Verifiable C

Applying the Verified Software Toolchain to C programs

Version 2.0
January 16, 2018

Andrew W. Appel
with Lennart Beringer, Qinxiang Cao, Josiah Dodds
# Contents

Verifiable C

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>ii</td>
</tr>
<tr>
<td>1 Overview</td>
<td>5</td>
</tr>
<tr>
<td>2 Installation</td>
<td>7</td>
</tr>
<tr>
<td>3 Workflow, loadpaths</td>
<td>8</td>
</tr>
<tr>
<td>4 Verifiable C and clightgen</td>
<td>9</td>
</tr>
<tr>
<td>5 ASTs: abstract syntax trees</td>
<td>10</td>
</tr>
<tr>
<td>6 Use the IDE</td>
<td>12</td>
</tr>
<tr>
<td>7 Functional model, API spec</td>
<td>13</td>
</tr>
<tr>
<td>8 Proof of the sumarray program</td>
<td>17</td>
</tr>
<tr>
<td>9 start_function</td>
<td>18</td>
</tr>
<tr>
<td>10 forward</td>
<td>21</td>
</tr>
<tr>
<td>11 If, While, For</td>
<td>24</td>
</tr>
<tr>
<td>12 While loops</td>
<td>25</td>
</tr>
<tr>
<td>13 PROP() LOCAL() SEP()</td>
<td>27</td>
</tr>
<tr>
<td>14 Entailments</td>
<td>28</td>
</tr>
<tr>
<td>15 Array subscripts</td>
<td>30</td>
</tr>
<tr>
<td>16 At the end of the loop body</td>
<td>32</td>
</tr>
<tr>
<td>17 Returning from a function</td>
<td>34</td>
</tr>
<tr>
<td>18 Global variables and main()</td>
<td>35</td>
</tr>
<tr>
<td>19 Function calls</td>
<td>37</td>
</tr>
<tr>
<td>20 Tying all the functions together</td>
<td>38</td>
</tr>
<tr>
<td>21 Separation logic: EX, *, emp, !!</td>
<td>39</td>
</tr>
<tr>
<td>22 EX, Intros, Exists</td>
<td>40</td>
</tr>
<tr>
<td>23 Integers: nat, Z, int</td>
<td>42</td>
</tr>
<tr>
<td>24 Int, Int8, Int16, Int64, Ptrofs</td>
<td>44</td>
</tr>
<tr>
<td>25 Values: Vint, Vptr</td>
<td>45</td>
</tr>
<tr>
<td>26 C types</td>
<td>46</td>
</tr>
<tr>
<td>27 CompSpecs</td>
<td>47</td>
</tr>
<tr>
<td>28 retype</td>
<td>48</td>
</tr>
<tr>
<td>29 Uninitialized data, default_val</td>
<td>49</td>
</tr>
</tbody>
</table>

Copyright © 2018 Andrew W. Appel
data_at
field_at
reptype', repinj
LOCAL defs: temp, lvar, gvar
go_lower
saturate_local
field_compatible, field_address
value_fits
cancel
entailer!
normalize
assert_PROP
Welltypedness of variables
Shares
Pointer comparisons
Proof of the reverse program
Alternate proof of reverse
Global variables
For loops (special case)
For loops (general case)
Manipulating preconditions
The Frame rule
32-bit Integers
CompCert C abstract syntax
C light semantics
Splitting arrays
sublist
Later
Mapsto and func_ptr
with_library: Library functions
malloc/free
exit


<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>62</td>
<td>Function pointers</td>
<td>98</td>
</tr>
<tr>
<td>63</td>
<td>Axioms of separation logic</td>
<td>100</td>
</tr>
<tr>
<td>64</td>
<td>Obscure higher-order axioms</td>
<td>101</td>
</tr>
<tr>
<td>65</td>
<td>Proving larg(ish) programs</td>
<td>102</td>
</tr>
<tr>
<td>66</td>
<td>Separate compilation, semax_ext</td>
<td>104</td>
</tr>
<tr>
<td>67</td>
<td>Catalog of tactics/lemmas</td>
<td>105</td>
</tr>
</tbody>
</table>
1 Overview

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

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

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

This soundness proof comes in two parts: The program logic is proved sound with respect to the semantics of CompCert C, by a team of researchers primarily at Princeton University; and the C compiler is proved correct with respect to those same semantics, by a team of researchers primarily at INRIA. This chain of proofs from top to bottom, connected in Coq at specification interfaces, is part of the *Verified Software Toolchain*. 
To use Verifiable C, one must have had some experience using Coq, and some familiarity with the basic principles of Hoare logic. These can be obtained by studying Pierce’s *Software Foundations* interactive textbook, and doing the exercises all the way to chapter “Hoare2.”

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

*Program Logics for Certified Compilers* (Cambridge University Press, 2014) describes *Verifiable C* version 1.1. If you are interested in the semantic model, soundness proof, or memory model of VST, the book is well worth reading. But it is not a reference manual.

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

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

Coq 8.7.1, from coq.inria.fr. Follow the standard installation instructions.

CompCert 3.2, from http://compcert.inria.fr/download.html. Build the clightgen tool, using these commands: ./configure -clightgen x86_32-linux; make. You might replace x86_32-linux with x86_32-macosx or x86_32-cygwin. Verifiable C should work on other 32-bit architectures as well, but has not been extensively tested. Verifiable C has not been tested on 64-bit architectures, but a near-future release will support these.

VST 2.0, from vst.cs.princeton.edu, or else an appropriate version from https://github.com/PrincetonUniversity/VST. After unpacking, read the BUILD_ORGANIZATION file (or simply make -j).

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

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

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

LOAD PATHS. Interactive development environments (CoqIDE or Proof General) will need their load paths properly initialized through command-line arguments. Running make in vst creates a file .loadpath with the right arguments for proof development of the VST itself or of its progs/examples. For example,

cqide `cat .loadpath` progs/verif_reverse.v &

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

Example:

cd my-own-directory
cqide `cat my/path/to/VST/.loadpath-export` myfile.v

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

\(^1\)Named after Robert W. Floyd (1936–2001), a pioneer in program verification.
4 Verifiable C and clightgen

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

- No casting between integers and pointers.
- No goto statements.
- Only structured switch statements (no Duff’s device).

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

Suppose you have a C source program broken into three files x.c y.c z.c.

clightgen -normalize x.c y.c z.c

This produces the files x.v y.v z.v containing Coq representations of ASTs.

Clightgen invokes the standard macro-preprocessor (to handle define and include), parses, type-checks, and produces ASTs. We translate all three files in one call to Clightgen, so that the global names in the C program (“extern” identifiers) will have consistent symbol-table indexes (ident values) across all three files.

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

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

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

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

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

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

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

Definition _main : ident := 54%positive.
Definition _s : ident := 50%positive.
...

Definition f_sumarray := {
    fn_return := tint; ...
    fn_params := ((\a, (tptr tint)) :: (_n, tint) :: nil);
    fn_temps := ((\i, tint) :: (_s, tint) :: (_x, tint) :: nil);
    fn_body :=
        (Ssequence
            (Sset _i (Econst_int (Int.repr 0) tint))
            (Ssequence (Sset _s (Econst_int (Int.repr 0) tint))
                (Ssequence ... )))
    }.

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

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

\[
\text{Inductive type} : \text{Type} := \\
\quad \text{Tvoid} : \text{type} \\
\quad \text{Tint} : \text{intsize} \rightarrow \text{signedness} \rightarrow \text{attr} \rightarrow \text{type} \\
\quad \text{Tpointer} : \text{type} \rightarrow \text{attr} \rightarrow \text{type} \\
\quad \text{Tstruct} : \text{ident} \rightarrow \text{attr} \rightarrow \text{type} \\
\quad \ldots 
\]

See also Chapter 26 (C types).
6 Use the IDE

Chapter 7 through Chapter 20 are meant to be read while you have the file progs/verif_sumarray.v open in a window of your interactive development environment for Coq. You can use Proof General, CoqIDE, or any other IDE that supports Coq.

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

Before starting the IDE, review Chapter 3 (Workflow) to see how to set up load paths.
A program without a specification cannot be incorrect, it can only be surprising. (Paraphrase of J. J. Horning, 1982)

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

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

We start verif_sumarray.v with some standard boilerplate:

```coq
Require Import VST.floyd.proofauto.
Require Import VST.progs.sumarray.
Instance CompSpecs : comspecs. make_comspecs 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 27 and Chapter 47.

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

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

```coq
Definition sum_Z : list Z -> Z := fold_right Z.add 0.
```

A functional model contains not only definitions; it’s also useful to include theorems about this mathematical domain:
Lemma sum\_Z\_app: \( \forall a \ b, \text{sum}\_Z (a++b) = \text{sum}\_Z a + \text{sum}\_Z b. \)

Proof. intros. induction a; simpl; omega. Qed.

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

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

**Definition** sumarray\_spec : ident * funspec :=
DECLARE _sumarray
WITH a: val, sh : share, contents : list Z, size: Z
PRE [ _a OF (tptr tuint), _n OF tint ]
PROP(readable\_share sh;
  0 \leq size \leq \text{Int.max\_signed};
  \text{Forall (fun x \Rightarrow 0 \leq x \leq \text{Int.max}\_\text{unsigned}) contents})
LOCAL(temp _a a; temp _n (Vint (Int.repr size)))
SEP(data\_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a)
POST [ tuint ]
PROP()
LOCAL(temp ret\_temp (Vint (Int.repr (sum\_Z contents))))
SEP(data\_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a).

The funspec begins, **Definition** f\_spec := DECLARE _f ... where \( f \) is the name of the C function, and \( _f : \text{ident} \) is Coq’s name for the identifier that denotes \( f \) in the AST of the C program (see page 10).
A function is specified by its *precondition* and its *postcondition*. The WITH clause quantifies over Coq values that may appear in both the precondition and the postcondition. The precondition is parameterized by the C-language function parameters, and the postcondition is parameterized by a identifier ret_temp, which is short for “the temporary variable holding the return value.”

Function preconditions, postconditions, and loop invariants are *assertions* about the state of variables and memory at a particular program point. In an assertion `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 true independent of program state. In the function precondition above, the statement `0 <= size <= Int.max_signed` is true *just within the scope of the quantification of the variable size*; it is bound by `WITH`, and spans the PRE and POST assertions.

The LOCAL propositions \( \vec{Q} \) are *variable bindings* of type `localdef`. Here, the function-parameters \( a \) and \( n \) are treated as nonaddressable local variables, or “temp” variables. The `localdef (temp .a a)` says that (in this program state) the contents of C local variable .a is the Coq value a. In general, a C scalar variable holds something of type `val`; this type is defined by CompCert as,

```plaintext
Inductive val: Type := Vundef: val | Vint: int -> val | Vlong: int64 -> val
```

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

The *postcondition* is introduced by `POST [ tuint ]`, indicating that this function returns a value of type `unsigned int`. There are no PROP statements in this postcondition—no forever-true facts hold now, that
weren’t already true on entry to the function. The LOCAL must not mention the function parameters or local variables, because they are destroyed on function exit (and because your local variable names will not be meaningful to the function’s caller); 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 \( \text{sum_int}(contents) \), and leaves the array unaltered.”

**INTEGER OVERFLOW.** In Verifiable C’s signed integer arithmetic, you must prove (if the system cannot prove automatically) that no overflow occurs. In unsigned integers, arithmetic is treated as modulo-\( 2^n \) (where \( n \) is typically 32 or 64), and overflow is not an issue. See Chapter 23.

The function \( \text{Int.repr}: \mathbb{Z} \rightarrow \text{int} \) truncates mathematical integers into 32-bit integers by taking the (sign-extended) low-order 32 bits. \( \text{Int.signed}: \text{int} \rightarrow \mathbb{Z} \) injects back into the signed integers.

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

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

\[ \text{Int.unsigned}(\text{Int.repr}(\text{sum}_Z \ contents)) \neq (\text{sum}_Z \ contents). \]

In general, for a claim about \( \text{Int.repr}(x) \) to be useful, one also needs a claim that \( 0 \leq x \leq \text{Int.max}_\text{unsigned} \) or \( \text{Int.min}_\text{signed} \leq x \leq \text{Int.max}_\text{signed} \). The caller of this function will probably need to prove \( 0 \leq \text{sum}_Z \ contents \leq \text{Int.max}_\text{unsigned} \) in order to make much use of the postcondition.
8 Proof of the sumarray program

To prove correctness of a whole program,

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

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

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

The function specs, built using DECLARE, are listed in the argument to with_library. Chapter 59 describes with_library.

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

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

Lemma body_sumarray: semax_body Vprog Gprog f_sumarray sumarray_spec.

Here, f_sumarray is the actual function body (AST of the C code) as parsed by clightgen; you can read it in sumarray.v. You can read body_sumarray as saying, In the context of Vprog and Gprog, the function body f_sumarray satisfies its specification sumarray_spec. We need the context in case the sumarray function refers to a global variable (Vprog provides the variable’s type) or calls a global function (Gprog provides the function’s API spec).
The predicate `semax_body` states the Hoare triple of the function body, \( \Delta \vdash \{ \text{Pre} \} c \{ \text{Post} \} \). \text{Pre} and \text{Post} are taken from the funspec for \( f \), \( c \) is the body of \( F \), and the type-context \( \Delta \) is calculated from the global type-context overlaid with the parameter- and local-types of the function.

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

**Lemma** body_sumarray: `semax_body Vprog Gprog f_sumarray sumarray_spec`.

**Proof.**

`start_function`.

The proof goal now looks like this:

```
Espec : OracleKind
a : val
sh : share
contents : list Z
size : Z
Delta_specs := abbreviate : PTree.t funspec
Delta := abbreviate : tycontext
SH : readable_share sh
H : 0 ≤ size ≤ Int.max_signed
H0 : Forall (fun x : Z ⇒ 0 ≤ x ≤ Int.max_unsigned) contents
POSTCONDITION := abbreviate : ret_assert
MORE_COMMANDS := abbreviate : statement
---------------------------------------------------------------------(1/1)
semax Delta
(PROP ()
  LOCAL(temp _a a; temp _n (Vint (Int.repr size)))
  SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a))
(Ssequence (Sset_i (Econst_int (Int.repr 0) tint)) MORE_COMMANDS)
POSTCONDITION
```
First we have *Espec*, which you can ignore for now (it characterizes the outside world, but sumarray.c does not do any I/O). Then a,sh,contents,size are exactly the variables of the WITH clause of sumarray_spec.

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

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

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

$$\text{semax} (\Delta: \text{tycontext}) (P: \text{environ} \rightarrow \text{mpred}) (c: \text{statement}) (R: \text{ret_assert}).$$

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

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

Because we do *forward* Hoare-logic proof, we won’t care about the postcondition until we get to the end of $c$, so here we hide it away in an abbreviation. Here, the command $c$ is a long sequence starting with $i=0;\ldots more$, and we hide the *more* in an abbreviation MORE_COMMANDS.
The precondition of this semax has LOCAL and SEP parts taken directly from the funspec (the PROP clauses have been moved above the line). The statement \((\text{Sset} .i (\text{Econst}_{\text{int}} (\text{Int}.\text{repr} 0) \text{ tint}))\) is the AST generated by clightgen from the C statement \(i=0;\).
We do Hoare logic proof by forward symbolic execution. On page 18 we show the proof goal at the beginning of the sumarray function body. In a forward Hoare logic proof of \( \{P\} i = 0; more \{R\} \) we might first apply the sequence rule,

\[
\begin{align*}
\{P\} i = 0; \{Q\} & \quad \{Q\} more \{R\} \\
\{P\} i = 0; more \{R\} & 
\end{align*}
\]

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

\[
\begin{align*}
\text{Espec, a, sh, contents, size, Delta_spec, SH, H, H0 as before} \\
\text{Delta} := \text{abbreviate : tycontext} \\
\text{POSTCONDITION} := \text{abbreviate : ret_assert} \\
\text{MORE_COMMANDS} := \text{abbreviate : statement} \\
\end{align*}
\]

\[
\text{semex Delta} \\
\text{(PROP (}) \\
\text{LOCAL(temp}_i (\text{Vint (Int.repr 0))}; \text{temp}_a a; \\
\text{temp}_n (\text{Vint (Int.repr size)))} \\
\text{SEP(data_at sh (tarray uint size) (map Vint (map Int.repr contents)) a))} \\
\text{(Ssequence (Sset}_s (\text{Econst_int (Int.repr 0) uint}) \text{MORE_COMMANDS})} \\
\text{POSTCONDITION}
\]

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

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

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

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

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

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

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

$x = \ast E;$ Memory load.

$x = \text{a}[E];$ Array load.

$x = E\rightarrow\text{fld};$ Field load.

$x = E\rightarrow f_1.f_2;$ Nested field load; see Chapter 31.

$x = E\rightarrow f_1[i].f_2;$ Fields and subscripts; see Chapter 31.

$E_1 = E_2;$ Memory store. Expression $E_2$ must not dereference memory.  
Expression $E_1$ must be equivalent to a single memory store via some access path (see Chapter 31), and the precondition must contain an appropriate storable data_at or field_at.

**if** $(E) C_1 \text{ else } C_2$ For an if-statement, use forward_if and (perhaps) provide a postcondition.

**while** $(E) C$ For a while-loop, use the forward_while tactic (page 25) and provide a loop invariant.

**break;** The forward tactic works.

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

$x = f(a_1,\ldots,a_n)$; For a function call, use forward_call (see Chapter 19).
11 If, While, For

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

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

- **if** then *s_1* else *s_2*; *s_3*…
  
  Use forward_if *Q*, where *Q* is the *join postcondition*, the precondition of statement *s_3*.

- **if** then *s_1* else *s_2*;}{…
  
  When the if-statement appears at the end of a block, so the postcondition is already known, you can do forward_if. That is, you don’t need to supply a join postcondition if POSTCONDITION is fully instantiated, without any unification variables. You can unfold abbreviate in POSTCONDITION to see.

- **while**(*e*)*s*;… (no break statements in *s*)
  
  You write forward_while *Q*, where *Q* is a loop invariant. See Chapter 12.

- **while**(*e*)*s*;… (with break statements in *s*)
  
  You must treat this as if it were **for**(*e*; *e*;) *s*, and use the forward_for tactic; see below.

- **for**(*e_1*; *i < e_2*; *i + +)* *s* (no break statements in *s*)
  
  If *e_2* is loop-invariant, you may be able to use forward_for_simple_bound; see Chapter 48.

- **for**(*e_1*; *e_2*; *e_3*) *s* (no break statements in *s*)
  
  Use forward_for *Q* *Q’*, where *Q* is the loop invariant and *Q’* is the assertion that goes right before the *increment* command, *e_3*. See Chapter 49.

- **for**(*e_1*; *e_2*; *e_3*) *s*;…
  
  If *s* contains break statements and there are more commands after the loop, you will need to write forward_for *Q* *Q’* *R*, where *Q* and *Q’* are as above, and *R* is the join-postcondition. See Chapter 49.
12 While loops

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

Espec, Delta_specs, Delta
a : val, sh : share, contents : list Z, size : Z
SH : readable_share sh
H : 0 ≤ size ≤ Int.max_signed
H0 : Forall (fun x : Z ⇒ 0 ≤ x ≤ Int.max_unsigned) contents
POSTCONDITION := abbreviate : ret_assert
MORE_COMMANDS, LOOP_BODY := abbreviate : statement

----------------------------------------------------------------(1/1)
semax Delta
(PROP ()
  LOCAL(temp _s (Vint (Int.repr 0)); temp _i (Vint (Int.repr 0));
        temp _a a; temp _n (Vint (Int.repr size)))
  SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a))
(Ssequence
  (Swhile (Ebinop Olt (Etempvar-i tint) (Etempvar-n tint) tint)
    LOOP_BODY)
  MORE_COMMANDS)
POSTCONDITION

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

EX i : Z,
  PROP(0 ≤ i ≤ size)
  LOCAL(temp _a a; temp _i (Vint (Int.repr i)); temp _n (Vint (Int.repr size));
        temp _s (Vint (Int.repr (sum_Z (sublist 0 i contents)))))
  SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a).
The existential binds \( i \), the iteration-dependent value of the local variable named \( _i \). In general, there may be any number of EX quantifiers.

The forward-while tactic will generate four subgoals to be proved:

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

Let’s take a look at that first subgoal:

\[
\begin{align*}
\text{(above-the-line hypotheses elided)} \\
\text{ENTAIL Delta,} \\
\text{PROP(} & \\
\text{LOCAL(} & \\
\text{temp }_s (\text{Vint (Int.repr 0)}); \text{ temp }_i (\text{Vint (Int.repr 0)}); \\
& \text{ temp }_a a; \text{ temp }_n (\text{Vint (Int.repr size)})) \\
\text{SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a)} \\
\text{\[\exists i : Z,} \\
\text{PROP(} & \leq i \leq \text{size}) \\
\text{LOCAL(} & \\
\text{temp }_a a; \text{ temp }_i (\text{Vint (Int.repr } i)); \\
& \text{ temp }_n (\text{Vint (Int.repr size)}); \\
& \text{ temp }_s (\text{Vint (Int.repr } (\text{sum}_Z (\text{sublist 0 } i \text{ contents}))))) \\
\text{SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a)}
\end{align*}
\]

This is an entailment goal; Chapter 14 shows how to prove such goals.
Each element of a SEP clause is a spatial predicate, that is, a predicate on some part of the memory. The Coq type for a spatial predicate is mpred; it can be thought of as mem→Prop (but is not quite the same, for quite technical semantic reasons).

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

The LOCAL part of an assertion is a local predicate, that is, a predicate about the values of local variables. It’s an ordinary conjunction (not separating) of its individual temp and lvar clauses. You can think of a local predicate as Coq type environ→Prop, where environ is the type of run-time local-variable frames.

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

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

The existential quantifier EX can also be used on canonical forms, e.g., EX x:T, PROP(⃗P)LOCAL(⃗Q)SEP(⃗R).
14 Entailments

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

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

Verifiable C's rule of consequence is,

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

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

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

H : 0 ≤ size ≤ Int.max_signed

(other above-the-line hypotheses elided)

$\text{ENTAIL } \Delta,$

PROP()

LOCAL(temp _s (Vint (Int.repr 0)); temp _i (Vint (Int.repr 0));

  temp _a a; temp _n (Vint (Int.repr size)))

SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a)

$\vdash$ 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 tuint size) (map Vint (map Int.repr contents)) a)
We instantiate the existential with the only value that works here, zero: Exists 0. Chapter 22 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 39). In this case, the goal solves entirely automatically. In particular, \( 0 \leq i \leq \text{size} \) 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 \( \Delta, PQR \vdash tc\_expr \Delta e \) where \( e \) is (the abstract syntax of) a C expression; in the particular case of a while loop, \( e \) is the negation of the loop-test expression. The assertion \( tc\_expr \Delta e \) says that executing \( e \) won’t crash: all the variables it references exist and are initialized; and it doesn’t divide by zero, et cetera.

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

ENTAIL Delta, PROP(...) LOCAL(...) SEP(...) \( \vdash tc\_expr \Delta \)
\( \ (\text{Eunop Onotbool (Ebinop Olt (Etempvar \_i tint) (Etempvar \_n tint) tint) tint)} \)

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 \( !!\text{True} \), (equivalent to \( \text{TT} \), the “true” mpred). That’s because the typechecker is calculational, as Chapter 25 of Program Logics for Certified Compilers explains.
THE THIRD SUBGOAL of forward_while in progs/verif_sumarray.v is the body of the while loop: \{x=a[i]; s+=x; i++;\}.

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

Let's examine the proof goal:

\[
\begin{align*}
SH & : \text{readable\_share } sh \\
H & : 0 \leq \text{size} \leq \text{Int.max\_signed} \\
H0 : \text{Forall} \ (\text{fun} \ x : \text{Z} \Rightarrow 0 \leq x \leq \text{Int.max\_unsigned}) \ \text{contents} \\
i & : \text{Z} \\
HRE : i < \text{size} \\
H1 : 0 \leq i \leq \text{size}
\end{align*}
\]

\[
\begin{align*}
\text{semex Delta} \\
\text{(PROP ()} \\
\text{LOCAL(temp .a a; temp .i (Vint (Int.repr i)));} \\
\text{temp .n (Vint (Int.repr size));} \\
\text{temp .s (Vint (Int.repr (sum-Z (sublist 0 i contents))))}) \\
\text{SEP(data\_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a))} \\
\text{(Ssequence} \\
\text{Sset \_x} \\
\text{(Ederef} \\
\text{(Ebinop Oadd (Etempvar \_a (tptr tuint)) (Etempvar \_i tint) (tptr tuint)) tuint)) \text{MORE\_COMMANDS} \text{POSTCONDITION}
\end{align*}
\]

The Coq variable \(i\) was introduced automatically by forward_while from the existential variable, the EX \(i:Z\) of the loop invariant.
Going forward through \( x = a[i] \); will be enabled by the `data_at` in the precondition, as long as the subscript value is less than the length of contents. One important property of `data_at` in \( \pi \) \( (\text{tarray } \tau \ n) \) \( \sigma \) \( p \) is that \( n = Z\text{length}(\sigma) \). If we had that fact above the line, then (using assumptions \( \text{HRE} \) and \( \text{H} \)) it would be easy to prove \( 0 \leq i < Z\text{length} \) contents.

Therefore, we write,

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

Chapter 41 describes `assert_PROP`, which (like Coq's standard `assert`) will put \( Z\text{length} \) contents=\( \text{size} \) above the line. The first subgoal of `assert_PROP` requires us to prove the proposition, using facts from the current Hoare precondition (which would not be accessible to Coq's standard `assert`). The reason this one is so easily provable is that `entailer!` extracts the \( n = Z\text{length}(\sigma) \) fact from `data_at` and puts it above the line.

The second subgoal is just like the subgoal we had before doing `assert_PROP`, but with the new proposition above the line. Now that \( \text{H}_2: Z\text{length} \) contents \( = \) \( \text{size} \) is above the line, `forward` succeeds on the array subscript.

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

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

H : 0 ≤ size ≤ Int.max_signed
H0 : Forall (fun x : Z ⇒ 0 ≤ x ≤ Int.max_unsigned) contents
HRE : i < size
H1 : 0 ≤ i ≤ size

(other above-the-line hypotheses elided)

ENTAIL Delta,

PROP()
LOCAL(temp _i (Vint (Int.add (Int.repr i) (Int.repr 1)));
  temp _s
    (force_val
      (sem_add_default tint tint
        (Vint (Int.repr (sum_Z (sublist 0 i contents)))))
      (Znth i (map Vint (map Int.repr contents)) 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 tuint size) (map Vint (map Int.repr contents)) a)

⇓ EX a₀ : Z,

PROP(0 ≤ a₀ ≤ size)
LOCAL(temp _a a; temp _i (Vint (Int.repr a₀));
  temp _n (Vint (Int.repr size));
  temp _s (Vint (Int.repr (sum_Z (sublist 0 a₀ contents))))

SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a)

The right-hand side of this entailment is just the loop invariant. As usual at the end of a loop body, there is an existentially quantified variable that must be instantiated with an iteration-dependent value. In this case it’s obvious: the quantified variable represents the contents of C local variable _i, so we do, Exists (i+1).
The resulting entailment has many trivial parts and a nontrivial residue. The usual way to get to the hard part is to run entailer!, which we do now. After clearing away the irrelevant hypotheses, we have:

\[
\begin{align*}
H &: 0 \leq \text{Zlength contents} \leq \text{Int.max.signed} \\
\text{HRE} &: i < \text{Zlength contents} \\
\text{H1} &: 0 \leq i \leq \text{Zlength contents} \\
\end{align*}
\]

\[
\begin{align*}
\text{Vint} \left( \text{Int.repr} \left( \text{sum-Z} \left( \text{sublist} 0 (i + 1) \text{contents}) \right) \right) \right) &= \\
\text{Vint} \left( \text{Int.repr} \left( \text{sum-Z} \left( \text{sublist} 0 i \text{contents} \right) + \text{Znth} i \text{contents} 0 \right) \right) \\
\end{align*}
\]

Applying \text{f.equal} twice, leaves the goal,

\[
\begin{align*}
\text{sum.Z} \left( \text{sublist} 0 (i + 1) \text{contents}) &= \\
\text{sum.Z} \left( \text{sublist} 0 i \text{contents} \right) + \text{Znth} i \text{contents} 0 \\
\end{align*}
\]

Now the lemma \text{sublist.split} is helpful here:

\[
\begin{align*}
\text{sublist.split}: \forall l m h \text{al}, \ 0 \leq l \leq m \leq h \leq |\text{al}| \rightarrow \\
\text{sublist} l h \text{al} &= \text{sublist} l m \text{al} ++ \text{sublist} m h \text{al} \\
\end{align*}
\]

So we do, rewrite \text{(sublist.split 0 i (i+1))} by omega. A bit more rewriting with the theory of \text{sum.Z} and \text{sublist} finishes the proof.

See also: Chapter 56 (sublist).
17 Returning from a function

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

\[ H4 : \text{Forall} \ (\text{value\_fits} \ \text{tu}nt) \ (\text{map} \ \text{Vint} \ (\text{map} \ \text{Int}\text{.repr} \ \text{contents})) \]
\[ H2 : \text{field\_compatible} \ (\text{Tarray} \ \text{tu}nt \ (\text{Zlength} …) \ \text{noattr}) \ [\ a \ (\text{other above-the-line hypotheses elided}) \]

\[ \text{data\_at} \ \text{sh} \ (\text{tarray} \ \text{tu}nt \ (\text{Zlength} …)) \ (\text{map} \ \text{Vint} \ (\text{map} \ \text{Int}\text{.repr} \ \text{contents})) \ a \]
\[ \vdash \text{!!} (\text{Vint} \ (\text{Int}\text{.repr} \ (\text{sum\_Z} \ \text{contents}))) = \]
\[ \text{Vint} \ (\text{Int}\text{.repr} \ (\text{sum\_Z} \ (\text{sublist} 0 \ i \ \text{contents})))) \]

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 35).

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

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

We can proceed by using entailer! The remaining subgoal solves easily in the theory of sublists. The proof of the function sumarray is now complete.
18 Global variables and \texttt{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 \texttt{main} function is special: it has spatial assertions for \textit{all} the global variables. Then it may pass these on, piecemeal, to the functions it calls on an as-needed basis.

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

\begin{verbatim}
Definition \texttt{main\_spec} :=
  DECLARE \_main WITH u : unit
  PRE [ ] \texttt{main\_pre prog u}
  POST [ tint ]
      (* application-specific postcondition *)
  PROP()
  LOCAL(temp ret\_temp (Vint (Int.repr (1+2+3+4))))
  SEP(TT).
\end{verbatim}

The first four lines are always the same for any program. \texttt{main\_pre} calculates the precondition automatically from the list of extern global variables and initializers of the program.

Now, when we prove that \texttt{main} satisfies its funspec,

\begin{verbatim}
Lemma body\_main: \texttt{semax\_body Vprog Gprog f\_main main\_spec}.
Proof.
start\_function.
\end{verbatim}

the \texttt{start\_function} tactic “unpacks” \texttt{main\_pre} into an assertion:
18. **GLOBAL VARIABLES AND main()**

\[ v\_four : \text{val} \]
--------------------------------------(1/1)

\text{semax Delta}
\hspace{1em}(\text{PROP } (\text{LOCAL}(g\text{var }\_\text{four }v\_\text{four}))
\hspace{1em}(\text{SEP(data\_at }E\text{ws (tarray }t\text{uint }4)
\hspace{1em}(\text{map }V\text{int }[\text{Int\_repr }1; \text{Int\_repr }2; \text{Int\_repr }3; \text{Int\_repr }4]) \hspace{1em}v\_\text{four}))
\hspace{1em}(\ldots \text{function body} \ldots)
\hspace{1em}\text{POSTCONDITION}

The **LOCAL** clause means that the C global variable \_four is at memory address \_four. See Chapter 33.

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 43.

The \text{sumarray} program’s main\_spec \textit{postcondition} is specific to this program: we say that main returns the value \(1 + 2 + 3 + 4\).

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

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

Now it’s time to prove the function-call statement, \( s = \text{sumarray}(\text{four}, 4) \). When proving a function call, one must supply a \textit{witness} for the WITH clause of the function-spec. The \texttt{sumarray} function’s WITH clause (page 14) starts,

\textbf{Definition} \texttt{sumarray-spec} := \\
\text{DECLARE \_sumarray} \\
\text{WITH } a: \text{val}, \ sh: \text{share}, \ contents: \text{list } \text{Z}, \ \text{size: Z}

so the type of the witness will be \((\text{val}* (\text{share} * (\text{list } \text{Z} \ * \text{Z})))\). To choose the witness, examine your actual parameter values (along with the precondition of the funspec) to see what witness would be consistent; here, we use \((v_{\text{four}}, \text{Ews}, \text{four-contents}, 4)\) as follows:

\text{forward-call } (v_{\text{four}}, \text{Ews}, \text{four-contents}, 4).

The \texttt{forward-call} tactic (usually) leaves subgoals: you must prove that your current precondition implies the funspec’s precondition. Here, these solve easily, as shown in the proof script.

Finally, we are at the return statement. See Chapter 17. In this case, the forward tactic is able to prove (using a form of the \texttt{entailer} tactic) that the current assertion implies the postcondition of \_main.
20 Tying all the functions together

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

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

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

Lemma body_sumarray: semax_body Vprog Gprog f_sumarray sumarray_spec.

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

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

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

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

The calls to semax_func_cons must appear in the same order as the functions appear in prog.(prog_defs).
21 Separation logic: EX, *, emp, !!

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

\[ R ::= \begin{array}{ll}
\text{emp} & \text{empty} \\
\text{TT} & \text{True} \\
\text{FF} & \text{False} \\
R_1 * R_2 & \text{separating conjunction} \\
R_1 && R_2 & \text{ordinary conjunction} \\
\text{field\_at } \pi \tau \vec{f}ld v p & \text{“field maps-to”} \\
\text{data\_at } \pi \tau v p & \text{“maps-to”} \\
\text{array\_at } \tau \pi v lo hi & \text{array slice} \\
!! P & \text{pure proposition} \\
\text{EX } x : T, R & \text{existential quantification} \\
\text{ALL } x : T, R & \text{universal quantification} \\
R_1 \parallel R_2 & \text{disjunction} \\
\text{wand } R R' & \text{magic wand } R \rightarrow* R' \\
\ldots & \text{other operators, including user definitions}
\end{array} \]
In a canonical-form lifted assertion, existentials can occur at the outside, or in one of the base-level conjuncts within the SEP clause. The left-hand side of this assertion has both:

\[
\text{ENTAIL } \Delta, \quad (* \text{this example in progs/tutorial1.v } *)
\]
\[
\text{EX } x:Z, \\
\text{PROP}(0 \leq x) \text{ LOCAL(temp } \_i \text{ (Vint (Int.repr } x))
\]
\[
\text{SEP(EX } y:Z, \text{ !!}(x < y) \&\& \text{ data_at } \pi \text{ tint (Vint (Int.repr } y)) \text{ p})
\]
\[
\vdash \text{EX } u: Z, \\
\text{PROP}(0 < u) \text{ LOCAL()}
\]
\[
\text{SEP(data_at } \pi \text{ tint (Vint (Int.repr } u)) \text{ p})
\]

To prove this entailment, one can first move \( x \) and \( y \) “above the line” by the tactic **Intros** \( a \ b \):

\[
a: Z \\
b: Z \\
H: 0 \leq a \\
H0: a < b
\]

\[
\text{ENTAIL } \Delta, \\
\text{PROP()} \text{ LOCAL(temp } \_i \text{ (Vint (Int.repr } a))}
\]
\[
\text{SEP(data_at } \pi \text{ tint (Vint (Int.repr } b)) \text{ p})
\]
\[
\vdash \text{EX } u: Z, \\
\text{PROP}(0 < u) \text{ LOCAL()}
\]
\[
\text{SEP(data_at } \pi \text{ tint (Vint (Int.repr } u)) \text{ p})
\]

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 \texttt{Intros} with no arguments to do that.

If you want to Intro an existential \textit{without} PROP-introduction and *-flattening, you can just use \texttt{Intro} a, instead of \texttt{Intros} a.

Then, instantiate \(u\) by \texttt{Exists} \(b\).

\[
\begin{align*}
  a & : Z \\
  b & : Z \\
  H & : 0 \leq a \\
  H0 & : a < b
\end{align*}
\]

This entailment proves straightforwardly by entailer!. 

\texttt{ENTAIL \Delta,} 
\begin{align*}
  \text{PROP()} & \text{LOCAL(temp \_i (Vint (Int.repr a)))} \\
  \text{SEP(data\_at \_i tint (Vint (Int.repr b)) p)} \\
  \vdash \text{PROP(0 < b) LOCAL()} \\
  \text{SEP(data\_at \_i tint (Vint (Int.repr b)) p)} \\
\end{align*}
23 Integers: nat, Z, int

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

C-language integer values are represented by the type Int.int (or just int for short), which are 32-bit two’s complement signed or unsigned integers with mod-$2^{32}$ arithmetic. Chapter 52 describes the operations on the int type.

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

Conversions between Z and int are done as follows:

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

with the following lemmas:

- $\text{Int.repr}(\text{Int.unsigned } z) = z$
- $0 \leq z \leq \text{Int.max_unsinged}$
  $\text{Int.unsigned}(\text{Int.repr } z) = z$
- $\text{Int.repr}(\text{Int.signed } z) = z$
- $\text{Int.min_signed} \leq z \leq \text{Int.max_signed}$
  $\text{Int.signed}(\text{Int.repr } z) = z$

Int.repr truncates to a 32-bit twos-complement representation (losing information if the input is out of range). Int.signed and Int.unsigned are different injections back to Z that never lose information.
When doing proofs about signed integers, you must prove that your integers never overflow; when doing proofs about unsigned integers, it’s still a good idea to prove that you avoid overflow. That is, if the C variable \_x contains the value Vint (Int.repr x), then make sure x is in the appropriate range. Let’s assume that \_x is a signed integer, i.e. declared in C as int x; then the hypothesis is,

\( H: \text{Int.min\_signed} \leq x \leq \text{Int.max\_signed} \quad (\ast \text{this example in progs/tutorial1.v} \ast) \)

If you maintain this hypothesis “above the line”, then Floyd’s tactical proof automation can solve goals such as \( \text{Int.signed} (\text{Int.repr} x) = x \). Also, to solve goals such as,

\[
\ldots \\
H2 : 0 \leq n \leq \text{Int.max\_signed} \quad (\ast \text{this example in progs/tutorial1.v} \ast) \\
\ldots
\]

\[
\text{------------------------} \\
\text{Int.min\_signed} \leq 0 \leq n
\]

you can use the rep\_omega tactic, which is basically just omega with knowledge of the values of \( \text{Int.min\_signed} \), \( \text{Int.max\_signed} \), and \( \text{Int.max\_unsigned} \).

To take advantage of this, put conjuncts into the PROP part of your function precondition such as \( 0 \leq i < n; \ n \leq \text{Int.max\_signed} \). Then the start\_function tactic will move them above the line, and the other tactics mentioned above will make use of them.

To see an example in action, look at progs/verif\_sumarray.v. The funspec’s precondition contains,

\[
\text{PROP}(\ldots \quad 0 \leq \text{size} \leq \text{Int.max\_signed}; \\
\quad \text{Forall} \ (\text{fun} \ x \Rightarrow 0 \leq x \leq \text{Int.max\_unsigned}) \ \text{contents})
\]

to ensure that size is representable as a nonnegative signed integer, and each element of contents is representable as an unsigned.
24 \textbf{Int, Int8, Int16, Int64, Ptrofs}

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

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

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

\begin{itemize}
  \item \textbf{Int8} for char (signed or unsigned)
  \item \textbf{Int16} for short (signed or unsigned)
  \item \textbf{Int} for int (signed or unsigned)
  \item \textbf{Int64} for long (signed or unsigned)
  \item \textbf{Ptrofs} for size\_t
\end{itemize}

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

The following definitions are used for shorthand:

\begin{itemize}
  \item \textbf{Definition} int = Int.int.
  \item \textbf{Definition} int64 = Int64.int.
  \item \textbf{Definition} ptrofs = Ptrofs.int.
\end{itemize}
25 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 → ptofs → val.

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

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

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

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

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

Vundef is not always treated as distinct from a defined value. For example, \(p \mapsto\text{Vint5} \vdash p \mapsto\text{Vundef}\), where \(\mapsto\) is the data_at operator (Chapter 30). That is, \(p \mapsto\text{Vundef}\) really means \(\exists v, p \mapsto v\). Vundef could mean “truly uninitialized” or it could mean “initialized but arbitrary.”
26 C types

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

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

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

Inductive type : Type :=
  | Tvoid: type
  | TInt: intsize → signedness → attr → type
  | TLong: signedness → attr → type
  | TFloat: floatsize → attr → type
  | TPointer: type → attr → type
  | TArray: type → Z → attr → type
  | TFunction: typelist → type → calling_convention → type
  | TStruct: ident → attr → type
  | TUnion: ident → attr → type

with typelist : Type :=
  | TNil: typelist
  | TCons: type → typelist → typelist.

We have abbreviations for commonly used types:

Definition tint = TInt I32 Signed noattr.
Definition tuint = TInt I32 Unsigned noattr.
Definition tschar = TInt I8 Signed noattr.
Definition tuchar = TInt I8 Unsigned noattr.
Definition tarray (t: type) (n: Z) = TArray t n noattr.
Definition tptr (t: type) := TPointer t noattr.
The C language has a namespace for struct- and union-identifiers, that is, *composite types*. In this example, struct foo {int value; struct foo *tail} a,b; the “global variables” namespace contains a,b, and the “struct and union” namespace contains foo.

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

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

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

