{Int.repr}\ i))\ (\mathsf{Int.repr}\ j)\ = \mathsf{Int.add}\ z\ (\mathsf{Int.repr}(i+j))$$

$$z > 0 \to (\mathsf{align}\ 0\ z\ =\ 0) \qquad \mathsf{force\_int}(\mathsf{Vint}\ i) = i$$

$$(\mathsf{min\_signed} \leq z \leq \mathsf{max\_signed}) \to \mathsf{Int.signed}(\mathsf{Int.repr}\ z) = z$$

$$(0 \leq z \leq \mathsf{max\_unsigned}) \to \mathsf{Int.unsigned}(\mathsf{Int.repr}\ z) = z$$

$$(\mathsf{Int.unsigned}\ i < 2^n) \to \mathsf{Int.zero\_ext}\ n\ i = i$$

$$(-2^{n-1} \leq \mathsf{Int.signed}\ i < 2^{n-1}) \to \mathsf{Int.sign\_ext}\ n\ i = i$$

map, fst, snd, . . .

$$\mathsf{map}\ f\ (x :: y) = f\,x :: \mathsf{map}\ f\ y \qquad \mathsf{map\ nil} = \mathsf{nil} \qquad \mathsf{fst}(x,y) = x$$

$$\mathsf{snd}(x,y) = y \qquad (\mathsf{isptr}\ v) \to \mathsf{force\_ptr}\ v = v \qquad \mathsf{isptr}\ (\mathsf{force\_ptr}\ v) = \mathsf{isptr}\ v$$

$$(\mathsf{is\_pointer\_or\_null}\ v) \to \mathsf{ptr\_eq}\ v\ v\ =\ \mathrm{True}$$

Unlifting.

$$`f\ \rho\ = f\ \ \text{[when f has arity 0]} \qquad `f\ a_1\ \rho\ = f\ (a_1\ \rho)\ \ \text{[when f has arity 1]}$$

$$`f\ a_1\ a_2\ \rho\ = f\ (a_1\ \rho)\ (a_2\ \rho)\ \ \text{[when f has arity 2, etc.]} \qquad (P * Q)\rho = P\rho * Q\rho$$

$$(P \mathbin{\&\&} Q)\rho = P\rho \mathbin{\&\&} Q\rho \qquad (!!P)\rho = !!P \qquad !!(P \wedge Q) = !!P \mathbin{\&\&} !!Q$$

$$(\mathsf{EX}\, x : A, P\,x)\rho\ =\ \mathsf{EX}\, x : A,\ P\,x\,\rho \qquad `(\mathsf{EX}\ x : B,\ P x) = \mathsf{EX}\ x : B,\ `(P x))$$

$$`(P * Q) =\ `P\ *\ `Q \qquad `(P \mathbin{\&\&} Q) =\ `P \mathbin{\&\&}\ `Q$$

Type checking and miscellaneous.

$$\text{tc\_andp tc\_TT } e \;=\; e \qquad\qquad \text{tc\_andp } e \text{ tc\_TT } \;=\; e$$

$$\text{eval\_id } x \text{ (env\_set } \rho\ x\ v) = v$$

$$x \neq y \rightarrow (\text{eval\_id } x \text{ (env\_set } \rho\ y\ v) = \text{eval\_id } x\ v)$$

$$\text{isptr } v \;\rightarrow\; (\text{eval\_cast\_neutral } v \;=\; v)$$

$$(\exists t.\, \text{tc\_val}\, t\, v \wedge \text{is\_pointer\_type}\, t) \;\rightarrow\; (\text{eval\_cast\_neutral } v \;=\; v)$$

Expression evaluation. (autorewrite with eval, but in fact these are usually handled just by simpl or unfold.)

$$\text{deref\_noload(tarray } t\ n) = (\text{fun } v \Rightarrow v) \qquad \text{eval\_expr(Etempvar } i\ t) = \text{eval\_id i}$$

$$\text{eval\_expr(Econst\_int i t)} = \text{`(Vint } i)$$

$$\text{eval\_expr(Ebinop } op\ a\ b\ t) =$$

$$\text{`(eval\_binop } op \text{ (typeof}\, a) \text{ (typeof}\, b)) \text{ (eval\_expr } a) \text{ (eval\_expr } b)$$

$$\text{eval\_expr(Eunop } op\ a\ t) = \text{`(eval\_unop } op \text{ (typeof}\, a)) \text{ (eval\_expr } a)$$

$$\text{eval\_expr(Ecast } e\ t) = \text{`(eval\_cast(typeof } e)\ t) \text{ (eval\_expr } e)$$

$$\text{eval\_lvalue(Ederef } e\ t) = \text{`force\_ptr (eval\_expr } e)$$

Function return values.

$$\text{get\_result(Some } x) = \text{get\_result1}(x) \qquad \text{retval(get\_result1 } i\ \rho) = \text{eval\_id } i\ \rho$$

$$\text{retval(env\_set } \rho \text{ ret\_temp } v) \;=\; v$$

$$\text{retval(make\_args(ret\_temp :: nil) } (v \text{ :: nil})\ \rho) \;=\; v$$

$$\text{ret\_type(initialized } i\ \Delta) = \text{ret\_type}(\Delta)$$

Postconditions. (autorewrite **with** ret_assert.)

$$\mathsf{normal\_ret\_assert\ FF\ ek\ vl} \;=\; \mathsf{FF}$$

$$\mathsf{frame\_ret\_assert(normal\_ret\_assert}\ P)\ Q = \mathsf{normal\_ret\_assert}\ (P * Q)$$

$$\mathsf{frame\_ret\_assert}\ P\ \mathsf{emp} \;=\; P$$

$$\mathsf{frame\_ret\_assert}\ P\ Q\ \mathsf{EK\_return}\ vl \;=\; P\ \mathsf{EK\_return}\ vl\ *\ Q$$

$$\mathsf{frame\_ret\_assert(loop1\_ret\_assert}\ P\ Q)\ R =$$

$$\mathsf{loop1\_ret\_assert}\ (P * R)(\mathsf{frame\_ret\_assert}\ Q\ R)$$

$$\mathsf{frame\_ret\_assert(loop2\_ret\_assert}\ P\ Q)\ R =$$

$$\mathsf{loop2\_ret\_assert}\ (P * R)(\mathsf{frame\_ret\_assert}\ Q\ R)$$

$$\mathsf{overridePost}\ P\ (\mathsf{normal\_ret\_assert}\ Q) = \mathsf{normal\_ret\_assert}\ P$$

$$\mathsf{normal\_ret\_assert}\ P\ ek\ vl \;= (!!(ek = \mathsf{EK\_normal}) \,\&\&\, (!!(vl = \mathsf{None}) \,\&\&\, P))$$

$$\mathsf{loop1\_ret\_assert}\ P\ Q\ \mathsf{EK\_normal\ None} \;=\; P$$

$$\mathsf{overridePost}\ P\ R\ \mathsf{EK\_normal\ None} = P$$

$$\mathsf{overridePost}\ P\ R\ \mathsf{EK\_return} \;=\; R\ \mathsf{EK\_return}$$

IN ADDITION TO REWRITING, normalize applies the following lemmas:

$$P \vdash \mathsf{TT} \qquad \mathsf{FF} \vdash P \qquad P \vdash P * \mathsf{TT} \qquad (\forall x.\ (P \vdash Q)) \to (EX x : A,\ P \vdash Q)$$

$$(P \to (\mathsf{TT} \vdash Q)) \to (!!P \vdash Q) \qquad (P \to (Q \vdash R)) \to (!!P \,\&\&\, Q \vdash R)$$

and does some rewriting and substitution when P is an equality in the goal, $(P \to (Q \vdash R))$.

Given the goal $x \to P$, where x is not a Prop, normalize avoids doing an intro. This allows the user to choose an appropriate name for x.

44 assert_PROP

Consider the proof state of verif_sumarray.v, just after (* Prove postcondition of loop body implies loop invariant. *). We have,

```
H : 0 ≤ i ≤ size
________________________
semax Delta
  (PROP () LOCAL(...)
   SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a))
  x = x[i]; ...
  POSTCONDITION
```

We desire, above the line, Zlength contents = size. This is not provable from anything above the line. But it is provable from the precondition (PROP/LOCAL/SEP).

Whenever a pure proposition (Prop) is provable from the precondition, you can bring it above the line using assert_PROP.

For example, assert_PROP(Zlength contents = size) gives you an entailment proof goal:

```
H : 0 ≤ i ≤ size
________________________
ENTAIL Delta,
  (PROP () LOCAL(...)
   SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a))
⊢ !! (Zlength contents = size).
```

Then, typically, you use entailer to prove the assertion. For example:

```
assert_PROP (Zlength contents = size). {
  entailer!. do 2 rewrite Zlength_map. reflexivity.
}
```

45 sep_apply

The sep_apply tactic is used to replace conjuncts in the precondition of an entailment. Suppose you have this situation:

H : C*A |– J
___
A∗B∗C∗D ⊢E

You can do sep_apply H to obtain,

H : C*A |– J
___
J∗B∗D ⊢E

Or suppose you have, **Lemma** L: ∀x y, F(x)∗G(y)=H(x,y)
and your proof goal is, A∗G(1)∗C∗F(2) ⊢E
then you can do sep_apply L to obtain, H(2,1)∗A∗C ⊢E.

sep_apply also works on the precondition of semax or on the SEP part of an ENTAIL goal.

Pure propositions: If your hypothesis or lemma has the form, P∗Q ⊢!!S then sep_apply behaves as if it were written P∗Q ⊢!!S && (P∗Q). That is, if the right-hand side is a pure proposition, then the left-hand-side is not deleted.

Rewriting: If your hypothesis or lemma has the form, P∗Q = R then sep_apply will apply P∗Q ⊢R.

46 Larger steps of automation

Sometimes one may not want to deal with each step of symbolic execution. In these cases, one can use fastforward, which will symbolically execute as many commands as it can.

If one wants to control how many commands fastforward executes, one can use fastforward n, where n is the amount desired. If one wants a more aggressive version, which normalizes more often, one can use fastforward! or fastforward! n instead.

If one wants to expand the tactic, one can redefine fastforward_semax_pre_simpl to run before calling forward or other related tactics like forward_if. One can also redefine fastforward_semax_post_simpl to run after all the other tactics tried fail.

One can also enable printing of the tactics being executed by,

```
Ltac2 Set fastforward_debug := true.
```

Another option is to attempt to fully solve the current goal, by using finish. Similarly to fastforward, there is a more aggressive variant of finish as well, finish!.

