Contents

Chapter 1

Library VC.sumarray

```
From Coq Require Import String List ZArith.
From compcert Require Import Coqlib Integers Floats AST Ctypes Cop Clight Clightdefs.
Local Open Scope Z_scope.
Local Open Scope string_scope.

Module Info.
  Definition version := "3.7".
  Definition build_number := "".
  Definition build_tag := "".
  Definition arch := "x86".
  Definition model := "32sse2".
  Definition abi := "standard".
  Definition bitsize := 32.
  Definition big_endian := false.
  Definition source_file := "sumarray.c".
  Definition normalized := true.
End Info.

Definition ___builtin_ais_annot : ident := 1%positive.
Definition ___builtin_annot : ident := 10%positive.
Definition ___builtin_annot_intval : ident := 11%positive.
Definition ___builtin_bswap : ident := 3%positive.
Definition ___builtin_bswap16 : ident := 5%positive.
Definition ___builtin_bswap32 : ident := 4%positive.
Definition ___builtin_bswap64 : ident := 2%positive.
Definition ___builtin_clz : ident := 36%positive.
Definition ___builtin_clzl : ident := 37%positive.
Definition ___builtin_clzll : ident := 38%positive.
Definition ___builtin_ctz : ident := 39%positive.
Definition ___builtin_ctzl : ident := 40%positive.
Definition ___builtin_ctzll : ident := 41%positive.
```

```
Definition ___builtin_debug : ident := 52%positive.
Definition ___builtin_fabs : ident := 6%positive.
Definition ___builtin_fmadd : ident := 44%positive.
Definition ___builtin_fmax : ident := 42%positive.
Definition ___builtin_fmin : ident := 43%positive.
Definition ___builtin_fmsub : ident := 45%positive.
Definition ___builtin_fnmadd : ident := 46%positive.
Definition ___builtin_fnmsub : ident := 47%positive.
Definition ___builtin_fsqrt : ident := 7%positive.
Definition ___builtin_membar : ident := 12%positive.
Definition ___builtin_memcpy_aligned : ident := 8%positive.
Definition ___builtin_read16_reversed : ident := 48%positive.
Definition ___builtin_read32_reversed : ident := 49%positive.
Definition ___builtin_sel : ident := 9%positive.
Definition ___builtin_va_arg : ident := 14%positive.
Definition ___builtin_va_copy : ident := 15%positive.
Definition ___builtin_va_end : ident := 16%positive.
Definition ___builtin_va_start : ident := 13%positive.
Definition ___builtin_write16_reversed : ident := 50%positive.
Definition ___builtin_write32_reversed : ident := 51%positive.
Definition ___compcert_i64_dtos : ident := 21%positive.
Definition ___compcert_i64_dtou : ident := 22%positive.
Definition ___compcert_i64_sar : ident := 33%positive.
Definition ___compcert_i64_sdiv : ident := 27%positive.
Definition ___compcert_i64_shl : ident := 31%positive.
Definition ___compcert_i64_shr : ident := 32%positive.
Definition ___compcert_i64_smod : ident := 29%positive.
Definition ___compcert_i64_smulh : ident := 34%positive.
Definition ___compcert_i64_stod : ident := 23%positive.
Definition ___compcert_i64_stof : ident := 25%positive.
Definition ___compcert_i64_udiv : ident := 28%positive.
Definition ___compcert_i64_umod : ident := 30%positive.
Definition ___compcert_i64_umulh : ident := 35%positive.
Definition ___compcert_i64_utod : ident := 24%positive.
Definition ___compcert_i64_utof : ident := 26%positive.
Definition ___compcert_va_composite : ident := 20%positive.
Definition ___compcert_va_float64 : ident := 19%positive.
Definition ___compcert_va_int32 : ident := 17%positive.
Definition ___compcert_va_int64 : ident := 18%positive.
Definition _a : ident := 53%positive.
Definition _four : ident := 58%positive.
Definition _i : ident := 55%positive.
```

```
Definition _main : ident := 59%positive.
Definition _n : ident := 54%positive.
Definition _s : ident := 56%positive.
Definition _sumarray : ident := 57%positive.
Definition _t'1 : ident := 60%positive.

Definition f_sumarray := {|
  fn_return := tuint;
  fn_callconv := cc_default;
  fn_params := ((_a, (tptr tuint)) :: (_n, tint) :: nil);
  fn_vars := nil;
  fn_temps := ((_i, tint) :: (_s, tuint) :: (_t'1, tuint) :: nil);
  fn_body :=
(Ssequence
  (Sset _i (Econst_int (Int.repr 0) tint))
  (Ssequence
    (Sset _s (Econst_int (Int.repr 0) tint))
    (Ssequence
      (Swhile
        (Ebinop Olt (Etempvar _i tint) (Etempvar _n tint) tint)
        (Ssequence
          (Ssequence
            (Sset _t'1
              (Ederef
                (Ebinop Oadd (Etempvar _a (tptr tuint)) (Etempvar _i tint)
                  (tptr tuint)) tuint))
            (Sset _s
              (Ebinop Oadd (Etempvar _s tuint) (Etempvar _t'1 tuint) tuint)))
          (Sset _i
            (Ebinop Oadd (Etempvar _i tint) (Econst_int (Int.repr 1) tint)
              tint))))
      (Sreturn (Some (Etempvar _s tuint))))))
|}.

Definition v_four := {|
  gvar_info := (tarray tuint 4);
  gvar_init := (Init_int32 (Int.repr 1) :: Init_int32 (Int.repr 2) ::
                Init_int32 (Int.repr 3) :: Init_int32 (Int.repr 4) :: nil);
  gvar_readonly := false;
  gvar_volatile := false
|}.

Definition f_main := {|
  fn_return := tint;
  fn_callconv := cc_default;
```

```
  fn_params := nil;
  fn_vars := nil;
  fn_temps := ((_s, tuint) :: (_t'1, tuint) :: nil);
  fn_body :=
(Ssequence
  (Ssequence
    (Ssequence
      (Scall (Some _t'1)
        (Evar _sumarray (Tfunction (Tcons (tptr tuint) (Tcons tint Tnil))
                          tuint cc_default))
        ((Evar _four (tarray tuint 4)) :: (Econst_int (Int.repr 4) tint) ::
         nil))
      (Sset _s (Etempvar _t'1 tuint)))
    (Sreturn (Some (Ecast (Etempvar _s tuint) tint))))
  (Sreturn (Some (Econst_int (Int.repr 0) tint))))
|}.

Definition composites : list composite_definition :=
nil.

Definition global_definitions : list (ident × globdef fundef type) :=
((___builtin_ais_annot,
   Gfun(External (EF_builtin "__builtin_ais_annot"
                   (mksignature (AST.Tint :: nil) AST.Tvoid
                     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
     (Tcons (tptr tschar) Tnil) tvoid
     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
 (___builtin_bswap64,
   Gfun(External (EF_builtin "__builtin_bswap64"
                   (mksignature (AST.Tlong :: nil) AST.Tlong cc_default))
     (Tcons tulong Tnil) tulong cc_default)) ::
 (___builtin_bswap,
   Gfun(External (EF_builtin "__builtin_bswap"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
     (Tcons tuint Tnil) tuint cc_default)) ::
 (___builtin_bswap32,
   Gfun(External (EF_builtin "__builtin_bswap32"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
     (Tcons tuint Tnil) tuint cc_default)) ::
 (___builtin_bswap16,
   Gfun(External (EF_builtin "__builtin_bswap16"
                   (mksignature (AST.Tint :: nil) AST.Tint16unsigned
                     cc_default)) (Tcons tushort Tnil) tushort cc_default)) ::
 (___builtin_fabs,
```

```
  Gfun(External (EF_builtin "__builtin_fabs"
                   (mksignature (AST.Tfloat :: nil) AST.Tfloat cc_default))
    (Tcons tdouble Tnil) tdouble cc_default)) ::
(___builtin_fsqrt,
  Gfun(External (EF_builtin "__builtin_fsqrt"
                   (mksignature (AST.Tfloat :: nil) AST.Tfloat cc_default))
    (Tcons tdouble Tnil) tdouble cc_default)) ::
(___builtin_memcpy_aligned,
  Gfun(External (EF_builtin "__builtin_memcpy_aligned"
                   (mksignature
                     (AST.Tint :: AST.Tint :: AST.Tint :: AST.Tint :: nil)
                     AST.Tvoid cc_default))
    (Tcons (tptr tvoid)
      (Tcons (tptr tvoid) (Tcons tuint (Tcons tuint Tnil)))) tvoid
    cc_default)) ::
(___builtin_sel,
  Gfun(External (EF_builtin "__builtin_sel"
                   (mksignature (AST.Tint :: nil) AST.Tvoid
                     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
    (Tcons tbool Tnil) tvoid
    {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
(___builtin_annot,
  Gfun(External (EF_builtin "__builtin_annot"
                   (mksignature (AST.Tint :: nil) AST.Tvoid
                     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
    (Tcons (tptr tschar) Tnil) tvoid
    {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
(___builtin_annot_intval,
  Gfun(External (EF_builtin "__builtin_annot_intval"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tint
                     cc_default)) (Tcons (tptr tschar) (Tcons tint Tnil))
    tint cc_default)) ::
(___builtin_membar,
  Gfun(External (EF_builtin "__builtin_membar"
                   (mksignature nil AST.Tvoid cc_default)) Tnil tvoid
    cc_default)) ::
(___builtin_va_start,
  Gfun(External (EF_builtin "__builtin_va_start"
                   (mksignature (AST.Tint :: nil) AST.Tvoid cc_default))
    (Tcons (tptr tvoid) Tnil) tvoid cc_default)) ::
(___builtin_va_arg,
  Gfun(External (EF_builtin "__builtin_va_arg"
```

```
                       (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                         cc_default)) (Tcons (tptr tvoid) (Tcons tuint Tnil))
     tvoid cc_default)) ::
(___builtin_va_copy,
   Gfun(External (EF_builtin "__builtin_va_copy"
                       (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                         cc_default))
     (Tcons (tptr tvoid) (Tcons (tptr tvoid) Tnil)) tvoid cc_default)) ::
(___builtin_va_end,
   Gfun(External (EF_builtin "__builtin_va_end"
                       (mksignature (AST.Tint :: nil) AST.Tvoid cc_default))
     (Tcons (tptr tvoid) Tnil) tvoid cc_default)) ::
(___compcert_va_int32,
   Gfun(External (EF_external "__compcert_va_int32"
                       (mksignature (AST.Tint :: nil) AST.Tint cc_default))
     (Tcons (tptr tvoid) Tnil) tuint cc_default)) ::
(___compcert_va_int64,
   Gfun(External (EF_external "__compcert_va_int64"
                       (mksignature (AST.Tint :: nil) AST.Tlong cc_default))
     (Tcons (tptr tvoid) Tnil) tulong cc_default)) ::
(___compcert_va_float64,
   Gfun(External (EF_external "__compcert_va_float64"
                       (mksignature (AST.Tint :: nil) AST.Tfloat cc_default))
     (Tcons (tptr tvoid) Tnil) tdouble cc_default)) ::
(___compcert_va_composite,
   Gfun(External (EF_external "__compcert_va_composite"
                       (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tint
                         cc_default)) (Tcons (tptr tvoid) (Tcons tuint Tnil))
     (tptr tvoid) cc_default)) ::
(___compcert_i64_dtos,
   Gfun(External (EF_runtime "__compcert_i64_dtos"
                       (mksignature (AST.Tfloat :: nil) AST.Tlong cc_default))
     (Tcons tdouble Tnil) tlong cc_default)) ::
(___compcert_i64_dtou,
   Gfun(External (EF_runtime "__compcert_i64_dtou"
                       (mksignature (AST.Tfloat :: nil) AST.Tlong cc_default))
     (Tcons tdouble Tnil) tulong cc_default)) ::
(___compcert_i64_stod,
   Gfun(External (EF_runtime "__compcert_i64_stod"
                       (mksignature (AST.Tlong :: nil) AST.Tfloat cc_default))
     (Tcons tlong Tnil) tdouble cc_default)) ::
(___compcert_i64_utod,
```

```
  Gfun(External (EF_runtime "__compcert_i64_utod"
                   (mksignature (AST.Tlong :: nil) AST.Tfloat cc_default))
    (Tcons tulong Tnil) tdouble cc_default)) ::
(___compcert_i64_stof,
  Gfun(External (EF_runtime "__compcert_i64_stof"
                   (mksignature (AST.Tlong :: nil) AST.Tsingle cc_default))
    (Tcons tlong Tnil) tfloat cc_default)) ::
(___compcert_i64_utof,
  Gfun(External (EF_runtime "__compcert_i64_utof"
                   (mksignature (AST.Tlong :: nil) AST.Tsingle cc_default))
    (Tcons tulong Tnil) tfloat cc_default)) ::
(___compcert_i64_sdiv,
  Gfun(External (EF_runtime "__compcert_i64_sdiv"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tlong Tnil)) tlong
    cc_default)) ::
(___compcert_i64_udiv,
  Gfun(External (EF_runtime "__compcert_i64_udiv"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tulong (Tcons tulong Tnil)) tulong
    cc_default)) ::
(___compcert_i64_smod,
  Gfun(External (EF_runtime "__compcert_i64_smod"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tlong Tnil)) tlong
    cc_default)) ::
(___compcert_i64_umod,
  Gfun(External (EF_runtime "__compcert_i64_umod"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tulong (Tcons tulong Tnil)) tulong
    cc_default)) ::
(___compcert_i64_shl,
  Gfun(External (EF_runtime "__compcert_i64_shl"
                   (mksignature (AST.Tlong :: AST.Tint :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tint Tnil)) tlong
    cc_default)) ::
(___compcert_i64_shr,
  Gfun(External (EF_runtime "__compcert_i64_shr"
                   (mksignature (AST.Tlong :: AST.Tint :: nil) AST.Tlong
                     cc_default)) (Tcons tulong (Tcons tint Tnil)) tulong
    cc_default)) ::
(___compcert_i64_sar,
```

```
  Gfun(External (EF_runtime "__compcert_i64_sar"
                   (mksignature (AST.Tlong :: AST.Tint :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tint Tnil)) tlong
    cc_default)) ::
(___compcert_i64_smulh,
  Gfun(External (EF_runtime "__compcert_i64_smulh"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tlong Tnil)) tlong
    cc_default)) ::
(___compcert_i64_umulh,
  Gfun(External (EF_runtime "__compcert_i64_umulh"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tulong (Tcons tulong Tnil)) tulong
    cc_default)) ::
(___builtin_clz,
  Gfun(External (EF_builtin "__builtin_clz"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_clzl,
  Gfun(External (EF_builtin "__builtin_clzl"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_clzll,
  Gfun(External (EF_builtin "__builtin_clzll"
                   (mksignature (AST.Tlong :: nil) AST.Tint cc_default))
    (Tcons tulong Tnil) tint cc_default)) ::
(___builtin_ctz,
  Gfun(External (EF_builtin "__builtin_ctz"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_ctzl,
  Gfun(External (EF_builtin "__builtin_ctzl"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_ctzll,
  Gfun(External (EF_builtin "__builtin_ctzll"
                   (mksignature (AST.Tlong :: nil) AST.Tint cc_default))
    (Tcons tulong Tnil) tint cc_default)) ::
(___builtin_fmax,
  Gfun(External (EF_builtin "__builtin_fmax"
                   (mksignature (AST.Tfloat :: AST.Tfloat :: nil) AST.Tfloat
                     cc_default)) (Tcons tdouble (Tcons tdouble Tnil))
```

```
    tdouble cc_default)) ::
(___builtin_fmin,
  Gfun(External (EF_builtin "__builtin_fmin"
                    (mksignature (AST.Tfloat :: AST.Tfloat :: nil) AST.Tfloat
                      cc_default)) (Tcons tdouble (Tcons tdouble Tnil))
    tdouble cc_default)) ::
(___builtin_fmadd,
  Gfun(External (EF_builtin "__builtin_fmadd"
                    (mksignature
                      (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                      AST.Tfloat cc_default))
    (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
    cc_default)) ::
(___builtin_fmsub,
  Gfun(External (EF_builtin "__builtin_fmsub"
                    (mksignature
                      (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                      AST.Tfloat cc_default))
    (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
    cc_default)) ::
(___builtin_fnmadd,
  Gfun(External (EF_builtin "__builtin_fnmadd"
                    (mksignature
                      (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                      AST.Tfloat cc_default))
    (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
    cc_default)) ::
(___builtin_fnmsub,
  Gfun(External (EF_builtin "__builtin_fnmsub"
                    (mksignature
                      (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                      AST.Tfloat cc_default))
    (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
    cc_default)) ::
(___builtin_read16_reversed,
  Gfun(External (EF_builtin "__builtin_read16_reversed"
                    (mksignature (AST.Tint :: nil) AST.Tint16unsigned
                      cc_default)) (Tcons (tptr tushort) Tnil) tushort
    cc_default)) ::
(___builtin_read32_reversed,
  Gfun(External (EF_builtin "__builtin_read32_reversed"
                    (mksignature (AST.Tint :: nil) AST.Tint cc_default))
```

```
     (Tcons (tptr tuint) Tnil) tuint cc_default)) ::
 (___builtin_write16_reversed,
   Gfun(External (EF_builtin "__builtin_write16_reversed"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                     cc_default)) (Tcons (tptr tushort) (Tcons tushort Tnil))
     tvoid cc_default)) ::
 (___builtin_write32_reversed,
   Gfun(External (EF_builtin "__builtin_write32_reversed"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                     cc_default)) (Tcons (tptr tuint) (Tcons tuint Tnil))
     tvoid cc_default)) ::
 (___builtin_debug,
   Gfun(External (EF_external "__builtin_debug"
                   (mksignature (AST.Tint :: nil) AST.Tvoid
                     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
     (Tcons tint Tnil) tvoid
     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
 (_sumarray, Gfun(Internal f_sumarray)) :: (_four, Gvar v_four) ::
 (_main, Gfun(Internal f_main)) :: nil).

Definition public_idents : list ident :=
(_main :: _four :: _sumarray :: ___builtin_debug ::
 ___builtin_write32_reversed :: ___builtin_write16_reversed ::
 ___builtin_read32_reversed :: ___builtin_read16_reversed ::
 ___builtin_fnmsub :: ___builtin_fnmadd :: ___builtin_fmsub ::
 ___builtin_fmadd :: ___builtin_fmin :: ___builtin_fmax ::
 ___builtin_ctzll :: ___builtin_ctzl :: ___builtin_ctz :: ___builtin_clzll ::
 ___builtin_clzl :: ___builtin_clz :: ___compcert_i64_umulh ::
 ___compcert_i64_smulh :: ___compcert_i64_sar :: ___compcert_i64_shr ::
 ___compcert_i64_shl :: ___compcert_i64_umod :: ___compcert_i64_smod ::
 ___compcert_i64_udiv :: ___compcert_i64_sdiv :: ___compcert_i64_utof ::
 ___compcert_i64_stof :: ___compcert_i64_utod :: ___compcert_i64_stod ::
 ___compcert_i64_dtou :: ___compcert_i64_dtos :: ___compcert_va_composite ::
 ___compcert_va_float64 :: ___compcert_va_int64 :: ___compcert_va_int32 ::
 ___builtin_va_end :: ___builtin_va_copy :: ___builtin_va_arg ::
 ___builtin_va_start :: ___builtin_membar :: ___builtin_annot_intval ::
 ___builtin_annot :: ___builtin_sel :: ___builtin_memcpy_aligned ::
 ___builtin_fsqrt :: ___builtin_fabs :: ___builtin_bswap16 ::
 ___builtin_bswap32 :: ___builtin_bswap :: ___builtin_bswap64 ::
 ___builtin_ais_annot :: nil).

Definition prog : Clight.program :=
  mkprogram composites global_definitions public_idents _main Logic.I.
```

Chapter 2

Library VC.reverse

```
From Coq Require Import String List ZArith.
From compcert Require Import Coqlib Integers Floats AST Ctypes Cop Clight Clightdefs.
Local Open Scope Z_scope.
Local Open Scope string_scope.

Module Info.
  Definition version := "3.7".
  Definition build_number := "".
  Definition build_tag := "".
  Definition arch := "x86".
  Definition model := "32sse2".
  Definition abi := "standard".
  Definition bitsize := 32.
  Definition big_endian := false.
  Definition source_file := "reverse.c".
  Definition normalized := true.
End Info.

Definition ___builtin_ais_annot : ident := 4%positive.
Definition ___builtin_annot : ident := 13%positive.
Definition ___builtin_annot_intval : ident := 14%positive.
Definition ___builtin_bswap : ident := 6%positive.
Definition ___builtin_bswap16 : ident := 8%positive.
Definition ___builtin_bswap32 : ident := 7%positive.
Definition ___builtin_bswap64 : ident := 5%positive.
Definition ___builtin_clz : ident := 39%positive.
Definition ___builtin_clzl : ident := 40%positive.
Definition ___builtin_clzll : ident := 41%positive.
Definition ___builtin_ctz : ident := 42%positive.
Definition ___builtin_ctzl : ident := 43%positive.
Definition ___builtin_ctzll : ident := 44%positive.
```

```
Definition ___builtin_debug : ident := 55%positive.
Definition ___builtin_fabs : ident := 9%positive.
Definition ___builtin_fmadd : ident := 47%positive.
Definition ___builtin_fmax : ident := 45%positive.
Definition ___builtin_fmin : ident := 46%positive.
Definition ___builtin_fmsub : ident := 48%positive.
Definition ___builtin_fnmadd : ident := 49%positive.
Definition ___builtin_fnmsub : ident := 50%positive.
Definition ___builtin_fsqrt : ident := 10%positive.
Definition ___builtin_membar : ident := 15%positive.
Definition ___builtin_memcpy_aligned : ident := 11%positive.
Definition ___builtin_read16_reversed : ident := 51%positive.
Definition ___builtin_read32_reversed : ident := 52%positive.
Definition ___builtin_sel : ident := 12%positive.
Definition ___builtin_va_arg : ident := 17%positive.
Definition ___builtin_va_copy : ident := 18%positive.
Definition ___builtin_va_end : ident := 19%positive.
Definition ___builtin_va_start : ident := 16%positive.
Definition ___builtin_write16_reversed : ident := 53%positive.
Definition ___builtin_write32_reversed : ident := 54%positive.
Definition ___compcert_i64_dtos : ident := 24%positive.
Definition ___compcert_i64_dtou : ident := 25%positive.
Definition ___compcert_i64_sar : ident := 36%positive.
Definition ___compcert_i64_sdiv : ident := 30%positive.
Definition ___compcert_i64_shl : ident := 34%positive.
Definition ___compcert_i64_shr : ident := 35%positive.
Definition ___compcert_i64_smod : ident := 32%positive.
Definition ___compcert_i64_smulh : ident := 37%positive.
Definition ___compcert_i64_stod : ident := 26%positive.
Definition ___compcert_i64_stof : ident := 28%positive.
Definition ___compcert_i64_udiv : ident := 31%positive.
Definition ___compcert_i64_umod : ident := 33%positive.
Definition ___compcert_i64_umulh : ident := 38%positive.
Definition ___compcert_i64_utod : ident := 27%positive.
Definition ___compcert_i64_utof : ident := 29%positive.
Definition ___compcert_va_composite : ident := 23%positive.
Definition ___compcert_va_float64 : ident := 22%positive.
Definition ___compcert_va_int32 : ident := 20%positive.
Definition ___compcert_va_int64 : ident := 21%positive.
Definition _h : ident := 60%positive.
Definition _head : ident := 1%positive.
Definition _list : ident := 2%positive.
```

```
Definition _main : ident := 66%positive.
Definition _p : ident := 57%positive.
Definition _r : ident := 65%positive.
Definition _reverse : ident := 64%positive.
Definition _s : ident := 58%positive.
Definition _sumlist : ident := 61%positive.
Definition _t : ident := 59%positive.
Definition _tail : ident := 3%positive.
Definition _three : ident := 56%positive.
Definition _v : ident := 63%positive.
Definition _w : ident := 62%positive.
Definition _t'1 : ident := 67%positive.
Definition _t'2 : ident := 68%positive.

Definition v_three := {|
  gvar_info := (tarray (Tstruct _list noattr) 3);
  gvar_init := (Init_int32 (Int.repr 1) ::
                Init_addrof _three (Ptrofs.repr 8) ::
                Init_int32 (Int.repr 2) ::
                Init_addrof _three (Ptrofs.repr 16) ::
                Init_int32 (Int.repr 3) :: Init_int32 (Int.repr 0) :: nil);
  gvar_readonly := false;
  gvar_volatile := false
|}.

Definition f_sumlist := {|
  fn_return := tuint;
  fn_callconv := cc_default;
  fn_params := ((_p, (tptr (Tstruct _list noattr))) :: nil);
  fn_vars := nil;
  fn_temps := ((_s, tuint) :: (_t, (tptr (Tstruct _list noattr))) ::
               (_h, tuint) :: nil);
  fn_body :=
(Ssequence
  (Sset _s (Econst_int (Int.repr 0) tint))
  (Ssequence
    (Sset _t (Etempvar _p (tptr (Tstruct _list noattr))))
    (Ssequence
      (Swhile
        (Etempvar _t (tptr (Tstruct _list noattr)))
        (Ssequence
          (Sset _h
            (Efield
              (Ederef (Etempvar _t (tptr (Tstruct _list noattr)))
```

```
                (Tstruct _list noattr)) _head tuint))
          (Ssequence
            (Sset _t
              (Efield
                (Ederef (Etempvar _t (tptr (Tstruct _list noattr)))
                  (Tstruct _list noattr)) _tail
                (tptr (Tstruct _list noattr))))
            (Sset _s
              (Ebinop Oadd (Etempvar _s tuint) (Etempvar _h tuint) tuint)))))
      (Sreturn (Some (Etempvar _s tuint))))))
|}.

Definition f_reverse := {|
  fn_return := (tptr (Tstruct _list noattr));
  fn_callconv := cc_default;
  fn_params := ((_p, (tptr (Tstruct _list noattr))) :: nil);
  fn_vars := nil;
  fn_temps := ((_w, (tptr (Tstruct _list noattr))) ::
               (_t, (tptr (Tstruct _list noattr))) ::
               (_v, (tptr (Tstruct _list noattr))) :: nil);
  fn_body :=
(Ssequence
  (Sset _w (Ecast (Econst_int (Int.repr 0) tint) (tptr tvoid)))
  (Ssequence
    (Sset _v (Etempvar _p (tptr (Tstruct _list noattr))))
    (Ssequence
      (Swhile
        (Etempvar _v (tptr (Tstruct _list noattr)))
        (Ssequence
          (Sset _t
            (Efield
              (Ederef (Etempvar _v (tptr (Tstruct _list noattr)))
                (Tstruct _list noattr)) _tail (tptr (Tstruct _list noattr))))
          (Ssequence
            (Sassign
              (Efield
                (Ederef (Etempvar _v (tptr (Tstruct _list noattr)))
                  (Tstruct _list noattr)) _tail
                (tptr (Tstruct _list noattr)))
              (Etempvar _w (tptr (Tstruct _list noattr))))
            (Ssequence
              (Sset _w (Etempvar _v (tptr (Tstruct _list noattr))))
              (Sset _v (Etempvar _t (tptr (Tstruct _list noattr))))))))
```

```
      (Sreturn (Some (Etempvar _w (tptr (Tstruct _list noattr))))))))
|}.
Definition f_main := {|
  fn_return := tint;
  fn_callconv := cc_default;
  fn_params := nil;
  fn_vars := nil;
  fn_temps := ((_r, (tptr (Tstruct _list noattr))) :: (_s, tuint) ::
               (_t'2, tuint) :: (_t'1, (tptr (Tstruct _list noattr))) :: nil);
  fn_body :=
(Ssequence
  (Ssequence
    (Ssequence
      (Scall (Some _t'1)
        (Evar _reverse (Tfunction (Tcons (tptr (Tstruct _list noattr)) Tnil)
                         (tptr (Tstruct _list noattr)) cc_default))
        ((Evar _three (tarray (Tstruct _list noattr) 3)) :: nil))
      (Sset _r (Etempvar _t'1 (tptr (Tstruct _list noattr)))))
    (Ssequence
      (Ssequence
        (Scall (Some _t'2)
          (Evar _sumlist (Tfunction
                           (Tcons (tptr (Tstruct _list noattr)) Tnil) tuint
                           cc_default))
          ((Etempvar _r (tptr (Tstruct _list noattr))) :: nil))
        (Sset _s (Etempvar _t'2 tuint)))
      (Sreturn (Some (Ecast (Etempvar _s tuint) tint)))))
  (Sreturn (Some (Econst_int (Int.repr 0) tint))))
|}.
Definition composites : list composite_definition :=
(Composite _list Struct
   ((_head, tuint) :: (_tail, (tptr (Tstruct _list noattr))) :: nil)
   noattr :: nil).
Definition global_definitions : list (ident × globdef fundef type) :=
((___builtin_ais_annot,
   Gfun(External (EF_builtin "__builtin_ais_annot"
                   (mksignature (AST.Tint :: nil) AST.Tvoid
                     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
     (Tcons (tptr tschar) Tnil) tvoid
     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
 (___builtin_bswap64,
   Gfun(External (EF_builtin "__builtin_bswap64"
```

```
                       (mksignature (AST.Tlong :: nil) AST.Tlong cc_default))
     (Tcons tulong Tnil) tulong cc_default)) ::
(___builtin_bswap,
   Gfun(External (EF_builtin "__builtin_bswap"
                       (mksignature (AST.Tint :: nil) AST.Tint cc_default))
     (Tcons tuint Tnil) tuint cc_default)) ::
(___builtin_bswap32,
   Gfun(External (EF_builtin "__builtin_bswap32"
                       (mksignature (AST.Tint :: nil) AST.Tint cc_default))
     (Tcons tuint Tnil) tuint cc_default)) ::
(___builtin_bswap16,
   Gfun(External (EF_builtin "__builtin_bswap16"
                       (mksignature (AST.Tint :: nil) AST.Tint16unsigned
                         cc_default)) (Tcons tushort Tnil) tushort cc_default)) ::
(___builtin_fabs,
   Gfun(External (EF_builtin "__builtin_fabs"
                       (mksignature (AST.Tfloat :: nil) AST.Tfloat cc_default))
     (Tcons tdouble Tnil) tdouble cc_default)) ::
(___builtin_fsqrt,
   Gfun(External (EF_builtin "__builtin_fsqrt"
                       (mksignature (AST.Tfloat :: nil) AST.Tfloat cc_default))
     (Tcons tdouble Tnil) tdouble cc_default)) ::
(___builtin_memcpy_aligned,
   Gfun(External (EF_builtin "__builtin_memcpy_aligned"
                       (mksignature
                         (AST.Tint :: AST.Tint :: AST.Tint :: AST.Tint :: nil)
                         AST.Tvoid cc_default))
     (Tcons (tptr tvoid)
       (Tcons (tptr tvoid) (Tcons tuint (Tcons tuint Tnil)))) tvoid
     cc_default)) ::
(___builtin_sel,
   Gfun(External (EF_builtin "__builtin_sel"
                       (mksignature (AST.Tint :: nil) AST.Tvoid
                         {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
     (Tcons tbool Tnil) tvoid
     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
(___builtin_annot,
   Gfun(External (EF_builtin "__builtin_annot"
                       (mksignature (AST.Tint :: nil) AST.Tvoid
                         {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
     (Tcons (tptr tschar) Tnil) tvoid
     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
```

```
(___builtin_annot_intval,
  Gfun(External (EF_builtin "__builtin_annot_intval"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tint
                     cc_default)) (Tcons (tptr tschar) (Tcons tint Tnil))
    tint cc_default)) ::
(___builtin_membar,
  Gfun(External (EF_builtin "__builtin_membar"
                   (mksignature nil AST.Tvoid cc_default)) Tnil tvoid
    cc_default)) ::
(___builtin_va_start,
  Gfun(External (EF_builtin "__builtin_va_start"
                   (mksignature (AST.Tint :: nil) AST.Tvoid cc_default))
    (Tcons (tptr tvoid) Tnil) tvoid cc_default)) ::
(___builtin_va_arg,
  Gfun(External (EF_builtin "__builtin_va_arg"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                     cc_default)) (Tcons (tptr tvoid) (Tcons tuint Tnil))
    tvoid cc_default)) ::
(___builtin_va_copy,
  Gfun(External (EF_builtin "__builtin_va_copy"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                     cc_default))
    (Tcons (tptr tvoid) (Tcons (tptr tvoid) Tnil)) tvoid cc_default)) ::
(___builtin_va_end,
  Gfun(External (EF_builtin "__builtin_va_end"
                   (mksignature (AST.Tint :: nil) AST.Tvoid cc_default))
    (Tcons (tptr tvoid) Tnil) tvoid cc_default)) ::
(___compcert_va_int32,
  Gfun(External (EF_external "__compcert_va_int32"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons (tptr tvoid) Tnil) tuint cc_default)) ::
(___compcert_va_int64,
  Gfun(External (EF_external "__compcert_va_int64"
                   (mksignature (AST.Tint :: nil) AST.Tlong cc_default))
    (Tcons (tptr tvoid) Tnil) tulong cc_default)) ::
(___compcert_va_float64,
  Gfun(External (EF_external "__compcert_va_float64"
                   (mksignature (AST.Tint :: nil) AST.Tfloat cc_default))
    (Tcons (tptr tvoid) Tnil) tdouble cc_default)) ::
(___compcert_va_composite,
  Gfun(External (EF_external "__compcert_va_composite"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tint
```

```
                    cc_default)) (Tcons (tptr tvoid) (Tcons tuint Tnil))
    (tptr tvoid) cc_default)) ::
(___compcert_i64_dtos,
  Gfun(External (EF_runtime "__compcert_i64_dtos"
                   (mksignature (AST.Tfloat :: nil) AST.Tlong cc_default))
    (Tcons tdouble Tnil) tlong cc_default)) ::
(___compcert_i64_dtou,
  Gfun(External (EF_runtime "__compcert_i64_dtou"
                   (mksignature (AST.Tfloat :: nil) AST.Tlong cc_default))
    (Tcons tdouble Tnil) tulong cc_default)) ::
(___compcert_i64_stod,
  Gfun(External (EF_runtime "__compcert_i64_stod"
                   (mksignature (AST.Tlong :: nil) AST.Tfloat cc_default))
    (Tcons tlong Tnil) tdouble cc_default)) ::
(___compcert_i64_utod,
  Gfun(External (EF_runtime "__compcert_i64_utod"
                   (mksignature (AST.Tlong :: nil) AST.Tfloat cc_default))
    (Tcons tulong Tnil) tdouble cc_default)) ::
(___compcert_i64_stof,
  Gfun(External (EF_runtime "__compcert_i64_stof"
                   (mksignature (AST.Tlong :: nil) AST.Tsingle cc_default))
    (Tcons tlong Tnil) tfloat cc_default)) ::
(___compcert_i64_utof,
  Gfun(External (EF_runtime "__compcert_i64_utof"
                   (mksignature (AST.Tlong :: nil) AST.Tsingle cc_default))
    (Tcons tulong Tnil) tfloat cc_default)) ::
(___compcert_i64_sdiv,
  Gfun(External (EF_runtime "__compcert_i64_sdiv"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tlong Tnil)) tlong
    cc_default)) ::
(___compcert_i64_udiv,
  Gfun(External (EF_runtime "__compcert_i64_udiv"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tulong (Tcons tulong Tnil)) tulong
    cc_default)) ::
(___compcert_i64_smod,
  Gfun(External (EF_runtime "__compcert_i64_smod"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tlong Tnil)) tlong
    cc_default)) ::
(___compcert_i64_umod,
```

```
  Gfun(External (EF_runtime "__compcert_i64_umod"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tulong (Tcons tulong Tnil)) tulong
    cc_default)) ::
(___compcert_i64_shl,
  Gfun(External (EF_runtime "__compcert_i64_shl"
                   (mksignature (AST.Tlong :: AST.Tint :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tint Tnil)) tlong
    cc_default)) ::
(___compcert_i64_shr,
  Gfun(External (EF_runtime "__compcert_i64_shr"
                   (mksignature (AST.Tlong :: AST.Tint :: nil) AST.Tlong
                     cc_default)) (Tcons tulong (Tcons tint Tnil)) tulong
    cc_default)) ::
(___compcert_i64_sar,
  Gfun(External (EF_runtime "__compcert_i64_sar"
                   (mksignature (AST.Tlong :: AST.Tint :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tint Tnil)) tlong
    cc_default)) ::
(___compcert_i64_smulh,
  Gfun(External (EF_runtime "__compcert_i64_smulh"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tlong Tnil)) tlong
    cc_default)) ::
(___compcert_i64_umulh,
  Gfun(External (EF_runtime "__compcert_i64_umulh"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tulong (Tcons tulong Tnil)) tulong
    cc_default)) ::
(___builtin_clz,
  Gfun(External (EF_builtin "__builtin_clz"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_clzl,
  Gfun(External (EF_builtin "__builtin_clzl"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_clzll,
  Gfun(External (EF_builtin "__builtin_clzll"
                   (mksignature (AST.Tlong :: nil) AST.Tint cc_default))
    (Tcons tulong Tnil) tint cc_default)) ::
(___builtin_ctz,
```

```
  Gfun(External (EF_builtin "__builtin_ctz"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
     (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_ctzl,
  Gfun(External (EF_builtin "__builtin_ctzl"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
     (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_ctzll,
  Gfun(External (EF_builtin "__builtin_ctzll"
                   (mksignature (AST.Tlong :: nil) AST.Tint cc_default))
     (Tcons tulong Tnil) tint cc_default)) ::
(___builtin_fmax,
  Gfun(External (EF_builtin "__builtin_fmax"
                   (mksignature (AST.Tfloat :: AST.Tfloat :: nil) AST.Tfloat
                     cc_default)) (Tcons tdouble (Tcons tdouble Tnil))
     tdouble cc_default)) ::
(___builtin_fmin,
  Gfun(External (EF_builtin "__builtin_fmin"
                   (mksignature (AST.Tfloat :: AST.Tfloat :: nil) AST.Tfloat
                     cc_default)) (Tcons tdouble (Tcons tdouble Tnil))
     tdouble cc_default)) ::
(___builtin_fmadd,
  Gfun(External (EF_builtin "__builtin_fmadd"
                   (mksignature
                     (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                     AST.Tfloat cc_default))
     (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
     cc_default)) ::
(___builtin_fmsub,
  Gfun(External (EF_builtin "__builtin_fmsub"
                   (mksignature
                     (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                     AST.Tfloat cc_default))
     (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
     cc_default)) ::
(___builtin_fnmadd,
  Gfun(External (EF_builtin "__builtin_fnmadd"
                   (mksignature
                     (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                     AST.Tfloat cc_default))
     (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
     cc_default)) ::
```

```
  (___builtin_fnmsub,
   Gfun(External (EF_builtin "__builtin_fnmsub"
                   (mksignature
                     (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                     AST.Tfloat cc_default))
     (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
     cc_default)) ::
  (___builtin_read16_reversed,
   Gfun(External (EF_builtin "__builtin_read16_reversed"
                   (mksignature (AST.Tint :: nil) AST.Tint16unsigned
                     cc_default)) (Tcons (tptr tushort) Tnil) tushort
     cc_default)) ::
  (___builtin_read32_reversed,
   Gfun(External (EF_builtin "__builtin_read32_reversed"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
     (Tcons (tptr tuint) Tnil) tuint cc_default)) ::
  (___builtin_write16_reversed,
   Gfun(External (EF_builtin "__builtin_write16_reversed"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                     cc_default)) (Tcons (tptr tushort) (Tcons tushort Tnil))
     tvoid cc_default)) ::
  (___builtin_write32_reversed,
   Gfun(External (EF_builtin "__builtin_write32_reversed"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                     cc_default)) (Tcons (tptr tuint) (Tcons tuint Tnil))
     tvoid cc_default)) ::
  (___builtin_debug,
   Gfun(External (EF_external "__builtin_debug"
                   (mksignature (AST.Tint :: nil) AST.Tvoid
                     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
     (Tcons tint Tnil) tvoid
     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
  (_three, Gvar v_three) :: (_sumlist, Gfun(Internal f_sumlist)) ::
  (_reverse, Gfun(Internal f_reverse)) :: (_main, Gfun(Internal f_main)) ::
  nil).

Definition public_idents : list ident :=
(_main :: _reverse :: _sumlist :: _three :: ___builtin_debug ::
  ___builtin_write32_reversed :: ___builtin_write16_reversed ::
  ___builtin_read32_reversed :: ___builtin_read16_reversed ::
  ___builtin_fnmsub :: ___builtin_fnmadd :: ___builtin_fmsub ::
  ___builtin_fmadd :: ___builtin_fmin :: ___builtin_fmax ::
  ___builtin_ctzll :: ___builtin_ctzl :: ___builtin_ctz :: ___builtin_clzll ::
```

```
  ___builtin_clzl :: ___builtin_clz :: ___compcert_i64_umulh ::
  ___compcert_i64_smulh :: ___compcert_i64_sar :: ___compcert_i64_shr ::
  ___compcert_i64_shl :: ___compcert_i64_umod :: ___compcert_i64_smod ::
  ___compcert_i64_udiv :: ___compcert_i64_sdiv :: ___compcert_i64_utof ::
  ___compcert_i64_stof :: ___compcert_i64_utod :: ___compcert_i64_stod ::
  ___compcert_i64_dtou :: ___compcert_i64_dtos :: ___compcert_va_composite ::
  ___compcert_va_float64 :: ___compcert_va_int64 :: ___compcert_va_int32 ::
  ___builtin_va_end :: ___builtin_va_copy :: ___builtin_va_arg ::
  ___builtin_va_start :: ___builtin_membar :: ___builtin_annot_intval ::
  ___builtin_annot :: ___builtin_sel :: ___builtin_memcpy_aligned ::
  ___builtin_fsqrt :: ___builtin_fabs :: ___builtin_bswap16 ::
  ___builtin_bswap32 :: ___builtin_bswap :: ___builtin_bswap64 ::
  ___builtin_ais_annot :: nil).

Definition prog : Clight.program :=
  mkprogram composites global_definitions public_idents _main Logic.I.
```

