Verifiable C

Applying the Verified Software Toolchain to C programs

Version 1.8
August 9, 2017

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*.

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.

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 to 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 8.6, from coq.inria.fr. Follow the standard installation instructions.

CompCert 2.7.2, from https://github.com/ildyria/CompCert/tree/v2.7.2. (This is an unofficial release, since no official release of CompCert 2.7 is ported to Coq8.6.) Build the *clightgen* tool, using these commands: ./configure ia32-linux; make clightgen. You might replace ia32-linux with ia32-macosx or ia32-cygwin. Verifiable C should work on other 32-bit architectures as well, but has not been extensively tested.

VST 1.8, 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).

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

- Write a C program F.c.
- Run clightgen 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, floyd.proofauto.

LOAD PATHS. Interactive development environments (CoqIDE or Proof General) will need their load paths properly initialized through command-line arguments. Running make in vst creates a file .loadpath with the right arguments. You can then do (for example),
coqide `cat .loadpath` progs/verif_reverse.v
See the heading USING PROOF GENERAL AND COQIDE *in the file* BUILD_ORGANIZATION *for more information.*

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

3 *Verifiable C programming*

See PLCC Chapter 22

Verifiable C is a *language* (subset of C) and a *program logic* (higher-order impredicative concurrent separation logic).

In writing Verifiable C programs you must:

- Make each memory dereference into a top level expression (PLCC page 143)
- Avoid casting between integers and pointers.
- Avoid goto and switch statements.
- * Avoid nesting function calls and assignments inside subexpressions.
- * Factor && and || operators into **if** statements (to capture short circuiting behavior).

The items marked * are accomplished automatically by CompCert's clightgen tool. That is, if you have function calls or assignments inside expressions, clightgen will factor the your program adding extra assignments to temporary variables.

There's a special treatment of malloc/free; see Chapter 57.

4 Clightgen and ASTs

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

```
#include <stddef.h>

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

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

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

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; the definitions _main and _s are abbreviations for these. The AST for sumarray is in the function-definition f_sumarray.

There you can see that sumarray's return type is is int. To represent the syntax of C type-expressions, CompCert defines,

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

and we abbreviate tint := Tint I32 Signed noattr.

5 Use the IDE

Chapter 6 through Chapter 18 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, read about load paths, at the heading USING PROOF GENERAL AND COQIDE in the file VST/BUILD_ORGANIZATION.

6 Functional spec, 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 specification
2. API specification
3. Function-body correctness proofs, one per file.

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 SPEC. 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 spec 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; omega.
Qed.
```

The data types used in a functional spec 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 spec is about mathematics, not about C programming.

THE APPLICATION PROGRAMMER INTERFACE 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.

We start verif_sumarray.v with some standard boilerplate:

```
Require Import floyd.proofauto.
Require Import 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 25 and Chapter 44.

After the boilerplate (and the functional spec), we have the function specifications for each function in the API spec:

```
Definition sumarray_spec :=
DECLARE _sumarray
 WITH a: val, sh : share, contents : list Z, size: Z
 PRE [ _a OF (tptr tint), _n OF tint ]
    PROP(readable_share sh;
          0 ≤ size ≤ Int.max_signed;
          Forall (fun x ⇒ Int.min_signed ≤ x ≤ Int.max_signed) contents)
    LOCAL(temp _a a; temp _n (Vint (Int.repr size)))
    SEP(data_at sh (tarray tint size) (map Vint (map Int.repr contents)) a)
  POST [ tint ]
    PROP()
    LOCAL(temp ret_temp (Vint (Int.repr (sum_Z contents))))
    SEP(data_at sh (tarray tint 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 9).

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."

An assertion in Verifiable C's *separation logic* can be written at either of two levels: The *lifted level*, implicitly quantifying over all local-variable states; or the *base level*, at a particular local-variable state. Program assertions are written at the lifted level, for which the notation is PROP(...) LOCAL(...) SEP(...).

In an assertion PROP($\vec{P}$) LOCAL($\vec{Q}$) SEP($\vec{R}$), the propositions in the sequence $\vec{P}$ are all of Coq type Prop. They describe things that are forever true, independent of program state. Of course, in the function precondition above, the statement 0 ≤ size ≤ Int.max_signed is "forever" true *just within the scope of the quantification of the variable* size; it is bound by WITH, and spans the PRE and POST assertions.

The LOCAL propositions $\vec{Q}$ are *variable bindings* of type localdef. Here, the function-parameters a and n are treated as nonaddressable local variables, or "temp" variables. The localdef (temp _a a) says that (in this program state) the contents of C local variable _a is the Coq value a. In general, a C scalar variable holds something of type 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 →int →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 integers*, with access-permission sh, and the contents of that array is the sequence map Vint contents.

THE POSTCONDITION is introduced by POST [tint], indicating that this function returns a value of type int. There are no PROP statements in the postcondition, because no forever-true facts exist in the world that weren't already true on entry to the function. (This is typical!) The LOCAL *must not mention* the function parameters, because they are destroyed on function exit; it will only mention the return-temporary ret_temp. 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; function-parameter _a contains value a and _n 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. The C language specification says that a C compiler *may* treat signed integer overflow by wrapping around mod 2^n, where n is the word size (e.g., 32). In practice, almost all C compilers (including CompCert) do this wraparound, and it is part of the CompCert C light operational semantics. See Chapter 22. The function Int.repr: Z → int truncates mathematical integers into 32-bit integers by taking the (sign-extended) low-order 32 bits. Int.signed: int → Z injects back into the signed integers.

