

# *Verifiable C*

*Applying the Verified Software Toolchain  
to C programs*

*Version 1.3.1  
September 30, 2014*

*Andrew W. Appel*  
*with Josiah Dodds*

# Contents

Verifiable C	i
Contents	ii
1 Getting started	4
2 Differences from PLCC	6
3 Memory predicates	7
4 Separation Logic	8
5 Mapsto and func_ptr	10
6 Shares	11
7 CompCert C	13
8 Verifiable C programming	14
9 32-bit Integers	15
10 Values	18
11 C types	19
12 C expression syntax	23
13 C operators	24
14 C expression evaluation	27
15 C type checking	29
16 Lifted separation logic	30
17 Canonical forms	32
18 Supercanonical forms	33
19 Go_lower	34
20 Normalize	35
21 Entailer	40
22 Cancel	41
23 The Hoare triple	42
24 Later	43
25 Specifying a function	44
26 Specifying all functions	46
27 Proving a function	47
28 Manipulating preconditions	51
29 The Frame rule	53

<b>30</b>	<b>Pointer comparisons</b>	<b>54</b>
<b>31</b>	<b>Structured data</b>	<b>55</b>
<b>32</b>	<b>Signed and unsigned integers</b>	<b>59</b>
<b>33</b>	<b>For loops</b>	<b>61</b>
<b>34</b>	<b>Nested Loads</b>	<b>63</b>
<b>35</b>	<b>Cygwin</b>	<b>68</b>

# 1 Getting started

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

*Program Logics for Certified Compilers*,

by Andrew W. Appel *et al.*, Cambridge University Press, 2014.

TO INSTALL THE VST SEPARATION LOGIC FOR C LIGHT:

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

See also the file `vst/BUILD_ORGANIZATION`.

---

<sup>1</sup>Microsoft Windows / Cygwin users: See Chapter 35 of this manual.

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

- Write a C program *F.c*.
- Run `clightgen F.c` to translate it into a Coq file *F.v*.
- Write a verification of *F.v* in a file such as `verif_F.v`. That latter file will import both *F.v* and the VST *Floyd*<sup>2</sup> program verification system, `floyd.proofauto`.

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

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

Before doing proofs of your own, you may find it helpful to step through this tutorial on C light expressions and assertions:

```
cd examples/floyd_tut; coqide tutorial.v
```

(this tutorial sets up its own load paths.)

---

<sup>2</sup>Named after Robert W. Floyd (1936–2001), a pioneer in program verification.

## 2 Differences from PLCC

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

- `field_mapsto` is now called `field_at`, and it is dependently typed; see [Chapter 31](#) of this manual.
- `typed_mapsto` is renamed to `data_at`, and last two arguments are swapped.
- `umapsto` (“untyped mapsto”) no longer exists.
- `mapsto  $\pi t v w$`  now permits either ( $w = \text{Vundef}$ ) or the value  $w$  belongs to type  $t$ . This permits describing uninitialized locations, i.e., `mapsto_  $\pi t v = \text{mapsto_ } \pi t v \text{ Vundef}$` . See [Chapter 31](#) of this manual.
- Supercanonical form is now suggested; see [Chapter 18](#) of this manual.
- For function calls, do not use `forward` (except to get advice about the witness type); instead, use `forward_call`. See page [50](#).
- C functions may now fall through the end of the function body, and this is (per the C semantics) equivalent to a `return;` statement.

### 3 *Memory predicates*

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

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

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

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

# 4 Separation Logic

8  
(see PLCC Chapter 12)

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



```

Class SepLog (A: Type) {ND: NatDed A} := mkSepLog {
  emp: A;
  sepcon: A → A → A;      (Notation * )
  wand: A → A → A;        (Notation -*; here written →*)
  ewand: A → A → A;       (no notation; here written ⇨)
  sepcon_assoc: ∀ P Q R, (P * Q) * R = P * (Q * R);
  sepcon_comm: ∀ P Q, P * Q = Q * P;
  wand_sepcon_adjoint: ∀ (P Q R: A), P * Q ⊢ R ↔ P ⊢ Q * R;
  sepcon_andp_prop: ∀ P Q R, P * (!Q && R) = !Q && (P * R);
  sepcon_derives: ∀ P P' Q Q' : A, P ⊢ P' → Q ⊢ Q' → P * Q ⊢ P' * Q';
  ewand_sepcon: ∀ (P Q R : A), (P * Q) ⇨ R = P ⇨ (Q ⇨ R);
  ewand_TT_sepcon: ∀ (P Q R: A),
    (P * Q) && (R ⇨ TT) ⊢ (P && (R ⇨ TT)) * (Q && (R ⇨ TT));
  exclude_elsewhere: ∀ P Q: A, P * Q ⊢ (P && (Q ⇨ TT)) * Q;
  ewand_conflict: ∀ P Q R, P * Q ⊢ FF → P && (Q ⇨ R) ⊢ FF
}.

```

```

Class Indir (A: Type) {ND: NatDed A} := mkIndir {
  later: A → A; (Notation ▷)
  now_later: ∀ P: A, P ⊢ ▷ P;
  later_K: ∀ P Q, ▷ (P → Q) ⊢ (▷ P → ▷ Q);
  later_allp: ∀ T (F: T → A), ▷ (ALL x:T, F x) = ALL x:T, ▷ (F x);
  later_exp: ∀ T (F: T → A), EX x:T, ▷ (F x) ⊢ ▷ (EX x: F x);
  later_exp': ∀ T (any:T) F, ▷ (EX x: F x) = EX x:T, ▷ (F x);
  later_imp: ∀ P Q, ▷ (P → Q) = (▷ P → ▷ Q);
  loeb: ∀ P, ▷ P ⊢ P → TT ⊢ P
}.

```

```

Class SepIndir (A: Type) {NA: NatDed A}{SA: SepLog A}{IA: Indir A} :=
  mkSepIndir {
    later_sepcon: ∀ P Q, ▷ (P * Q) = ▷ P * ▷ Q;
    later_wand: ∀ P Q, ▷ (P →* Q) = ▷ P →* ▷ Q;
    later_ewand: ∀ P Q, ▷ (P ⇨ Q) = (▷ P) ⇨ (▷ Q)
  }.

```

# 5 Mapsto and func\_ptr

10  
(see PLCC section 24)

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

**Parameter** address\_mapsto:

memory\_chunk  $\rightarrow$  val  $\rightarrow$  share  $\rightarrow$  share  $\rightarrow$  address  $\rightarrow$  mpred.

**Parameter** func\_ptr : funspec  $\rightarrow$  val  $\rightarrow$  mpred.

func\_ptr  $\phi$   $v$  means that value  $v$  is a pointer to a function with specification  $\phi$ .

address\_mapsto expresses what is typically written  $x \mapsto y$  in separation logic, that is, a singleton heap containing just value  $y$  at address  $x$ . But we almost always use one of the following derived forms:

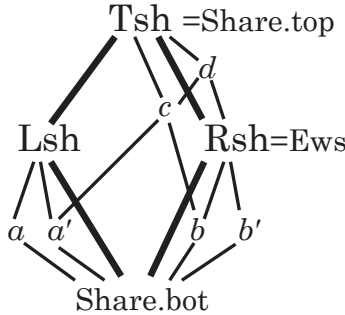
mapsto ( $\pi$ :share) ( $t$ :type) ( $v$  w: val) : mpred describes a singleton heap with just one value  $w$  of (C-language) type  $t$  at address  $v$ , with permission-share  $\pi$ .

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

field\_at ( $\pi$ : share) ( $t$ : type) ( $f$ : list ident) ( $w$ : retype (nested\\_field\\_type2  $f$ ) ( $v$ : val)) : mpred describes a heap that holds just field fld of struct-value  $v$ , belonging to struct-type  $t$ , containing value  $w$ . If type  $t$  describes a nested struct type, then  $f$  can actually be a path of field selections that descends into the nested structures. If  $f$  is the empty path, then the field is equivalent to data\_at. The type of  $w$  is a dependent type. *Note: arguments  $w, v$  are swapped compared to the PLCC book.*

field\_at\_( $\pi$ : share) ( $t$ : type) (fld: ident) ( $v$ : val) : mpred  
is the corresponding uninitialized structure-field.

The mapsto operator (and related operators) take a *permission share*, here written  $\pi$  and typically written sh in Coq, expressing whether the mapsto grants read permission, write permission, or some other fractional permission.



