Verifiable C

Applying the Verified Software Toolchain to C programs

Version 1.5+
December 29, 2015

Andrew W. Appel

with Josiah Dodds and Qinxiang Cao

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. There are other proofs above (regarding proof automation tools for the use of the program logic) and below (regarding the execution of assembly language on a concurrent machine). This chain of proofs from top to bottom, connected in Coq at specification interfaces, is called the *Verified Software Toolchain*.

Verified
Software
Toolchain

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

It is also useful to read the brief introductions to Hoare Logic and Separation Logic, covered in Appel's *Program Logics for Certified Compilers*, Chapters 2 and 3.

2 *Installation*

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

1. Coq 8.4pl6, from coq.inria.fr. Follow the standard installation instructions.
2. CompCert 2.5 (or 2.5.1 if available), from compcert.inria.fr. You will want to build the *clightgen* tool, using these commands: ./configure ia32-linux; make clightgen. You might replace ia32-linux with ia32-macosx or ia32-cygwin.
3. VST 1.6 (if available), or else a recent stable trunk version such as f42918bceb. Follow the instructions in the file BUILD_ORGANIZATION.

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 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.

4 Use the IDE

Chapter 5 through Chapter 15 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.

5 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 22 and **??**.

After the boilerplace (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 id_f ... where f is the name of the C function.

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." But really, the Coq variable _a does not have type (pointer-to-int); it has type ident (see page 8).

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 $0 \le \mathsf{size} \le \mathsf{Int.max\_signed}$ 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, the contents of a C scalar variable is always a 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 19. 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 return 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 Int.signed(Int.repr (sum_Z contents)) ≠ (sum_Z contents). In general, for a claim about Int.repr(x) to be *useful*, one also needs a claim that $0 \leq x \leq$ Int.max_unsigned or Int.min_signed $\leq x \leq$ Int.max_signed. The caller of this function will probably need to prove Int.min_signed $\leq$ sum_Z contents $\leq$ Int.max_signed in order to make much use of the post-condition.

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

6 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 : funspecs := sumarray_spec :: main_spec::nil.

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).

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).

7 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, you can 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. It's written, semax Δ P c Q, where P is the precondition, c is the command, and Q is the postcondition. 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_COMMMANDS.

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;.

8 forward

We do Hoare logic proof by forward symbolic execution. On page 17 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.

Another forward goes through s=0; to yield a proof goal with a LOCAL binding for the _s variable.

9 *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.

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 10 shows how to prove such goals.

10 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 variable environment, mapping identifiers (ident) to the values of globals, addressable locals, and tempvars (nonaddressable 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 $\Delta\ P\ c\ Q$ 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_sumarr
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)
$\vdash$ 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 18 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 59). In this case, the goal solves entirely automatically. In particular, $0 \le i \le$ 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.

11 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 in 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.

So, forward succeeds; but it leaves an array-bounds subgoal:

ENTAIL Delta, PROP(...) LOCAL(...) SEP(...)
⊢ tc_expr Delta (Etempvar _a (tptr tint)) &&
    local `(tc_val tint (Znth i (map Vint (map Int.repr contents)) Vundef)) &&
    (tc_expr Delta (Etempvar _i tint) && TT)

The two tc_expr conjuncts are trivial (they are $\beta\eta$-equal to TT) but the middle conjunct is nontrivial. To clean things up, we run entailer!, which leaves this subgoal:

HRE : $i <$ Zlength (map Vint (map Int.repr contents))
H1 : $0 \leq i \leq$ Zlength (map Vint (map Int.repr contents))
  (other above-the-line hypotheses elided)

---

is_int I32 Signed (Znth i (map Vint (map Int.repr contents)) Vundef)

For the load to succeed, the i element of (map Vint (map Int.repr contents)) must actually be an integer, not an undefined value. To prove this, we use the Znth_map lemma to move the Znth inside the Vint, leaving the goal,

is_int I32 Signed (Vint (Znth i (map Int.repr contents) Int.zero))

This is an instance of is_int I32 Signed (Vint ...) which is $\beta\eta$-equal to True. However, when we rewrote by Znth_map, that left a subgoal,

HRE : $i <$ Zlength (map Vint (map Int.repr contents))
H1 : $0 \leq i \leq$ Zlength (map Vint (map Int.repr contents))
  (other above-the-line hypotheses elided)

$0 \leq i <$ Zlength (map Int.repr contents)

This solves straightforwardly as shown in the proof script.

12 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 entailmant 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 (map Vint (map Int.repr contents)) ≤ Int.max_signed
HRE : i < Zlength (map Vint (map Int.repr contents))
H1 : 0 ≤ i ≤ Zlength (map Vint (map Int.repr contents))
------------------------------------(1/1)
Vint (Int.repr (sum_Z (sublist 0 (i + 1) contents))) =
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))
```

The sem_add_default comes from the semantics of C expression evaluation: adding integers means one thing, but adding an integer to a Vundef is undefined, and so on. To clear that sludge out of the way, we move the Znth inside the Vint just as on page 27, then simpl, yielding this goal:

```
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.add (Int.repr (sum_Z (sublist 0 i contents)))
        (Int.repr (Znth i contents 0)))
```

The lemma add_repr: $\forall i\, j$, Int.add (Int.repr i) (Int.repr j) = Int.repr $(i+j)$ is useful here; followed by f_equal, leaves:

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

Now the lemma sublist_split: $\forall l\, m\, h$ al, $0 \le l \le m \le h \le |\mathsf{al}| \to$ 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.

13 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. (When this automatically, it leaves no proof goal at all.) The goal is a *lowered* entailment (on mpred assertions).

After doing simpl to clear away some C-expression-evaluation sludge, 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 30).

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.

14 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 28.

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 35.

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 binds variables a:val, sh:share, contents:list Z, size: Z, so the type of the witness will be (val*(share*(list Z * list 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.

15 *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_skipn.
semax_func_cons body_sumarray.
semax_func_cons body_main.
apply semax_func_nil.
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).

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

The *base level* separation logic is built, like any separation logic, from predicates on "heaplets". The grammar of base-level separation-logic expressions is,

R	::=	emp	empty
		$R_1 * R_2$	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

17 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.

18 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* gratuitous 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: $0 \leq a$
H0: $a < b$

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

This entailment proves straightforwardly by entailer!.

19 *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.

For most purposes, specifications and proofs of C programs should use Z instead of 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:

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

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

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

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

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 $0 \leq i < n$; $n \leq$ Int.max_signed. Then the start_function tactic will move them above the line, and the other tactics mentioned above will make use of them.

To see an example in action, look at progs/verif_sumarray.v. The 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.

20 *Values:* Vint, Vptr

```
Definition block : Type := positive.

Inductive val: Type :=
  | Vundef: val
  | Vint: int → val
  | Vlong: int64 → val
  | Vfloat: float → val
  | Vsingle: float32 → val
  | Vptr: block → 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 $(\mathsf{Vptr}\,b\,z) + (\mathsf{Vint}\ i) \;=\; \mathsf{Vptr}\,b\,(z+i)$. Of course, the C-language + operator first multiplies i by the size of the array-element that 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 25). 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."

21 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.
```

22 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 38.

23 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.

24 *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)

25 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

26 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).
```

27 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*.

28 *Localdefs:* temp, lvar, gvar

The LOCALpart 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.

29 *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 $\Delta,\vec{P},\vec{Q},\vec{R}$ 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 $\Delta,\vec{P},\vec{Q},\vec{R}$ 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, substitute 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.

30 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:
$$\mathsf{field\_at}\ \pi\ t\ path\ v\ p\ \vdash !!(\mathsf{field\_compatible}\ t\ path\ p \wedge \mathsf{value\_fits}\ (\mathsf{nested\_field\_type}\ t\ path)\ v)$$
$$\mathsf{data\_at}\ \pi\ t\ v\ p\ \vdash !!(\mathsf{field\_compatible}\ t\ \mathsf{nil}\ p \wedge \mathsf{value\_fits}\ t\ v)$$
memory_block_local_facts:
$$\mathsf{memory\_block}\ \pi\ n\ p\ \vdash !!\ \mathsf{isptr}\ p$$

The assertion $(\mathsf{Zlength}\ v = n)$ is actually a consequence of value_fits when t is an array type. See Chapter 32.

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.

31 *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,

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

but don't write it that way! For one thing, the converse is not valid:

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

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

this is a *nonspatial* assertion, 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) put field_compatible assertions above the line; see Chapter 30.

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

32 *value_fits*

The spatial maps-to assertion, data_at π 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,(…,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.

33 *Cancel*

(PLCC Ch. 26)

Given an entailment $(A_1 * A_2) * ((A_3 * A_4) * A_5) \vdash A'_4 * (A'_5 * A'_1) * (A'_3 * A'_2)$ for any associative-commutative rearrangement of the A_i, and where (for each i), A_i is $\beta\eta$ equivalent to A'_i, then the cancel tactic will solve the goal. When we say A_i is $\beta\eta$ equivalence to A'_i, that is equivalent to saying that (change (A_i) **with** (A'_i)) would succeed.

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 $A_2 * A_3 * A_5 \vdash B_1 * B_2$.

If the goal is $(A_1 * A_2) * ((A_3 * A_4) * A_5) \vdash A'_4 * \top * A'_1$, where some terms cancel and the rest can be absorbed into $\top$, then cancel will solve the goal.

If the goal has the form

$$\frac{F := ?224 : \mathsf{list}(\mathsf{environ} \to \mathsf{mpred})}{(A_1 * A_2) * ((A_3 * A_4) * A_5) \vdash A'_4 * (\mathsf{fold\_right\ sepcon\ emp}\ F) * A'_1}$$

where F is a *frame* that is an abbreviation for an uninstantiated logical variable of type list(environ→mpred), then the cancel tactic will perform *frame inference*: it will unfold the definition F, instantiate the variable (in this case, to $A_2 :: A_3 :: A_5 ::$ 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:

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

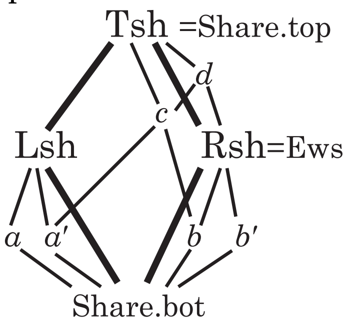
This goal is provable by first rearranging to $(A * C) * D \vdash (B * C) * D$. But cancel may aggressively cancel C and D, leaving $A \vdash B$, which is not provable. You might wonder, what kind of crazy hypothesis is $A * C \vdash B * C$; but indeed such "context-dependent" cancellations do occur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

34 entailer!