```
Require Import VST.floyd.proofauto. (* Import Verifiable C library *)
Require Import myprogram. (* AST of my program *)
```

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 66.
28 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: \( \forall (t: \text{type}), \)

\[
\text{reptype } t = \\
\text{match } t \text{ with} \\
| \text{Tvoid } \Rightarrow \text{unit} \\
| \text{Tint } \_ \_ \Rightarrow \text{val} \\
| \text{Tlong } \_ \_ \Rightarrow \text{val} \\
| \text{Tfloat } \_ \_ \Rightarrow \text{val} \\
| \text{Tpointer } \_ \_ \Rightarrow \text{val} \\
| \text{Tarray } t0 \_ \_ \Rightarrow \text{list } (\text{reptype } t0) \\
| \text{Tfunction } \_ \_ \Rightarrow \text{unit} \\
| \text{Tstruct } id \_ \Rightarrow \text{reptype-structlist } (\text{co-members } (\text{get-co id})) \\
| \text{Tunion } id \_ \Rightarrow \text{reptype-unionlist } (\text{co-members } (\text{get-co id})) \\
\text{end}
\]

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

\[
\text{struct list} \{ \text{int hd; struct list } *\text{tl}; \}; \\
\text{struct one} \{ \text{struct list } *\text{p}; \}; \\
\text{struct three} \{ \text{int a; struct list } *\text{p; double } x; \};
\]

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.
29 Uninitialized data, default_val

CompCert represents uninitialized atomic (integer, pointer, float) values as $\text{Vundef : val}$.

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

\[
\text{default_val} : \forall \{\text{cs : compspecs}\} \ (t : \text{type}), \ \text{reptype} \ t.
\]

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;};
```