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

Share.split Tsh = (Lsh,Rsh)  
 Share.split Lsh = (a,a')      Share.split Rsh = (b,b')  
 $a' \oplus b = c$        $\text{lub}(c, \text{Rsh}) = a' \oplus \text{Rsh} = d$

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  $(\text{mapsto } \pi \ t \ v \ w)$  holds, then the share  $\pi$  must include at least a read share, thus this give permission to load memory at address  $v$  to get a value  $w$  of type  $t$ .

To make sure  $\pi$  has enough permission to write (i.e.,  $\text{Rsh} \sqsubseteq \pi$ , we can say `writable_share  $\pi$  : Prop`.

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

Sequential programs usually have little need of any shares except the  $\text{Tsh}$  and  $\text{Ews}$ . However, many function specifications can be parameterized over any share, and this sort of generalized specification makes the functions usable in more contexts.

In C it is undefined to test deallocated pointers for equality or inequalities, so the Hoare-logic rule for pointer comparison also requires some permission-share; see [page 54](#).

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

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

**Integers.** 32-bit (and 8-bit, 16-bit, 64-bit) signed/unsigned integers.

**Floats.** IEEE floating point numbers.

**Values.** The `val` type: integer + float + pointer + undefined.

**AST.** Generic support for abstract syntax.

**Ctypes.** C-language types and structure-field-offset computations.

**Cop.** Semantics of C-language arithmetic operators.

**Clight.** Abstract syntax of C-light expressions, statements, and functions.

**veric.expr.** (from VST, not CompCert) Semantics of expression evaluation.

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

# 8 Verifiable C programming 14

22 *See PLCC Chapter*

In writing Verifiable C programs you must:

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

The clightgen tool automatically:

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

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

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

# 9 32-bit Integers

15  
(compcert/lib/Integers.v)

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

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

**Definition** wordsize: nat := 32. (*\* also instantiations for 8, 16, 64 \**)

**Definition** modulus : Z := two\_power\_nat wordsize.

**Definition** half\_modulus : Z := modulus / 2.

**Definition** max\_unsigned : Z := modulus - 1.

**Definition** max\_signed : Z := half\_modulus - 1.

**Definition** min\_signed : Z := -half\_modulus.

**Parameter** int : Type.

**Parameter** unsigned : int → Z.

**Parameter** signed : int → Z.

**Parameter** repr : Z → int.

**Definition** zero := repr 0.

**Definition** eq (x y: int) : bool.

**Definition** lt (x y: int) : bool.

**Definition** ltu (x y: int) : bool.

**Definition** neg (x: int): int := repr (- unsigned x).

**Definition** add (x y: int): int := repr (unsigned x + unsigned y).

**Definition** sub (x y: int): int := repr (unsigned x - unsigned y).

**Definition** mul (x y: int): int := repr (unsigned x \* unsigned y).

**Definition** divs (x y: int) : int.

**Definition** mods (x y: int) : int.

**Definition** divu (x y: int) : int.

**Definition** modu (x y: int) : int.

**Definition** and (x y: int): int := bitwise\_binop andb x y.

**Definition** or (x y: int): int := bitwise\_binop orb x y.

**Definition** xor (x y: int) : int := bitwise\_binop xorb x y.

**Definition** not (x: int) : int := xor x mone.

**Definition** shl (x y: int): int.

**Definition** shru (x y: int): int.

**Definition** shr ( $x\ y: \text{int}$ ):  $\text{int}$ .

**Definition** rol ( $x\ y: \text{int}$ ) :  $\text{int}$ .

**Definition** ror ( $x\ y: \text{int}$ ) :  $\text{int}$ .

**Definition** rolm ( $x\ a\ m: \text{int}$ ):  $\text{int}$ .

**Definition** cmp ( $c: \text{comparison}$ ) ( $x\ y: \text{int}$ ) :  $\text{bool}$ .

**Definition** cmpu ( $c: \text{comparison}$ ) ( $x\ y: \text{int}$ ) :  $\text{bool}$ .

**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:  $\text{unsigned } \text{zero} = 0$ .

**Theorem** unsigned\_one:  $\text{unsigned } \text{one} = 1$ .

**Theorem** signed\_zero:  $\text{signed } \text{zero} = 0$ .

**Theorem** eq\_sym:  $\forall x\ y, \text{eq } x\ y = \text{eq } y\ x$ .

**Theorem** eq\_spec:  $\forall (x\ y: \text{int}), \text{if } \text{eq } x\ y \text{ then } x = y \text{ else } x <> y$ .

**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:  $\text{neg zero} = \text{zero}$ .

**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:  $\forall x, \text{sub zero } x = \text{neg } x$ .

**Theorem** sub\_add\_opp:  $\forall x y, \text{sub } x y = \text{add } x(\text{neg } y)$ .

**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:  $\forall x y, \text{mul } x y = \text{repr}(\text{signed } x * \text{signed } y)$ .

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

**Definition** block : Type := positive.

**Inductive** val: Type :=

| Vundef: val  
| Vint: int  $\rightarrow$  val  
| Vlong: int64  $\rightarrow$  val  
| Vfloat: float  $\rightarrow$  val  
| Vptr: block  $\rightarrow$  int  $\rightarrow$  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.

Vfloat( $f$ ) is an floating-point value, where  $f$  is a Flocq 64-bit floating-point number.

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  $(\text{Vptr } b\ z) + (\text{Vint } i) = \text{Vptr } b\ (z + i)$ . Of course, the C-language  $+$  operator first multiplies  $i$  by the size of the array-element that Vptr  $b\ z$  points to.

# 11 C types

(compcert/cfrontend/Ctypes.v)

**Inductive** signedness := Signed | Unsigned.

**Inductive** intsize := I8 | I16 | I32 | IBool.

**Inductive** floatsize := F32 | F64.

**Record** attr : Type := mk\_attr {  
   attr\_volatile: bool  
 }.

**Definition** noattr := { | attr\_volatile := false | }.

**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 → type  
 | Tstruct: ident → fieldlist → attr → type  
 | Tunion: ident → fieldlist → attr → type  
 | Tcomp\_ptr: ident → attr → type

**with** typelist : Type :=

| Tnil: typelist  
 | Tcons: type → typelist → typelist

**with** fieldlist : Type :=

| Fnil: fieldlist  
 | Fcons: ident → type → fieldlist → fieldlist.

**Definition** typeconv (ty: type) : type :=

**match** ty **with**

| Tint (I8 | I16 | IBool) \_a ⇒ Tint I32 Signed a  
 | Tarray t sz a ⇒ Tpointer t a  
 | Tfunction \_ ⇒ Tpointer ty noattr

```
| _ ⇒ ty
end.
```

```
Fixpoint alignof (t: type) : Z :=
  match t with
  | Tint l8 _ ⇒ 1
  | Tint l16 _ ⇒ 2
  | Tint l32 _ ⇒ 4
  | Tlong _ ⇒ 8
  | Tfloat F32 _ ⇒ 4
  | Tfloat F64 _ ⇒ 8
  | Tpointer _ ⇒ 4
  ... et cetera
end.
```

(\*\* Size of a type, in bytes. \*)

```
Fixpoint sizeof (t: type) : Z :=
  match t with
  | Tint l8 _ ⇒ 1
  | Tint l16 _ ⇒ 2
  | Tint l32 _ ⇒ 4
  | Tlong _ ⇒ 8
  | Tfloat F32 _ ⇒ 4
  | Tfloat F64 _ ⇒ 8
  | Tpointer _ ⇒ 4
  ... et cetera
end.
```

**Lemma** sizeof\_pos:  $\forall t, \text{sizeof } t > 0$ .

**Definition** field\_offset (id: ident) (fld: fieldlist) : res Z.

**Fixpoint** field\_type (id: ident) (fld: fieldlist) {struct fld} : res type.

**Inductive** mode: Type :=

| By\_value: memory\_chunk → mode  
 | By\_reference: mode  
 | By\_copy: mode  
 | By\_nothing: mode.