Chapter 3

Library VC.append

```
From Coq Require Import String List ZArith.
From compcert Require Import Coqlib Integers Floats AST Ctypes Cop Clight Clightdefs.
Local Open Scope Z_scope.
Local Open Scope string_scope.

Module INFO.
  Definition version := "3.7".
  Definition build_number := "".
  Definition build_tag := "".
  Definition arch := "x86".
  Definition model := "32sse2".
  Definition abi := "standard".
  Definition bitsize := 32.
  Definition big_endian := false.
  Definition source_file := "append.c".
  Definition normalized := true.
End INFO.

Definition ___builtin_ais_annot : ident := 4%positive.
Definition ___builtin_annot : ident := 13%positive.
Definition ___builtin_annot_intval : ident := 14%positive.
Definition ___builtin_bswap : ident := 6%positive.
Definition ___builtin_bswap16 : ident := 8%positive.
Definition ___builtin_bswap32 : ident := 7%positive.
Definition ___builtin_bswap64 : ident := 5%positive.
Definition ___builtin_clz : ident := 39%positive.
Definition ___builtin_clzl : ident := 40%positive.
Definition ___builtin_clzll : ident := 41%positive.
Definition ___builtin_ctz : ident := 42%positive.
Definition ___builtin_ctzl : ident := 43%positive.
Definition ___builtin_ctzll : ident := 44%positive.
```

```
Definition ___builtin_debug : ident := 55%positive.
Definition ___builtin_fabs : ident := 9%positive.
Definition ___builtin_fmadd : ident := 47%positive.
Definition ___builtin_fmax : ident := 45%positive.
Definition ___builtin_fmin : ident := 46%positive.
Definition ___builtin_fmsub : ident := 48%positive.
Definition ___builtin_fnmadd : ident := 49%positive.
Definition ___builtin_fnmsub : ident := 50%positive.
Definition ___builtin_fsqrt : ident := 10%positive.
Definition ___builtin_membar : ident := 15%positive.
Definition ___builtin_memcpy_aligned : ident := 11%positive.
Definition ___builtin_read16_reversed : ident := 51%positive.
Definition ___builtin_read32_reversed : ident := 52%positive.
Definition ___builtin_sel : ident := 12%positive.
Definition ___builtin_va_arg : ident := 17%positive.
Definition ___builtin_va_copy : ident := 18%positive.
Definition ___builtin_va_end : ident := 19%positive.
Definition ___builtin_va_start : ident := 16%positive.
Definition ___builtin_write16_reversed : ident := 53%positive.
Definition ___builtin_write32_reversed : ident := 54%positive.
Definition ___compcert_i64_dtos : ident := 24%positive.
Definition ___compcert_i64_dtou : ident := 25%positive.
Definition ___compcert_i64_sar : ident := 36%positive.
Definition ___compcert_i64_sdiv : ident := 30%positive.
Definition ___compcert_i64_shl : ident := 34%positive.
Definition ___compcert_i64_shr : ident := 35%positive.
Definition ___compcert_i64_smod : ident := 32%positive.
Definition ___compcert_i64_smulh : ident := 37%positive.
Definition ___compcert_i64_stod : ident := 26%positive.
Definition ___compcert_i64_stof : ident := 28%positive.
Definition ___compcert_i64_udiv : ident := 31%positive.
Definition ___compcert_i64_umod : ident := 33%positive.
Definition ___compcert_i64_umulh : ident := 38%positive.
Definition ___compcert_i64_utod : ident := 27%positive.
Definition ___compcert_i64_utof : ident := 29%positive.
Definition ___compcert_va_composite : ident := 23%positive.
Definition ___compcert_va_float64 : ident := 22%positive.
Definition ___compcert_va_int32 : ident := 20%positive.
Definition ___compcert_va_int64 : ident := 21%positive.
Definition _append : ident := 60%positive.
Definition _append2 : ident := 63%positive.
Definition _curp : ident := 62%positive.
```

```
Definition _head : ident := 1%positive.
Definition _list : ident := 2%positive.
Definition _main : ident := 64%positive.
Definition _retp : ident := 61%positive.
Definition _t : ident := 58%positive.
Definition _tail : ident := 3%positive.
Definition _u : ident := 59%positive.
Definition _x : ident := 56%positive.
Definition _y : ident := 57%positive.
Definition _t'1 : ident := 65%positive.
Definition _t'2 : ident := 66%positive.
Definition _t'3 : ident := 67%positive.

Definition f_append := {|
  fn_return := (tptr (Tstruct _list noattr));
  fn_callconv := cc_default;
  fn_params := ((_x, (tptr (Tstruct _list noattr))) ::
                (_y, (tptr (Tstruct _list noattr))) :: nil);
  fn_vars := nil;
  fn_temps := ((_t, (tptr (Tstruct _list noattr))) ::
               (_u, (tptr (Tstruct _list noattr))) :: nil);
  fn_body :=
(Sifthenelse (Ebinop Oeq (Etempvar _x (tptr (Tstruct _list noattr)))
               (Ecast (Econst_int (Int.repr 0) tint) (tptr tvoid)) tint)
  (Sreturn (Some (Etempvar _y (tptr (Tstruct _list noattr)))))
  (Ssequence
    (Sset _t (Etempvar _x (tptr (Tstruct _list noattr))))
    (Ssequence
      (Sset _u
        (Efield
          (Ederef (Etempvar _t (tptr (Tstruct _list noattr)))
            (Tstruct _list noattr)) _tail (tptr (Tstruct _list noattr))))
      (Ssequence
        (Swhile
          (Ebinop One (Etempvar _u (tptr (Tstruct _list noattr)))
            (Ecast (Econst_int (Int.repr 0) tint) (tptr tvoid)) tint)
          (Ssequence
            (Sset _t (Etempvar _u (tptr (Tstruct _list noattr))))
            (Sset _u
              (Efield
                (Ederef (Etempvar _t (tptr (Tstruct _list noattr)))
                  (Tstruct _list noattr)) _tail
                (tptr (Tstruct _list noattr))))))
```

```
        (Ssequence
          (Sassign
            (Efield
              (Ederef (Etempvar _t (tptr (Tstruct _list noattr)))
                (Tstruct _list noattr)) _tail (tptr (Tstruct _list noattr)))
            (Etempvar _y (tptr (Tstruct _list noattr))))
          (Sreturn (Some (Etempvar _x (tptr (Tstruct _list noattr))))))))))
|}.

Definition f_append2 := {|
  fn_return := (tptr (Tstruct _list noattr));
  fn_callconv := cc_default;
  fn_params := ((_x, (tptr (Tstruct _list noattr))) ::
                (_y, (tptr (Tstruct _list noattr))) :: nil);
  fn_vars := ((_x, (tptr (Tstruct _list noattr))) :: nil);
  fn_temps := ((_retp, (tptr (tptr (Tstruct _list noattr)))) ::
               (_curp, (tptr (tptr (Tstruct _list noattr)))) ::
               (_t'3, (tptr (Tstruct _list noattr))) ::
               (_t'2, (tptr (Tstruct _list noattr))) ::
               (_t'1, (tptr (Tstruct _list noattr))) :: nil);
  fn_body :=
(Ssequence
  (Sassign (Evar _x (tptr (Tstruct _list noattr)))
    (Etempvar _x (tptr (Tstruct _list noattr))))
  (Ssequence
    (Sset _retp
      (Eaddrof (Evar _x (tptr (Tstruct _list noattr)))
        (tptr (tptr (Tstruct _list noattr)))))
    (Ssequence
      (Sset _curp
        (Eaddrof (Evar _x (tptr (Tstruct _list noattr)))
          (tptr (tptr (Tstruct _list noattr)))))
      (Ssequence
        (Sloop
          (Ssequence
            (Ssequence
              (Sset _t'3
                (Ederef (Etempvar _curp (tptr (tptr (Tstruct _list noattr))))
                  (tptr (Tstruct _list noattr))))
              (Sifthenelse (Ebinop One
                             (Etempvar _t'3 (tptr (Tstruct _list noattr)))
                             (Ecast (Econst_int (Int.repr 0) tint)
                               (tptr tvoid)) tint)
```

```
                  Sskip
                  Sbreak))
              (Ssequence
                (Sset _t'2
                  (Ederef (Etempvar _curp (tptr (tptr (Tstruct _list noattr))))
                    (tptr (Tstruct _list noattr))))
                (Sset _curp
                  (Eaddrof
                    (Efield
                      (Ederef (Etempvar _t'2 (tptr (Tstruct _list noattr)))
                        (Tstruct _list noattr)) _tail
                      (tptr (Tstruct _list noattr)))
                    (tptr (tptr (Tstruct _list noattr)))))))
            Sskip)
          (Ssequence
            (Sassign
              (Ederef (Etempvar _curp (tptr (tptr (Tstruct _list noattr))))
                (tptr (Tstruct _list noattr)))
              (Etempvar _y (tptr (Tstruct _list noattr))))
            (Ssequence
              (Sset _t'1
                (Ederef (Etempvar _retp (tptr (tptr (Tstruct _list noattr))))
                  (tptr (Tstruct _list noattr))))
              (Sreturn (Some (Etempvar _t'1 (tptr (Tstruct _list noattr))))))))))
|}.
Definition composites : list composite_definition :=
(Composite _list Struct
   ((_head, tint) :: (_tail, (tptr (Tstruct _list noattr))) :: nil)
   noattr :: nil).
Definition global_definitions : list (ident × globdef fundef type) :=
((___builtin_ais_annot,
   Gfun(External (EF_builtin "__builtin_ais_annot"
                   (mksignature (AST.Tint :: nil) AST.Tvoid
                     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
     (Tcons (tptr tschar) Tnil) tvoid
     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
 (___builtin_bswap64,
   Gfun(External (EF_builtin "__builtin_bswap64"
                   (mksignature (AST.Tlong :: nil) AST.Tlong cc_default))
     (Tcons tulong Tnil) tulong cc_default)) ::
 (___builtin_bswap,
   Gfun(External (EF_builtin "__builtin_bswap"
```

```
                    (mksignature (AST.Tint :: nil) AST.Tint cc_default))
   (Tcons tuint Tnil) tuint cc_default)) ::
(___builtin_bswap32,
  Gfun(External (EF_builtin "__builtin_bswap32"
                    (mksignature (AST.Tint :: nil) AST.Tint cc_default))
   (Tcons tuint Tnil) tuint cc_default)) ::
(___builtin_bswap16,
  Gfun(External (EF_builtin "__builtin_bswap16"
                    (mksignature (AST.Tint :: nil) AST.Tint16unsigned
                      cc_default)) (Tcons tushort Tnil) tushort cc_default)) ::
(___builtin_fabs,
  Gfun(External (EF_builtin "__builtin_fabs"
                    (mksignature (AST.Tfloat :: nil) AST.Tfloat cc_default))
   (Tcons tdouble Tnil) tdouble cc_default)) ::
(___builtin_fsqrt,
  Gfun(External (EF_builtin "__builtin_fsqrt"
                    (mksignature (AST.Tfloat :: nil) AST.Tfloat cc_default))
   (Tcons tdouble Tnil) tdouble cc_default)) ::
(___builtin_memcpy_aligned,
  Gfun(External (EF_builtin "__builtin_memcpy_aligned"
                    (mksignature
                      (AST.Tint :: AST.Tint :: AST.Tint :: AST.Tint :: nil)
                      AST.Tvoid cc_default))
   (Tcons (tptr tvoid)
     (Tcons (tptr tvoid) (Tcons tuint (Tcons tuint Tnil)))) tvoid
   cc_default)) ::
(___builtin_sel,
  Gfun(External (EF_builtin "__builtin_sel"
                    (mksignature (AST.Tint :: nil) AST.Tvoid
                      {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
   (Tcons tbool Tnil) tvoid
   {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
(___builtin_annot,
  Gfun(External (EF_builtin "__builtin_annot"
                    (mksignature (AST.Tint :: nil) AST.Tvoid
                      {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
   (Tcons (tptr tschar) Tnil) tvoid
   {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
(___builtin_annot_intval,
  Gfun(External (EF_builtin "__builtin_annot_intval"
                    (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tint
                      cc_default)) (Tcons (tptr tschar) (Tcons tint Tnil))
```

```
    tint cc_default)) ::
(___builtin_membar,
  Gfun(External (EF_builtin "__builtin_membar"
                   (mksignature nil AST.Tvoid cc_default)) Tnil tvoid
    cc_default)) ::
(___builtin_va_start,
  Gfun(External (EF_builtin "__builtin_va_start"
                   (mksignature (AST.Tint :: nil) AST.Tvoid cc_default))
    (Tcons (tptr tvoid) Tnil) tvoid cc_default)) ::
(___builtin_va_arg,
  Gfun(External (EF_builtin "__builtin_va_arg"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                     cc_default)) (Tcons (tptr tvoid) (Tcons tuint Tnil))
    tvoid cc_default)) ::
(___builtin_va_copy,
  Gfun(External (EF_builtin "__builtin_va_copy"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                     cc_default))
    (Tcons (tptr tvoid) (Tcons (tptr tvoid) Tnil)) tvoid cc_default)) ::
(___builtin_va_end,
  Gfun(External (EF_builtin "__builtin_va_end"
                   (mksignature (AST.Tint :: nil) AST.Tvoid cc_default))
    (Tcons (tptr tvoid) Tnil) tvoid cc_default)) ::
(___compcert_va_int32,
  Gfun(External (EF_external "__compcert_va_int32"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons (tptr tvoid) Tnil) tuint cc_default)) ::
(___compcert_va_int64,
  Gfun(External (EF_external "__compcert_va_int64"
                   (mksignature (AST.Tint :: nil) AST.Tlong cc_default))
    (Tcons (tptr tvoid) Tnil) tulong cc_default)) ::
(___compcert_va_float64,
  Gfun(External (EF_external "__compcert_va_float64"
                   (mksignature (AST.Tint :: nil) AST.Tfloat cc_default))
    (Tcons (tptr tvoid) Tnil) tdouble cc_default)) ::
(___compcert_va_composite,
  Gfun(External (EF_external "__compcert_va_composite"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tint
                     cc_default)) (Tcons (tptr tvoid) (Tcons tuint Tnil))
    (tptr tvoid) cc_default)) ::
(___compcert_i64_dtos,
  Gfun(External (EF_runtime "__compcert_i64_dtos"
```

```
                    (mksignature (AST.Tfloat :: nil) AST.Tlong cc_default))
    (Tcons tdouble Tnil) tlong cc_default)) ::
(___compcert_i64_dtou,
  Gfun(External (EF_runtime "__compcert_i64_dtou"
                    (mksignature (AST.Tfloat :: nil) AST.Tlong cc_default))
    (Tcons tdouble Tnil) tulong cc_default)) ::
(___compcert_i64_stod,
  Gfun(External (EF_runtime "__compcert_i64_stod"
                    (mksignature (AST.Tlong :: nil) AST.Tfloat cc_default))
    (Tcons tlong Tnil) tdouble cc_default)) ::
(___compcert_i64_utod,
  Gfun(External (EF_runtime "__compcert_i64_utod"
                    (mksignature (AST.Tlong :: nil) AST.Tfloat cc_default))
    (Tcons tulong Tnil) tdouble cc_default)) ::
(___compcert_i64_stof,
  Gfun(External (EF_runtime "__compcert_i64_stof"
                    (mksignature (AST.Tlong :: nil) AST.Tsingle cc_default))
    (Tcons tlong Tnil) tfloat cc_default)) ::
(___compcert_i64_utof,
  Gfun(External (EF_runtime "__compcert_i64_utof"
                    (mksignature (AST.Tlong :: nil) AST.Tsingle cc_default))
    (Tcons tulong Tnil) tfloat cc_default)) ::
(___compcert_i64_sdiv,
  Gfun(External (EF_runtime "__compcert_i64_sdiv"
                    (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                      cc_default)) (Tcons tlong (Tcons tlong Tnil)) tlong
    cc_default)) ::
(___compcert_i64_udiv,
  Gfun(External (EF_runtime "__compcert_i64_udiv"
                    (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                      cc_default)) (Tcons tulong (Tcons tulong Tnil)) tulong
    cc_default)) ::
(___compcert_i64_smod,
  Gfun(External (EF_runtime "__compcert_i64_smod"
                    (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                      cc_default)) (Tcons tlong (Tcons tlong Tnil)) tlong
    cc_default)) ::
(___compcert_i64_umod,
  Gfun(External (EF_runtime "__compcert_i64_umod"
                    (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                      cc_default)) (Tcons tulong (Tcons tulong Tnil)) tulong
    cc_default)) ::
```

```
(___compcert_i64_shl,
  Gfun(External (EF_runtime "__compcert_i64_shl"
                   (mksignature (AST.Tlong :: AST.Tint :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tint Tnil)) tlong
    cc_default)) ::
(___compcert_i64_shr,
  Gfun(External (EF_runtime "__compcert_i64_shr"
                   (mksignature (AST.Tlong :: AST.Tint :: nil) AST.Tlong
                     cc_default)) (Tcons tulong (Tcons tint Tnil)) tulong
    cc_default)) ::
(___compcert_i64_sar,
  Gfun(External (EF_runtime "__compcert_i64_sar"
                   (mksignature (AST.Tlong :: AST.Tint :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tint Tnil)) tlong
    cc_default)) ::
(___compcert_i64_smulh,
  Gfun(External (EF_runtime "__compcert_i64_smulh"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tlong Tnil)) tlong
    cc_default)) ::
(___compcert_i64_umulh,
  Gfun(External (EF_runtime "__compcert_i64_umulh"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tulong (Tcons tulong Tnil)) tulong
    cc_default)) ::
(___builtin_clz,
  Gfun(External (EF_builtin "__builtin_clz"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_clzl,
  Gfun(External (EF_builtin "__builtin_clzl"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_clzll,
  Gfun(External (EF_builtin "__builtin_clzll"
                   (mksignature (AST.Tlong :: nil) AST.Tint cc_default))
    (Tcons tulong Tnil) tint cc_default)) ::
(___builtin_ctz,
  Gfun(External (EF_builtin "__builtin_ctz"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_ctzl,
```

```
  Gfun(External (EF_builtin "__builtin_ctzl"
                     (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_ctzll,
  Gfun(External (EF_builtin "__builtin_ctzll"
                     (mksignature (AST.Tlong :: nil) AST.Tint cc_default))
    (Tcons tulong Tnil) tint cc_default)) ::
(___builtin_fmax,
  Gfun(External (EF_builtin "__builtin_fmax"
                     (mksignature (AST.Tfloat :: AST.Tfloat :: nil) AST.Tfloat
                       cc_default)) (Tcons tdouble (Tcons tdouble Tnil))
    tdouble cc_default)) ::
(___builtin_fmin,
  Gfun(External (EF_builtin "__builtin_fmin"
                     (mksignature (AST.Tfloat :: AST.Tfloat :: nil) AST.Tfloat
                       cc_default)) (Tcons tdouble (Tcons tdouble Tnil))
    tdouble cc_default)) ::
(___builtin_fmadd,
  Gfun(External (EF_builtin "__builtin_fmadd"
                     (mksignature
                       (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                       AST.Tfloat cc_default))
    (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
    cc_default)) ::
(___builtin_fmsub,
  Gfun(External (EF_builtin "__builtin_fmsub"
                     (mksignature
                       (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                       AST.Tfloat cc_default))
    (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
    cc_default)) ::
(___builtin_fnmadd,
  Gfun(External (EF_builtin "__builtin_fnmadd"
                     (mksignature
                       (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                       AST.Tfloat cc_default))
    (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
    cc_default)) ::
(___builtin_fnmsub,
  Gfun(External (EF_builtin "__builtin_fnmsub"
                     (mksignature
                       (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
```

```
                         AST.Tfloat cc_default))
      (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
      cc_default)) ::
 (___builtin_read16_reversed,
    Gfun(External (EF_builtin "__builtin_read16_reversed"
                    (mksignature (AST.Tint :: nil) AST.Tint16unsigned
                      cc_default)) (Tcons (tptr tushort) Tnil) tushort
      cc_default)) ::
 (___builtin_read32_reversed,
    Gfun(External (EF_builtin "__builtin_read32_reversed"
                    (mksignature (AST.Tint :: nil) AST.Tint cc_default))
      (Tcons (tptr tuint) Tnil) tuint cc_default)) ::
 (___builtin_write16_reversed,
    Gfun(External (EF_builtin "__builtin_write16_reversed"
                    (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                      cc_default)) (Tcons (tptr tushort) (Tcons tushort Tnil))
      tvoid cc_default)) ::
 (___builtin_write32_reversed,
    Gfun(External (EF_builtin "__builtin_write32_reversed"
                    (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                      cc_default)) (Tcons (tptr tuint) (Tcons tuint Tnil))
      tvoid cc_default)) ::
 (___builtin_debug,
    Gfun(External (EF_external "__builtin_debug"
                    (mksignature (AST.Tint :: nil) AST.Tvoid
                      {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
      (Tcons tint Tnil) tvoid
      {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
 (_append, Gfun(Internal f_append)) ::
 (_append2, Gfun(Internal f_append2)) :: nil).

Definition public_idents : list ident :=
(_append2 :: _append :: ___builtin_debug :: ___builtin_write32_reversed ::
 ___builtin_write16_reversed :: ___builtin_read32_reversed ::
 ___builtin_read16_reversed :: ___builtin_fnmsub :: ___builtin_fnmadd ::
 ___builtin_fmsub :: ___builtin_fmadd :: ___builtin_fmin ::
 ___builtin_fmax :: ___builtin_ctzll :: ___builtin_ctzl :: ___builtin_ctz ::
 ___builtin_clzll :: ___builtin_clzl :: ___builtin_clz ::
 ___compcert_i64_umulh :: ___compcert_i64_smulh :: ___compcert_i64_sar ::
 ___compcert_i64_shr :: ___compcert_i64_shl :: ___compcert_i64_umod ::
 ___compcert_i64_smod :: ___compcert_i64_udiv :: ___compcert_i64_sdiv ::
 ___compcert_i64_utof :: ___compcert_i64_stof :: ___compcert_i64_utod ::
 ___compcert_i64_stod :: ___compcert_i64_dtou :: ___compcert_i64_dtos ::
```

```
___compcert_va_composite :: ___compcert_va_float64 ::
___compcert_va_int64 :: ___compcert_va_int32 :: ___builtin_va_end ::
___builtin_va_copy :: ___builtin_va_arg :: ___builtin_va_start ::
___builtin_membar :: ___builtin_annot_intval :: ___builtin_annot ::
___builtin_sel :: ___builtin_memcpy_aligned :: ___builtin_fsqrt ::
___builtin_fabs :: ___builtin_bswap16 :: ___builtin_bswap32 ::
___builtin_bswap :: ___builtin_bswap64 :: ___builtin_ais_annot :: nil).

Definition prog : Clight.program :=
  mkprogram composites global_definitions public_idents _main Logic.I.
```