There are a variety of tactics that one can redefine to add to finish's functionality, as follows.

finish_pre_solve_simpl: To simplify the goal before trying to solve it

finish_retry_solve_simpl: To simplify the goal before trying to solve it again after failing. Used for more costly tactics.

finish_fast_solve: Like the name suggests, for tactics that solve and are fast. Not for entailments.

finish_slow_solve: Similarly, for tactics that solve and are slow. Not for entailments.

finish_entailer_solve: For tactics that solve entailments

There is also a HintDb named finish, which can be used to add Resolve, **Rewrite** and **Unfold** hints.

Similar to fastforward, one can also enable printing of the tactics being executed by,

Ltac2 **Set** finish_debug := true.

47 Welltypedness of variables

Verifiable C's typechecker ensures this about C-program variables: if a variable is initialized, then it contains a value of its declared type.

Function parameters (accessed by Etempvar expressions) are always initialized. Nonaddressable local variables (accessed by Etempvar expressions) and address-taken local variables (accessed by Evar) may be uninitialized or initialized. Global variables (accessed by Evar) are always initialized.

The typechecker keeps track of the initialization status of local nonaddressable variables, *conservatively:* if on all paths from function entry to the current point—assuming that the conditions on if-expressions and while-expressions are uninterpreted/nondeterministic—there is an assignment to variable x, then x is known to be initialized.

Addressable local variables do not have initialization status tracked by the typechecker; instead, this is tracked in the separation logic, by data_at assertions such as $v \mapsto \_$ (uninitialized) or $v \mapsto i$ (initialized).

Proofs using the forward tactic will typically generate proof obligations (for the user to solve) of the form,

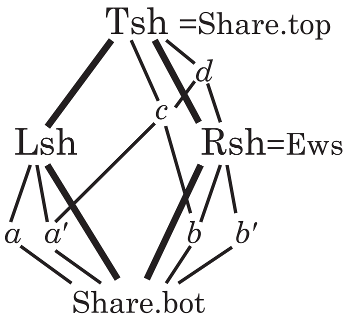
$$\text{ENTAIL } \Delta, \text{PROP}(\vec{P})\ \text{LOCAL}(\vec{Q})\ \text{SEP}(\vec{R}) \vdash \text{PROP}(\vec{P}')\ \text{LOCAL}(\vec{Q}')\ \text{SEP}(\vec{R}')$$

Δ keeps track of which nonaddressable local variables are initialized; says that all references to local variables contain values of the right type; and says that all addressable locals and globals point to an appropriate block of memory.

Using go_lower or entailer on an ENTAIL goal causes a tc_val assertion to be placed above the line for each initialized tempvar. As explained at page 63, this tc_val may be simplified into an is_int hypothesis, or even removed if vacuous.

48 Shares

Operators such as data_at take a *permission share*, expressing whether the assertion grants read permission, write permission, or some other fractional permission.



The *top* share, written Tsh or Share.top, gives total permission: to deallocate any cells within the footprint of this mapsto, to read, to write.

Share.split Tsh $=$ (Lsh, Rsh)
Share.split Lsh $= (a, a')$ Share.split Rsh $= (b, b')$
$a' \oplus b = c$ lub(c, Rsh) $= a' \oplus$ Rsh $= d$
$\forall sh.$ writable_share sh $\rightarrow$ readable_share sh
writable_share Ews readable_share b
writable_share d readable_share c
writable_share Tsh $\neg$readable_share Lsh

Any share may be split into a *left half* and a *right half*. The left and right of the top share are given distinguished names Lsh, Rsh.

The right-half share of the top share (or any share containing it such as d) is sufficient to grant *write permission* to the data: "the right share is the write share." A thread of execution holding only Lsh—or subshares of it such as a, a'—can neither read or write the object, but such shares are not completely useless: holding any nonempty share prevents other threads from deallocating the object.

Any subshare of Rsh, in fact any share that overlaps Rsh, grants *read* permission to the object. Overlap can be tested using the glb (greatest lower bound) operator.

Whenever (data_at sh t w v) holds, then the share sh must include at least a read share, thus this gives permission to load memory at address v to get a value w of type t.

To make sure sh has enough permission to write (i.e., Rsh $\subset sh$, we can say writable_share sh : Prop.

To test whether a share sh is empty or nonempty, use sepalg.identity sh or sepalg.nonidentity sh.

Writable extern global variables come with the "extern writable share" Ews; so does memoryobtained from malloc. Stack-allocated addressable locals come with the "top share" Tsh. Read-only globals come with the share Ers, the "extern readable share."

Sequential programs usually have little need of any shares except the Tsh and Ews. However, many function specifications can be parameterized over any share (example: sumarray_spec on page 15); that kind of generalized specification makes the functions usable in more contexts.

In C it is undefined to test deallocated pointers for equality or inequalities, so the Hoare-logic rule for pointer comparison also requires some permission-share; see page 79.

49 Pointer comparisons

In C, if p and q are expressions of type pointer-to-something, testing $p{=}q$ or $p!{=}q$ is defined only if: p is NULL, or points within a currently allocated object, or points at the end of a currently allocated object; and similarly for q. Testing $p{<}q$ (etc.) has even stricter requirements: p and q must be pointers into the *same* allocated object.

Verifiable C enforces this by creating "type-checking" conditions for the evaluation of such pointer-comparison expressions. Before reasoning about the result of evaluating expression $p{=}{=}q$, you must first prove tc_expr Δ (Ebinop Oeq (Etempvar _p (tptr tint)) (Etempvar _q (tptr tint))), where tc_expr is the type-checking condition for that expression. This simplifies into an entailment with the current precondition on the left, and denote_tc_comparable p q on the right.

The entailer(!) has a solver for such proof goals. It uses the hint database valid_pointer. It relies on spatial terms on the l.h.s. of the entailment, such as data_at π t v p which guarantees that p points to something.

The file progs/verif_ptr_compare.v illustrates pointer comparisons.

50 Proof of the reverse program

Program Logics for Certified Compilers, Chapter 3 shows a program that reverses a linked list (destructively, in place), along with a proof of correctness. (Chapters 2 and 3 available free here.)

That proof is based on a general notion of *list segments*. Here we show a simpler proof that does not use segments, but see Chapter 51 for proof that corresponds to Chapters 3 and 27 of PLCC.

The C program is in progs/reverse.c:

```
struct list {unsigned head; struct list *tail;};

struct list *reverse (struct list *p) {
  struct list *w, *t, *v;
  w = NULL;
  v = p;
  while (v) { t = v→tail;   v→tail = w;   w = v;   v = t; }
  return w;
}
```

Please open your CoqIDE or Proof General to progs/verif_reverse2.v. As usual, in progs/verif_reverse2.v we import the clightgen-produced file reverse.v and then build CompSpecs and Vprog (see page 14, Chapter 29, Chapter 52).

For the struct list used in this program, we can define the notion of *linked list* $x \overset{\sigma}{\leadsto}$ nil with a recursive definition:

```
Fixpoint listrep (sigma: list val) (x: val) : mpred :=
 match sigma with
 | h::hs ⇒   EX y:val, data_at Tsh t_struct_list (h,y) x * listrep hs y
 | nil ⇒     !! (x = nullval) && emp
 end.
```

That is, listrep σ x describes a null-terminated linked list starting at pointer p, with permission-share Tsh, representing the sequence σ.

The API spec (see also Chapter 8) for reverse is,

Definition reverse_spec :=
DECLARE _reverse
 WITH σ: list val, p: val
 PRE [_p OF (tptr t_struct_list)]
  PROP() LOCAL(temp _p p)SEP (listrep σ p)
 POST [(tptr t_struct_list)]
  EX q:val, PROP() LOCAL(temp _p q)SEP (listrep (rev σ) q).

The precondition says (for p the function parameter) $p \overset{\sigma}{\leadsto}$ nil, and the postcondition says that (for q the return value) $q \overset{\text{rev } \sigma}{\leadsto}$ nil.

In your IDE, enter the Lemma body_reverse and move after the start_function tactic. As expected, the precondition for the function-body is

PROP() LOCAL(temp _p p) SEP(listrep σ p).

After forward through two assignment statements (w=NULL; v=p;) the LOCAL part also contains temp _v p; temp _w (Vint (Int.repr 0)).

The loop invariant for the while loop is quite similar to the one given in PLCC Chapter 3 page 20:

$$\exists \sigma_1, \sigma_2.\ \ \sigma = \mathrm{rev}(\sigma_1) \cdot \sigma_2\ \wedge\ v \overset{\sigma_2}{\leadsto} 0 * w \overset{\sigma_1}{\leadsto} 0$$

It's quite typical for loop invariants to existentially quantify over the values that are different iteration-to-iteration. We represent this in PROP/LOCAL/SEP notation as,

 EX σ_1: list val, EX σ_2 : list val, EX w: val, EX v: val,
  PROP(σ = rev σ_1 ++ σ_2)
  LOCAL(temp _w w; temp _v v)
  SEP(listrep σ_1 w; listrep σ_2 v).

We apply forward_while with this invariant, and (as usual) we have four subgoals: (1) precondition implies loop invariant, (2) loop invariant implies typechecking of loop-termination test, (3) loop body preserves invariant, and (4) after the loop.

(1) To prove the precondition implies the loop invariant, we instantiate σ_1 with nil and σ_2 with σ; we instantiate w with NULL and v with p. But this leaves the goal,

ENTAIL Δ, PROP() LOCAL(temp _v p; temp _w nullval; temp _p p)
SEP(listrep σ p)
$\vdash$PROP(σ = rev [] ++ σ) LOCAL(temp _w nullval; temp _v p)
SEP(listrep [] nullval; listrep σ p)

The PROP and LOCAL parts are trivially solvable by the entailer. We can remove the SEP conjunct (listrep [] nullval) by unfolding that occurrence of listrep, leaving !!(nullval=nullval)&&emp.

(2) The type-checking condition is not trivial, as it is a pointer comparison (see Chapter 49), but the entailer! solves it anyway.

(3) The loop body starts by assuming the *loop invariant* and the truth of the *loop test*. Their propositional parts have already been moved above the line at the comment (* loop body preserves invariant *). That is, HRE: isptr v says that the loop test is true, and H: σ = rev σ_1 ++ σ_2 is from the invariant.

The first statement in the loop body, t=v→tail; loads from the list cell at v. But our SEP assertion for v is, listrep σ_2 v. The assertion listrep σ_2 v is not a data_at that we can load from. So we can unfold this occurrence of listrep, but *still* there is no data_at unless we know that σ_2 is $h :: r$ for some h, r.