35 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: page **??**), and this sort 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 112.

36 Pointer comparisons

37 Proving larg(ish) programs

When your program is not all in one .c file, see also Chapter 38. 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 Delta* can also be invoked individually, to simply display the type context Delta in consise form, 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 : *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 := *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 : *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.

38 *Separate compilation,* semax_ext

What to do when your program is spread over multiple .c files.

Code preparation

In order to separate the namespaces of multiple files compiled by CompCert's clightgen tool, it is necessary to apply

python fix_clightgen.py file1.v ...fileN.v

The script reads in the named files, concisely renames variables etc by making up new positives, and writes the modified files back to the given names.

39 Appendix: catalog of major tactics / commands

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.

[ui] andp_left2 Used frequently in combination with *derives_refl* to clean up proofs states at the end of a loop or function, or after a *forward_seq*.
Subordinate tactics: ...
Superordinate tactics: ...
Defined in XXX.v

[u] cancel Main vehicle for resolving entailments between mpred lists. Performs symbolic cancellation, i.e. performs little inspection of the internals of mpreds. May turn a provable goal into an unprovable goal, by cancelling too aggressively. Occasionally needs to be finished off with a *derives_refl*, in case of syntatically not mtaching compspecs[1].

Subordinate tactics: ...
Superordinate tactics: ...
Defined in XXX.v

[u] derives_refl' (Actually, a lemma, not a tactic). Often followed by $f_e qual$ to solve, say, an entailment between two almost identical *data_at* predicates.

[1]Do these cases still occur? Any other cases?

[u] entailer Vehicle for proving entailments at the mpred level or the assertion level…
Say sth about (non-)inclusion of saturate_locals, normalize, cancel?
Subordinate tactics: …
Superordinate tactics: …
Defined in XXX.v

[ui] derives_refl and derives_refl' Subordinate tactics: …
Superordinate tactics: …
Defined in XXX.v

[u] drop_LOCAL n where $n : nat$. Removes the nth entry of a the LOCAL block of a semax_statement's precondition[2].

Subordinate tactics: …
Superordinate tactics: …
Defined in XXX.v

[u] entailer! Vehicle for proving entailments at the mpred level or the assertion level…In contrast to *entailer*, *entailer!* may turn a provable entailment into an unprovable one, but is usually more efficient.
Say sth about (non-)inclusion of saturate_locals, normalize, cancel?
Subordinate tactics: …
Superordinate tactics: …
Defined in XXX.v

[u] forward Subordinate tactics: …
Superordinate tactics: …
Defined in XXX.

[u] forward_call ARGS Tactic for stepping through the call to a specified function, where *ARGS* is the product of arguments, i.e. an in-order instantiation of the function specification's WITH clause (Note: little

[2] or of an assertion-level entailment's LHS?

type checking is applied to *ARGS* and the tactic may behave erratically if incorrect/wrongly-typed arguments are supplied).

Always leaves a *semax*-subgoal for the program continuation. Frequently, leaves an entailment subgoal for establishing the function procondition, and/or a subgoal for a typing side condition.

Subordinate tactics: ...
Superordinate tactics: None.
Defined in XXX.v, with the underlying proof rules being defined in YYY.v

[u] forward_for Including its variants for simple bounds etc

Is expected to leave the following subgoals:...

Subordinate tactics: ...
Superordinate tactics: ...
Defined in XXX

[u] forward_seq Subordinate tactics: ...
Superordinate tactics: ...
Defined in XXX.

[i] mkConciseDelta V G F Delta 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.

Superordinate tactics: *semax_subcommand*
Defined in semax_tactics.v.

[u] semax_subcommand V G F 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 are as in the statement of surrounding semax-body lemma, i.e. are of type *varspecs*, *funspecs*, and *function*.

Subordinate tactic: *mkConciseDelta*. Defined in semax_tactics.v.

40 BOUNDARY

Everything beyond here is junk, left over from the old Verifiable C manual, that needs to be rewritten and moved above the boundary.

41 Getting started

This *summary reference manual* is a brief guide to the VST Separation Logic for the C language. The Verified Software Toolchain and the principles of its program logics are described in the book:

Program Logics for Certified Compilers,
by Andrew W. Appel *et al.*, Cambridge University Press, 2014.

TO INSTALL THE VST SEPARATION LOGIC FOR C LIGHT:

1. Get VST from vst.cs.princeton.edu/download, or get the bleeding-edge version from https://github.com/PrincetonUniversity/VST.
2. Examine vst/compcert/VERSION to determine which version of CompCert to download. The VST comes with a copy of the CompCert front-end, in vst/compcert/, but (at present) CompCert's clightgen utility is not buildable from just the front-end distributed with VST. You'll need clightgen to translate .c files into .v files containing C light abstract syntax. Thus it's recommended to download and build CompCert.
3. Get CompCert from compcert.inria.fr/download.html and run ./configure to list configurations. Select the correct option for your machine, then run ./configure <option> followed by make clightgen. Create a file vst/CONFIGURE containing a definition for CompCert's location; if vst and CompCert are installed in the same parent directly, use COMPCERT=../compcert

   If you have not installed CompCert, use the CompCert front-end packaged with VST. Do not create a CONFIGURE file, and do: cd vst/compcert; ./make
4. In the vst directory, make.

See also the file vst/BUILD_ORGANIZATION.

The verif_reverse.v example is described in PLCC Chapter 27. You might find it interesting to open this in the IDE, using the command shown above, and interactively step through the definitions and proofs.

Before doing proofs of your own, you may find it helpful to step through this tutorial on C light expressions and assertions:
cd examples/floyd_tut; coqide tutorial.v
(this tutorial sets up its own load paths.)

42 Differences from PLCC

The book *Program Logics for Certified Compilers* (Cambridge University Press, early 2014) describes *Verifiable C* version 1.1. More recent VST versions differ in the following ways from what the PLCC book describes:

- In the LOCAL component of an assertion,
  temp i v is the recommended way to write `(eq v) (eval_id i), and
  var i t v is the recommended way to write `(eq v) (eval_var i t).
  See Chapter 55 of this manual.
- The type-checker now has a more refined view of char and short types (see Chapter 57 of this manual).
- field_mapsto is now called field_at, and it is dependently typed; see Chapter 68 of this manual.
- 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. See Chapter 68 of this manual.
- Supercanonical form is now suggested; see Chapter 55 of this manual.
- For function calls, do not use forward (except to get advice about the witness type); instead, use forward_call. See page 108.
- C functions may now fall through the end of the function body, and this is (per the C semantics) equivalent to a return; statement.

43 Memory predicates

The axiomatic semantics (Hoare Logic of Separation) treats memories abstractly. One never has a variable m of type *memory*. Instead, one uses the Hoare Logic to manipulate predicates P on memories. Our type of "memory predicates" is called mpred

Although intuitively mpred "feels like" the type memory → Prop, the underlying semantic model is different; thus we keep the type mpred abstract (opaque). See *Program Logics for Certified Compilers (PLCC)* for more explanation.

On the type mpred we form a natural deduction system NatDed(mpred) with conjuction &&, disjunction ||, etc.; a separation logic SepLog(mpred) with separating conjunction $*$ and emp; and an indirection theory Indir(mpred) with ⊳ "later."

The natural deduction system has a sequent (entailment) operator written P |-- Q in Coq (written $P \vdash Q$ in print), where P,Q : mpred. We write bientailment simply as $P = Q$ since we assume axioms of extensionality.