\[
\begin{align*}
\text{default_val tint} &= \text{Vundef} \\
\text{default_val (tptr tint)} &= \text{Vundef} \\
\text{default_val (tarray tint 4)} &= [\text{Vundef}; \text{Vundef}; \text{Vundef}; \text{V undef}] \\
\text{default_val (tarray t n)} &= \text{list_repeat (Z.to_nat } n \text{) (default_val } t) \\
\text{default_val (Tstruct .list noattr)} &= (\text{Vundef, Vundef})
\end{align*}
\]
Consider a C program with these declarations:

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

Assume these definitions in Coq:

```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 \_`:

```coq
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
Consider the example in progs/nest2.c

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

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

\[
\text{LOCAL}(gvar \_pb \ pb) \quad \text{SEP}(\text{data\_at } sh \ t\_struct\_b (u,(v,w)) \ pb)
\]

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

The tactic \( \text{unfold\_data\_at} \ 1\%\text{nat} \) changes the SEP part of the assertion as follows:

\[
\text{SEP}(\text{field\_at } Ews \ t\_struct\_b (DOT \_y1) (Vint u) \ pb;)
\quad \text{field\_at } Ews \ t\_struct\_b (DOT \_y2) (Vfloat v, Vint w) \ pb)
\]

and then doing \( \text{unfold\_field\_at} \ 2\%\text{nat} \) unfolds the second field_at,

\[
\text{SEP}(\text{field\_at } Ews \ t\_struct\_b (DOT \_y1) (Vint u) \ pb;)
\quad \text{field\_at } Ews \ t\_struct\_b (DOT \_y2 DOT \_x1) (Vfloat v) \ pb;)
\quad \text{field\_at } Ews \ t\_struct\_b (DOT \_y2 DOT \_x2) (Vint w) \ pb)
\]

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 \( \pi \tau v p \) is just short for field_at \( \pi \tau \text{nil} v p \).

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

The `reptype` function maps C types to 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`: \(\forall (t: \text{type}),\) `reptype t = match t with | Tvoid \Rightarrow unit | Tint _ _ \Rightarrow \text{int} | Tlong _ _ \Rightarrow \text{Int64\text{.int}} | Tfloat _ _ \Rightarrow \text{float} | Tpointer _ _ \Rightarrow \text{pointer\_val} | Tarray t0 _ _ \Rightarrow \text{list (reptype t0)} | Tfunction _ _ \Rightarrow \text{unit} | Tstruct id _ \Rightarrow \text{reptype\'_structlist (co\_members (get\_co id))} | Tunion id _ \Rightarrow \text{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) : \text{reptype t} \rightarrow \text{reptype t} := ...
The program progs/nest2.c (verified in progs/verif_nest2.v) illustrates the use of reptype' and repinj.

```c
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 :=

```plaintext
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) p)
POST [ tint ]
PROP() LOCAL(temp ret_temp (Vint (snd (snd v))))
SEP(data_at Ews t_struct_b (repinj_v p) 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 :=

```plaintext
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).
```
33 LOCAL defs: temp, lvar, gvar

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

**Inductive** localdef : Type :=
  | temp: ident → val → localdef
  | lvar: ident → type → val → localdef
  | gvar: ident → val → localdef
  | sgvar: ident → val → localdef
  | localprop: Prop → localdef.

**temp** \(i\ v\) binds a nonaddressable local variable \(i\) to its value \(v\).

**lvar** \(i\ t\ v\) binds an *addressable* local variable \(i\) (of type \(t\)) to its *address* \(v\).

**gvar** \(i\ v\) binds a *visible global* variable \(i\) to its *address* \(v\).

**sgvar** \(i\ v\) binds a *possibly shadowed global* variable \(i\) to its *address* \(v\).

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

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

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

```
PROP() LOCAL(temp -p q; lvar -g tint q; sgvar -g p}
SEP(data_at Ews tint (Vint (Int.repr 2)) p;
    data_at Tsh tint (Vint (Int.repr 6)) q)
```

To describe a shadowed global variable \(-g\) that is still there in memory but (temporarily) cannot be referred to by its name in the C program.
Normally one does not use this tactic directly, it is invoked as the first step of entailer or entailer!

Given a lifted entailment $\text{ENTAIL } \Delta$, \text{PROP}(\vec{P}) \text{ LOCAL}(\vec{Q}) \text{ 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 $\vec{Q}$ are (computable to) ground terms, and the analogous (tree) property for $\Delta$. 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 $\Delta$ and $\vec{Q}$. When the entailment is not fully concrete, or when there existential quantifiers outside PROP, the tactic old_go_lower can still be useful.

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

For each localdef in $\vec{Q}$ (such as temp i v), go_lower looks up i in $\Delta$ 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 $\ast$. $S'$ is the residue of $S$ after lowering to the base separation logic and deleting its (provable) localdefs.
Normally one does not use this tactic directly, it is invoked by entailer or entailer!

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

An example local fact: `data_at Ews (tarray tint n) v p ⊢ !! (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:
  field_at π t path v p ⊢ !! (field_compatible t path p
  ∧ value_fits (nested_field_type t path) v)
  data_at π t v p ⊢ !! (field_compatible t nil p ∧ value_fits t v)
memory_block_local_facts:
  memory_block π n p ⊢ !! isptr p
```

The assertion $(Zlength v = n)$ is actually a consequence of value_fits when $t$ is an array type. See Chapter 37.

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 $R_1 \ast R_2 \ast \ldots \ast R_n \vdash S$ and adds saturate-local facts for each of the $R_i$, though it avoids adding duplicate hypotheses above the line.


36 field\_compatible, field\_address

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

```c
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 \texttt{pa} is \(p\). Then the value of \texttt{q} is \(p + \delta\); how can we reason about \(\delta\)?

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

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

\[
\begin{align*}
\text{data\_at } \pi \; \text{t\_struct\_b} \; (i, (f, j)) & \vdash \\
\text{data\_at } \pi \; \text{tint} \; i \; p \; \ast \; \text{data\_at } \pi \; \text{t\_struct\_a} \; (f, j) \; (\text{offset\_val} \; p \; 12)
\end{align*}
\]

**but the converse is not valid:**

\[
\begin{align*}
\text{data\_at } \pi \; \text{tint} \; i \; p \; \ast \; \text{data\_at } \pi \; \text{t\_struct\_a} \; (f, j) \; (\text{offset\_val} \; p \; 12) \\
\n\n\n\n\n\n\n\n
\]

\[
\forall \text{data\_at } \pi \; \text{t\_struct\_b} \; (i, (f, j)) \; p
\]

The reasons: we don’t know that \(p + 12\) satisfies the alignment requirements for \texttt{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: \texttt{field\_compatible t\_struct\_b nil p : Prop.}

which says that an entire \texttt{struct b} value can fit at address \(p\). Note that
this is a nonspatial assertion about addresses, independent of the contents of memory.

In order to assist with reasoning about reassembly of data structures, saturate_local (and therefore entailer) puts field_compatible assertions above the line; see Chapter 35.

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

\[
\text{Definition } \text{field_address} \ (t: \text{type}) \ (path: \text{list gfield}) \ (p: \text{val}) : \text{val} := \\
\text{if field_compatible_dec } t \ path \ p \\
\text{then offset_val (Int.repr (nested_field_offset } t \ path)) \ p \\
\text{else Vundef}
\]

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

\[
\text{data_at } \pi \ \text{tint } i \ p \\
\ast \ \text{data_at } \pi \ \text{t_struct_a } (f,j) \ (\text{field_address } t \text{-struct_b } (\text{DOT } y2 \ \text{DOT } x2) \ p) \\
\vdash \ \text{data_at } \pi \ \text{t_struct_b } (i,(f,j)) \ p
\]

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

Sometimes when dealing with arrays, you want a pointer that might possibly point just one past the end of the array; that is, points to at least zero things. In this case, use field_address0 t path p, which is built from field_compatible0. It has slightly looser requirements for how close p can be to the end of memory.
37 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:

\[
value_fits\ t\ v = \text{tc-val}'\ t\ v \quad (\text{when } t\ \text{is an integer, float, or pointer type})
\]

\[
value_fits\ (\text{tarray } t'\ n)\ v = (Z\text{length } v = Z.\text{max } 0\ n) \land \text{Forall} (value_fits\ t')\ v
\]

\[
value_fits\ (\text{Tstruct } i\ \text{noattr})\ (v_1,(v_2,(\ldots,v_n))) = \quad \text{when the fields of struct } i\ \text{are } f_1,\ldots,f_n
\]

\[\text{value_fits\ (field\_type } f_1\ v_1) \land \ldots \land \text{value_fits\ (field\_type } f_n\ v_n)\]

The predicate tc_val' says,

**Definition** tc_val' (t: type) (v: val) := v \neq \text{Vundef} \rightarrow \text{tc_val} t\ v.

**Definition** tc_val (t: type) : val \rightarrow \text{Prop} :=

\[
\text{match } t\ \text{with}
\]

\[
| \text{Tvoid} \Rightarrow \text{False}
| \text{Tint } sz\ sg \Rightarrow \text{is\_int } sz\ sg
| \text{Tlong} \Rightarrow \text{is\_long}
| \text{Tfloat F32} \Rightarrow \text{is\_single}
| \text{Tfloat F64} \Rightarrow \text{is\_float}
| \text{Tpointer} \Rightarrow \text{is\_pointer\_or\_null}
| \text{Tarray} \Rightarrow \text{is\_ptr}
| \text{Tstruct} \Rightarrow \text{is\_struct}
\]

end

So, an atomic value (int, float, pointer) fits either when it is Vundef or when it type-checks. We permit Vundef to “fit,” in order to accommodate partially initialized data structures in C.
Since $\tau$ is usually concrete, $\text{tc\_val} \ \tau \ v$ immediately unfolds to something like,

TC0: is\_int l32 Signed (Vint i)
TC1: is\_int l8 Unsigned (Vint c)
TC2: is\_int l8 Signed (Vint d)
TC3: is\_pointer\_or\_null p
TC4: isptr q

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

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

TC1: $0 \leq \text{Int\_unsigned} \ c \leq \text{Byte\_max\_unsigned}$
TC2: $\text{Byte\_min\_signed} \leq \text{Int\_signed} \ c \leq \text{Byte\_max\_signed}$
38 cancel

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

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

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

\[ F := \text{?224 : list(\text{environ} \rightarrow \text{mpred})} \]

\[ (A_1 \ast A_2) \ast ((A_3 \ast A_4) \ast A_5) \vdash A_4 \ast (\text{fold\_right sepcon emp } F) \ast A_1 \]

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

\textbf{WARNING}: cancel can turn a provable entailment into an unprovable entailment. Consider this:

\[ A \ast C \vdash B \ast C \]

\[ A \ast D \ast C \vdash C \ast B \ast D \]

This goal is provable by first rearranging to \((A \ast C) \ast D \vdash (B \ast C) \ast D\). 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 \(A \ast C \vdash B \ast C\); but indeed such “context-dependent” cancellations do occur in the theory of linked lists; see PLCC Chapter 19.

\textbf{CANCEL does not use} \(\beta\eta\) equality, as that could be slow in some cases. That means sometimes cancel leaves a residual subgoal \(A \vdash A'\) where \(A =_\beta A'\); sometimes the only differences are in (invisible) implicit arguments. You can apply \text{derives\_refl} to solve such residual goals.
39 entailer!

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

When go_lower is applicable, the entailers start by applying it (see Chapter 34).

Then: saturate_local (see Chapter 35).

