Verifiable C

Applying the Verified Software Toolchain to C programs

Version 1.5+
September 22, 2015

Andrew W. Appel

with Josiah Dodds and Qinxiang Cao

Contents

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

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

See also the file vst/BUILD_ORGANIZATION.

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*[1] program verification system, floyd.proofauto.

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

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.)

[1]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:

- In the LOCAL component of an assertion,
  temp i v is the recommended way to write `(eq v) (eval_id i), and
  var i t v is the recommended way to write `(eq v) (eval_var i t).
  See Chapter 18 of this manual.
- The type-checker now has a more refined view of char and short types (see Chapter 20 of this manual).
- field_mapsto is now called field_at, and it is dependently typed; see Chapter 32 of this manual.
- typed_mapsto is renamed to data_at, and last two arguments are swapped.
- umapsto ("untyped mapsto") no longer exists.
- mapsto π t v w now permits either (w =Vundef) or the value w belongs to type t. This permits describing uninitialized locations, i.e., mapsto_π t v = mapsto_ π t v Vundef. See Chapter 32 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 52.
- 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 $\rightarrow$ Prop, the underlying semantic model is different; thus we keep the type mpred abstract (opaque). See *Program Logics for Certified Compilers (PLCC)* for more explanation.

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

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

4 Separation Logic

(see PLCC Chapter 12)

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

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

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

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

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

func_ptr ϕ v means that value v is a pointer to a function with specification ϕ.

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

mapsto (π: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 π.

mapsto_ (π: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 π.

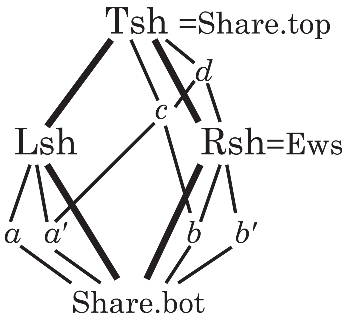
field_at (π: share) (t: type) (f: list ident) (w: reptype (nested_field_type2 f) (v: va
describes a heap that holds just field fld of struct-value v, belonging to struct-type t, containing value w. If type t describes a nested struct type, then f can actually be a path of field selections that descends 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_(π: share) (t: type) (fld: ident) (v: val) : mpred
is the corresponding uninitialized structure-field.

6 Shares

(See PLCC Chapters 11,41)

The mapsto operator (and related operators) take a *permission share*, here written π 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')
$a' \oplus b = c$ lub(c, Rsh) = $a' \oplus$ 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 (mapsto π t v w) holds, then the share π 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 π has enough permission to write (i.e., Rsh $\subset \pi$, we can say writable_share π : Prop.

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

Sequential programs usually have little need of any shares except the Tsh and Ews. However, many function specifications can be parameterized over any share, 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 56.

7 *CompCert C*

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

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

Integers. 32-bit (and 8-bit, 16-bit, 64-bit) signed/unsigned integers.
Floats. IEEE floating point numbers.
Values. The val type: integer + float + pointer + undefined.
AST. Generic support for abstract syntax.
Ctypes. C-language types and structure-field-offset computations.
Cop. Semantics of C-language arithmetic operators.
Clight. Abstract syntax of C-light expressions, statements, and functions.
veric.expr. (from VST, not CompCert) Semantics of expression evaluation.

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

8 Verifiable C programming

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

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

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

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

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

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

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

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

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

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

10 *Values*

Definition block : Type := positive.

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

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

Vint(i) is an integer value, where i is a CompCert 32-bit integer.

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

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 I8 _ _⇒ 1
  | Tint I16 _ _⇒ 2
  | Tint I32 _ _⇒ 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 I8 _ _⇒ 1
  | Tint I16 _ _⇒ 2
  | Tint I32 _ _⇒ 4
  | Tlong _ _⇒ 8
  | Tfloat F32 _⇒ 4
  | Tfloat F64 _⇒ 8
  | Tpointer _ _⇒ 4
  ... et cetera
  end.