The postcondition guarantees that the value returned is Int.repr (sum_Z contents). But what if $\sum s \geq 2^{31}$, so the sum doesn't fit in a 32-bit signed integer? Then

$\mathsf{Int.signed(Int.repr\ (sum\_Z\ contents))} \neq \mathsf{(sum\_Z\ contents)}$. In general, for a claim about $\mathsf{Int.repr}(x)$ to be *useful*, one also needs a claim that $0 \le x \le \mathsf{Int.max\_unsigned}$ or $\mathsf{Int.min\_signed} \le x \le \mathsf{Int.max\_signed}$. The caller of this function will probably need to prove $\mathsf{Int.min\_signed} \le \mathsf{sum\_Z\ contents} \le \mathsf{Int.max\_signed}$ in order to make much use of the post-condition.

What if s is the sequence $[\mathsf{Int.max\_signed;\ 5;\ 1\text{-}Int.max\_signed}]$? Then $\sum s = 6$. Does the program really work? Answer: Yes, by the miracle of modular arithmetic.

7 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 same order the functions appear in the program (in particular, the same order they appear in prog.(prog_defs), in sumarray.v). **??** 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).

8 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 ⇒ Int.min_signed ≤x ≤Int.max_signed) 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 tint 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) (e.g.,) unfold abbreviate in POSTCONDITION.

Below the line we have one proof goal: the Hoare triple of the function body. In this judgment $\Delta \vdash \{P\}\, c\, \{R\}$, written 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.

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;.

9 forward

We do Hoare logic proof by forward symbolic execution. On page 18 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. 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 tint size) (map Vint (map Int.repr contents)) a))
  (Ssequence (Sset _s (Econst_int (Int.repr 0) tint)) MORE_COMMANDS)
  POSTCONDITION
```

Notice that the precondition of this semax is really the *postcondition* 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, the expression E must not contain side effects or function calls. The variable x must be a nonaddressable local variable.

c_1; c_2 Sequencing of two 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). The expression &p→next does not actually load or store (it just computes an address) and is permitted.

x= *E; Memory load.

x= a[E]; Array load.

x= E→*fld*; Field load.

x= E→$f_1.f_2$; Nested field load.

x= E→$f_1[i].f_2$; Fields and subscripts ... When the right-hand side is equivalent to a single memory-load via some access *path* (struct-fields and array-subscripts) from pointer value p, the SEP component of the precondition must contain an appropriately typed item of the form data_at $\pi\ t\ v\ p$ such that the *path* from p in an object of type t leads to a field (or array slot) that can be loaded into _x. Or, field_at $\pi\ t\ path'\ v\ p'$, such that where $path'$ is a suffix of *path*, and p' is the address reached by starting at p and following the prefix. Share π must be a readable_share.

$E_1 = E_2$; Memory store. Expression E_2 must not dereference memory. Expression E_1 must be equivalent to a single memory store via some access *path* (as described above for loads), and there must be an appropriate storable data_at or field_at. Or E_1 may be an addressable local variable. Share π must be a writable_share.

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 25) 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(W), where W is a witness, a tuple corresponding (componentwise) to the WITH clause of the function specification. (If you do just forward, you'll get a message with advice about the *type* of W.)

This results a proof goal to show that the precondition implies the function precondition and includes an uninstantiated variable: The Frame represents the part of the spacial precondition that is unchanged by the function call. It will generally be instantiated by a call to cancel.

10 If, While, For, call

To do forward proof through if-statements, while-loops, and for-loops, you need to provide additional information: join-postconditions, loop invariants, etc. If the first statement (not counting Ssequence) in the current proof goal is Sifthenelse, Swhile, Sfor, Sloop, Scall, or you can simply apply the forward tactic: it will fail, with an error message explaining what you need to provide; that is, it suggests the appropriate choice from this menu:

- **if** e **then** s_1 **else** s_2; $s_3\ldots$
  you do, forward_**if** Q, where Q is the precondition of statement s_3. We call it the *join postcondition*, as it's the assertion that goes where s_1 and s_2 join back together.
- **if** e **then** s_1 **else** s_2;}... When the if-statement appears at the end of a basic 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 do unfold abbreviate **in** POSTCONDITION to see what's there.
- **while**$(e)s;\ldots$ (no break statements in s)
  You write forward_while Q, where Q is a loop invariant. See Chapter 11.
- **while**$(e)s;\ldots$ (with break statements in s)
  You must treat this as if it were **for**$(;e;)$ s, and use the forward_for tactic; see below.
- **for**$(e_1;i<e_2;i++)$ s (no break statements in s)
  If e_2 is loop-invariant, you may be able to use forward_for_simple_bound; see Chapter 45.
- **for**$(e_1;e_2;e_3)$ s (no break statements in s)
  Use forward_for Q Q', where Q is the loop invariant and Q' is the assertion that goes right before the *increment* command, e_3. See Chapter 46.
- **for**$(e_1;e_2;e_3)$ $s;\ldots$
  If s contains break statements *and* there are more commands after

the loop, you will need to write forward_for Q Q' R, where Q and Q' are as above, and R is the join-postcondition. See Chapter 46.

- $f(e_1,\ldots,e_n)$
  Use forward_call; see 17.

11 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 ⇒ Int.min_signed ≤x ≤Int.max_signed) 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 tint 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 the EX binds that value. For example, the invariant for this loop is,

```
Definition sumarray_Inv a0 sh contents size :=
EX i: Z,
 PROP(0 ≤i ≤size)
 LOCAL(temp _a a0; 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 tint size) (map Vint (map Int.repr contents)) a0).
```

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 proven:

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 *not* loop condition) is a good precondition for the proof of 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 tint 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 tint size) (map Vint (map Int.repr contents)) a)
```

This is an *entailment* goal; Chapter 12 shows how to prove such goals.