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

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

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

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

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

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

Generic separation-logic simplifications.

\[
\begin{align*}
P \ast \text{emp} &= P \\
\text{emp} \ast P &= P \\
P \&\& \text{TT} &= P \\
\text{TT} \&\& P &= P \\
P \&\& \text{FF} &= \text{FF} \\
\text{FF} \&\& P &= \text{FF} \\
P \ast \text{FF} &= \text{FF} \\
\text{FF} \ast P &= \text{FF} \\
P \&\& P &= P \\
(EX \_ : A, P) &= P \\
\text{local ‘True = TT}
\end{align*}
\]

Pull EX and !! out of *-conjunctions.

\[
\begin{align*}
(EX x : A, P) \ast Q &= EX x : A, P \ast Q \\
(EX x : A, P) \&\& Q &= EX x : A, P \&\& Q \\
P \ast (EX x : A, Q) &= EX x : A, P \ast Q \\
P \&\& (EX x : A, Q) &= EX x : A, P \&\& Q \\
P \ast (!!Q \&\& R) &= !!Q \&\& (P \ast R) \\
(!!Q \&\& P) \ast R &= !!Q \&\& (P \ast R)
\end{align*}
\]

Delete auto-provable propositions.

\[
\begin{align*}
P \rightarrow (!!P \&\& Q = Q) \\
P \rightarrow (!!P = \text{TT})
\end{align*}
\]

Integer arithmetic.

\[
\begin{align*}
n + 0 &= n \\
0 + n &= n \\
n \ast 1 &= n \\
1 \ast n &= n \\
\text{sizeof tuchar} &= 1 \\
\text{align} n 1 &= n \\
(z > 0) \rightarrow (\text{align} 0 z = 0) \\
(z \geq 0) \rightarrow (Z.\text{max} 0 z = z)
\end{align*}
\]
**Int32 arithmetic.**

- \(\text{Int.sub } x \ x = \text{Int.zero}\)
- \(\text{Int.sub } x \ \text{Int.zero} = x\)
- \(\text{Int.add } x \ (\text{Int.neg } x) = \text{Int.zero}\)
- \(\text{Int.add } x \ \text{Int.zero} = x\)
- \(\text{Int.add } \text{Int.zero} \ x = x\)
- \(x \neq y \rightarrow \text{offset_val}((\text{offset_val} \ v \ i) \ j) = \text{offset_val} \ v \ (\text{Int.add} \ i \ j)\)
- \(\text{Int.add}(\text{Int.repr} \ i)(\text{Int.repr} \ j) = \text{Int.repr}(i + j)\)
- \(\text{Int.add}(\text{Int.add} \ z \ (\text{Int.repr} \ i))(\text{Int.repr} \ j) = \text{Int.add} \ z \ (\text{Int.repr}(i + j))\)
- \(z > 0 \rightarrow (\text{align } 0 \ z = 0)\)
- \(\text{force_int}(\text{Vint} \ i) = i\)
- \((\text{min_signed} \leq z \leq \text{max_signed}) \rightarrow \text{Int.signed}(\text{Int.repr} \ z) = z\)
- \((0 \leq z \leq \text{max_unsigned}) \rightarrow \text{Int.unsigned}(\text{Int.repr} \ z) = z\)
- \((\text{Int.unsigned} \ i < 2^n) \rightarrow \text{Int.zero_ext} \ n \ i = i\)
- \((-2^{n-1} \leq \text{Int.signed} \ i < 2^{n-1}) \rightarrow \text{Int.sign_ext} \ n \ i = i\)

**map, fst, snd, ...**

- \(\text{map } f \ (x :: y) = f \ x :: \text{map } f \ y\)
- \(\text{map nil} = \text{nil}\)
- \(\text{fst}(x, y) = x\)
- \(\text{snd}(x, y) = y\)
- \((\text{isptr } v) \rightarrow \text{force_ptr} \ v = v\)
- \(\text{isptr} (\text{force_ptr} \ v) = \text{isptr} \ v\)
- \((\text{is_pointer_or_null } v) \rightarrow \text{ptr_eq} \ v \ v = \text{True}\)

**Unlifting.**

- \(\text{‘f } \rho = f \) [when \(f\ has \text{arity 0}]\)
- \(\text{‘f } a_1 \rho = f \ (a_1 \rho) \) [when \(f\ has \text{arity 1}]\)
- \(\text{‘f } a_1 \ a_2 \rho = f \ (a_1 \rho) \ (a_2 \rho) \) [when \(f\ has \text{arity 2, etc.}]\)
- \((P \ast Q)\rho = P \rho \ast Q \rho\)
- \((P \&\& Q)\rho = P \rho \&\& Q \rho\)
- \((!!P)\rho = !!P\)
- \((!!(P \&\& Q)) = !!P \&\& !!Q\)
- \((\text{EX } x : A, \ P x)\rho = \text{EX } x : A, \ P x \rho\)
- \((\text{EX x : B, P x}) = \text{EX x : B, ‘}(P x))\)
- \(\text{‘}(P \ast Q) = ‘P \ast ‘Q\)
- \(\text{‘}(P \&\& Q) = ‘P \&\& ‘Q\)
Type checking and miscellaneous.

\[
\text{tc\_andp\ tc\_TT\ e = e \quad \text{tc\_andp\ e\ tc\_TT} = e}
\]

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

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

\[
is\text{ptr\ }v \rightarrow (\text{eval\_cast\_neutral\ }v = v)
\]

\[
(\exists t.\ \text{tc\_val\ }t\ v \land \text{is\_pointer\_type\ }t) \rightarrow (\text{eval\_cast\_neutral\ }v = v)
\]

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

\[
deref\_noload(\text{tarray\ }t\ n) = (\text{fun\ }v \Rightarrow v) \quad \text{eval\_expr(ETempvar}\ i\ t) = \text{eval\_id\ }i
\]

\[
\text{eval\_expr(Econst\_int}\ i\ t) = \text{'}(Vint\ i)
\]

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

\[
\text{'}(\text{eval\_binop\ }op\ (\text{typeof\ }a)\ (\text{typeof\ }b))\ (\text{eval\_expr\ }a)\ (\text{eval\_expr\ }b)
\]

\[
\text{eval\_expr(Eunop\ op\ a\ t) = '\text{eval\_unop\ }op\ (\text{typeof\ }a)\ (\text{eval\_expr\ }a)'}
\]

\[
\text{eval\_expr(Ecast\ e\ t) = 'eval\_cast\ (typeof\ } e)\ t\ (\text{eval\_expr\ e)'}
\]

\[
\text{eval\_lvalue(Ederef\ e\ t) = 'force\_ptr\ (eval\_expr\ e)'}
\]

Function return values.

\[
\text{get\_result(Some\ }x) = \text{get\_result1}(x) \quad \text{retval(get\_result1}\ i\ \rho) = \text{eval\_id\ }i\ \rho
\]

\[
\text{retval(env\_set\ }\rho\ \text{ret\_temp\ }v) = v
\]

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

\[
\text{ret\_type(initialized}\ i\ \Delta) = \text{ret\_type}(\Delta)
\]
**Postconditions.** (autorewrite with ret_assert.)

\[
\begin{align*}
\text{normal\_ret\_assert } FF \ ek \ vl &= FF \\
\text{frame\_ret\_assert}(\text{normal\_ret\_assert } P) \ Q &= \text{normal\_ret\_assert } (P \ast Q) \\
\text{frame\_ret\_assert } P \ \text{emp} &= P \\
\text{frame\_ret\_assert } P \ Q \ \text{EK\_return } vl &= P \ \text{EK\_return } vl \ast Q \\
\text{frame\_ret\_assert}(\text{loop1\_ret\_assert } P \ Q) \ R &= \\
\text{loop1\_ret\_assert } (P \ast R)(\text{frame\_ret\_assert } Q \ R) \\
\text{frame\_ret\_assert}(\text{loop2\_ret\_assert } P \ Q) \ R &= \\
\text{loop2\_ret\_assert } (P \ast R)(\text{frame\_ret\_assert } Q \ R) \\
\text{overridePost } P \ (\text{normal\_ret\_assert } Q) &= \text{normal\_ret\_assert } P \\
\text{normal\_ret\_assert } P \ ek \ vl &= (!!(ek = \text{EK\_normal}) \&\& (!!(vl = \text{None}) \&\& P)) \\
\text{loop1\_ret\_assert } P \ Q \ \text{EK\_normal None} &= P \\
\text{overridePost } P \ R \ \text{EK\_normal None} &= P \\
\text{overridePost } P \ R \ \text{EK\_return} &= R \ \text{EK\_return}
\end{align*}
\]

**In addition to rewriting**, normalize applies the following lemmas:

\[
\begin{align*}
P \vdash TT & \quad \text{FF} \vdash P & \quad P \vdash P \ast TT & \quad (\forall x. (P \vdash Q)) \rightarrow (EX x : A, P \vdash Q) \\
(P \rightarrow (TT \vdash Q)) & \rightarrow (!!(P \vdash Q)) & \quad (P \rightarrow (Q \vdash R)) & \rightarrow (!!(P \&\& Q \vdash R)
\end{align*}
\]

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

Given the goal \( x \rightarrow P \), where \( x \) is not a Prop, normalize avoids doing an intro. This allows the user to choose an appropriate name for \( x \).
Consider the proof state of verif_sumarray.v, just after (* Prove postcondition of loop body implies loop invariant. *). We have,

\[ H : 0 \leq i \leq \text{size} \]

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

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

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

For example, \( \text{assert}_\text{PROP}(\text{Zlength contents} = \text{size}) \) gives you an entailment proof goal:

\[ H : 0 \leq i \leq \text{size} \]

```
ENTAIL Delta,
  (PROP () LOCAL(...)
   SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a))
\vdash \text{!! (Zlength contents} = \text{size}).
```

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

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

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

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

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

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

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

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

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

Using go_lower or entailer on an ENTAIL goal causes a tc_val assertion to be placed above the line for each initialized tempvar. As explained at page 59, this tc_val may be simplified into an is_int hypothesis, or even removed if vacuous.
43 Shares

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

\[
\begin{array}{c}
Tsh = \text{Share.top} \\
\text{Lsh} \\
\text{Rsh} = \text{Ews} \\
\text{Share.bot}
\end{array}
\]

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.

\[
\begin{align*}
\text{Share.split } Tsh &= (\text{Lsh, Rsh}) \\
\text{Share.split } Lsh &= (a, a') \\
\text{Share.split } Rsh &= (b, b') \\
a' \oplus b &= c \\
\text{lub}(c, Rsh) &= a' \oplus Rsh = d \\
\forall sh. \text{ writable_share } sh &\rightarrow \text{ readable_share } sh \\
\text{writable_share } Ews &\rightarrow \text{ readable_share } b \\
\text{writable_share } d &\rightarrow \text{ readable_share } c \\
\text{writable_share } Tsh &\rightarrow \neg \text{readable_share } Lsh
\end{align*}
\]

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 \((\text{data\_at } s h\ t\ w\ v)\) holds, then the share \(s h\) must include at least a read share, thus this gives permission to load memory at address \(v\) to get a value \(w\) of type \(t\).

To make sure \(s h\) has enough permission to write (i.e., \(Rsh \subset s h\), we can say \(\text{writable\_share } s h : \text{Prop}\).

To test whether a share \(s h\) is empty or nonempty, use \(\text{sepalg\_identity } s h\) or \(\text{sepalg\_nonidentity } s h\).

Memory obtained from \(\text{malloc}\) 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” \(E ws\) 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 \(E ws\). However, many function specifications can be parameterized over any share (example: \(\text{sumarray\_spec}\) on page 14); that kind of generalized specification makes the functions usable in more contexts.

In C it is undefined to test deallocated pointers for equality or inequalities, so the Hoare-logic rule for pointer comparison also requires some permission-share; see page 71.
44 Pointer comparisons

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

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

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

The file progs/verif_ptr_compare.v illustrates pointer comparisons.
45 Proof of the reverse program

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

That proof is based on a general notion of list segments. Here we show a simpler proof that does not use segments, but see Chapter 46 for proof that corresponds to Chapters 3 and 27 of PLCC.

The C program is in progs/reverse.c:

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

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

Please open your CoqIDE or Proof General to progs/verif_reverse2.v. As usual, in progs/verif_reverse2.v we import the clightgen-produced file reverse.v and then build CompSpecs and Vprog (see page 13, Chapter 27, Chapter 47).

For the struct list used in this program, we can define the notion of linked list \( x \stackrel{\sigma}{\Rightarrow} \text{nil} \) with a recursive definition:

```coq
Fixpoint listrep (sigma: list val) (x: val) : mpred :=
    match sigma with
    | h::hs => EX y:val, data_at Tsh t struct.list (h,y) x * listrep hs y
    | nil => !! (x = nullval) && emp
    end.
```
That is, listrep $\sigma$ $x$ describes a null-terminated linked list starting at pointer $p$, with permission-share $Tsh$, representing the sequence $\sigma$.

The API spec (see also Chapter 7) for reverse is,

**Definition** $\text{reverse}$.spec :=
DECLARE $_\text{reverse}$
WITH $\sigma$: list val, $p$: val
PRE [ $p$ OF (tptr t.struct.list) ]
PROP() LOCAL(temp $p$) SEP (listrep $\sigma$ $p$)
POST [ (tptr t.struct.list) ]
EX $q$: val, PROP() LOCAL(temp $p$ $q$) SEP (listrep (rev $\sigma$) $q$).

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

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

PROP() LOCAL(temp $p$) SEP(listrep $\sigma$ $p$).

After forward through two assignment statements ($w=\text{NULL}; v=p;)$ the LOCAL part also contains temp $v$ $p$; temp $w$ (Vint (Int.repr 0)).