Chapter 4

Library VC.stack

```
From Coq Require Import String List ZArith.
From compcert Require Import Coqlib Integers Floats AST Ctypes Cop Clight Clightdefs.
Local Open Scope Z_scope.
Local Open Scope string_scope.

Module Info.
  Definition version := "3.7".
  Definition build_number := "".
  Definition build_tag := "".
  Definition arch := "x86".
  Definition model := "32sse2".
  Definition abi := "standard".
  Definition bitsize := 32.
  Definition big_endian := false.
  Definition source_file := "stack.c".
  Definition normalized := true.
End Info.

Definition ___builtin_ais_annot : ident := 6%positive.
Definition ___builtin_annot : ident := 15%positive.
Definition ___builtin_annot_intval : ident := 16%positive.
Definition ___builtin_bswap : ident := 8%positive.
Definition ___builtin_bswap16 : ident := 10%positive.
Definition ___builtin_bswap32 : ident := 9%positive.
Definition ___builtin_bswap64 : ident := 7%positive.
Definition ___builtin_clz : ident := 41%positive.
Definition ___builtin_clzl : ident := 42%positive.
Definition ___builtin_clzll : ident := 43%positive.
Definition ___builtin_ctz : ident := 44%positive.
Definition ___builtin_ctzl : ident := 45%positive.
Definition ___builtin_ctzll : ident := 46%positive.
```

```
Definition ___builtin_debug : ident := 57%positive.
Definition ___builtin_fabs : ident := 11%positive.
Definition ___builtin_fmadd : ident := 49%positive.
Definition ___builtin_fmax : ident := 47%positive.
Definition ___builtin_fmin : ident := 48%positive.
Definition ___builtin_fmsub : ident := 50%positive.
Definition ___builtin_fnmadd : ident := 51%positive.
Definition ___builtin_fnmsub : ident := 52%positive.
Definition ___builtin_fsqrt : ident := 12%positive.
Definition ___builtin_membar : ident := 17%positive.
Definition ___builtin_memcpy_aligned : ident := 13%positive.
Definition ___builtin_read16_reversed : ident := 53%positive.
Definition ___builtin_read32_reversed : ident := 54%positive.
Definition ___builtin_sel : ident := 14%positive.
Definition ___builtin_va_arg : ident := 19%positive.
Definition ___builtin_va_copy : ident := 20%positive.
Definition ___builtin_va_end : ident := 21%positive.
Definition ___builtin_va_start : ident := 18%positive.
Definition ___builtin_write16_reversed : ident := 55%positive.
Definition ___builtin_write32_reversed : ident := 56%positive.
Definition ___compcert_i64_dtos : ident := 26%positive.
Definition ___compcert_i64_dtou : ident := 27%positive.
Definition ___compcert_i64_sar : ident := 38%positive.
Definition ___compcert_i64_sdiv : ident := 32%positive.
Definition ___compcert_i64_shl : ident := 36%positive.
Definition ___compcert_i64_shr : ident := 37%positive.
Definition ___compcert_i64_smod : ident := 34%positive.
Definition ___compcert_i64_smulh : ident := 39%positive.
Definition ___compcert_i64_stod : ident := 28%positive.
Definition ___compcert_i64_stof : ident := 30%positive.
Definition ___compcert_i64_udiv : ident := 33%positive.
Definition ___compcert_i64_umod : ident := 35%positive.
Definition ___compcert_i64_umulh : ident := 40%positive.
Definition ___compcert_i64_utod : ident := 29%positive.
Definition ___compcert_i64_utof : ident := 31%positive.
Definition ___compcert_va_composite : ident := 25%positive.
Definition ___compcert_va_float64 : ident := 24%positive.
Definition ___compcert_va_int32 : ident := 22%positive.
Definition ___compcert_va_int64 : ident := 23%positive.
Definition _cons : ident := 2%positive.
Definition _exit : ident := 60%positive.
Definition _free : ident := 59%positive.
```

```
Definition _i : ident := 63%positive.
Definition _main : ident := 73%positive.
Definition _malloc : ident := 58%positive.
Definition _n : ident := 68%positive.
Definition _newstack : ident := 62%positive.
Definition _next : ident := 3%positive.
Definition _p : ident := 61%positive.
Definition _pop : ident := 66%positive.
Definition _pop_and_add : ident := 72%positive.
Definition _push : ident := 65%positive.
Definition _push_increasing : ident := 69%positive.
Definition _q : ident := 64%positive.
Definition _s : ident := 71%positive.
Definition _st : ident := 67%positive.
Definition _stack : ident := 5%positive.
Definition _t : ident := 70%positive.
Definition _top : ident := 4%positive.
Definition _value : ident := 1%positive.
Definition _t'1 : ident := 74%positive.
Definition _t'2 : ident := 75%positive.

Definition f_newstack := {|
  fn_return := (tptr (Tstruct _stack noattr));
  fn_callconv := cc_default;
  fn_params := nil;
  fn_vars := nil;
  fn_temps := ((_p, (tptr (Tstruct _stack noattr))) ::
               (_t'1, (tptr tvoid)) :: nil);
  fn_body :=
(Ssequence
  (Ssequence
    (Scall (Some _t'1)
      (Evar _malloc (Tfunction (Tcons tuint Tnil) (tptr tvoid) cc_default))
      ((Esizeof (Tstruct _stack noattr) tuint) :: nil))
    (Sset _p
      (Ecast (Etempvar _t'1 (tptr tvoid)) (tptr (Tstruct _stack noattr)))))
  (Ssequence
    (Sifthenelse (Eunop Onotbool (Etempvar _p (tptr (Tstruct _stack noattr)))
                   tint)
      (Scall None (Evar _exit (Tfunction (Tcons tint Tnil) tvoid cc_default))
        ((Econst_int (Int.repr 1) tint) :: nil))
      Sskip)
    (Ssequence
```

```
      (Sassign
        (Efield
          (Ederef (Etempvar _p (tptr (Tstruct _stack noattr)))
            (Tstruct _stack noattr)) _top (tptr (Tstruct _cons noattr)))
        (Ecast (Econst_int (Int.repr 0) tint) (tptr tvoid)))
      (Sreturn (Some (Etempvar _p (tptr (Tstruct _stack noattr)))))))
|}.

Definition f_push := {|
  fn_return := tvoid;
  fn_callconv := cc_default;
  fn_params := ((_p, (tptr (Tstruct _stack noattr))) :: (_i, tint) :: nil);
  fn_vars := nil;
  fn_temps := ((_q, (tptr (Tstruct _cons noattr))) :: (_t'1, (tptr tvoid)) ::
                (_t'2, (tptr (Tstruct _cons noattr))) :: nil);
  fn_body :=
(Ssequence
  (Ssequence
    (Scall (Some _t'1)
      (Evar _malloc (Tfunction (Tcons tuint Tnil) (tptr tvoid) cc_default))
      ((Esizeof (Tstruct _cons noattr) tuint) :: nil))
    (Sset _q
      (Ecast (Etempvar _t'1 (tptr tvoid)) (tptr (Tstruct _cons noattr)))))
  (Ssequence
    (Sifthenelse (Eunop Onotbool (Etempvar _q (tptr (Tstruct _cons noattr)))
                   tint)
      (Scall None (Evar _exit (Tfunction (Tcons tint Tnil) tvoid cc_default))
        ((Econst_int (Int.repr 1) tint) :: nil))
      Sskip)
    (Ssequence
      (Sassign
        (Efield
          (Ederef (Etempvar _q (tptr (Tstruct _cons noattr)))
            (Tstruct _cons noattr)) _value tint) (Etempvar _i tint))
      (Ssequence
        (Ssequence
          (Sset _t'2
            (Efield
              (Ederef (Etempvar _p (tptr (Tstruct _stack noattr)))
                (Tstruct _stack noattr)) _top (tptr (Tstruct _cons noattr))))
          (Sassign
            (Efield
              (Ederef (Etempvar _q (tptr (Tstruct _cons noattr)))
```

```
                (Tstruct _cons noattr)) _next (tptr (Tstruct _cons noattr)))
            (Etempvar _t'2 (tptr (Tstruct _cons noattr)))))
        (Sassign
          (Efield
            (Ederef (Etempvar _p (tptr (Tstruct _stack noattr)))
              (Tstruct _stack noattr)) _top (tptr (Tstruct _cons noattr)))
          (Etempvar _q (tptr (Tstruct _cons noattr))))))))
|}.
Definition f_pop := {|
  fn_return := tint;
  fn_callconv := cc_default;
  fn_params := ((_p, (tptr (Tstruct _stack noattr))) :: nil);
  fn_vars := nil;
  fn_temps := ((_q, (tptr (Tstruct _cons noattr))) :: (_i, tint) ::
               (_t'1, (tptr (Tstruct _cons noattr))) :: nil);
  fn_body :=
(Ssequence
  (Sset _q
    (Efield
      (Ederef (Etempvar _p (tptr (Tstruct _stack noattr)))
        (Tstruct _stack noattr)) _top (tptr (Tstruct _cons noattr))))
  (Ssequence
    (Ssequence
      (Sset _t'1
        (Efield
          (Ederef (Etempvar _q (tptr (Tstruct _cons noattr)))
            (Tstruct _cons noattr)) _next (tptr (Tstruct _cons noattr))))
      (Sassign
        (Efield
          (Ederef (Etempvar _p (tptr (Tstruct _stack noattr)))
            (Tstruct _stack noattr)) _top (tptr (Tstruct _cons noattr)))
        (Etempvar _t'1 (tptr (Tstruct _cons noattr)))))
    (Ssequence
      (Sset _i
        (Efield
          (Ederef (Etempvar _q (tptr (Tstruct _cons noattr)))
            (Tstruct _cons noattr)) _value tint))
      (Ssequence
        (Scall None
          (Evar _free (Tfunction (Tcons (tptr tvoid) Tnil) tvoid cc_default))
          ((Etempvar _q (tptr (Tstruct _cons noattr))) :: nil))
        (Sreturn (Some (Etempvar _i tint)))))))
```

```
|}.
Definition f_push_increasing := {|
  fn_return := tvoid;
  fn_callconv := cc_default;
  fn_params := ((_st, (tptr (Tstruct _stack noattr))) :: (_n, tint) :: nil);
  fn_vars := nil;
  fn_temps := ((_i, tint) :: nil);
  fn_body :=
(Ssequence
  (Sset _i (Econst_int (Int.repr 0) tint))
  (Swhile
    (Ebinop Olt (Etempvar _i tint) (Etempvar _n tint) tint)
    (Ssequence
      (Sset _i
        (Ebinop Oadd (Etempvar _i tint) (Econst_int (Int.repr 1) tint) tint))
      (Scall None
        (Evar _push (Tfunction
                      (Tcons (tptr (Tstruct _stack noattr))
                        (Tcons tint Tnil)) tvoid cc_default))
        ((Etempvar _st (tptr (Tstruct _stack noattr))) ::
         (Etempvar _i tint) :: nil)))))
|}.
Definition f_pop_and_add := {|
  fn_return := tint;
  fn_callconv := cc_default;
  fn_params := ((_st, (tptr (Tstruct _stack noattr))) :: (_n, tint) :: nil);
  fn_vars := nil;
  fn_temps := ((_i, tint) :: (_t, tint) :: (_s, tint) :: (_t'1, tint) :: nil);
  fn_body :=
(Ssequence
  (Sset _i (Econst_int (Int.repr 0) tint))
  (Ssequence
    (Sset _s (Econst_int (Int.repr 0) tint))
    (Ssequence
      (Swhile
        (Ebinop Olt (Etempvar _i tint) (Etempvar _n tint) tint)
        (Ssequence
          (Ssequence
            (Scall (Some _t'1)
              (Evar _pop (Tfunction
                           (Tcons (tptr (Tstruct _stack noattr)) Tnil) tint
                           cc_default))
```

```
              ((Etempvar _st (tptr (Tstruct _stack noattr))) :: nil))
            (Sset _t (Etempvar _t'1 tint)))
          (Ssequence
            (Sset _s
              (Ebinop Oadd (Etempvar _s tint) (Etempvar _t tint) tint))
            (Sset _i
              (Ebinop Oadd (Etempvar _i tint) (Econst_int (Int.repr 1) tint)
                tint)))))
      (Sreturn (Some (Etempvar _s tint))))))
|}.

Definition f_main := {|
  fn_return := tint;
  fn_callconv := cc_default;
  fn_params := nil;
  fn_vars := nil;
  fn_temps := ((_st, (tptr (Tstruct _stack noattr))) :: (_i, tint) ::
               (_t, tint) :: (_s, tint) :: (_t'2, tint) ::
               (_t'1, (tptr (Tstruct _stack noattr))) :: nil);
  fn_body :=
(Ssequence
  (Ssequence
    (Ssequence
      (Scall (Some _t'1)
        (Evar _newstack (Tfunction Tnil (tptr (Tstruct _stack noattr))
                          cc_default)) nil)
      (Sset _st (Etempvar _t'1 (tptr (Tstruct _stack noattr)))))
    (Ssequence
      (Scall None
        (Evar _push_increasing (Tfunction
                                 (Tcons (tptr (Tstruct _stack noattr))
                                   (Tcons tint Tnil)) tvoid cc_default))
        ((Etempvar _st (tptr (Tstruct _stack noattr))) ::
         (Econst_int (Int.repr 10) tint) :: nil))
      (Ssequence
        (Ssequence
          (Scall (Some _t'2)
            (Evar _pop_and_add (Tfunction
                                 (Tcons (tptr (Tstruct _stack noattr))
                                   (Tcons tint Tnil)) tint cc_default))
            ((Etempvar _st (tptr (Tstruct _stack noattr))) ::
             (Econst_int (Int.repr 10) tint) :: nil))
          (Sset _s (Etempvar _t'2 tint)))
```

```
          (Sreturn (Some (Etempvar _s tint))))))
   (Sreturn (Some (Econst_int (Int.repr 0) tint))))
|}.

Definition composites : list composite_definition :=
(Composite _cons Struct
    ((_value, tint) :: (_next, (tptr (Tstruct _cons noattr))) :: nil)
    noattr ::
 Composite _stack Struct
    ((_top, (tptr (Tstruct _cons noattr))) :: nil)
    noattr :: nil).

Definition global_definitions : list (ident × globdef fundef type) :=
((___builtin_ais_annot,
    Gfun(External (EF_builtin "__builtin_ais_annot"
                    (mksignature (AST.Tint :: nil) AST.Tvoid
                      {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
      (Tcons (tptr tschar) Tnil) tvoid
      {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
 (___builtin_bswap64,
    Gfun(External (EF_builtin "__builtin_bswap64"
                    (mksignature (AST.Tlong :: nil) AST.Tlong cc_default))
      (Tcons tulong Tnil) tulong cc_default)) ::
 (___builtin_bswap,
    Gfun(External (EF_builtin "__builtin_bswap"
                    (mksignature (AST.Tint :: nil) AST.Tint cc_default))
      (Tcons tuint Tnil) tuint cc_default)) ::
 (___builtin_bswap32,
    Gfun(External (EF_builtin "__builtin_bswap32"
                    (mksignature (AST.Tint :: nil) AST.Tint cc_default))
      (Tcons tuint Tnil) tuint cc_default)) ::
 (___builtin_bswap16,
    Gfun(External (EF_builtin "__builtin_bswap16"
                    (mksignature (AST.Tint :: nil) AST.Tint16unsigned
                      cc_default)) (Tcons tushort Tnil) tushort cc_default)) ::
 (___builtin_fabs,
    Gfun(External (EF_builtin "__builtin_fabs"
                    (mksignature (AST.Tfloat :: nil) AST.Tfloat cc_default))
      (Tcons tdouble Tnil) tdouble cc_default)) ::
 (___builtin_fsqrt,
    Gfun(External (EF_builtin "__builtin_fsqrt"
                    (mksignature (AST.Tfloat :: nil) AST.Tfloat cc_default))
      (Tcons tdouble Tnil) tdouble cc_default)) ::
 (___builtin_memcpy_aligned,
```

```
  Gfun(External (EF_builtin "__builtin_memcpy_aligned"
                   (mksignature
                     (AST.Tint :: AST.Tint :: AST.Tint :: AST.Tint :: nil)
                     AST.Tvoid cc_default))
    (Tcons (tptr tvoid)
      (Tcons (tptr tvoid) (Tcons tuint (Tcons tuint Tnil)))) tvoid
    cc_default)) ::
(___builtin_sel,
  Gfun(External (EF_builtin "__builtin_sel"
                   (mksignature (AST.Tint :: nil) AST.Tvoid
                     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
    (Tcons tbool Tnil) tvoid
    {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
(___builtin_annot,
  Gfun(External (EF_builtin "__builtin_annot"
                   (mksignature (AST.Tint :: nil) AST.Tvoid
                     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
    (Tcons (tptr tschar) Tnil) tvoid
    {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
(___builtin_annot_intval,
  Gfun(External (EF_builtin "__builtin_annot_intval"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tint
                     cc_default)) (Tcons (tptr tschar) (Tcons tint Tnil))
    tint cc_default)) ::
(___builtin_membar,
  Gfun(External (EF_builtin "__builtin_membar"
                   (mksignature nil AST.Tvoid cc_default)) Tnil tvoid
    cc_default)) ::
(___builtin_va_start,
  Gfun(External (EF_builtin "__builtin_va_start"
                   (mksignature (AST.Tint :: nil) AST.Tvoid cc_default))
    (Tcons (tptr tvoid) Tnil) tvoid cc_default)) ::
(___builtin_va_arg,
  Gfun(External (EF_builtin "__builtin_va_arg"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                     cc_default)) (Tcons (tptr tvoid) (Tcons tuint Tnil))
    tvoid cc_default)) ::
(___builtin_va_copy,
  Gfun(External (EF_builtin "__builtin_va_copy"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                     cc_default))
    (Tcons (tptr tvoid) (Tcons (tptr tvoid) Tnil)) tvoid cc_default)) ::
```

```
(___builtin_va_end,
  Gfun(External (EF_builtin "__builtin_va_end"
                   (mksignature (AST.Tint :: nil) AST.Tvoid cc_default))
    (Tcons (tptr tvoid) Tnil) tvoid cc_default)) ::
(___compcert_va_int32,
  Gfun(External (EF_external "__compcert_va_int32"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons (tptr tvoid) Tnil) tuint cc_default)) ::
(___compcert_va_int64,
  Gfun(External (EF_external "__compcert_va_int64"
                   (mksignature (AST.Tint :: nil) AST.Tlong cc_default))
    (Tcons (tptr tvoid) Tnil) tulong cc_default)) ::
(___compcert_va_float64,
  Gfun(External (EF_external "__compcert_va_float64"
                   (mksignature (AST.Tint :: nil) AST.Tfloat cc_default))
    (Tcons (tptr tvoid) Tnil) tdouble cc_default)) ::
(___compcert_va_composite,
  Gfun(External (EF_external "__compcert_va_composite"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tint
                     cc_default)) (Tcons (tptr tvoid) (Tcons tuint Tnil))
    (tptr tvoid) cc_default)) ::
(___compcert_i64_dtos,
  Gfun(External (EF_runtime "__compcert_i64_dtos"
                   (mksignature (AST.Tfloat :: nil) AST.Tlong cc_default))
    (Tcons tdouble Tnil) tlong cc_default)) ::
(___compcert_i64_dtou,
  Gfun(External (EF_runtime "__compcert_i64_dtou"
                   (mksignature (AST.Tfloat :: nil) AST.Tlong cc_default))
    (Tcons tdouble Tnil) tulong cc_default)) ::
(___compcert_i64_stod,
  Gfun(External (EF_runtime "__compcert_i64_stod"
                   (mksignature (AST.Tlong :: nil) AST.Tfloat cc_default))
    (Tcons tlong Tnil) tdouble cc_default)) ::
(___compcert_i64_utod,
  Gfun(External (EF_runtime "__compcert_i64_utod"
                   (mksignature (AST.Tlong :: nil) AST.Tfloat cc_default))
    (Tcons tulong Tnil) tdouble cc_default)) ::
(___compcert_i64_stof,
  Gfun(External (EF_runtime "__compcert_i64_stof"
                   (mksignature (AST.Tlong :: nil) AST.Tsingle cc_default))
    (Tcons tlong Tnil) tfloat cc_default)) ::
(___compcert_i64_utof,
```

```
  Gfun(External (EF_runtime "__compcert_i64_utof"
                   (mksignature (AST.Tlong :: nil) AST.Tsingle cc_default))
    (Tcons tulong Tnil) tfloat cc_default)) ::
(___compcert_i64_sdiv,
  Gfun(External (EF_runtime "__compcert_i64_sdiv"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tlong Tnil)) tlong
    cc_default)) ::
(___compcert_i64_udiv,
  Gfun(External (EF_runtime "__compcert_i64_udiv"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tulong (Tcons tulong Tnil)) tulong
    cc_default)) ::
(___compcert_i64_smod,
  Gfun(External (EF_runtime "__compcert_i64_smod"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tlong Tnil)) tlong
    cc_default)) ::
(___compcert_i64_umod,
  Gfun(External (EF_runtime "__compcert_i64_umod"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tulong (Tcons tulong Tnil)) tulong
    cc_default)) ::
(___compcert_i64_shl,
  Gfun(External (EF_runtime "__compcert_i64_shl"
                   (mksignature (AST.Tlong :: AST.Tint :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tint Tnil)) tlong
    cc_default)) ::
(___compcert_i64_shr,
  Gfun(External (EF_runtime "__compcert_i64_shr"
                   (mksignature (AST.Tlong :: AST.Tint :: nil) AST.Tlong
                     cc_default)) (Tcons tulong (Tcons tint Tnil)) tulong
    cc_default)) ::
(___compcert_i64_sar,
  Gfun(External (EF_runtime "__compcert_i64_sar"
                   (mksignature (AST.Tlong :: AST.Tint :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tint Tnil)) tlong
    cc_default)) ::
(___compcert_i64_smulh,
  Gfun(External (EF_runtime "__compcert_i64_smulh"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tlong Tnil)) tlong
```

```
    cc_default)) ::
(___compcert_i64_umulh,
  Gfun(External (EF_runtime "__compcert_i64_umulh"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tulong (Tcons tulong Tnil)) tulong
    cc_default)) ::
(___builtin_clz,
  Gfun(External (EF_builtin "__builtin_clz"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_clzl,
  Gfun(External (EF_builtin "__builtin_clzl"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_clzll,
  Gfun(External (EF_builtin "__builtin_clzll"
                   (mksignature (AST.Tlong :: nil) AST.Tint cc_default))
    (Tcons tulong Tnil) tint cc_default)) ::
(___builtin_ctz,
  Gfun(External (EF_builtin "__builtin_ctz"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_ctzl,
  Gfun(External (EF_builtin "__builtin_ctzl"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_ctzll,
  Gfun(External (EF_builtin "__builtin_ctzll"
                   (mksignature (AST.Tlong :: nil) AST.Tint cc_default))
    (Tcons tulong Tnil) tint cc_default)) ::
(___builtin_fmax,
  Gfun(External (EF_builtin "__builtin_fmax"
                   (mksignature (AST.Tfloat :: AST.Tfloat :: nil) AST.Tfloat
                     cc_default)) (Tcons tdouble (Tcons tdouble Tnil))
    tdouble cc_default)) ::
(___builtin_fmin,
  Gfun(External (EF_builtin "__builtin_fmin"
                   (mksignature (AST.Tfloat :: AST.Tfloat :: nil) AST.Tfloat
                     cc_default)) (Tcons tdouble (Tcons tdouble Tnil))
    tdouble cc_default)) ::
(___builtin_fmadd,
  Gfun(External (EF_builtin "__builtin_fmadd"
```

```
                   (mksignature
                     (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                     AST.Tfloat cc_default))
    (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
    cc_default)) ::
(___builtin_fmsub,
  Gfun(External (EF_builtin "__builtin_fmsub"
                   (mksignature
                     (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                     AST.Tfloat cc_default))
    (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
    cc_default)) ::
(___builtin_fnmadd,
  Gfun(External (EF_builtin "__builtin_fnmadd"
                   (mksignature
                     (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                     AST.Tfloat cc_default))
    (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
    cc_default)) ::
(___builtin_fnmsub,
  Gfun(External (EF_builtin "__builtin_fnmsub"
                   (mksignature
                     (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                     AST.Tfloat cc_default))
    (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
    cc_default)) ::
(___builtin_read16_reversed,
  Gfun(External (EF_builtin "__builtin_read16_reversed"
                   (mksignature (AST.Tint :: nil) AST.Tint16unsigned
                     cc_default)) (Tcons (tptr tushort) Tnil) tushort
    cc_default)) ::
(___builtin_read32_reversed,
  Gfun(External (EF_builtin "__builtin_read32_reversed"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons (tptr tuint) Tnil) tuint cc_default)) ::
(___builtin_write16_reversed,
  Gfun(External (EF_builtin "__builtin_write16_reversed"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                     cc_default)) (Tcons (tptr tushort) (Tcons tushort Tnil))
    tvoid cc_default)) ::
(___builtin_write32_reversed,
  Gfun(External (EF_builtin "__builtin_write32_reversed"
```

```
                    (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                      cc_default)) (Tcons (tptr tuint) (Tcons tuint Tnil))
     tvoid cc_default)) ::
 (___builtin_debug,
   Gfun(External (EF_external "__builtin_debug"
                    (mksignature (AST.Tint :: nil) AST.Tvoid
                      {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
     (Tcons tint Tnil) tvoid
     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
 (_malloc,
   Gfun(External EF_malloc (Tcons tuint Tnil) (tptr tvoid) cc_default)) ::
 (_free, Gfun(External EF_free (Tcons (tptr tvoid) Tnil) tvoid cc_default)) ::
 (_exit,
   Gfun(External (EF_external "exit"
                    (mksignature (AST.Tint :: nil) AST.Tvoid cc_default))
     (Tcons tint Tnil) tvoid cc_default)) ::
 (_newstack, Gfun(Internal f_newstack)) :: (_push, Gfun(Internal f_push)) ::
 (_pop, Gfun(Internal f_pop)) ::
 (_push_increasing, Gfun(Internal f_push_increasing)) ::
 (_pop_and_add, Gfun(Internal f_pop_and_add)) ::
 (_main, Gfun(Internal f_main)) :: nil).

Definition public_idents : list ident :=
(_main :: _pop_and_add :: _push_increasing :: _pop :: _push :: _newstack ::
 _exit :: _free :: _malloc :: ___builtin_debug ::
 ___builtin_write32_reversed :: ___builtin_write16_reversed ::
 ___builtin_read32_reversed :: ___builtin_read16_reversed ::
 ___builtin_fnmsub :: ___builtin_fnmadd :: ___builtin_fmsub ::
 ___builtin_fmadd :: ___builtin_fmin :: ___builtin_fmax ::
 ___builtin_ctzll :: ___builtin_ctzl :: ___builtin_ctz :: ___builtin_clzll ::
 ___builtin_clzl :: ___builtin_clz :: ___compcert_i64_umulh ::
 ___compcert_i64_smulh :: ___compcert_i64_sar :: ___compcert_i64_shr ::
 ___compcert_i64_shl :: ___compcert_i64_umod :: ___compcert_i64_smod ::
 ___compcert_i64_udiv :: ___compcert_i64_sdiv :: ___compcert_i64_utof ::
 ___compcert_i64_stof :: ___compcert_i64_utod :: ___compcert_i64_stod ::
 ___compcert_i64_dtou :: ___compcert_i64_dtos :: ___compcert_va_composite ::
 ___compcert_va_float64 :: ___compcert_va_int64 :: ___compcert_va_int32 ::
 ___builtin_va_end :: ___builtin_va_copy :: ___builtin_va_arg ::
 ___builtin_va_start :: ___builtin_membar :: ___builtin_annot_intval ::
 ___builtin_annot :: ___builtin_sel :: ___builtin_memcpy_aligned ::
 ___builtin_fsqrt :: ___builtin_fabs :: ___builtin_bswap16 ::
 ___builtin_bswap32 :: ___builtin_bswap :: ___builtin_bswap64 ::
 ___builtin_ais_annot :: nil).
```

```
Definition prog : Clight.program :=
  mkprogram composites global_definitions public_idents _main Logic.I.
```

Chapter 5

Library VC.strlib

```
From Coq Require Import String List ZArith.
From compcert Require Import Coqlib Integers Floats AST Ctypes Cop Clight Clightdefs.
Local Open Scope Z_scope.
Local Open Scope string_scope.

Module Info.
  Definition version := "3.7".
  Definition build_number := "".
  Definition build_tag := "".
  Definition arch := "x86".
  Definition model := "32sse2".
  Definition abi := "standard".
  Definition bitsize := 32.
  Definition big_endian := false.
  Definition source_file := "strlib.c".
  Definition normalized := true.
End Info.

Definition ___builtin_ais_annot : ident := 1%positive.
Definition ___builtin_annot : ident := 10%positive.
Definition ___builtin_annot_intval : ident := 11%positive.
Definition ___builtin_bswap : ident := 3%positive.
Definition ___builtin_bswap16 : ident := 5%positive.
Definition ___builtin_bswap32 : ident := 4%positive.
Definition ___builtin_bswap64 : ident := 2%positive.
Definition ___builtin_clz : ident := 36%positive.
Definition ___builtin_clzl : ident := 37%positive.
Definition ___builtin_clzll : ident := 38%positive.
Definition ___builtin_ctz : ident := 39%positive.
Definition ___builtin_ctzl : ident := 40%positive.
Definition ___builtin_ctzll : ident := 41%positive.
```

```
Definition ___builtin_debug : ident := 52%positive.
Definition ___builtin_fabs : ident := 6%positive.
Definition ___builtin_fmadd : ident := 44%positive.
Definition ___builtin_fmax : ident := 42%positive.
Definition ___builtin_fmin : ident := 43%positive.
Definition ___builtin_fmsub : ident := 45%positive.
Definition ___builtin_fnmadd : ident := 46%positive.
Definition ___builtin_fnmsub : ident := 47%positive.
Definition ___builtin_fsqrt : ident := 7%positive.
Definition ___builtin_membar : ident := 12%positive.
Definition ___builtin_memcpy_aligned : ident := 8%positive.
Definition ___builtin_read16_reversed : ident := 48%positive.
Definition ___builtin_read32_reversed : ident := 49%positive.
Definition ___builtin_sel : ident := 9%positive.
Definition ___builtin_va_arg : ident := 14%positive.
Definition ___builtin_va_copy : ident := 15%positive.
Definition ___builtin_va_end : ident := 16%positive.
Definition ___builtin_va_start : ident := 13%positive.
Definition ___builtin_write16_reversed : ident := 50%positive.
Definition ___builtin_write32_reversed : ident := 51%positive.
Definition ___compcert_i64_dtos : ident := 21%positive.
Definition ___compcert_i64_dtou : ident := 22%positive.
Definition ___compcert_i64_sar : ident := 33%positive.
Definition ___compcert_i64_sdiv : ident := 27%positive.
Definition ___compcert_i64_shl : ident := 31%positive.
Definition ___compcert_i64_shr : ident := 32%positive.
Definition ___compcert_i64_smod : ident := 29%positive.
Definition ___compcert_i64_smulh : ident := 34%positive.
Definition ___compcert_i64_stod : ident := 23%positive.
Definition ___compcert_i64_stof : ident := 25%positive.
Definition ___compcert_i64_udiv : ident := 28%positive.
Definition ___compcert_i64_umod : ident := 30%positive.
Definition ___compcert_i64_umulh : ident := 35%positive.
Definition ___compcert_i64_utod : ident := 24%positive.
Definition ___compcert_i64_utof : ident := 26%positive.
Definition ___compcert_va_composite : ident := 20%positive.
Definition ___compcert_va_float64 : ident := 19%positive.
Definition ___compcert_va_int32 : ident := 17%positive.
Definition ___compcert_va_int64 : ident := 18%positive.
Definition ___stringlit_1 : ident := 70%positive.
Definition _buf : ident := 69%positive.
Definition _c : ident := 56%positive.
```

```
Definition _d : ident := 57%positive.
Definition _d1 : ident := 66%positive.
Definition _d2 : ident := 67%positive.
Definition _dest : ident := 59%positive.
Definition _example_call_strcpy : ident := 71%positive.
Definition _i : ident := 54%positive.
Definition _j : ident := 62%positive.
Definition _main : ident := 72%positive.
Definition _src : ident := 60%positive.
Definition _str : ident := 53%positive.
Definition _str1 : ident := 64%positive.
Definition _str2 : ident := 65%positive.
Definition _strcat : ident := 63%positive.
Definition _strchr : ident := 58%positive.
Definition _strcmp : ident := 68%positive.
Definition _strcpy : ident := 61%positive.
Definition _strlen : ident := 55%positive.
Definition _t'1 : ident := 73%positive.
Definition _t'2 : ident := 74%positive.
Definition _t'3 : ident := 75%positive.

Definition v___stringlit_1 := {|
  gvar_info := (tarray tschar 6);
  gvar_init := (Init_int8 (Int.repr 72) :: Init_int8 (Int.repr 101) ::
                Init_int8 (Int.repr 108) :: Init_int8 (Int.repr 108) ::
                Init_int8 (Int.repr 111) :: Init_int8 (Int.repr 0) :: nil);
  gvar_readonly := true;
  gvar_volatile := false
|}.

Definition f_strlen := {|
  fn_return := tuint;
  fn_callconv := cc_default;
  fn_params := ((_str, (tptr tschar)) :: nil);
  fn_vars := nil;
  fn_temps := ((_i, tuint) :: (_t'1, tschar) :: nil);
  fn_body :=
(Ssequence
  (Sset _i (Econst_int (Int.repr 0) tint))
  (Sloop
    (Ssequence
      Sskip
      (Ssequence
        (Sset _t'1
```

```
          (Ederef
            (Ebinop Oadd (Etempvar _str (tptr tschar)) (Etempvar _i tuint)
              (tptr tschar)) tschar))
        (Sifthenelse (Ebinop Oeq (Etempvar _t'1 tschar)
                       (Econst_int (Int.repr 0) tint) tint)
          (Sreturn (Some (Etempvar _i tuint)))
          Sskip)))
    (Sset _i
      (Ebinop Oadd (Etempvar _i tuint) (Econst_int (Int.repr 1) tint) tuint))))
|}.

Definition f_strchr := {|
  fn_return := (tptr tschar);
  fn_callconv := cc_default;
  fn_params := ((_str, (tptr tschar)) :: (_c, tint) :: nil);
  fn_vars := nil;
  fn_temps := ((_i, tuint) :: (_d, tschar) :: (_t'1, tschar) :: nil);
  fn_body :=
(Ssequence
  (Sset _i (Econst_int (Int.repr 0) tint))
  (Sloop
    (Ssequence
      Sskip
      (Ssequence
        (Ssequence
          (Sset _t'1
            (Ederef
              (Ebinop Oadd (Etempvar _str (tptr tschar)) (Etempvar _i tuint)
                (tptr tschar)) tschar))
          (Sset _d (Ecast (Etempvar _t'1 tschar) tschar)))
        (Ssequence
          (Sifthenelse (Ebinop Oeq (Etempvar _d tschar) (Etempvar _c tint)
                         tint)
            (Sreturn (Some (Ebinop Oadd (Etempvar _str (tptr tschar))
                             (Etempvar _i tuint) (tptr tschar))))
            Sskip)
          (Sifthenelse (Ebinop Oeq (Etempvar _d tschar)
                         (Econst_int (Int.repr 0) tint) tint)
            (Sreturn (Some (Econst_int (Int.repr 0) tint)))
            Sskip))))
    (Sset _i
      (Ebinop Oadd (Etempvar _i tuint) (Econst_int (Int.repr 1) tint) tuint))))
|}.
```

```
Definition f_strcpy := {|
  fn_return := (tptr tschar);
  fn_callconv := cc_default;
  fn_params := ((_dest, (tptr tschar)) :: (_src, (tptr tschar)) :: nil);
  fn_vars := nil;
  fn_temps := ((_i, tuint) :: (_d, tschar) :: (_t'1, tschar) :: nil);
  fn_body :=
(Ssequence
  (Sset _i (Econst_int (Int.repr 0) tint))
  (Sloop
    (Ssequence
      Sskip
      (Ssequence
        (Ssequence
          (Sset _t'1
            (Ederef
              (Ebinop Oadd (Etempvar _src (tptr tschar)) (Etempvar _i tuint)
                (tptr tschar)) tschar))
          (Sset _d (Ecast (Etempvar _t'1 tschar) tschar)))
        (Ssequence
          (Sassign
            (Ederef
              (Ebinop Oadd (Etempvar _dest (tptr tschar)) (Etempvar _i tuint)
                (tptr tschar)) tschar) (Etempvar _d tschar))
          (Sifthenelse (Ebinop Oeq (Etempvar _d tschar)
                         (Econst_int (Int.repr 0) tint) tint)
            (Sreturn (Some (Etempvar _dest (tptr tschar))))
            Sskip))))
    (Sset _i
      (Ebinop Oadd (Etempvar _i tuint) (Econst_int (Int.repr 1) tint) tuint))))
|}.
Definition f_strcat := {|
  fn_return := (tptr tschar);
  fn_callconv := cc_default;
  fn_params := ((_dest, (tptr tschar)) :: (_src, (tptr tschar)) :: nil);
  fn_vars := nil;
  fn_temps := ((_i, tuint) :: (_j, tuint) :: (_d, tschar) ::
               (_t'2, tschar) :: (_t'1, tschar) :: nil);
  fn_body :=
(Ssequence
  (Ssequence
    (Sset _i (Econst_int (Int.repr 0) tint))
```

```
  (Sloop
    (Ssequence
      Sskip
      (Ssequence
        (Ssequence
          (Sset _t'2
            (Ederef
              (Ebinop Oadd (Etempvar _dest (tptr tschar))
                (Etempvar _i tuint) (tptr tschar)) tschar))
          (Sset _d (Ecast (Etempvar _t'2 tschar) tschar)))
        (Sifthenelse (Ebinop Oeq (Etempvar _d tschar)
                       (Econst_int (Int.repr 0) tint) tint)
          Sbreak
          Sskip)))
    (Sset _i
      (Ebinop Oadd (Etempvar _i tuint) (Econst_int (Int.repr 1) tint)
        tuint))))
(Ssequence
  (Sset _j (Econst_int (Int.repr 0) tint))
  (Sloop
    (Ssequence
      Sskip
      (Ssequence
        (Ssequence
          (Sset _t'1
            (Ederef
              (Ebinop Oadd (Etempvar _src (tptr tschar))
                (Etempvar _j tuint) (tptr tschar)) tschar))
          (Sset _d (Ecast (Etempvar _t'1 tschar) tschar)))
        (Ssequence
          (Sassign
            (Ederef
              (Ebinop Oadd (Etempvar _dest (tptr tschar))
                (Ebinop Oadd (Etempvar _i tuint) (Etempvar _j tuint) tuint)
                (tptr tschar)) tschar) (Etempvar _d tschar))
          (Sifthenelse (Ebinop Oeq (Etempvar _d tschar)
                         (Econst_int (Int.repr 0) tint) tint)
            (Sreturn (Some (Etempvar _dest (tptr tschar))))
            Sskip))))
    (Sset _j
      (Ebinop Oadd (Etempvar _j tuint) (Econst_int (Int.repr 1) tint)
        tuint)))))
```

```
|}.
Definition f_strcmp := {|
  fn_return := tint;
  fn_callconv := cc_default;
  fn_params := ((_str1, (tptr tschar)) :: (_str2, (tptr tschar)) :: nil);
  fn_vars := nil;
  fn_temps := ((_i, tuint) :: (_d1, tschar) :: (_d2, tschar) ::
               (_t'1, tint) :: (_t'3, tschar) :: (_t'2, tschar) :: nil);
  fn_body :=
(Ssequence
  (Sset _i (Econst_int (Int.repr 0) tint))
  (Sloop
    (Ssequence
      Sskip
      (Ssequence
        (Ssequence
          (Sset _t'3
            (Ederef
              (Ebinop Oadd (Etempvar _str1 (tptr tschar)) (Etempvar _i tuint)
                (tptr tschar)) tschar))
          (Sset _d1 (Ecast (Etempvar _t'3 tschar) tschar)))
        (Ssequence
          (Ssequence
            (Sset _t'2
              (Ederef
                (Ebinop Oadd (Etempvar _str2 (tptr tschar))
                  (Etempvar _i tuint) (tptr tschar)) tschar))
            (Sset _d2 (Ecast (Etempvar _t'2 tschar) tschar)))
          (Ssequence
            (Sifthenelse (Ebinop Oeq (Etempvar _d1 tschar)
                           (Econst_int (Int.repr 0) tint) tint)
              (Sset _t'1
                (Ecast
                  (Ebinop Oeq (Etempvar _d2 tschar)
                    (Econst_int (Int.repr 0) tint) tint) tbool))
              (Sset _t'1 (Econst_int (Int.repr 0) tint)))
            (Sifthenelse (Etempvar _t'1 tint)
              (Sreturn (Some (Econst_int (Int.repr 0) tint)))
              (Sifthenelse (Ebinop Olt (Etempvar _d1 tschar)
                             (Etempvar _d2 tschar) tint)
                (Sreturn (Some (Eunop Oneg (Econst_int (Int.repr 1) tint)
                                 tint)))
```

```
                    (Sifthenelse (Ebinop Ogt (Etempvar _d1 tschar)
                                    (Etempvar _d2 tschar) tint)
                      (Sreturn (Some (Econst_int (Int.repr 1) tint)))
                      Sskip)))))))
    (Sset _i
      (Ebinop Oadd (Etempvar _i tuint) (Econst_int (Int.repr 1) tint) tuint))))
|}.
Definition f_example_call_strcpy := {|
  fn_return := tint;
  fn_callconv := cc_default;
  fn_params := nil;
  fn_vars := ((_buf, (tarray tschar 10)) :: nil);
  fn_temps := ((_t'1, tschar) :: nil);
  fn_body :=
(Ssequence
  (Scall None
    (Evar _strcpy (Tfunction (Tcons (tptr tschar) (Tcons (tptr tschar) Tnil))
                    (tptr tschar) cc_default))
    ((Evar _buf (tarray tschar 10)) ::
     (Evar ___stringlit_1 (tarray tschar 6)) :: nil))
  (Ssequence
    (Sset _t'1
      (Ederef
        (Ebinop Oadd (Evar _buf (tarray tschar 10))
          (Econst_int (Int.repr 0) tint) (tptr tschar)) tschar))
    (Sreturn (Some (Etempvar _t'1 tschar)))))
|}.
Definition composites : list composite_definition :=
nil.
Definition global_definitions : list (ident × globdef fundef type) :=
((___stringlit_1, Gvar v___stringlit_1) ::
 (___builtin_ais_annot,
   Gfun(External (EF_builtin "__builtin_ais_annot"
                   (mksignature (AST.Tint :: nil) AST.Tvoid
                     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
     (Tcons (tptr tschar) Tnil) tvoid
     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
 (___builtin_bswap64,
   Gfun(External (EF_builtin "__builtin_bswap64"
                   (mksignature (AST.Tlong :: nil) AST.Tlong cc_default))
     (Tcons tulong Tnil) tulong cc_default)) ::
 (___builtin_bswap,
```

```
  Gfun(External (EF_builtin "__builtin_bswap"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
     (Tcons tuint Tnil) tuint cc_default)) ::
(___builtin_bswap32,
  Gfun(External (EF_builtin "__builtin_bswap32"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
     (Tcons tuint Tnil) tuint cc_default)) ::
(___builtin_bswap16,
  Gfun(External (EF_builtin "__builtin_bswap16"
                   (mksignature (AST.Tint :: nil) AST.Tint16unsigned
                     cc_default)) (Tcons tushort Tnil) tushort cc_default)) ::
(___builtin_fabs,
  Gfun(External (EF_builtin "__builtin_fabs"
                   (mksignature (AST.Tfloat :: nil) AST.Tfloat cc_default))
     (Tcons tdouble Tnil) tdouble cc_default)) ::
(___builtin_fsqrt,
  Gfun(External (EF_builtin "__builtin_fsqrt"
                   (mksignature (AST.Tfloat :: nil) AST.Tfloat cc_default))
     (Tcons tdouble Tnil) tdouble cc_default)) ::
(___builtin_memcpy_aligned,
  Gfun(External (EF_builtin "__builtin_memcpy_aligned"
                   (mksignature
                     (AST.Tint :: AST.Tint :: AST.Tint :: AST.Tint :: nil)
                     AST.Tvoid cc_default))
     (Tcons (tptr tvoid)
       (Tcons (tptr tvoid) (Tcons tuint (Tcons tuint Tnil)))) tvoid
     cc_default)) ::
(___builtin_sel,
  Gfun(External (EF_builtin "__builtin_sel"
                   (mksignature (AST.Tint :: nil) AST.Tvoid
                     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
     (Tcons tbool Tnil) tvoid
     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
(___builtin_annot,
  Gfun(External (EF_builtin "__builtin_annot"
                   (mksignature (AST.Tint :: nil) AST.Tvoid
                     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
     (Tcons (tptr tschar) Tnil) tvoid
     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
(___builtin_annot_intval,
  Gfun(External (EF_builtin "__builtin_annot_intval"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tint
```

```
                          cc_default)) (Tcons (tptr tschar) (Tcons tint Tnil))
     tint cc_default)) ::
 (___builtin_membar,
   Gfun(External (EF_builtin "__builtin_membar"
                        (mksignature nil AST.Tvoid cc_default)) Tnil tvoid
     cc_default)) ::
 (___builtin_va_start,
   Gfun(External (EF_builtin "__builtin_va_start"
                        (mksignature (AST.Tint :: nil) AST.Tvoid cc_default))
     (Tcons (tptr tvoid) Tnil) tvoid cc_default)) ::
 (___builtin_va_arg,
   Gfun(External (EF_builtin "__builtin_va_arg"
                        (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                          cc_default)) (Tcons (tptr tvoid) (Tcons tuint Tnil))
     tvoid cc_default)) ::
 (___builtin_va_copy,
   Gfun(External (EF_builtin "__builtin_va_copy"
                        (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                          cc_default))
     (Tcons (tptr tvoid) (Tcons (tptr tvoid) Tnil)) tvoid cc_default)) ::
 (___builtin_va_end,
   Gfun(External (EF_builtin "__builtin_va_end"
                        (mksignature (AST.Tint :: nil) AST.Tvoid cc_default))
     (Tcons (tptr tvoid) Tnil) tvoid cc_default)) ::
 (___compcert_va_int32,
   Gfun(External (EF_external "__compcert_va_int32"
                        (mksignature (AST.Tint :: nil) AST.Tint cc_default))
     (Tcons (tptr tvoid) Tnil) tuint cc_default)) ::
 (___compcert_va_int64,
   Gfun(External (EF_external "__compcert_va_int64"
                        (mksignature (AST.Tint :: nil) AST.Tlong cc_default))
     (Tcons (tptr tvoid) Tnil) tulong cc_default)) ::
 (___compcert_va_float64,
   Gfun(External (EF_external "__compcert_va_float64"
                        (mksignature (AST.Tint :: nil) AST.Tfloat cc_default))
     (Tcons (tptr tvoid) Tnil) tdouble cc_default)) ::
 (___compcert_va_composite,
   Gfun(External (EF_external "__compcert_va_composite"
                        (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tint
                          cc_default)) (Tcons (tptr tvoid) (Tcons tuint Tnil))
     (tptr tvoid) cc_default)) ::
 (___compcert_i64_dtos,
```

```
  Gfun(External (EF_runtime "__compcert_i64_dtos"
                   (mksignature (AST.Tfloat :: nil) AST.Tlong cc_default))
    (Tcons tdouble Tnil) tlong cc_default)) ::
(___compcert_i64_dtou,
  Gfun(External (EF_runtime "__compcert_i64_dtou"
                   (mksignature (AST.Tfloat :: nil) AST.Tlong cc_default))
    (Tcons tdouble Tnil) tulong cc_default)) ::
(___compcert_i64_stod,
  Gfun(External (EF_runtime "__compcert_i64_stod"
                   (mksignature (AST.Tlong :: nil) AST.Tfloat cc_default))
    (Tcons tlong Tnil) tdouble cc_default)) ::
(___compcert_i64_utod,
  Gfun(External (EF_runtime "__compcert_i64_utod"
                   (mksignature (AST.Tlong :: nil) AST.Tfloat cc_default))
    (Tcons tulong Tnil) tdouble cc_default)) ::
(___compcert_i64_stof,
  Gfun(External (EF_runtime "__compcert_i64_stof"
                   (mksignature (AST.Tlong :: nil) AST.Tsingle cc_default))
    (Tcons tlong Tnil) tfloat cc_default)) ::
(___compcert_i64_utof,
  Gfun(External (EF_runtime "__compcert_i64_utof"
                   (mksignature (AST.Tlong :: nil) AST.Tsingle cc_default))
    (Tcons tulong Tnil) tfloat cc_default)) ::
(___compcert_i64_sdiv,
  Gfun(External (EF_runtime "__compcert_i64_sdiv"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tlong Tnil)) tlong
    cc_default)) ::
(___compcert_i64_udiv,
  Gfun(External (EF_runtime "__compcert_i64_udiv"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tulong (Tcons tulong Tnil)) tulong
    cc_default)) ::
(___compcert_i64_smod,
  Gfun(External (EF_runtime "__compcert_i64_smod"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tlong Tnil)) tlong
    cc_default)) ::
(___compcert_i64_umod,
  Gfun(External (EF_runtime "__compcert_i64_umod"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tulong (Tcons tulong Tnil)) tulong
```

```
    cc_default)) ::
(___compcert_i64_shl,
  Gfun(External (EF_runtime "__compcert_i64_shl"
                   (mksignature (AST.Tlong :: AST.Tint :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tint Tnil)) tlong
    cc_default)) ::
(___compcert_i64_shr,
  Gfun(External (EF_runtime "__compcert_i64_shr"
                   (mksignature (AST.Tlong :: AST.Tint :: nil) AST.Tlong
                     cc_default)) (Tcons tulong (Tcons tint Tnil)) tulong
    cc_default)) ::
(___compcert_i64_sar,
  Gfun(External (EF_runtime "__compcert_i64_sar"
                   (mksignature (AST.Tlong :: AST.Tint :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tint Tnil)) tlong
    cc_default)) ::
(___compcert_i64_smulh,
  Gfun(External (EF_runtime "__compcert_i64_smulh"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tlong (Tcons tlong Tnil)) tlong
    cc_default)) ::
(___compcert_i64_umulh,
  Gfun(External (EF_runtime "__compcert_i64_umulh"
                   (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                     cc_default)) (Tcons tulong (Tcons tulong Tnil)) tulong
    cc_default)) ::
(___builtin_clz,
  Gfun(External (EF_builtin "__builtin_clz"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_clzl,
  Gfun(External (EF_builtin "__builtin_clzl"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_clzll,
  Gfun(External (EF_builtin "__builtin_clzll"
                   (mksignature (AST.Tlong :: nil) AST.Tint cc_default))
    (Tcons tulong Tnil) tint cc_default)) ::
(___builtin_ctz,
  Gfun(External (EF_builtin "__builtin_ctz"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
```

```
(___builtin_ctzl,
  Gfun(External (EF_builtin "__builtin_ctzl"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_ctzll,
  Gfun(External (EF_builtin "__builtin_ctzll"
                   (mksignature (AST.Tlong :: nil) AST.Tint cc_default))
    (Tcons tulong Tnil) tint cc_default)) ::
(___builtin_fmax,
  Gfun(External (EF_builtin "__builtin_fmax"
                   (mksignature (AST.Tfloat :: AST.Tfloat :: nil) AST.Tfloat
                     cc_default)) (Tcons tdouble (Tcons tdouble Tnil))
    tdouble cc_default)) ::
(___builtin_fmin,
  Gfun(External (EF_builtin "__builtin_fmin"
                   (mksignature (AST.Tfloat :: AST.Tfloat :: nil) AST.Tfloat
                     cc_default)) (Tcons tdouble (Tcons tdouble Tnil))
    tdouble cc_default)) ::
(___builtin_fmadd,
  Gfun(External (EF_builtin "__builtin_fmadd"
                   (mksignature
                     (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                     AST.Tfloat cc_default))
    (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
    cc_default)) ::
(___builtin_fmsub,
  Gfun(External (EF_builtin "__builtin_fmsub"
                   (mksignature
                     (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                     AST.Tfloat cc_default))
    (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
    cc_default)) ::
(___builtin_fnmadd,
  Gfun(External (EF_builtin "__builtin_fnmadd"
                   (mksignature
                     (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                     AST.Tfloat cc_default))
    (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
    cc_default)) ::
(___builtin_fnmsub,
  Gfun(External (EF_builtin "__builtin_fnmsub"
                   (mksignature
```

```
                          (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                          AST.Tfloat cc_default))
      (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
      cc_default)) ::
  (___builtin_read16_reversed,
    Gfun(External (EF_builtin "__builtin_read16_reversed"
                        (mksignature (AST.Tint :: nil) AST.Tint16unsigned
                          cc_default)) (Tcons (tptr tushort) Tnil) tushort
      cc_default)) ::
  (___builtin_read32_reversed,
    Gfun(External (EF_builtin "__builtin_read32_reversed"
                        (mksignature (AST.Tint :: nil) AST.Tint cc_default))
      (Tcons (tptr tuint) Tnil) tuint cc_default)) ::
  (___builtin_write16_reversed,
    Gfun(External (EF_builtin "__builtin_write16_reversed"
                        (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                          cc_default)) (Tcons (tptr tushort) (Tcons tushort Tnil))
      tvoid cc_default)) ::
  (___builtin_write32_reversed,
    Gfun(External (EF_builtin "__builtin_write32_reversed"
                        (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                          cc_default)) (Tcons (tptr tuint) (Tcons tuint Tnil))
      tvoid cc_default)) ::
  (___builtin_debug,
    Gfun(External (EF_external "__builtin_debug"
                        (mksignature (AST.Tint :: nil) AST.Tvoid
                          {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
      (Tcons tint Tnil) tvoid
      {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
  (_strlen, Gfun(Internal f_strlen)) :: (_strchr, Gfun(Internal f_strchr)) ::
  (_strcpy, Gfun(Internal f_strcpy)) :: (_strcat, Gfun(Internal f_strcat)) ::
  (_strcmp, Gfun(Internal f_strcmp)) ::
  (_example_call_strcpy, Gfun(Internal f_example_call_strcpy)) :: nil).

Definition public_idents : list ident :=
(_example_call_strcpy :: _strcmp :: _strcat :: _strcpy :: _strchr ::
 _strlen :: ___builtin_debug :: ___builtin_write32_reversed ::
 ___builtin_write16_reversed :: ___builtin_read32_reversed ::
 ___builtin_read16_reversed :: ___builtin_fnmsub :: ___builtin_fnmadd ::
 ___builtin_fmsub :: ___builtin_fmadd :: ___builtin_fmin ::
 ___builtin_fmax :: ___builtin_ctzll :: ___builtin_ctzl :: ___builtin_ctz ::
 ___builtin_clzll :: ___builtin_clzl :: ___builtin_clz ::
 ___compcert_i64_umulh :: ___compcert_i64_smulh :: ___compcert_i64_sar ::
```

```
  ___compcert_i64_shr :: ___compcert_i64_shl :: ___compcert_i64_umod ::
  ___compcert_i64_smod :: ___compcert_i64_udiv :: ___compcert_i64_sdiv ::
  ___compcert_i64_utof :: ___compcert_i64_stof :: ___compcert_i64_utod ::
  ___compcert_i64_stod :: ___compcert_i64_dtou :: ___compcert_i64_dtos ::
  ___compcert_va_composite :: ___compcert_va_float64 ::
  ___compcert_va_int64 :: ___compcert_va_int32 :: ___builtin_va_end ::
  ___builtin_va_copy :: ___builtin_va_arg :: ___builtin_va_start ::
  ___builtin_membar :: ___builtin_annot_intval :: ___builtin_annot ::
  ___builtin_sel :: ___builtin_memcpy_aligned :: ___builtin_fsqrt ::
  ___builtin_fabs :: ___builtin_bswap16 :: ___builtin_bswap32 ::
  ___builtin_bswap :: ___builtin_bswap64 :: ___builtin_ais_annot :: nil).

Definition prog : Clight.program :=
  mkprogram composites global_definitions public_idents _main Logic.I.
```