We destruct σ_2 leaving two cases: σ_2 = nil and $\sigma_2 = h :: r$. The first case is a contradiction—by the definition of listrep, we must have v == nullptr, but that's incompatible with isptr(v) above the line.

In the second case, we have (below the line) $\exists y, \ldots$ that binds the value of the tail-pointer of the first cons cell. We move that above the line by **Intros** y.

NOW THAT THE FIRST LIST-CELL IS UNFOLDED, it's easy to go forward through the four commands of the loop body. Now we are (* at end of loop body, re-establish invariant *).

We choose values to instantiate the existentials: **Exists** (h::σ_1,r,v,y). (Note that forward_while has uncurried the four separate EX quantifiers into a single 4-tuple EX.) Then entailer! leaves two subgoals:

```
-------------------------------------------(1/2)
rev σ1 ++ h :: r = (rev σ1 ++ [h]) ++ r
-------------------------------------------(2/2)
  listrep σ1 w * field_at Tsh t_struct_list [] (h,w) v * listrep r y
⊢listrep (h :: σ1) v * listrep r y
```

Indeed, entailer! always leaves at most two subgoals: at most one propositional goal, and at most one cancellation (spatial) goal. Here, the propositional goal is easily dispatched in the theory of (Coq) lists.

The second subgoal requires unrolling the r.h.s. list segment, by unfolding the appropriate instance of listrep. Then we appropriately instantiate some existentials, call on the *entailer!* again, and the goal is solved.

(4) After the loop, we must prove that the loop invariant *and the negation of the loop-test condition* is a sufficient precondition for the next statement(s). In this case, the next statement is a return; one can *always* go forward through a return, but now we have to prove that our current assertion implies the function postcondition. This is fairly straightfoward.

51 *Alternate proof of* reverse

Chapter 27 of PLCC describes a proof of the same list-reverse program, based on a general theory of list segments. That proof is shown in progs/verif_reverse.v.

The general theory is in progs/list_dt.v. It accommodates list segments over any C struct type, no matter how many fields. Here, we import the LsegSpecial module of that theory, covering the "ordinary" case appropriate for the reverse.c program.

Require Import VST.progs.list_dt. **Import** LsegSpecial.

Then we *instantiate* that theory for our particular struct list by providing the listspec operator with the *names* of the struct (_list) and the link field (_tail).

Instance LS: listspec _list _tail.
Proof. eapply mk_listspec; reflexivity. **Defined**.

All other fields (in this case, just _head) are treated as "data" fields.

Now, lseg LS π σ p q is a list segment starting at pointer p, ending at q, with permission-share π and contents σ.

In general, with multiple data fields, the type of σ is constructed via reptype (see Chapter 30). In this example, with one data field, the type of σ computes to list val.

52 Global variables

In the C language, "extern" global variables live in the same namespace as local variables, but they are shadowed by any same-name local definition. In the C light operational semantics, global variables live in the same namespace as *addressable* local variables (both referenced by the expression-abstract-syntax constructor Evar), but in a different namespace from *nonaddressable* locals (expression-abstract-syntax constructor Etempvar).[1]

In the program-AST produced by clightgen, globals (and their initializers) are listed as Gvars in the prog_defs. These are accessed (automatically) in two ways by the Verifiable C program logic. First, their names and types are gathered into Vprog as shown on page 14 (try the Coq command Print Vprog to see this list). Second, their initializers are translated into data_at conjuncts of separation logic as part of the main_pre definition (see page 38).

When proving semax_body for the main function, the start_function tactic takes these definitions from main_pre and puts them in the precondition of the function body. In some cases this is done using the more-primitive mapsto operator[2], in other cases it uses the higher-level (and more standard) data_at[3].

---

[1]This difference in namespace treatment cannot matter in a program translated by CompCert clightgen from C, because no as-translated expression will exercise the difference.

[2]For example, examine the proof state in progs/verif_reverse.v immediately after start_function in Lemma body_main; and see the conversion to data_at done by the setup_globals lemma in that file.

[3]For example, examine the proof state in progs/verif_sumarray.v immediately after start_function in Lemma body_main.

53 For loops (special case)

MANY FOR-LOOPS HAVE THE FORM, for (*init*; i < *hi*; i++) *body* such that the expression *hi* will evaluate to the same value every time around the loop. This upper-bound expression need not be a literal constant, it just needs to be invariant.

For these loops you can use the tactic,

forward_for_simple_bound n (EX i:Z, PROP($\vec{P}$) LOCAL($\vec{Q}$) SEP($\vec{R}$)%assert.
forward_for_simple_bound n (EX i:Z, EX x:A, PROP...LOCAL...SEP...)%assert.

where n is the upper bound: a Coq value of type Z such that *hi* will evaluate to n. This tactic generates simpler subgoals than the general forward_for tactic.

The loop invariant is (EX i:Z, PROP($\vec{P}$) LOCAL($\vec{Q}$) SEP($\vec{R}$)), where i is the value (in each iteration) of the loop iteration variable _i. You *must* have an existential quantifier for the *value* of the loop-iteration variable. You *may* have a second ∃ for a value of your choice that depends on i.

You must omit from Q any mention of the loop iteration variable _i. The tactic will insert the binding temp _i i. You need not write $i \leq hi$ in P, the tactic will insert it.

AN EXAMPLE of a for-loop proof is in progs/verif_sumarray2.v. This is an alternate implementation of progs/sumarray.c (see Chapter 14) that uses a for loop instead of a while loop:

```
unsigned sumarray(unsigned a[], int n) { /* sumarray2.c */
  int i; unsigned s=0;
  for (i=0; i<n; i++) { s += a[i]; }
  return s;
}
```

Also see progs/verif_min.v for several approaches to the specification/verification of another for-loop.

54 For loops (general iterators)

The C-language for loop has the general form, for (*init*; *test*; *incr*) *body*. If your for-loop has an iteration variable that is tested by the *test* and adjusted by the *incr*, then you can probably use forward_for, described in this chapter. If not, use forward_loop (see the next chapter).

Let *Inv* be the loop invariant, established by the initializer and preserved by the body-plus-increment. Let *PreInc* be the assertion just before the increment. Both *Inv* and *PreInc* have type $A \rightarrow$ environ $\rightarrow$ mpred, where A is the Coq type of the abstract values carried by your iteraction variable; typically this is just Z.

Post is the join-postcondition of the loop; you don't need to provide it if *either* (1) there are no break statements in the loop, or (2) the postcondition is already provided in your proof context (typically because a close-brace follows the entire loop). Depending on whether you need *Post*, verify the loop with,

forward_for *Inv*. if your loop has no break or continue statements; or
forward_for *Inv* continue: *PreInc*. if no break statements; or
forward_for *Inv* continue: *PreInc* break: *Post*.

This is demonstrated in body_sumarray_alt from progs/verif_sumarray2.v.

```
unsigned sumarray(unsigned a[], int n) {
  int i; unsigned s;
  s=0;
  for (i=0;
    /* Inv : loop invariant */
          i<n; i++) {
    s += a[i];
    /* PreInc : pre-increment invariant */
  }
  /* Post : loop postcondition */
  return s;
}
```

55 *Loops (fully general)*

The C-language for loop has the general form, for (*init*; *test*; *incr*) *body*.

The C-language while loop with break and continue is equivalent to a for loop with empty *init* and *incr*.

The C-language infinite-loop, written for(;;)*c* or while(1)*c* is also a form of the for-loop.

The most general tactic for proving any of these loops is,
forward_loop *Inv* continue: *PreInc* break: *Post*.

The assertion *Inv* : environ → mpred is the loop invariant.
PreInc : environ → mpred is the invariant just before the *incr*.
The assertion *Post* : environ → mpred is the postcondition of the loop.

If your *incr* is empty (or Sskip), or if the *body* has no continue statements, you can omit continue: *PreInc*.

If your postcondition is already fully determined (POSTCOND contains no unification variables), then you can omit break: *Post*.

If you're not sure whether to omit the break: or continue: assertions, just try forward_loop *Inv* without them, and Floyd will advise you.

56 Manipulating preconditions

In some cases you cannot go forward until the precondition has a certain form. For example, to go forward through t=v→tail; there must be a data_at or field_at in the SEP clause of the precondition that gives a value for _tail field of t. As page 83 describes, a listrep can be unfolded to expose such a SEP conjunct.

Faced with the proof goal, semax Δ $(\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R}))$ c Post where $\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R})$ does not match the requirements for forward symbolic execution, you have several choices:

- Use the rule of consequence explicitly:
  apply semax_pre **with** PROP($\vec{P}'$)LOCAL($\vec{Q}'$)SEP($\vec{R}'$),
  then prove ENTAIL Δ, $\vec{P};\vec{Q};\vec{R} \vdash \vec{P}';\vec{Q}';\vec{R}'$.
- Use the rule of consequence implicitly, by using tactics (page 90) that modify the precondition.
- Do rewriting in the precondition, either directly by the standard rewrite and change tactics, or by normalize (page 68).
- Extract propositions and existentials from the precondition, by using **Intros** (page 43) or normalize.
- Flatten stars into semicolons, in the SEP clause, by **Intros**.
- Use the freezer (page 92) to temporarily "frame away" spatial conjuncts.

TACTICS FOR MANIPULATING PRECONDITIONS. In many of these tactics we select specific conjucts from the SEP items, that is, the semicolon-separated list of separating conjuncts. These tactic refer to the list by zero-based position number, 0,1,2,. . . .

For example, suppose the goal is a semax or entailment containing PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP(a;b;c;d;e;f;g;h;i;j). Then:

focus_SEP $i\ j\ k$. Bring items #i,j,k to the front of the SEP list.

focus_SEP 5. *results in* PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP(f;a;b;c;d;e;g;h;i;j).
focus_SEP 0. *results in* PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP(a;b;c;d;e;f;g;h;i;j).
focus_SEP 1 3. *results in* PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP(b;d;a;c;e;f;g;h;i;j)
focus_SEP 3 1. *results in* PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP(d;b;a;c;e;f;g;h;i;j)

gather_SEP $i\ j\ k$. Bring items #i,j,k to the front of the SEP list and conjoin them into a single element.

gather_SEP 5. *results in* PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP(f;a;b;c;d;e;g;h;i;j).
gather_SEP 1 3. *results in* PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP(b∗d;a;c;e;f;g;h;i;j)
gather_SEP 3 1. *results in* PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP(d∗b;a;c;e;f;g;h;i;j)

replace_SEP $i\ R$. Replace the ith element the SEP list with the assertion R, and leave a subgoal to prove.

replace_SEP 3 R. *results in* PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP(a;b;c;R;e;f;g;h;i;j).

with subgoal PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP(d) $\vdash R$.

replace_in_pre $S\ S'$. Replace S with S' anywhere it occurs in the precondition then leave $(\vec{P};\vec{Q};\vec{R}) \vdash (\vec{P};\vec{Q};\vec{R})[S'/S]$ as a subgoal.

frame_SEP $i\ j\ k$. Apply the frame rule, keeping only elements i,j,k of the SEP list. See Chapter 57.