**Definition** access\_mode (ty: type) : mode :=

**match** ty **with**

| Tint I8 Signed \_ ⇒ By\_value Mint8signed  
 | Tint I8 Unsigned \_ ⇒ By\_value Mint8unsigned  
 | Tint I16 Signed \_ ⇒ By\_value Mint16signed  
 | Tint I16 Unsigned \_ ⇒ By\_value Mint16unsigned  
 | Tint I32 \_ ⇒ By\_value Mint32  
 | Tint IBool \_ ⇒ By\_value Mint8unsigned  
 | Tlong \_ ⇒ By\_value Mint64  
 | Tfloat F32 \_ ⇒ By\_value Mfloat32  
 | Tfloat F64 \_ ⇒ By\_value Mfloat64  
 | Tvoid ⇒ By\_nothing  
 | Tpointer \_ ⇒ By\_value Mint32  
 | Tarray \_\_\_ ⇒ By\_reference  
 | Tfunction \_ ⇒ By\_reference  
 | Tstruct \_\_\_ ⇒ By\_copy  
 | Tunion \_\_\_ ⇒ By\_copy  
 | Tcomp\_ptr \_ ⇒ By\_nothing

**end.**

CompCert handles self-referential structure types in the following way that deserves at least some explanation, not provided here:

```
Fixpoint unroll_composite (cid: ident) (comp: type) (ty: type) : type :=
  match ty with
  | Tvoid  $\Rightarrow$  ty
  | Tint ___  $\Rightarrow$  ty
  | Tlong __  $\Rightarrow$  ty
  | Tfloat __  $\Rightarrow$  ty
  | Tpointer t1 a  $\Rightarrow$  Tpointer (unroll_composite t1) a
  | Tarray t1 sz a  $\Rightarrow$  Tarray (unroll_composite t1) sz a
  | Tfunction t1 t2  $\Rightarrow$ 
    Tfunction (unroll_composite_list t1) (unroll_composite t2)
  | Tstruct id fld a  $\Rightarrow$ 
    if ident_eq id cid then ty
    else Tstruct id (unroll_composite_fields fld) a
  | Tunion id fld a  $\Rightarrow$ 
    if ident_eq id cid then ty
    else Tunion id (unroll_composite_fields fld) a
  | Tcomp_ptr id a  $\Rightarrow$ 
    if ident_eq id cid then Tpointer comp a else ty
  end
```

**with** unroll\_composite\_list cid comp (tl: typelist) : typelist := ...

**with** unroll\_composite\_fields cid comp (fld: fieldlist) : fieldlist := ...

**Lemma** alignof\_unroll\_composite:

$\forall$  cid comp ty, alignof (unroll\_composite cid comp ty) = alignof ty.

**Lemma** sizeof\_unroll\_composite:

$\forall$  cid comp ty, sizeof (unroll\_composite cid comp ty) = sizeof ty.

# 12 C expression syntax

(compcert/cfrontend/Clight.v)

**Inductive** `expr` : `Type` :=

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

**Definition** `typeof` (e: `expr`) : `type` :=

**match** e **with**

| `Econst_int` \_ty ⇒ ty  
 | `Econst_float` \_ty ⇒ ty  
 | `Evar` \_ty ⇒ ty  
 | ... *et cetera*.

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

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



```
| neg_default ⇒ None
end.
```

Function sem\_add (v1:val) (t1:type) (v2: val) (t2:type) : option val :=

```
match classify_add t1 t2 with
```

```
| add_case_ii sg ⇒ (**r integer addition *)
```

```
  match v1, v2 with
```

```
  | Vint n1, Vint n2 ⇒ Some (Vint (Int.add n1 n2))
```

```
  | -, _ ⇒ None
```

```
  end
```

```
| add_case_ff ⇒ (**r float addition *)
```

```
  match v1, v2 with
```

```
  | Vfloat n1, Vfloat n2 ⇒ Some (Vfloat (Float.add n1 n2))
```

```
  | -, _ ⇒ None
```

```
  end
```

```
| add_case_if sg ⇒ (**r int plus float *)
```

```
  match v1, v2 with
```

```
  | Vint n1, Vfloat n2 ⇒ Some (Vfloat (Float.add (cast_int_float sg n1) n2))
```

```
  | -, _ ⇒ None
```

```
  end
```

```
| ... (cases omitted)
```

```
| add_case_ip ty _ ⇒ (**r integer plus pointer *)
```

```
  match v1, v2 with
```

```
  | Vint n1, Vptr b2 ofs2 ⇒
```

```
    Some (Vptr b2 (Int.add ofs2 (Int.mul (Int.repr (sizeof ty)) n1)))
```

```
  | -, _ ⇒ None
```

```
  end
```

```
| add_default ⇒ None
```

```
end.
```

Function sem\_sub (v1:val) (t1:type) (v2: val) (t2:type) : option val.

Function sem\_mul (v1:val) (t1:type) (v2: val) (t2:type) : option val.

Function sem\_div (v1:val) (t1:type) (v2: val) (t2:type) : option val.

Function sem\_mod (v1:val) (t1:type) (v2: val) (t2:type) : option val.

Function sem\_and (v1:val) (t1:type) (v2: val) (t2:type) : option val.

```

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

```

```

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

```

# 14 C expression evaluation (vst/veric/expr.v)

**Definition** eval\_id (id: ident) ( $\rho$ : environ).  
*(\* look up the tempory variable ``id'' in  $\rho$  \*)*

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

**Definition** eval\_unop (op: Cop.unary\_operation) (t1 : type) (v1 : val) : val.

**Definition** eval\_binop (op: Cop.binary\_operation)  
 (t1 t2 : type) (v1 v2: val) : val.

**Definition** force\_ptr (v: val) : val :=  
**match** v **with** Vptr l ofs  $\Rightarrow$  v | \_  $\Rightarrow$  Vundef **end**.

**Definition** eval\_struct\_field (delta: Z) (v: val) : val.  
*(\* offset the pointer-value v by delta \*)*

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

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

**Definition** deref\_noload (ty: type) (v: val) : val.  
*(\* For By\_reference types such as arrays that dereference  
 without actually fetching \*)*  
**match** access\_mode ty **with** By\_reference  $\Rightarrow$  v | \_  $\Rightarrow$  Vundef **end**.

**Fixpoint** eval\_expr (e: expr) : environ  $\rightarrow$  val :=

**match** e **with**

| Econst\_int i ty  $\Rightarrow$  `(Vint i)  
 | Econst\_float f ty  $\Rightarrow$  `(Vfloat f)  
 | Etempvar id ty  $\Rightarrow$  eval\_id id  
 | Eaddrof a ty  $\Rightarrow$  eval\_lvalue a  
 | Eunop op a ty  $\Rightarrow$  `(eval\_unop op (typeof a)) (eval\_expr a)  
 | Ebinop op a1 a2 ty  $\Rightarrow$   
     `(eval\_binop op (typeof a1) (typeof a2))  
       (eval\_expr a1) (eval\_expr a2)  
 | Ecast a ty  $\Rightarrow$  `(eval\_cast (typeof a) ty) (eval\_expr a)  
 | Evar id ty  $\Rightarrow$  `(deref\_noload ty) (eval\_var id ty)  
 | Ederef a ty  $\Rightarrow$  `(deref\_noload ty) (`force\_ptr (eval\_expr a))  
 | Efield a i ty  $\Rightarrow$  `(deref\_noload ty)  
     (`(eval\_field (typeof a) i) (eval\_lvalue a))

**end**

**with** eval\_lvalue (e: expr) : environ  $\rightarrow$  val :=

**match** e **with**

| Evar id ty  $\Rightarrow$  eval\_var id ty  
 | Ederef a ty  $\Rightarrow$  `force\_ptr (eval\_expr a)  
 | Efield a i ty  $\Rightarrow$  `(eval\_field (typeof a) i) (eval\_lvalue a)  
 | \_  $\Rightarrow$  `Vundef

**end.**

# 15 C type checking

29  
(See PLCC Chapter 25)

Ideally, you will never notice the typechecker, but it may occasionally generate side conditions that can not be solved automatically. If you get a proof goal from the typechecker, it will be an entailment  $P \vdash \text{denote\_tc\_assert } (\dots)$ . PLCC Chapter Chapter 26 discusses what you can do to solve these goals.

If you are asked to prove an entailment where the typechecking condition evaluates to False, this is because your program is not written in Verifiable C. You may need to perform some local transformations on your C program in order to proceed. We listed these transformations on [page 14](#).