Chapter 6

Library VC.hash

```
From Coq Require Import String List ZArith.
From compcert Require Import Coqlib Integers Floats AST Ctypes Cop Clight Clightdefs.
Local Open Scope Z_scope.
Local Open Scope string_scope.

Module Info.
  Definition version := "3.7".
  Definition build_number := "".
  Definition build_tag := "".
  Definition arch := "x86".
  Definition model := "32sse2".
  Definition abi := "standard".
  Definition bitsize := 32.
  Definition big_endian := false.
  Definition source_file := "hash.c".
  Definition normalized := true.
End Info.

Definition ___builtin_ais_annot : ident := 7%positive.
Definition ___builtin_annot : ident := 16%positive.
Definition ___builtin_annot_intval : ident := 17%positive.
Definition ___builtin_bswap : ident := 9%positive.
Definition ___builtin_bswap16 : ident := 11%positive.
Definition ___builtin_bswap32 : ident := 10%positive.
Definition ___builtin_bswap64 : ident := 8%positive.
Definition ___builtin_clz : ident := 42%positive.
Definition ___builtin_clzl : ident := 43%positive.
Definition ___builtin_clzll : ident := 44%positive.
Definition ___builtin_ctz : ident := 45%positive.
Definition ___builtin_ctzl : ident := 46%positive.
Definition ___builtin_ctzll : ident := 47%positive.
```

```
Definition ___builtin_debug : ident := 58%positive.
Definition ___builtin_fabs : ident := 12%positive.
Definition ___builtin_fmadd : ident := 50%positive.
Definition ___builtin_fmax : ident := 48%positive.
Definition ___builtin_fmin : ident := 49%positive.
Definition ___builtin_fmsub : ident := 51%positive.
Definition ___builtin_fnmadd : ident := 52%positive.
Definition ___builtin_fnmsub : ident := 53%positive.
Definition ___builtin_fsqrt : ident := 13%positive.
Definition ___builtin_membar : ident := 18%positive.
Definition ___builtin_memcpy_aligned : ident := 14%positive.
Definition ___builtin_read16_reversed : ident := 54%positive.
Definition ___builtin_read32_reversed : ident := 55%positive.
Definition ___builtin_sel : ident := 15%positive.
Definition ___builtin_va_arg : ident := 20%positive.
Definition ___builtin_va_copy : ident := 21%positive.
Definition ___builtin_va_end : ident := 22%positive.
Definition ___builtin_va_start : ident := 19%positive.
Definition ___builtin_write16_reversed : ident := 56%positive.
Definition ___builtin_write32_reversed : ident := 57%positive.
Definition ___compcert_i64_dtos : ident := 27%positive.
Definition ___compcert_i64_dtou : ident := 28%positive.
Definition ___compcert_i64_sar : ident := 39%positive.
Definition ___compcert_i64_sdiv : ident := 33%positive.
Definition ___compcert_i64_shl : ident := 37%positive.
Definition ___compcert_i64_shr : ident := 38%positive.
Definition ___compcert_i64_smod : ident := 35%positive.
Definition ___compcert_i64_smulh : ident := 40%positive.
Definition ___compcert_i64_stod : ident := 29%positive.
Definition ___compcert_i64_stof : ident := 31%positive.
Definition ___compcert_i64_udiv : ident := 34%positive.
Definition ___compcert_i64_umod : ident := 36%positive.
Definition ___compcert_i64_umulh : ident := 41%positive.
Definition ___compcert_i64_utod : ident := 30%positive.
Definition ___compcert_i64_utof : ident := 32%positive.
Definition ___compcert_va_composite : ident := 26%positive.
Definition ___compcert_va_float64 : ident := 25%positive.
Definition ___compcert_va_int32 : ident := 23%positive.
Definition ___compcert_va_int64 : ident := 24%positive.
Definition _b : ident := 89%positive.
Definition _buckets : ident := 5%positive.
Definition _c : ident := 67%positive.
```

```
Definition _cell : ident := 3%positive.
Definition _copy_string : ident := 70%positive.
Definition _count : ident := 2%positive.
Definition _exit : ident := 60%positive.
Definition _get : ident := 76%positive.
Definition _h : ident := 74%positive.
Definition _hash : ident := 68%positive.
Definition _hashtable : ident := 6%positive.
Definition _i : ident := 66%positive.
Definition _incr : ident := 80%positive.
Definition _incr_list : ident := 79%positive.
Definition _incrx : ident := 81%positive.
Definition _key : ident := 1%positive.
Definition _main : ident := 82%positive.
Definition _malloc : ident := 59%positive.
Definition _n : ident := 65%positive.
Definition _new_cell : ident := 72%positive.
Definition _new_table : ident := 71%positive.
Definition _next : ident := 4%positive.
Definition _p : ident := 69%positive.
Definition _r : ident := 78%positive.
Definition _r0 : ident := 77%positive.
Definition _s : ident := 64%positive.
Definition _strcmp : ident := 63%positive.
Definition _strcpy : ident := 62%positive.
Definition _strlen : ident := 61%positive.
Definition _table : ident := 73%positive.
Definition _t'1 : ident := 83%positive.
Definition _t'2 : ident := 84%positive.
Definition _t'3 : ident := 85%positive.
Definition _t'4 : ident := 86%positive.
Definition _t'5 : ident := 87%positive.
Definition _t'6 : ident := 88%positive.

Definition f_hash := {|
  fn_return := tuint;
  fn_callconv := cc_default;
  fn_params := ((_s, (tptr tschar)) :: nil);
  fn_vars := nil;
  fn_temps := ((_n, tuint) :: (_i, tuint) :: (_c, tint) :: nil);
  fn_body :=
(Ssequence
  (Sset _n (Econst_int (Int.repr 0) tint))
```

```
  (Ssequence
    (Sset _i (Econst_int (Int.repr 0) tint))
    (Ssequence
      (Sset _c
        (Ederef
          (Ebinop Oadd (Etempvar _s (tptr tschar)) (Etempvar _i tuint)
            (tptr tschar)) tschar))
      (Ssequence
        (Swhile
          (Etempvar _c tint)
          (Ssequence
            (Sset _n
              (Ebinop Oadd
                (Ebinop Omul (Etempvar _n tuint)
                  (Econst_int (Int.repr 65599) tuint) tuint)
                (Ecast (Etempvar _c tint) tuint) tuint))
            (Ssequence
              (Sset _i
                (Ebinop Oadd (Etempvar _i tuint)
                  (Econst_int (Int.repr 1) tint) tuint))
              (Sset _c
                (Ederef
                  (Ebinop Oadd (Etempvar _s (tptr tschar))
                    (Etempvar _i tuint) (tptr tschar)) tschar)))))
        (Sreturn (Some (Etempvar _n tuint))))))
|}.

Definition f_copy_string := {|
  fn_return := (tptr tschar);
  fn_callconv := cc_default;
  fn_params := ((_s, (tptr tschar)) :: nil);
  fn_vars := nil;
  fn_temps := ((_i, tint) :: (_n, tint) :: (_p, (tptr tschar)) ::
               (_t'2, (tptr tvoid)) :: (_t'1, tuint) :: nil);
  fn_body :=
(Ssequence
  (Ssequence
    (Scall (Some _t'1)
      (Evar _strlen (Tfunction (Tcons (tptr tschar) Tnil) tuint cc_default))
      ((Etempvar _s (tptr tschar)) :: nil))
    (Sset _n
      (Ebinop Oadd (Etempvar _t'1 tuint) (Econst_int (Int.repr 1) tint)
        tuint)))
```

```
  (Ssequence
    (Ssequence
      (Scall (Some _t'2)
        (Evar _malloc (Tfunction (Tcons tuint Tnil) (tptr tvoid) cc_default))
        ((Etempvar _n tint) :: nil))
      (Sset _p (Etempvar _t'2 (tptr tvoid))))
    (Ssequence
      (Sifthenelse (Eunop Onotbool (Etempvar _p (tptr tschar)) tint)
        (Scall None
          (Evar _exit (Tfunction (Tcons tint Tnil) tvoid cc_default))
          ((Econst_int (Int.repr 1) tint) :: nil))
        Sskip)
      (Ssequence
        (Scall None
          (Evar _strcpy (Tfunction
                          (Tcons (tptr tschar) (Tcons (tptr tschar) Tnil))
                          (tptr tschar) cc_default))
          ((Etempvar _p (tptr tschar)) :: (Etempvar _s (tptr tschar)) :: nil))
        (Sreturn (Some (Etempvar _p (tptr tschar)))))))
|}.

Definition f_new_table := {|
  fn_return := (tptr (Tstruct _hashtable noattr));
  fn_callconv := cc_default;
  fn_params := nil;
  fn_vars := nil;
  fn_temps := ((_i, tint) :: (_p, (tptr (Tstruct _hashtable noattr))) ::
               (_t'1, (tptr tvoid)) :: nil);
  fn_body :=
(Ssequence
  (Ssequence
    (Scall (Some _t'1)
      (Evar _malloc (Tfunction (Tcons tuint Tnil) (tptr tvoid) cc_default))
      ((Esizeof (Tstruct _hashtable noattr) tuint) :: nil))
    (Sset _p
      (Ecast (Etempvar _t'1 (tptr tvoid)) (tptr (Tstruct _hashtable noattr)))))
  (Ssequence
    (Sifthenelse (Eunop Onotbool
                   (Etempvar _p (tptr (Tstruct _hashtable noattr))) tint)
      (Scall None (Evar _exit (Tfunction (Tcons tint Tnil) tvoid cc_default))
        ((Econst_int (Int.repr 1) tint) :: nil))
      Sskip)
    (Ssequence
```

```
      (Ssequence
        (Sset _i (Econst_int (Int.repr 0) tint))
        (Sloop
          (Ssequence
            (Sifthenelse (Ebinop Olt (Etempvar _i tint)
                               (Econst_int (Int.repr 109) tint) tint)
              Sskip
              Sbreak)
            (Sassign
              (Ederef
                (Ebinop Oadd
                  (Efield
                    (Ederef (Etempvar _p (tptr (Tstruct _hashtable noattr)))
                      (Tstruct _hashtable noattr)) _buckets
                    (tarray (tptr (Tstruct _cell noattr)) 109))
                  (Etempvar _i tint) (tptr (tptr (Tstruct _cell noattr))))
                (tptr (Tstruct _cell noattr)))
              (Ecast (Econst_int (Int.repr 0) tint) (tptr tvoid))))
          (Sset _i
            (Ebinop Oadd (Etempvar _i tint) (Econst_int (Int.repr 1) tint)
              tint))))
      (Sreturn (Some (Etempvar _p (tptr (Tstruct _hashtable noattr)))))))
|}.
Definition f_new_cell := {|
  fn_return := (tptr (Tstruct _cell noattr));
  fn_callconv := cc_default;
  fn_params := ((_key, (tptr tschar)) :: (_count, tint) ::
                (_next, (tptr (Tstruct _cell noattr))) :: nil);
  fn_vars := nil;
  fn_temps := ((_p, (tptr (Tstruct _cell noattr))) ::
               (_t'2, (tptr tschar)) :: (_t'1, (tptr tvoid)) :: nil);
  fn_body :=
(Ssequence
  (Ssequence
    (Scall (Some _t'1)
      (Evar _malloc (Tfunction (Tcons tuint Tnil) (tptr tvoid) cc_default))
      ((Esizeof (Tstruct _cell noattr) tuint) :: nil))
    (Sset _p
      (Ecast (Etempvar _t'1 (tptr tvoid)) (tptr (Tstruct _cell noattr)))))
  (Ssequence
    (Sifthenelse (Eunop Onotbool (Etempvar _p (tptr (Tstruct _cell noattr)))
                   tint)
```

```
        (Scall None (Evar _exit (Tfunction (Tcons tint Tnil) tvoid cc_default))
          ((Econst_int (Int.repr 1) tint) :: nil))
        Sskip)
      (Ssequence
        (Ssequence
          (Scall (Some _t'2)
            (Evar _copy_string (Tfunction (Tcons (tptr tschar) Tnil)
                                 (tptr tschar) cc_default))
            ((Etempvar _key (tptr tschar)) :: nil))
          (Sassign
            (Efield
              (Ederef (Etempvar _p (tptr (Tstruct _cell noattr)))
                (Tstruct _cell noattr)) _key (tptr tschar))
            (Etempvar _t'2 (tptr tschar))))
        (Ssequence
          (Sassign
            (Efield
              (Ederef (Etempvar _p (tptr (Tstruct _cell noattr)))
                (Tstruct _cell noattr)) _count tuint) (Etempvar _count tint))
          (Ssequence
            (Sassign
              (Efield
                (Ederef (Etempvar _p (tptr (Tstruct _cell noattr)))
                  (Tstruct _cell noattr)) _next (tptr (Tstruct _cell noattr)))
              (Etempvar _next (tptr (Tstruct _cell noattr))))
            (Sreturn (Some (Etempvar _p (tptr (Tstruct _cell noattr)))))))))))
|}.
Definition f_get := {|
  fn_return := tuint;
  fn_callconv := cc_default;
  fn_params := ((_table, (tptr (Tstruct _hashtable noattr))) ::
                (_s, (tptr tschar)) :: nil);
  fn_vars := nil;
  fn_temps := ((_h, tuint) :: (_b, tuint) ::
               (_p, (tptr (Tstruct _cell noattr))) :: (_t'2, tint) ::
               (_t'1, tuint) :: (_t'4, (tptr tschar)) :: (_t'3, tuint) ::
               nil);
  fn_body :=
(Ssequence
  (Ssequence
    (Scall (Some _t'1)
      (Evar _hash (Tfunction (Tcons (tptr tschar) Tnil) tuint cc_default))
```

```
    ((Etempvar _s (tptr tschar)) :: nil))
  (Sset _h (Etempvar _t'1 tuint)))
(Ssequence
  (Sset _b
    (Ebinop Omod (Etempvar _h tuint) (Econst_int (Int.repr 109) tint)
      tuint))
  (Ssequence
    (Sset _p
      (Ederef
        (Ebinop Oadd
          (Efield
            (Ederef (Etempvar _table (tptr (Tstruct _hashtable noattr)))
              (Tstruct _hashtable noattr)) _buckets
            (tarray (tptr (Tstruct _cell noattr)) 109)) (Etempvar _b tuint)
          (tptr (tptr (Tstruct _cell noattr))))
        (tptr (Tstruct _cell noattr))))
    (Ssequence
      (Swhile
        (Etempvar _p (tptr (Tstruct _cell noattr)))
        (Ssequence
          (Ssequence
            (Ssequence
              (Sset _t'4
                (Efield
                  (Ederef (Etempvar _p (tptr (Tstruct _cell noattr)))
                    (Tstruct _cell noattr)) _key (tptr tschar)))
              (Scall (Some _t'2)
                (Evar _strcmp (Tfunction
                                (Tcons (tptr tschar)
                                  (Tcons (tptr tschar) Tnil)) tint
                                cc_default))
                ((Etempvar _t'4 (tptr tschar)) ::
                 (Etempvar _s (tptr tschar)) :: nil)))
            (Sifthenelse (Ebinop Oeq (Etempvar _t'2 tint)
                           (Econst_int (Int.repr 0) tint) tint)
              (Ssequence
                (Sset _t'3
                  (Efield
                    (Ederef (Etempvar _p (tptr (Tstruct _cell noattr)))
                      (Tstruct _cell noattr)) _count tuint))
                (Sreturn (Some (Etempvar _t'3 tuint))))
              Sskip))
```

```
            (Sset _p
              (Efield
                (Ederef (Etempvar _p (tptr (Tstruct _cell noattr)))
                  (Tstruct _cell noattr)) _next
                (tptr (Tstruct _cell noattr))))))
        (Sreturn (Some (Econst_int (Int.repr 0) tint)))))))
|}.

Definition f_incr_list := {|
  fn_return := tvoid;
  fn_callconv := cc_default;
  fn_params := ((_r0, (tptr (tptr (Tstruct _cell noattr)))) ::
                (_s, (tptr tschar)) :: nil);
  fn_vars := nil;
  fn_temps := ((_p, (tptr (Tstruct _cell noattr))) ::
               (_r, (tptr (tptr (Tstruct _cell noattr)))) :: (_t'2, tint) ::
               (_t'1, (tptr (Tstruct _cell noattr))) ::
               (_t'4, (tptr tschar)) :: (_t'3, tuint) :: nil);
  fn_body :=
(Ssequence
  (Sset _r (Etempvar _r0 (tptr (tptr (Tstruct _cell noattr)))))
  (Sloop
    (Ssequence
      Sskip
      (Ssequence
        (Sset _p
          (Ederef (Etempvar _r (tptr (tptr (Tstruct _cell noattr))))
            (tptr (Tstruct _cell noattr))))
        (Ssequence
          (Sifthenelse (Eunop Onotbool
                         (Etempvar _p (tptr (Tstruct _cell noattr))) tint)
            (Ssequence
              (Ssequence
                (Scall (Some _t'1)
                  (Evar _new_cell (Tfunction
                                    (Tcons (tptr tschar)
                                      (Tcons tint
                                        (Tcons (tptr (Tstruct _cell noattr))
                                          Tnil)))
                                    (tptr (Tstruct _cell noattr)) cc_default))
                  ((Etempvar _s (tptr tschar)) ::
                   (Econst_int (Int.repr 1) tint) ::
                   (Ecast (Econst_int (Int.repr 0) tint) (tptr tvoid)) ::
```

```
                    nil))
                 (Sassign
                    (Ederef (Etempvar _r (tptr (tptr (Tstruct _cell noattr))))
                       (tptr (Tstruct _cell noattr)))
                    (Etempvar _t'1 (tptr (Tstruct _cell noattr)))))
              (Sreturn None))
           Sskip)
        (Ssequence
           (Ssequence
              (Sset _t'4
                 (Efield
                    (Ederef (Etempvar _p (tptr (Tstruct _cell noattr)))
                       (Tstruct _cell noattr)) _key (tptr tschar)))
              (Scall (Some _t'2)
                 (Evar _strcmp (Tfunction
                                     (Tcons (tptr tschar)
                                        (Tcons (tptr tschar) Tnil)) tint
                                     cc_default))
                 ((Etempvar _t'4 (tptr tschar)) ::
                  (Etempvar _s (tptr tschar)) :: nil)))
           (Sifthenelse (Ebinop Oeq (Etempvar _t'2 tint)
                              (Econst_int (Int.repr 0) tint) tint)
              (Ssequence
                 (Ssequence
                    (Sset _t'3
                       (Efield
                          (Ederef (Etempvar _p (tptr (Tstruct _cell noattr)))
                             (Tstruct _cell noattr)) _count tuint))
                    (Sassign
                       (Efield
                          (Ederef (Etempvar _p (tptr (Tstruct _cell noattr)))
                             (Tstruct _cell noattr)) _count tuint)
                       (Ebinop Oadd (Etempvar _t'3 tuint)
                          (Econst_int (Int.repr 1) tint) tuint)))
                 (Sreturn None))
              Sskip)))))
(Sset _r
   (Eaddrof
      (Efield
         (Ederef (Etempvar _p (tptr (Tstruct _cell noattr)))
            (Tstruct _cell noattr)) _next (tptr (Tstruct _cell noattr)))
      (tptr (tptr (Tstruct _cell noattr))))))
```

```
|}.
Definition f_incr := {|
  fn_return := tvoid;
  fn_callconv := cc_default;
  fn_params := ((_table, (tptr (Tstruct _hashtable noattr))) ::
                (_s, (tptr tschar)) :: nil);
  fn_vars := nil;
  fn_temps := ((_h, tuint) :: (_b, tuint) :: (_t'1, tuint) :: nil);
  fn_body :=
(Ssequence
  (Ssequence
    (Scall (Some _t'1)
      (Evar _hash (Tfunction (Tcons (tptr tschar) Tnil) tuint cc_default))
      ((Etempvar _s (tptr tschar)) :: nil))
    (Sset _h (Etempvar _t'1 tuint)))
  (Ssequence
    (Sset _b
      (Ebinop Omod (Etempvar _h tuint) (Econst_int (Int.repr 109) tint)
        tuint))
    (Scall None
      (Evar _incr_list (Tfunction
                         (Tcons (tptr (tptr (Tstruct _cell noattr)))
                           (Tcons (tptr tschar) Tnil)) tvoid cc_default))
      ((Ebinop Oadd
          (Efield
            (Ederef (Etempvar _table (tptr (Tstruct _hashtable noattr)))
              (Tstruct _hashtable noattr)) _buckets
            (tarray (tptr (Tstruct _cell noattr)) 109)) (Etempvar _b tuint)
          (tptr (tptr (Tstruct _cell noattr)))) ::
       (Etempvar _s (tptr tschar)) :: nil))))
|}.
Definition f_incrx := {|
  fn_return := tvoid;
  fn_callconv := cc_default;
  fn_params := ((_table, (tptr (Tstruct _hashtable noattr))) ::
                (_s, (tptr tschar)) :: nil);
  fn_vars := nil;
  fn_temps := ((_h, tuint) :: (_b, tuint) ::
               (_p, (tptr (Tstruct _cell noattr))) ::
               (_t'3, (tptr (Tstruct _cell noattr))) :: (_t'2, tint) ::
               (_t'1, tuint) :: (_t'6, (tptr tschar)) :: (_t'5, tuint) ::
               (_t'4, (tptr (Tstruct _cell noattr))) :: nil);
```

```
  fn_body :=
(Ssequence
  (Ssequence
    (Scall (Some _t'1)
      (Evar _hash (Tfunction (Tcons (tptr tschar) Tnil) tuint cc_default))
      ((Etempvar _s (tptr tschar)) :: nil))
    (Sset _h (Etempvar _t'1 tuint)))
  (Ssequence
    (Sset _b
      (Ebinop Omod (Etempvar _h tuint) (Econst_int (Int.repr 109) tint)
        tuint))
    (Ssequence
      (Sset _p
        (Ederef
          (Ebinop Oadd
            (Efield
              (Ederef (Etempvar _table (tptr (Tstruct _hashtable noattr)))
                (Tstruct _hashtable noattr)) _buckets
              (tarray (tptr (Tstruct _cell noattr)) 109)) (Etempvar _b tuint)
            (tptr (tptr (Tstruct _cell noattr))))
          (tptr (Tstruct _cell noattr))))
      (Ssequence
        (Swhile
          (Etempvar _p (tptr (Tstruct _cell noattr)))
          (Ssequence
            (Ssequence
              (Ssequence
                (Sset _t'6
                  (Efield
                    (Ederef (Etempvar _p (tptr (Tstruct _cell noattr)))
                      (Tstruct _cell noattr)) _key (tptr tschar)))
                (Scall (Some _t'2)
                  (Evar _strcmp (Tfunction
                                  (Tcons (tptr tschar)
                                    (Tcons (tptr tschar) Tnil)) tint
                                  cc_default))
                  ((Etempvar _t'6 (tptr tschar)) ::
                   (Etempvar _s (tptr tschar)) :: nil)))
              (Sifthenelse (Ebinop Oeq (Etempvar _t'2 tint)
                             (Econst_int (Int.repr 0) tint) tint)
                (Ssequence
                  (Ssequence
```

```
              (Sset _t'5
                (Efield
                  (Ederef (Etempvar _p (tptr (Tstruct _cell noattr)))
                    (Tstruct _cell noattr)) _count tuint))
              (Sassign
                (Efield
                  (Ederef (Etempvar _p (tptr (Tstruct _cell noattr)))
                    (Tstruct _cell noattr)) _count tuint)
                (Ebinop Oadd (Etempvar _t'5 tuint)
                  (Econst_int (Int.repr 1) tint) tuint)))
            (Sreturn None))
          Sskip))
      (Sset _p
        (Efield
          (Ederef (Etempvar _p (tptr (Tstruct _cell noattr)))
            (Tstruct _cell noattr)) _next
          (tptr (Tstruct _cell noattr))))))
  (Ssequence
    (Ssequence
      (Sset _t'4
        (Ederef
          (Ebinop Oadd
            (Efield
              (Ederef
                (Etempvar _table (tptr (Tstruct _hashtable noattr)))
                (Tstruct _hashtable noattr)) _buckets
              (tarray (tptr (Tstruct _cell noattr)) 109))
            (Etempvar _b tuint) (tptr (tptr (Tstruct _cell noattr))))
          (tptr (Tstruct _cell noattr))))
      (Scall (Some _t'3)
        (Evar _new_cell (Tfunction
                          (Tcons (tptr tschar)
                            (Tcons tint
                              (Tcons (tptr (Tstruct _cell noattr))
                                Tnil))) (tptr (Tstruct _cell noattr))
                          cc_default))
        ((Etempvar _s (tptr tschar)) ::
         (Econst_int (Int.repr 1) tint) ::
         (Etempvar _t'4 (tptr (Tstruct _cell noattr))) :: nil)))
    (Sassign
      (Ederef
        (Ebinop Oadd
```

```
                    (Efield
                      (Ederef
                        (Etempvar _table (tptr (Tstruct _hashtable noattr)))
                        (Tstruct _hashtable noattr)) _buckets
                      (tarray (tptr (Tstruct _cell noattr)) 109))
                    (Etempvar _b tuint) (tptr (tptr (Tstruct _cell noattr))))
                  (tptr (Tstruct _cell noattr)))
                (Etempvar _t'3 (tptr (Tstruct _cell noattr))))))))
|}.

Definition composites : list composite_definition :=
(Composite _cell Struct
   ((_key, (tptr tschar)) :: (_count, tuint) ::
    (_next, (tptr (Tstruct _cell noattr))) :: nil)
   noattr ::
 Composite _hashtable Struct
   ((_buckets, (tarray (tptr (Tstruct _cell noattr)) 109)) :: nil)
   noattr :: nil).

Definition global_definitions : list (ident × globdef fundef type) :=
((___builtin_ais_annot,
   Gfun(External (EF_builtin "__builtin_ais_annot"
                   (mksignature (AST.Tint :: nil) AST.Tvoid
                     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
     (Tcons (tptr tschar) Tnil) tvoid
     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
 (___builtin_bswap64,
   Gfun(External (EF_builtin "__builtin_bswap64"
                   (mksignature (AST.Tlong :: nil) AST.Tlong cc_default))
     (Tcons tulong Tnil) tulong cc_default)) ::
 (___builtin_bswap,
   Gfun(External (EF_builtin "__builtin_bswap"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
     (Tcons tuint Tnil) tuint cc_default)) ::
 (___builtin_bswap32,
   Gfun(External (EF_builtin "__builtin_bswap32"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
     (Tcons tuint Tnil) tuint cc_default)) ::
 (___builtin_bswap16,
   Gfun(External (EF_builtin "__builtin_bswap16"
                   (mksignature (AST.Tint :: nil) AST.Tint16unsigned
                     cc_default)) (Tcons tushort Tnil) tushort cc_default)) ::
 (___builtin_fabs,
   Gfun(External (EF_builtin "__builtin_fabs"
```

```
                    (mksignature (AST.Tfloat :: nil) AST.Tfloat cc_default))
   (Tcons tdouble Tnil) tdouble cc_default)) ::
(___builtin_fsqrt,
  Gfun(External (EF_builtin "__builtin_fsqrt"
                    (mksignature (AST.Tfloat :: nil) AST.Tfloat cc_default))
   (Tcons tdouble Tnil) tdouble cc_default)) ::
(___builtin_memcpy_aligned,
  Gfun(External (EF_builtin "__builtin_memcpy_aligned"
                    (mksignature
                      (AST.Tint :: AST.Tint :: AST.Tint :: AST.Tint :: nil)
                      AST.Tvoid cc_default))
   (Tcons (tptr tvoid)
     (Tcons (tptr tvoid) (Tcons tuint (Tcons tuint Tnil)))) tvoid
   cc_default)) ::
(___builtin_sel,
  Gfun(External (EF_builtin "__builtin_sel"
                    (mksignature (AST.Tint :: nil) AST.Tvoid
                      {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
   (Tcons tbool Tnil) tvoid
   {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
(___builtin_annot,
  Gfun(External (EF_builtin "__builtin_annot"
                    (mksignature (AST.Tint :: nil) AST.Tvoid
                      {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
   (Tcons (tptr tschar) Tnil) tvoid
   {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
(___builtin_annot_intval,
  Gfun(External (EF_builtin "__builtin_annot_intval"
                    (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tint
                      cc_default)) (Tcons (tptr tschar) (Tcons tint Tnil))
   tint cc_default)) ::
(___builtin_membar,
  Gfun(External (EF_builtin "__builtin_membar"
                    (mksignature nil AST.Tvoid cc_default)) Tnil tvoid
   cc_default)) ::
(___builtin_va_start,
  Gfun(External (EF_builtin "__builtin_va_start"
                    (mksignature (AST.Tint :: nil) AST.Tvoid cc_default))
   (Tcons (tptr tvoid) Tnil) tvoid cc_default)) ::
(___builtin_va_arg,
  Gfun(External (EF_builtin "__builtin_va_arg"
                    (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
```

```
                  cc_default)) (Tcons (tptr tvoid) (Tcons tuint Tnil))
    tvoid cc_default)) ::
  (___builtin_va_copy,
   Gfun(External (EF_builtin "__builtin_va_copy"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                     cc_default))
     (Tcons (tptr tvoid) (Tcons (tptr tvoid) Tnil)) tvoid cc_default)) ::
  (___builtin_va_end,
   Gfun(External (EF_builtin "__builtin_va_end"
                   (mksignature (AST.Tint :: nil) AST.Tvoid cc_default))
     (Tcons (tptr tvoid) Tnil) tvoid cc_default)) ::
  (___compcert_va_int32,
   Gfun(External (EF_external "__compcert_va_int32"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
     (Tcons (tptr tvoid) Tnil) tuint cc_default)) ::
  (___compcert_va_int64,
   Gfun(External (EF_external "__compcert_va_int64"
                   (mksignature (AST.Tint :: nil) AST.Tlong cc_default))
     (Tcons (tptr tvoid) Tnil) tulong cc_default)) ::
  (___compcert_va_float64,
   Gfun(External (EF_external "__compcert_va_float64"
                   (mksignature (AST.Tint :: nil) AST.Tfloat cc_default))
     (Tcons (tptr tvoid) Tnil) tdouble cc_default)) ::
  (___compcert_va_composite,
   Gfun(External (EF_external "__compcert_va_composite"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tint
                     cc_default)) (Tcons (tptr tvoid) (Tcons tuint Tnil))
     (tptr tvoid) cc_default)) ::
  (___compcert_i64_dtos,
   Gfun(External (EF_runtime "__compcert_i64_dtos"
                   (mksignature (AST.Tfloat :: nil) AST.Tlong cc_default))
     (Tcons tdouble Tnil) tlong cc_default)) ::
  (___compcert_i64_dtou,
   Gfun(External (EF_runtime "__compcert_i64_dtou"
                   (mksignature (AST.Tfloat :: nil) AST.Tlong cc_default))
     (Tcons tdouble Tnil) tulong cc_default)) ::
  (___compcert_i64_stod,
   Gfun(External (EF_runtime "__compcert_i64_stod"
                   (mksignature (AST.Tlong :: nil) AST.Tfloat cc_default))
     (Tcons tlong Tnil) tdouble cc_default)) ::
  (___compcert_i64_utod,
   Gfun(External (EF_runtime "__compcert_i64_utod"
```

```
                      (mksignature (AST.Tlong :: nil) AST.Tfloat cc_default))
   (Tcons tulong Tnil) tdouble cc_default)) ::
(___compcert_i64_stof,
  Gfun(External (EF_runtime "__compcert_i64_stof"
                      (mksignature (AST.Tlong :: nil) AST.Tsingle cc_default))
   (Tcons tlong Tnil) tfloat cc_default)) ::
(___compcert_i64_utof,
  Gfun(External (EF_runtime "__compcert_i64_utof"
                      (mksignature (AST.Tlong :: nil) AST.Tsingle cc_default))
   (Tcons tulong Tnil) tfloat cc_default)) ::
(___compcert_i64_sdiv,
  Gfun(External (EF_runtime "__compcert_i64_sdiv"
                      (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                        cc_default)) (Tcons tlong (Tcons tlong Tnil)) tlong
   cc_default)) ::
(___compcert_i64_udiv,
  Gfun(External (EF_runtime "__compcert_i64_udiv"
                      (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                        cc_default)) (Tcons tulong (Tcons tulong Tnil)) tulong
   cc_default)) ::
(___compcert_i64_smod,
  Gfun(External (EF_runtime "__compcert_i64_smod"
                      (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                        cc_default)) (Tcons tlong (Tcons tlong Tnil)) tlong
   cc_default)) ::
(___compcert_i64_umod,
  Gfun(External (EF_runtime "__compcert_i64_umod"
                      (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                        cc_default)) (Tcons tulong (Tcons tulong Tnil)) tulong
   cc_default)) ::
(___compcert_i64_shl,
  Gfun(External (EF_runtime "__compcert_i64_shl"
                      (mksignature (AST.Tlong :: AST.Tint :: nil) AST.Tlong
                        cc_default)) (Tcons tlong (Tcons tint Tnil)) tlong
   cc_default)) ::
(___compcert_i64_shr,
  Gfun(External (EF_runtime "__compcert_i64_shr"
                      (mksignature (AST.Tlong :: AST.Tint :: nil) AST.Tlong
                        cc_default)) (Tcons tulong (Tcons tint Tnil)) tulong
   cc_default)) ::
(___compcert_i64_sar,
  Gfun(External (EF_runtime "__compcert_i64_sar"
```

```
                    (mksignature (AST.Tlong :: AST.Tint :: nil) AST.Tlong
                      cc_default)) (Tcons tlong (Tcons tint Tnil)) tlong
    cc_default)) ::
(___compcert_i64_smulh,
  Gfun(External (EF_runtime "__compcert_i64_smulh"
                    (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                      cc_default)) (Tcons tlong (Tcons tlong Tnil)) tlong
    cc_default)) ::
(___compcert_i64_umulh,
  Gfun(External (EF_runtime "__compcert_i64_umulh"
                    (mksignature (AST.Tlong :: AST.Tlong :: nil) AST.Tlong
                      cc_default)) (Tcons tulong (Tcons tulong Tnil)) tulong
    cc_default)) ::
(___builtin_clz,
  Gfun(External (EF_builtin "__builtin_clz"
                    (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_clzl,
  Gfun(External (EF_builtin "__builtin_clzl"
                    (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_clzll,
  Gfun(External (EF_builtin "__builtin_clzll"
                    (mksignature (AST.Tlong :: nil) AST.Tint cc_default))
    (Tcons tulong Tnil) tint cc_default)) ::
(___builtin_ctz,
  Gfun(External (EF_builtin "__builtin_ctz"
                    (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_ctzl,
  Gfun(External (EF_builtin "__builtin_ctzl"
                    (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons tuint Tnil) tint cc_default)) ::
(___builtin_ctzll,
  Gfun(External (EF_builtin "__builtin_ctzll"
                    (mksignature (AST.Tlong :: nil) AST.Tint cc_default))
    (Tcons tulong Tnil) tint cc_default)) ::
(___builtin_fmax,
  Gfun(External (EF_builtin "__builtin_fmax"
                    (mksignature (AST.Tfloat :: AST.Tfloat :: nil) AST.Tfloat
                      cc_default)) (Tcons tdouble (Tcons tdouble Tnil))
    tdouble cc_default)) ::
```

```
(___builtin_fmin,
  Gfun(External (EF_builtin "__builtin_fmin"
                    (mksignature (AST.Tfloat :: AST.Tfloat :: nil) AST.Tfloat
                      cc_default)) (Tcons tdouble (Tcons tdouble Tnil))
    tdouble cc_default)) ::
(___builtin_fmadd,
  Gfun(External (EF_builtin "__builtin_fmadd"
                    (mksignature
                      (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                      AST.Tfloat cc_default))
    (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
    cc_default)) ::
(___builtin_fmsub,
  Gfun(External (EF_builtin "__builtin_fmsub"
                    (mksignature
                      (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                      AST.Tfloat cc_default))
    (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
    cc_default)) ::
(___builtin_fnmadd,
  Gfun(External (EF_builtin "__builtin_fnmadd"
                    (mksignature
                      (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                      AST.Tfloat cc_default))
    (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
    cc_default)) ::
(___builtin_fnmsub,
  Gfun(External (EF_builtin "__builtin_fnmsub"
                    (mksignature
                      (AST.Tfloat :: AST.Tfloat :: AST.Tfloat :: nil)
                      AST.Tfloat cc_default))
    (Tcons tdouble (Tcons tdouble (Tcons tdouble Tnil))) tdouble
    cc_default)) ::
(___builtin_read16_reversed,
  Gfun(External (EF_builtin "__builtin_read16_reversed"
                    (mksignature (AST.Tint :: nil) AST.Tint16unsigned
                      cc_default)) (Tcons (tptr tushort) Tnil) tushort
    cc_default)) ::
(___builtin_read32_reversed,
  Gfun(External (EF_builtin "__builtin_read32_reversed"
                    (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons (tptr tuint) Tnil) tuint cc_default)) ::
```

```
(___builtin_write16_reversed,
  Gfun(External (EF_builtin "__builtin_write16_reversed"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                     cc_default)) (Tcons (tptr tushort) (Tcons tushort Tnil))
    tvoid cc_default)) ::
(___builtin_write32_reversed,
  Gfun(External (EF_builtin "__builtin_write32_reversed"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tvoid
                     cc_default)) (Tcons (tptr tuint) (Tcons tuint Tnil))
    tvoid cc_default)) ::
(___builtin_debug,
  Gfun(External (EF_external "__builtin_debug"
                   (mksignature (AST.Tint :: nil) AST.Tvoid
                     {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|}))
    (Tcons tint Tnil) tvoid
    {|cc_vararg:=true; cc_unproto:=false; cc_structret:=false|})) ::
(_malloc,
  Gfun(External EF_malloc (Tcons tuint Tnil) (tptr tvoid) cc_default)) ::
(_exit,
  Gfun(External (EF_external "exit"
                   (mksignature (AST.Tint :: nil) AST.Tvoid cc_default))
    (Tcons tint Tnil) tvoid cc_default)) ::
(_strlen,
  Gfun(External (EF_external "strlen"
                   (mksignature (AST.Tint :: nil) AST.Tint cc_default))
    (Tcons (tptr tschar) Tnil) tuint cc_default)) ::
(_strcpy,
  Gfun(External (EF_external "strcpy"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tint
                     cc_default))
    (Tcons (tptr tschar) (Tcons (tptr tschar) Tnil)) (tptr tschar)
    cc_default)) ::
(_strcmp,
  Gfun(External (EF_external "strcmp"
                   (mksignature (AST.Tint :: AST.Tint :: nil) AST.Tint
                     cc_default))
    (Tcons (tptr tschar) (Tcons (tptr tschar) Tnil)) tint cc_default)) ::
(_hash, Gfun(Internal f_hash)) ::
(_copy_string, Gfun(Internal f_copy_string)) ::
(_new_table, Gfun(Internal f_new_table)) ::
(_new_cell, Gfun(Internal f_new_cell)) :: (_get, Gfun(Internal f_get)) ::
(_incr_list, Gfun(Internal f_incr_list)) ::
```

```
  (_incr, Gfun(Internal f_incr)) :: (_incrx, Gfun(Internal f_incrx)) :: nil).

Definition public_idents : list ident :=
(_incrx :: _incr :: _incr_list :: _get :: _new_cell :: _new_table ::
 _copy_string :: _hash :: _strcmp :: _strcpy :: _strlen :: _exit ::
 _malloc :: ___builtin_debug :: ___builtin_write32_reversed ::
 ___builtin_write16_reversed :: ___builtin_read32_reversed ::
 ___builtin_read16_reversed :: ___builtin_fnmsub :: ___builtin_fnmadd ::
 ___builtin_fmsub :: ___builtin_fmadd :: ___builtin_fmin ::
 ___builtin_fmax :: ___builtin_ctzll :: ___builtin_ctzl :: ___builtin_ctz ::
 ___builtin_clzll :: ___builtin_clzl :: ___builtin_clz ::
 ___compcert_i64_umulh :: ___compcert_i64_smulh :: ___compcert_i64_sar ::
 ___compcert_i64_shr :: ___compcert_i64_shl :: ___compcert_i64_umod ::
 ___compcert_i64_smod :: ___compcert_i64_udiv :: ___compcert_i64_sdiv ::
 ___compcert_i64_utof :: ___compcert_i64_stof :: ___compcert_i64_utod ::
 ___compcert_i64_stod :: ___compcert_i64_dtou :: ___compcert_i64_dtos ::
 ___compcert_va_composite :: ___compcert_va_float64 ::
 ___compcert_va_int64 :: ___compcert_va_int32 :: ___builtin_va_end ::
 ___builtin_va_copy :: ___builtin_va_arg :: ___builtin_va_start ::
 ___builtin_membar :: ___builtin_annot_intval :: ___builtin_annot ::
 ___builtin_sel :: ___builtin_memcpy_aligned :: ___builtin_fsqrt ::
 ___builtin_fabs :: ___builtin_bswap16 :: ___builtin_bswap32 ::
 ___builtin_bswap :: ___builtin_bswap64 :: ___builtin_ais_annot :: nil).

Definition prog : Clight.program :=
  mkprogram composites global_definitions public_idents _main Logic.I.
```