57 The Frame rule

Separation Logic supports the Frame rule,

$$\text{Frame}\frac{\{P\}\,c\,\{Q\}}{\{P * F\}\,c\,\{Q * F\}}$$

In VST, we recommend you use the freeze tactic instead; see Chapter 58. But if you really want to use the frame rule, here is how.

Suppose you have the proof goal,

semax Δ PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP(R_0;R_1;R_2) $(c_1;c_2);c_3$ *Post*

and suppose you want to "frame out" R_1 for the duration of $c_1;c_2$, and have it back again for c_3.

First, you grab the first 2 statements using the tactic, first_N_statements 2%nat. (This works the same regardless of the nesting structure of the semicolons; it reassociates as needed.)

This leaves the two subgoals,

semax Δ PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP(R_0;R_1;R_2) $c_1;c_2$ (normal_ret_assert?88)
semax Δ ?88 c_3 *Post*

In the first subgoal, do frame_SEP 0 2 to retain only R_0;R_2.

semax Δ PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP(R_0;R_2) $c_1;c_2$ ...

Now you'll see that (in the precondition of the second subgoal) the unification variable ?88 has been instantiated in such a way that R_1 is added back in. Now you can prove the two subgoals, in order.

58 *The Freezer* (freeze, thaw)

Suppose you have the proof goal,

$$\mathsf{semax}\ \Delta\ \mathsf{PROP}(\vec{P})\mathsf{LOCAL}(\vec{Q})\mathsf{SEP}(R_0;R_1;R_2)\ \ c_1;c_2;c_3\ \ \textit{Post}$$

and suppose you want to "frame out" R_0 and R_2 for the duration of $c_1;c_2$, and have them back again for c_3. Instead of using the frame rule, you can use the freezer.

First, say freeze FR1 := R0 R2.
The name FR1 is up to you; R0 and R2 must be patterns (perhaps with underscores, for example (data_at _ _ _ p)) that match conjuncts from the SEP clause.

Now the proof goal looks like this:

$$\mathsf{semax}\ \Delta\ \mathsf{PROP}(\vec{P})\mathsf{LOCAL}(\vec{Q})\mathsf{SEP}(\mathrm{FRZL}\ \mathsf{FR1};\ R_1)\ \ c_1;c_2;c_3\ \ \textit{Post}$$

with a definition FR1 := ... above the line.

You can also write *freeze* $F := -\ \textit{pattern}_1\ \textit{pattern}_2 \ldots \textit{pattern}_n$
to freeze into F every conjunct *except* those that match the patterns.

Proceed forward through c_1 and c_2; then you can give the command thaw FR1 that unfolds (and clears) the FR1 definition.

Freezers can coexist and be arbitrarily nested, and be thawed independently; freezer-conjuncts participate in cancel and other separation-logic operations.

59 32-bit Integers

The VST program logic uses CompCert's 32-bit integer type.

Inductive comparison := Ceq | Cne | Clt | Cle | Cgt | Cge.
Int.wordsize: nat = 32.
Int.modulus : Z = 2^{32}.
Int.max_unsigned : Z = $2^{32}-1$.
Int.max_signed : Z = $2^{31}-1$.
Int.min_signed : Z = -2^{31}.

Int.int : Type.
Int.unsigned : int → Z.
Int.signed : int → Z.
Int.repr : Z → int.

Int.zero := Int.repr 0.

(* Operators of type int→int→bool *)
Int.eq Int.lt Int.ltu Int.cmp(c:comparison) Int.cmpu(c:comparison)

(* Operators of type int→int *)
Int.neg Int.not

(* Operators of type int→int→int *)
Int.add Int.sub Int.mul Int.divs Int.mods Int.divu Int.modu
Int.and Int.or Int.xor Int.shl Int.shru Int.shr Int.rol Int.ror Int.rolm

Lemma eq_dec: ∀(x y: int), {x = y} + {x <> y}.
Theorem unsigned_range: ∀i, 0 ≤ unsigned i < modulus.
Theorem unsigned_range_2: ∀i, 0 ≤ unsigned i ≤ max_unsigned.
Theorem signed_range: ∀i, min_signed ≤ signed i ≤ max_signed.
Theorem repr_unsigned: ∀i, repr (unsigned i) = i.
Lemma repr_signed: ∀i, repr (signed i) = i.
Theorem unsigned_repr:
    ∀z, 0 ≤ z ≤ max_unsigned → unsigned (repr z) = z.

Theorem signed_repr:
∀z, min_signed ≤ z ≤ max_signed → signed (repr z) = z.
Theorem signed_eq_unsigned:
∀x, unsigned x ≤ max_signed → signed x = unsigned x.

Theorem unsigned_zero: unsigned zero = 0.
Theorem unsigned_one: unsigned one = 1.
Theorem signed_zero: signed zero = 0.

Theorem eq_sym: ∀x y, eq x y = eq y x.
Theorem eq_spec: ∀(x y: int), **if** eq x y **then** x = y **else** x <> y.
Theorem eq_true: ∀x, eq x x = true.
Theorem eq_false: ∀x y, x <> y → eq x y = false.

Theorem add_unsigned: ∀x y, add x y = repr (unsigned x + unsigned y).
Theorem add_signed: ∀x y, add x y = repr (signed x + signed y).
Theorem add_commut: ∀x y, add x y = add y x.
Theorem add_zero: ∀x, add x zero = x.
Theorem add_zero_l: ∀x, add zero x = x.
Theorem add_assoc: ∀x y z, add (add x y) z = add x (add y z).

Theorem neg_repr: ∀z, neg (repr z) = repr (-z).
Theorem neg_zero: neg zero = zero.
Theorem neg_involutive: ∀x, neg (neg x) = x.
Theorem neg_add_distr: ∀x y, neg(add x y) = add (neg x) (neg y).

Theorem sub_zero_l: ∀x, sub x zero = x.
Theorem sub_zero_r: ∀x, sub zero x = neg x.
Theorem sub_add_opp: ∀x y, sub x y = add x (neg y).
Theorem sub_idem: ∀x, sub x x = zero.
Theorem sub_add_l: ∀x y z, sub (add x y) z = add (sub x z) y.
Theorem sub_add_r: ∀x y z, sub x (add y z) = add (sub x z) (neg y).
Theorem sub_shifted: ∀x y z, sub (add x z) (add y z) = sub x y.
Theorem sub_signed: ∀x y, sub x y = repr (signed x -signed y).

Theorem mul_commut: ∀x y, mul x y = mul y x.
Theorem mul_zero: ∀x, mul x zero = zero.
Theorem mul_one: ∀x, mul x one = x.
Theorem mul_assoc: ∀x y z, mul (mul x y) z = mul x (mul y z).
Theorem mul_add_distr_l: ∀x y z, mul (add x y) z = add (mul x z) (mul y z).
Theorem mul_signed: ∀x y, mul x y = repr (signed x ∗ signed y).

and many more axioms for the bitwise operators, shift operators, signed/unsigned division and mod operators.

60 CompCert C abstract syntax

The CompCert verified C compiler translates standard C source programs into an abstract syntax for *CompCert C*, and then translates that into abstract syntax for *C light*. Then VST Separation Logic is applied to the C light abstract syntax. C light programs proved correct using the VST separation logic can then be compiled (by CompCert) to assembly language.

C light syntax is defined by these Coq files from CompCert:

Integers. 32-bit (and 8-bit, 16-bit, 64-bit) signed/unsigned integers.
Floats. IEEE floating point numbers.
Values. The val type: integer + float + pointer + undefined.
AST. Generic support for abstract syntax.
Ctypes. C-language types and structure-field-offset computations.
Clight. C-light expressions, statements, and functions.

You will see C light abstract syntax constructors in the Hoare triples (semax) that you are verifying. We summarize the constructors here.

```
Inductive expr : Type :=
(* 1  *)       | Econst_int: int → type → expr
(* 1.0 *)      | Econst_float: float → type → expr (* double precision *)
(* 1.0f0 *)    | Econst_single: float → type → expr (* single precision *)
(* 1L *)       | Econst_long: int64 → type → expr
(* x *)        | Evar: ident → type → expr
(* x *)        | Etempvar: ident → type → expr
(* *e *)       | Ederef: expr → type → expr
(* &e *)       | Eaddrof: expr → type → expr
(* ~e *)       | Eunop: unary_operation → expr → type → expr
(* e+e *)      | Ebinop: binary_operation → expr → expr → type → expr
(* (int)e *)   | Ecast: expr → type → expr
(* e.f *)      | Efield: expr → ident → type → expr.
```

```
Inductive unary_operation := Onotbool | Onotint | Oneg | Oabsfloat.
Inductive binary_operation := Oadd | Osub | Omul | Odiv | Omod
 | Oand | Oor | Oxor | Oshl | Oeq | One | Olt | Ogt | Ole | Oge.

Inductive statement : Type :=
(* /**/;*)          | Sskip : statement
(* E1=E2; *)        | Sassign : expr → expr → statement (* memory store *)
(* x=E; *)          | Sset : ident → expr → statement (* tempvar assign *)
(* x=f(...); *)     | Scall: option ident → expr → list expr → statement
(* x=b(...); *)     | Sbuiltin: option ident → external_function → typelist →
                                                list expr → statement
(* s1; s2 *)        | Ssequence : statement → statement → statement
(* if() else {} *)  | Sifthenelse : expr → statement → statement → statement
(* for (;;s2) s1 *) | Sloop: statement → statement → statement
(* break; *)        | Sbreak : statement
(* continue; *)     | Scontinue : statement
(* return E; *)     | Sreturn : option expr → statement
                    | Sswitch : expr → labeled_statements → statement
                    | Slabel : label → statement → statement
                    | Sgoto : label → statement.
```

61 C light semantics

The operational semantics of C light statements and expressions is given in compcert/cfrontend/Clight.v. We do not expose these semantics *directly* to the user of Verifiable C. Instead, the *statement* semantics is reformulated as semax, an axiomatic (Hoare-logic style) semantics. The *expression* semantics is reformulated in veric/expr.v and veric/Cop2.v as a *computational*[1] *big-step evaluation semantics*. In each case, a soundness proof relates the Verifiable C semantics to the CompCert Clight semantics.