12 Entailments

An *entailment* in separation logic, $P \vdash Q$, says that any state satisfying P must also satisfy Q. What's in a state? Local-variable environment, heap (addressable memory), even the state of the outside world. VST's type mpred, *memory predicate*, can be thought of as mem→Prop (but is not quite the same, for quite technical semantic reasons). That is, an mpred is a test on the heap only, and cannot "see" the local variables (tempvars) of the C program.

Type environ is a local/global to the *values* of tempvars (nonaddressable locals) and to the *addresses* of globals and addressable locals. A *lifted predicate* of type environ→mpred can "see" both the heap and the local/global variables. The Pre/Post arguments of Hoare triples (semax Δ Pre c Post) are lifted predicates.

At present, Verifiable C has a notion of external-world state, in the Espec: OracleKind, but it is not well developed; enhancements will be needed for reasoning about input/output.

Our language for lifted predicates uses $\mathsf{PROP}(\vec{P})\mathsf{LOCAL}(\vec{Q})\mathsf{SEP}(\vec{R})$, where $\vec{R}$ is a list of mpreds. Our language for mpreds uses primitives such as data_at and emp, along with connectives such as the $*$ and $-\!\!*$ of separation logic. In both languages there is an EX operator for existential quantification.

Separation logic's rule of consequence is shown here

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

at left in traditional notation, and at right as in Verifiable C. The type-context Δ constrains values of locals and globals. Using this axiom, called semax_pre_post on a proof goal semax Δ P c Q yields three subgoals: another semax and two (lifted) entailments, $\Delta, P \vdash P'$ and $\Delta, Q \vdash Q'$.

The standard form of a lifted entailment is ENTAIL Δ, PQR $\vdash$PQR′, where PQR and PQR′ 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 tint 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 tint size) (map Vint (map Int.repr contents)) a)
```

We instantiate the existential with the only value that works here, zero: **Exists** 0. Chapter 21 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 37). In this case, the goal solves entirely automatically. In particular, $0 \leq i \leq$ size solves by omega; 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 entailment guarantees that e executes without crashing: 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. That's because the typechecker is *calculational*, as Chapter 25 of *Program Logics for Certified Compilers* explains.

13 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 of these leaves a subgoal—proving that the subscript i is in range. Let's examine the proof goal:

```
SH : readable_share sh
H : 0 ≤size ≤Int.max_signed
H0 : Forall (fun x : Z ⇒ Int.min_signed ≤x ≤Int.max_signed) 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 tint size) (map Vint (map Int.repr contents)) a))
  (Ssequence
     (Sset _x
        (Ederef
           (Ebinop Oadd (Etempvar _a (tptr tint)) (Etempvar _i tint)
              (tptr tint)) tint)) 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.

The command x=a[i]; is a *load* from data-struture a. For this to succeed, there must be a data_at (or field_at) assertion about a in the SEP clauses of the precondition; the permission share in that data_at must grant read access; and the subscript must be in range. Indeed, the data_at is there, and the share is taken care of automatically by the hypothesis SH above the line.

If we were to try forward right now, it would fail with a message, “Please make sure omega or auto can prove ($0 \leq i <$ Zlength *contents*).” That’s what would ensure that the subscript i is within the bounds of the array.

Therefore, we write,

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

The reason this is so easily provable is that data_at sh (tarray τ n) σ a holds only when Zlength(σ) $= n$.

Now that we have the assertion H_2: Zlength contents = size above the line, forward succeeds on the array subscript.

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

14 Splitting sublists

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 ⇒ Int.min_signed ≤ x ≤ Int.max_signed) 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)) Vundef)));
temp _x (Znth i (map Vint (map Int.repr contents)) Vundef); temp _a a;
temp _n (Vint (Int.repr size)))
SEP(data_at sh (tarray tint 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 tint 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 0))
```

Applying f_equal twice, leaves the goal,

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

Now the lemma sublist_split: $\forall l\, m\, h\, \mathsf{al},\quad 0 \le l \le m \le h \le |\mathsf{al}| \to$ sublist l h al $=$ sublist l m al ++ sublist m h al is helpful here: rewrite (sublist_split 0 i (i+1)) by omega. A bit more rewriting with the theory of sum_Z and sublist finishes the proof.

15 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).

After doing simpl to do some C-expression-evaluation, we have

```
H4 : Forall (value_fits tint) (map Vint (map Int.repr contents))
H2 : field_compatible (Tarray tint (Zlength ...) noattr) [] a
   (other above-the-line hypotheses elided)
_____________________________________________________
data_at sh (tarray tint (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 33).

This entailment's right-hand side has no spatial predicates. That's because 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.

In a situation like this—where saturate_local has already been done *and* the r.h.s. of the entailment is purely nonspatial— *almost always* there's no more useful information in the left hand side that hasn't already been extracted by saturate_local. We can throw away the l.h.s. with apply prop_right (or by entailer! but that's a bit slower).

The remaining subgoal solves easily in the theory of sublists. The proof of the function sumarray is now complete.

16 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 main always looks the same:

```
Definition main_spec :=
 DECLARE _main WITH u : unit
       PRE [] main_pre prog u
       POST [ tint ] main_post prog u.
```

main_pre calculates the precondition automatically from (the list of extern global variables and initializers of) the program. Then, when we prove that main satisfies its funspec,

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

the start_function tactic "unpacks" main_pre into an assertion:

```
four : val
------------------------------------(1/1)
semax Delta
 (PROP () LOCAL(gvar _four four)
  SEP(data_at Ews (tarray tint 4)
     (map Vint [Int.repr 1; Int.repr 2; Int.repr 3; Int.repr 4]) four))
 (... function body...)
 POSTCONDITION
```

The LOCAL clause means that the C global variable _four is at memory address *four*. (If we had omitted the name tactic in the proof script above, then start_functon would have chosen some other name for this value.) See Chapter 31.