Chapter 7

Library VC.hints

```
Require Import VST.floyd.proofauto.
Require Import VST.floyd.library.

Ltac verif_stack_free_hint1 :=
match goal with
⊢ semax ?D (PROPx _ (LOCALx ?Q (SEPx ?R)))
            (Ssequence
                 (Scall _ (Evar ?free (Tfunction (Tcons (tptr tvoid) Tnil) tvoid cc_default))
                 (Etempvar ?i _ :: _)) _) _ ⇒
  match Q with context [temp i ?q] ⇒
   match R with context [data_at _ ?t _ q] ⇒
        unify ((glob_specs D) ! free) (Some library.free_spec');
        idtac "When doing forward_call through this call to" free
  "you need to supply a WITH-witness of type (type*val*globals) and you need to supply a proof that"
    q "<>nullval. Look in your SEP clauses for 'data_at _" t "_" q"', which will be useful for both."
"Regarding the proof, assert_PROP(...) will make use of the fact that data_at cannot be a nullval."
"Regarding the witness, you should look at the funspec declared for" free
"to see what will be needed; look in Verif_stack.v at free_spec_example."
"But in particular, for the type, you can use the second argument of the data_at, that is," t
".";
  match goal with a : globals ⊢ _ ⇒
     idtac "Regarding the 'globals', you have" a ": globals above the line."
   end
  end end
end.

Ltac verif_stack_malloc_hint1_aux D R c :=
  match c with
```

```
  | Ssequence ?c1 _ ⇒ verif_stack_malloc_hint1_aux D R c1
  | Scall _ (Evar ?malloc
                  (Tfunction (Tcons tuint Tnil) (tptr tvoid) cc_default))
                 (cons (Esizeof ?t _) nil) ⇒
      match R with context [mem_mgr ?gv] ⇒
          idtac "try 'forward_call (" t "," gv ")'"
     end
  end.

Ltac verif_stack_malloc_hint1 :=
match goal with ⊢ semax ?D (PROPx _ (LOCALx _ (SEPx ?R))) ?c _ ⇒
      verif_stack_malloc_hint1_aux D R c
end.

Ltac vc_special_hint :=
  first
     [ verif_stack_free_hint1
     | verif_stack_malloc_hint1
     ];
  idtac "THAT WAS NOT A STANDARD VST HINT, IT IS SPECIAL FOR THE VC
VOLUME OF SOFTWARE FOUNDATIONS."
  "STANDARD VST HINTS WOULD BE AS FOLLOWS: ".

Ltac hint_special ::= try vc_special_hint.
```

Chapter 8

Library VC.Preface

8.1 Preface

8.2 Welcome

Here's a good way to build formally verified correct software:

- Write your program in an expressive language with a good proof theory (the Gallina language embedded in Coq's logic).
- Prove it correct in Coq.
- Extract it to ML and compile it with an optimizing ML compiler.

Unfortunately, for some applications you cannot afford to use a higher-order garbage-collected functional programming language such as Gallina or ML. Perhaps you are writing an operating-system kernel, or a bit-shuffling cryptographic primitive, or the runtime system and garbage-collector of your functional language! In those cases, you might want to use a low-level imperative systems programming language such as C.

But you still want your OS, or crypto, or GC, to be correct! So you should use machine-checked program verification in Coq. For that purpose, you can use *Verifiable C*, a program logic and proof system for C.

What is a program logic? One example of a program logic is the Hoare logic that you studied in the *Programming Language Foundations* volume of this series. (If you have not done so already, study the Hoare and Hoare2 chapters of that volume, and do the exercises.)

Verifiable C is based on a 21st-century version of Hoare logic called *higher-order impredicative concurrent separation logic*. Back in the 20th century, computer scientists discovered that Hoare Logic was not very good at verifying programs with pointer data structures; so *separation logic* was developed. Hoare Logic was clumsy at verifying concurrent programs, so *concurrent separation logic* was developed. Hoare Logic could not handle higher-order object-oriented programming patterns or function-closures, so *higher-order impredicative program logics* were developed.

This electronic book is Volume 5 of the *Software Foundations* series, which presents the mathematical underpinnings of reliable software. The principal novelty of *Software Foundations* is that it is one hundred percent formalized and machine-checked: the entire text is literally a script for Coq. It is intended to be read alongside an interactive session with Coq. All the details in the text are fully formalized in Coq, and the exercises are designed to be worked using Coq.

Before studying this volume, you should be a competent user of Coq:

- Study *Software Foundations Volume 1* (Logical Foundations), and do the exercises!

- Study the Hoare and Hoare2 chapters of *Software Foundations Volume 2* (Programming Language Foundations), and do the exercises!

- Study the Sort, SearchTree, and ADT chapters of *Software Foundations Volume 3* (Verified Functional Algorithms), and do the exercises!

You will also need a working knowledge of the C programming language.

8.3 Practicalities

8.3.1 System Requirements

Coq runs on Windows, Linux, and OS X. The Preface of Volume 1 describes the Coq installation you will need. This edition was built with Coq 8.12.0.

You will need VST installed. You can do that either by installing it as part of the standard "Coq Platform" that is released with each new version of Coq, or using opam (the package is named coq-vst). At the end of this chapter is a test to make sure you have the right version of VST installed.

IF YOU USE OPAM, the following opam commands may be useful:

- opam repo add coq-released

- opam pin coq 8.12.0

- opam install coq-vst.2.6 (*this will take 30 minutes or more*)

- (to use coqide:) opam pin lablgtk3 3.0.beta5

- (to use coqide:) opam install coqide

You do not need to install CompCert clightgen to do the exercises in this volume. But if you wish to modify and reparse the .c files, or verify C programs of your own, install the CompCert verified optimizing C compiler. You can get CompCert from compcert.inria.fr, or (starting with Coq 8.12) in the standard "Coq Package" or by opam (the package is named coq-compcert).

8.3.2 Downloading the Coq Files

A tar file containing the full sources for the "release version" of this book (as a collection of Coq scripts and HTML files) is available at https://softwarefoundations.cis.upenn.edu.

(If you are using the book as part of a class, your professor may give you access to a locally modified version of the files, which you should use instead of the release version.)

8.3.3 Installation

Unpack the *vc.tgz* tar file into a vc directory. In this *vc* directory, the *make* command builds it.

8.3.4 Exercises

Each chapter includes exercises. Each is marked with a "star rating," which can be interpreted as follows:

- One star: easy exercises that underscore points in the text and that, for most readers, should take only a minute or two. Get in the habit of working these as you reach them.
- Two stars: straightforward exercises (five or ten minutes).
- Three stars: exercises requiring a bit of thought (ten minutes to half an hour).
- Four and five stars: more difficult exercises (half an hour and up).

Please do not post solutions to the exercises in any public place: Software Foundations is widely used both for self-study and for university courses. Having solutions easily available makes it much less useful for courses, which typically have graded homework assignments. The authors especially request that readers not post solutions to the exercises anyplace where they can be found by search engines.

8.3.5 Recommended Citation Format

If you want to refer to this volume in your own writing, please do so as follows:

@book {Appel:SF5, author = {Andrew W. Appel and Qinxiang Cao} title = "Verifiable C" series = "Software Foundations", volume = "5", year = "2020", publisher = "Electronic textbook", note = {Version 0.9.7, \URLhttp://softwarefoundations.cis.upenn.edu }, }

8.3.6 For Instructors and Contributors

If you plan to use these materials in your own course, you will undoubtedly find things you'd like to change, improve, or add. Your contributions are welcome! Please see the Preface to *Logical Foundations* for instructions.

8.4 Thanks

Development of the *Software Foundations* series has been supported, in part, by the National Science Foundation under the NSF Expeditions grant 1521523, *The Science of Deep Specification*.

8.5 Check for the right version of VST

```
Require Import Coq.Strings.String.
Open Scope string.
Require Import VST.veric.version.

Goal release = "2.6".
reflexivity || fail "The wrong version of VST is installed".
Abort.
```

Chapter 9

Library VC.Verif_sumarray

9.1 Verif_sumarray: Introduction to Verifiable C

9.1.1 Verified Software Toolchain

The Verified Software Toolchain is a toolset for proving the functional correctness of C programs, with

- a *program logic* called Verifiable C, based on separation logic.
- a *proof automation system* called VST-Floyd that assists you in applying the program logic to your program.
- a soundness proof in Coq, guaranteeing that whatever properties you prove about your program will actually hold in any execution of the C source-language operational semantics. And this proof *composes* with the correctness proof of the CompCert verified optimizing C compiler, so you can also get a guarantee about the behavior of the assembly language program.

This volume of *Software Foundations* teaches you how to use Verifiable C and VST-Floyd to prove C programs correct. In the process you'll learn some key concepts of Hoare Logic and Separation Logic. This book does *not* cover VST's soundness proof (which is described in the book *Program Logics for Certified Compilers Appel* 2014 (in Bib.v)).

9.1.2 How to use this textbook

The first two chapters (this one and Verif_reverse) are a feature-by-feature introduction to Verifiable C, demonstrated on two example C programs: adding up an array and reversing a linked list. These chapters are best understood if you step through them in Coq, where you can see the proof goals at each stage; they are less useful to read in HTML. These two chapters closely follow the first 48 mini-chapters of the *Verifiable C Reference Manual*, *VC.pdf*, that is distributed with VST – and you can find a copy distributed with this volume of *Software Foundations*. The first two chapters have no exercises.

This *Verifiable C* volume of *Software Foundations* is self-contained, so you should not need to look things up in the reference manual *VC.pdf*. But to use features of Verifiable C beyond what's needed for this textbook, *VC.pdf* can be very useful. The words SEE ALSO suggest which chapters of the reference manual cover the features discussed in this text.

The remaining 7 chapters are *mainly* exercises. The best way to learn is by doing it yourself – so each chapter presents a little C program, and guides you through verifying it yourself. The "capstone exercise" is the verification of a hash table with external chaining.

9.1.3 A C program to add up an array

Here is a little C program, *sumarray.c*:

9.1.4 Workflow

SEE ALSO: VC.pdf Chapter 3 (Workflow), Chapter 4 (*Verifiable C and clightgen*), Chapter 5 (*ASTs*)

To verify a C program, such as *sumarray.c*, use the CompCert front end to parse it into an Abstract Syntax Tree (AST). For all the chapters in this volume of *Software Foundations* we've done that for you, so you don't have to install clightgen; but generally what you would do is,

clightgen -normalize sumarray.c

You would have installed *clightgen* as part of the CompCert tools, by mentioning the -clightgen option when you run ./configure when building CompCert.

The output of *clightgen* would be a file *sumarray.v* that contains the Coq inductive data structure describing the syntax trees of the source program. You can open *sumarray.v* in the current directory and inspect it.

9.1.5 Let's verify!

SEE ALSO: VC.pdf Chapter 7 (*Functional model, API spec*)

This file, *Verif_sumarray.v*, contains a *specification* of the functional correctness of the program *sumarray.c*, followed by a proof that the program satisfies its specification.

For larger programs, one would typically break this down into three or more files:

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

Make sure you have the right version of VST installed

```
Require VC.Preface.
```

Standard boilerplate

Every API specification begins with the same standard boilerplate; the only thing that changes is the name of the program – in this case, sumarray.

```
Require Import VST.floyd.proofauto.
Require Import VC.sumarray.
Instance CompSpecs : compspecs. make_compspecs prog. Defined.
Definition Vprog : varspecs. mk_varspecs prog. Defined.
```

The first line imports Verifiable C and its *Floyd* proof-automation library. The second line imports the AST of the program to be verified. The third line processes all the struct and union definitions in the AST, and the fourth line processes global variable declarations.

Functional model

To prove correctness of *sumarray.c*, we start by writing a *functional model* of adding up a sequence. We can use a list-fold to express the sum of all the elements in a list of integers:

```
Definition sum_Z : list Z → Z := fold_right Z.add 0.
```

Then we prove properties of the functional model: in this case, how sum_Z interacts with list append.

```
Lemma sum_Z_app:
  ∀ a b, sum_Z (a++b) = sum_Z a + sum_Z b.
Proof.
  intros. induction a; simpl; lia.
Qed.
```

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

9.1.6 API spec for the sumarray.c program

The Application Programmer Interface (API) of a C program is expressed in its header file: function prototypes and data-structure definitions that explain how to call upon the modules' functionality. In Verifiable C, an *API specification* is written as a series of *function specifications* (**funspec**s) corresponding to the function prototypes.

```
Definition sumarray_spec : ident × funspec :=
DECLARE _sumarray
 WITH a: val, sh : share, contents : list Z, size: Z
 PRE [ tptr tuint, tint ]
  PROP (readable_share sh; 0 ≤ size ≤ Int.max_signed;
        Forall (fun x ⇒ 0 ≤ x ≤ Int.max_unsigned) contents)
  PARAMS (a; Vint (Int.repr size))
  SEP (data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a)
 POST [ tuint ]
  PROP () RETURN (Vint (Int.repr (sum_Z contents)))
  SEP (data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a).
```

This *DECLARE* statement has type ident×**funspec**. That is, it associates the name of a function (the identifier _sumarray) with a function-specification. The identifier _sumarray comes directly from the C program, as parsed by *clightgen*. If you are curious, you can look in *sumarray.v* (the output of *clightgen*) for `Definition _sumarray := ....` Later in *sumarray.v*, you can see `Definition f_sumarray` that is the C-language function body (represented as a syntax tree).

A function is specified by its *precondition* and its *postcondition*. The *WITH* clause quantifies over Coq values that may appear in both the precondition and the postcondition. The precondition has access to the function parameters (in this case a and *size*) and the postcondition has access to the return value (sum_Z *contents*).

Function preconditions, postconditions, and loop invariants are *assertions* about the state of variables and memory at a particular program point. In an assertion $PROP(P)$ $LOCAL(Q)$ $SEP(R)$, the propositions in the sequence P are all of Coq type `Prop`. They describe things that are true independent of program state. In the precondition above, the statement $0 \leq size \leq Int.max\_signed$ is true *just within the scope of the quantification of the variable size*; that variable is bound by *WITH*, and spans the *PRE* and *POST* assertions.

The *LOCAL* clause, describing what's in C local variables, takes different forms depending on context:

- In a function-precondition, we write PROP/PARAMS/SEP, that is, the *PARAMS* lists the values of C function parameters (in order).
- In a function-postcondition, we write $RETURN(v)$ to indicate the return value of the function.
- Within a function body (in assertions and invariants) we write *LOCAL* to describe the values of local variables (including parameters).

Whether it is PARAMS or RETURN or LOCAL, we are talking about the *values* contained in parameters or local variables. In general, a C scalar variable holds something of type **val**; this type is defined by CompCert as, `Print val`.

In an assertion $PROP(P)\ LOCAL(Q)\ SEP(R)$, the SEP conjuncts R are *spatial assertions* in separation logic. In our example precondition, there's just one SEP conjunct, a data_at assertion saying that at address a in memory, there is a data structure of type

array*size* of unsigned int;

with access-permission *sh*, and the contents of that array is the sequence map Vint (map *Int.repr contents*).

The postcondition is introduced by $POST$ [tuint], indicating that this function returns a value of type *unsigned* **int**. There are no $PROP$ statements in this postcondition–no forever-true facts hold now, that weren't already true on entry to the function.

$RETURN(v)$ gives the return value v; $RETURN()$ for void functions.

The postcondition's SEP clause mentions all the spatial resources from the precondition, minus ones that have been freed (deallocated), plus ones that have been malloc'd (allocated).

So, overall, the specification for sumarray is this: "At any call to sumarray, there exist values *a*, *sh*, *contents*, *size* such that *sh* gives at least read-permission; *size* is representable as a nonnegative 32-bit signed integer; function-parameter _a contains value *a* and _n contains the 32-bit representation of *size*; and there's an array in memory at address *a* with permission *sh* containing *contents*. The function returns a value equal to *sum_int*(*contents*), and leaves the array in memory unaltered."

Function specification for main()

The function-spec for *main* has a special form, which we discuss below in the section called *Global variables and main*. In particular, its precondition is defined using main_pre.

```
Definition main_spec :=
 DECLARE _main
  WITH gv : globals
  PRE [] main_pre prog tt gv
  POST [ tint ]
     PROP()
     RETURN (Vint (Int.repr (1+2+3+4)))
     SEP(TT).
```

This postcondition says we have indeed added up the global array *four*.

Integer overflow

In Verifiable C's signed integer arithmetic, you must prove (if the system cannot prove automatically) that no overflow occurs. For unsigned integers, arithmetic is treated as modulo-2ˆn (where n is typically 32 or 64), and overflow is not an issue. The function *Int.repr*: $\mathsf{Z} \rightarrow$ **int** truncates mathematical integers into 32-bit integers by taking the (sign-extended) low-order 32 bits. *Int.signed*: **int** $\rightarrow \mathsf{Z}$ injects back into the signed integers.

The sumarray program uses unsigned arithmetic for *s* and the array contents; it uses signed arithmetic for *i*.

The postcondition guarantees that the value returned is *Int.repr* (sum_Z *contents*). But what if the sum of all the s is larger than 2ˆ32, so the sum doesn't fit in a 32-bit signed integer? Then *Int.unsigned*(*Int.repr* (sum_Z *contents*)) $\neq$ sum_Z *contents*. In general, for a claim about *Int.repr*(x) to be *useful* one also needs to know that $0 \leq x \leq$ *Int.max_unsigned* or *Int.min_signed* $\leq x \leq$ *Int.max_signed*. The caller of sumarray will probably need to prove $0 \leq$ sum_Z *contents* $\leq$ *Int.max_unsigned* in order to make much use of the postcondition.

9.1.7 Packaging the Gprog and Vprog

SEE ALSO: VC.pdf Chapter 8 (*Proof of the sumarray program*)

To prove the correctness of a whole program,

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

The first step is easy:

```
Definition Gprog := [sumarray_spec; main_spec].
```

What's in Gprog are the funspecs that we built using *DECLARE*. (In multi-module programs we would also include imported funspecs.)

In addition to Gprog, the API spec contains Vprog, the list of global-variable type-specs. This was computed automatically by the *mk_varspecs* tactic, in the "boilerplate" code above.

```
Print Vprog.
Print varspecs.
```

That is, for each C language global variable, Vprog gives its name and its C-language type.

9.1.8 Proof of the sumarray program

Now comes the proof that f_sumarray, the body of the sumarray() function, satisfies sumarray_spec, in global context (Vprog,Gprog). Lemma body_sumarray: semax_body Vprog Gprog f_sumarray sumarray_spec.

Here, f_sumarray is the actual function body (AST of the C code) as parsed by *clightgen*; you can read it in *sumarray.v*. You can read body_sumarray as claiming: In the context of Vprog and Gprog, the function body f_sumarray satisfies its specification sumarray_spec. We need the context in case the sumarray function refers to a global variable (Vprog provides the variable's type) or calls a global function (Gprog provides the function's API spec).