44 Separation Logic

(see PLCC Chapter 12)

```
Class NatDed (A: Type) := mkNatDed {
  andp: A → A → A;        (Notation &&)
  orp: A → A → A;        (Notation ||)
  exp: ∀{T:Type}, (T → A) → A;        (Notation EX)
  allp: ∀{T:Type}, (T → A) → A;        (Notation ALL)
  imp: A → A → A;        (Notation -->, here written ⟶)
  prop: Prop → A;        (Notation !! )
  derives: A → A → Prop;        (Notation |--, here written ⊢)
  pred_ext: ∀P Q, P⊢Q → Q⊢P → P=Q;
  derives_refl: ∀P, P ⊢P;
  derives_trans: ∀{P Q R}, P ⊢Q → Q ⊢R → P⊢R;
  TT := !!True;
  FF := !!False;
  andp_right: ∀X P Q:A, X⊢P → X⊢Q → X⊢(P&&Q);
  andp_left1: ∀P Q R:A, P⊢R → P&&Q ⊢R;
  andp_left2: ∀P Q R:A, Q⊢R → P&&Q ⊢R;
  orp_left: ∀P Q R, P⊢R → Q⊢R →  P||Q ⊢R;
  orp_right1: ∀P Q R, P⊢Q → P⊢ Q||R;
  orp_right2: ∀P Q R, P⊢R → P⊢ Q||R;
  exp_right: ∀{B: Type}(x:B)(P:A)(Q: B→A), P⊢Q x → P⊢ EX x:B, Q;
  exp_left: ∀{B: Type}(P:B→A)(Q:A), (∀ x, P x ⊢Q) → EX x:B,P ⊢Q;
  allp_left: ∀{B}(P: B → A) x Q, P x⊢Q → ALL x:B,P⊢Q;
  allp_right: ∀{B}(P: A)(Q:B→A), (∀ v, P⊢ Q v) → P⊢ ALL x:B,Q;
  imp_andp_adjoint: ∀P Q R, P&&Q⊢R  ↔  P⊢(Q⟶R);
  prop_left: ∀(P: Prop) Q, (P → (TT⊢Q)) → !!P ⊢Q;
  prop_right: ∀(P: Prop) Q, P → (Q⊢ !!P);
  not_prop_right: ∀(P:A)(Q:Prop), (Q → (P⊢FF))→  P⊢!!(~Q)
}.
```

```
Class SepLog (A: Type) {ND: NatDed A} := mkSepLog {
  emp: A;
  sepcon: A → A → A;      (Notation ∗ )
  wand: A → A → A;      (Notation -∗; here written −∗ )
  ewand: A → A → A;      (no notation; here written −◦ )
  sepcon_assoc: ∀P Q R, (P∗Q)∗R = P∗(Q∗R);
  sepcon_comm: ∀P Q, P∗Q = Q∗P;
  wand_sepcon_adjoint: ∀(P Q R: A), P∗Q⊢R ↔P ⊢Q−∗R;
  sepcon_andp_prop: ∀P Q R, P∗(!!Q && R) = !!Q && (P∗R);
  sepcon_derives: ∀P P' Q Q' : A, P⊢P' → Q⊢Q' → P∗Q ⊢P'∗Q';
  ewand_sepcon: ∀(P Q R : A), (P∗Q)−◦ R = P −◦ (Q −◦ R);
  ewand_TT_sepcon: ∀(P Q R: A),
       (P∗Q)&&(R−◦TT) ⊢(P &&(R−◦TT))∗(Q && (R−◦TT));
  exclude_elsewhere: ∀P Q: A, P∗Q ⊢(P &&(Q−◦ TT))∗Q;
  ewand_conflict: ∀P Q R, P∗Q⊢FF → P&&(Q−◦ R) ⊢FF
}.
Class Indir (A: Type) {ND: NatDed A} := mkIndir {
  later: A → A; (Notation ▷ )
  now_later: ∀P: A, P ⊢▷ P;
  later_K: ∀P Q, ▷ (P⟶Q) ⊢(▷ P ⟶▷ Q);
  later_allp: ∀T (F: T→A), ▷ (ALL x:T, F x) = ALL x:T, ▷ (F x);
  later_exp: ∀T (F: T→A), EX x:T, ▷ (F x) ⊢▷ (EX x: F x);
  later_exp': ∀T (any:T) F, ▷ (EX x: F x) = EX x:T, ▷ (F x);
  later_imp: ∀P Q, ▷ (P⟶Q) = (▷ P ⟶▷ Q);
  loeb: ∀P, ▷ P ⊢P → TT ⊢P
}.
Class SepIndir (A: Type) {NA: NatDed A}{SA: SepLog A}{IA: Indir A} :=
 mkSepIndir {
  later_sepcon: ∀P Q, ▷ (P ∗ Q) = ▷ P ∗ ▷ Q;
  later_wand: ∀P Q, ▷ (P −∗ Q) = ▷ P −∗ ▷ Q;
  later_ewand: ∀P Q, ▷ (P −◦ Q) = (▷ P) −◦ (▷ Q)
}.
```

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

Aside from the standard operators and axioms of separation logic, we have exactly two primitive 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.

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. But we almost always use one of the following 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.

field_at (sh: share) (t: type) (f: list ident) (w: reptype (nested_field_type2 f) (v: v
describes a heap that holds just field fld of struct-value v, belonging to struct-type t, containing value w. If type t describes a nested struct type, then f can actually be a path of field selections that descends into the nested structures. If f is the empty path, then the field is equivalent to data_at. The type of w is a dependent type. *Note: arguments w,v are swapped compared to the PLCC book.*

field_at_(sh: share) (t: type) (fld: ident) (v: val) : mpred
is the corresponding uninitialized structure-field.

46 CompCert C

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.
Cop. Semantics of C-language arithmetic operators.
Clight. Abstract syntax of C-light expressions, statements, and functions.
veric.expr. (from VST, not CompCert) Semantics of expression evaluation.

Some of the important types and operators are described over the next few pages.

47 Verifiable C programming

See PLCC Chapter 22

In writing Verifiable C programs you must:

- Make each dereference into a top level expression (PLCC page 143)
- Make most pointer comparisons into a top level expression (PLCC page 145)
- Remove casts between int and pointer types (result in values that crash if used)

The clightgen tool automatically:

- Factors function calls into top level expressions
- Factors logical and/or operators into if statements (to capture short circuiting behavior)

Proof automation detects these two transformations and processes them with a single tactic application.

If your program uses malloc or free, you must declare and specify these as external functions. If you don't want to keep track of the size of each allocated object, you may want to change the interface of the free function. We do this in our example definitions of malloc and free in progs/queue.c and their specifications in progs/verif_queue.v.

48 32-bit Integers

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

```
Inductive comparison := Ceq | Cne | Clt | Cle | Cgt | Cge.
Definition wordsize: nat := 32. (* also instantiations for 8, 16, 64 *)
Definition modulus : Z := two_power_nat wordsize.
Definition half_modulus : Z := modulus / 2.
Definition max_unsigned : Z := modulus -1.
Definition max_signed : Z := half_modulus -1.
Definition min_signed : Z := -half_modulus.

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

Definition zero := repr 0.

Definition eq (x y: int) : bool.
Definition lt (x y: int) : bool.
Definition ltu (x y: int) : bool.
Definition neg (x: int): int := repr (- unsigned x).
Definition add (x y: int): int := repr (unsigned x + unsigned y).
Definition sub (x y: int): int := repr (unsigned x -unsigned y).
Definition mul (x y: int): int := repr (unsigned x * unsigned y).
Definition divs (x y: int) : int.
Definition mods (x y: int) : int.
Definition divu (x y: int) : int.
Definition modu (x y: int) : int.
Definition and (x y: int): int := bitwise_binop andb x y.
Definition or (x y: int): int := bitwise_binop orb x y.
Definition xor (x y: int) : int := bitwise_binop xorb x y.
Definition not (x: int) : int := xor x mone.
Definition shl (x y: int): int.
Definition shru (x y: int): int.
```

Definition shr (x y: int): int.
Definition rol (x y: int) : int.
Definition ror (x y: int) : int.
Definition rolm (x a m: int): int.
Definition cmp (c: comparison) (x y: int) : bool.
Definition cmpu (c: comparison) (x y: int) : bool.

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

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

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

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

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

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

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

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

49 C expression syntax

(compcert/cfrontend/Clight.v)

```
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.
```

```
Definition typeof (e: expr) : type :=
  match e with
  | Econst_int _ty ⇒ ty
  | Econst_float _ty ⇒ ty
  | Evar _ty ⇒ ty
  | ... et cetera.
```

50 C operators


```
Function bool_val (v: val) (t: type) : option bool :=
  match classify_bool t with
  | bool_case_i ⇒
      match v with
      | Vint n ⇒ Some (negb (Int.eq n Int.zero))
      | _ ⇒ None
      end
  | bool_case_f ⇒
      match v with
      | Vfloat f ⇒ Some (negb (Float.cmp Ceq f Float.zero))
      | _ ⇒ None
      end
  | bool_case_p ⇒
      match v with
      | Vint n ⇒ Some (negb (Int.eq n Int.zero))
      | Vptr b ofs ⇒ Some true
      | _ ⇒ None
      end
  | bool_default ⇒ None
  end.

Function sem_neg (v: val) (ty: type) : option val :=
  match classify_neg ty with
  | neg_case_i sg ⇒
      match v with
      | Vint n ⇒ Some (Vint (Int.neg n))
      | _ ⇒ None
      end
  | neg_case_f ⇒
      match v with
      | Vfloat f ⇒ Some (Vfloat (Float.neg f))
      | _ ⇒ None
      end
```

```
  | neg_default ⇒ None
  end.

Function sem_add (v1:val) (t1:type) (v2: val) (t2:type) : option val :=
  match classify_add t1 t2 with
  | add_case_ii sg ⇒ (**r integer addition *)
      match v1, v2 with
      | Vint n1, Vint n2 ⇒ Some (Vint (Int.add n1 n2))
      | _, _ ⇒ None
      end
  | add_case_ff ⇒ (**r float addition *)
      match v1, v2 with
      | Vfloat n1, Vfloat n2 ⇒ Some (Vfloat (Float.add n1 n2))
      | _, _ ⇒ None
      end
  | add_case_if sg ⇒ (**r int plus float *)
      match v1, v2 with
      | Vint n1, Vfloat n2 ⇒ Some (Vfloat (Float.add (cast_int_float sg n1) n2))
      | _, _ ⇒ None
      end
  | ... (cases omitted)
  | add_case_ip ty _ ⇒ (**r integer plus pointer *)
      match v1,v2 with
      | Vint n1, Vptr b2 ofs2 ⇒
         Some (Vptr b2 (Int.add ofs2 (Int.mul (Int.repr (sizeof ty)) n1)))
      | _, _ ⇒ None
      end
  | add_default ⇒ None
end.

Function sem_sub (v1:val) (t1:type) (v2: val) (t2:type) : option val.
Function sem_mul (v1:val) (t1:type) (v2: val) (t2:type) : option val.
Function sem_div (v1:val) (t1:type) (v2: val) (t2:type) : option val.
Function sem_mod (v1:val) (t1:type) (v2: val) (t2:type) : option val.
Function sem_and (v1:val) (t1:type) (v2: val) (t2:type) : option val.
```

```
Function sem_cmp (c:comparison)
                 (v1: val) (t1: type) (v2: val) (t2: type)
                 (m: mem): option val :=
  match classify_cmp t1 t2 with
  | cmp_case_ii Signed ⇒
      match v1,v2 with
      | Vint n1, Vint n2 ⇒ Some (Val.of_bool (Int.cmp c n1 n2))
      | _, _ ⇒ None
      end
  | ... (many more cases )
  end.

Definition sem_binary_operation
    (op: binary_operation)
    (v1: val) (t1: type) (v2: val) (t2:type)
    (m: mem): option val :=
  match op with
  | Oadd ⇒ sem_add v1 t1 v2 t2
  | Osub ⇒ sem_sub v1 t1 v2 t2
  | Omul ⇒ sem_mul v1 t1 v2 t2
  | Omod ⇒ sem_mod v1 t1 v2 t2
  | Odiv ⇒ sem_div v1 t1 v2 t2
  | Oand ⇒ sem_and v1 t1 v2 t2
  | Oor ⇒ sem_or v1 t1 v2 t2
  | Oxor ⇒ sem_xor v1 t1 v2 t2
  | Oshl ⇒ sem_shl v1 t1 v2 t2
  | Oshr ⇒ sem_shr v1 t1 v2 t2
  | Oeq ⇒ sem_cmp Ceq v1 t1 v2 t2 m
  | One ⇒ sem_cmp Cne v1 t1 v2 t2 m
  | Olt ⇒ sem_cmp Clt v1 t1 v2 t2 m
  | Ogt ⇒ sem_cmp Cgt v1 t1 v2 t2 m
  | Ole ⇒ sem_cmp Cle v1 t1 v2 t2 m
  | Oge ⇒ sem_cmp Cge v1 t1 v2 t2 m
  end.
```

51 C expression evaluation

(vst/veric/expr.v)

Definition eval_id (id: ident) (ρ: environ).
  (∗ look up the tempory variable ``id'' in ρ ∗)

Definition eval_cast (t t': type) (v: val) : val.
  (∗ cast value v from type t to type t', but beware! There are
    be three types involved, if you include the native type of v. ∗)

Definition eval_unop (op: Cop.unary_operation) (t1 : type) (v1 : val) : val.