Rules for semax are given in veric/SeparationLogic.v—but you rarely use these rules directly. Instead, derived lemmas regarding semax are proved in floyd/*.v and Floyd's forward tactic applies them (semi)automatically.

The following functions (from veric/expr.v) define expression evaluation:

```
eval_id {CS: compspecs} (id: ident) : environ →val.
          (* evaluate a tempvar *)
eval_var {CS: compspecs} (id: ident) (ty: type) : environ →val.
          (* evaluate an lvar or gvar, addressable local or global variable *)
eval_cast (t t': type) (v: val) : val.
      (* cast value v from type t to type t', but beware! There are
        three types involved, including native type of v. *)
eval_unop (op: unary_operation) (t1 : type) (v1 : val) : val.
eval_binop{CS:compspecs} (op:binary_operation) (t1 t2: type) (v1 v2: val): val.
eval_lvalue {CS: compspecs} (e: expr) : environ →val.
     (* evalue an l-expression, one that denotes a loadable/storable place*)
eval_expr {CS: compspecs} (e: expr) : environ →val.
     (* evalue an r-expression, one that is not storable *)
```

The *environ* argument is for looking up the values of local and global variables. However, in most cases where Verifiable C users see eval_lvalue or eval_expr—in subgoals generated by the forward tactic—all the variables

[1]that is, defined by Fixpoint instead of by Inductive.

have already been substituted by values. Thus the environment is not needed.

The expression-evaluation functions call upon several helper functions from veric/Cop2.v:

```
sem_cast: type → type → val → option val.
sem_cast_* (* several helper functions for sem_cast *)
bool_val: type → val → option bool.
bool_val_*: (* helper functions *)
sem_notbool: type → val → option val.
sem_neg: type → val → option val.
sem_sub {CS: compspecs}: type → type → val → val → option val.
sem_sub_*: (* helper functions *)
sem_add {CS: compspecs}: type → type → val → val → option val.
sem_add_*: (* helper functions *)
sem_mul: type → type → val → val → option val.
sem_div: type → type → val → val → option val.
sem_mod: type → type → val → val → option val.
sem_and: type → type → val → val → option val.
sem_or: type → type → val → val → option val.
sem_xor: type → type → val → val → option val.
sem_shl: type → type → val → val → option val.
sem_shr: type → type → val → val → option val.
sem_cmp: comparison → type → type → (...) → val → val → option val.
sem_unary_operation: unary_operation → type → val → option val.
sem_binary_operation {CS: compspecs}:
    binary_operation → type → type → mem → val → val → option val.
```

The details are not so important to remember. The main point is that Coq expressions of the form sem_. . . *should* simplify away, provided that their arguments are instantiated with concrete operators, concrete constructors Vint/Vptr/Vfloat, and concrete C types. The *int* values (etc.) carried inside Vint/Vptr/Vfloat *do not* need to be concrete: they can be Coq variables. This is the essence of proof by symbolic execution.

62 *Splitting arrays*

Consider this example, from the main function of progs/verif_sumarray2.v:

data_at sh (tarray tuint k) al p : mpred

The data_at predicate here says that in memory starting at address p there is an array of k slots containing, respectively, the elements of the sequence al.

Suppose we have a function sumarray(unsigned a[], int n) that takes an array of length n, and we apply it to a "slice" of p: sumarray(p+i,k-i); where $0 \le i \le k$. The precondition of the sumarray funspec has data_at sh (tarray tint n) In this case, we would like $a = \&(p[i])$, $n = k - j$, and bl = the sublist of al from i to $k-1$.

To prove this function-call by forward_call, we must split up (data_at sh (tarray tint k) al p) into two conjuncts:
(data_at sh (tarray tint i) (sublist 0 i al) p $*$
data_at sh (tarray tuint $(k-i)$) (sublist i k al) q),
where q is the pointer to the array slice beginning at address $p+i$. We write this as, q = field_address0 (tarray tint k) [ArraySubsc i] p. That is, given a pointer p to a data structure described by (tarray tint k), calculate the *address* for subscripting the ith element. (See Chapter 39)

As shown in the body_main proof in progs/verif_sumarray2.v, the lemma split_array proves the equivalence of these two predicates, using the VST-Floyd lemma split2_data_at_Tarray. Then the data_at ... q predicate can satisfy the precondition of sumarray, while the p slice will be part of the "frame" for the function call.

See also: split3_data_at_Tarray.

63 sublist

Since VST 2.6, we recommend using the new list_solve *and* list_simplify *tactics, described in Chapter 64 and Chapter 65. Using **autorewrite with sublist** is less efficient, and in certain corner cases, can turn provable goals into unprovable goals.*

Chapter 62 explained that we often need to reason about slices of arrays whose contents are sublists of lists. For that we have a function sublist i j l which makes a new list out of the elements $i \ldots j-1$ of list l.

To simplify expressions involving, sublist, ++, map, Zlength, Znth, and list_repeat, use **autorewrite with sublist**.

Often, you find equations "above the line" of the form,

H: n = Zlength (map Vint (map Int.repr contents))

You may find it useful to do autorewrite **with** sublist **in** $*\vdash$ to change this to n=Zlength contents before proceeding with (autorewrite **with** sublist) below the line.

These rules comprise the sublist *rewrite database*:

sublist_nil': $i = j \rightarrow$ sublist i j l = [].
app_nil_l: [] ++ l = l.
app_nil_r: l ++ [] = l.
Zlength_rev: Zlength (rev l) = Zlength l.
sublist_rejoin': $0 \le i \le j = j' \le k \le$ Zlength l $\rightarrow$
    sublist i j l ++ sublist j' k l = sublist i k l.
subsub1: $a-(a-b)=b$.
Znth_list_repeat_inrange: $0 \le i \le n \rightarrow$ Znth i (list_repeat (Z.to_nat n) a) = a.
Zlength_cons: Zlength (a::l) = Z.succ (Zlength l).
Zlength_nil: Zlength [] = 0.
Zlength_app: Zlength (l ++ l') = Zlength l ++ Zlength l'.
Zlength_map: Zlength (map f l) = Zlength l.
list_repeat_0: list_repeat (Z.to_nat 0) = [].

Zlength_list_repeat: $0 \le n \to$ Zlength (list_repeat (Z.to_nat n)) $= n$.
Zlength_sublist: $0 \le i \le j \le$ Zlength $l \to$ Zlength(sublist i j l) $= j - i$.
sublist_sublist: $0 \le m \to 0 \le k \le i \le j - m \to$
sublist k i (sublist m j l) $=$ sublist $(k+m)$ $(i+m)$ l.
sublist_app1: $0 \le i \le j \le$ Zlength $l \to$ sublist i j $(l$ ++ $l')$ $=$ sublist i j l.
sublist_app2: $0 \le$ Zlength $l \le i \to$
sublist i j $(l$ ++ $l')$ $=$ sublist ($i -$ Zlength l) ($j -$ Zlength l) l'.
sublist_list_repeat: $0 \le i \le j \le k \to$
sublist i j (list_repeat (Z.to_nat k) v) $=$ list_repeat (Z.to_nat $(j-i)$) v.
sublist_same: $i = 0 \to j =$ Zlength $l \to$ sublist i j l $= l$.
app_Znth1: $i <$ Zlength $l \to$ Znth i $(l$ ++ $l')$ $=$ Znth i l.
app_Znth2: $i \ge$ Zlength $l \to$ Znth i $(l$ ++ $l')$ $=$ Znth $i -$ Zlength l l'.
Znth_sublist: $0 \le i \to 0 \le j < k - i \to$ Znth j (sublist i k l) $=$ Znth $(j + i)$ l.

along with miscellaneous Z arithmetic:

$$n - 0 = n \quad 0 + n = n \quad n + 0 = n \quad n \le m \to \max(n,m) = m$$
$$n + m - n = m \quad n + m - m = n \quad m - n + n = m \quad n - n = 0$$
$$n + m - (n + p) = m - p \qquad \text{etcetera.}$$

64 list_solve

One often needs to prove goals about lists. list_solve is a convenient solver for many practical proof goals involving lists.

list_solve supports operators: Zlength, Znth, nil ([]), cons (::), Zrepeat, app (++), upd_Znth, sublist, and map.

list_solve supports four kinds of typical proof goals:

- linear arithmetic involving lengths of lists,

  e.g. Zlength $(l \mathbin{++} l') \geq$ Zlength l;
- goal involving nth elements of lists (not limited to equality),

  e.g. Znth i $(l \mathbin{++} l')$ = Znth i l;
- equality of lists,

  e.g. $l_1 \mathbin{++} l_2 = l_3 \mathbin{++} l_4$;
- entailment of array contents,

  e.g. data_at sh (tarray τ n) $(l_1 \mathbin{++} l_2)$ p $\vdash$
  data_at sh (tarray τ n) $(l_3 \mathbin{++} l_4)$ p.

The way that list_solve supports assumptions in the following forms, is to interpret them as quantified properties:

- $l = l'$ is replaced by Zlength l = Zlength l' $\wedge$ $\forall i,\ 0 \leq i <$ Zlength l $\rightarrow$ Znth i l = Znth i l'.
- In x l is replaced by $\exists i,\ 0 \leq i <$ Zlength l $\wedge$ x = Znth i l.
- ~In x l is replaced by $\forall i,\ 0 \leq i <$ Zlength l $\rightarrow x \neq$ Znth i l.
- sorted $(\leq)$ l is replaced by $\forall i\ j,\ 0 \leq i \leq j <$ Zlength l $\rightarrow$
  Znth i $l \leq$ Znth j l.

The theory of lists with concatenation and nth-element is known to be undecidable.[1] So list_solve has such restriction that when encountering quantified properties like $\forall i,\ \mathsf{P}\ (\mathsf{Znth}\ i\ l)\ (\mathsf{Znth}\ (i+k)\ l)$, it asks user to prove $k = 0$ if it cannot prove it automatically. If $k = 0$ is not provable, list_solve does not support this goal. User might need to perform an induction before using list_solve.

list_simplify is an alternate tactic for list_solve, like ring_simplify to ring. It performs transformations in the same way as list_solve, and solves the goal if list_solve can solve, but leaves the unsolved goals to the user, so you may solve these goals by hand or figure out why the goal is not solved. list_simplify will not change a provable goal into unprovable goals.