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

# 16 *Lifted separation logic* (See PLCC Chapter 21)

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

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

$$\begin{aligned}
 (P \&\& Q) : A \rightarrow B &:= \text{fun } a \Rightarrow Pa \&\& Qa \\
 (P \parallel Q) : A \rightarrow B &:= \text{fun } a \Rightarrow Pa \parallel Qa \\
 (\exists x. R(x)) : A \rightarrow B &:= \text{fun } a \Rightarrow \exists x. Rx a \\
 (\forall x. R(x)) : A \rightarrow B &:= \text{fun } a \Rightarrow \forall x. Rx a \\
 (P \longrightarrow Q) : A \rightarrow B &:= \text{fun } a \Rightarrow Pa \longrightarrow Qa \\
 (P \vdash Q) : A \rightarrow B &:= \forall a. Pa \vdash Qa \\
 (P * Q) : A \rightarrow B &:= \text{fun } a \Rightarrow Pa * Qa \\
 (P \multimap Q) : A \rightarrow B &:= \text{fun } a \Rightarrow Pa \multimap Qa
 \end{aligned}$$

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

**Instance**  $\text{LiftNatDed}(A B: \text{Type})\{\text{ND}: \text{NatDed } B\}: \text{NatDed } (A \rightarrow B) := \text{mkNatDed } (A \rightarrow B)$

```

(*andp*) (fun P Q x => andp (P x) (Q x))
(*orp*) (fun P Q x => orp (P x) (Q x))
(*exp*) (fun {T} (F: T -> A -> B) (a: A) => exp (fun x => F x a))
(*allp*) (fun {T} (F: T -> A -> B) (a: A) => allp (fun x => F x a))
(*imp*) (fun P Q x => imp (P x) (Q x))
(*prop*) (fun P x => prop P)
(*derives*) (fun P Q => forall x, derives (P x) (Q x))

```

-----

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

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

Consider this assertion:

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

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

```
`(mapsto π tint) (eval_id _x) (eval_id _y)
* `(mapsto π tint) (eval_id _u) `(Vint Int.zero)
```

Each of the first two backquotes lifts a function from type  $\text{val} \rightarrow \text{val} \rightarrow \text{mpred}$  to type  $(\text{environ} \rightarrow \text{val}) \rightarrow (\text{environ} \rightarrow \text{val}) \rightarrow (\text{environ} \rightarrow \text{mpred})$ , and the third one lifts from  $\text{val}$  to  $\text{environ} \rightarrow \text{val}$ .

# 17 Canonical forms

32  
(See PLCC section 26)

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

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

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

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

**Definition**  $\text{PROP}_x (P : \text{list Prop}) (Q : \text{assert}) :=$   
andp (prop (fold\_right and True P)) Q.

**Notation** "'PROP' ( x ; .. ; y ) z" :=  
(PROP<sub>x</sub> (cons x%type .. (cons y%type nil) ..) z) (at level 10) : logic.

**Notation** "'PROP' ( ) z" := (PROP<sub>x</sub> nil z) (at level 10) : logic.

**Definition**  $\text{LOCAL}_x (Q : \text{list (enviro} \rightarrow \text{Prop)}) (R : \text{assert}) :=$   
andp (local (fold\_right ( `and) ( `True) Q)) R.

**Notation** "'LOCAL' ( x ; .. ; y ) z" :=  
(LOCAL<sub>x</sub> (cons x%type .. (cons y%type nil) ..) z) (at level 9) : logic.

**Notation** "'LOCAL' ( ) z" := (LOCAL<sub>x</sub> nil z) (at level 9) : logic.

**Definition**  $\text{SEP}_x (R : \text{list assert}) : \text{assert} := \text{fold\_right sepcon emp } R$ .

**Notation** "'SEP' ( x ; .. ; y )" :=  
(SEP<sub>x</sub> (cons x%logic .. (cons y%logic nil) ..)) (at level 8) : logic.

**Notation** "'SEP' ( ) " := (SEP<sub>x</sub> nil) (at level 8) : logic.

**Notation** "'SEP' ( ) " := (SEP<sub>x</sub> nil) (at level 8) : logic.

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



# 18 Supercanonical forms

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

- every element of  $\vec{Q}$  has the form  $\text{`}(eq\ V)\ (\text{eval\_id}\ i)$ , where  $V$  is any Coq expression of type  $\text{val}$  and  $i$  is  $\beta\eta$ -equivalent to a constant (a ground term of type  $\text{ident}$ ).
- every element of  $\vec{R}$  is either  $\text{`}(E)$  where  $E$  is a Coq expression of type  $\text{mpred}$ , or  $\text{`}(F)\ (\text{evalvar}\ i\ t)$  where  $F$  is a Coq expression of type  $\text{val} \rightarrow \text{mpred}$  and  $i$  is  $\beta\eta$ -equivalent to a constant.

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

- In VST 1.3, forward through ordinary assignment statements (not loads/stores) is up to 10 times faster for supercanonical preconditions than for ordinary (canonical) preconditions.
- In future versions of VST, forward through other kinds of statements will be faster supercanonical preconditions.
- Future versions of the forward tactic may *require* the precondition to be in supercanonical form.

# 19 Go\_lower

An entailment  $\text{PROP}(\vec{P}) \text{LOCAL}(\vec{Q}) \text{SEP}(\vec{R}) \vdash \text{PROP}(\vec{P}') \text{LOCAL}(\vec{Q}') \text{SEP}(\vec{R}')$  is a sequent in our *lifted* separation logic; each side has type  $\text{environ} \rightarrow \text{mpred}$ . By definition of the lifted entailment  $\vdash$  it means exactly,  $\forall \rho. \text{PROP}(\vec{P}) \text{LOCAL}(\vec{Q}) \text{SEP}(\vec{R}) \rho \vdash \text{PROP}(\vec{P}') \text{LOCAL}(\vec{Q}') \text{SEP}(\vec{R}') \rho$ .

There are two ways to prove such an entailment: Explicitly introduce  $\rho$  (descend into an entailment on  $\text{mpred}$ ) and unfold the  $\text{PROP}/\text{LOCAL}/\text{SEP}$  form; or stay in canonical form and rewrite in the lifted logic. Either way may be appropriate; this chapter describes how to descend. The `go_lower` tactic, described on this page, is rarely called directly; it is the first step of the `entailer` tactic (page 40) when applied to lifted entailments.

The tactic `go_lower` tactic does the following:

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

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

## 20 Normalize

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

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

### Generic separation-logic simplifications.

$$\begin{array}{l}
 P * \text{emp} = P \qquad \text{emp} * P = P \qquad P \&\& \top = P \qquad \top \&\& P = P \\
 (EX x : A, P) * Q = EX x : A, P * Q \qquad P * (EX x : A, Q) = EX x : A, P * Q \\
 (EX x : A, P) \&\& Q = EX x : A, P \&\& Q \qquad P \&\& (EX x : A, Q) = EX x : A, P \&\& Q \\
 P * (!!Q \&\& R) = !!Q \&\& (P * R) \qquad (!!Q \&\& P) * R = !!Q \&\& (P * R) \qquad P \&\& \perp = \perp \\
 \perp \&\& P = \perp \qquad P * \perp = \perp \qquad \perp * P = \perp \qquad P \rightarrow (!!P \&\& Q = Q) \\
 P \rightarrow (!!P = \top) \qquad P \&\& P = P \qquad (EX \_ : \_, P) = P \qquad \text{local 'True} = \top
 \end{array}$$

### Unlifting.

$$\begin{array}{l}
 'f \ \rho = f \text{ [when } f \text{ has arity } 0] \qquad 'f \ a_1 \ \rho = f \ (a_1 \ \rho) \text{ [when } f \text{ has arity } 1] \\
 'f \ a_1 \ a_2 \ \rho = f \ (a_1 \ \rho) \ (a_2 \ \rho) \text{ [when } f \text{ has arity } 2, \text{ etc.]} \\
 \text{local } P \ \rho = !! (P \ \rho) \qquad (P * Q) \rho = P \rho * Q \rho \qquad (P \&\& Q) \rho = P \rho \&\& Q \rho \\
 (!!P) \rho = !!P \qquad !! (P \wedge Q) = !!P \&\& !!Q \\
 (EX x : A, P x) \rho = EX x : A, P x \rho \qquad '(EX x : B, P x) = EX x : B, '(P x) \\
 '(P * Q) = 'P * 'Q \qquad '(P \&\& Q) = 'P \&\& 'Q
 \end{array}$$