Definition eval_binop (op: Cop.binary_operation)
                (t1 t2 : type) (v1 v2: val) : val.

Definition force_ptr (v: val) : val :=
    match v **with** Vptr l ofs ⇒ v | _ ⇒ Vundef **end**.

Definition eval_struct_field (delta: Z) (v: val) : val.
  (∗ offset the pointer-value v by delta ∗)

Definition eval_field (ty: type) (fld: ident) (v: val) : val.
  (∗ calculate the lvalue of (but do not fetch/dereference!)
    a structure/union field of value v ∗)

Definition eval_var (id:ident) (ty: type) (rho: environ) : val.
  (∗ Get the lvalue (address of) an addressable local variable
    (if there is one of that name) or else a global variable ∗)

Definition deref_noload (ty: type) (v: val) : val.
  (∗ For By_reference types such as arrays that dereference
    without actually fetching ∗)
 match access_mode ty **with** By_reference ⇒ v | _ ⇒ Vundef **end**.

```
Fixpoint eval_expr (e: expr) : environ →val :=
 match e with
 | Econst_int i ty ⇒ `(Vint i)
 | Econst_float f ty ⇒ `(Vfloat f)
 | Etempvar id ty ⇒ eval_id id
 | Eaddrof a ty ⇒ eval_lvalue a
 | Eunop op a ty ⇒ `(eval_unop op (typeof a)) (eval_expr a)
 | Ebinop op a1 a2 ty ⇒
             `(eval_binop op (typeof a1) (typeof a2))
               (eval_expr a1) (eval_expr a2)
 | Ecast a ty ⇒ `(eval_cast (typeof a) ty) (eval_expr a)
 | Evar id ty ⇒ `(deref_noload ty) (eval_var id ty)
 | Ederef a ty ⇒ `(deref_noload ty) (`force_ptr (eval_expr a))
 | Efield a i ty ⇒ `(deref_noload ty)
                      (`(eval_field (typeof a) i) (eval_lvalue a))
 end

 with eval_lvalue (e: expr) : environ →val :=
 match e with
 | Evar id ty ⇒ eval_var id ty
 | Ederef a ty ⇒ `force_ptr (eval_expr a)
 | Efield a i ty ⇒ `(eval_field (typeof a) i) (eval_lvalue a)
 | _⇒ `Vundef
 end.
```

52 C type checking

(See PLCC Chapter 25)

Ideally, you will never notice the typechecker, but it may occasionally generate side conditions that can not be solved automatically. If you get a proof goal from the typechecker, it will be an entailment P ⊢denote_tc_assert (...). PLCC Chapter 26 discusses what you can do to solve these goals.

If you are asked to prove an entailment where the typechecking condition evaluates to False, this may be because your program is not written in Verifiable C. You may need to perform some local transformations on your C program in order to proceed. We listed these transformations on page 77.

The type-context will always be visible in your proof in a line that looks like Delta := abbreviate : tycontext. The abbreviate hides the implementation of the type context (which is generally large and uninteresting). The query_context tactic shows the result of looking up a variable in a typecontext. The tactic query_context Delta _p. will add hypothesis QUERY : (temp_types Delta) ! _p = Some (tptr t_struct_list, true). This means that in Delta, _p is a temporary variable with type tptr t_struct_list and that it is known to be initialized.

53 Lifted separation logic *(See PLCC Chapter 21)*

Assertions in our Hoare triple of separation are presented as env → mpred, that is, functions from environment to memory-predicate, using our natural deduction system NatDed(mpred) and separation logic SepLog(mpred).

Given a separation logic over a type B of formulas, and an arbitrary type A, we can define a *lifted* separation logic over functions $A \to B$. The operations are simply lifted pointwise over the elements of A. Let $P,Q : A \to B$, let $R : T \to A \to B$ then define,

$$\begin{array}{rrcl}
(P \mathbin{\&\&} Q): & A \to B & := & \text{fun}\ a \Rightarrow Pa \mathbin{\&\&} Qa \\
(P \parallel Q): & A \to B & := & \text{fun}\ a \Rightarrow Pa \parallel Qa \\
(\exists x.R(x)): & A \to B & := & \text{fun}\ a \Rightarrow \exists x.\ Rxa \\
(\forall x.R(x)): & A \to B & := & \text{fun}\ a \Rightarrow \forall x.\ Rxa \\
(P \longrightarrow Q): & A \to B & := & \text{fun}\ a \Rightarrow Pa \longrightarrow Qa \\
(P \vdash Q): & A \to B & := & \forall a.\ Pa \vdash Qa \\
(P * Q): & A \to B & := & \text{fun}\ a \Rightarrow Pa * Qa \\
(P \mathrel{-\!\!*} Q): & A \to B & := & \text{fun}\ a \Rightarrow Pa \mathrel{-\!\!*} Qa
\end{array}$$

In Coq we formalize the typeclass instances LiftNatDed, LiftSepLog, etc., as shown below. For a type B, whenever NatDed B and SepLog B (and so on) have been defined, the lifted instances NatDed (A→B) and SepLog (A→B) (and so on) are automagically provided by the typeclass system.

```
Instance LiftNatDed(A B: Type){ND: NatDed B}: NatDed (A→B):=
 mkNatDed (A →B)
    (*andp*) (fun P Q x ⇒ andp (P x) (Q x))
    (*orp*) (fun P Q x ⇒ orp (P x) (Q x))
    (*exp*) (fun {T} (F: T →A →B) (a: A) ⇒ exp (fun x ⇒ F x a))
    (*allp*) (fun {T} (F: T →A →B) (a: A) ⇒ allp (fun x ⇒ F x a))
    (*imp*) (fun P Q x ⇒ imp (P x) (Q x))
    (*prop*) (fun P x ⇒ prop P)
    (*derives*) (fun P Q ⇒ ∀x, derives (P x) (Q x))
     - - - - - - - - - - - - - - - - - -.
```

```
Instance LiftSepLog (A B: Type) {NB: NatDed B}{SB: SepLog B}
       : SepLog (A → B).
 apply (mkSepLog (A → B) _(fun ρ ⇒ emp)
            (fun P Q ρ ⇒ P ρ ∗ Q ρ) (fun P Q ρ ⇒ P ρ -∗ Q ρ)).
 (∗ fill in proofs here ∗)
```

In particular, if P and Q are functions of type environ→mpred then we can write $P * Q$, $P \&\& Q$, and so on.

Consider this assertion:

```
fun ρ ⇒ mapsto sh tint (eval_id _x ρ) (eval_id _y ρ)
          ∗ mapsto sh tint (eval_id _u ρ) (Vint Int.zero)
```

which might appear as the precondition of a Hoare triple. It represents $(x \mapsto y) * (u \mapsto 0)$ written in informal separation logic, where x, y, u are C-language variables of integer type. Because it can be inconvenient to manipulate explicit lambda expressions and explicit environment variables ρ, we may write it in lifted form,

```
  `(mapsto sh tint) (eval_id _x) (eval_id _y)
∗ `(mapsto sh tint) (eval_id _u) `(Vint Int.zero)
```

Each of the first two backquotes lifts a function from type val→val→mpred to type (environ→val)→(environ→val)→(environ→mpred), and the third one lifts from val to environ→val.

54 Canonical forms

(See PLCC section 26)

We write a *canonical form* of an assertion as,

$$\mathrm{PROP}(P_0;P_1;\ldots,P_{l-1})\ \mathrm{LOCAL}(Q_0;Q_1;\ldots,Q_{m-1})\ \mathrm{SEP}(R_0;R_1;\ldots,R_{n-1})$$

The $P_i : \mathsf{Prop}$ are Coq propositions—these are independent of the program variables and the memory. The $Q_i : \mathsf{environ} \to \mathsf{Prop}$ are local—they depend on program variables but not on memory. The $R_i : \mathsf{environ} \to \mathsf{mpred}$ are assertions of separation logic, which may depend on both program variables and memory.

The PROP/LOCAL/SEP form is defined formally as,

```
Definition PROPx (P: list Prop) (Q: assert) :=
          andp (prop (fold_right and True P)) Q.
Notation "'PROP' ( x ; .. ; y ) z" :=
       (PROPx (cons x%type .. (cons y%type nil) ..) z) (at level 10) : logic.
Notation "'PROP' ( ) z" := (PROPx nil z) (at level 10) : logic.

Definition LOCALx (Q: list (environ →Prop)) (R: assert) :=
                 andp (local (fold_right (`and) (`True) Q)) R.
Notation " 'LOCAL' ( x ; .. ; y ) z" :=
      (LOCALx (cons x%type .. (cons y%type nil) ..) z) (at level 9) : logic.
Notation " 'LOCAL' ( ) z" := (LOCALx nil z) (at level 9) : logic.

Definition SEPx (R: list assert) : assert := fold_right sepcon emp R.

Notation " 'SEP' ( x ; .. ; y )" :=
        (SEPx (cons x%logic .. (cons y%logic nil) ..)) (at level 8) : logic.
Notation " 'SEP' ( ) " := (SEPx nil) (at level 8) : logic.
Notation " 'SEP' () " := (SEPx nil) (at level 8) : logic.
```

Thus, $\mathrm{PROP}(P_0;P_1)\mathrm{LOCAL}(Q_0;Q_1)\mathrm{SEP}(R_0;R_1)$ is equivalent to $\mathsf{prop}\,P_0 \wedge \mathsf{prop}\,P_1\ \&\&\ `\mathsf{prop}\,Q_0\ \&\&\ `\mathsf{prop}\,Q_1\ \&\&\ (R_0 * R_1)$.