[1]The reason is that an element may have relationship with other elements in the same list, directly or indirectly, and that leads to complicated deduction.. For example, $\mathsf{sublist}\ 1\ (\mathsf{Zlength}\ l)\ l = \mathsf{sublist}\ 0\ (\mathsf{Zlength}\ l\ \text{-}1)\ l$ indicates $\mathsf{Znth}\ i\ l = \mathsf{Znth}\ (i+1)\ l$ for every i and so all the elements in l are equal. Such kind of reasoning relies on induction and is hard to automate. Also see Aaron R. Bradley, Zohar Manna, and Henny B. Sipma, *What's Decidable About Arrays?*, Lecture Notes in Computer Science, vol 3855. Springer, Berlin, Heidelberg, 2006.

65 list_solve (advanced)

You can enhance list_solve by adding new rules.

Adding a macro A macro is an operator that can be expressed by other operators. For example,

Definition rotate {X} (l : list X) k :=
  sublist k (Zlength l) l ++ sublist 0 k l.

Add rotate to hint database, so list_solve will unfold it automatically.

Hint Unfold rotate : list_solve_unfold.

If a macro is expressed by other operators not by conversion but by Leibniz equality, e.g.

Lemma firstn_sublist: firstn (Z.to_nat i) l = sublist 0 i l,

add the lemma to rewrite database by

Hint Rewrite @firstn_sublist : list_solve_rewrite.

Adding a new kind of quantified property list_solve can be customized to handle predicates on lists that can be expressed by quantified properties. For example,

Lemma Forall_Znth : ∀{A} {d : Inhabitant A} l P,
  Forall P l ↔∀i, 0 ≤i < Zlength l →P (Znth i l).
Hint Rewrite Forall_Znth : list_prop_rewrite.

Adding a new operator list_solve handles operators, e.g. app and map, by using rules that reduce terms with head symbols Zlength and Znth to simpler terms, e.g.

Zlength_app: Zlength (l ++ l') = Zlength l + Zlength l',
Znth_map: Znth i (map f l) = f (Znth i l).

list_solve can support new operators if reduction rules are provided. For example, to add the operator rev : list ?A →list ?A that reverses a list, the following reduction rules should be provided:

Zlength_rev: Zlength (rev l) = Zlength l;
Znth_rev: $0 \leq i <$ Zlength $l \rightarrow$ Znth i (rev l) = Znth (Zlength l -i -1) l.

The following commands add these reduction rules to hint databases. Sometimes, "@" is necessary to prevent the rules from being specialized for a certain type before being added to the hint databases. The **using** keyword in commands tells the rewrite database to prove the side condition about index, $0 \leq i \leq$ Zlength l, by internal tactic Zlength_solve in list_solve.

Hint Rewrite Zlength_rev : Zlength.
Hint Rewrite @Znth_rev **using** Zlength_solve : Znth.

If the reduction rule for Zlength also has side conditions about indices, for example,

Zlength_map2: Zlength l_1 = Zlength $l_2 \rightarrow$ Zlength (map2 f l_1 l_2) = Zlength l_1,

the tactic to prove side condition should be provided to the rewrite database as

Hint Rewrite Zlength_map2 **using** (try Zlength_solve; fail **4**) : Zlength.

The adjusted failure level 4 is important for internal mechanism in list_solve.

There is another way to add rule for Zlength by hacking into list_solve's internal tactics. It utilizes caching mechanism in list_solve, so it is more efficient when the length of a list appears for multiple times. See commented code in progs/verif_dotprod.v and progs/verif_revarray.v for detail.

66 rep_lia: lia with representation facts [was rep_omega]

To solve goals such as

H: Zlength al < 50
----
0 ≤ Zlength al ≤ Int.max_signed

----
0 ≤ Int.unsigned (Int.repr i) ≤ Int.max_unsigned.

you want to use the lia tactic *augmented* by many facts about the *representations* of integers: the numeric values of Int.min_signed, Int.max_signed, etc.; the fact that Zlength is nonnegative; the fact that 0 ≤ Int.unsigned z ≤ Int.max_unsigned, and so on.

The rep_lia tactic does this. In addition, it "knows" all the facts in the **Hint Rewrite** : rep_lia database; see the next chapter.

67 *Opaque constants*

Suppose your C program has an array of a million elements:

```
int a[1000000];
```

Then you will have SEP conjuncts such as

```
data_at sh (tarray tint 1000000) (default_val (tarray tint 1000000)) p
```

That default_val (tarray tint 1000000) "simplifies" to:
Vundef::Vundef::...999997... Vundef::Vundef::nil, which will blow up Coq.

You might try to avoid blow-ups by writing,

```
Definition N = 1000000.
Opaque N.
data_at sh (tarray tint N) (default_val (tarray tint N)) p
```

and indeed, that's better (because simpl and simple apply won't unfold N), but it's not good enough (because reflexivity and auto *will* unfold N). See Coq issue #5301.

A better solution is:

```
Definition N : Z := proj1_sig (opaque_constant 1000000).
Definition N_eq : N=1000000 := proj2_sig (opaque_constant _).
Hint Rewrite N_eq : rep_lia.
```

This makes N opaque to all tactics, *except* that the rep_lia tactic (and any that use the rep_lia hint database) can expand N.

The progs/tutorial1.v, shows an example of this, in Lemmas exercise4 through exercise4c.

68 computable

One of the simplest, cheapest (in terms of Coq proof-term size) ways of solving a goal is with Coq's compute tactic. But sometimes compute blows up, if it's performed on a goal with opaque constants, or where call-by-value evaluation happens to be very expensive.

Floyd's computable tactic first examines the goal to make sure it won't blow up, and then solves it using compute (followed by other simple tactics), as long as the goal contains only the following operators:

```
(* nat constants *) O S        (* positive constants *) xH xI xO
(* Z constants *) Zpos Zneg Z0
(* Z operators *) + - * / mod max opp < ≤ > ≥ = <>
Ceq Cne Clt Cle Cgt Cge ∧
two_power_nat
{Int,Int64,Ptrofs}.{eq,lt,ltuadd,sub,mul,neg,cmp,cmpu,repr,signed,unsigned}
{Int,Int64,Ptrofs}.{max_unsigned,max_signed,min_signed,modulus,zwordsize}
(* any 0-arity (constant) definitions will be unfolded *)
```

You may add other operators to the computable hint database. For example, sizeof has already been added:

```
Lemma computable_sizeof: ∀cs x, computable x → computable (@sizeof cs x).
Proof. intros. apply computable_any. Qed.
Hint Resolve computable_sizeof : computable.
```

Adding this lemma to the Hint database tells the computable tactic to consider sizeof x "safe" to compute, as long as its argument x is computable.

69 Loop peeling and other manipulations

Sometimes a loop is easier to verify by first transforming it into another loop. For example, `for (init; test; incr) body` if not proved using the specialized for-loop tactic forward_for_simple_bound, must be proved by forward_for that requires *two* loop invariants: one just before the test and another just before the incr. (See Chapter 53 and Chapter 54.)

However, as long as the body does not contain any (outer-level) continue statements, then this loop is equivalent to `init; while (test) body` that can be proved using forward_while with just one continue statement. This equivalence is stated as the Lemma semax_loop_nocontinue (and its variant semax_loop_nocontinue1); the forward_for and forward_loop tactics apply this lemma automatically when appropriate, to relieve the user of the obligation of proving the just-before-the-incr invariant.

LOOP PEELING. In some loops, it makes sense to prove the first iteration differently than the rest; or the loop invariant is established *during* the first iteration instead of before it. For example, progs/verif_peel.v shows the verification of this loop:

```
int f (int b) {int i, a; for (i=b+1; i*i>b; i--) a=i; return a; }
```

The natural invariant, $0 \leq i < b < (i+1)*(i+1) \ \wedge \ a = i+1$, does not hold until the first iteration is completed.

Lemma semax_while_peel peels the first iteration from a while loop, as demonstrated in progs/verif_peel.v; Lemma semax_loop_unroll1 peels the first iteration of a general Sloop.

70 Later

Many of the Hoare rules (e.g., see PLCC, page 161) have the operater $\triangleright$ (pronounced "later") in their precondition:

$$\text{semax\_set\_forward}\frac{}{\Delta \vdash \{\triangleright P\}\ x := e\ \{\exists v. x = (e[v/x]) \wedge P[v/x]\}}$$

The modal assertion $\triangleright P$ is a slightly weaker version of the assertion P. It is used for reasoning by induction over how many steps left we intend to run the program. The most important thing to know about $\triangleright$later is that P is stronger than $\triangleright P$, that is, $P \vdash \triangleright P$; and that operators such as $*$, &&, ALL (and so on) commute with later: $\triangleright(P * Q) = (\triangleright P) * (\triangleright Q)$.

This means that if we are trying to apply a rule such as semax_set_forward; and if we have a precondition such as

$$\mathsf{local}\ (\mathsf{tc\_expr}\ \Delta\ \mathsf{e})\ \&\&\ \triangleright \mathsf{local}\ (\mathsf{tc\_temp\_id}\ \mathsf{id}\ \mathsf{t}\ \Delta\ \mathsf{e})\ \&\&\ (P_1 * \triangleright P_2)$$

then we can use the rule of consequence to *weaken* this precondition to

$$\triangleright (\mathsf{local}\ (\mathsf{tc\_expr}\ \Delta\ \mathsf{e})\ \&\&\ \mathsf{local}\ (\mathsf{tc\_temp\_id}\ \mathsf{id}\ \mathsf{t}\ \Delta\ \mathsf{e})\ \&\&\ (P_1 * P_2))$$

and then apply semax_set_forward. We do the same for many other kinds of command rules.

This weakening of the precondition is done automatically by the forward tactic, as long as there is only one $\triangleright$later in a row at any point among the various conjuncts of the precondition.

A more sophisticated understanding of $\triangleright$ is needed to build proof rules for recursive data types and for some kinds of object-oriented programming; see PLCC Chapter 19.

71 Mapsto and func_ptr

Aside from the standard operators and axioms of separation logic, the core separation logic has just two primitive spatial predicates:

Parameter address_mapsto:
 memory_chunk → val → share → share → address → mpred.
Parameter func_ptr : funspec → val → mpred.

func_ptr ϕ v means that value v is a pointer to a function with specification ϕ; see Chapter 78.

address_mapsto expresses what is typically written $x \mapsto y$ in separation logic, that is, a singleton heap containing just value y at address x.

From this, we construct two low-level derived forms:

mapsto (sh:share) (t:type) (v w: val) : mpred describes a singleton heap with just one value w of (C-language) type t at address v, with permission-share sh.

mapsto_ (sh:share) (t:type) (v:val) : mpred describes an *uninitialized* singleton heap with space to hold a value of type t at address v, with permission-share sh.

From these primitives, field_at and data_at are constructed.