**Pulling nonspatial propositions out of spatial ones.**

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

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

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

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

**Canonical forms.**

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

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

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

**Modular Integer arithmetic.**

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

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

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

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

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

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

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

**Type checking and miscellaneous.**

$$\begin{aligned}
\text{tc\_formals}((i, t) :: r) &= \text{'and } ((\text{tc\_val } t) (\text{eval\_id } i) (\text{tc\_formals } r)) \\
\text{tc\_formals nil} &= \text{'T} \quad \text{tc\_andp tc\_TT } e = e \quad \text{tc\_andp } e \text{ tc\_TT} = e \\
\text{eval\_id } x (\text{env\_set } \rho \ x \ v) &= v \\
x \neq y &\rightarrow (\text{eval\_id } x (\text{env\_set } \rho \ y \ v) = \text{eval\_id } x \ v) \\
\text{isptr } v &\rightarrow (\text{eval\_cast\_neutral } v = v) \\
(\exists t. \text{tc\_val } t \ v \wedge \text{is\_pointer\_type } t) &\rightarrow (\text{eval\_cast\_neutral } v = v)
\end{aligned}$$
**Expression evaluation. (autorewrite with eval, but in fact these are usually handled just by simpl or unfold.)**

$$\begin{aligned}
&\text{deref\_noload}(\text{tarray } t \ n) = (\text{fun } v \Rightarrow v) \\
\text{eval\_expr}(\text{Etempvar } i \ t) &= \text{eval\_id } i \quad \text{eval\_expr}(\text{Econst\_int } i \ t) = \text{'(Vint } i) \\
&\text{eval\_expr}(\text{Ebinop } op \ a \ b \ t) = \\
&\quad \text{'(eval\_binop } op \ (\text{typeof } a) \ (\text{typeof } b)) (\text{eval\_expr } a) (\text{eval\_expr } b) \\
\text{eval\_expr}(\text{Eunop } op \ a \ t) &= \text{'(eval\_unop } op \ (\text{typeof } a)) (\text{eval\_expr } a) \\
\text{eval\_expr}(\text{Ecast } e \ t) &= \text{'(eval\_cast}(\text{typeof } e) \ t) (\text{eval\_expr } e) \\
\text{eval\_lvalue}(\text{Ederef } e \ t) &= \text{'force\_ptr } (\text{eval\_expr } e)
\end{aligned}$$

**Structure fields.**

$$\text{field\_mapsto } \pi \ t \ fl d \ (\text{force\_ptr } v) = \text{field\_mapsto } \pi \ t \ fl d \ v$$

$$\text{field\_mapsto\_ } \pi \ t \ fl d \ (\text{force\_ptr } v) = \text{field\_mapsto\_ } \pi \ t \ fl d \ v$$

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

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

$$\text{memory\_block } \pi \ \text{Int.zero} \ (\text{Vptr } b \ z) = \text{emp}$$
**Postconditions.** (autorewrite **with** `ret_assert.`)
$$\text{normal\_ret\_assert } \perp \ ek \ vl = \perp$$

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

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

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

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

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

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

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

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

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

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

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

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

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

**Function return values.**

```

bind_ret (Some v) t Q = (!!tc_val t v && 'Q(make_args(ret_temp :: nil) (v ::
  nil)))      make_args' σ a ρ = make_args (map fst (fst σ)) (a ρ) ρ
              make_args(i :: l)(v :: r)ρ = env_set(make_args(l)(r)ρ) i v
make_args nil nil = globals_only    get_result(Some x) = get_result1(x)
retval(get_result1 i ρ) = eval_id i ρ    retval(env_set ρ ret_temp v) = v
retval(make_args(ret_temp :: nil) (v :: nil) ρ) = v
ret_type(initialized i Δ) = ret_type(Δ)

```

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

$$\begin{array}{llll}
P \vdash \top & \perp \vdash P & P \vdash P * \top & (\forall x. (P \vdash Q)) \rightarrow (EX x : A, P \vdash Q) \\
(P \rightarrow (\top \vdash Q)) \rightarrow (!!P \vdash Q) & (P \rightarrow (Q \vdash R)) \rightarrow (!!P \&\& Q \vdash R)
\end{array}$$

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

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

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

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



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

When we say  $A_i$  is  $\beta\eta$  equivalence to  $A'_i$ , that is equivalent to saying that (change  $(A_i)$  **with**  $(A'_i)$ ) would succeed.

If the goal has the form  $(A_1 * A_2) * ((A_3 * A_4) * A_5) \vdash (A'_4 * B_1 * A'_1) * B_2$  where there is only a partial match, then cancel will remove the matching conjuncts and leave a subgoal such as  $A_2 * A_3 * A_5 \vdash B_1 * B_2$ .

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

If the goal has the form

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


---


$$(A_1 * A_2) * ((A_3 * A_4) * A_5) \vdash A'_4 * (\text{fold\_right sepcon emp } F) * A'_1$$

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  $F$ , 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  $\text{evvar}(F : \text{list}(\text{environ} \rightarrow \text{mpred}))$ . This is typically done automatically, as part of forward symbolic execution through a function call.

# 23 The Hoare triple

42  
(See PLCC Chapter 24)

In the judgment  $\Delta \vdash \{P\} c \{R\}$ , written in Coq as

`semax ( $\Delta$ : tycontext) ( $P$ : environ $\rightarrow$ mpred) ( $c$ : statement) ( $R$ : ret_assert)`

$\Delta$  is a *type context*, giving types of function parameters, local variables, and global variables; and giving *specifications* (funspec) of global functions.

$P$  is the precondition;

$c$  is a command in the C language; and

$R$  is the postcondition. Because a  $c$  statement can exit in different ways (fall-through, continue, break, return), a `ret_assert` has predicates for all of these cases.

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

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

**Axiom** `semax.set.forward`:  $\forall \Delta \ P \ (x: \text{ident}) \ (e: \text{expr})$ ,  
`semax  $\Delta \ (\triangleright \ (\text{local} \ (\text{tc\_expr} \ \Delta \ e) \ \&\&$`   
`$\text{local} \ (\text{tc\_temp\_id} \ \text{id} \ (\text{typeof} \ e) \ \Delta \ e) \ \&\& \ P))$`   
`( $\text{Sset} \ x \ e$ )`  
`( $\text{normal\_ret\_assert}$`   
`( $\text{EX} \ \text{old}:\text{val}$ ,`  
`$\text{local} \ (\text{`eq} \ (\text{eval\_id} \ x) \ (\text{subst} \ x \ (\text{`old}) \ (\text{eval\_expr} \ e))) \ \&\&$`   
`$\text{subst} \ x \ (\text{`old}) \ P$ )).`

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

Many of the Hoare rules, such as the one on [page 42](#),

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

have the operator  $\triangleright$  (pronounced “later”) in their precondition.

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

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

`local (tc_expr  $\Delta$  e) &&  $\triangleright$ local (tc_temp_id id t  $\Delta$  e) && ( $P_1 * \triangleright P_2$ )`

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

`$\triangleright$ (local (tc_expr  $\Delta$  e) && local (tc_temp_id id t  $\Delta$  e) && ( $P_1 * P_2$ ))`

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

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

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

## 25 Specifying a function 44 (See PLCC Chapter 27)

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

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

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

**Definition**  $F_{\text{spec}} :=$

```
DECLARE _F
  WITH  $a_1 : \tau_1, \dots a_k : \tau_k$ 
  PRE [  $x_1$  OF  $t_1, \dots, x_n$  OF  $t_n$  ]  $P$ 
  POST [  $t_{\text{ret}}$  ]  $Q$ .
```

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

**Definition** sumlist\_spec :=

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

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

**Definition** sumlist\_funspec : funspec :=

```
  WITH sh : share, contents : list int, p: val,
```

```

PRE [ _p OF (tptr t_struct_list)]
  local `(eq p) (eval_id _p))
  && `(lseg LS sh contents p nullval)
POST [ tint ]
  local `(eq (Vint (sum_int contents))) retval)
  && `(lseg LS sh contents p nullval).

```

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

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

**Definition** `sumlist_spec :=`  
`DECLARE _sumlist`  
`WITH sh : share, contents : list int, p: val,`  

```

PRE [ _p OF (tptr t_struct_list)]
  PROP() LOCAL `(eq p) (eval_id _p))
  SEP `(lseg LS sh contents p nullval))
POST [ tint ]
  local `(eq (Vint (sum_int contents))) retval)
  && `(lseg LS sh contents p nullval).

```

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

# 26 Specifying all functions

Chapter 27

46  
(See PLCC

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

$$\Gamma : \text{list} (\text{ident} * \text{funspec}) := (\iota_1, \phi_1) :: (\iota_2, \phi_2) :: (\iota_3, \phi_3) :: (\iota_4, \phi_4) :: \text{nil}.$$

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

$$V : \text{list} (\text{ident} * \text{type}) := (x_1, t_1) :: (x_2, t_2) :: \text{nil}$$

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

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

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

**Lemma**  $\text{body\_}F : \text{semax\_body } V \ \Gamma \ \text{f\_}F \ F\_spec.$

The predicate `semax_body` states that function  $F$ 's *implementation* (function body) meets its *specification* (`funspec`). The definition of the predicate, written in `veric/SeparationLogic.v`, basically states the Hoare triple of the function body,  $\Delta \vdash \{Pre\} c \{Post\}$ , where  $Pre$  and  $Post$  are taken from the `funspec` for  $f$ ,  $c$  is the `fn_body` of the function  $F$ , 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—unfolding certain definitions, destructing certain tuples, and putting the precondition in canonical form.