Lemma sizeof_pos: ∀ t, 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 ⇒ ty
  | Tint _ _ _⇒ ty
  | Tlong _ _⇒ ty
  | Tfloat _ _⇒ ty
  | Tpointer t1 a ⇒ Tpointer (unroll_composite t1) a
  | Tarray t1 sz a ⇒ Tarray (unroll_composite t1) sz a
  | Tfunction t1 t2 ⇒
        Tfunction (unroll_composite_list t1) (unroll_composite t2)
  | Tstruct id fld a ⇒
        if ident_eq id cid then ty
        else Tstruct id (unroll_composite_fields fld) a
  | Tunion id fld a ⇒
        if ident_eq id cid then ty
        else Tunion id (unroll_composite_fields fld) a
  | Tcomp_ptr id a ⇒
        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:
  ∀cid comp ty, alignof (unroll_composite cid comp ty) = alignof ty.

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

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

13 C operators


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

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

```
  | neg_default ⇒ None
  end.

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

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

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

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

14 C expression evaluation

(vst/veric/expr.v)

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

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

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

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

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

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

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

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

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

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

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

15 C type checking

(See PLCC Chapter 25)

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

If you are asked to prove an entailment where the typechecking condition evaluates to False, this may be because your program is not written in Verifiable C. You may need to perform some local transformations on your C program in order to proceed. We listed these transformations on page 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. The tactic query_context Delta _p. will add hypothesis QUERY : (temp_types Delta) ! _p = Some (tptr t_struct_list, true). This means that in Delta, _p is a temporary variable with type tptr t_struct_list and that it is known to be initialized.

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

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

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

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

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

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

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

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

Consider this assertion:

```
fun ρ ⇒ mapsto 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 ρ, 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 val→val→mpred to type (environ→val)→(environ→val)→(environ→mpred), and the third one lifts from val to environ→val.

17 Canonical forms

(See PLCC section 26)

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

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

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

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

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

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

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

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

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

18 Supercanonical forms

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

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

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

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

19 Go_lower

(See PLCC Chapter 26)

An entailment PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP($\vec{R}$) $\vdash$ PROP($\vec{P}'$)LOCAL($\vec{Q}'$)SEP($\vec{R}'$) is a sequent in our *lifted* separation logic; each side has type environ→mpred. By definition of the lifted entailment $\vdash$ it means exactly, $\forall\rho.$ PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP($\vec{R}$)$\rho \vdash$ PROP($\vec{P}'$)LOCAL($\vec{Q}'$)SEP($\vec{R}'$)ρ. There are two ways to prove such an entailment: Explicitly introduce ρ (descend into an entailment on mpred) and unfold the PROP/LOCAL/SEP form; or stay in canonical form and rewrite in the lifted logic. Either way may be appropriate; this chapter describes how to descend. The go_lower tactic, described on this page, is rarely called directly; it is the first step of the entailer tactic (page 42) when applied to lifted entailments.

The tactic go_lower tactic does the following:

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

Local variables that appear in the lifted canonical form as (eval_id _x) will be replaced by Coq variables x, provided that: (1) $\vec{Q}$ includes a clause of the form (tc_environ Δ), and (2) there is a hypothesis name x _x "above the line." (See PLCC section 26). In addition, a typechecking hypothesis for x will be introduced above the line (see Chapter 20).

20 Welltypedness of variables

The typechecker ensures some invariants about the values of C-program variables: if a variable is initialized, it contains a value of its declared type.

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

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

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

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

$$\mathrm{PROP}(\vec{P})\ \mathrm{LOCAL}(\mathsf{tc\_environ}\ \Delta; \vec{Q})\ \mathrm{SEP}(\vec{R})\ \vdash \mathrm{PROP}(\vec{P}')\ \mathrm{LOCAL}(\vec{Q}')\ \mathrm{SEP}(\vec{R}')$$

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

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

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

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

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

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

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

```
TC1: 0 <= Int.unsigned c <= Byte.max_unsigned
TC2: Byte.min_signed <= Int.signed c <= Byte.max_signed
```

21 Normalize

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

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

Generic separation-logic simplifications.