55 *Supercanonical forms*

A canonical form $\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R})$ is *supercanonical* if:

- Every element of $\vec{Q}$ has the form temp i V or var i t V, where V is a Coq expression of type val and i is $\beta\eta$-equivalent to a constant (a ground term of type ident). The term temp i V (of type environ→Prop) is equivalent to `(eq $V$) (eval_id $i$). The term var $i$ $t$ $V$ (of type environ→Prop) is equivalent to `(eq V) (eval_var i t).
- Every element of R is `(E) where E is a Coq expression of type mpred.

When assertions (preconditions of semax) are kept in supercanonical form, the forward tactic for symbolic execution runs *much* faster. That is,

- forward through assignment statements (including loads/stores) is up to 10 times faster for supercanonical preconditions than for ordinary (canonical) preconditions.
- Future versions of the forward tactic may *require* the precondition to be in supercanonical form.

56 Go_lower

(See PLCC Chapter 26)

An entailment PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP($\vec{R}$) $\vdash$ PROP($\vec{P}'$)LOCAL($\vec{Q}'$)SEP($\vec{R}'$) is a sequent in our *lifted* separation logic; each side has type environ→ mpred. By definition of the lifted entailment $\vdash$ it means exactly, $\forall \rho.$ PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP($\vec{R}$)$\rho \vdash$ PROP($\vec{P}'$)LOCAL($\vec{Q}'$)SEP($\vec{R}'$)ρ. There are two ways to prove such an entailment: Explicitly introduce ρ (descend into an entailment on mpred) and unfold the PROP/LOCAL/SEP form; or stay in canonical form and rewrite in the lifted logic. Either way may be appropriate; this chapter describes how to descend. The go_lower tactic, described on this page, is rarely called directly; it is the first step of the entailer tactic (page 100) when applied to lifted entailments.

The tactic go_lower tactic does the following:

1. intros ?rho, as described above.
2. If the first conjunct of the left-hand-side LOCALs is tc_environ Δ ρ, move it above the line; this be useful in step 6.
3. Unfold definitions for *canonical forms* (PROPx LOCALx SEPx), *expression evaluation* (eval_exprlist eval_expr eval_lvalue cast_expropt eval_cast eval_binop eval_unop), *casting* (eval_cast classify_cast) *typechecking* (tc_expropt tc_expr tc_lvalue typecheck_expr typecheck_lvalue denote_tc_assert), *function postcondition operators* (function_body_ret_assert make_args' bind_ret get_result1 retval), *lifting operators* (liftx LiftEnviron Tarrow Tend lift_S lift_T lift_prod lift_last lifted lift_uncurry_open lift_curry local lift lift0 lift1 lift2 lift3).
4. Simplify by simpl.
5. Rewrite by the rewrite-hint environment go_lower, which contains just a very few rules to evaluate certain environment lookups.
6. Recognize local variables.

Local variables that appear in the lifted canonical form as (eval_id _x) will be replaced by Coq variables x, provided that: (1) $\vec{Q}$ includes a clause of the form (tc_environ Δ), and (2) there is a hypothesis name x _x "above the line." (See PLCC section 26). In addition, a typechecking hypothesis for x will be introduced above the line (see Chapter 57).

57 *Welltypedness of variables*

The typechecker ensures some invariants about the values of C-program variables: if a variable is initialized, 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.

The initialization status of addressable local variables is tracked in the separation logic, by assertions such as $v \mapsto \_$ or $v \mapsto i$ for uninitialized and initialized variables, respectively.

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.

The go_lower tactic (usually) deletes the assertion tc_environ Δ ρ after deriving type-checking assertions of the form tc_val τ v for each variable v of type τ; it puts these assertions above the line.

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

Since τ is 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
```

58 Normalize

The normalize tactic performs autorewrite **with** norm and several other transformations. Many of the simplifications performed by normalize on entailments (whether lifted or unlifted) can be done more efficiently and systematically by entailer. However, on Hoare triples, entailer does not apply, and normalize is quite appropriate.

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

Generic separation-logic simplifications.

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

$$(EX x : A, P) * Q = EX x : A, P * Q \qquad P * (EX x : A, Q) = EX x : A, P * Q$$

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

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

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

$$\bot * P = \bot \qquad P \to (!!P \mathbin{\&\&} Q = Q) \qquad P \to (!!P = \top) \qquad P \mathbin{\&\&} P = P$$

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

Unlifting.

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

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

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

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

$$\text{`}(\mathsf{EX}\ x : B,\ P x) = \mathsf{EX}\ x : B,\ \text{`}(P x)) \qquad \text{`}(P * Q) =\ \text{`}P * \text{`}Q$$

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

Pulling nonspatial propositions out of spatial ones.

$$\mathsf{local}\ P\ \ \&\&\ \ !!Q = !!Q\ \ \&\&\ \ \mathsf{local}\ P$$

$$\mathsf{local}\ P\ \ \&\&\ \ (!!Q\ \&\&\ R) = !!Q\ \&\&\ (\mathsf{local}\ P\ \ \&\&\ \ R)$$

$$(\mathsf{local}\ P\ \ \&\&\ \ Q) * R = \mathsf{local}\ P\ \ \&\&\ \ (Q * R)$$

$$Q * (\mathsf{local}\ P\ \ \&\&\ \ R) = \mathsf{local}\ P\ \ \&\&\ \ (Q * R)$$

Canonical forms.

$$\mathsf{local}\ Q_1\ \&\&\ (\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R})) = \mathrm{PROP}(\vec{P})\mathrm{LOCAL}(Q_1;\vec{Q})\mathrm{SEP}(\vec{R})$$

$$\mathrm{PROP}\vec{P}\mathrm{LOCAL}\vec{Q}\mathrm{SEP}(!!P_1;\vec{R}) = \mathrm{PROP}(P_1;\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R})$$

$$\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\mathsf{local}Q_1;\vec{R}) = \mathrm{PROP}(\vec{P})\mathrm{LOCAL}(Q_1;\vec{Q})\mathrm{SEP}(\vec{R})$$

Modular Integer arithmetic.

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

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

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

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

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

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

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

Type checking and miscellaneous.

$$\text{tc\_formals}((i,t)::r) = \text{`and (`(tc\_val } t) \text{ (eval\_id } i) \text{ (tc\_formals } r)$$

$$\text{tc\_formals nil} = \text{`}\top \qquad \text{tc\_andp tc\_TT } e \;=\; e \qquad \text{tc\_andp } e \text{ tc\_TT } \;=\; e$$

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

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

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

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

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

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

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

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

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

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

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

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

Structure fields.

$$\text{field\_mapsto } sh\ t\ fld \text{ (force\_ptr } v) = \text{field\_mapsto } sh\ t\ fld\ v$$

$$\text{field\_mapsto\_ } sh\ t\ fld \text{ (force\_ptr } v) = \text{field\_mapsto\_ } sh\ t\ fld\ v$$

$$\text{field\_mapsto } sh\ t\ x \text{ (offset\_val } v \text{ Int.zero)} = \text{field\_mapsto } sh\ t\ x\ v$$

$$\text{field\_mapsto\_ } sh\ t\ x \text{ (offset\_val } v \text{ Int.zero)} = \text{field\_mapsto\_ } sh\ t\ x\ v$$

$$\text{memory\_block } sh \text{ Int.zero (Vptr } b\ z) \;=\; \text{emp}$$

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

$$\mathsf{normal\_ret\_assert}\ \bot\ \mathsf{ek}\ \mathsf{vl}\ =\ \bot$$

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

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

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

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

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

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

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

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

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

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

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

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

$$\mathsf{function\_body\_ret\_assert}\ t\ P\ \mathsf{EK\_return}\ vl\ =\ \mathsf{bind\_ret}\ vl\ t\ P$$

Function return values.

$$\mathsf{bind\_ret}\ (\mathsf{Some}\ v)\ t\ Q = (!!\mathsf{tc\_val}\ t\ v\ \&\&\ `Q(\mathsf{make\_args}(\mathsf{ret\_temp} :: \mathsf{nil})\ (v :: \mathsf{nil})))$$

$$\mathsf{make\_args}'\ \sigma\ a\ \rho\ = \mathsf{make\_args}\ (\mathsf{map\ fst}\ (\mathsf{fst}\ \sigma))\ (a\ \rho)\ \rho$$

$$\mathsf{make\_args}(i :: l)(v :: r)\rho = \mathsf{env\_set}(\mathsf{make\_args}(l)(r)\rho)\ i\ v$$

$$\mathsf{make\_args\ nil\ nil}\ =\ \mathsf{globals\_only} \qquad \mathsf{get\_result}(\mathsf{Some}\ x) = \mathsf{get\_result1}(x)$$

$$\mathsf{retval}(\mathsf{get\_result1}\ i\ \rho) = \mathsf{eval\_id}\ i\ \rho \qquad \mathsf{retval}(\mathsf{env\_set}\ \rho\ \mathsf{ret\_temp}\ v)\ =\ v$$

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

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

IN ADDITION TO REWRITING, the normalize tactic applies the following rules:

$$P \vdash \top \qquad \bot \vdash P \qquad P \vdash P * \top \qquad (\forall x.\ (P \vdash Q)) \to (EX x : A,\ P \vdash Q)$$

$$(P \to (\top \vdash Q)) \to (!!P \vdash Q) \qquad (P \to (Q \vdash R)) \to (!!P\ \&\&\ Q \vdash R)$$

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, the normalize avoids doing an intro. This allows the user to choose an appropriate name for x.

59 Entailer

Our entailer tactic is a partial solver for entailments in the separation logic over mpred. If it cannot solve the goal entirely, it leaves a simplified subgoal for the user to prove. The algorithm is this:

1. Apply go_lower if the goal is in the lifted separation logic.
2. Gather all the pure propositions to a single pure proposition (in each of the hypothesis and conclusion).
3. Given the resulting goal $!!(P_1 \wedge \ldots \wedge P_n) \&\& (Q_1 * \ldots * Q_m) \vdash !!(P'_1 \wedge \ldots \wedge P'_{n'}) \&\& (Q'_1 \ldots * Q'_{m'})$, move each of the pure propositions P_i "above the line." Any P_i that's an easy consequence of other above-the-line hypotheses is deleted. Certain kinds of P_i are simplified in some ways.
4. For each of the Q_i, saturate_local extracts any pure propositions that are consequences of spatial facts, and inserts them above the line if they are not already present. For example, $p \mapsto_\tau q$ has two pure consequences: isptr p (meaning that p is a pointer value, not an integer or float) and tc_val τ q (that the value q has type τ).
5. For any equations $(x = \ldots)$ or $(\ldots = x)$ above the line, substitute x.
6. Simplify C-language comparisons.
7. Rewriting: the normalize tactic, as explained in Chapter 14.
8. Repeat from step 2, as long as progress is made.
9. Now the proof goal has the form $(Q_1 \ldots * Q_m) \vdash !!(P'_1 \wedge \ldots \wedge P'_{n'}) \&\& (Q'_1 \ldots * Q'_{m'})$. Any of the P'_i provable by auto are removed. If $Q_1 * \ldots * Q_m \vdash Q'_1 * \ldots * Q'_{m'}$ is trivially proved, then the entire $\&\& Q'_1 * \ldots * Q'_{m'}$ is removed.

AT THIS POINT the entailment may have been solved entirely. Or there may be some remaining P'_i and/or Q'_i proof goals on the right hand side.

60 The Hoare triple

(See PLCC Chapter 24)

In the 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 giving *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.

The *basic* VST separation logic is specified in vst/veric/SeparationLogic.v, and contains rules such as,

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