**Lemma** `body_F`: `semax_body V  $\Gamma$  f_F F_spec`.

**Proof.**

`start_function.`

`name x _x.`

`name y _y.`

`name z _z.`

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

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

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

We prove such judgments as follows:

1. Manipulate the precondition  $\text{PROP}(\vec{P})\text{LOCAL}(\vec{Q})\text{SEP}(\vec{R})$  until it takes a form suitable for forward symbolic execution through the first statement in the command  $c$ . (In this we are effectively using the rule of consequence.)

2. Apply a forward tactic to step into  $c$ . This will produce zero or more entailments  $A \vdash B$  to prove, where  $A$  is in canonical form; and zero or more semax judgments to prove.
3. Prove the entailments, typically using `go_lower`; prove the judgment, i.e., back to step 1.

Each kind of C command has different requirements on the form of the precondition, for the forward tactic to succeed. In each of the following cases, the expression  $E$  must not contain loads, stores, side effects, function calls, or pointer comparisons. The variable  $x$  must be a nonaddressable local variable.

$c_1; c_2$  Sequencing of two commands. The forward tactic will work on  $c_1$  first.

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

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

$x = *E$ ; Memory load. The SEP component of the precondition must contain an item of the form `(mapsto  $\pi$  t) e v`, where  $e$  is equivalent to `(eval_expr E)`. For example, if  $E$  is just an identifier `(Etempvar _y t)`, then  $e$  could be either `(eval_expr (Etempvar _y t))` or `(eval_id _y)`.

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

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



- $*E_1 = E_2$ ; Memory store. The SEP component of the precondition must contain an item of the form  $\text{`mapsto } \pi \ t) \ e_1 \ \vee$  or an item  $\text{`mapsto } \pi \ t) \ e_1$ , where  $e_1$  is equivalent to  $(\text{eval\_expr } E_1)$ .
- $\mathbf{a}[E_1] = E_2$ ; Array store. This is equivalent to  $\mathbf{*(a + E_1) = E_2}$ ; and is handled by the previous case.
- $E_1 \rightarrow fld = E_2$ ; Field store. This can be handled by the general store case, but a special-purpose field-store rule is easier to use. The SEP component of the precondition must contain either  $\text{`field.at } \pi \ t \ fld) \ \vee \ e_1$  or  $\text{`field\_mapsto } \pi \ t \ fld) \ e_1$ , where  $t$  is the structure type to which the field  $fld$  belongs, and  $e_1$  is equivalent to  $(\text{eval\_expr } E_1)$ . The share  $\pi$  must be strong enough to grant write permission, that is,  $\text{writable\_share}(\pi)$ .
- $x = E_1 \text{ op } E_2$ ; If  $E_1$  or  $E_2$  evaluate to *pointers*, and  $\text{op}$  is a comparison operator ( $=, !=, <, <=, >, \geq$ ), then  $E_1 \text{ op } E_2$  must not occur except in this special-case assignment rule. When  $E_1$  and  $E_2$  both have numeric values, the ordinary *assignment statement* rule applies.

Pointer comparisons are tricky in CompCert C for reasons explained at PLCC page 249; the program logic uses the `semax_ptr_compare` rule (PLCC page 164). After applying the forward tactic, the user will be left with some proof obligations: Prove that both  $E_1$  and  $E_2$  evaluate to allocated locations (i.e., that the precondition implies  $E_1 \xrightarrow{\pi_1} \_ * TT$  and also implies  $E_2 \xrightarrow{\pi_2} \_ * TT$ , for any  $\pi_1$  and  $\pi_2$ ). If the comparison is any of  $>, <, \geq, \leq$ , prove that  $E_1$  and  $E_2$  both point within the same allocated object. These are preconditions for even being permitted to test the pointers for equality (or inequality). See also [page 54](#).

- if (E)  $C_1$  else  $C_2$**  No restrictions on the form of the precondition. forward will create 3 subgoals: (1) prove that the precondition entails  $\text{tc\_expr } \Delta E$ . For many expressions  $E$ , the condition  $\text{tc\_expr } \Delta E$  is simply  $\text{TT}$ , which is trivial to prove. (2) the **then** clause... (3) the **else** clause... .

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

- return E**; No special precondition, except that the presence/absence of  $E$  must match the nonvoid/void return type of the function. The proof goal left by forward is to show that the precondition (with appropriate

substitution for the abstract variable `ret_var`) entails the function's postcondition.

$x = f(a_1, \dots, a_n)$ ; For a function call, use `forward_call(W)`, where  $W$  is a witness, a tuple corresponding (componentwise) to the `WITH` clause of the function specification. (If you do just forward, you'll get a message with advice about the *type* of  $W$ .)

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

## 28 Manipulating preconditions

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

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

then instantiate the existentials; this finally gives us

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

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

Faced with the proof goal  $\text{semax } \Delta \text{ (PROP}(\vec{P})\text{LOCAL}(\vec{Q})\text{SEP}(\vec{R})) \text{ } c \text{ } Post$  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 `semax_pre` **with**  $\text{PROP}(\vec{P}')\text{LOCAL}(\vec{Q}')\text{SEP}(\vec{R}')$ ,  
then prove  $\vec{P}; \vec{Q}; \vec{R} \vdash \vec{P}'; \vec{Q}'; \vec{R}'$  using `go_lower` ([page 34](#)).
- Use the rule of consequence implicitly, by using tactics that modify the precondition (and may leave entailments for you to prove).
- Do rewriting in the precondition, either directly by the standard `rewrite` and `change` tactics, or by `normalize`.
- Extract propositions and existentials from the precondition, by using `normalize` (or by applying the rules `extract_exists_pre` and `semax_extract_PROP`).

**TACTICS FOR MANIPULATING PRECONDITIONS.** In many of these tactics we select specific 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,... 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:

$\text{focus\_SEP } i \ j \ k$ . Bring items  $\#i, j, k$  to the front of the SEP list.

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

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

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

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

$\text{gather\_SEP } i \ j \ k$ . Bring items  $\#i, j, k$  to the front of the SEP list and conjoin them into a single element.

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

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

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

$\text{replace\_SEP } i \ R$ . Replace the  $i$ th element the SEP list with the assertion  $R$ , and leave a subgoal to prove.

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

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

$\text{replace\_in\_pre } S \ S'$ . 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.

$\text{frame\_SEP } i \ j \ k$ . Apply the frame rule, keeping only elements  $i, j, k$  of the SEP list. See Chapter 29.

## 29 The Frame rule

Separation Logic supports the Frame rule,

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

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

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

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 *semax\_seq'* to peel off the first command  $(c_1; c_2)$  in the new sequence:

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

$\text{semax } \Delta' \text{ ?88 } c_3 \text{ Post}$

Then *frame\_SEP* 0 2 to retain only  $R_0; R_2$ .

$\text{semax } \Delta \text{ PROP}(\vec{P})\text{LOCAL}(\vec{Q})\text{SEP}(R_0; R_2) \ c_1; c_2 \ \dots$

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.

## 30 *Pointer comparisons*

54  
See PLCC Chapter 25

Pointer comparisons can be split into two cases:

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

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

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

## 31 Structured data

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

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

```
mapsto  $\pi$  tint p h *
  mapsto  $\pi$  (tptr t_struct.list) (offset_val p (Vint (Int.repr 4))) t

field_at  $\pi$  t_struct.list (_head::nil) h p * field_at  $\pi$  t_struct.list (_tail::nil) t p

data_at  $\pi$  t_struct.list (h,t) p

field_at  $\pi$  t_struct.list nil (h,t) p
```

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

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

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

When  $\tau$  is a struct type and  $n$  is a nat, the tactic `unfold_data_at n` unfolds the  $n$ th occurrence of `data_at  $\pi$   $\tau$`  to a series of `field_at  $\pi$   $\tau$  ( $f::nil$ )`, where the  $f$  are the various fields of the struct  $\tau$ . For example, it would unfold the

third assertion above to look like the second one.