The SEP clause means that there's data of type "array of 4 integers" at address *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 40.

17 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 13) 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 (four,Ews,four_contents,4).

forward_call (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.

The postcondition of the call statement (which is the precondition of the next return statement) has an existential, EX vret:val. This comes directly from the existential in the funspec's postcondition. To move vret above the line, simply **Intros** vret.

Finally, we are at the return statement. The forward tactic is easily able to prove that the current assertion implies the postcondition of _main, because main_post is basically an abbreviation for True.

18 *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 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 all_funcs_correct: semax_func Vprog Gprog (prog_funct prog) Gprog.
Proof.
unfold Gprog, prog, prog_funct; simpl.
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 are listed in Gprog and the same order as they appear in prog.(prog_defs).

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

We have discussed the *lifted* separation logic, the language of *assertions* PROP(...) LOCAL(...) SEP(...) whose LOCAL clause can see local variables, and whose SEP clause can see the heap. Often we want to hold all local variables constant and reason only about the heap; for this we use the *base level* separation logic.

Like most separation logics, it is built from predicates on "heaplets". The grammar of base-level separation-logic expressions is,

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 (rare)
	$R_1 \| R_2$	disjunction
	wand $R\ R'$	magic wand $R \mathrel{-\!\!*} R'$ (rare)
	...	other operators, including user definitions

20 PROP( ) LOCAL( ) SEP( )

The *lifted* separation logic can "see" local and global variables of the C program, in addition to the contents of the heap (pointer dereferences) that the base level separation logic can see. The *canonical form* of a lifted assertion is $\mathsf{PROP}(\vec{P})\mathsf{LOCAL}(\vec{Q})\mathsf{SEP}(\vec{R})$, 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.

Lifted assertions can occur in other forms than canonical form; in fact, anything of type environ→mpred is a lifted assertion. But canonical form is most convenient for forward symbolic execution (Hoare-logic rules).

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

Entailments in canonical form are normally of the form, $\mathsf{ENTAIL}\ \Delta,\ PQR \vdash PQR'$, where PQR is a lifted assertion in canonical form, PQR' is a lifted assertion not necessarily in canonical form, and Δ is a type context. The $\vdash$ operator is written |-- in Coq.

This notation is equivalent to $(\mathsf{tc\_environ}\ \Delta\ \&\&\ PQR) \vdash PQR'$. That is, Δ just provides extra assertions on the left-hand side of the entailment.

21 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. This assertion has both:

ENTAIL Δ,
    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. 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.

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

This entailment proves straightforwardly by entailer!.

22 *Integers:* nat, Z, int (compcert/lib/Integers.v)

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 49 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 integers, the recommended proof technique is to make sure your integers never 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

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
...
------------------------
Int.min_signed ≤ 0 ≤ n
```

you can use the repable_signed tactic, which is basically just omega 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 $0 \le i < n$; $n \le$ 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 array size and index (variables size and i) are kept within bounds; but the *contents* of the array might overflow when added up, which is why add_elem uses Int.add instead of Z.add.

23 *Values:* Vint, Vptr *(compcert/common/Values.v)*

Definition block : Type := positive.

Inductive val: Type :=
  | Vundef: val
  | Vint: int → val
  | Vlong: int64 → val
  | Vfloat: float → val
  | Vsingle: float32 → val
  | Vptr: block → int → 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 (Vptr $b\,z$) + (Vint i) = Vptr $b\,(z+i)$. Of course, the C-language + operator first multiplies i by the size of the array-element that 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 28). 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."

24 C types

(compcert/cfrontend/Ctypes.v)

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.
```

25 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 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 63.

26 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.

27 *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)

28 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, gvar _L L)
SEP(data_at Ews t_list (Vint (Int.repr 0), nullval) 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 L. There is a struct list at 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 $L \mapsto (0, \text{NULL})$ to mean data_at Tsh t_list (Vint (Int.repr 0), nullval) 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

29 reptype', repinj

```
struct a {double x1; int x2;};            TL;DR
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, p : val
  PRE []
    PROP() LOCAL(gvar _p p)
    SEP(data_at Ews t_struct_b (repinj _ v) p)
  POST [ tint ]
    PROP() LOCAL(temp ret_temp (Vint (snd (snd v))))
    SEP(data_at Ews t_struct_b (repinj _ v) 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, p : val
  PRE []
    PROP() LOCAL(gvar _p p)
    SEP(data_at Ews t_struct_b (Vint (Int.repr i), (Vfloat x, Vint j)) p)
  POST [ tint ]
    PROP() LOCAL(temp ret_temp (Vint j))
    SEP(data_at Ews t_struct_b (Vint (Int.repr i), (Vfloat x, Vint j)) p).
```

30 field_at

Consider again the example in progs/nest2.c

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

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(gvar _pb pb)
SEP(data_at *sh* t_struct_b (y1,(x1,x2)) pb)

The postcondition (after the load) would include the new LOCALfact, temp _i x2.