Now, the proof of body_sumarray.

```
Proof.
```

If you are reading this as a static document, you should consider switching to your favorite Coq development environment, in which you can step through the rest of this chapter, tactic by tactic, and examine the proof state at each point.

start_function

SEE ALSO: VC.pdf Chapter 9 (*start_function*)

The predicate semax_body states the Hoare triple of the function body, $Delta \vdash \{Pre\}$ c $\{Post\}$, where *Pre* and *Post* are taken from the **funspec**, c is the body of the function, and the type-context *Delta* is calculated from the global type-context overlaid with the parameter- and local-types of the function.

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

start_function.

Some of the assumptions you now see above the line are,

- *a*, *sh*, *contents*, *size*, taken directly from the WITH clause of sumarray_spec;
- *Delta_specs*, the context in which Floyd's proof tactics will look up the specifications of global functions;
- *Delta*, the context in which Floyd will look up the types of local and global variables;
- *SH*,*H*,*H0*, taken exactly from the *PROP* clauses of sumarray_spec's precondition.

There are also two *abbreviations* above the line, *POSTCONDITION* and *MORE_COMMANDS*, discussed below.

Forward symbolic execution

SEE ALSO: VC.pdf Chapter 10 (*forward*).

We do Hoare logic proof by forward symbolic execution. At the beginning of this function body, our proof goal is a Hoare triple about the statement (i=0; ...*more commands*...). In a forward Hoare logic proof of $\{P\}$(i=0;...*more*...)$\{R\}$ we might first apply the sequence rule,

$$\frac{\{P\}(i{=}0;)\{Q\} \quad \{Q\}(...more...)\{R\}}{\{P\}(i{=}0;...more...)\{R\}}$$

assuming we could derive some appropriate assertion Q. For many kinds of statements (assignments, return, break, continue) Q is derived automatically by the forward tactic, which applies a strongest-postcondition style of proof rule. Let us now apply the forward tactic:

forward.

Look at the precondition of the current proof goal, that is, the second argument of *semax*; it has the form *PROP*(...) *LOCAL*(...) *SEP*(...). That precondition is also the *postcondition* of i=0;. It's much like the *precondition* of i=0; except for one change: we now know that i is equal to 0, which is expressed in the *LOCAL* part as temp _i (Vint (*Int.repr* 0)).

`Check 0. Check (Int.repr 0). Check (Vint (Int.repr 0)). Check (temp _i (Vint (Int.repr 0))).`

abbreviate, MORE_COMMANDS, POSTCONDITION

When doing forward symbolic execution (forward Floyd/Hoare proof) through a large function, you don't usually want to see the entire function-body in your proof subgoal. Therefore the system abbreviates some things for you, using the magic of Coq's implicit arguments.

```
Check @abbreviate.
About abbreviate.
```

We see here that abbreviate is just the identity function, with *both* of its arguments implicit!

To examine the actual contents of MORE_COMMANDS, just do this:

`unfold abbreviate in MORE_COMMANDS.`

or alternately, `subst MORE_COMMANDS; unfold abbreviate.`

Similarly, to see the POSTCONDITION, just do,

`unfold abbreviate in POSTCONDITION.`

Hint

In any VST proof state, the *hint* tactic will print a suggestion (if it can) that will help you make progress in the proof. In stepping through the case study in this chapter, insert *hint* at any point to see what it says.

hint.

Then delete the hints! (They slow down replay of your proof.)

The hint here suggests using *abbreviate_semax*, which will undo the unfold abbreviate that we did above. Really this is optional; if we don't do *abbreviate_semax*, the next forward tactic will do it for us.

abbreviate_semax.
hint.

This time, the hint suggests that we try 'forward'.

Forward through another assignment statement.

forward.

The forward tactic works on assignment statements, break, continue, and return.

While loops, forward_while

SEE ALSO: VC.pdf Chapter 12 (*if, while, for*) and Chapter 13 (*while loops*).

To do symbolic execution through a *while* loop, use the *forward_while* tactic; you must supply a loop invariant. *forward_while*

```
(EX i: Z,
  PROP (0 ≤ i ≤ size)
  LOCAL (temp _a a;
           temp _i (Vint (Int.repr i));
           temp _n (Vint (Int.repr size));
           temp _s (Vint (Int.repr (sum_Z (sublist 0 i contents)))))
  SEP (data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a)).
```

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

forward_while leaves four subgoals; here we label them with the - bullet. - *hint.*

The first subgoal is to prove that the current assertion (precondition) entails the loop invariant.

Proving separation-logic entailments

SEE ALSO: VC.pdf Chapter 14 (*PROP LOCAL SEP*) and Chapter 15 (*Entailments*)

This proof goal is an *entailment*, *ENTAIL Delta*, *P* |– *Q*, meaning "in context *Delta*, any state that satisfies *P* will also satisfy *Q*."

In this case, the right-hand-side of this entailment is existentially quantified; it says: there exists a value *i* such that (among other things) temp _i (Vint (*Int.repr i*)), that is, the C variable _i contains the value *i*. But the left-hand-side of the entailment says temp _i (Vint (*Int.repr* 0)), that is, the C variable _i contains 0.

This is analogous to the following situation:

```
Set Nested Proofs Allowed.
Goal ∀ (f: Z→Z) (x: Z), f(x)=0 → ∃ i:Z, f(x)=i.
  intros.
```

To prove such a goal, one uses Coq's "exists" tactic to demonstrate a value for *i*: ∃ 0.

```
  auto.
Qed.
```

In a separation logic entailment, one can prove an *EX* on the right-hand side by using the *Exists* tactic to demonstrate a value for the quantified variable: *Exists* 0.

Notice that *i* has now been replace with 0 on the right side.

To prove entailments, we usually use the entailer! tactic to simplify the entailment as much as possible–or in many cases, to prove it entirely.

entailer!.

In this case, it solves entirely; in other cases, entailer! leaves subgoals for you to prove.

Type-checking the loop test

\- *hint.*

The second subgoal of *forward_while* is always to prove that the loop-test expression can evaluate without crashing–that is, all the variables it references exist and are initialized, it doesn't divide by zero, et cetera.

We call this a "type-checking condition", the predicate *tc_expr*. In this case, it's the while-loop test $i{<}n$ that must execute, so we see *tc_expr Delta* (! (_i < _n)) on the right-hand side of the entailment.

Very often, these *tc_expr* goals solve automatically by entailer!. *entailer*!.

and indeed, this subgoal is solved.

Proving that the loop body preserves the loop invariant

\- *hint.*

The third subgoal of *forward_while* is to prove that the loop body preserves the loop invariant. We must forward-symbolic-execute through the loop body.

SEE ALSO: VC.pdf Chapter 16 (*Array subscripts*)

Examine the proof goal at the beginning of the loop body. Above the line is the variable i, introduced automatically by *forward_while* from the existential *EX* i:Z in the loop invariant.

The first C command in the loop body is the array subscript, _x = a[_i]; . In order to prove this statement, the forward tactic needs to be able to prove that i is within bounds of the array. When we try forward, it fails:

Fail forward.

SEE ALSO: VST.pdf, Chapter "assert_PROP"

The required information to prove Zlength *contents* = *size* comes from the *precondition* of the current *semax* goal. In the precondition, we have

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

The data_at predicate always enforces that the "contents" list for an array is exactly the same length as the size of the array.

To make use of precondition facts in an assertion, use *assert_PROP*.

assert_PROP (Zlength *contents* = *size*). {

The proof goal is an entailment, with the current precondition on the left, and the proposition to be proved on the right. As usual, to prove an entailment, we use the entailer! tactic to simplify the proof goal:

entailer!.

Indeed, entailer! has done almost all the work. If you want to see how entailer! did it, undo the last step and use these two tactics: *go_lower*. *saturate_local*. The job of *go_lower* is to process the PROP and LOCAL parts of the entailment; and *saturate_local* derives all the propositional facts derivable from the mpreds on the left-hand-side, and puts those facts above the line. In this case, above the line is, Zlength (*unfold_reptype* (map Vint (map *Int.repr contents*))) = *size* which is the fact we need. *hint.*

The *hint* suggests that *list_solve* solves this goal, Zlength *contents* = Zlength (map Vint (map *Int.repr contents*)). Indeed, *list_solve* knows a lot of things about the interaction of list operators: Zlength, map, sublist, etc.

Or, we can solve the goal "by hand": `do 2 rewrite Zlength_map. reflexivity.`

}

hint.

Now that we have Zlength *contents* = *size* above the, we can go forward through the array-subscript statement. *forward.*

Now forward through the rest of the loop body. *forward. forward.*

SEE ALSO: VC.pdf Chapter 17 (*At the end of the loop body*)

We have reached the end of the loop body, and it's time to prove that the *current precondition* (which is the postcondition of the loop body) entails the loop invariant. *Exists* (i+1).

entailer!.

`f_equal. f_equal.`

Here the proof goal is,

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

We will prove this in stages:

sum_Z (sublist 0 (i + 1) contents) = sum_Z (sublist 0 i contents ++ sublist i (i+1) contents) = sum_Z (sublist 0 i contents) + sum_Z (sublist i (i+1) contents) = sum_Z (sublist 0 i contents) + sum_Z (Znth i contents :: nil) = sum_Z (sublist 0 i contents) + Znth i contents

`rewrite (sublist_split 0 i (i+1)) by lia.`

`rewrite sum_Z_app. rewrite (sublist_one i) by lia.`

`simpl. lia.`

After the loop, our precondition is the conjunction of the loop invariant and the negation of the loop test.

SEE ALSO: VC.pdf Chapter 18 (*Returning from a function*)

- *hint.*

You can always go forward through a `return` statement. The resulting proof goal is an entailment, that the current precondition implies the function's postcondition.

forward.

Here we prove that the postcondition of the function body entails the postcondition demanded by the function specification. *entailer*!.

hint.

`autorewrite with sublist in *⊢.`

hint.

`autorewrite with sublist.`

hint.

```
reflexivity.
Qed.
```

9.1.9 Global variables and main()

SEE ALSO: VC.pdf Chapter 19 (*Global variables and main*) `Definition four_contents := [1; 2; 3; 4].`

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

C programs may have extern global variables, either with explicit initializers or implicitly initialized to zero. Because they live in memory, they need to be described by a separation logic predicate, a "resource" that gets passed from one function to another via the SEP part of funspec preconditions and postconditions. Initially, all the global-variable resources are passed into the *main* function, as its precondition. The built-in operator main_pre calculates this precondition of *main* by examining all the global declarations of the program.

In this program, there is one global variable,

unsigned four4 = {1,2,3,4};

and we can see its SEP assertion in the precondition of the current proof goal:

```
data_at Ews (tarray tuint 4)
        (map Vint [Int.repr 1; Int.repr 2; Int.repr 3; Int.repr 4])
        (gv _four)
```

SEE ALSO: VC.pdf Chapter 20 (*Function calls*)

We are ready to prove the function-call, s = sumarray(*four*,4); We use the *forward_call* tactic, and for the argument we must supply a tuple of values that instantiates the WITH clause of the called function's funspec. In *DECLARE* _sumarray, the *WITH* clause reads, *WITH a*: val, *sh* : share, *contents* : list Z, *size*: Z. Therefore the argument to *forward_call* must be a four-tuple of type, (val × share × list Z × Z). *forward_call*

```
  (gv _four, Ews, four_contents, 4).
```

The subgoal of *forward_call* is that we have to prove the PROP part of the sumarray function's precondition.

```
 split3. auto. computable. repeat constructor; computable.
```

Now we are after the function-call, and we can go forward through the return statement. *forward*. Qed.

9.1.10 Tying all the functions together

SEE ALSO: VC.pdf Chapter 21 (*Tying all the functions together*)

The C program may do input/output, affecting the state of the outside world. This state is described (abstractly) by the *Espec*, the "external specification." The sumarray program does not do any input/output, so we can use a trivial *Espec*. We provide this to the semax_prog proofs (below, in the prog_correct lemma) as follows:

```
Existing Instance NullExtension.Espec.
```

This is a *typeclass instance*. If you're not familiar with typeclasses, don't worry, just treat this as "boilerplate" that you can ignore.

An entire C program is proved correct if all the functions satisfy their funspecs. We listed all those functions (upon whose specifications we depend) in the Gprog definition. The judgment semax_prog prog Vprog Gprog says, "In the program prog, whose varspecs are Vprog and whose funspecs are Gprog, every function mentioned in Gprog does satisfy its specification."

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

9.1.11 Additional recommended reading

Recommended: read VC.pdf Chapters 22-47 (up to *Pointer comparisons*)

Chapter 10

Library VC.Verif_reverse

10.1 Verif_reverse: Linked lists in Verifiable C

This chapter demonstrates some more features of Verifiable C. There are no exercises in this chapter.

10.1.1 Running Example

Here is a little C program, *reverse.c*:

SEE ALSO VC.pdf Chapter 46 (*Proof of the reverse program*)

As usual, we import the Verifiable C system VST.floyd.proofauto, then the program to be verified, in this case reverse. Then we give the standard boilerplate definitions of CompSpecs and Vprog.

```
Require VC.Preface. Require Import VST.floyd.proofauto.
Require Import VC.reverse.
Instance CompSpecs : compspecs. make_compspecs prog. Defined.
Definition Vprog : varspecs. mk_varspecs prog. Defined.
```

10.1.2 Inductive definition of linked lists

Tstruct _list noattr is the AST (abstract syntax tree) description of the C-language type struct list. We will be using this a lot, so we make an abbreviation for it, t_list:

```
Definition t_list := Tstruct _list noattr.
```

We will define a separation-logic predicate, listrep *sigma* *p*, to describe the concept that the address *p* in memory is a linked list that represents the mathematical sequence *sigma*. Here, *sigma* is a list of val, which is C's "value" type: integers, pointers, floats, etc.

```
Fixpoint listrep (sigma: list val) (p: val) : mpred :=
 match sigma with
 | h::hs ⇒
    EX y:val, data_at Tsh t_list (h,y) p × listrep hs y
```

```
| nil ⇒
    !! (p = nullval) && emp
end.
```

This says, if *sigma* has head h and tail hs, then there is a cons cell at address p with components (h,y). This cons cell is described by data_at Tsh t_list (h,y) p. Separate from that, at address y, there is the representation of the rest of the list, listrep hs y. The memory footprint for listrep $(h::hs)$ p contains the first cons cell at address p, and the rest of the cons cells in the list starting at address y.

But if *sigma* is nil, then p is the null pointer, and the memory footprint is empty (emp). The fact p=nullval is a pure proposition (Coq `Prop`); we inject this into the assertion language (Coq mpred) using the !! operator.

Because !!P (for a proposition P) does not specify any footprint (whether empty or otherwise), we do not use the separating conjunction $\times$ to combine it with emp; !!P has no *spatial* specification to separate from. Instead, we use the ordinary conjunction &&.

Now, we want to prevent the `simpl` tactic from automatically unfolding listrep. This is a design choice that you might make differently, in which case, leave out the *Arguments* command.

```
Arguments listrep sigma p : simpl never.
```

10.1.3 Hint databases for spatial operators

Whenever you define a new spatial operator–a definition of type mpred such as listrep–it's useful to populate two hint databases.

- The *saturate_local* hint is a lemma that extracts pure propositional facts from a spatial fact.
- The valid_pointer hint is a lemma that extracts a valid-pointer fact from a spatial lemma.

  Consider this proof goal:

```
Lemma data_at_isptr_example1:
  ∀ (h y p : val) ,
   data_at Tsh t_list (h,y) p |- !! isptr p.
Proof.
intros.
```

isptr p means p is a non-null pointer, not NULL or Vundef or a floating-point number: `Print isptr.`

```
entailer!.
Qed.
```

```
Lemma data_at_isptr_example2:
  ∀ (h y p : val) ,
```

```
    data_at Tsh t_list (h,y) p |-- !! isptr p.
Proof.
intros.
```

Let's look more closely at how entailer! solves this goal. First, it finds all the pure propositions Prop that it can deduce from the mpred conjuncts on the left-hand side, and puts them above the line. *saturate_local.*

The *saturate_local* tactic uses a Hint database (also called *saturate_local*) to look up the individual conjuncts on the left-hand side (this particular entailment has just one conjunct). Print *HintDb saturate_local.*

In this case, the new propositions above the line are labeled *H* and *H0*.

Next, if the proof goal has just a proposition !!*P* on the right, entailer! throws away the left-hand-side and tries to prove *P*. (This is rather aggressive, and can sometimes lose information, that is, sometimes entailer! will turn a provable goal into an unprovable goal.)

```
apply prop_right.
```

It happens that field_compatible _ _ *p* implies isptr *p*, Check field_compatible_isptr.

So therefore, field_compatible_isptr solves the goal. eapply field_compatible_isptr; eauto.

Now you have some insight into how entailer! works. Qed.

But when you define a new spatial predicate mpred such as listrep, the *saturate_local* tactic does not know how to deduce Prop facts from the listrep conjunct:

```
Lemma listrep_facts_example:
 ∀ sigma p,
    listrep sigma p |-- !! (isptr p ∨ p=nullval).
Proof.
intros.
entailer!.
```

Here, entailer! threw away the left-hand-side and left an unprovable goal. Let's see why. Abort.

```
Lemma listrep_facts_example:
 ∀ sigma p,
    listrep sigma p |-- !! (isptr p ∨ p=nullval).
Proof.
intros.
```

First entailer! would use *saturate_local* to see (from the Hint database) what can be deduced from listrep *sigma p*. *saturate_local.*

But *saturate_local* did not add anything above the line. That's because there's no Hint in the Hint database for listrep. Therefore we must add one. The conventional name for such a lemma is *f_local_facts*, if your new predicate is named *f*. Abort.

```
Lemma listrep_local_facts:
  ∀ sigma p,
   listrep sigma p |--
   !! (is_pointer_or_null p ∧ (p=nullval ↔ sigma=nil)).
```

For each spatial predicate `Definition` $f(\_)$: mpred, there should be *one* “local fact”, a lemma of the form $f(\_)$ |-- !! _. On the right hand side, put all the propositions you can derive from $f(\_)$. In this case, we know:

- p is either a pointer or null (it's never Vundef, or *Vfloat*, or a nonzero Vint).
- p is null, if and only if *sigma* is nil.

`Proof.`
`intros.`
We will prove this entailment by induction on sigma *revert p*; `induction` *sigma*; `intros` *p*.
- In the base case, *sigma* is nil. We can unfold the definition of listrep to see what that means. `unfold` listrep.

Now we have an entailment with a proposition p=nullval on the left. To move that proposition above the line, we could do *Intros*, but it's easier just to call on entailer! to see how it can simplify (and perhaps partially solve) this entailment goal: *entailer*!.
`split`; `auto`.
- In the inductive case, we can again unfold the definition of listrep (a::*sigma*); but then it's good to fold listrep *sigma*. Replace the semicolon ; with a period in the next line, to see why.
`unfold` listrep; `fold` listrep.

Warning! Sometimes entailer! is too aggressive. If we use it here, it will throw away the left-hand side because it doesn't understand how to look inside an EXistential quantitier. The exclamation point ! is a warning that entailer! can turn a provable goal into an unprovable goal. Uncomment the next line and see what happens. Then put the comment marks back.

The preferred way to handle *EX y*:*t* on the left-hand-side of an entailment is to use *Intros y*. Uncomment this to try it out, then put the comment marks back.

A less agressive entailment-reducer is entailer without the exclamation point. This one never turns a provable goal into an unprovable goal. Here it will Intro the EX-bound variable y. *entailer*.

Should you use entailer! or entailer in ordinary proofs? Usually entailer! is best: it's faster, and it does more work for you. Only if you find that entailer! has gone into a dead end, should you use entailer instead.

Here it is safe to use entailer! *entailer*!.

Notice that entailer! has put several facts above the line: field_compatible t_list [] p and *value_fits* t_list (a,y) come from the *saturate_local* hint database, from the data_at conjunct; and is_pointer_or_null y and y=nullval $\leftrightarrow$ *sigma*=[] come from the the listrep conjunct, using the induction hypothesis *IHsigma*.

Now, let's split the goal and take the two cases separately. `split`; `intro`.
+
`clear` - *H H2*.
`subst` *p*.
It happens that field_compatible _ _ p implies isptr p, `Check` field_compatible_isptr.

The predicate isptr excludes the null pointer, `Print isptr.`

```
Print nullval.
```

Therefore *H* is a contradiction. We can proceed with, `Check field_compatible_nullval.`

```
  eapply field_compatible_nullval; eauto.
 +
   inversion H2.
Qed.
```

Now we add this lemma to the Hint database called *saturate_local* `Hint Resolve listrep_local_facts : saturate_local.`

Valid pointers, and the valid_pointer Hint database

In the C language, you can do a pointer comparison such as *p*!=*NULL* or *p*==*q* only if *p* is a *valid pointer*, that is, either NULL or actually pointing within an allocated object. One way to prove that *p* is valid is if, for example, data_at Tsh t_list (*h*,*y*) *p*, meaning that *p* is pointing at a list cell. There is a hint database valid_pointer from which the predicate valid_pointer *p* can be proved automatically. For example:

```
Lemma struct_list_valid_pointer_example:
  ∀ h y p,
   data_at Tsh t_list (h,y) p |-- valid_pointer p.
Proof.
  intros.
  auto with valid_pointer.
Qed.
```

However, the hint database does not know about user-defined separation-logic predicates (mpred) such as listrep; for example:

```
Lemma listrep_valid_pointer_example:
 ∀ sigma p,
   listrep sigma p |-- valid_pointer p.
Proof.
 intros.
 auto with valid_pointer.
```

Notice that `auto with`... did not solve the proof goal `Abort.`

Therefore, we should prove the appropriate lemma, and add it to the Hint database.

```
Lemma listrep_valid_pointer:
  ∀ sigma p,
   listrep sigma p |-- valid_pointer p.
Proof.
 intros.
```

The main point is to unfold listrep. `unfold listrep.`

Now we can prove it by case analysis on sigma; we don't even need induction. `destruct` *sigma*; `simpl`.
- The nil case is easy: *hint*.
  entailer!.
- The cons case *Intros y*.

Now this solves using the Hint database valid_pointer, because the data_at Tsh t_list (v,y) p on the left is enough to prove the goal. `auto with` *valid_pointer*.
`Qed`.

Now we add this lemma to the Hint database `Hint Resolve` *listrep_valid_pointer* : *valid_pointer*.

10.1.4 Specification of the reverse function.

A **funspec** characterizes the precondition required for calling the function and the postcondition guaranteed by the function. `Definition` reverse_spec : ident × **funspec** :=

```
DECLARE _reverse
 WITH sigma : list val, p: val
 PRE [ tptr t_list ]
    PROP () PARAMS (p) SEP (listrep sigma p)
 POST [ (tptr t_list) ]
   EX q:val,
    PROP () RETURN (q) SEP (listrep(rev sigma) q).
```

- The WITH clause says, there is a value *sigma*: list **val** and a value p: **val**, visible in both the precondition and the postcondition.
- The PREcondition says,
  - There is one function-parameter, whose C type is "pointer to struct list"
  - PARAMS: The parameter contains the Coq value p;
  - SEP: in memory at address p there is a linked list representing *sigma*.
- The POSTcondition says,
  - the function returns a value whose C type is "pointer to struct list"; and
  - there exists a value q: **val**, such that
  - RETURN: the function's return value is q
  - SEP: in memory at address q there is a linked list representing rev *sigma*.

The global function spec characterizes the preconditions/postconditions of all the functions that your proved-correct program will call. Normally you include all the functions here, but in this tutorial example we include only one. `Definition` Gprog : funspecs := [reverse_spec].

10.1.5 Proof of the reverse function

For each function definition in the C program, prove that the function-body (in this case, f_reverse) satisfies its specification (in this case, reverse_spec). `Lemma body_reverse: semax_body Vprog Gprog f_reverse reverse_spec.`
`Proof.`

The start_function tactic "opens up" a semax_body proof goal into a Hoare triple.

start_function.

As usual, the current assertion (precondition) is derived from the PRE clause of the function specification, reverse_spec, and the current command w=0; *...more...* is the function body of f_reverse.

The first statement (command) in the function-body is the assignment statement *w=NULL*;, where *NULL* is a C *#define* that exands to "cast 0 to void-pointer", (*void* ×)0, here ugly-printed as (tptr tvoid)(0). To apply the separation-logic assignment rule to this command, simply use the tactic forward :

forward.

The new *semax* judgment is for the rest of the function body *after* the command *w=NULL*. The precondition of this *semax* is actually the postcondition of the *w=NULL* statement. It's much like the precondition of *w=NULL*, but contains the additional LOCAL fact, temp _w (Vint (*Int.repr* 0)), that is, the variable _w contains nullval.

We can view the Hoare-logic proof of this program as a "symbolic execution", where the symbolic states are assertions. We can symbolically execute the next command by saying forward again.

forward.

Examine the precondition, and notice that now we have the additional fact, temp _v *p*.

We cannot the next step using forward ...

Fail forward.

... because the next command is a *while* loop.

10.1.6 The loop invariant

To prove a while-loop, you must supply a loop invariant, such as
(EX s1 ... PROP(...)LOCAL(...)SEP(...).

forward_while

```
(EX s1: list val, EX s2 : list val,
 EX w: val, EX v: val,
  PROP (sigma = rev s1 ++ s2)
  LOCAL (temp _w w; temp _v v)
  SEP (listrep s1 w; listrep s2 v)).
```

The forward_while tactic leaves four subgoals, which we mark with - (the Coq “bullet”)

-

hint.

On the left-hand side of this entailment is the precondition (that we had already established by forward symbolic execution to this point) for the entire while-loop. On the right-hand side is the loop invariant, that we just gave to the *forward_while* tactic. Because the right_hand side has for existentials, a good proof strategy is to choose values for them, using the *Exists* tactic.

Exists (@nil **val**) *sigma* nullval *p*.

Now we have a quantifier-free proof goal; let us see whether entailer! can solve some parts of it.

entailer!.

Indeed, the entailer! did a fine job. What’s left is a property of our user-defined listrep predicate: emp |– listrep [] nullval.

`unfold` listrep.

Now that the user-defined predicate is unfolded, entailer! can solve the residual entailment.

entailer!.

-

hint.

The second subgoal of *forward_while* is to prove that the loop-test condition can execute without crashing. Consider, for example, the C-language while loop, *while* ($a[i]>0$) ..., where the value of i might exceed the bounds of the array. Then this would be a “buffer overrun”, and is “undefined behavior” (“stuck”) in the C semantics. We must prove that, given the current precondition (in this case, the loop invariant), the loop test is not “undefined behavior.” This proof goal takes the form, *current-precondition* |– *tc_expr Delta e*, where e is the loop-test expression. You can pronounce *tc_expr* as “type-check expression”, since the Verifiable C type-checker ensures that such expressions are safe (sometimes with a subgoal for you to prove).

Fortunately, in most cases the entailer! solves *tc_expr* goals completely automatically:

entailer!.

-

hint.

As usual in any Hoare logic (including Separation Logic), we need to prove that the loop body preserves the loop invariant, more precisely,

- {Inv /\ Test} body {Inv}

where Test is the loop-test condition. Here, the loop-test condition in the original C code is (v), and its manifestation above the line is the hypothesis *HRE*: isptr v, meaning that v is a (non-null) pointer.

The loop invariant was *EX s1:_, EX s2:_, EX w:_, EX v:_, ...*, and here all the existentially quantified variables on the left side of the entailment have been moved above the line: *s1*, *s2*: **val** and *w*,*v*: **val**.

The PROP part of the loop invariant was *sigma* = rev *s1* ++ *s2*, and it has also been moved above the line, as hypothesis *H*.

So now we would like to do forward-symbolic execution through the four assignment statements in the loop body.

Fail forward.

But we cannot go forward through *t*=*v*→*tail*; because that would require a SEP conjunct in the precondition of the form data_at *sh* t_list (_,_) *v*, and there is no such conjunct. Actually, there is such a conjunct, but it is hiding inside listrep *s2 v*. That is, there is such a conjunct as long as *s2* is not nil. Let's do case analysis on s2:

`destruct` *s2* `as` [| *h r*].

+

Suppose *s2*=nil. If we unfold listrep . . . `unfold` listrep `at` 2.

then we learn that *v*=nullval. To move this fact (or any proposition) from the precondition to above-the-line, we use *Intros*:

Intros.

Now, above the line, we have *v*=nullval and isptr *v*; this is a contradiction.

`subst`. *contradiction.*

+

Suppose *s2*=*h*::*r*. We can unfold/fold the listrep conjunct for *h*::*r*; if you don't remember why we do `unfold`/`fold`, then replace the semicolon (between the fold and the unfold) with a period and see what happens.

`unfold` listrep `at` 2; `fold` listrep.

By the definition of listrep, at address *v* there must exist a value *y* and a list cell containing (h,y). So let us move *y* above the line:

Intros y.

Now we have the appropriate SEP conjuncts to be able to go forward through the loop body

forward. *forward.* *forward.* *forward.*

At the end of loop body; we must reestablish the loop invariant. The left-hand-side of this entailment is the current assertion (after the loop body); the right-hand side is simply our loop invariant. (Unfortunately, the *forward_while* tactic has "uncurried" the existentials into a single EX that binds a 4-tuple.) Since the proof goal is a complicated-looking entailment, let's see if entailer! can simplify it a bit:

entailer!.

Now, we can provide new values for *s1*,*s2*,*w*,*v* to instantiate the four existentials; these are, respectively, *h*::*s1*,*r*,*v*,*y*.

Exists (*h*::*s1* ,*r* ,*v* ,*y*).

Again, we have a complicated-looking entailment; we ask entailer! to reduce it some more.

entailer!.
× simpl. rewrite app_ass. auto.
× unfold listrep at 3; fold listrep.
Exists w. *entailer*!.

-

As usual in any Hoare logic (including Separation Logic), the postcondition of a while-loop is {Inv /\ not Test}, where Inv is the loop invariant and Test is the loop test. Here, all the EXistentials and PROPs of the loop invariant have been moved above the line as *s1*,*s2*,*w*,*v*,*HRE*,*H*.

We can always go forward through a return statement: *forward*.

Exists w; *entailer*!.
rewrite (proj1 *H1*) by auto.
unfold listrep at 2; fold listrep.
entailer!.
rewrite ← app_nil_end, rev_involutive.
auto.
Qed.

10.1.7 Why separation logic?

If we review our functional correctness proof for *reverse.c*, it may not be obvious why we need separation logic at all. Let's take a close look.

First, we build "separation" into the definition of listrep. The following is our definition:

Fixpoint listrep (sigma: list val) (p: val) : mpred := match sigma with | h::hs => EX y:val, data_at Tsh t_list (h,y) p * listrep hs y | nil => !! (p = nullval) && emp end.

In the nonempty list case, the head element is described by

data_at Tsh t_list (h,y) p

which is separated (by the separating conjunction *) from the rest of the list

listrep hs y.

This separation ensures that no address could be used for more than once in a linked list. For example, considering a linked list of length at least 2,

listrep (a :: b :: l) x.

We know that there must be two addresses y and z such that

data_at Tsh t_list (a,y) x * data_at Tsh t_list (b,z) y * listrep l z.

The "separating conjunction" × tells us that x and y must be different! Formally, we can prove the following two lemmas:

```
Lemma listrep_len_ge2_fact: ∀ (a b x: val) (l: list val),
  listrep (a :: b :: l) x |-
  EX y: val, EX z: val,
      data_at Tsh t_list (a,y) x ×
      data_at Tsh t_list (b,z) y ×
      listrep l z.
Proof.
  intros.
  unfold listrep; fold listrep.
  Intros y z.
  Exists y z.
  cancel.
Qed.

Lemma listrep_len_ge2_address_different: ∀ (a b x y z: val) (l: list val),
  data_at Tsh t_list (a,y) x ×
  data_at Tsh t_list (b,z) y ×
  listrep l z |-
  !! (x ≠ y).
Proof.
  intros.
```

To prove that the addresses are different, we do case analysis first. If $x = y$, we use the following theorem: `Check data_at_conflict.`

It says that we can derives address anti-aliasing from the "separation" defined by ×. If $x \neq y$, the right side is already proved. `destruct (Val.eq x y); [| apply prop_right; auto].`

```
  subst x.
  sep_apply (data_at_conflict Tsh t_list (a, y)).
  + auto.
  + entailer!.
Qed.
```

Actually, even the property $x \neq y$ is not strong enough! We need to know that x does not overlap with *any field* of record y, for example (in C notation) x != &(y→*tail*) and &(x→*tail*) != y. Otherwise, when storing into y→*tail*, we couldn't know that x→*head* is not altered.

Without separation logic, we could still define *listrep'* using extra clauses for address anti-aliasing. For example, a length-3 linked list listrep (a :: b :: c :: nil) x can be: exists y and z, such that (a, y) is stored at x, (b, z) is stored at y, (c, nullval) is stored at z and x, y and z are different from each other. In general, that assertion will be quadratically long (as a function of the length of the linked list). Then, to make sure x→*head* is not at the same address as y→*tail*, we'd need even more assertions.

In our program correctness proof, we do (implicitly) use the fact that different *SEP* clauses describe disjoint heaplets. Here is an intermediate step in the proof of body_reverse.

(We rarely state intermediate proof goals such as this one. We do it here to illustrate a point about separating conjunction.

```
Lemma body_reverse_step: ∀
  {Espec : OracleKind}
  (sigma : list val)
  (s1 : list val)
  (h : val)
  (r : list val)
  (w v : val)
  (HRE : isptr v)
  (H : sigma = rev s1 ++ h :: r)
  (y : val),
  semax (func_tycontext f_reverse Vprog Gprog nil)
    (PROP ( )
     LOCAL (temp _t y; temp _w w; temp _v v)
     SEP (listrep s1 w; data_at Tsh t_list (h, y) v; listrep r y))
    (Ssequence
       (Sassign
          (Efield
             (Ederef (Etempvar _v (tptr (Tstruct _list noattr)))
                        (Tstruct _list noattr))
             _tail (tptr (Tstruct _list noattr)))
          (Etempvar _w (tptr (Tstruct _list noattr))))
       (Ssequence (Sset _w (Etempvar _v (tptr (Tstruct _list noattr))))
                  (Sset _v (Etempvar _t (tptr (Tstruct _list noattr))))))
    (normal_ret_assert
       (PROP ( )
        LOCAL (temp _v y; temp _w v; temp _t y)
        SEP (listrep s1 w; data_at Tsh t_list (h, w) v; listrep r y))).
Proof.
  intros.
  abbreviate_semax.
```

Now, our proof goal is:

```
semax Delta
  (PROP ( )
   LOCAL (temp _t y; temp _w w; temp _v v)
   SEP (listrep s1 w; data_at Tsh t_list (h, y) v; listrep r y))
  ((_v → _tail) = _w; MORE_COMMANDS)
  POSTCONDITION.
```

The next forward tactic will do symbolic execution of $v \rightarrow tail = w$. *forward.* `Abort.`

When C programs manipulate pointer data structures (or slices of arrays), address anti-aliasing plays an important role in their correctness proofs. Separation logic is essential for reasoning about updates to these structures. Verifiable C's SEP clause ensures separation between all its conjuncts.

Chapter 11

Library VC.Verif_stack

11.1 Verif_stack: Stack ADT implemented by linked lists

Here is a little C program, *stack.c*

11.1.1 Let's verify!

```
Require VC.Preface. Require Import VST.floyd.proofauto.
Require Import VST.floyd.library.
Require Import VC.stack.
Instance CompSpecs : compspecs. make_compspecs prog. Defined.
Definition Vprog : varspecs. mk_varspecs prog. Defined.
Require Import VC.hints.
```

11.1.2 Malloc and free

When you use C's malloc/free library, you write p=*malloc*(n); to get a pointer p to a block of n bytes; when you're done with that block, you call *free*(p) to dispose of it. How does the *free* function know how many bytes to dispose?

The answer is, the malloc/free library puts an extra "header" field just before address p, so really you get this:

+———–+ | header | +———–+ p–>| zero | +———–+ | one | +———–+ | two | +———–+

where in this case, *header*=3.