72 gvars: Private global variables

If your C module (typically, a .c file, but it could be part of a .c file or several .c files) accesses private global variables, you may want to avoid mentioning their names in the public interface.

Definition MyModuleGlobs (gv: globals) : mpred :=
  (∗ for example ∗) data_at Tsh t_struct_foo some_value (gv _MyVar).

DECLARE _myfunction
WITH ..., gv: globals
PRE [t_1,t_2]
  PROP(...) PARAMS(v_1;v_2) GLOBALS(gv) SEP(...; MyModuleGlobs gv)
POST [...]
  PROP() RETURN(...) SEP(...; MyModuleGlobs gv).

The client of myfunction sees that there is a private conjunct MyModuleGlobs gv that (presumably) uses some global variables of MyModule, but it does not see their names.

THE FILE progs/verif_libglob.v demonstrates the verification of a module that uses private global variables.

Inside the semax_body proof of _myfunction, the PARAMS/GLOBALS is transformed as follows:

PROP(...)
LOCAL(temp _x1 v_1; temp _x2 v_2; gvars gv)
SEP(...; MyModuleGlobs gv)

That is, the temp components of the LOCAL give access to specific local variables, and the gvars component gives access to *all* the global variables.

73 with_library: *Library functions*

A CompCert C program is implicitly linked with dozens of "built-in" and library functions. In the .v file produced by clightgen, the prog_defs component of your prog lists these as External definitions, along with the Internal definitions of your own functions. *Every one of these needs exactly one funspec,* of the form DECLARE...WITH..., and this funspec must be *proved* with a semax_ext proof.

Fortunately, if your program does not use a given library function f, then the funspec DECLARE _f WITH...PRE[...] False POST... with a **False** precondition is easy to prove! The tactic with_library *prog* [s_1;s_2;...;s_n] augments your explicit funspec-list [s_1;s_2;...;s_n] with such trivial funspecs for the other functions in the program *prog*.

```
Definition Gprog := ltac:(with_library prog [sumarray_spec; main_spec]).
```

YOU MAY WISH to use standard library functions such as malloc, free, exit. These are axiomatized (with external funspecs) in floyd.library. To use them, **Require Import** VST.floyd.library *after* you import floyd.proofauto. This imports a (floyd.library.)with_library tactic hiding the standard (floyd.forward.)with_library tactic; the new one includes *axiomatized* specifications for malloc, free, exit, etc. We haven't proved the implementations against the axioms, so if you don't trust them, then don't import floyd.library.

The next chapters explain the specifications of certain standard-library functions.

74 *malloc / free*

The C library's malloc and free functions have these specifications:

```
DECLARE _malloc
  WITH cs: compspecs, t:type
  PRE [ tuint ]
    PROP(0 ≤ sizeof t ≤ Int.max_unsigned;
          complete_legal_cosu_type t = true;
          natural_aligned natural_alignment t = true)
    PARAMS(Vint (Int.repr (sizeof t)))
    SEP()
  POST [ tptr tvoid ] EX p:_,
    PROP()
    RETURN(p)
    SEP(if eq_dec p nullval then emp
        else (malloc_token Ews t p * data_at_ Ews t p)).

DECLARE _free
   WITH cs: compspecs, t: type, p:val
   PRE [ tptr tvoid ]
       PROP()
       PARAMS(p)
       SEP(malloc_token Ews t p; data_at_ Ews t p)
    POST [ Tvoid ]
       PROP() RETURN() SEP().
```

You must **Import** VST.floyd.library. Then the with_library tactic (Chapter 73) makes these funspecs available in your Gprog.

The purpose of the malloc_token is to describe the special record-descriptor that tells free how big the allocated record was. See progs/verif_queue.v for a demonstration of malloc/free.

75 *exit*

Import VST.floyd.library. before you define
Gprog := ltac:(with_library prog [...]).
and you will get:

```
DECLARE _exit
 WITH errcode: Z
 PRE [ tint ]
   PROP() PARAMS(errcode) SEP()
 POST [ tvoid ]
   PROP(False) RETURN() SEP().
```

76 Union casting

Normally in C, if you store a value to one field of a union type, you should fetch back from the same field. But there are some special cases where you can perform a type conversion by storing to one field, then fetching from another. These examples are illustrated in progs/union.c and progs/verif_union.c.

```
/* convert const char* to char */
const char *x; char *y;
union const_or_not {const char *c; char *n; } u;
u.c = x;
y = u.n;
```

This conversion is "easy" because VST's type for const char * is exactly the same as its type for char *. We just need this lemma:

```
Lemma unconst_aux:
  ∀(x: val) v,
      data_at Tsh (Tunion _const_or_not noattr) (inl x) v =
      data_at Tsh (Tunion _const_or_not noattr) (inr x) v.
Proof. reflexivity. Qed.
```

Here, the inl x and inr x correspond to the left and right sides of the Coq sum type that VST uses to represent *the data stored in* the first or second fields of C's union type. And in this case, the since the field types are the same, the data_at representations are also the same. You can use this lemmas in the proof of the two commands u.c=x; y=u.n;

```
forward. rewrite uconst_aux. forward.
```

CONVERTING A FLOAT TO/FROM ITS INTEGER REPRESENTATION as sign, exponent, mantissa can be done as follows:

```
float x; unsigned int n;
union f_or_i {float f; unsigned int i; } u;
u.f = x; n = u.i;
```

In this case, the representation of a Vfloat is not the same as a Vint, but VST's forward tactic has a special hack built in. When two C commands appear sequentially, where the first stores to field A a union, and the second loads from a different field B of the same union; and where *both* are numeric types of the same size; then the postcondition of the first command has a data_at adapted for B rather than A. You'll also get a warning message about "Converting numeric representations".

When two C commands appear in a row, where the first stores to field A a union, and the second loads from a different field B of the same union, but A and B are *not* numeric types of the same size, then you get a warning message "Suggestion: you are storing to one field of a union, then loading from another...". In such a case, you may wish to rewrite by a lemma such as unconst_aux as described above.

You can disable these warning messages by,

```
Ltac union_hack_message id1 id2 ::= idtac.
Ltac numeric_forward_store_union_hack id1 id2 ::= idtac.
```

77 Old-style funspecs

Until VST version 2.5, function preconditions were written a different way. Instead of writing PROP/PARAMS/GLOBALS/SEP they were written as PROP/LOCAL/SEP. Here's an example; compare with the new-style funspec on page 15.

```
Definition sumarray_spec : ident * funspec :=
DECLARE _sumarray
 WITH a: val, sh : share, contents : list Z, size: Z
 PRE [ _a OF (tptr tuint), _n OF tint ]
    PROP(readable_share sh;
          0 ≤ size ≤ Int.max_signed;
          Forall (fun x ⇒ 0 ≤ x ≤ Int.max_unsigned) contents)
    LOCAL(temp _a a; temp _n (Vint (Int.repr size)))
    SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a)
  POST [ tuint ]
    PROP()
    LOCAL(temp ret_temp (Vint (Int.repr (sum_Z contents))))
    SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a).
```

Notice also that the PRE list is different: each parameter is written _x OF t, where _x is the C-language identifier used in the program.

In the old funspec notation, the return-value part of the postcondition is written LOCAL(temp ret_temp v) instead of RETURN(v).

VST proofs that use old-style funspecs should access the old-style notation and old-style definitions by,

Require Import VST.floyd.Funspec_old_**Notation**.

This brings in a different notation scope, in which WITH/PRE/POST works differently.

Whenever you do start_function (in the semax_body of an old-style funspec) or forward_call (calling a function with an old-style funspec), the Floyd tactics automatically convert to a new-style funspec. For that conversion to work, the tactics must be able to prove (from what's above the line, and from the PROP and SEP clauses) that each of the temp values is not Vundef.

78 Function pointers

Parameter func_ptr : funspec →val →mpred.
Definition func_ptr' f v := func_ptr f v && emp.

func_ptr ϕ v	means that v is a pointer to a function with funspec ϕ.
func_ptr' ϕ v	is a form more suitable to be a conjunct of a SEP clause.

Verifiable C's program logic is powerful enough to reason expressively about function pointers (see PLCC Chapters 24 and 29). Object-oriented programming with function pointers is illustrated, in two different styles, by the programs progs/message.c and progs/object.c, and their verifications, progs/verif_message.c and progs/verif_object.c.

In this chapter, we illustrate using the much simpler program, progs/funcptr.c.

```
int myfunc (int i) { return i+1; }
void *a[] = {myfunc};
int main (void) {
  int (*f)(int);
  int j;
  f = &myfunc;
  j = f(3);
  return j;
}
```

The verification, in progs/verif_funcptr.v, defines

Definition myfunc_spec := DECLARE _myfunc myspec.

where myspec is a Definition for a WITH...PRE...POST specification.

Near the beginning of **Lemma** body_main, notice that we have GLOBALS(gv) in the precondition. That gv is needed by the tactic make_func_ptr _myfunc, which adds func_ptr' myspec (gv _myfunc) to the

SEP clause. It “knows” to use myspec because it looks up _myfunc in Delta (which, in turn, is derived from Gprog).

Now, forward through the assigment f=myfunc works as you might expect, adding the LOCAL clause temp _f p.

To call a function-variable, such as this program’s j=f(3); just use forward_call as usual. However, in such a case, forward_call will find its funspec in a func_ptr' SEP-clause, rather than as a global entry in Delta as for ordinary function calls.

79 Axioms of separation logic

These axioms of separation logic are often useful, although generally it is the automation tactics (entailer,cancel) that apply them.

pred_ext: P⊢Q → Q⊢P → P=Q.
derives_refl: P ⊢P.
derives_trans: P ⊢Q → Q ⊢R → P⊢R.
andp_right: X⊢P → X⊢Q → X⊢(P&&Q).
andp_left1: P⊢R → P&&Q ⊢R.
andp_left2: Q⊢R → P&&Q ⊢R.
orp_left: P⊢R → Q⊢R → P||Q ⊢R.
orp_right1: P⊢Q → P⊢ Q||R.
orp_right2: P⊢R → P⊢ Q||R.
exp_right: ∀{B: Type}(x:B)(P:mpred)(Q: B →mpred),
    P⊢Q x → P⊢ EX x:B, Q.
exp_left: ∀{B: Type}(P:B →mpred)(Q:mpred),
    (∀ x, P x ⊢Q) → EX x:B,P ⊢Q.
allp_left: ∀{B}(P: B →mpred) x Q, P x⊢Q → ALL x:B,P⊢Q.
allp_right: ∀{B}(P: mpred)(Q:B →mpred),
    (∀ v, P⊢ Q v) → P⊢ ALL x:B,Q.
prop_left: ∀(P: Prop) Q, (P →(TT⊢Q)) → !!P ⊢Q.
prop_right: ∀(P: Prop) Q, P → (Q⊢ !!P).
not_prop_right: ∀(P:mpred)(Q:Prop), (Q →(P⊢FF)) → P⊢!!(~Q).

sepcon_assoc: (P∗Q)∗R = P∗(Q∗R).
sepcon_comm: P Q, P∗Q = Q∗P.
sepcon_andp_prop: P∗(!!Q && R) = !!Q && (P∗R).
derives_extract_prop: (P →Q ⊢R) → !!P && Q ⊢R.
sepcon_derives: P⊢P' → Q⊢Q' → P∗Q ⊢P'∗Q'.