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

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

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

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

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

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

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

Unlifting.

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

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

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

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

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

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

Pulling nonspatial propositions out of spatial ones.

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

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

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

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

Canonical forms.

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

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

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

Modular Integer arithmetic.

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

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

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

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

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

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

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

Type checking and miscellaneous.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Structure fields.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Function return values.

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

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

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

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

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

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

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

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

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

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

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

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

22 Entailer

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

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

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

23 Cancel

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

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

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

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

If the goal has the form

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

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

The frame may have been created by evar(F: list(environ→mpred)). This is typically done automatically, as part of forward symbolic execution through a function call.

24 The Hoare triple

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

semax (Δ: tycontext) (P: environ→mpred) (c: statement) (R: ret_assert)

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

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

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

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

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

25 *Later*

(See PLCC Chapter 15)

Many of the Hoare rules, such as the one on page 44,

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

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

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

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

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

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

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

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

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

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

26 Specifying a function (See PLCC Chapter 27)

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

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

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

```
Definition F_spec :=
 DECLARE _F
  WITH a1 : τ1, ... ak : τk
  PRE [ x1 OF t1, ... , xn OF tn ]  P
  POST [ tret ]  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.

27 Specifying all functions (PLCC Chapter 27)

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

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

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

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

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

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

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

Lemma body_F: semax_body V Γ f_F F_spec.

28 *Proving a function*

(See PLCC Chapter 27)

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 Δ is calculated from the global type-context overlaid with the parameter- and local-types of the function.

To prove this, we begin with the tactic start_function, which takes care of some simple bookkeeping—unfolding certain definitions, destructing certain tuples, and putting the precondition in canonical form.

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

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

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

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

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 π t) e v, where e is equivalent to (eval_expr E). For example, if E is just an identifer (Etempvar _y t), then e could be either (eval_expr (Etempvar _y t)) or (eval_id _y).

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

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

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

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

$E_1 \rightarrow$*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 `` `(field_at $\pi$ $t$ $fld$) $v$ $e_1$`` or `` `(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 (eval_expr E_1). The share π must be strong enough to grant write permission, that is, writable_share(π).

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

Pointer comparisons are tricky in CompCert C for reasons explained at PLCC page 249; the program logic uses the semax_ptr_compare rule (PLCC page 164). After applying the forward tactic, the user will be left with some proof obligations: Prove that both E_1 and E_2 evaluate to allocated locations (i.e., that the precondition implies $E_1 \overset{\pi_1}{\mapsto} \_ * TT$ and also implies $E_2 \overset{\pi_2}{\mapsto} \_ * TT$, for any π_1 and π_2). If the comparison is any of >, <, ≥, <=, 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 56.

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

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

return E; No special precondition, except that the presence/absence of E must match the nonvoid/void return type of the function. The proof goal left by forward is to show that the precondition (with appropriate substitution for the abstract variable ret_var) entails the function's postcondition.

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

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

29 *Manipulating preconditions*

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

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

then instantiate the existentials; this finally gives us

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

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

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

- Use the rule of consequence explicitly:
  apply semax_pre **with** PROP($\vec{P}'$)LOCAL($\vec{Q}'$)SEP($\vec{R}'$),
  then prove $\vec{P};\vec{Q};\vec{R} \vdash \vec{P}';\vec{Q}';\vec{R}'$ using go_lower (page 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 conjucts from the SEP items, that is, the semicolon-separated list of separating conjuncts. These tactic refer to the list by zero-based position number, 0,1,2,. . . . For example, suppose the goal is a semax or entailment containing PROP($\vec{P}$)LOCAL($\vec{Q}$)SEP(a;b;c;d;e;f;g;h;i;j). Then:

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

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

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

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

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

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

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

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

frame_SEP i j k. Apply the frame rule, keeping only elements i,j,k of the SEP list. See Chapter 30.

30 The Frame rule

Separation Logic supports the Frame rule,

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

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

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

and suppose you want to "frame out" R_2 for the duration of $c_1;c_2$, and have it back again for c_3. First you rewrite by seq_assoc to yield the goal

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

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

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

semax Δ' ?88 c_3 *Post*

Then frame_SEP 0 2 to retain only R_0;R_2.

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

Now you'll see that (in the precondition of the second subgoal) the unification variable ?88 has been instantiated in such a way that R_2 is added back in.

31 Pointer comparisons

See PLCC Chapter 25

Pointer comparisons can be split into two cases:

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

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

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

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

field_at π t_struct_list [head] h p * field_at π t_struct_list [_tail] t p

data_at π t_struct_list (h,t) p

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

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

WHY ARE THE ARGUMENTS BACKWARDS? We write field_at π t_struct_list [head] h p where Reynolds would have written p.head$\mapsto h$, and we write data_at π 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 π t_struct_list (h,t)) (eval_id _p)tptr t_struct_list))

which simplify to

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

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

PARTIALLY INITIALIZED DATA STRUCTURES. Consider the program

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

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

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

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

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

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

At point 3, we can write either of,

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

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

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

If p→head and p→tail may be uninitialized, this is fine. But if the structure is known to be initialized, the precondition as written

does not express this fact. One would need to add the conjunct
!!(is_int (fst data) ∧ is_pointer_or_null (snd data))
at the point marked *(**)*.

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

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

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

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

Using reptype' one could write,

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

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

33 *Nested structs*

Consider a nested struct; shown here is exactly the example in progs/nest2.c, so you can examine the proofs in progs/verif_nest2.c.

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

struct b pb; struct a pa; int i;
```