The tactic (unfold_data_at 1%nat) changes the SEP part of the assertion as follows:

```
SEP(field_at Ews t_struct_b (DOT _y1) (Vint y1) pb;
    field_at Ews t_struct_b (DOT _y2) (Vfloat x1, Vint x2) pb)
```

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

```
SEP(field_at Ews t_struct_b (DOT _y1) (Vint y1) pb;
    field_at Ews t_struct_b (DOT _y2 DOT _x1) (Vfloat x1) pb;
    field_at Ews t_struct_b (DOT _y2 DOT _x2) (Vint x2) 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*.

31 *Localdefs:* temp, lvar, gvar

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
| gvar: ident → val → localdef
| sgvar: ident → val → localdef
| localprop: Prop → 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.
gvar i v binds a *visible global* variable i to its *address* v.
sgvar i v binds a *possibly shadowed global* variable i to its *address* v.

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 write

PROP() LOCAL(temp _p q; lvar _g tint q; sgvar _g p}
SEP(data_at Ews tint (Vint (Int.repr 2)) p; 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.

32 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 $\vec{Q}$ be (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.

33 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:
$\quad \mathsf{field\_at}\ \pi\ t\ path\ v\ p \vdash !!(\mathsf{field\_compatible}\ t\ path\ p$
$\qquad\qquad \wedge\ \mathsf{value\_fits}\ (\mathsf{nested\_field\_type}\ t\ path)\ v)$
$\quad \mathsf{data\_at}\ \pi\ t\ v\ p \vdash !!(\mathsf{field\_compatible}\ t\ \mathsf{nil}\ p \wedge \mathsf{value\_fits}\ t\ v)$
memory_block_local_facts:
$\quad \mathsf{memory\_block}\ \pi\ n\ p \vdash !!\ \mathsf{isptr}\ p$

The assertion $(\mathsf{Zlength}\ v = n)$ is actually a consequence of value_fits when t is an array type. See Chapter 35.

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.

34 *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.

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

data_at π t_struct_b $(i,(f,j))$ p $\vdash$
  data_at π tint i p $*$ data_at π t_struct_a (f,j) (offset_val p 12)

but the converse is not valid:

data_at π tint i p $*$ data_at π t_struct_a (f,j) (offset_val p 12)
  $\nvdash$ data_at π t_struct_b $(i,(f,j))$ p

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, a pure proposition, 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 33.

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 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 properly aligned, has not crossed the end-of-memory boundary). And therefore:

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.

35 *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) (v: val) :=
  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 ≤ Int.unsigned c ≤ Byte.max_unsigned
TC2: Byte.min_signed ≤ Int.signed c ≤ Byte.max_signed

36 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

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

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 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 **??** and PLCC Chapter 19.

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

37 entailer!

The 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 32).

Then: saturate_local (see Chapter 33).

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.

38 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.

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

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

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

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

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

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

$$P * (!!Q \,\&\&\, R) = !!Q \,\&\&\, (P * R) \qquad (!!Q \,\&\&\, P) * R = !!Q \,\&\&\, (P * R)$$

Delete auto-provable propositions.

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

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.

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

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

$$`(P * Q) =\ `P\ *\ `Q \qquad `(P\ \&\&\ Q) =\ `P\ \&\&\ `Q$$

Type checking and miscellaneous.

$$\mathsf{tc\_andp}\ \mathsf{tc\_TT}\ e\ =\ e \qquad\qquad \mathsf{tc\_andp}\ e\ \mathsf{tc\_TT}\ =\ e$$

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

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

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

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

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

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

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

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

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

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

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

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

Function return values.

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

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

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

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

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.

39 Welltypedness of variables

The 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,

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

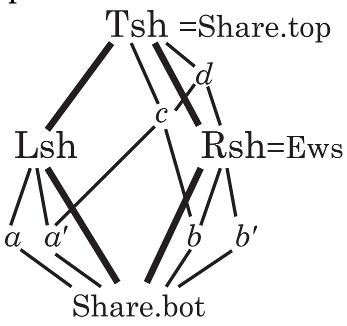
Δ keeps track of which nonaddressable local variables are initialized; says that all references to local variables contain values of the right type; and says that all addressable locals and globals point to an appropriate block of memory.

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 59, this tc_val may be simplified into an is_int hypothesis, or even removed if vacuous.

40 Shares

(See PLCC Chapters 11,41)

The mapsto operator (and related operators) take a *permission share*, expressing whether the mapsto 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	¬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 (mapsto *sh* t v w) holds, then the share *sh* must include at least a read share, thus this give 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., $\mathsf{Rsh} \subset sh$, we can say writable_share *sh* : Prop.

Memory obtained from malloc comes with the top share Tsh. Writable extern global variables and stack-allocated addressable locals (which of course must not be deallocated) come with the "extern writable share" Ews which is equal to Rsh. Read-only globals come with a half-share of Rsh.

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 13); 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 70.

41 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 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.

42 Proof of the reverse program

Program Logics for Certified Compilers, Chapter 3 describes the notion of *list segments* and their application to a proof of the list-reverse function. (Chapters 2 and 3 available free here; the whole e-book available cheap here or here; or buy the hardcover.)

In this chapter we will demonstrate this proof in Verifiable C, on the C program in progs/reverse.c. Please open your CoqIDE or Proof General to progs/verif_reverse.v.

```
/* reverse.c */
#include <stddef.h>

struct list {int head; struct list *tail;};

struct list three[] = { {1, three+1}, {2, three+2}, {3, NULL} };

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;
}

int main (void) {
  struct list *r; int s;
  r = reverse(three);   s = sumlist(r);   return s;
}
```

As usual, in progs/verif_reverse.v we import the clightgen-produced file reverse.v and build CompSpecs and Vprog (see page 13, Chapter 25, Chapter 44).

For the struct list used in *this* program, struct list {int head; struct list *tail;}; we can define the notion of *list segment* $x \overset{\sigma}{\leadsto} z$ with a recursive definition:

```
Fixpoint lseg (sh: share)
            (contents: list val) (x z: val) : mpred :=
 match contents with
 | h::hs ⇒ !! (x<>z) &&
              EX y:val, data_at sh (Tstruct _list noattr) (h,y) x
                   * lseg sh hs y z
 | nil ⇒ !! (ptr_eq x z) && emp
 end.
```

But instead, we make a general theory of 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 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 26). In this example, with one data field, the type of σ computes to list val.

We'll skip over the sumlist function and its verification.

The API spec (see also Chapter 6) for reverse is,

Definition reverse_spec :=
DECLARE _reverse
 WITH sh: share, $contents$: list val, p: val
 PRE [_p OF (tptr t_struct_list)]
  PROP(writable_share sh)
  LOCAL(temp _p p)
  SEP(lseg LS sh $contents$ p nullval)
 POST [(tptr t_struct_list)]
  EX p:val,
   PROP() LOCAL(temp ret_temp p)
   SEP(lseg LS sh (rev $contents$) p nullval).

The precondition says (for p the function parameter) $p \overset{\sigma}{\leadsto}$ nil, and the postcondition says that (for p the return value) $p \overset{\text{rev}\ \sigma}{\leadsto}$ nil. This is basically the specification given in PLCC Chapter 3, page 20.

Also, the list must have write permission (writable_share sh), because the list-reverse is an in-place destructive update.

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(lseg LS sh $contents$ p nullval).

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:

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

It's quite typical for loop invariants to existentially quantify over the values that are different iteration-to-iteration.

Definition reverse_Inv (sh: share) ($contents$: list val) : environ→mpred :=
EX cts_1: list val, EX cts_2 : list val, EX w: val, EX v: val,
PROP($contents$ = rev cts_1 ++ cts_2)
LOCAL(temp _w w; temp _v v)
SEP(lseg LS sh cts_1 w nullval; lseg LS sh cts_2 v nullval).

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 cts_1 with nil and cts_2 with $contents$; 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(lseg LS sh $contents$ p nullval)
⊢PROP($contents$ = rev [] ++ $contents$) LOCAL(temp _w nullval; temp _v p)
SEP(lseg LS sh [] nullval nullval;
lseg LS sh $contents$ p nullval)

The PROP and LOCAL parts are trivially solvable by the entailer. We can remove the SEP conjunct (lseg LS sh [] nullval nullval) by rewriting in the theory of list segments:

Lemma lseg_eq: ∀(LS : listspec _list _tail) (π : share) (l : list _) (v : val),
is_pointer_or_null v →
lseg LS π l v v = !!(l = []) && emp.

(2) The type-checking condition is not trivial, as it is a pointer comparison (see Chapter 41), 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: $contents$ = rev cts_1 ++ cts_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, lseg LS sh cts_2 v nullval. A list-segment isn't necessarily loadable, i.e., we cannot necessarily fetch v→tail; what we need to unfold the lseg, using this lemma:

Lemma lseg_nonnull: ∀(LS : listspec _list _tail) (π : share) (l : list _) v,
  typed_true (tptr t_struct_list) v →
  lseg LS π l v nullval =
    EX h:_, EX r:_, EX y:val,
      !!(l=h::r ∧ is_pointer_or_null y) &&
      list_cell LS π h x ∗
      field_at π t_struct_list (SUB _tail) y x ∗
      lseg LS π r y z.

That is, if $v \neq$ nullval, then the list-segment $v \overset{\sigma}{\leadsto}$ nullval is not empty: there exists a record $x \mapsto (h, y)$ and a residual list $y \overset{\sigma'}{\leadsto}$ nullval. Actually, here it is more convenient to use a corollary of this lemma, semax_lseg_nonnull, that is adapted to unfolding *the first lseg in the SEP clause of a semax precondition*. The typed_true premise solves easily by entailer!.

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::cts1,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 cts_1 ++ h :: r = (rev cts_1 ++ [h]) ++ r
---(2/2)
  list_cell LS sh h v ∗ field_at sh t_struct_list (DOT _tail) w v
    ∗ lseg LS sh cts_1 w nullval
⊢lseg LS sh (h :: cts_1) v nullval

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, which we do with lseg_unroll. 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.

43 list_cell, assert_PROP

In progs/verif_reverse.v, in the **Lemma** body_sumlist, move to the comment (* Prove that loop body preserves invariant *), and then three or four lines to just before assert_PROP.

This proof state is very similar to the one in the loop body of the body_reverse lemma (page 75):

```
contents, cts1, cts2 : list int;   p, t, y : val;   i : int
SH : readable_share sh
HRE : isptr t
H : contents = cts1 ++ i :: cts2
H1 : is_pointer_or_null y
________________________________________
semax Delta
  (PROP () LOCAL(temp _t t; temp _s (Vint (sum_int cts1)))
   SEP(list_cell LS sh (Vint i) t;
      field_at sh list_struct [StructField _tail] y t;
      lseg LS sh (map Vint cts2) y nullval; lseg LS sh (map Vint cts1) p t))
  h = t → head; ...
  POSTCONDITION
```

Here, the operator list_cell (from the general theory of list segments) describes "all the fields but the link." In our particular LS there is exactly one data field, which fact we state as a lemma:

```
Lemma list_cell_eq: ∀sh i p ,
  sepalg.nonidentity sh →
  field_compatible t_struct_list [] p →
  list_cell LS sh (Vint i) p =
  field_at sh t_struct_list (DOT _head) (Vint i) p.
```

To rewrite by list_cell_eq, we need to get a field_compatible fact above the line. Such facts are promiscuously introduced by saturate_local as part of calling entailer!, but we are not currently proving an entailment. No matter; we can prove one artificially:

assert_PROP (field_compatible t_struct_list nil t) as FC by entailer!.

The assert_prop tactic creates an ENTAIL proof goal with *the current semax precondition* on the left, and the named proposition on the right. That proposition is then put *above the line*; really this is a use of the rule of consequence. It's an easy way to get this field_compatible fact above the line.

44 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 13 (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 35).

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.

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

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 *id*. 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 $i < hi$ 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 11) that uses a for loop instead of a while loop:

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

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

46 For loops (general case)

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

This is demonstrated in the lemma body_sumarray_alt in the file progs/verif_sumarray2.v.

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

The *Inv* and *PreInc* should have type $A \to$ environ $\to$ mpred, where A is the type of some iteration-dependent quantity (in this example, Z, to hold the value of loop iteration variable i), and environ $\to$ mpred is the usual type of assertions.

47 *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. page 75 describes a situation where a list segment had to be unfolded to expose such a SEP conjunct.

Faced with the proof goal, semax Δ (PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP($\vec{R}$)) c *Post* where PROP($\vec{P}$)LOCAL($\vec{Q}$)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 83) that modify the precondition.
- Do rewriting in the precondition, either directly by the standard rewrite and change tactics, or by normalize (page 63).
- Extract propositions and existentials from the precondition, by using **Intros** (page 41) or normalize.
- Flatten stars into semicolons, in the SEP clause, by **Intros**.
- Use the freezer (page 104) 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 48.

48 The Frame rule

Separation Logic supports the Frame rule,

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

To use this in a forward proof, 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_2 for the duration of $c_1;c_2$, and have it back again for c_3. First you rewrite by seq_assoc to yield the goal

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

Then eapply semax_seq' to peel off the first command $(c_1;c_2)$ in the new sequence:

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

semax Δ' ?88 c_3 *Post*

Then 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_2 is added back in.

49 32-bit Integers

(compcert/lib/Integers.v)

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: $\forall$x y, mul x y = mul y x.
Theorem mul_zero: $\forall$x, mul x zero = zero.
Theorem mul_one: $\forall$x, mul x one = x.
Theorem mul_assoc: $\forall$x y z, mul (mul x y) z = mul x (mul y z).
Theorem mul_add_distr_l: $\forall$x y z, mul (add x y) z = add (mul x z) (mul y z).
Theorem mul_signed: $\forall$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.

50 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.
```

51 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 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 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.

52 Splitting arrays

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

data_at sh (tarray tint 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(int 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) bl a. 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 tint $(k - i)$) (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 34)

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.

53 sublist

Chapter 52 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.

These rules comprise the sublist *rewrite database*:

sublist_nil': $i = j \to$ 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 $\to$
 sublist $i\ j\ l$ ++ sublist $j'\ k\ l$ = sublist $i\ k\ l$.
subsub1: $a - (a - b) = b$.
Znth_list_repeat_inrange: $0 \le i \le n \to$ Znth i (list_repeat (Z.to_nat n) a) d = 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: $0 \le i \le j \le$ Zlengthal $\to$ 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') d = Znth $i\ l\ d$.
app_Znth2: $i \ge$ Zlength l $\to$ Znth i (l ++ l') d = Znth $i -$ Zlength l l' d.
Znth_sublist: $0 \le i \to 0 \le j < k - i \to$ Znth j (sublist $i\ k\ l$) d = Znth $(j+i)$ l d.

along with miscellaneous Z arithmetic:

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

Therefore, autorewrite **with** sublist is a good way to simplify expressions involving sublist, ++, map, Zlength, Znth, and list_repeat.

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.

54 Later

(See PLCC Chapter 15)

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.

55 *Mapsto and func_ptr* *(see PLCC section 24)*

Aside from the standard operators and axioms of separation logic, the core separation logic has just two primitive spatial (memory) 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 **??**.

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.

56 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 a 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* $[s_1;s_2;\ldots;s_n]$ augments your explicit funspec-list $[s_1;s_2;\ldots;s_n]$ 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** 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.

57 *malloc/free*

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

```
DECLARE _malloc
   WITH n:Z
   PRE [ 1%positive OF tuint ]
       PROP(0 ≤ n ≤ Int.max_unsigned)
       LOCAL(temp 1%positive (Vint (Int.repr n)))
       SEP()
    POST [ tptr tvoid ] EX p:_,
       PROP()
       LOCAL(temp ret_temp p)
       SEP(if eq_dec p nullval then emp
            else (malloc_token Tsh n p * memory_block Tsh n p)).

DECLARE _free
   WITH p:val, n:Z
   PRE [ 1%positive OF tptr tvoid ]
       PROP()
       LOCAL(temp 1%positive p)
       SEP(malloc_token Tsh n p; memory_block Tsh n p)
    POST [ Tvoid ]
       PROP()
       LOCAL()
       SEP().
```

You must **Import** floyd.library. Then these funspecs are made available in your Gprog by the use of the with_library tactic (Chapter 56).

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.

58 *exit*

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

```
DECLARE _exit
 WITH u: unit
 PRE [1%positive OF tint]
   PROP() LOCAL() SEP()
 POST [ tvoid ]
   PROP(False) LOCAL() SEP().
```

59 *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 LOCAL(gvar _myfunc p) in the precondition. That gvar is needed by the tactic make_func_ptr _myfunc, which adds func_ptr' myspec p 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.

Note: Unfortunately, in order to get the gvar _myfunc into the precondition of main, there must be some initialized global variable that refers to myfunc. That's the purpose of the (otherwise useless) array a in this program. And suppose you wanted to do make_func_ptr in some function *other* than main. Then you'd need to add this gvar to the LOCAL clause of that function's precondition, and pass it down from main. Both of these infelicities ought to be remedied in a future release.