80 Obscure higher-order axioms

The wand $-\!\!*$ operator is "magic wand," ewand $-\!\circ$ is "existential magic wand," and $\triangleright$ is pronounced "later" and written |> in Coq.

see PLCC, Chapter 19.

imp_andp_adjoint: $\mathsf{P\&\&Q \vdash R \;\leftrightarrow\; P \vdash (Q \longrightarrow R)}$.
wand_sepcon_adjoint: $\mathsf{P*Q \vdash R \;\leftrightarrow P \vdash Q -\!\!* R}$.
ewand_sepcon: $\mathsf{(P*Q) -\!\circ R = P -\!\circ (Q -\!\circ R)}$.
ewand_TT_sepcon: $\mathsf{\forall (P\ Q\ R{:}\ A)}$,
$\qquad \mathsf{(P*Q)\&\&(R -\!\circ TT) \vdash (P\ \&\&(R -\!\circ TT))*(Q\ \&\&\ (R -\!\circ TT))}$.
exclude_elsewhere: $\mathsf{P*Q \vdash (P\ \&\&(Q -\!\circ\ TT))*Q}$.
ewand_conflict: $\mathsf{P*Q \vdash FF \;\rightarrow\; P\&\&(Q -\!\circ R) \vdash FF}$

now_later: $\mathsf{P \vdash \triangleright P}$.
later_K: $\mathsf{\triangleright (P \longrightarrow Q) \vdash (\triangleright P \longrightarrow \triangleright Q)}$.
later_allp: $\mathsf{\forall T\ (F{:}\ T \rightarrow mpred), \triangleright (ALL\ x{:}T,\ F\ x) = ALL\ x{:}T, \triangleright (F\ x)}$.
later_exp: $\mathsf{\forall T\ (F{:}\ T \rightarrow mpred), EX\ x{:}T, \triangleright (F\ x) \vdash \triangleright (EX\ x{:}\ F\ x)}$.
later_exp': $\mathsf{\forall T\ (any{:}T)\ F, \triangleright (EX\ x{:}\ F\ x) \;=\; EX\ x{:}T, \triangleright (F\ x)}$.
later_imp: $\mathsf{\triangleright (P \longrightarrow Q) \quad = \quad (\triangleright P \longrightarrow \triangleright Q)}$.
loeb: $\mathsf{\triangleright P \vdash P \rightarrow TT \vdash P}$.
later_sepcon: $\mathsf{\triangleright (P * Q) = \triangleright P * \triangleright Q}$.
later_wand: $\mathsf{\triangleright (P -\!\!* Q) = \triangleright P -\!\!* \triangleright Q}$.
later_ewand: $\mathsf{\triangleright (P -\!\circ Q) = (\triangleright P) -\!\circ (\triangleright Q)}$.

81 Proving larg(ish) programs

When your program is not all in one .c file, see also Chapter 82. Whether or not your program is all in one .c file, you can prove the individual function bodies in separate .v files. This uses less memory, and (on a multicore computer with parallel make) saves time. To do this, put your API spec (up to the construction of Gprog in one file; then each semax_body proof in a separate file that imports the API spec.

EXTRACTION OF SUBORDINATE SEMAX-GOALS. To ease memory pressure and recompilation time, it is often advisable to partition the proof of a function into several lemmas. Any proof state whose goal is a semax-term can be extracted as a stand-alone statement by invoking tactic *semax_subcommand* V G F. The three arguments are as in the statement of surrounding semax-body lemma, i.e. are of type *varspecs*, *funspecs*, and *function*.

The subordinate tactic *mkConciseDelta* V G F Δ can also be invoked individually, to concisely display the type context Δ as the application of a sequence of initializations to the host function's func_tycontext.

82 Separate compilation, semax_ext

This chapter is obsolete, as is the progs/evenodd example. There's a newer, better way of doing modular verification of modular programs: Verified Software Units (VSU).

What to do when your program is spread over multiple .c files. See progs/even.c and progs/odd.c for an example.

CompCert's clightgen tool translates your .c file into a .v file in which each C-language identifier is assigned a positive number in the AST (Abstract Syntax Tree) representation. When you have several .c files, you need consistent numbering of the identifiers in the .v files. One way to achieve this is to run clightgen on all the .c files at once:

```
clightgen even.c odd.c
```

This generates even.v and odd.v with consistent names. (It's not exactly separate compilation, but it will have to suffice for now.)

Now, you can do *modular verification of modular programs.* This is illustrated in,

progs/verif_evenodd_spec.v Specifications of the functions.
progs/verif_even.v Verification of even.c.
progs/verif_odd.v Verification of odd.c.

Linking of the final proofs is described by Stewart.[1].

[1] Gordon Stewart, *Verified Separate Compilation for C*, PhD Thesis, Department of Computer Science, Princeton University, April 2015

83 Concurrency

Verifiable C can now be used to verify concurrent programs. For more information and examples of how to use this feature, see the concurrency manual in VST/doc/concurrency.pdf.

84 Catalog of tactics / lemmas

Below is an alphabetic catalog of the major floyd tactics. In addition to short descriptions, the entries indicate whether a tactic (or tactic notation) is typically user-applied [u], primarily of internal use [i] or is expected to be used at development-time but unlikely to appear in a finished proof script [d]. We also mention major interdependencies between tactics, and their points of definition.

assert_PROP P (tactic; Chapter 44) Put the proposition P above the line, if it is provable from the current precondition.

cancel (tactic; page 65) Deletes identical spatial conjuncts from both sides of a base-level entailment.

data_at_conflict p (tactic) equivalent to field_at_conflict p nil.

deadvars! (tactic) Removes from the LOCAL block of the current precondition, any variables that are irrelevant to the rest of program execution. Fails if there is no such variable.

derives_refl (lemma) $A \vdash A$. Useful after cancel to handle $\beta\eta$-equality; see page 65.

derives_refl' (lemma) $A = B \;\rightarrow\; A \vdash B$.

drop_LOCAL n (tactic, where $n : nat$). Removes the nth entry of a the LOCAL block of a semax or ENTAIL precondition.

drop_LOCALs [_i; _j] Removes variables _i and _j from the LOCAL block of a semax or ENTAIL precondition.

entailer (tactic; page 67, page 31) Proves (lifted or base-level) entailments, possibly leaving a residue for the user to prove.

entailer! (tactic; page 67, page 31) Like entailer, but faster and more powerful—however, it sometimes turns a provable goal into an unprovable goal.

entailer!! (tactic; page 67, page 31) Like entailer!, but does not put hypotheses above the line derived from SEP and LOCAL clauses.

Exists v (tactic; Chapter 24) Instantiate an EX existential on the right-hand side of an entailment.

fastforward n (tactic; page 74) Do forward symbolic execution through n C statements.

fastforward! n (tactic; page 74) Like fastforward n, but more aggressive and slower.

field_at_conflict p *fld* (tactic) Solves an entailment of the form
$\ldots *$ field_at *sh* t *fld* v_1 p $* \ldots *$ field_at *sh* t *fld* v_2 p $* \ldots \vdash \_$
based on the contradiction that the same field-assertion cannot $*$-separate. Usually invoked automatically by entailer (or entailer!) to prove goals such as $!!(p{<>}q)$. Needs to be able to prove (or compute) the fact that $0 <$ sizeof (nested_field_type t *fld*);
for data_at_conflict that's equivalent to $0 <$ sizeof t.

finish (tactic; page 74) Attempts to solve a goal of any form.

finish! (tactic; page 74) Like finish, but more aggressive and slower.

forward (tactic; page 23) Do forward Hoare-logic proof through one C statement (assignment, break, continue, return).

forward_call ***ARGS*** (tactic; page 40) Forward Hoare-logic proof through one C function-call, where *ARGS* is a witness for the WITH clause of the funspec.

forward_for (tactic; page 88) Hoare-logic proof for the for statement, general case.

forward_for_simple_bound n ***Inv*** (tactic; page 86) When a for-loop has the form for $(init;\ i < hi;\ i{+}{+})$, where n is the *value* of *hi*, and *Inv* is the loop invariant.

forward_if Q (tactic; page 27) Hoare-logic proof for the **if** statement, where Q may be omitted if at the end of a block, where the postcondition is already given.

forward_while *Inv* (tactic; Chapter 14) Forward Hoare-logic proof of a while loop, with loop invariant *Inv*.

list_solve (tactic; Chapter 64) Solve goals that arise from lists with Zlength, concatentation, sublist, and Znth.

make_compspecs *prog* (tactic; page 14)

mk_varspecs *prog* (tactic; page 14

mkConciseDelta V G F Δ (tactic; page 125) Applicable to a proof state with a semax goal. Simplies the Δ component to the application of a sequence of initializations to the host function's func_tycontext. Used to prepare the current proof goal for abstracting/factoring out as a separate lemma.

rep_lia (tactic; page 107) Solves goals in linear integer arithmetic (like lia) enhanced by extra facts about 32-bit and 64-bit modular representations of integers (Int and Int64 modules).

semax_subcommand V G F (tactic) Applicable to a proof state with a semax goal. Extracts the current proof state as a stand-alone statement that can be copy-and pasted to a separate file. The three arguments should be copied from the statement of surrounding semax-body lemma: V : varspecs, G : funspecs, F : function.

start_function (tactic; Chapter 10) Unpack the funspec's pre- and postcondition into a Hoare triple describing the function body.

sublist_split (lemma; page 36) Break a sublist into the concatentation of two smaller sublists.

unfold_data_at (tactic; page 54) When t is a struct (or array) type, break apart data_at sh t v p into a separating conjunction of its individual fields (or array elements).

unfold_field_at (tactic; page 54) Like unfold_data_at, but starts with field_at sh t $path$ v p.

with_library (tactic; Chapter 73) Complete the funspecs by inserting stub specifications for all unspecified functions; and (if **Import** VST.floyd.library is done) adding standard specifications for malloc, free, exit.