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

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

```
`(data_at  $\pi$  t_struct_list (h,t)) (eval_id _p) tptr t_struct_list))
```

which simplify to

```
`(field_at  $\pi$  t_struct_list (_head::nil) h * field_at  $\pi$  t_struct_list (_tail::nil) t)
  (eval_id _p (tptr t_struct_list))
```

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

PARTIALLY INITIALIZED DATA STRUCTURES. Consider the program

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

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



```

field_at Tsh t_struct_list (_head::nil) Vundef p
      * field_at Tsh t_struct_list (_tail::nil) Vundef p
field_at_ Tsh t_struct_list (_head::nil) p * field_at_ Tsh t_struct_list (_tail::nil) p
data_at Tsh t_struct_list (Vundef,Vundef) p
data_at_ Tsh t_struct_list p

```

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

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

```

field_at Tsh t_struct_list (_head::nil) (Vint(Int.repr 3)) p
      * field_at Tsh t_struct_list (_tail::nil) Vundef p
field_at Tsh t_struct_list (_head::nil) (Vint(Int.repr 3))
      * field_at_ Tsh t_struct_list (_tail::nil) p
data_at Tsh t_struct_list (Vint(Int.repr 3), Vundef) p

```

At point 3, we can write either of,

```

field_at Tsh t_struct_list (_head::nil) (Vint(Int.repr 3)) p
      * field_at Tsh t_struct_list (_tail::nil) (Vint Int.zero) p
data_at Tsh t_struct_list (Vint(Int.repr 3), Vint Int.zero) p

```

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

```

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

```

If  $p \rightarrow \text{head}$  and  $p \rightarrow \text{tail}$  may be uninitialized, this is fine. But if the structure is known to be initialized, the precondition as written does not express this

fact. One would need to add the conjunct  
 $!!(\text{is\_int } (\text{fst data}) \wedge \text{is\_pointer\_or\_null } (\text{snd data}))$   
 at the point marked  $(**)$ .

The function  $\text{reptype}' : \text{type} \rightarrow \text{Type}$  expresses the type of *initialized* data structures. For example,  $\text{reptype}' \text{ t\_struct\_list}$  is  $(\text{int} * \text{val})$ . The function  
 $\text{repinj } (t : \text{type}) : \text{reptype}' t \rightarrow \text{reptype } t$

expresses injections from (possibly) undefined to defined values. Suppose  
 $(h : \text{int}, t : \text{val})$  is a value of type  $\text{reptype}' \text{ t\_struct\_list}$ . Then  
 $\text{repinj } \text{t\_struct\_list } (h, t) = (\text{Vint } h, t)$

Using  $\text{reptype}'$  one could write,

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

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

## 32 Signed and unsigned integers

Mathematical proofs use the mathematical integers (the  $\mathbb{Z}$  type in Coq); C programs use 32-bit signed or unsigned integers. They are related as follows:

`Int.repr`:  $\mathbb{Z} \rightarrow \text{int}$ .

`Int.unsigned`:  $\text{int} \rightarrow \mathbb{Z}$ .

`Int.signed`:  $\text{int} \rightarrow \mathbb{Z}$ .

with the following lemmas:

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

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

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

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

`Int.repr` truncates to a 32-bit two's-complement representation (losing information if the input is out of range). `Int.signed` and `Int.unsigned` are different injections back to  $\mathbb{Z}$  that never lose information.

When doing proofs about integers, the recommended proof technique is to make sure your integers never overflow. That is, if the C variable `_x` contains the value `Vint (Int.repr x)`, then make sure `x` is in the appropriate range. Let's assume that `_x` is a signed integer, i.e. declared in C as `int x`; then the hypothesis is,

H:  $\text{Int.min\_signed} \leq x \leq \text{Int.max\_signed}$

If you maintain this hypothesis “above the line”, then the `normalize` tactic can automatically rewrite with `Int.signed (Int.repr x) = x`. Also, to solve goals such as,

```

...
H2 : 0 <= n <= Int.max_signed
...
-----
Int.min_signed <= 0 <= n

```

you can use the `reple_signed` tactic, which is basically just `omega` with knowledge of the values of `Int.min_signed`, `Int.max_signed`, and `Int.max_unsigned`.

To take advantage of this, put conjuncts into the `PROP` part of your function precondition such as  $0 \leq i < n$ ;  $n \leq \text{Int.max\_signed}$ . 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 array size and index (variables `size` and `i`) are kept within bounds; but the *contents* of the array might overflow when added up, which is why `add_elem` uses `Int.add` instead of `Z.add`.

## 33 For loops

The C-language for loop has the general form,

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

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

`forward_for` *Inv PreIncr PostCond*

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

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

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

*The following feature will appear in VST version 1.5.*

Many for-loops have this special form, *for* (*init*; *id* < *hi*; *id*++) *body* such that the expression *hi* will evaluate to the same value every time around the loop. This upper-bound expression need not be a literal constant, it just needs to be invariant. Then you can use the tactic,

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

where *n* is the upper bound: a Coq value of type *Z* such that *hi* will evaluate to *n*. The loop invariant is given by the expression (EX *i*:Z, PROP( $\vec{P}$ ) LOCAL( $\vec{Q}$ ) SEP( $\vec{R}$ ), where *i* is the value (in each iteration) of the loop iteration variable *id*. This tactic generates simpler subgoals than the general `forward_for` tactic.

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

## 34 Nested Loads

*This experimental feature will appear in VST release 1.5.*

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

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

$\text{rel\_expr} : \text{expr} \rightarrow \text{val} \rightarrow \text{rho} \rightarrow \text{mpred}$ .

$\text{rel\_lvalue} : \text{expr} \rightarrow \text{val} \rightarrow \text{rho} \rightarrow \text{mpred}$ .

The assertion  $\text{rel\_expr } e \ v \ \rho$  says, “expression  $e$  evaluates to value  $v$  in environment  $\rho$  and in the current memory.” The  $\text{rel\_lvalue}$  evaluates the expression as an  $l$ -value, to a pointer to the data.

Evaluation rules for  $\text{rel\_expr}$  are listed here:

$\text{rel\_expr\_const\_int} : \quad \forall (i : \text{int}) \ \tau \ (P : \text{mpred}) \ (\rho : \text{environ}),$

$P \vdash \text{rel\_expr} \ (\text{Econst\_int } i \ \tau) \ (\text{Vint } i) \ \rho.$

$\text{rel\_expr\_const\_float} : \quad \forall (f : \text{float}) \ \tau \ P \ (\rho : \text{environ}),$

$P \vdash \text{rel\_expr} \ (\text{Econst\_float } f \ \tau) \ (\text{Vfloat } f) \ \rho.$

$\text{rel\_expr\_const\_long} : \quad \forall (i : \text{int64}) \ \tau \ P \ \rho,$

$P \vdash \text{rel\_expr} \ (\text{Econst\_long } i \ \tau) \ (\text{Vlong } i) \ \rho.$

$\text{rel\_expr\_tempvar} : \quad \forall (\text{id} : \text{ident}) \ \tau \ (v : \text{val}) \ P \ \rho,$

$\text{Map.get} \ (\text{te\_of } \rho) \ \text{id} = \text{Some } v \rightarrow$

$P \vdash \text{rel\_expr} \ (\text{Etempvar } \text{id} \ \tau) \ v \ \rho.$

$\text{rel\_expr\_addrof} : \quad \forall (e : \text{expr}) \ \tau \ (v : \text{val}) \ P \ \rho,$

$P \vdash \text{rel\_lvalue } e \ v \ \rho \rightarrow$

$P \vdash \text{rel\_expr} \ (\text{Eaddrof } e \ \tau) \ v \ \rho.$

$\text{rel\_expr\_unop} : \quad \forall P \ (e_1 : \text{expr}) \ (v_1 \ v : \text{val}) \ \tau \ \text{op} \ \rho,$

$P \vdash \text{rel\_expr } e_1 \ v_1 \ \rho \rightarrow$

$\text{Cop.sem.unary\_operation } \text{op} \ v_1 \ (\text{typeof } e_1) = \text{Some } v \rightarrow$