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

$$\exists \sigma_1, \sigma_2. \ \sigma = \text{rev}(\sigma_1) \cdot \sigma_2 \land v \stackrel{\sigma_2}{\Rightarrow} 0 \ast w \stackrel{\sigma_1}{\Rightarrow} 0$$

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

EX $\sigma_1$: list val, EX $\sigma_2$: list val, EX $w$: val, EX $v$: val,
PROP($\sigma = \text{rev} \sigma_1 ++ \sigma_2$)
LOCAL(temp $w$ $w$; temp $v$ $v$)
SEP(listrep $\sigma_1$ $w$; listrep $\sigma_2$ $v$).
We apply forward_while with this invariant, and (as usual) we have four subgoals: (1) precondition implies loop invariant, (2) loop invariant implies typechecking of loop-termination test, (3) loop body preserves invariant, and (4) after the loop.

(1) To prove the precondition implies the loop invariant, we instantiate $\sigma_1$ with nil and $\sigma_2$ with $\sigma$; we instantiate $w$ with NULL and $v$ with $p$. But this leaves the goal,

\[
\text{ENTAIL } \Delta, \text{ PROP(} ) \text{ LOCAL(temp }_v p; \text{ temp }_w \text{ nullval; temp }_p p) \\
\text{ SEP(listrep } \sigma \text{ } p) \\
\vdash \text{ PROP(} \sigma = \text{ rev }[] + + \sigma \text{ ) LOCAL(temp }_w \text{ nullval; temp }_v p) \\
\text{ SEP(listrep }[] \text{ nullval; listrep } \sigma \text{ } p)
\]

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

(2) The type-checking condition is not trivial, as it is a pointer comparison (see Chapter 44), but the entailer! solves it anyway.

(3) The loop body starts by assuming the *loop invariant* and the truth of the *loop test*. Their propositional parts have already been moved above the line at the comment (* loop body preserves invariant *). That is, HRE: isptr $v$ says that the loop test is true, and H: $\sigma = \text{ rev } \sigma_1 + + \sigma_2$ is from the invariant.

The first statement in the loop body, $t=v\rightarrow \text{ tail}$; loads from the list cell at $v$. But our SEP assertion for $v$ is, listrep $\sigma_2 \ v$. The assertion listrep $\sigma_2 \ v$ is not a data_at that we can load from. So we can unfold this occurrence of listrep, but *still* there is no data_at unless we know that $\sigma_2$ is $h :: r$ for some $h, r$.

We destruct $\sigma_2$ leaving two cases: $\sigma_2 = \text{ nil}$ and $\sigma_2 = h :: r$. The first case is a contradiction—by the definition of listrep, we must have $v == \text{ nullptr}$, but that’s incompatible with isptr($v$) above the line.
In the second case, we have (below the line) \( \exists y, \ldots \) that binds the value of the tail-pointer of the first cons cell. We move that above the line by \textbf{Intros} \( y \).

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

We choose values to instantiate the existentials: \textbf{Exists} \((h::\sigma_1, r, v, y)\). (Note that forward\_while has uncurried the four separate EX quantifiers into a single 4-tuple EX.) Then entailer! leaves two subgoals:

\[
\begin{align*}
\text{rev } \sigma_1 & \text{ ++ } h :: r = (\text{rev } \sigma_1 \text{ ++ } [h]) \text{ ++ } r \\
\text{listrep } \sigma_1 & \text{ w * field\_at Tsh t\_struct\_list } [] (h, w) \text{ v * listrep } r \ y \\
\vdash \text{listrep } (h :: \sigma_1) \text{ v * listrep } r \ y
\end{align*}
\]

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

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

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

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

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

Require Import VST.progs.list_dt. Import LsegSpecial.

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

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

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

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

In general, with multiple data fields, the type of σ is constructed via reptype (see Chapter 28). In this example, with one data field, the type of σ computes to list val.
47 Global variables

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

In the program-AST produced by clightgen, globals (and their initializers) are listed as `Gvars` in the `prog_defs`. These are accessed (automatically) in two ways by the Verifiable C program logic. First, their names and types are gathered into `Vprog` as shown on page 13 (try the Coq command `Print Vprog` to see this list). Second, their initializers are translated into `data_at` conjuncts of separation logic as part of the `main_pre` definition (see page 35).

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

---

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

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

3For example, examine the proof state in `progs/verif_sumarray.v` immediately after `start_function` in Lemma `body_main`. 
48 For loops (special case)

Many for-loops have the form, \( \text{for} (\text{init}; i < hi; i++) \text{ body} \) such that the expression \( hi \) will evaluate to the same value every time around the loop. This upper-bound expression need not be a literal constant, it just needs to be invariant.

For these loops you can use the tactic,

\[
\text{forward\_for\_simple\_bound } n \quad (\text{EX } i:Z, \text{PROP}(\vec{P}) \text{ LOCAL}(\vec{Q}) \text{ SEP}(\vec{R})).
\]

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

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

You must omit from \( Q \) any mention of the loop iteration variable \(-i\). The tactic will insert the binding \( \text{temp } -i \). You need not write \( i \leq hi \) in \( P \), the tactic will insert it.

An example of a for-loop proof is in progs/verif_sumarray2.v. This is an alternate implementation of progs/sumarray.c (see Chapter 12) that uses a for loop instead of a while loop:

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

Also see progs/verif_min.v for several approaches to the specification/verification of another for-loop.
49 For loops (general case)

The C-language for loop has the general form, for \((\text{init}; \text{test}; \text{incr}) \text{ body}\). Let \(\text{Inv}\) be the loop invariant, established by the initializer and preserved by the body-plus-increment. Let \(\text{PreInc}\) be the assertion just before the increment. \(\text{Post}\) is the join-postcondition of the loop; you don’t need to provide it if either (1) there are no break statements in the loop, or (2) the postcondition is already provided in your proof context (typically because a close-brace follows the entire loop). Depending on whether you need \(\text{Post}\), verify the loop with,

\[
\text{forward}\_\text{for Inv PreInc.} \quad \text{or} \quad \text{forward}\_\text{for Inv PreInc Post.}
\]

This is demonstrated in the lemma \text{body}\_sumarray\_alt in the file \text{progs/verif}\_\text{sumarray2.v}.

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

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

In some cases you cannot go forward until the precondition has a certain form. For example, to go forward through \( t = v \rightarrow \text{tail} \); there must be a data\_at or field\_at in the SEP clause of the precondition that gives a value for _tail field of \( t \). As page 75 describes, a listrep can be unfolded to expose such a SEP conjunct.

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

- Use the rule of consequence explicitly:
  apply \( \text{semax\_pre} \) with \( \text{PROP}(\vec{P}')\text{LOCAL}(\vec{Q}')\text{SEP}(\vec{R}') \),
  then prove \( \text{ENTAIL} \; \Delta, \; \vec{P}';\vec{Q}';\vec{R} \; \vdash \; \vec{P}'';\vec{Q}'';\vec{R}'' \).
- Use the rule of consequence implicitly, by using tactics (page 81) that modify the precondition.
- Do rewriting in the precondition, either directly by the standard rewrite and change tactics, or by normalize (page 63).
- Extract propositions and existentials from the precondition, by using \textbf{Intros} (page 40) or normalize.
- Flatten stars into semicolons, in the SEP clause, by \textbf{Intros}.
- Use the freezer (page 102) to temporarily “frame away” spatial conjuncts.
TACTICS FOR MANIPULATING PRECONDITIONS. In many of these tactics we select specific conjuncts 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,\ldots.

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

- **focus\_SEP \( i \ j \ k \).** Bring items \#\( i,j,k \) to the front of the SEP list.
  - **focus\_SEP 5.** results in \( \text{PROP}(\vec{P})\text{LOCAL}(\vec{Q})\text{SEP}(f;a;b;c;d;e;g;h;i;j) \).
  - **focus\_SEP 0.** results in \( \text{PROP}(\vec{P})\text{LOCAL}(\vec{Q})\text{SEP}(a;b;c;d;e;f;g;h;i;j) \).
  - **focus\_SEP 1 3.** results in \( \text{PROP}(\vec{P})\text{LOCAL}(\vec{Q})\text{SEP}(b;d;a;c;e;f;g;h;i;j) \).
  - **focus\_SEP 3 1.** results in \( \text{PROP}(\vec{P})\text{LOCAL}(\vec{Q})\text{SEP}(d;b;a;c;e;f;g;h;i;j) \).

- **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 \( \text{PROP}(\vec{P})\text{LOCAL}(\vec{Q})\text{SEP}(f;a;b;c;d;e;g;h;i;j) \).
  - **gather\_SEP 1 3.** results in \( \text{PROP}(\vec{P})\text{LOCAL}(\vec{Q})\text{SEP}(b*d;a;c;e;f;g;h;i;j) \).
  - **gather\_SEP 3 1.** results in \( \text{PROP}(\vec{P})\text{LOCAL}(\vec{Q})\text{SEP}(d*b;a;c;e;f;g;h;i;j) \).

- **replace\_SEP \( i \ R \).** Replace the \( i \)th element the SEP list with the assertion \( R \), and leave a subgoal to prove.
  - **replace\_SEP 3 R.** results in \( \text{PROP}(\vec{P})\text{LOCAL}(\vec{Q})\text{SEP}(a;b;c;R;e;f;g;h;i;j) \).
    with subgoal \( \text{PROP}(\vec{P})\text{LOCAL}(\vec{Q})\text{SEP}(d) \vdash R \).

- **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 51.
51 The Frame rule

Separation Logic supports the Frame rule,

\[
\text{Frame} \quad \frac{\{P\} c \{Q\}}{\{P * F\} c \{Q * F\}}
\]

To use this in a forward proof, suppose you have the proof goal,

\[
\text{semax } \Delta \ \text{PROP}(\vec{P}) \ \text{LOCAL}(\vec{Q}) \ \text{SEP}(R_0; R_1; R_2) \ c_1; c_2; c_3 \ \text{Post}
\]

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

\[
\text{semax } \Delta \ \text{PROP}(\vec{P}) \ \text{LOCAL}(\vec{Q}) \ \text{SEP}(R_0; R_1; R_2) \ (c_1; c_2); c_3 \ \text{Post}
\]

Then eapply \text{semax-seq’} to peel off the first command \((c_1; c_2)\) in the new sequence:

\[
\text{semax } \Delta \ \text{PROP}(\vec{P}) \ \text{LOCAL}(\vec{Q}) \ \text{SEP}(R_0; R_1; R_2) \ c_1; c_2 \ ?88
\]

\[
\text{semax } \Delta' \ ?88 \ c_3 \ \text{Post}
\]

Then \text{frame-SEP 0 2} to retain only \(R_0; R_2\).

\[
\text{semax } \Delta \ \text{PROP}(\vec{P}) \ \text{LOCAL}(\vec{Q}) \ \text{SEP}(R_0; R_2) \ c_1; c_2 \ \ldots
\]

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.
52 32-bit Integers

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

**Inductive** comparison := Ceq | Cne | Clt | Cle | Cgt | Cge.

Int.wordsize: nat = 32.
Int.modulus : Z = \(2^{32}\).
Int.max_unsigned : Z = \(2^{32} - 1\).
Int.max_signed : Z = \(2^{31} - 1\).
Int.min_signed : Z = \(-2^{31}\).

Int.int : Type.
Int.unsigned : int \(\rightarrow\) Z.
Int.signed : int \(\rightarrow\) Z.
Int.repr : Z \(\rightarrow\) int.

Int.zero := Int.repr 0.

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

(* Operators of type int \(\rightarrow\) int *)
Int.neg  Int.not

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

**Lemma** eq_dec: \(\forall (x \ y: \text{int}), \{x = y\} + \{x <\> y\}\).

**Theorem** unsigned_range: \(\forall i, 0 \leq \text{unsigned} \ i < \text{modulus}\).

**Theorem** unsigned_range_2: \(\forall i, 0 \leq \text{unsigned} \ i \leq \text{max_unsigned}\).

**Theorem** signed_range: \(\forall i, \text{min_signed} \leq \text{signed} \ i \leq \text{max_signed}\).

**Theorem** repr_unsigned: \(\forall i, \text{repr} (\text{unsigned} \ i) = i\).

**Lemma** repr_signed: \(\forall i, \text{repr} (\text{signed} \ i) = i\).

**Theorem** unsigned_repr:
\[
\forall z, 0 \leq z \leq \text{max_unsigned} \rightarrow \text{unsigned} \ (\text{repr} \ z) = z.
\]
Theorem signed_repr:
\[ \forall z, \text{min_signed} \leq z \leq \text{max_signed} \rightarrow \text{signed} (\text{repr} z) = z. \]

Theorem signed_eq_unsigned:
\[ \forall x, \text{unsigned} x \leq \text{max_signed} \rightarrow \text{signed} x = \text{unsigned} x. \]

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

Theorem eq_sym: \[ \forall x y, \text{eq} x y = \text{eq} y x. \]

Theorem eq_spec: \[ \forall (x y: \text{int}), \text{if eq} x y \text{ then} x = y \text{ else} x <> y. \]

Theorem eq_true: \[ \forall x, \text{eq} x x = \text{true}. \]

Theorem eq_false: \[ \forall x y, x <> y \rightarrow \text{eq} x y = \text{false}. \]

Theorem add_unsigned: \[ \forall x y, \text{add} x y = \text{repr} (\text{unsigned} x + \text{unsigned} y). \]

Theorem add_signed: \[ \forall x y, \text{add} x y = \text{repr} (\text{signed} x + \text{signed} y). \]

Theorem add_commut: \[ \forall x y, \text{add} x y = \text{add} y x. \]

Theorem add_zero: \[ \forall x, \text{add} x \text{ zero} = x. \]

Theorem add_zero_l: \[ \forall x, \text{add} \text{ zero} x = x. \]

Theorem add_assoc: \[ \forall x y z, \text{add} (\text{add} x y) z = \text{add} x (\text{add} y z). \]

Theorem neg_repr: \[ \forall z, \text{neg} (\text{repr} z) = \text{repr} (-z). \]

Theorem neg_zero: neg zero = zero.