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

```
LOCAL(`(eq pb) (eval_var _pb))
SEP( `(data_at π t_struct_b (y1,(x1,x2)) pb); Frame)
```

where Frame has some unrelated spatial conjuncts. The postcondition (after the load) would include the new LOCALfact, `(eq x2) (eval_id i).

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

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

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

```
SEP(`(field_at Ews t_struct_b [_y1] (Vint y1) pb);
    `(field_at Ews t_struct_b [_x1;_y2] (Vfloat x1) pb);
    `(field_at Ews t_struct_b [_x2;_y2] (Vint x2) pb);
    Frame)
```

The third argument of field_at represents the *path* of structure-fields that leads to a given substructure. The empty path (nil) works too; it “leads” to the entire structure.

34 Signed and unsigned integers

Mathematical proofs use the mathematical integers (the Z type in Coq); C progams use 32-bit signed or unsigned integers. They are related as follows:

$\mathsf{Int.repr}$: $\mathsf{Z} \rightarrow \mathsf{int}$.
$\mathsf{Int.unsigned}$: $\mathsf{int} \rightarrow \mathsf{Z}$.
$\mathsf{Int.signed}$: $\mathsf{int} \rightarrow \mathsf{Z}$.

with the following lemmas:

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

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

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

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

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

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

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

If you maintain this hypothesis "above the line", then the normalize tactic can automatically rewrite with $\mathsf{Int.signed}\ (\mathsf{Int.repr}\ x) = x$. Also, to solve goals such as,

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

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

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

To see an example in action, look at progs/verif_sumarray.v. The array size and index (variables size and i) are kept within bounds; but the *contents* of the array might overflow when added up, which is why add_elem uses Int.add instead of Z.add.

35 *For loops*

The C-language for loop has the general form,

for $(init;\ test;\ incr)\ body$

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

forward_for *Inv* *PreIncr* *PostCond*

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

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

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

The following feature will appear in VST version 1.5.

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

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

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

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

36 Nested Loads

This experimental feature will appear in VST release 1.5.

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

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

rel_expr : expr $\rightarrow$ val $\rightarrow$ rho $\rightarrow$ mpred.
rel_lvalue: expr $\rightarrow$ val $\rightarrow$ rho $\rightarrow$ mpred.

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

Evaluation rules for rel_expr are listed here:

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

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

The primitive nested-load assignment rule is,

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

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

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

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

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

the tactic-application forward_nl yields the new proof goal,

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