$P \vdash \text{rel\_expr } (\text{Eunop } op \ e_1 \ \tau) \ v \ \rho.$   
 $\text{rel\_expr\_binop: } \quad \forall (e_1 \ e_2 : \text{expr}) \ (v_1 \ v_2 \ v : \text{val}) \ \tau \ op \ P \ \rho,$   
 $P \vdash \text{rel\_expr } e_1 \ v_1 \ \rho \rightarrow$   
 $P \vdash \text{rel\_expr } e_2 \ v_2 \ \rho \rightarrow$   
 $(\forall \ m : \text{Memory.Mem.mem},$   
 $\quad \text{Cop.sem.binary\_operation } op \ v_1 \ e \ (\text{typeof } e_1) \ v_2 \ (\text{typeof } e_2) \ m = \text{Some } v) \rightarrow$   
 $P \vdash \text{rel\_expr } (\text{Ebinop } op \ e_1 \ e_2 \ \tau) \ v \ \rho.$   
 $\text{rel\_expr\_cast: } \quad \forall (e_1 : \text{expr}) \ (v_1 \ v : \text{val}) \ \tau \ P \ \rho,$   
 $P \vdash \text{rel\_expr } e_1 \ v_1 \ \rho \rightarrow$   
 $\text{Cop.sem.cast } v_1 \ (\text{typeof } e_1) \ \tau = \text{Some } v \rightarrow$   
 $P \vdash \text{rel\_expr } (\text{Ecast } e_1 \ \tau) \ v \ \rho.$   
 $\text{rel\_expr\_lvalue: } \quad \forall (a : \text{expr}) \ (\text{sh} : \text{Share.t}) \ (v_1 \ v_2 : \text{val}) \ P \ \rho,$   
 $P \vdash \text{rel\_lvalue } a \ v_1 \ \rho \rightarrow$   
 $P \vdash \text{mapsto } sh \ (\text{typeof } a) \ v_1 \ v_2 * \text{TT} \rightarrow$   
 $v_2 <> \text{Vundef} \rightarrow$   
 $P \vdash \text{rel\_expr } a \ v_2 \ \rho.$   
 $\text{rel\_lvalue\_local: } \quad \forall (\text{id} : \text{ident}) \ \tau \ (b : \text{block}) \ P \ \rho,$   
 $P \vdash !!(\text{Map.get } (\text{ve\_of } \rho) \ \text{id} = \text{Some } (b, \tau)) \rightarrow$   
 $P \vdash \text{rel\_lvalue } (\text{Evar } \text{id} \ \tau) \ (\text{Vptr } b \ \text{Int.zero}) \ \rho.$   
 $\text{rel\_lvalue\_global: } \quad \forall (\text{id} : \text{ident}) \ \tau \ (v : \text{val}) \ P \ \rho,$   
 $P$   
 $\vdash !!(\text{Map.get } (\text{ve\_of } \rho) \ \text{id} = \text{None} \wedge$   
 $\quad \text{Map.get } (\text{ge\_of } \rho) \ \text{id} = \text{Some } (v, \tau)) \rightarrow$   
 $P \vdash \text{rel\_lvalue } (\text{Evar } \text{id} \ \tau) \ v \ \rho.$   
 $\text{rel\_lvalue\_deref: } \quad \forall (a : \text{expr}) \ (b : \text{block}) \ (z : \text{int}) \ \tau \ P \ \rho,$   
 $P \vdash \text{rel\_expr } a \ (\text{Vptr } b \ z) \ \rho \rightarrow$   
 $P \vdash \text{rel\_lvalue } (\text{Ederef } a \ \tau) \ (\text{Vptr } b \ z) \ \rho.$   
 $\text{rel\_lvalue\_field\_struct: } \quad \forall (i \ \text{id} : \text{ident}) \ \tau \ e \ (b : \text{block}) \ (z : \text{int}) \ (\text{fList} : \text{fieldlist}) \ a$   
 $\text{typeof } e = \text{Tstruct } \text{id} \ \text{fList} \ \text{att} \rightarrow$   
 $\text{field\_offset } i \ \text{fList} = \text{Errors.OK } \delta \rightarrow$   
 $P \vdash \text{rel\_expr } e \ (\text{Vptr } b \ z) \ \rho \rightarrow$   
 $P \vdash \text{rel\_lvalue } (\text{Efield } e \ i \ \tau) \ (\text{Vptr } b \ (\text{Int.add } z \ (\text{Int.repr } \delta))) \ \rho.$



The primitive nested-load assignment rule is,

**Axiom** `semax_loadstore`:

$$\begin{aligned} &\forall v0\ v1\ v2\ \Delta\ e1\ e2\ sh\ P\ P', \\ &\quad \text{writable\_share}\ sh \rightarrow \\ &\quad P \vdash !!\ (\text{tc\_val}\ (\text{typeof}\ e1)\ v2) \\ &\quad \quad \&\&\ \text{rel\_lvalue}\ e1\ v1 \\ &\quad \quad \&\&\ \text{rel\_expr}\ (\text{Ecast}\ e2\ (\text{typeof}\ e1))\ v2 \\ &\quad \quad \&\&\ (\text{mapsto}\ sh\ (\text{typeof}\ e1)\ v1\ v0) * P') \rightarrow \\ &\text{semax}\ \Delta\ (\triangleright P)\ (\text{Sassign}\ e1\ e2) \\ &\quad (\text{normal\_ret\_assert}\ (\text{mapsto}\ sh\ (\text{typeof}\ e1)\ v1\ v2) * P')). \end{aligned}$$

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

**Lemma** `semax_loadstore_array`:

$$\begin{aligned} &\forall n\ vi\ lo\ hi\ t1\ (\text{contents: } Z \rightarrow \text{reptype}\ t1)\ v1\ v2\ \Delta\ e1\ ei\ e2\ sh\ P\ Q\ R, \\ &\quad \text{reptype}\ t1 = \text{val} \rightarrow \\ &\quad \text{type\_is\_by\_value}\ t1 \rightarrow \\ &\quad \text{legal\_alignas.type}\ t1 = \text{true} \rightarrow \\ &\quad \text{typeof}\ e1 = \text{tptr}\ t1 \rightarrow \\ &\quad \text{typeof}\ ei = \text{tint} \rightarrow \\ &\quad \text{PROP}_x\ P\ (\text{LOCAL}_x\ Q\ (\text{SEP}_x\ R)) \\ &\quad \quad \vdash \text{rel\_expr}\ e1\ v1 \\ &\quad \quad \&\&\ \text{rel\_expr}\ ei\ (\text{Vint}\ (\text{Int.repr}\ vi)) \\ &\quad \quad \&\&\ \text{rel\_expr}\ (\text{Ecast}\ e2\ t1)\ v2 \rightarrow \\ &\quad \text{nth\_error}\ R\ n = \text{Some}\ (\text{array\_at}\ t1\ sh\ \text{contents}\ lo\ hi\ v1)) \rightarrow \\ &\quad \text{writable\_share}\ sh \rightarrow \\ &\quad \text{tc\_val}\ t1\ v2 \rightarrow \\ &\quad \text{in\_range}\ lo\ hi\ vi \rightarrow \\ &\quad \text{semax}\ \Delta\ (\triangleright \text{PROP}_x\ P\ (\text{LOCAL}_x\ Q\ (\text{SEP}_x\ R))) \\ &\quad (\text{Sassign}\ (\text{Ederef}\ (\text{Ebinop}\ \text{Oadd}\ e1\ ei\ (\text{tptr}\ t1))\ t1)\ e2) \\ &\quad (\text{normal\_ret\_assert} \\ &\quad \quad (\text{PROP}_x\ P\ (\text{LOCAL}_x\ Q\ (\text{SEP}_x \\ &\quad \quad \quad (\text{replace\_nth}\ n\ R \\ &\quad \quad \quad \quad \text{array\_at}\ t1\ sh\ (\text{upd}\ \text{contents}\ vi\ (\text{valinject}\ \_ \ v2))\ lo\ hi\ v1)))))). \end{aligned}$$

Proof-automation support is available for `semax_loadstore_array` and `rel_expr`, in the form of the `forward_n1` (for “forward nested loads”) tactic. For example, with this proof goal,

`semax Delta`

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

the tactic-application `forward_n1` yields the new proof goal,

`semax Delta`

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

```
`(array_at tdouble Tsh (Vfloat oo fz) 0 n z)))  
MORE_COMMANDS  
POSTCONDITION
```

Users of Microsoft Windows should install VST under Cygwin. Unfortunately, CompCert 2.3 does not (completely) build under Cygwin. That is, CompCert's Coq development builds fine; the subset of CompCert imported into VST (which is mirrored in `vst/compcert`) has no problems. The problem is with the Ocaml build of the extracted code. CompCert uses the Menhir parser generator, which does not build in Cygwin (as of July 2014).

This means that you cannot use CompCert's `clightgen` front-end to translate `.c` files into `.v` files, on cygwin. You will have to use a CompCert installation on a Mac or Linux machine (or virtual machine) to do this.