Theorem neg_involutive: \[ \forall x, \text{neg} (\text{neg} x) = x. \]

Theorem neg_add_distr: \[ \forall x y, \text{neg} (\text{add} x y) = \text{add} (\text{neg} x) (\text{neg} y). \]

Theorem sub_zero_l: \[ \forall x, \text{sub} x \text{ zero} = x. \]

Theorem sub_zero_r: \[ \forall x, \text{sub} \text{ zero} x = \text{neg} x. \]

Theorem sub_add_oppos: \[ \forall x y, \text{sub} x y = \text{add} x (\text{neg} y). \]

Theorem sub_idem: \[ \forall x, \text{sub} x x = \text{zero}. \]

Theorem sub_add_l: \[ \forall x y z, \text{sub} (\text{add} x y) z = \text{add} (\text{sub} x z) y. \]

Theorem sub_add_r: \[ \forall x y z, \text{sub} x (\text{add} y z) = \text{add} (\text{sub} x z) (\text{neg} y). \]

Theorem sub_shifted: \[ \forall x y z, \text{sub} (\text{add} x z) (\text{add} y z) = \text{sub} x y. \]

Theorem sub_signed: \[ \forall x y, \text{sub} x y = \text{repr} (\text{signed} x -\text{signed} y). \]
Theorem mul_commut: \( \forall x \, y, \, \text{mul} \, x \, y = \text{mul} \, y \, x. \)

Theorem mul_zero: \( \forall x, \, \text{mul} \, x \, \text{zero} = \text{zero}. \)

Theorem mul_one: \( \forall x, \, \text{mul} \, x \, \text{one} = x. \)

Theorem mul_assoc: \( \forall x \, y \, z, \, \text{mul} \, (\text{mul} \, x \, y) \, z = \text{mul} \, x \, (\text{mul} \, y \, z). \)

Theorem mul_add_distr_l: \( \forall x \, y \, z, \, \text{mul} \, (\text{add} \, x \, y) \, z = \text{add} \, (\text{mul} \, x \, z) \, (\text{mul} \, y \, z). \)

Theorem mul_signed: \( \forall x \, y, \, \text{mul} \, x \, y = \text{repr} \, (\text{signed} \, x \, \ast \, \text{signed} \, y). \)

and many more axioms for the bitwise operators, shift operators, signed/unsigned division and mod operators.
**53 CompCert C abstract syntax**

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

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

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

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

**Inductive** `expr : Type :=

\[
\begin{align*}
(* \ 1 \ *) \quad & \text{Econst\_int: int \rightarrow type \rightarrow expr} \\
(* \ 1.0 \ *) \quad & \text{Econst\_float: float \rightarrow type \rightarrow expr (\ast double\ precision \ast)} \\
(* \ 1.0f0 \ *) \quad & \text{Econst\_single: float \rightarrow type \rightarrow expr (\ast single\ precision \ast)} \\
(* \ 1L \ *) \quad & \text{Econst\_long: int64 \rightarrow type \rightarrow expr} \\
(* \ x \ *) \quad & \text{Evar: ident \rightarrow type \rightarrow expr} \\
(* \ x \ *) \quad & \text{Etempvar: ident \rightarrow type \rightarrow expr} \\
(* \ *e \ *) \quad & \text{Ederef: expr \rightarrow type \rightarrow expr} \\
(* \ &e \ *) \quad & \text{Eaddrof: expr \rightarrow type \rightarrow expr} \\
(* \ ~e \ *) \quad & \text{Eunop: unary\_operation \rightarrow expr \rightarrow type \rightarrow expr} \\
(* \ e+e \ *) \quad & \text{Ebinop: binary\_operation \rightarrow expr \rightarrow expr \rightarrow type \rightarrow expr} \\
(* \ (int)e \ *) \quad & \text{Ecast: expr \rightarrow type \rightarrow expr} \\
(* \ e.f \ *) \quad & \text{Efield: expr \rightarrow ident \rightarrow type \rightarrow expr}.
\]
\textbf{Inductive} unary-operation := Onotbool | Onotint | Oneg | Oabsfloat.
\textbf{Inductive} binary-operation := Oadd | Osub | Omul | Odiv | Omod \\
| Oand | Oor | Oxor | Oshl | Oeq | One | Olt | Ogt | Ole | Oge.

\textbf{Inductive} statement : Type := \\
(* /**;/*) | Sskip : statement \\
(* E_1=E_2; *) | Sassign : expr \rightarrow expr \rightarrow statement (* memory store *) \\
(* x=E; *) | Sset : ident \rightarrow expr \rightarrow statement (* tempvar assign *) \\
(* x=f(...); *) | Scall : option ident \rightarrow expr \rightarrow list expr \rightarrow statement \\
(* x=b(...); *) | Sbuiltin : option ident \rightarrow external_function \rightarrow typelist \rightarrow list expr \rightarrow statement \\
(* s_1; s_2 *) | Ssequence : statement \rightarrow statement \rightarrow statement \\
(* if() else {} *) | Sifthenelse : expr \rightarrow statement \rightarrow statement \rightarrow statement \\
(* for (;;s_2) s_1 *) | Sloop : statement \rightarrow statement \rightarrow statement \\
(* break; *) | Sbreak : statement \\
(* continue; *) | Scontinue : statement \\
(* return E; *) | Sreturn : option expr \rightarrow statement \\
| Sswitch : expr \rightarrow labeled_statements \rightarrow statement \\
| Slabeled : label \rightarrow statement \rightarrow statement \\
| Sgoto : label \rightarrow statement.
54 C light semantics

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

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

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

- `eval_id` \{CS: compspecs\} (id: ident) : environ → val.
  (* evaluate a tempvar *)

- `eval_var` \{CS: compspecs\} (id: ident) (ty: type) : environ → val.
  (* evaluate an lvar or gvar, addressable local or global variable *)

- `eval_cast` (t t’: type) (v: val) : val.
  (* cast value v from type t to type t’, but beware! There are three types involved, including native type of v. *)

- `eval_unop` (op: unary-operation) (t1 : type) (v1 : val) : val.

- `eval_binop` \{CS: compspecs\} (op: binary-operation) (t1 t2: type) (v1 v2: val) : val.

- `eval_lvalue` \{CS: compspecs\} (e: expr) : environ → val.
  (* evaluate an l-expression, one that denotes a loadable/storable place*)

- `eval_expr` \{CS: compspecs\} (e: expr) : environ → val.
  (* evaluate an r-expression, one that is not storable *)

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

\(^1\)that is, defined by Fixpoint instead of by Inductive.
have already been substituted by values. Thus the environment is not needed.

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

\[
\begin{align*}
\text{sem\_cast: } & \text{type } \rightarrow \text{type } \rightarrow \text{val } \rightarrow \text{option val.} \\
\text{sem\_cast\_*: } & \text{(* several helper functions for sem\_cast *)} \\
\text{bool\_val: } & \text{type } \rightarrow \text{val } \rightarrow \text{option bool.} \\
\text{bool\_val\_*: } & \text{(* helper functions *)} \\
\text{sem\_notbool: } & \text{type } \rightarrow \text{val } \rightarrow \text{option val.} \\
\text{sem\_neg: } & \text{type } \rightarrow \text{val } \rightarrow \text{option val.} \\
\text{sem\_sub \{CS: compspecs\}: } & \text{type } \rightarrow \text{type } \rightarrow \text{val } \rightarrow \text{val } \rightarrow \text{option val.} \\
\text{sem\_sub\_*: } & \text{(* helper functions *)} \\
\text{sem\_add \{CS: compspecs\}: } & \text{type } \rightarrow \text{type } \rightarrow \text{val } \rightarrow \text{val } \rightarrow \text{option val.} \\
\text{sem\_add\_*: } & \text{(* helper functions *)} \\
\text{sem\_mul: } & \text{type } \rightarrow \text{type } \rightarrow \text{val } \rightarrow \text{val } \rightarrow \text{option val.} \\
\text{sem\_div: } & \text{type } \rightarrow \text{type } \rightarrow \text{val } \rightarrow \text{val } \rightarrow \text{option val.} \\
\text{sem\_mod: } & \text{type } \rightarrow \text{type } \rightarrow \text{val } \rightarrow \text{val } \rightarrow \text{option val.} \\
\text{sem\_and: } & \text{type } \rightarrow \text{type } \rightarrow \text{val } \rightarrow \text{val } \rightarrow \text{option val.} \\
\text{sem\_or: } & \text{type } \rightarrow \text{type } \rightarrow \text{val } \rightarrow \text{val } \rightarrow \text{option val.} \\
\text{sem\_xor: } & \text{type } \rightarrow \text{type } \rightarrow \text{val } \rightarrow \text{val } \rightarrow \text{option val.} \\
\text{sem\_shl: } & \text{type } \rightarrow \text{type } \rightarrow \text{val } \rightarrow \text{val } \rightarrow \text{option val.} \\
\text{sem\_shr: } & \text{type } \rightarrow \text{type } \rightarrow \text{val } \rightarrow \text{val } \rightarrow \text{option val.} \\
\text{sem\_cmp: } & \text{comparison } \rightarrow \text{type } \rightarrow \text{type } \rightarrow \text{(...)} \rightarrow \text{val } \rightarrow \text{val } \rightarrow \text{option val.} \\
\text{sem\_unary\_operation: } & \text{unary\_operation } \rightarrow \text{type } \rightarrow \text{val } \rightarrow \text{option val.} \\
\text{sem\_binary\_operation \{CS: compspecs\}:} \\
& \text{binary\_operation } \rightarrow \text{type } \rightarrow \text{type } \rightarrow \text{mem } \rightarrow \text{val } \rightarrow \text{val } \rightarrow \text{option val.}
\end{align*}
\]

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

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

```plaintext
data_at \textit{sh} (\text{tarray tuint } k) \textit{al} p : \text{mpred}
```

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

Suppose we have a function \textit{sumarray(unsigned a[], int n)} that takes an array of length \textit{n}, and we apply it to a “slice” of \textit{p}: \textit{sumarray(p+i,k-i)}; where \(0 \leq i \leq k\). The precondition of the \textit{sumarray} funspec has \textit{data_at sh (tarray tint n) al}.

In this case, we would like \(a = \& (p[i]), n = k - j, \text{ and } bl = \text{ the sublist of } al \text{ from } i \text{ to } k - 1\).

To prove this function-call by \textit{forward_call}, we must split up \(\text{(data}_\text{at } \text{sh (tarray tint } k) \text{ al } p)\) into two conjuncts:

\[
\begin{align*}
&\text{data}_\text{at } \text{sh (tarray tint i) (sublist 0 i al) p } \ast \\
&\quad \text{data}_\text{at } \text{sh (tarray tint (k - i)) (sublist i k al) q},
\end{align*}
\]

where \textit{q} is the pointer to the array slice beginning at address \textit{p + i}. We write this as, \(q = \text{field_address0 (tarray tint k) [ArraySubsc i] } p\). That is, given a pointer \textit{p} to a data structure described by \(\text{tarray tint k}\), calculate the \textit{address} for subscripting the \textit{i}th element. (See Chapter 36)

As shown in the body\_main proof in progs/verif_sumarray2.v, the lemma \texttt{split\_array} proves the equivalence of these two predicates, using the VST-Floyd lemma \texttt{split2\_data\_at\_Tarray}. Then the \textit{data_at ... q} predicate can satisfy the precondition of \textit{sumarray}, while the \textit{p} slice will be part of the “frame” for the function call.

See also: \texttt{split3\_data\_at\_Tarray}. 

---

**Note:**

- **data_at** predicate checks if there is an array of slots starting at a specific address.
- **sumarray** function applies to a slice of an array.
- **split2_data_at_Tarray** lemma is used to prove predicate equivalence.
- **field_address0** function calculates the address for subscripting.
- **body_main** proof demonstrates the correctness of the split process.
Chapter 55 explained that we often need to reason about slices of arrays whose contents are sublists of lists. For that we have a function sublist \( i \ j \ l \) which makes a new list out of the elements \( i \ldots j-1 \) of list \( l \).

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

Often, you find equations “above the line” of the form,

\[ H: n = \text{Zlength} \left( \text{map} \ V\text{int} \left( \text{map} \ \text{Int}\text{.repr} \ \text{contents} \right) \right) \]

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

These rules comprise the sublist rewrite database:

- sublist-nil'': \( i = j \rightarrow \text{sublist} \ i \ j \ l = [] \).
- app-nil.l: \( [] ++ l = l \).
- app-nil.r: \( l +++ [] = l \).
- Zlength_rev: \( \text{Zlength} \ (\text{rev} \ l) = \text{Zlength} \ l \).
- sublist_rejoin': \( 0 \leq i \leq j = j' \leq k \leq \text{Zlength} l \rightarrow \text{sublist} \ i \ j \ l +++ \text{sublist} \ j' \ k \ l = \text{sublist} \ i \ k \ l \).
- sublist_app1: \( 0 \leq i \leq j \leq \text{Zlength} l \rightarrow \text{sublist} \ i \ j \ (l +++ l') = \text{sublist} \ i \ j \ l \).
- zlength_cons: \( \text{Zlength} \ (a::l) = \text{Z.succ} \ (\text{Zlength} \ l) \).
- sublist_sublist: \( 0 \leq m \rightarrow 0 \leq k \leq i \leq j - m \rightarrow \text{sublist} \ k \ i \ (\text{sublist} \ m \ j \ l) = \text{sublist} \ (k + m) \ (i + m) \ l \).
- sublist_app1: \( 0 \leq i \leq j \leq \text{Zlength} l \rightarrow \text{sublist} \ i \ j \ (l +++ l') = \text{sublist} \ i \ j \ l \).
sublist_app2: $0 \leq \text{Zlength} l \leq i \rightarrow$

sublist $i\ j\ (l\ ++\ l') = \text{sublist}\ (i - \text{Zlength} l)\ (j - \text{Zlength} l)\ l'$.

sublist_list_repeat: $0 \leq i \leq j \leq k \rightarrow$

sublist $i\ j\ (\text{list_repeat}\ (\text{Z.to_nat}\ k)\ v) = \text{list_repeat}\ (\text{Z.to_nat}\ (j - i))\ v$.

sublist_same: $i = 0 \rightarrow j = \text{Zlength} l \rightarrow \text{sublist}\ i\ j\ l = l$.

app_Znth1: $i < \text{Zlength} l \rightarrow \text{Znth} i\ (l\ ++\ l')\ d = \text{Znth} i\ l\ d$.

app_Znth2: $i \geq \text{Zlength} l \rightarrow \text{Znth} i\ (l\ ++\ l')\ d = \text{Znth} i - \text{Zlength} l\ l'\ d$.