```
Axiom semax_set_forward: ∀ Δ P (x: ident) (e: expr),
  semax Δ (▷ (local (tc_expr Δ e) &&
                    local (tc_temp_id id (typeof e) Δ e) && P))
    (Sset x e)
    (normal_ret_assert
      (EX old:val,
        local (`eq (eval_id x) (subst x (`old) (eval_expr e))) &&
        subst x (`old) P)).
```

However, most C-program verifications will not use the *basic* rules, but will use derived rules whose preconditions are in canonical (PROP/LOCAL/SEP) form. Furthermore, program verifications do not even use the derived rules directly, but use *symbolic execution tactics* that choose which derived rules to apply. So we will not show the rules here; we describe how to use the tactical system.

61 Later

(See PLCC Chapter 15)

Many of the Hoare rules, such as the one on page 101,

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

have the operater $\triangleright$ (pronounced "later") in their precondition.

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.

62 Specifying a function (See PLCC Chapter 27)

Let F be a C-language function, $t_{\rm ret}\ F\ (t_1 x_1,\ t_2 x_2,\ \ldots t_n x_n)$ { ... }. The formal parameters are $\vec{x} : \vec{t}$ (that is, $x_1 : t_1, x_2 : t_2, \ldots x_n : t_n$) and the return type is $t_{\rm ret}$.

Specify F with precondition $P(\vec{a} : \vec{\tau})(\vec{x} : \vec{t})$ and postcondition $Q(\vec{a} : \vec{\tau})(retval)$ where $\vec{a}$ are logical variables that both the precondition and the postcondition can refer to.

The x_i are *C-language variable identifiers*, and the t_i are *C-language types* (tint, tfloat, tptr(tint), etc.). The a_i are *Coq variables* and the τ_i are *Coq types*.

Definition F_spec :=
 DECLARE _F
  WITH $a_1 : \tau_1, \ldots\ a_k : \tau_k$
  PRE [x_1 OF t_1, ... , x_n OF t_n] P
  POST [$t_{\rm ret}$] Q.

Example: for a C function, int sumlist (struct list *p);

```
Definition sumlist_spec :=
 DECLARE _sumlist
  WITH sh : share, contents : list int, p: val,
  PRE [ _p OF (tptr t_struct_list)]
        local (`(eq p) (eval_id _p))
        && `(lseg LS sh contents p nullval)
  POST [ tint ]
        local (`(eq (Vint (sum_int contents))) retval)
        && `(lseg LS sh contents p nullval).
```

The specification itself is an object of type ident∗funspec, and in some cases it can be useful to define the components separately:

Definition sumlist_funspec : funspec :=

```
  WITH sh : share, contents : list int, p: val,
  PRE [ _p OF (tptr t_struct_list)]
        local (`(eq p) (eval_id _p))
        && `(lseg LS sh contents p nullval)
  POST [ tint ]
        local (`(eq (Vint (sum_int contents))) retval)
        && `(lseg LS sh contents p nullval).

Definition sumlist_spec : ident*funspec :=
  DECLARE _sumlist sumlist_funspec.
```

The precondition may be written in *simple form*, as shown above, or in *canonical form*:

```
Definition sumlist_spec :=
 DECLARE _sumlist
  WITH sh : share, contents : list int, p: val,
  PRE [ _p OF (tptr t_struct_list)]
        PROP() LOCAL(`(eq p) (eval_id _p))
        SEP(`(lseg LS sh contents p nullval))
  POST [ tint ]
        local (`(eq (Vint (sum_int contents))) retval)
        && `(lseg LS sh contents p nullval).
```

At present, postconditions may not use PROP/LOCAL/SEP form.

63 Specifying all functions (PLCC Chapter 27)

We give each function a *specification*, typically using the DECLARE/WITH/PRE/POST notation. Then we combine these together into a *global specification*:

Γ : list (ident*funspec) := (ι_1, ϕ_1) :: (ι_2, ϕ_2) :: (ι_3, ϕ_3) :: (ι_4, ϕ_4) :: nil.

We also make a *global variables type specification*, listing the types of all extern global variables:

V : list (ident*type) := (x_1, t_1) :: (x_2, t_2) :: nil

The *initialization values* of extern globals are not part of V, as (generally) they are not invariant over program execution—global variables can be updated by storing into them. Initializers are accessible in the precondition to the _main function.

C-language functions can call each other, and themselves, and access global variables. Correctness proofs of individual functions can take advantage of the specifications of all global functions and types of global variables. Thus we construct Γ and V before proving correctness of any functions.

The next step (in a program proof) is to prove correctness of each function. For each function F in a C program, CompCert clightgen produces _F : ident. f_F: function. where function is a record telling the parameters and locals (and their types) and the function body. The predicate semax_body states that F meets its specification; for each F we must prove:

Lemma body_F: semax_body V Γ f_F F_spec.

64 Proving a function

(See PLCC Chapter 27)

Lemma body_F: semax_body V Γ f_F F_spec.
Proof.
start_function.
name x _x.
name y _y.
name z _z.

Then, for each function parameter and nonaddressable local variable (scalar local variable whose address is never taken), we write a name declaration; in each case, _x is the identifier definition that clightgen has created from the source-language name, and x is the Coq name that we wish to use for the *value* of variable _x at various points. The only purpose of the name tactic is to assist the go_lower tactic in choosing nice names.

At this point the proof goal will be a judgment of the form,

$$\mathsf{semax}\ \ \Delta\ \ (\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R}))\ \ c\ \ \mathit{Post}.$$

We prove such judgments as follows:

1. Manipulate the precondition $\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R})$ until it takes a form suitable for forward symbolic execution through the first statement in the command c. (In this we are effectively using the rule of consequence.)
2. Apply a forward tactic to step into c. This will produce zero or more entailments $A \vdash B$ to prove, where A is in canonical form; and zero or more semax judgments to prove.
3. Prove the entailments, typically using go_lower; prove the judgment, i.e., back to step 1.

Each kind of C command has different requirements on the form of the precondition, for the forward tactic to succeed. In each of the following cases, the expression E must not contain loads, stores, side effects, function calls, or pointer comparisons. 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). Expression E must not contain pointer-comparisons. 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. The SEP component of the precondition must contain an item of the form `(mapsto sh t) e v, where e is equivalent to (eval_expr E). For example, if E is just an identifer (Etempvar _y t), then e could be either (eval_expr (Etempvar _y t)) or (eval_id _y).

x= a[E]; Array load. This is just a memory load, equivalent to x= *(a+E);.

x= E→*fld*; Field load. This is equivalent to x= ∗(E.*fld*) and can actually be handled by the "memory load" case, but a special-purpose field-load rule is easier to use (and will be automatically applied by the forward tactic). In this case the SEP component of the precondition must contain `(field_at sh t *fld*) v e, where t is the structure type to which the field *fld* belongs, and e is equivalent to (eval_expr E).

*E_1 = E_2; Memory store. The SEP component of the precondition must contain an item of the form `(mapsto $sh$ t) $e_1$ $v$ or an item `(mapsto_sh t) e_1, where e_1 is equivalent to (eval_expr E_1).

a[E_1]=E_2; Array store. This is equivalent to *(a+E_1)=E_2; and is handled by the previous case.

E_1 → *fld* = E_2; Field store. This can be handled by the general store case, but a special-purpose field-store rule is easier to use. The SEP component of the precondition must contain either `(field_at $sh$ $t$ *fld*) $v$ $e_1$ or `(field_mapsto_ sh t *fld*) e_1, where t is the structure type to which the field *fld* belongs, and e_1 is equivalent to (eval_expr E_1). The share sh must be strong enough to grant write permission, that is, writable_share(sh).

$x = E_1\ op\ E_2$; If E_1 or E_2 evaluate to *pointers*, and *op* is a comparison operator (=, !=, <, ≤, >, ≥), then $E_1\ op\ E_2$ must not occur except in this special-case assignment rule. When E_1 and E_2 both have numeric values, the ordinary *assignment statement* rule applies.

Pointer comparisons are tricky in CompCert C for reasons explained at PLCC page 249; the program logic uses the semax_ptr_compare rule (PLCC page 164). After applying the forward tactic, the user will be left with some proof obligations: Prove that both E_1 and E_2 evaluate to allocated locations (i.e., that the precondition implies $E_1 \overset{sh_1}{\mapsto} \_ * TT$ and also implies $E_2 \overset{sh_2}{\mapsto} \_ * TT$, for any sh_1 and sh_2). If the comparison is any of $>, <, \geq, \leq$, prove that E_1 and E_2 both point within the same allocated object. These are preconditions for even being permitted to test the pointers for equality (or inequality). See also page 112.

if (E) C_1 **else** C_2 No restrictions on the form of the precondition. forward will create 3 subgoals: (1) prove that the precondition entails tc_expr Δ E. For many expressions E, the condition tc_expr Δ E is simply TT, which is trivial to prove. (2) the **then** clause... (3) the **else** clause... .

while (E) C For a while-loop, use the forward_while tactic (page **??**).

return E; No special precondition, except that 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.

65 Manipulating preconditions

In some cases you cannot go forward until the precondition has a certain form. For example, in ordinary separation logic we might have $\{p \neq q \wedge p \leadsto q\}\, x := p \rightarrow \mathit{tail}\, \{\mathit{Post}\}$. In order to use the proof rule for load, we must use the rule of consequence, to prove,

$$p \neq q \wedge p \leadsto q \;\vdash\; p \neq q \wedge \exists h,t.\, p \mapsto (h,t) * t \leadsto q$$

then instantiate the existentials; this finally gives us

$$\{p \neq q \wedge p \mapsto (h,t) * t \leadsto q\}\, x := p \rightarrow \mathit{tail}\, \{\mathit{Post}\}$$

which is provable by the standard load rule of separation logic.

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 $\vec{P};\vec{Q};\vec{R} \vdash \vec{P}';\vec{Q}';\vec{R}'$ using go_lower (page 92).
- Use the rule of consequence implicitly, by using tactics that modify the precondition (and may leave entailments for you to prove).
- Do rewriting in the precondition, either directly by the standard rewrite and change tactics, or by normalize.
- Extract propositions and existentials from the precondition, by using normalize (or by applying the rules extract_exists_pre and semax_extract_PROP).

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 66.

66 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,

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

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

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

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

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

$\mathsf{semax}\ \Delta'\quad ?88\quad c_3\quad \mathit{Post}$

Then frame_SEP 0 2 to retain only $R_0;R_2$.

$\mathsf{semax}\ \Delta\ \mathsf{PROP}(\vec{P})\mathsf{LOCAL}(\vec{Q})\mathsf{SEP}(R_0;R_2)\quad c_1;c_2\quad \ldots$

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.

67 Pointer comparisons

See PLCC Chapter 25

Pointer comparisons can be split into two cases:

1. Comparisons between two expressions that evaluate to be pointers. In this case, both of the pointers must be to *allocated* objects, or the expression will not evaluate
2. Comparisons between an expression that evaluates to the null pointer and any expression that evaluates to a value with a pointer type. This expression will always evaluate

If you are sure that your pointer comparison falls into the first case, you may treat it exactly like any other expression. The proof may eventually generate a side-condition asking you to prove that one of the expressions evaluates to the null pointer. If your pointer comparison might be between two pointers, however, the expression should be factored into its own statement (PLCC page 145).

When you use forward on a pointer comparison you might get a side condition with a disjunction. The left and right sides of the disjunction correspond to the first and second type of comparison above. In simple cases, the tactic can solve the disjunction automatically.

68 Structured data

The C programming language has struct and array to represent structured data. The *Verifiable C* logic provides operators field_at, array_at, and data_at to describe assertions about structs and arrays.

Given a struct definition, struct list {int head; struct list *tail;}; the clightgen utility produces the type t_struct_list describing fields head and tail. Then these assertions are all equivalent:

mapsto *sh* tint p h *
    mapsto *sh* (tptr t_struct_list) (offset_val p (Vint (Int.repr 4))) t

field_at *sh* t_struct_list [_head] h p * field_at *sh* t_struct_list [_tail] t p

data_at *sh* t_struct_list (h,t) p

field_at *sh* t_struct_list nil (h,t) p

The version using mapsto is correct (assuming a 32-bit configuration of CompCert) but rather ugly; the second version is useful when you want to "frame out" a particular field; the third version describes the contents of all structure-fields at once.

The data_at predicate is dependently typed; the *type* of its third argument (h,t) depends on the *value* of its second argument. The dependent type is expressed by the function, reptype: type → Type that converts C-language types into Coq Types.

Here, reptype t_struct_list = val*val, so the type of (h,t) is (val*val). The value h may be Vint(i) or Vundef, and t may be Vpointer b z, Vint Int.zero, or Vundef. The Vundef values represent uninitialized data fields.

When τ is a struct type and n is a nat, the tactic unfold_data_at n unfolds the nth occurrence of data_at sh τ to a series of field_at sh τ (f::nil), where the f are the various fields of the struct τ. For example, it would unfold the third assertion above to look like the second one.

The forward tactic, when the next command is a load or store command, can operate directly on data_at assertions; it is not necessary to unfold them to individual field_at conjuncts. This is a new feature of VST 1.5.

WHY ARE THE ARGUMENTS BACKWARDS? We write field_at *sh* t_struct_list [head] h p where Reynolds would have written $p.\text{head} \mapsto h$, and we write data_at *sh* t_struct_list (h,t) p where Reynolds would have written $p \hookrightarrow (h,t)$. Putting the *contents* argument before the *pointer* argument makes it easier to express identities in our lifted separation logic. That is, we commonly have formulas such as

`(data_at *sh* t_struct_list (h,t)) (eval_id _p)tptr t_struct_list))

which simplify to

`(field_at *sh* t_struct_list [head] h * field_at *sh* t_struct_list [_tail] t)
    (eval_id _p (tptr t_struct_list))

Expressing these equivalences with the arguments in the other order would lead to extra lambdas, which are (ironically) no fun at all.

PARTIALLY INITIALIZED DATA STRUCTURES. Consider the program

```
struct list *f(void) {
  struct list *p = (struct list *)malloc(sizeof(struct list));
 /* 1 */ p→head= 3;
 /* 2 */ p→tail= NULL;
 /* 3 */ return p;
}
```

We do not want to assume that malloc returns initialized memory, so at point 1 the contents of head and tail are Vundef. We can write this as any of the following:

field_at Tsh t_struct_list [head] Vundef p
        * field_at Tsh t_struct_list [_tail] Vundef p

```
field_at_ Tsh t_struct_list [head] p * field_at_ Tsh t_struct_list [_tail] p
data_at Tsh t_struct_list (Vundef,Vundef) p
data_at_ Tsh t_struct_list p
```

If malloc returns fields that—operationally—contain defined values instead of Vundef, these assertions are still valid, as they ignore the contents of the fields.

At point 2, all the assertions above are still true, but they are weaker than the "appropriate" assertion, which may be written as any of,

```
field_at Tsh t_struct_list [head] (Vint(Int.repr 3)) p
        * field_at Tsh t_struct_list [_tail] Vundef p
field_at Tsh t_struct_list [head] (Vint(Int.repr 3))
        * field_at_ Tsh t_struct_list [_tail] p
data_at Tsh t_struct_list (Vint(Int.repr 3), Vundef) p
```

At point 3, we can write either of,

```
field_at Tsh t_struct_list [head] (Vint(Int.repr 3)) p
        * field_at Tsh t_struct_list [_tail] (Vint Int.zero) p
data_at Tsh t_struct_list (Vint(Int.repr 3), Vint Int.zero) p
```

FULLY INITIALIZED DATA STRUCTURES. In a function precondition it is sometimes convenient to write,

```
WITH data: reptype t_struct_list
PRE [_p OF tptr t_struct_list] (**)
    `(data_at Tsh t_struct_list data) (eval_id _p (tptr t_struct_list))
POST [ ... ] ...
```

If p→head and p→tail may be uninitialized, this is fine. But if the structure is known to be initialized, the precondition as written does not express this fact. One would need to add the conjunct
!!(is_int (fst data) ∧ is_pointer_or_null (snd data))
at the point marked *(**)*.

The function reptype': type → Type expresses the type of *initialized* data structures. For example, reptype' t_struct_list is (int∗val). The function

repinj (t: type): reptype' t → reptype t

expresses injections from (possibly) undefined to defined values. Suppose (h: int, t: val) is a value of type reptype' t_struct_list. Then

repinj t_struct_list (h,t) = (Vint h, t)

Using reptype' one could write,