In separation logic, we can describe this as

- *malloc_token* Ews p × data_at Ews (Tstruct _*mystruct* noattr) (zero,*one*,*two*) p

where *malloc_token* Ews p describes this picture:

+———–+ | header | +———–+ p–>

Of course, the malloc/free library might have a different way of "remembering" the size that p points to, so its representation of *malloc_token* is *not necessarily* a word at offset -1. Therefore, clients of the malloc/free library treat *malloc_token* as an abstract predicate. Now, the function-specifications of malloc and free are something like this:

```
Definition malloc_spec_example :=
 DECLARE _malloc
 WITH t:type
 PRE [ tuint ]
    PROP (0 ≤ sizeof t ≤ Int.max_unsigned;
          complete_legal_cosu_type t = true;
          natural_aligned natural_alignment t = true)
    PARAMS (Vint (Int.repr (sizeof t)))
    SEP ()
 POST [ tptr tvoid ] EX p:_,
    PROP ()
    RETURN (p)
    SEP (if eq_dec p nullval then emp
         else (malloc_token Ews t p × data_at_ Ews t p)).
Definition free_spec_example :=
 DECLARE _free
 WITH t: type, p:val
 PRE [ tptr tvoid ]
     PROP ()
     PARAMS (p)
     SEP (malloc_token Ews t p; data_at_ Ews t p)
 POST [ Tvoid ]
     PROP () RETURN () SEP ().
```

If your source program says *malloc*(sizeof(t)), your *forward_call* should supply (as a WITH-witness) the C type t. Malloc may choose to return NULL, in which case the SEP part of the postcondition is emp, or it may return a pointer, in which case you get data_at_ Ews t p, and as a free bonus you get a *malloc_token* Ews t p. But don't lose that *malloc_token*! You will need to supply it later to the *free* function when you dispose of the object.

The SEP predicate data_at_ Ews t p is an *uninitialized* structure of type t. It is equivalent to, data_at Ews t (default_val t) p. The default_val is basically a struct or array full of Vundef values.

11.1.3 Specification of linked lists

This is much like the linked lists in Verif_reverse.

```
Fixpoint listrep (il: list Z) (p: val) : mpred :=
 match il with
 | i::il' ⇒ EX y: val,
```

```
        malloc_token Ews (Tstruct _cons noattr) p ×
        data_at Ews (Tstruct _cons noattr) (Vint (Int.repr i),y) p ×
        listrep il' y
 | nil ⇒ !! (p = nullval) && emp
 end.
```

Proof automation for user-defined separation predicates works better if you disable automatic simplification, as follows: *Arguments* listrep *il* *p* : `simpl` *never*.

As usual, we should populate the Hint databases *saturate_local* and valid_pointer

Exercise: 1 star, standard (stack_listrep_properties) Lemma listrep_local_prop: ∀ *il* *p*, listrep *il* *p* |-
    !! (is_pointer_or_null *p* ∧ (*p*=nullval ↔ *il*=nil)).

See if you can remember how to prove this; or look again at Verif_reverse to see how it's done. *Admitted.*

`Hint Resolve` *listrep_local_prop* : *saturate_local.*

```
Lemma listrep_valid_pointer:
  ∀ il p,
   listrep il p |- valid_pointer p.
```

See if you can remember how to prove this; or look again at Verif_reverse to see how it's done. *Admitted.*

`Hint Resolve` *listrep_valid_pointer* : *valid_pointer.*

□

11.1.4 Specification of stack data structure

Our stack data structure looks like this:

+———–+ | token | +———–+ +——— p->| top——+—q->| linked list... +———
–+ +———

The stack object *p* points to a header node with one field *top* (plus a malloc token); the *contents* of the *top* field is some pointer *q* that points to a linked list.

```
Definition stack (il: list Z) (p: val) :=
 EX q: val,
  malloc_token Ews (Tstruct _stack noattr) p ×
  data_at Ews (Tstruct _stack noattr) q p ×
  listrep il q.
```

Arguments stack *il* *p* : `simpl` *never*.

Exercise: 1 star, standard (stack_properties) Lemma stack_local_prop: ∀ *il* *p*, stack *il* *p* |- !! (isptr *p*).

Admitted.

`Hint Resolve` *stack_local_prop* : *saturate_local.*

```
Lemma stack_valid_pointer:
  ∀ il p,
   stack il p |- valid_pointer p.
   Admitted.
Hint Resolve stack_valid_pointer : valid_pointer.
```
□

11.1.5 Function specifications for the stack operations

```
Definition newstack_spec : ident × funspec :=
 DECLARE _newstack
 WITH gv: globals
 PRE [ ]
    PROP () PARAMS() GLOBALS(gv) SEP (mem_mgr gv)
 POST [ tptr (Tstruct _stack noattr) ]
    EX p: val, PROP ( ) RETURN (p) SEP (stack nil p; mem_mgr gv).

Definition push_spec : ident × funspec :=
 DECLARE _push
 WITH p: val, i: Z, il: list Z, gv: globals
 PRE [ tptr (Tstruct _stack noattr), tint ]
    PROP (Int.min_signed ≤ i ≤ Int.max_signed)
    PARAMS (p; Vint (Int.repr i)) GLOBALS(gv)
    SEP (stack il p; mem_mgr gv)
 POST [ tvoid ]
    PROP ( ) RETURN () SEP (stack (i::il) p; mem_mgr gv).

Definition pop_spec : ident × funspec :=
 DECLARE _pop
 WITH p: val, i: Z, il: list Z, gv: globals
 PRE [ tptr (Tstruct _stack noattr) ]
    PROP ()
    PARAMS (p) GLOBALS(gv)
    SEP (stack (i::il) p; mem_mgr gv)
 POST [ tint ]
    PROP ( ) RETURN (Vint (Int.repr i)) SEP (stack il p; mem_mgr gv).
```

Putting all the funspecs together:

```
Definition Gprog : funspecs :=
        ltac:(with_library prog [
                   newstack_spec; push_spec; pop_spec
 ]).
```

11.1.6 Proofs of the function bodies

An *Abstract Data Type* (ADT) is a type provided with a *representation* and a set of *operations*. Clients of the ADT never see the representation, they only call upon the operations. Implementations of the operations do need to manipulate the representation directly.

In this case, stack is our ADT. The operations are *newstack*, *push*, and *pop*. Clients of these operations see only stack *il* *p*, where *il* is the list of values that the client has pushed onto the stack, and *p* is the client's "handle", the address of the representation of the stack. The client does not know whether the abstract list *il* is represented in C data structures by a singly linked list, a doubly linked list, an array, or some other data structure. The client *never unfolds* the Definition stack.

The operations *newstack*, *push*, *pop* are implemented in C, and they directly manipulate (in this case) a singly linked list. In proving the correctness of *newstack*, *push*, *pop*, we need to know the representation. Therefore,

Hint: At the beginning of body_pop, of body_push, and of body_newstack, the first thing you should do is `unfold stack in *`.

Exercise: 2 stars, standard (body_pop) Lemma body_pop: semax_body Vprog Gprog f_pop pop_spec.

```
Proof.
start_function.
    Admitted.
```

☐

Exercise: 2 stars, standard (body_push) Lemma body_push: semax_body Vprog Gprog f_push push_spec.

```
Proof.
start_function.
forward_call (Tstruct _cons noattr, gv).
simpl; split3; auto.
    Admitted.
```

☐

Exercise: 2 stars, standard (body_newstack) Lemma body_newstack: semax_body Vprog Gprog f_newstack newstack_spec.

```
Proof.
start_function.
    Admitted.
```

☐

Chapter 12

Library VC.Verif_triang

12.1 Verif_triang: A client of the stack functions

```
Require VC.Preface. Require Import VST.floyd.proofauto.
Require Import VST.floyd.library.
Require Import VC.stack.
Instance CompSpecs : compspecs. make_compspecs prog. Defined.
Definition Vprog : varspecs. mk_varspecs prog. Defined.
```

Here are some functions (in *stack.c*) that are clients of the stack ADT. First, push the numbers 1,2,...,n onto a stack, then pop the numbers off the stack and add them up. This computes the nth triangular number, 1+2+...+n = n(n+1)/2.

void push_increasing (struct stack *st, int n) { int i; i=0; while (i<n) { i++; push(st,i); } }

int pop_and_add (struct stack *st, int n) { int i=0; int t, s=0; while (i<n) { t=pop(st); s += t; i++; } return s; }

int main (void) { struct stack *st; int i,t,s; st = newstack(); push_increasing(st, 10); s = pop_and_add(st, 10); return s; }

Let's verify this program!

12.1.1 Proofs with integers

The natural numbers have arithmetic axioms that are not very nice. For example, you might expect that *a*-*b*+*b*=*a*, but that's not true: `Lemma nat_sub_add_yuck:`

```
  ¬ (∀ a b: nat, a-b+b=a)%nat.
Proof.
intros.
intro.
specialize (H 0 1)%nat.
simpl in H. inversion H.
Qed.
```

This just shows that if the negative numbers did not exist, it would be necessary to construct them! In reasoning about programs, as in many other kinds of mathematics, we should use the integers. In Coq the type is called **Z**. Lemma Z_sub_add_ok:
∀ a b : **Z**, a-b+b=a.
Proof. intros. *lia.* Qed.

The Z type does have an inductive definition . . . Print **Z**.

Let's consider a recursive function on Z, the function that turns 5 into the list 5::4::3::2::1::nil. In the natural numbers, that's easy to define:

```
Fixpoint decreasing_nat (n: nat) : list nat :=
 match n with S n' ⇒ n :: decreasing_nat n' | O ⇒ nil end.
```

But in the integers **Z**, we cannot simply pattern-match on successor ... *Fail* Fixpoint decreasing_Z (n: **Z**) : **list Z** :=
match n with *Z.succ n'* ⇒ n :: *decreasing_Z n'* | 0 ⇒ *nil* end.

... because Z.succ is a function, not a constructor.

There are two ways we might define a function to produce a decreasing list of Z. First, we might use *Z.of_nat* and *Z.to_nat*:

```
Fixpoint decreasing_Z1_aux (n: nat) : list Z :=
  match n with
  | S n' ⇒ Z.of_nat n :: decreasing_Z1_aux n'
  | O ⇒ nil
  end.
Definition decreasing_Z1 (n: Z) : list Z :=
     decreasing_Z1_aux (Z.to_nat n).
```

This will work, but in doing proofs the frequent conversion between Z and nat will be awkward. If possible, we'd like to stay in the integers as much as possible. So here's another way:

```
Check Z_gt_dec.
```

```
Function decreasing (n: Z) {measure Z.to_nat n}:=
 if Z_gt_dec n 0 then n :: decreasing (n-1) else nil.
Proof.
```

When you define a Function, you must provide a measure, that is, a function from your argument-type (in this case Z) to the natural numbers, and then you must prove that each recursive call within the function body decreases the measure. In this ecase, there's only one recursive call, so there's just one proof obligation: show that if n>0 then *Z.to_nat* (n-1) < *Z.to_nat n*. *lia.*
Defined.

Exercise: 2 stars, standard (Zinduction) Coq's standard induction principle for Z is not the one we usually want, so let us define a more natural induction scheme: Lemma Zinduction: ∀ (P: **Z** → Prop),

```
  P 0 →
  (∀ i, 0 < i → P (i-1) → P i) →
  ∀ n, 0 ≤ n → P n.
Proof.
intros.
rewrite ← (Z2Nat.id n) in * by lia.
set (j := Z.to_nat n) in *. clearbody j.
Check inj_S. Print Z.succ. Admitted.
```

□

A theorem about the nth triangular number

```
Definition add_list: list Z → Z := fold_right Z.add 0.
```

Exercise: 2 stars, standard (add_list_decreasing) Theorem: the sum of the list $(n)::(n\text{-}1)::$ ... :: 2::1 is $n*(n{+}1)/2$.

```
Lemma add_list_decreasing_eq_alt: ∀ n,
  0 ≤ n →
  (2 × (add_list (decreasing n)))%Z = (n × (n+1))%Z.
Proof.
  intros.
  pattern n; apply Zinduction.
  - reflexivity.
  - intros.
```

WARNING! When using functions defined by `Function`, don't unfold them! Temporarily remove the brackets from the next line to see what happens!

Instead of unfolding `decreasing` we use the equation that Coq automagically defines for the Function. Try the command `Search decreasing.` to see all the reasoning principles that Coq defined for the new `Function`. We will use this one: `Check decreasing_equation.`

```
rewrite decreasing_equation.
```

during the proof of this lemma, you may find the *ring_simplify* tactic useful. Read about it in the Coq reference manual. Basically, it takes formulas with multiplication and addition, and simplifies them. But you can do this without *ring_simplify*, using just ordinary rewriting with lemmas about Z.add and Z.mul. *Admitted.*

```
Lemma add_list_decreasing_eq: ∀ n,
  0 ≤ n →
  add_list (decreasing n) = n × (n+1) / 2.
Proof.
  intros.
  apply Z.div_unique_exact.
  Admitted.
```

□

Definitions copied from Verif_stack.v

We repeat here some material from Verif_stack.v. Normally we would break the .c file into separate modules, and do our Verifiable C proofs in separate modules; but for this example we leave out the modules. Just skip down to "End of the material repeated from Verif_stack.v".

Specification of linked lists in separation logic

```
Fixpoint listrep (il: list Z) (p: val) : mpred :=
 match il with
 | i::il' ⇒ EX y: val,
         malloc_token Ews (Tstruct _cons noattr) p ×
         data_at Ews (Tstruct _cons noattr) (Vint (Int.repr i),y) p ×
         listrep il' y
 | nil ⇒ !! (p = nullval) && emp
 end.

Lemma listrep_local_prop: ∀ il p, listrep il p |-
        !! (is_pointer_or_null p ∧ (p=nullval ↔ il=nil)).
Proof.
induction il; intro; simpl.
entailer!. intuition.
Intros y.
entailer!.
split; intros. subst.
eapply field_compatible_nullval; eauto.
inversion H3.
Qed.
Hint Resolve listrep_local_prop : saturate_local.

Lemma listrep_valid_pointer:
  ∀ il p,
   listrep il p |- valid_pointer p.
Proof.
   Admitted.
Hint Resolve listrep_valid_pointer : valid_pointer.
```

Specification of stack data structure

```
Definition stack (il: list Z) (p: val) :=
 EX q: val,
  malloc_token Ews (Tstruct _stack noattr) p ×
  data_at Ews (Tstruct _stack noattr) q p × listrep il q.

Lemma stack_local_prop: ∀ il p, stack il p |- !! (isptr p).
Proof.
```

```
    Admitted.
Hint Resolve stack_local_prop : saturate_local.

Lemma stack_valid_pointer:
  ∀ il p,
    stack il p |-- valid_pointer p.
Proof.
    Admitted.
Hint Resolve stack_valid_pointer : valid_pointer.

Definition newstack_spec : ident × funspec :=
 DECLARE _newstack
 WITH gv: globals
 PRE [ ]
    PROP () PARAMS() GLOBALS(gv) SEP (mem_mgr gv)
 POST [ tptr (Tstruct _stack noattr) ]
    EX p: val, PROP ( ) RETURN (p) SEP (stack nil p; mem_mgr gv).

Definition push_spec : ident × funspec :=
 DECLARE _push
 WITH p: val, i: Z, il: list Z, gv: globals
 PRE [ tptr (Tstruct _stack noattr), tint ]
    PROP (Int.min_signed ≤ i ≤ Int.max_signed)
    PARAMS (p; Vint (Int.repr i)) GLOBALS(gv)
    SEP (stack il p; mem_mgr gv)
 POST [ tvoid ]
    PROP ( ) RETURN() SEP (stack (i::il) p; mem_mgr gv).

Definition pop_spec : ident × funspec :=
 DECLARE _pop
 WITH p: val, i: Z, il: list Z, gv: globals
 PRE [ tptr (Tstruct _stack noattr) ]
    PROP ()
    PARAMS (p) GLOBALS(gv)
    SEP (stack (i::il) p; mem_mgr gv)
 POST [ tint ]
    PROP ( ) RETURN (Vint (Int.repr i)) SEP (stack il p; mem_mgr gv).
```

(End of the material repeated from Verif_stack.v)

12.1.2 Specification of the stack-client functions

Spend a few minutes studying these funspecs, and compare to the implementations in stack.c, until you understand why these might be appropriate specifications.

```
Definition push_increasing_spec :=
 DECLARE _push_increasing
```

```
 WITH st: val, n: Z, gv: globals
 PRE [ tptr (Tstruct _stack noattr), tint ]
   PROP (0 ≤ n ≤ Int.max_signed)
   PARAMS (st; Vint (Int.repr n)) GLOBALS(gv)
   SEP (stack nil st; mem_mgr gv)
 POST [ tvoid ]
   PROP() RETURN() SEP (stack (decreasing n) st; mem_mgr gv).
Definition pop_and_add_spec :=
 DECLARE _pop_and_add
 WITH st: val, il: list Z, gv: globals
 PRE [ tptr (Tstruct _stack noattr), tint ]
   PROP (Zlength il ≤ Int.max_signed;
         Forall (Z.le 0) il;
         add_list il ≤ Int.max_signed)
   PARAMS (st; Vint (Int.repr (Zlength il))) GLOBALS(gv)
   SEP (stack il st; mem_mgr gv)
 POST [ tint ]
   PROP()
   RETURN (Vint (Int.repr (add_list il)))
   SEP (stack nil st; mem_mgr gv).
Definition main_spec :=
 DECLARE _main
 WITH gv: globals
 PRE [ ] main_pre prog tt gv
 POST [ tint ]
   PROP( ) RETURN (Vint (Int.repr 55)) SEP( TT ).
```

Putting all the funspecs together

```
Definition Gprog : funspecs :=
        ltac:(with_library prog [
                   newstack_spec; push_spec; pop_spec;
                   push_increasing_spec; pop_and_add_spec; main_spec
 ]).
```

12.1.3 Proofs of the stack-client function-bodies

Exercise: 3 stars, standard (body_push_increasing) Lemma body_push_increasing: semax_body Vprog Gprog
f_push_increasing push_increasing_spec.

Admitted.

☐

Exercise: 2 stars, standard (add_list_lemmas) Lemma add_list_app:

```
  ∀ al bl, add_list (al++bl) = add_list al + add_list bl.
   Admitted.

Lemma add_list_nonneg:
 ∀ il,
  Forall (Z.le 0) il →
  0 ≤ add_list il.
   Admitted.
```

□

Exercise: 2 stars, standard (add_list_sublist_bounds) Lemma add_list_sublist_bounds:

```
 ∀ lo hi K il,
  0 ≤ lo ≤ hi →
  hi ≤ Zlength il →
  Forall (Z.le 0) il →
  0 ≤ add_list il ≤ K →
  0 ≤ add_list (sublist lo hi il) ≤ K.
Proof.
```

Hint: you don't need induction. Useful lemmas are, sublist_same, sublist_split, add_list_nonneg, add_list_app, *Forall_sublist*, and use the *hint* tactic to learn when the *list_solve* tactic will be useful. *Admitted.*

□

Exercise: 3 stars, standard (add_another) Suppose we have a list *il* of integers, il = [5;4;3;2;1], with Znth 0 $il = 5$, Znth 4 $il = 1$, and Zlength $il = 5$, and we want to add them all up, 5+4+3+2+1=15. Suppose we've already added up the first i of them (let i=2 for example), that is, 5+4=9, and we want to add the next one, that is, the *ith* one. That is, we want to add 9+3. How do we know that won't overflow the range of C-language signed integer arithmetic?

The proof goes: Every element of the list is nonnegative; the whole list adds up to a number <= Int.max_signed; and any sublist of an all-nonnegative list adds up to less-or-equal to the total of the whole list.

```
Lemma add_another:
 ∀ il,
   Forall (Z.le 0) il →
   add_list il ≤ Int.max_signed →
  ∀ i : Z,
  0 ≤ i < Zlength il →
  Int.min_signed ≤ Int.signed (Int.repr (add_list (sublist 0 i il))) +
                 Int.signed (Int.repr (Znth i il)) ≤ Int.max_signed.
Proof.
intros.
assert (0 ≤ add_list il). {
```

```
  admit.
}
 assert (0 ≤ add_list (sublist 0 i il) ≤ Int.max_signed). {
  admit.
 }
 assert (H4: 0 ≤ add_list (sublist 0 (i+1) il) ≤ Int.max_signed). {
  admit.
 }
 assert (0 ≤ Znth i il ≤ Int.max_signed). {
  replace (Znth i il) with (add_list (sublist i (i+1) il)).
  -
   admit.
  -
   admit.
}
```

Next: *Int.signed* (*Int.repr* (add_list (sublist 0 *i* *il*))) = add_list (sublist 0 *i* *il*). To prove that, we'll use *Int.signed_repr*: `Check Int.signed_repr.`

`rewrite Int.signed_repr by rep_lia.`

rep_lia is just like *lia*, but it also knows the numeric values of representation-related constants such as Int.min_signed. `rewrite Int.signed_repr by rep_lia.`

```
rewrite (sublist_split 0 i (i+1)) in H4 by list_solve.
rewrite add_list_app in H4.
rewrite sublist_len_1 in H4 by list_solve.
simpl in H4.
rep_lia.
   Admitted.
```

□

Exercise: 3 stars, standard (body_pop_and_add) `Lemma body_pop_and_add: semax_body Vprog Gprog f_pop_and_add pop_and_add_spec.`

```
Proof.
start_function.
forward.
forward.
forward_while (EX i:Z,
   PROP(0 ≤ i ≤ Zlength il)
   LOCAL (temp _st st;
          temp _i (Vint (Int.repr i));
          temp _n (Vint (Int.repr (Zlength il)));
         gvars gv)
   SEP (stack (sublist i (Zlength il) il) st; mem_mgr gv)).
```

+
admit.
+
entailer!.
+
forward_call (*st* , Znth *i il* , sublist (*i*+1) (Zlength *il*) *il* , *gv*).

This *forward_call* couldn't quite figure out the "Frame" for the function call. That is, it couldn't match up stack (sublist *i* (Zlength *il*) *il*) *st* with

stack (Znth i il :: sublist (i + 1) (Zlength il) il) st.

You have to help, by doing some rewrites with sublist_split, sublist_len_1 that prove

sublist i (Zlength il) il = Znth i il :: sublist (i+1) (Zlength il) il.

When you've rewritten the goal into,

stack (Znth i il :: sublist (i + 1) (Zlength il) il) st |-- stack (Znth i il :: sublist (i + 1) (Zlength il) il) st * fold_right_sepcon Frame

then just do *cancel.* *admit.*

And now we are ready to go forward through the C statement _s = _s + _t; *Fail forward.*

oops! we can't go forward through _s = _s + _t; because we forgot to mention temp _s in the loop invariant! Time to start over.

By the way, this statement _s = _s + _t is exactly where forward will ask you to prove a subgoal in which you can use lemma add_another. `Abort`.

Into this lemma, paste in the failed proof just above, but adjust the loop invariant: add a LOCAL assertion for _s. Lemma body_pop_and_add: semax_body Vprog Gprog f_pop_and_add pop_and_add_spec.
`Proof`.

Hint: choose the loop invariant for temp _s ??? in such a way that you can make use of Lemma add_another. *Admitted.*

□

Exercise: 3 stars, standard (body_main) Lemma body_main: semax_body Vprog Gprog f_main main_spec.
`Proof`.
start_function.

We assume that *triang.c* is linked with an implementation of malloc/free. That assumption is expressed by the *create_mem_mgr* axiom, which we can *sep_apply* here. On the other hand, if we want a complete verified system including libraries, then instead of importing floyd.library we would actually link with a malloc/free implementation, but that's beyond the scope of this chapter. *sep_apply* (*create_mem_mgr* *gv*).

You can see that this has produced the SEP conjunct *mem_mgr* *gv*, which is useful to satisfy the precondition of *newstack*, *push*, *pop*, etc. Now you can finish this proof.

Admitted.

□

Chapter 13

Library VC.Verif_append1

13.1 Verif_append1: List segments

Here is a little C program, *append.c*

```
Require VC.Preface.
```

```
Require Import VST.floyd.proofauto.
Require Import VC.append.
Instance CompSpecs : compspecs. make_compspecs prog. Defined.
Definition Vprog : varspecs. mk_varspecs prog. Defined.
```

13.1.1 Specification of the append function.

Here we just copy what we have defined in Verif_reverse

```
Definition t_list := Tstruct _list noattr.
```

```
Fixpoint listrep (sigma: list val) (p: val) : mpred :=
 match sigma with
 | h::hs ⇒
    EX y:val,
      data_at Tsh t_list (h,y) p × listrep hs y
 | nil ⇒
    !! (p = nullval) && emp
 end.
```

```
Arguments listrep sigma p : simpl never.
```

Then we can easily describe the functionality of this append.

```
Definition append_spec :=
 DECLARE _append
  WITH x: val, y: val, s1: list val, s2: list val
  PRE [ tptr t_list , tptr t_list]
     PROP()
```

```
      PARAMS (x; y)
      SEP (listrep s1 x; listrep s2 y)
   POST [ tptr t_list ]
     EX r: val,
      PROP()
      RETURN(r)
      SEP (listrep (s1++s2) r).
Definition Gprog : funspecs := [ append_spec ].
```

13.1.2 List segments.

When verifying this program, a critical step is to figure out a correct loop invariant. If we try to simulate this program, especially the loop in it, we may want a loop invariant which can be illustrated by the following diagram. (The following diagram is best demonstrated with a monospaced font.)

+—+—+ +—+—+ +—+—+ +—+—+ x ==> | | ===> ... ===> | | t ==> | b | u ==> | | ===> ... +—+—+ +—+—+ +—+—+ +—+—+

| <========== s1a ==========> | (b) | <===== s1c =====> | | <================== s1 ============================> |

+—+—+ +—+—+ +—+—+ y ==> | | ===> | | ==> | | ===> ... +—+—+ +—+—+ +—+—+

| <================= s2 ==================> |

To describe this loop invariant, we need a separation logic predicate to describe the partial linked list from address x to address t with contents $s1a$. This must a new predicate different from listrep because listrep describes linked lists ending with $NULL$ which is not the case here. We call this new predicate lseg, pronounced "list-segment".

```
Fixpoint lseg (contents: list val) (x z: val) : mpred :=
  match contents with
  | nil ⇒ !! (x = z) && emp
  | h::hs ⇒ EX y:val, data_at Tsh t_list (h, y) x × lseg hs y z
  end.
```

Arguments lseg *contents x z* : simpl *never*.

Now, we can prove some useful properties about lseg.

Exercise: 1 star, standard (singleton_lseg) Lemma singleton_lseg: ∀ (a: val) (x y: val),

```
  data_at Tsh t_list (a, y) x |-- lseg [a] x y.
Proof.
    Admitted.
```

☐

It is critical to observe that a partial linked list defined by lseg s x y may have a loop.

For example, the following diagram does satisfy lseg $[a;\ b]\ x\ y$. (The following diagram is best demonstrated with a monospaced font.)

+—+—+ +—+—+ x ==> | a | y ==> | b | y ====+ +—+—+ +- -+—+ | ^ | | | +=========+

We can prove this formally.

```
Lemma lseg_maybe_loop: ∀ (a b x y: val),
  data_at Tsh t_list (a, y) x × data_at Tsh t_list (b, y) y
  |- lseg [a; b] x y.
Proof.
  intros.
  unfold lseg.
  Exists y.
  Exists y.
  entailer!.
Qed.
```

Is our definition of lseg wrong? The answer is no because a loopy lseg cannot connect to a nonempty linked list. For instance, we can prove that

data_at Tsh t_list (a, y) x * data_at Tsh t_list (b, y) y * listrep *c* y

will lead to a contradication. Here, the first two separating conjuncts build a loopy lseg and the third separating conjunct is a nonempty listrep.

```
Lemma loopy_lseg_not_bad: ∀ (a b c x y: val),
  data_at Tsh t_list (a, y) x × data_at Tsh t_list (b, y) y × listrep [c] y
    |- FF.
Proof.
  intros.
  unfold listrep.
  Intros u.
  subst.
Check (data_at_conflict Tsh t_list (c, nullval)).
  sep_apply (data_at_conflict Tsh t_list (c, nullval)).
  + auto.
  + entailer!.
Qed.
```

Important note! The proof above demonstrates the use of the *sep_apply* tactic. Step through that part of the proof to see what *sep_apply* does.

Now we can prove the following theorems about partial linked lists and complete linked lists.

Exercise: 1 star, standard (lseg_lseg)

```
Lemma lseg_lseg: ∀ (s1 s2: list val) (x y z: val),
  lseg s1 x y × lseg s2 y z |- lseg (s1 ++ s2) x z.
Proof.
```

Admitted.
☐

Exercise: 1 star, standard (lseg_list) Lemma lseg_list: ∀ (*s1* *s2*: list val) (*x* *y*: val),
lseg *s1* *x* *y* × listrep *s2* *y* |- listrep (*s1* ++ *s2*) *x*.
Proof.
Admitted.
☐

Is it possible to define lseg in a different way so that loopy situations can be banned? Yes. We discuss this near the end of the chapter.

13.1.3 Proof of the append function

Before verifying the functional correctness of append, we still need to add lemmas to hint databases for separation logic predicates. Readers may copy proofs from Verif_reverse or just skip down to "End of the material".

Lemma listrep_local_facts:
∀ *sigma* *p*,
listrep *sigma* *p* |-
!! (is_pointer_or_null *p* ∧ (*p*=nullval ↔ *sigma*=nil)).
Proof.
Admitted.
Hint Resolve *listrep_local_facts* : *saturate_local.*

Lemma listrep_valid_pointer:
∀ *sigma* *p*,
listrep *sigma* *p* |- valid_pointer *p*.
Proof.
Admitted.
Hint Resolve *listrep_valid_pointer* : *valid_pointer.*

(End of the material repeated from Verif_reverse.v)

In C programs, we test whether the head pointer of a linked list is null to determine whether that list is empty or not. Thus, from a separating conjunct listrep *contents* x, it is useful to prove *contents* = nil (or *contents* ≠ nil) when knowing that x = nullval (or $x \neq$ nullval). The following two lemmas state such correlation. They will be used several times in the C function append's correctness proof.

Exercise: 1 star, standard (listrep_null) Lemma listrep_null: ∀ *contents* *x*,
x = nullval →
listrep *contents* *x* = !! (*contents*=nil) && emp.
Proof.

Hint: One way to prove P=Q, where P and Q are mpreds, is to apply pred_ext and then prove P|−Q and Q|−P. *Admitted.*

□

Exercise: 1 star, standard (listrep_nonnull) Lemma listrep_nonnull: ∀ *contents* *x*,
$x \neq$ nullval →
listrep *contents* *x* =
EX *h*: **val**, EX *hs*: list **val**, EX *y*:**val**,
!! (*contents* = *h* :: *hs*) && data_at Tsh t_list (*h*, *y*) *x* × listrep *hs* *y*.

Proof.

Again, pred_ext will be useful here. *Admitted.*

□

Now, let's prove this append function correct.

Exercise: 3 stars, standard (body_append) Lemma body_append: semax_body Vprog Gprog f_append append_spec.

Proof.

start_function.

forward_if. -

This if-then branch handles the cases in which *x* is null. In other words, *s1* should be nil. We can easily derive this by listrep_null. The rest of the proof in this branch is left as an exercise. rewrite (*listrep_null* _ *x*) by auto.

admit.

-

This time, we know that *x* is not null; thus *s1* should be nonempty. rewrite (*listrep_nonnull* _ *x*) by auto.

Intros h r u.

forward. *forward.*

After symbolically executing two assignment commands, we arrive at the while loop. As mentioned above, we can verify it using the following loop invariant. *forward_while*

```
( EX s1a: list val, EX b: val, EX s1c: list val, EX t: val, EX u: val,
  PROP (s1 = s1a ++ b :: s1c)
  LOCAL (temp _x x; temp _t t; temp _u u; temp _y y)
  SEP (lseg s1a x t;
       data_at Tsh t_list (b, u) t;
       listrep s1c u;
       listrep s2 y))%assert.
```

\+

Exists (@nil **val**) *h r x u*.

subst *s1*. *entailer*!. unfold lseg; *entailer*!.

\+

entailer!.

\+

We know *u* is not null from the fact that the loop condition is true. Thus we can

represent *s1c* in the form of (c :: *s1d*). `clear` *h r u H0*; `rename` *u0 into u*.
`rewrite` (*listrep_nonnull _ u*) `by auto`.
Intros c s1d z.
forward. *forward.*

In the end of the loop body, we need to re-establish the loop invariant. At this point, the memory layout can be illustrated by the following diagram.

(The following diagram is best demonstrated with a monospaced font.)

new t new u | | | | +—+—+ +—+—+ +—+—+ +—+—+ x ==> ... ===> | | t ==> | b | u ==> | c | z ==> | | ===> ... +—+—+ +—+—+ +—+—+ +—+—+
| <===== s1a =====> | (b) (c) | <==== s1d ====> | | <========== new s1a ========> | (new b) | <== new s1c ==> |
+—+—+ +—+—+ +—+—+ y ==> | | ===> | | ==> | | ===> ... +—+—+ +—+—+ +—+—+ | <=============== s2 ================> |

Clearly, *s1a* ++ *b* :: nil should be the new value of *s1a*; *c* should be the new value of *b*; *s1d* should be the new value of *s1c*; *u* should be the new value of *t*; and *z* should be the new value of *u*. The next command instantiates the existentially quantified variables in our loop invariant accordingly.

Exists ((*s1a* ++ *b* :: nil), *c*, *s1d*, *u*, *z*). `unfold` fst, snd.

As usual, we try entailer! to solve this proof goal. This time, entailer! does not solve it directly. Instead, two simplified proof goals are left. Their proofs are left for the reader, using app_assoc, singleton_lseg and lseg_lseg. *entailer!*.
× *admit.*
× *admit.*
+
After exiting the loop, the loop condition must be false, i.e. u is the null pointer. Thus *s1c* = nil and *s1* = *s1a* ++ [*b*]. `clear` *h r u H0*; `rename` *u0 into u*.
`rewrite` (*listrep_null s1c*) `by auto`.
Intros.
`subst` *s1c*.

The rest of the proof is standard. Hint, singleton_lseg, lseg_lseg and/or lseg_list may be useful. *admit.*

Admitted.

□

13.1.4 Additional exercises: more proofs about list segments

For verifying the C function append, it is enough to have only three separation logic proof rules about lseg: singleton_lseg, lseg_lseg and lseg_list. The following exercises are other important properties of lseg.

Exercise: 1 star, standard: (lseg2listrep) Lemma lseg2listrep: ∀ *s x*,

```
  lseg s x nullval |-- listrep s x.
Proof.
   Admitted.
   □
```

Exercise: 1 star, standard: (listrep2lseg) Lemma listrep2lseg: ∀ s x,

```
  listrep s x |-- lseg s x nullval.
Proof.
   Admitted.
   □
```

```
Corollary lseg_listrep_equiv: ∀ s x,
  lseg s x nullval = listrep s x.
Proof.
  intros.
  apply pred_ext.
  + apply lseg2listrep.
  + apply listrep2lseg.
Qed.
```

Exercise: 2 stars, standard: (lseg_lseg_inv) Lemma lseg_lseg_inv: ∀ s1 s2 x z,

```
  lseg (s1 ++ s2) x z |-- EX y: val, lseg s1 x y × lseg s2 y z.
Proof.
   Admitted.
   □
```

Exercise: 2 stars, standard: (loopy_lseg_no_connection) Lemma loopy_lseg_no_connection:

```
∀ s1 s2 x y z,
  s1 ≠ nil →
  s2 ≠ nil →
  x = y →
  lseg s1 x y × lseg s2 y z |-- FF.
Proof.
   Admitted.
   □
```

13.1.5 Additional exercises: loop-free list segments

In the following exercise, try to redo the proof above using a different partial-linked-list predicate.

We have mentioned that the lseg predicate allows a loopy structure, which is quite counterintuitive. Here is an alternate definition that prohibits loops:

```
Fixpoint nt_lseg (contents: list val) (x z: val) : mpred :=
```

```
  match contents with
  | nil ⇒ !! (x = z) && emp
  | h::hs ⇒ EX y:val, !! (x ≠ z)
                  && data_at Tsh t_list (h, y) x × nt_lseg hs y z
  end.
```

Arguments nt_lseg *contents x z* : simpl *never*.