60 Axioms of separation logic *(see PLCC Chapter 12)*

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

pred_ext: $P \vdash Q \rightarrow Q \vdash P \rightarrow P{=}Q$.
derives_refl: $P \vdash P$.
derives_trans: $P \vdash Q \rightarrow Q \vdash R \rightarrow P \vdash R$.
andp_right: $X \vdash P \rightarrow X \vdash Q \rightarrow X \vdash (P\&\&Q)$.
andp_left1: $P \vdash R \rightarrow P\&\&Q \vdash R$.
andp_left2: $Q \vdash R \rightarrow P\&\&Q \vdash R$.
orp_left: $P \vdash R \rightarrow Q \vdash R \rightarrow P||Q \vdash R$.
orp_right1: $P \vdash Q \rightarrow P \vdash Q||R$.
orp_right2: $P \vdash R \rightarrow P \vdash Q||R$.
exp_right: $\forall$ {B: Type}(x:B)(P:mpred)(Q: B $\rightarrow$ mpred),
    $P \vdash Q\ x \rightarrow P \vdash$ EX x:B, Q.
exp_left: $\forall$ {B: Type}(P:B $\rightarrow$ mpred)(Q:mpred),
    ($\forall$ x, P x $\vdash$ Q) $\rightarrow$ EX x:B,P $\vdash$ Q.
allp_left: $\forall$ {B}(P: B $\rightarrow$ mpred) x Q, P x $\vdash$ Q $\rightarrow$ ALL x:B,P $\vdash$ Q.
allp_right: $\forall$ {B}(P: mpred)(Q:B $\rightarrow$ mpred),
    ($\forall$ v, P $\vdash$ Q v) $\rightarrow$ P $\vdash$ ALL x:B,Q.
prop_left: $\forall$ (P: Prop) Q, (P $\rightarrow$ (TT $\vdash$ Q)) $\rightarrow$!!P $\vdash$ Q.
prop_right: $\forall$ (P: Prop) Q, P $\rightarrow$ (Q $\vdash$!!P).
not_prop_right: $\forall$ (P:mpred)(Q:Prop), (Q $\rightarrow$ (P $\vdash$ FF)) $\rightarrow$ P $\vdash$!!(~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 $\rightarrow$ Q $\vdash$ R) $\rightarrow$!!P && Q $\vdash$ R.
sepcon_derives: $P \vdash P' \rightarrow Q \vdash Q' \rightarrow P*Q \vdash P'*Q'$.

61 Obscure higher-order axioms

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

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

62 Proving larg(ish) programs

When your program is not all in one .c file, see also Chapter 63. 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.

THE FREEZER. A distinguishing feature of separation logic is the frame rule, i.e. the ability to modularly verify a statement w.r.t. its minimal resource footprint. Unfortunately, being phrased in terms of the syntatic program structure, the standard frame rule does not easily interact with forward symbolic execution as implemented by the Floyd tactics (and many other systems), as these continuously rearrange the associativity of statement sqeuencing to peel off the redex of the next *forward*, and (purposely) hide the program continuation as the abbreviation *MORE_COMMANDS*.

Resolving this conflict, Floyd's *freezer* abstraction provides a means for flexible framing, by implementing a veil that opaquely hides selected items of a SEP clause from non-symbolic treatment by non-freezer tactics.

The freezer abstraction consists of two main tactics, *freeze* N F and *thaw* F, where $N : \mathit{list\ nat}$ and F is a user-supplied (fresh) Coq name. The result of applying *freeze* $[i_1;\ldots;i_n]$ F to a semax goal is to remove items $i_1,\ldots,i_n$ from the precondition's SEP clause, inserting the item *FRZL F* at the head of the SEP list, and adding a hypothesis $F := \mathit{abbreviate}$ to Coq's proof context.

The term *FRZL F* participates symbolically in all non-freezer tactics just like any other SEP item, so can in particular be canceled, and included in a function call's frame. Unfolding a freezer is not tied to the associativity structure of program statements but can be achieved by invoking *thaw* F, which simply replaces *FRZL F* by the the list of F's constiuents. As multiple freezers can coexists and freezers can be arbitrarily nested, SEP-clauses R effectively contain forests of freezers, each constituent being thawable independently and freezer-level by freezer-level.

Wrapping single *forward* or *forward_call* commands in a freezer often speeds up the processing time noticably, as invocations of subordinate tactics *entailer*, *cancel*, etc. are supplied with smaller and more symbolic proof goals. In our experience, applying the freezer throughout the proof of an entire function body typically yields a speedup of about 30% on average with improvements of up to 55% in some cases, while also easing the memory pressure and freeing up valuable real estate on the user's screen.

A more invasive implementation of a freezer-like abstraction would refine the $\mathsf{PROP}(P)\ \mathsf{LOCAL}(Q)\ \mathsf{SEP}(R)$ structure to terms of the form $\mathsf{PROP}(P)\ \mathsf{LOCAL}(Q)\ \mathsf{SEP}(R)\ \mathsf{FR}(H)$ where $H : \mathit{list\ mpred}$. Again, terms in H would be treated opaquely by all tactics, and freezing/thawing would correspond to transfer rules between R and H. In either case, forward symbolic execution is reconciled with the frame rule, and the use of the mechanism is sound engineering practice as documentation of programmer's insight is combined with performance improvements.

63 *Separate compilation,* semax_ext

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

64 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.

cancel (tactic; page 61) 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 61.

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 62, page 27) Proves (lifted or base-level) entailments, possibly leaving a residue for the user to prove. The more aggressive entailer! should usually be used, but it sometimes turns a provable goal into an unprovable goal.

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.

forward (tactic; page 20) Do forward Hoare-logic proof through one C statement (assignment, break, continue, return).

forward_call ***ARGS*** (tactic; page 22) 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 81) Hoare-logic proof for the for statement, general case.

forward_for_simple_bound *n* ***Inv*** (tactic, page 80) 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_seq (tactic)

mkConciseDelta $V\ G\ F\ \Delta$ (tactic; page 104) 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.

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.

unfold_data_at (tactic; page 53) 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 53) Like unfold_data_at, but starts with field_at $sh\ t\ path\ v\ p$.