```
WITH data: reptype' t_struct_list
PRE [_p OF tptr t_struct_list]
    !!(is_pointer_or_null (snd data) &&
    `(data_at Tsh t_struct_list (repinj _data)) (eval_id _p (tptr t_struct_list))
POST [ ... ] ...
```

Notice that this only solves half the problem—for integers but not for pointers. Since defined pointers can be either NULL or a Vpointer, we use val to represent them, and the Coq type alone does not express the refinement. One could imagine a version of reptype' that uses a refinement type to accomplish this, but it might be unwieldy.

69 For loops

The C-language for loop has the general form,

for (*init*; *test*; *incr*) *body*

To solve a proof goal of this form (or when this is followed by other statements in sequence), use the tactic

forward_for *Inv* *PreIncr* *PostCond*

where *Inv, PreIncr, PostCond* are assertions (in PROP/LOCAL/SEP form):

Inv is the loop invariant, that holds immediately after the *init* command is executed and before each time the *test* is done; *PreIncr* is the invarint that holds immediately after the loop *body* and right before the *incr*;

PostCond is the assertion that holds after the loop is complete (whether by a break statement, or the test evaluating to false).

The following feature will appear in VST version 1.5.

Many for-loops have this special form, for ($init$; $id < hi$; id++) *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. Then you can use the tactic,

forward_for_simple_bound n (EX i:Z, 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. The loop invariant is given by the expression (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*. This tactic generates simpler subgoals than the general forward_for tactic.

When the loop has the form, for $(id{=}lo;\ id < hi;\ id{+}{+})\ body$
where lo is a literal constant, then the forward_for_simple_bound tactic will generate slightly simpler subgoals.

70 Nested Loads

This experimental feature will appear in VST release 1.5.

To handle assignment statements with nested loads, such as x[i]=y[i]+z[i]; the recommended method is to break it down into smaller statments compatible with separation logic: t=y[i]; u=z[i]; x[i]=t+u;. However, sometimes you may be proving correctness of preexisting or machine-generated C programs. Verifiable C has an ***experimental*** nested-load mechanism to support this.

We use an expression-evaluation relation $e \Downarrow v$ which comes in two flavors:

rel_expr : expr → val → rho → mpred.
rel_lvalue: expr → val → rho → mpred.

The assertion rel_expr $e\ v\ \rho$ says, "expression e evaluates to value v in environment ρ and in the current memory." The rel_lvalue evaluates the expression as an l-value, to a pointer to the data.

Evaluation rules for rel_expr are listed here:

rel_expr_const_int: $\forall (i : \text{int})\ \tau\ (P : \text{mpred})\ (\rho : \text{environ})$,
  $P \vdash$ rel_expr (Econst_int $i\ \tau$) (Vint i) ρ.
rel_expr_const_float: $\forall (f : \text{float})\ \tau\ P\ (\rho : \text{environ})$,
  $P \vdash$ rel_expr (Econst_float $f\ \tau$) (Vfloat f) ρ.
rel_expr_const_long: $\forall (i : \text{int64})\ \tau\ P\ \rho$,
  $P \vdash$ rel_expr (Econst_long $i\ \tau$) (Vlong i) ρ.
rel_expr_tempvar: $\forall (\text{id} : \text{ident})\ \tau\ (v : \text{val})\ P\ \rho$,
  Map.get (te_**of** ρ) id = Some v →
  $P \vdash$ rel_expr (Etempvar id τ) $v\ \rho$.
rel_expr_addrof: $\forall (e : \text{expr})\ \tau\ (v : \text{val})\ P\ \rho$,
  $P \vdash$ rel_lvalue $e\ v\ \rho$ →
  $P \vdash$ rel_expr (Eaddrof $e\ \tau$) $v\ \rho$.
rel_expr_unop: $\forall P\ (e_1 : \text{expr})\ (v_1\ v : \text{val})\ \tau\ op\ \rho$,
  $P \vdash$ rel_expr $e_1\ v_1\ \rho$ →

Cop.sem_unary_operation op v_1 (typeof e_1) = Some v →
P ⊢rel_expr (Eunop op e_1 τ) v ρ.
rel_expr_binop: ∀(e_1 e_2 : expr) (v_1 v_2 v : val) τ op P ρ,
P ⊢rel_expr e_1 v_1 ρ →
P ⊢rel_expr e_2 v_2 ρ →
(∀ m : Memory.Mem.mem,
Cop.sem_binary_operation op v_1 e (typeof e_1) v_2 (typeof e_2) m = Some v) →
P ⊢rel_expr (Ebinop op e_1 e_2 τ) v ρ.
rel_expr_cast: ∀(e_1 : expr) (v_1 v : val) τ P ρ,
P ⊢rel_expr e_1 v_1 ρ →
Cop.sem_cast v_1 (typeof e_1) τ = Some v →
P ⊢rel_expr (Ecast e_1 τ) v ρ.
rel_expr_lvalue: ∀(a : expr) (sh : Share.t) (v_1 v_2 : val) P ρ,
P ⊢rel_lvalue a v_1 ρ →
P ⊢mapsto sh (typeof a) v_1 v_2 * TT →
v_2 <> Vundef →
P ⊢rel_expr a v_2 ρ.
rel_lvalue_local: ∀(id : ident) τ (b : block) P ρ,
P ⊢!!(Map.get (ve_**of** ρ) id = Some (b, τ)) →
P ⊢rel_lvalue (Evar id τ) (Vptr b Int.zero) ρ.
rel_lvalue_global: ∀(id : ident) τ (v : val) P ρ,
P
⊢!!(Map.get (ve_**of** ρ) id = None ∧
Map.get (ge_**of** ρ) id = Some (v, τ)) →
P ⊢rel_lvalue (Evar id τ) v ρ.
rel_lvalue_deref: ∀(a : expr) (b : block) (z : int) τ P ρ,
P ⊢rel_expr a (Vptr b z) ρ →
P ⊢rel_lvalue (Ederef a τ) (Vptr b z) ρ.
rel_lvalue_field_struct: ∀(i id : ident) τ e (b : block) (z : int) (fList : fieldlist) att (
typeof e = Tstruct id fList att →
field_offset i fList = Errors.OK δ →
P ⊢rel_expr e (Vptr b z) ρ →
P ⊢rel_lvalue (Efield e i τ) (Vptr b (Int.add z (Int.repr δ))) ρ.

The primitive nested-load assignment rule is,

```
Axiom semax_loadstore:
 ∀v0 v1 v2 Δ e1 e2 sh P P',
   writable_share sh →
   P ⊢!! (tc_val (typeof e1) v2)
           && rel_lvalue e1 v1
           && rel_expr (Ecast e2 (typeof e1)) v2
           && (`(mapsto sh (typeof e1) v1 v0) ∗ P') →
   semax Δ (▷ P) (Sassign e1 e2)
          (normal_ret_assert (`(mapsto sh (typeof e1) v1 v2) ∗ P')).
```

but do not use this rule! It is best to use a derived rule, such as,

```
Lemma semax_loadstore_array:
 ∀n vi lo hi t1 (contents: Z →reptype t1) v1 v2 Δ e1 ei e2 sh P Q R,
  reptype t1 = val →
  type_is_by_value t1 →
  legal_alignas_type t1 = true →
  typeof e1 = tptr t1 →
  typeof ei = tint →
  PROPx P (LOCALx Q (SEPx R))
     ⊢rel_expr e1 v1
       && rel_expr ei (Vint (Int.repr vi))
       && rel_expr (Ecast e2 t1) v2 →
  nth_error R n = Some (`(array_at t1 sh contents lo hi v1)) →
  writable_share sh →
  tc_val t1 v2 →
  in_range lo hi vi →
  semax Δ (▷ PROPx P (LOCALx Q (SEPx R)))
   (Sassign (Ederef (Ebinop Oadd e1 ei (tptr t1)) t1) e2)
   (normal_ret_assert
    (PROPx P (LOCALx Q (SEPx
     (replace_nth n R
      `(array_at t1 sh (upd contents vi (valinject _ v2)) lo hi v1)))))).
```

Proof-automation support is available for semax_loadstore_array and rel_expr, in the form of the forward_nl (for "forward nested loads") tactic. For example, with this proof goal,

```
semax Delta
 (PROP ()
  LOCAL(`(eq (Vint (Int.repr i))) (eval_id _i); `(eq x) (eval_id _x);
  `(eq y) (eval_id _y); `(eq z) (eval_id _z))
  SEP(`(array_at tdouble Tsh (Vfloat oo fx) 0 n x);
  `(array_at tdouble Tsh (Vfloat oo fy) 0 n y);
  `(array_at tdouble Tsh (Vfloat oo fz) 0 n z)))
 (Ssequence
  (Sassign (* x[i] = y[i] + z[i]; *)
   (Ederef (Ebinop Oadd (Etempvar _x (tptr tdouble)) (Etempvar _i tint)
             (tptr tdouble)) tdouble)
    (Ebinop Oadd
     (Ederef (Ebinop Oadd (Etempvar _y (tptr tdouble)) (Etempvar _i tint)
               (tptr tdouble)) tdouble)
     (Ederef (Ebinop Oadd (Etempvar _z (tptr tdouble)) (Etempvar _i tint)
               (tptr tdouble)) tdouble) tdouble))
   MORE_COMMANDS)
 POSTCONDITION
```

the tactic-application forward_nl yields the new proof goal,

```
semax Delta
  (PROP ()
   LOCAL(`(eq (Vint (Int.repr i))) (eval_id _i); `(eq x) (eval_id _x);
   `(eq y) (eval_id _y); `(eq z) (eval_id _z))
   SEP
   (`(array_at tdouble Tsh
        (upd (Vfloat oo fx) i (Vfloat (Float.add (fy i) (fz i)))) 0 n x);
   `(array_at tdouble Tsh (Vfloat oo fy) 0 n y);
   `(array_at tdouble Tsh (Vfloat oo fz) 0 n z)))
  MORE_COMMANDS
  POSTCONDITION
```

71 JUNK

$$\text{makelock}\ \frac{R \text{ positive} \qquad R \text{ precise}}{\Delta \vdash \{l \mapsto 0\}\, \mathsf{makelock}\, l\, \{l \overset{\bullet}{\boxdot\!\!\rightarrow} R\}}$$

$$\text{freelock}\ \frac{}{\Delta \vdash \{R * l \overset{\bullet}{\boxdot\!\!\rightarrow} R\}\, \mathsf{freelock}\, l\, \{R * l \mapsto 0\}}$$

$$\text{splitlock}\ \frac{\pi_1 \oplus \pi_2 = \pi}{l \overset{\pi_1}{\boxdot\!\!\rightarrow} R \ * \ l \overset{\pi_2}{\boxdot\!\!\rightarrow} R \ \leftrightarrow \ l \overset{\pi}{\boxdot\!\!\rightarrow} R}$$

$$\text{acq}\ \frac{}{\Delta \vdash \{l \overset{\pi}{\boxdot\!\!\rightarrow} R\}\ \mathsf{acquire}\, l\ \{R * l \overset{\pi}{\boxdot\!\!\rightarrow} R\}}$$

$$\text{rel}\ \frac{}{\Delta \vdash \{R * l \overset{\pi}{\boxdot\!\!\rightarrow} R\}\ \mathsf{release}\, l\ \{l \overset{\pi}{\boxdot\!\!\rightarrow} R\}}$$

$$\text{spawn}\ \frac{}{\Delta \vdash \{(a : \{\vec{y} P\}\{\mathsf{emp}\}) \wedge (P\,x)[\vec{b}/\vec{y}] * F\}\ \mathsf{spawn}\ a(\vec{b})\ \{F\}}$$

$$\text{exit}\ \frac{}{\Delta \vdash \{\mathsf{emp}\}\ \mathsf{exit}()\ \{\bot\}}$$

Figure 71.1: CSL rules for threads and locks