Znth_sublist: $0 \leq i \rightarrow 0 \leq j < k - i \rightarrow \text{Znth} j\ (\text{sublist}\ i\ k\ l)\ d = \text{Znth} (j + i)\ l\ d$.

along with miscellaneous Z arithmetic:

\[
\begin{align*}
n - 0 &= n & 0 + n &= n & n + 0 &= n & n \leq m \rightarrow \text{max}(n, m) &= m \\
n + m - n &= m & n + m - m &= n & m - n + n &= m & n - n &= 0 \\
n + m - (n + p) &= m - p & \text{etcetera.}
\end{align*}
\]
57 Later

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

$\begin{align*}
\text{semax\_set\_forward} & \quad \Delta \vdash \{\triangleright P\} \quad x := e \
& \quad \{\exists v. x = (e[v/x]) \land P[v/x]\}
\end{align*}$

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 $\ast, \&\&$, ALL (and so on) commute with later: $\triangleright (P \ast Q) = (\triangleright P) \ast (\triangleright Q)$.

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

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

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

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

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

**Parameter** address\_mapsto:

\[
\text{memory\_chunk} \rightarrow \text{val} \rightarrow \text{share} \rightarrow \text{share} \rightarrow \text{address} \rightarrow \text{mpred}.
\]

**Parameter** func\_ptr : funspec \rightarrow \text{val} \rightarrow \text{mpred}.

func\_ptr \(\phi v\) means that value \(v\) is a pointer to a function with specification \(\phi\); see Chapter 62.

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

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

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

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

From these primitives, field\_at and data\_at are constructed.
59 with\_library: Library functions

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

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

Definition Gprog := ltac:(with\_library prog [sumarray\_spec; main\_spec]).

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

The next chapters explain the specifications of certain standard-library functions.
The C library’s malloc and free functions have these specifications:

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

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

You must Import VST.floyd.library. Then these funspecs are made available in your Gprog by the use of the with_library tactic (Chapter 59).

The purpose of the malloc_token is to describe the special record-descriptor that tells free how big the allocated record was.

See progs/verif_queue.v for a demonstration of malloc/free.
61 exit

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

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

Parameter func_ptr : funspec \to \text{val} \to \text{mpred}.

Definition func_ptr' f v := func_ptr f v && emp.

\text{func_ptr} \phi v \quad \text{means that } v \text{ is a pointer to a function with funspec } \phi.
\text{func_ptr'} \phi v \quad \text{is a form more suitable to be a conjunct of a SEP clause.}

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

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

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

The verification, in progs/verif_funcptr.v, defines

Definition myfunc_spec := DECLARE _myfunc myspec.

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

Near the beginning of Lemma body_main, notice that we have LOCAL(gvar _myfunc p) in the precondition. That gvar is needed by the tactic make_func_ptr _myfunc, which adds func_ptr' myspec p to the
SEP clause. It “knows” to use myspec because it looks up `_myfunc` in Delta (which, in turn, is derived from Gprog).

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

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

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

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

pred_ext: \( P \vdash Q \rightarrow Q \vdash P \rightarrow P=Q \).
derives_refl: \( P \vdash P \).
derives_trans: \( P \vdash Q \rightarrow Q \vdash R \rightarrow P \vdash R \).
andp_right: \( X \vdash P \rightarrow X \vdash Q \rightarrow X \vdash (P \& \& Q) \).
andp_left1: \( P \vdash R \rightarrow P \& \& Q \vdash R \).
andp_left2: \( Q \vdash R \rightarrow P \& \& Q \vdash R \).
orp_left: \( P \vdash Q \rightarrow P \vdash Q \| R \).
orp_right1: \( P \vdash Q \rightarrow P \vdash Q \| R \).
orp_right2: \( P \vdash R \rightarrow P \vdash Q \| R \).
exp_right: \( \forall \{B: \text{Type}\}(x:B)(P: \text{mpred})(Q: B \rightarrow \text{mpred}),
P \vdash Q x \rightarrow P \vdash \text{EX} x:B, Q \).
exp_left: \( \forall \{B: \text{Type}\}(P:B \rightarrow \text{mpred})(Q: \text{mpred}),
(P \vdash X, P \times X \vdash Q) \rightarrow P \vdash \text{EX} x:B, P \vdash Q \).
allp_left: \( \forall \{B\}(P: B \rightarrow \text{mpred}) \times Q, P \times X \vdash Q \rightarrow P \vdash \text{ALL} x:B, P \vdash Q \).
allp_right: \( \forall \{B\}(P: \text{mpred})(Q: B \rightarrow \text{mpred}),
(P \vdash X, P \times X \vdash Q v) \rightarrow P \vdash \text{ALL} x:B, Q \).
prop_left: \( \forall (P: \text{Prop}) Q, (P \rightarrow (TT \vdash Q)) \rightarrow \text{!!} P \vdash Q \).
prop_right: \( \forall (P: \text{Prop}) Q, P \rightarrow (Q \vdash \text{!!} P) \).
not_prop_right: \( \forall (P: \text{mpred})(Q: \text{Prop}), (Q \rightarrow (P \vdash \text{FF})) \rightarrow P \vdash \text{!!}(\sim Q) \).
sepcon_assoc: \( (P \ast Q) \ast R = P \ast (Q \ast R) \).
sepcon_comm: \( P \vdash Q, P \ast Q = Q \ast P \).
sepcon_andp_prop: \( P \ast (\text{!!} Q \& \& R) = \text{!!} Q \& \& (P \ast R) \).
derives_extract_prop: \( (P \rightarrow Q \vdash R) \rightarrow \text{!!} P \& \& Q \vdash R \).
sepcon_derivs: \( P \vdash P' \rightarrow Q \vdash Q' \rightarrow P \ast Q \vdash P' \ast Q' \).
64 Obscure higher-order axioms

The wand $\ast$ operator is “magic wand,” ewand $\circ$ is “existential magic wand,” and $\triangleright$ is pronounced “later” and written $\triangleright$ in Coq.

see PLCC, Chapter 19.

imp_andp_adjoint: $P && Q \vdash R \iff P \vdash (Q \rightarrow R)$.

wand_sepcon_adjoint: $P \ast Q \vdash R \iff P \vdash Q \ast R$.

ewand_sepcon: $(P \ast Q) \circ R = P \circ (Q \circ R)$.

ewand_TT_sepcon: $\forall (P \ Q \ R: A),
(P\ast Q) && (R\leftarrow TT) \vdash (P \&\&(R\rightarrow TT)) \ast (Q \&\& (R\rightarrow TT))$.

exclude_elsewhere: $P\ast Q \vdash (P \&\&(Q\rightarrow TT)) \ast Q$.

ewand_conflict: $P\ast Q \vdash FF \rightarrow P \&\&(Q\rightarrow R) \vdash FF$

now_later: $P \vdash \triangleright P$.

later_K: $\triangleright (P \rightarrow Q) \vdash (\triangleright P \rightarrow \triangleright Q)$.

later_allp: $\forall T (F: T \rightarrow \text{mpred}), \triangleright (\text{ALL} x:T, F x) = \text{ALL} x:T, \triangleright (F x)$.

later_exp: $\forall T (F: T \rightarrow \text{mpred}), \text{EX} x:T, \triangleright (F x) \vdash \triangleright (\text{EX} x:T, F x)$.

later_exp': $\forall T (\text{any}:T) F, \triangleright (\text{EX} x:T, F x) = \text{EX} x:T, \triangleright (F x)$.

later_imp: $\triangleright (P \rightarrow Q) = (\triangleright P \rightarrow \triangleright Q)$.

loeb: $\triangleright P \vdash P \rightarrow TT \vdash P$.

later_sepcon: $\triangleright (P \ast Q) = \triangleright P \ast \triangleright Q$.

later_wand: $\triangleright (P \ast Q) = \triangleright P \ast \triangleright Q$.

later_ewand: $\triangleright (P \circ Q) = (\triangleright P) \circ (\triangleright Q)$. 
65 Proving larg(ish) programs

When your program is not all in one .c file, see also Chapter 66. 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 ∆` can also be invoked individually, to concisely display the type context ∆ 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 sequencing 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 thatopaquely 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 $N : \text{list nat}$ 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 $\text{FRZL } F$ at the head of the SEP list, and adding a hypothesis $F ::= \text{abbreviate}$ to Coq’s proof context.

The term $\text{FRZL } F$ 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 $\text{FRZL } F$ by the the list of $F$’s constituents. 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 $\text{PROP}(P) \text{ LOCAL}(Q) \text{ SEP}(R)$ structure to terms of the form $\text{PROP}(P) \text{ LOCAL}(Q) \text{ SEP}(R) \text{ FR}(H)$ where $H : \text{list mpred}$. 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.
What to do when your program is spread over multiple .c files. See progs/even.c and progs/odd.c for an example.

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

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

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

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

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

Linking of the final proofs is described by Stewart.¹

---

67 Catalog of tactics / lemmas

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

assert_PROP \( P \) (tactic; Chapter 41) Put the proposition \( P \) above the line, if it is provable from the current precondition.
cancel (tactic; page 61) Deletes identical spatial conjuncts from both sides of a base-level entailment.
data_at_conflict \( p \) (tactic) equivalent to field_at_conflict \( p \) nil.
deadvars! (tactic) Removes from the LOCAL block of the current pre-condition, any variables that are irrelevant to the rest of program execution. Fails if there is no such variable.
derives_refl (lemma) \( A \vdash A \). Useful after cancel to handle \( \beta\eta \)-equality; see page 61.
derives_refl’ (lemma) \( A = B \rightarrow A \vdash B \).
drop_LOCAL \( n \) (tactic, where \( n : nat \)). Removes the \( n \)th entry of a the LOCAL block of a semax or ENTAIL precondition.
drop_LOCALs \( \ell_i ; \ell_j \) Removes variables \( \ell_i \) and \( \ell_j \) from the LOCAL block of a semax or ENTAIL precondition.
entailer (tactic; page 62, page 28) Proves (lifted or base-level) entailments, possibly leaving a residue for the user to prove.
entailer! (tactic; page 62, page 28) Like entailer, but faster and more powerful—however, it sometimes turns a provable goal into an unprovable goal.
Exists \( v \) (tactic; Chapter 22) Instantiate an EX existential on the right-hand side of an entailment.
field_at_conflict \( p \) \( fld \) (tactic) Solves an entailment of the form
\[ \ldots * \text{field_at} \ sh \ t \ fld \ v_1 \ p * \ldots * \text{field_at} \ sh \ t \ fld \ v_2 \ p * \ldots \vdash_- \]
based on the contradiction that the same field-assertion cannot *-separate. Usually invoked automatically by entailer (or entailer!)
to prove goals such as \((p<>q)\). Needs to be able to prove (or compute) the fact that \(0 < \text{sizeof} (\text{nested} \_\text{field} \_\text{type} t \_\text{fld})\); for data \_at\_conflict that’s equivalent to \(0 < \text{sizeof} t\).

**forward** (tactic; page 21) Do forward Hoare-logic proof through one C statement (assignment, break, continue, return).

**forward\_call ARGS** (tactic; page 37) Forward Hoare-logic proof through one C function-call, where ARGS is a witness for the WITH clause of the funspec.

**forward\_for** (tactic; page 79) Hoare-logic proof for the for statement, general case.

**forward\_for\_simple\_bound n Inv** (tactic; page 78) When a for-loop has the form for \((\text{init}; i < hi; i++]\), where \(n\) is the value of \(hi\), and \(Inv\) is the loop invariant.

**forward\_if Q** (tactic; page 24) Hoare-logic proof for the if statement, where \(Q\) may be omitted if at the end of a block, where the postcondition is already given.

**forward\_while Inv** (tactic; Chapter 12) Forward Hoare-logic proof of a while loop, with loop invariant \(Inv\).

**make\_compspecs prog** (tactic; page 13)

**mk\_varspecs prog** (tactic; page 13)

**mkConciseDelta V G F \(\Delta\)** (tactic; page 102) Applicable to a proof state with a semax goal. Simplifies the \(\Delta\) component to the application of a sequence of initializations to the host function’s func\_tycontext. Used to prepare the current proof goal for abstracting/factoring out as a separate lemma.

**name i _i** (tactic) Before start\_function, suggest the name \(i\) for the Coq variable associated with the value of C global variable \(_i\).

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

**start\_function** (tactic; Chapter 9) Unpack the funspec’s pre- and post-condition into a Hoare triple describing the function body.

**sublist\_split** (lemma; page 33) Break a sublist into the concatenation of two smaller sublists.
**unfold_data_at** (tactic; page 51) When $t$ is a struct (or array) type, break apart data_at $sh\ t\ v\ p$ into a separating conjunction of its individual fields (or array elements).

**unfold_field_at** (tactic; page 51) Like unfold_data_at, but starts with field_at $sh\ t\ path\ v\ p$.

**with_library** (tactic; Chapter 59) Complete the funspecs by inserting stub specifications for all unspecified functions; and (if **Import VST.floyd.library** is done) adding standard specifications for malloc, free, exit.