Here, "nt" means no-touch.

The difference between nt_lseg and lseg is the extra proposition $x \neq z$ in the nonempty situation. This extra clause in nt_lseg prevents loop structures.

The proof theories of nt_lseg and lseg are a bit different as well. The following diagram shows that the counterpart of lseg_lseg is not valid!

+—+—+ +—+—+ +—+—+ +—+—+ x ==> | a | u ==> | b | y ==> | c | v ==> | d |
u ===+ +—+—+ +- -+—+ +—+—+ +—+—+ | ^ | | | +==========================

In this example, both [a; b] and [c; d] are stored in loop-free partial linked lists but it is not true for their concatenation. In general, if (nt_lseg *s1 x y*) and (nt_lseg *s2 y z*) describe two loop-free partial linked lists, the assertion

(nt_lseg s1 x y * nt_lseg s2 y z)

cannot ensure that the structure is loop free. Specifically, the address z may be used in (nt_lseg *s1 x y*). In other words,

nt_lseg s1 x y * nt_lseg s2 y z |-/- nt_lseg (s1 ++ s2) x z

For nt_lseg, the following proof rules are useful.

Exercise: 2 stars, standard, optional (nt_lseg) Lemma singleton_nt_lseg: ∀ (*contents*: list val) (*a x y*: val),

```
  data_at Tsh t_list (a, y) x × listrep contents y |-
  nt_lseg [a] x y × listrep contents y.
Proof.
  Admitted.
```

□

Exercise: 2 stars, standard, optional (singleton_nt_lseg') Lemma singleton_nt_lseg': ∀ (*a b x y z*: val),

```
  data_at Tsh t_list (a, y) x × data_at Tsh t_list (b, z) y |-
  nt_lseg [a] x y × data_at Tsh t_list (b, z) y.
Proof.
  Admitted.
```

□

Exercise: 2 stars, standard, optional (nt_lseg_nt_lseg) Lemma nt_lseg_nt_lseg: ∀ (*s1 s2*: list val) (*a x y z u*: val),

```
  nt_lseg s1 x y × nt_lseg s2 y z × data_at Tsh t_list (a, u) z |-
  nt_lseg (s1 ++ s2) x z × data_at Tsh t_list (a, u) z.
```

Proof.

Hint: This lemma illustrates the most classic case where aggressive *cancel* can turn a provable goal into an unprovable goal. For that reason, you may need to use entailer rather than entailer! at one point. *Admitted.*

□

Exercise: 2 stars, standard, optional (nt_lseg_list) Lemma nt_lseg_list: ∀ (*s1* *s2*: **list val**) (*x* *y*: **val**),

nt_lseg *s1* *x* *y* × listrep *s2* *y* |- listrep (*s1* ++ *s2*) *x*.

Proof.

Admitted.

□

Now, we will use nt_lseg instead of lseg in the loop invariant to prove body_append.

Exercise: 3 stars, standard, optional (body_append_alter1) Lemma body_append_alter1: semax_body Vprog Gprog f_append append_spec.

Proof.

start_function.

forward_if. -

rewrite (*listrep_null* _ *x*) by auto.

admit.

-

rewrite (*listrep_nonnull* _ *x*) by auto.

Intros h r u.

forward. *forward.* Now use *forward_while* to verify this while loop. Remember, *forward_while* will generate four proof goals: current precondition implies loop invariant; loop test is safe to execute; loop body preserves invariant; and the correctness of after-loop commands. *Admitted.*

□

Chapter 14

Library VC.Verif_append2

14.1 Verif_append2: Magic wand, partial data structure

14.1.1 Separating Implication

Separating implication is another separation logic operator. It is written as -* in Verifiable C. Because of its shape, it is usually called "magic wand". The following `Locate` command and `Check` command show this notation definition and its typing information.

```
Require VC.Preface.
```

```
Require Import VST.floyd.proofauto.
```

```
Locate "-*".
Check wand.
```

In separation logic, a heaplet (piece of memory) m satisfies P -* Q if and only if: for any possible heaplet n, if n and m are disjoint and n satisfies P, then the combination of n and m will satisfy Q.

The most important proof rule for separating implication is the adjoint property. It says, $P \times Q$ derives R if and only if P derives Q-*R. This rule is called wand_sepcon_adjoint in Verifiable C.

```
Check wand_sepcon_adjoint.
```

Because of this property, we also call -* a right adjoint of $\times$. In propositional logic, implication $\rightarrow$ is a right adjoint of conjunction $\wedge$.

```
Lemma implies_and_adjoint:
  ∀ P Q R : Prop, (P ∧ Q → R) ↔ (P → (Q → R)).
Proof. intuition. Qed.
```

This intrinsic similarity gives -* the name "separating implication". The following are two other important properties of -*; we can easily find their counterparts about propositional-logic "implication".

Proof rules for separating implication:

```
Check wand_derives.
```

```
Check modus_ponens_wand.
```

Now, we learn to use the adjoint property to prove other separation-logic rules about -*. We will start from an easy one.

```
Lemma wand_trivial: ∀ P Q: mpred, P |-- Q -* (P × Q).
Proof.
  intros.
  rewrite ← wand_sepcon_adjoint.
  apply derives_refl.
Qed.
```

Then, we will reprove the modus ponens rule for -* and × from the adjoint property.

```
Lemma modus_ponens_wand_from_adjoint: ∀ P Q : mpred, P × (P -* Q) |-- Q.
Proof.
  intros.
  rewrite sepcon_comm.
  rewrite → wand_sepcon_adjoint.
  apply derives_refl.
Qed.
```

Now prove wand_derives using wand_sepcon_adjoint and modus_ponens_wand. You can use other proof rules about ×, such as sepcon_derives. Also, the tactic *sep_apply* may be useful.

Exercise: 2 stars, standard: (wand_derives) Lemma wand_derives_from_adjoint_and_modus_ponens:

```
  ∀ P P' Q Q' : mpred,
    P' |-- P → Q |-- Q' → P -* Q |-- P' -* Q'.
Proof.
    Admitted.
```

□

Theorem wand_frame_ver is the counterpart of implication's transitivity. As we will see, it allows "vertical composition" of wand frames.

```
Check wand_frame_ver.
```

Prove it by wand_sepcon_adjoint and *sep_apply* (modus_ponens_wand ...)

Exercise: 2 stars, standard: (wand_frame_ver) Lemma wand_frame_ver_from_adjoint_and_modus_po

```
  ∀ P Q R : mpred, (P -* Q) × (Q -* R) |-- P -* R.
Proof.
    Admitted.
```

□

More exercises: prove that emp -* emp and emp are equivalent.

Exercise: 3 stars, standard: (emp_wand_emp) Lemma emp_wand_emp_right: emp |-
emp -* emp.
Proof.
  Admitted.

Lemma emp_wand_emp_left: emp -* emp |- emp.
Proof.
  Admitted.
  □

14.1.2 List segments by magic wand

```
Require Import VC.append.
Require Import VC.Verif_append1.
```

In Verif_append1, we recursively defined a new separation logic predicate: list segment. That predicate describes a heaplet that contains a partial linked list.

In this chapter, we learn a different way of describing partial data structures–we use magic wand together with quantifiers.

This is a natural idea. Using linked lists as an example, adding a linked list to the tail of a partial linked list (or a list segment) will result in a complete linked list from the head. Thus, a partial linked list can be described by "the added list -* the complete list". Formally:

```
Definition wlseg (contents: list val) (x y: val) : mpred :=
  ALL tail: list val, listrep tail y -* listrep (contents ++ tail) x.
```

Here, "w" in "wlseg" represents "wand".

This definition is very different from lseg and is beautifully simple, and it generalizes nicely to other data structures such as trees.

Let's prove some basic properties of wlseg. The following lemmas show how a wand expression can be introduced (emp_wlseg and singleton_wlseg), how a wand expression can be eliminated (wlseg_list) and how two wand expressions can merge (wlseg_wlseg).

There are two logical operators in this definition, -* and the universal quantifier. Previously, we have learned how to prove properties about × and -*. To prove properties about universal quantifiers, we will use allp_left and allp_right.

```
Check allp_left.
Check allp_right.
```

The first property of wlseg is that we can introduce wlseg from emp.

```
Lemma emp_wlseg: ∀ (x: val),
  emp |- wlseg [] x x.
Proof.
  intros.
  unfold wlseg.
  apply allp_right; intro tail.
```

```
  rewrite ← wand_sepcon_adjoint.
  rewrite emp_sepcon.
  simpl app.
  apply derives_refl.
Qed.
```

Next, we show that two wlseg predicates can be merged into one.

```
Lemma wlseg_wlseg: ∀ (s1 s2: list val) (x y z: val),
  wlseg s2 y z × wlseg s1 x y |-- wlseg (s1 ++ s2) x z.
Proof.
 intros.
 unfold wlseg.
```

First, extract the universally quantified variable *tail* on the right side. `apply allp_right; intro tail.`

Next, instantiate the first quantified *tail0* on the left with *tail*. `rewrite → wand_sepcon_adjoint.`

```
 apply (allp_left _ tail).
 rewrite ← wand_sepcon_adjoint.
```

Then, instantiate the other quantified *tail0* on the left with *s2* ++ *tail*. `rewrite sepcon_comm, → wand_sepcon_adjoint.`

```
 apply (allp_left _ (s2 ++ tail)).
 rewrite ← wand_sepcon_adjoint, sepcon_comm.
```

Finally, complete the proof with wand_frame_ver. `rewrite ← app_assoc.`

```
 apply wand_frame_ver.
Qed.
```

This theorem wlseg_wlseg shares the same form with lseg_lseg. In fact, properties about lseg and wlseg are very similar. The following exercises are to prove the counterparts of singleton_lseg and lseg_list.

Exercise: 2 stars, standard: (singleton_wlseg) `Lemma singleton_wlseg: ∀ (a: val) (x y: val),`

```
  data_at Tsh t_list (a, y) x |-- wlseg [a] x y.
Proof.
```

Admitted.

☐

Exercise: 2 stars, standard: (wlseg_list) `Lemma wlseg_list: ∀ (s1 s2: list val) (x y: val),`

```
  wlseg s1 x y × listrep s2 y |-- listrep (s1 ++ s2) x.
Proof.
```

Admitted.

☐

14.1.3 Proof of the append function by wlseg

Now, we are ready to reprove the correctness for the C program append. This time, we will use wlseg to write the loop invariant.

Exercise: 3 stars, standard: (body_append_alter2) Lemma body_append_alter2: semax_body Vprog Gprog f_append append_spec.

```
Proof.
start_function.
forward_if. -
  rewrite (listrep_null _ x) by auto.
   admit.
-
  rewrite (listrep_nonnull _ x) by auto.
  Intros h r u.
  forward. forward.
```

Here, we use wlseg to represent a list segment. *forward_while*

```
    (EX s1a: list val, EX b: val, EX s1c: list val, EX t: val, EX u: val,
       PROP (s1 = s1a ++ b :: s1c)
       LOCAL (temp _x x; temp _t t; temp _u u; temp _y y)
       SEP (wlseg s1a x t;
              data_at Tsh t_list (b, u) t;
              listrep s1c u;
              listrep s2 y))%assert.
 +
```

To derive a loop invariant from the current assertion, the key point is to introduce wlseg. You may find emp_wlseg helpful here. *admit.*

```
 +
    entailer!.
 +
```

Step forward through the loop body; along the way you'll need to do other transformations on the current assertion, to uncover opportunities to step forward. At the end of the loop body, you need to prove that a list segment for *s1a* and a singleton cell for *b* forms a longer list segment, whose contents is *s1a* ++ *b* :: nil. You may find singleton_wlseg and wlseg_wlseg useful there. *admit.*

```
 +
```

After you symbolicly execute the return command, you need to establish one single linked list with contents *s1a* ++ *b* :: *s2* from a list segment for *s1a*, a singleton cell for *b* and another linked list for *s2*. You may find singleton_wlseg and wlseg_list useful there. *admit.*

Admitted.

□

14.1.4 The general idea: magic wand as frame

Let's review the proof script above. Before the loop, we first derive wlseg [] x x from emp. After every iteration of the loop body, we merge a piece of singleton list segment wlseg [b] t u into it. When exiting the loop, we get wlseg [$s1a$] x t where $s1 = s1a$ ++ [b]. Eventually, this list segment is merged with a tail listrep ([b] ++ $s2$) t, which results in listrep ($s1$ ++ $s2$) x.

From where the list segment is introduced in the proof, to where the list segment is eliminated by merging, the C program never modifies the submemory described by that wlseg predicate. In separation logic, such a separating conjunct is called a "frame". Thus, the general idea here is to use a wand expression to describe a partial data structure and this wand expression will act as a frame in program verification.

In the proof of body_append_alter2, we only need four of wlseg's properties: emp_wlseg, singleton_wlseg, wlseg_wlseg and wlseg_list. They are used to introduce, merge and eliminate wlseg predicates. Here are some general patterns beyond these specific rules.

```
Lemma wandQ_frame_elim_mpred: ∀ {A: Type} (P Q: A → mpred) (a: A),
  (ALL x : A, P x -* Q x) × P a |- Q a.
Proof.
  intros.
  rewrite → wand_sepcon_adjoint.
  apply (allp_left _ a).
  apply derives_refl.
Qed.
```

"ver" in the name of the next lemma stands for "vertical composition" of wand frames. One wand-frame is nested inside another.

```
Lemma wandQ_frame_ver_mpred: ∀ {A: Type} (P Q R: A → mpred),
  (ALL x : A, P x -* Q x) × (ALL x: A, Q x -* R x) |- ALL x: A, P x -* R x.
Proof.
  intros.
  apply allp_right; intro a.
  rewrite → wand_sepcon_adjoint.
  apply (allp_left _ a).
  rewrite ← wand_sepcon_adjoint.
  rewrite sepcon_comm, → wand_sepcon_adjoint.
  apply (allp_left _ a).
  rewrite ← wand_sepcon_adjoint, sepcon_comm.
  apply wand_frame_ver.
Qed.
```

14.1.5 Case study: list segments for linked list box

In the following exercise, you are going to apply the magic-wand-as-frame method on a slightly different data structure.

Consider the following C function, *append2*.

struct list * append2 (struct list * x, struct list * y) { struct list **retp, **curp; retp = & x; curp = & x; while (*curp != NULL) { curp = & ((*curp) -> tail); } *curp = y; return *retp; }

In comparison, this is append.

struct list *append (struct list *x, struct list *y) { struct list *t, *u; if (x==NULL) return y; else { t = x; u = t->tail; while (u!=NULL) { t = u; u = t->tail; } t->tail = y; return x; } }

In append, u always equals $t \rightarrow tail$ after every iteration. When exiting the loop, the value of u is always null; that is not important. More important is the address from whence the null value is loaded. A new value will be stored into that location in memory. The program variable t is used to remember that address.

The C function *append2* implements linked-list append in an alternative way. In this function, *curp*'s value is not an address in the linked list. Instead, it records where a linked list address is stored in memory. Specifically, when *curp* points the head pointer x, the value of *curp* is the address of x. When *curp* points to some intermediate linked list node, the value of *curp* is the predecessor node's *tail* field address. Using this implementation, we do not need to test whether x is null in the beginning.

The following separation logic predicate defines this data structure.

```
Definition t_list_box := tptr t_list.

Definition listboxrep (contents: list val) (x: val) :=
  EX y: val, data_at Tsh t_list_box y x × listrep contents x.

Definition lbseg (contents: list val) (x y: val) :=
  ALL tail: list val, listboxrep tail y -* listboxrep (contents ++ tail) x.
```

Previously, we have shown that we can introduce, eliminate and merge wand expressions by proving emp_wlseg, singleton_wlseg, wlseg_list and wlseg_wlseg. Now, your task is to prove lbseg's properties. Hint: proving wlseg's properties and proving lbseg's properties should be very similar.

Exercise: 1 star, standard: (emp_lbseg) Introducing a wand expression, lbseg, from emp.

```
Lemma emp_lbseg: ∀ (x: val),
  emp |- lbseg [] x x.
Proof.
  Admitted.
```

☐

Exercise: 2 stars, standard: (lbseg_lbseg) Merging two wand expressions.

```
Lemma lbseg_lbseg: ∀ (s1 s2: list val) (x y z: val),
  lbseg s2 y z × lbseg s1 x y |- lbseg (s1 ++ s2) x z.
Proof.
  Admitted.
```

□

Exercise: 2 stars, standard: (listbox_lbseg) Eliminating a wand expression. Lemma listbox_lbseg: ∀ (*s1 s2*: list val) (*x y*: val),
lbseg *s1 x y* × listboxrep *s2 y* |-- listboxrep (*s1* ++ *s2*) *x*.
Proof.
Admitted.
□

14.1.6 Comparison and connection: lseg vs. wlseg

We have demonstrated two different approaches to define a separation logic predicate for list segments. In Verif_append1 we define it using recursive definition over the list. In this chapter, we use a quantifed magic wand expression. It is natural to ask: what is the relation between lseg and wlseg? Are they equivalent to each other? They following theorems can offer a brief answer.

First of all, recursive defined lseg is a logically stronger predicate than wlseg. Lemma lseg2wlseg: ∀ *s x y*, lseg *s x y* |-- wlseg *s x y*.

```
Proof.
  intros.
  unfold wlseg.
  apply allp_right; intros tail.
  rewrite ← wand_sepcon_adjoint.
  sep_apply (lseg_list s tail x y).
  apply derives_refl.
Qed.
```

In some special cases, wlseg derives lseg as well. Lemma wlseg2lseg_nullval: ∀ *s x*, wlseg *s x* nullval |-- lseg *s x* nullval.

```
Proof.
  intros.
  unfold wlseg.
  apply allp_left with (@nil val).
  unfold listrep at 1.
  rewrite prop_true_andp by auto.
  entailer!.
  rewrite ← app_nil_end.
```

The proof goal now has the form: a wand expression derives some wand-free assertion. Usually, this is a tough task because there is no good way to eliminate magic wand on left side. But this proof goal is special. We can add an extra separating conjunct emp to the left side and use modus_ponens_wand to eliminate wand. rewrite ← (emp_sepcon (emp -* listrep *s x*)).
sep_apply (modus_ponens_wand emp (listrep *s x*)).

Then, easy! `apply` *listrep2lseg*.
```
Qed.
```

Combining these two lemmas above together, we know that wlseg-to-null equals lseg.

```
Lemma wlseg_nullval: ∀ s x, wlseg s x nullval = lseg s x nullval.
Proof.
  intros.
  apply pred_ext.
  + apply wlseg2lseg_nullval.
  + apply lseg2wlseg.
Qed.
```

```
Corollary wlseg_listrep_equiv: ∀ s x, wlseg s x nullval = listrep s x.
Proof.
  intros.
  rewrite wlseg_nullval, lseg_listrep_equiv.
  reflexivity.
Qed.
```

However, wlseg does not derive lseg in general. As mentioned above, to eliminate magic wand on the left side is hard. When $y \neq$ nullval, we cannot instantiate the universally quantified variable *tail* inside (wlseg s x y) to get the form emp -* _. The following is a counterexample of the general entailment from wlseg to lseg. On one hand, it is obvious that data_at_ Tsh t_list y |-/- lseg $[a]$ x y. On the other hand, data_at_ Tsh t_list y |-- wlseg $[a]$ x y. See the following theorem:

```
Lemma wlseg_weird: ∀ a x y,
  data_at_ Tsh t_list y |-- wlseg [a] x y.
Proof.
  intros.
  unfold wlseg.
  apply allp_right; intros s.
  rewrite ← wand_sepcon_adjoint.
  destruct s.
  + unfold listrep at 1.
    entailer!.
    destruct H as [H _].
    contradiction.
  + unfold listrep at 1; fold listrep.
    Intros u.
    sep_apply (data_at_conflict Tsh t_list (default_val t_list) (v, u) y); auto.
    entailer!.
Qed.
```

Chapter 15

Library VC.Verif_strlib

15.1 Verif_strlib: String functions

In this chapter we show how to prove the correctness of C programs that use null-terminated character strings.

Here are some functions from the C standard library, *strlib.c*

15.2 Standard boilerplate

```
Require VC.Preface. Require Import VST.floyd.proofauto.
Require Import VC.strlib.
Instance CompSpecs : compspecs. make_compspecs prog. Defined.
Definition Vprog : varspecs. mk_varspecs prog. Defined.
Require Import VC.hints. Require Import Coq.Strings.Ascii.
```

15.3 Representation of null-terminated strings.

Coq represents a string as a list-like Inductive of Ascii characters: `Locate` *string*. `Print` **string**.

The C programming language represents a *character* as a byte, that is, an 8-bit signed or unsigned integer. In Coq represent the 8-bit integers using the byte type. `Print` **byte**.

```
Search byte.
```

We can convert a Coq string to a list of bytes:

```
Fixpoint string_to_list_byte (s: string) : list byte :=
  match s with
  | EmptyString ⇒ nil
  | String a s' ⇒ Byte.repr (Z.of_N (Ascii.N_of_ascii a))
                                              :: string_to_list_byte s'
```

```
  end.
Definition Hello := "Hello"%string.
Definition Hello' := string_to_list_byte Hello.
Eval simpl in string_to_list_byte Hello.
Section StringDemo.
  Variable p : val.
```

To describe a single byte in memory, we can use data_at with the signed-character type:

```
Print tschar. Check (data_at Tsh tschar (Vint (Int.repr 72)) p).
```

This data_at Tsh tschar (Vint (*Int.repr* 72)) p is an mpred, that is, a memory predicate in separation logic. It says, at address p in memory there is a sequence of bytes whose length is appropriate for type tschar; that is, one byte. The contents of this sequence of bytes (one byte) is a representation of the integer 72. The ownership share (access permission) of memory at address p is the "top share" Tsh

```
Check (data_at Tsh tschar (Vbyte (Byte.repr 72))).
```

We can express the same thing using Vbyte (*Byte.repr* 72). In fact, Vbyte is not a primitive CompCert value, it is a definition: `Print Vbyte.`

```
Goal Vbyte (Byte.repr 72) = Vint (Int.repr 72).
Proof. reflexivity. Qed.
Goal Vbyte (Byte.repr 72) = Vint (Int.repr 72).
Proof.
  unfold Vbyte.
  rewrite Byte.signed_repr.
  auto.
  rep_lia.
Qed.
```

The C programming language represents a string of length k as an array of nonnull characters (bytes), terminated by a null character. We represent this in separation logic using the cstring memory-predicate:

```
Locate cstring. Print cstring.
```

Here is an example of a cstring predicate that says, At address p there is a null-terminated string representing "Hello".

```
Check (cstring Tsh Hello' p).
```

By unfolding the definition of cstring this is equivalent to,

```
Check (
  !! (¬ In Byte.zero Hello') &&
  data_at Tsh (tarray tschar (5 + 1)) (map Vbyte (Hello'++[Byte.zero])) p ).
```

This says, no element of the list Hello' is equal to *Byte.zero*. In memory at address p there is an array of 6 bytes, whose contents are the contents of Hello' with a *Byte.zero* appended at the end.

Sometimes we know that there is a null-terminated string inside an array of length n. That is, there are k nonnull characters (where $k<n$), followed by a null character, followed by n-(k+1) uninitialized (or don't-care) characters. We represent this with the cstringn predicate. `Print cstringn.`

`Check (cstringn Tsh Hello' 10 p).`

`End StringDemo.`

15.4 Reasoning about the contents of C strings

In separation logic proofs about C strings, we often find proof goals similar to the one exemplified by this lemma: `Lemma demonstrate_cstring1:`

```
 ∀ i contents
    (H: ¬ In Byte.zero contents)
    (H0: Znth i (contents ++ [Byte.zero]) ≠ Byte.zero)
    (H1: 0 ≤ i ≤ Zlength contents),
    0 ≤ i + 1 < Zlength (contents ++ [Byte.zero]).
Proof.
intros.
```

A null-terminated string is an array of characters with three parts:

- The contents of the string, none of which is the '\0' character;
- The null termination character, equal to Byte.zero;
- the remaining garbage in the array, after the null.

When processing a string, you should maintain three kinds of assumptions above the line:

- Hypothesis H above the line says that none of the

contents is the null character;

- Hypothesis *H0* typically comes from a loop test, $s[i]!{=}0$
- *H1* typically comes from a loop invariant: suppose a

a loop iteration variable _i (with value i) is traversing the array. We expect that loop to go up to but no farther than the null character, that is, one past the contents.

The cstring tactic processes all three of these hypotheses to conclude that $i <$ Zlength *contents*. `assert (H7: i < Zlength contents) by cstring.`

But actually, cstring tactic will prove any rep_lia consequence of that fact. For example:

```
clear H7.
autorewrite with sublist.
```

cstring.
Qed.

Here is another demonstration. When your loop on the string contents reaches the end, the loop test $s[i]!{=}0$ is false, so therefore $s[i]{=}0$. Lemma demonstrate_cstring2:

```
 ∀ i contents
    (H: ¬ In Byte.zero contents)
    (H0: Znth i (contents ++ [Byte.zero]) = Byte.zero)
    (H1: 0 ≤ i ≤ Zlength contents),
    i = Zlength contents.
Proof.
intros.
```

Hypothesis *H0* expresses that the loop test determined $s[i]{=}0$. The cstring can then prove that i = Zlength *contents*. *cstring*.
Qed.

15.5 Function specs

strlen(s) returns the length of the string s. Definition strlen_spec :=

```
 DECLARE _strlen
  WITH sh: share, s : list byte, str: val
  PRE [ tptr tschar ]
    PROP (readable_share sh)
    PARAMS (str)
    SEP (cstring sh s str)
  POST [ tuint ]
    PROP ()
    RETURN (Vptrofs (Ptrofs.repr (Zlength s)))
    SEP (cstring sh s str).
```

strcpy(*dest*,*src*) copies the string *src* to the array *dest*. Definition strcpy_spec :=

```
 DECLARE _strcpy
  WITH wsh: share, rsh: share, dest : val, n : Z, src : val, s : list byte
  PRE [ tptr tschar, tptr tschar ]
    PROP (writable_share wsh; readable_share rsh; Zlength s < n)
    PARAMS (dest; src)
    SEP (data_at_ wsh (tarray tschar n) dest; cstring rsh s src)
  POST [ tptr tschar ]
    PROP ()
    RETURN (dest)
    SEP (cstringn wsh s n dest; cstring rsh s src).
```

strcmp(*s1*,*s2*) compares strings s1 and s2 for lexicographic order. This funspec is an underspecification of the actual behavior, in that it specifies equality testing only. Definition

```
strcmp_spec :=
 DECLARE _strcmp
  WITH sh1 : share, sh2 : share, str1 : val, s1 : list byte, str2 : val,
       s2 : list byte
  PRE [ tptr tschar, tptr tschar ]
    PROP (readable_share sh1 ; readable_share sh2)
    PARAMS (str1 ; str2)
    SEP (cstring sh1 s1 str1 ; cstring sh2 s2 str2)
  POST [ tint ]
   EX i : int,
    PROP (if Int.eq_dec i Int.zero then s1 = s2 else s1 ≠ s2)
    RETURN (Vint i)
    SEP (cstring sh1 s1 str1 ; cstring sh2 s2 str2).
Definition Gprog : funspecs := [ strlen_spec; strcpy_spec; strcmp_spec ].
```

15.6 Proof of the *strlen* function

Exercise: 2 stars, standard (body_strlen) Lemma body_strlen: semax_body Vprog Gprog f_strlen strlen_spec.
Proof.
start_function.

Look at the proof goal below the line. We have the assertion,

PROP ( ) *LOCAL* (temp _str *str*) *SEP* (cstring *sh* *s* *str*))

When proving things about a string-manipulating function, the first decision is: Does this function treat the string *abstractly* or does it subscript the array and look at the individual characters?

- If abstract, then we should *not* unfold the definition cstring.
- If we subscript the array directly, we *must* unfold cstring.

Since this *strlen* function does access the array contents, we start by unfolding cstring. unfold cstring in *.

Now, we have a Proposition ¬ In *Byte.zero* *s* in the *SEP* clause of our assertion; we can move it above the line by *Intros*. *Intros.*
forward.

Now we are at a for-loop. Unlike a simple while-loop, a for-loop may:

- *break*; (prematurely terminate the loop)
- *continue*; (prematurely terminate the body, skipping to the increment)

So therefore the simple Hoare-logic *while* rule is not always applicable. The general form of Verifiable C's loop tactic is:

forward_loop Inv1 continue: Inv2 break: Inv3

where *Inv1* is the invariant that holds right before the loop test, *Inv2* is the invariant that holds right before the increment, and *Inv3* is the postcondition of the loop.

Providing *continue*: *Inv2* is optional, as is *break*: *Inv3*. In many cases the *forward_loop* tactic can figure out that the *continue*: invariant is not needed (if the loop doesn't contain a *continue* statement), or the *break*: postcondition is not needed (if there's no *break* statement, or if there are no commands after the loop).

So let's try this loop with only a single loop invariant: *forward_loop* (EX i : Z,

```
PROP (0 ≤ i < Zlength s + 1)
LOCAL (temp _str str; temp _i (Vptrofs (Ptrofs.repr i)))
SEP (data_at sh (tarray tschar (Zlength s + 1))
        (map Vbyte (s ++ [Byte.zero])) str)).
```

Look at the *LOCAL* clause that binds temp _i to the value Vptrofs (*Ptrofs.repr i*). What is that? The answer is, in reasoning about C programs, we need:

- 8-bit integers, *Byte.int* or simply byte
- 32-bit integers, *Int.int* or simply **int**
- 64-bit integers, *Int64.int*

But there is also the concept expressed in C as size_t, that is, *The integer type with the same number of bits as a pointer.* In this might be the same as 32-bit int, or it might be the same as 64-bit int.

So, just as CompCert has the *Int* module for reasoning about 32-bit integers and the Int64 module for 64-bit integers, it has also the *Ptrofs* module for reasoning about *pointer offsets*. *Ptrofs* is isomorphic to (but not identical to) either *Int64* or *Int*, depending on the boolean value *Archi.ptr64*.

```
Compute Archi.ptr64.
```

If this computes false, then this installation of Verifiable C is configured for 32-bit pointers; if true, then this Verifiable C is configured for 64-bit. But either way, to turn a *Ptrofs.int* value into a CompCert **val**, we have Vptrofs. And – just as we can write C programs that are portable to 32-bit or 64-bit pointers using size_t, we can write proofs scripts portable by using *Ptrofs*. Print Vptrofs.

```
assert (Example: Archi.ptr64=false →
          ∀ n, Vptrofs (Ptrofs.repr n) = Vint (Int.repr n)). {
 intro Hx; try discriminate Hx. all: intros.
all: hint.
all: autorewrite with norm.
all: auto.
} clear Example.
```

Now it's time to prove all the subgoals of *forward_loop*. ×
admit.
×
admit.
Admitted.
□

15.7 Proof of the *strcpy* function

Exercise: 2 stars, standard (strcpy_then_clause) The next lemma, or some variation of it, will be useful in the proof of the *strcpy* function (in the *then* clause of the *if* statement).

```
Lemma strcpy_then_clause:
  ∀ (wsh: share) (dest: val) (n: Z) (s: list byte),
  Zlength s < n →
  ¬ In Byte.zero s →
    data_at wsh (tarray tschar n)
      (map Vbyte (sublist 0 (Zlength s) s) ++
       upd_Znth 0 (list_repeat (Z.to_nat (n - Zlength s)) Vundef)
         (Vint (Int.repr (Byte.signed (Znth (Zlength s) (s ++ [Byte.zero]))))))
      dest
  |- data_at wsh (tarray tschar n)
        (map Vbyte (s ++ [Byte.zero]) ++
             list_repeat (Z.to_nat (n - (Zlength s + 1))) Vundef)
        dest.
Proof.
intros.
apply derives_refl'.
f_equal.
Check list_repeat_app'. Check upd_Znth_app1. Check app_Znth2. Check Znth_0_cons.
Admitted.
```

□

Exercise: 2 stars, standard (strcpy_else_clause)

```
Lemma strcpy_else_clause: ∀ wsh dest
n s i,
  Zlength s < n →
  ¬ In Byte.zero s →
  0 ≤ i < Zlength s + 1 →
  Znth i (s ++ [Byte.zero]) ≠ Byte.zero →
     data_at wsh (tarray tschar n)
      (upd_Znth i (map Vbyte (sublist 0 i s)
                   ++ list_repeat (Z.to_nat (n - i)) Vundef)
             (Vint (Int.repr (Byte.signed (Znth i (s ++ [Byte.zero])))))) dest
```

```
  |-- data_at wsh (tarray tschar n)
       (map Vbyte (sublist 0 (i + 1) s)
             ++ list_repeat (Z.to_nat (n - (i + 1))) Vundef) dest.
Proof.
intros.
apply derives_refl'.
f_equal.
```

Useful lemmas here will be: upd_Znth_app2, sublist_split, list_repeat_app', app_Znth1, map_app, app_ass, sublist_len_1. *Admitted.*

□

15.7.1 data_at is not injective!

The Vundef value means *uninitialized* or *undefined* or *defined but don't care*. Consider this lemma:

```
Lemma data_at_Vundef_example:
  ∀ i n sh p,
    0 ≤ i < n →
  data_at sh (tarray tschar n)
          (list_repeat (Z.to_nat (i+1)) (Vbyte Byte.zero)
              ++ list_repeat (Z.to_nat (n-(i+1))) Vundef) p
 |--
  data_at sh (tarray tschar n)
          (list_repeat (Z.to_nat i) (Vbyte Byte.zero)
              ++ list_repeat (Z.to_nat (n-i)) Vundef) p.
Proof.
intros.
```

The proof goal means: If cells $0 \leq j < i{+}1$ are zero and cells $i{+}1 \leq j < n$ are don't care, that implies the weaker statement that cells $0 \leq j < i$ are zero and cells $i \leq j < n$ are don't-care.

Now, let's try to prove it using the same technique as in strcpy_then_clause:

```
apply derives_refl'.
f_equal.
rewrite ← list_repeat_app' by lia.
replace (n-i) with (1 + (n-(i+1))) by lia.
rewrite ← list_repeat_app' by lia.
rewrite !app_ass.
f_equal.
f_equal.
```

Oops! The current proof goal is False! The problem was that we should not have applied derives_refl'. Abort.

```
Lemma data_at_Vundef_example:
  ∀ i n sh p,
```

```
    0 ≤ i < n →
  data_at sh (tarray tschar n)
           (list_repeat (Z.to_nat (i+1)) (Vbyte Byte.zero)
               ++ list_repeat (Z.to_nat (n-(i+1))) Vundef) p
 |-
  data_at sh (tarray tschar n)
           (list_repeat (Z.to_nat i) (Vbyte Byte.zero)
               ++ list_repeat (Z.to_nat (n-i)) Vundef) p.
Proof.
intros.
rewrite ← list_repeat_app' by lia.
replace (n-i) with (1 + (n-(i+1))) by lia.
rewrite ← list_repeat_app' by lia.
rewrite !app_ass.
Check split2_data_at_Tarray_app.
rewrite (split2_data_at_Tarray_app i) by list_solve.
rewrite (split2_data_at_Tarray_app 1) by list_solve.
rewrite (split2_data_at_Tarray_app i) by list_solve.
rewrite (split2_data_at_Tarray_app 1) by list_solve.
cancel.
Qed.
```

Why did that work? Let's look at a simpler example.

```
Lemma cancel_example:
 ∀ sh i j p q,
   data_at sh tint (Vint i) p × data_at sh tint (Vint j) q
 |- data_at sh tint (Vint i) p × data_at sh tint (Vundef) q.
Proof.
intros.
```

Uncomment the following line, and notice that it solves the goal. `apply sepcon_derives.`

-

In the first subgoal, we use the fact that |– is reflexive `apply derives_refl.`

-

In the second subgoal, we use the fact that *any* value implies a Vundef, or more generally, *any* value implies `default_val` *t* for any CompCert type t. `apply stronger_default_val.`

```
Qed.
```

The moral of the story is: When proving
data_at sh t a p |– data_at sh t b p
if (a=b) you can simplify the goal using
apply derives_refl'; f_equal.
but if a is strictly more defined than b, then derives_refl' is not appropriate.

Exercise: 3 stars, standard (body_strcpy) Lemma body_strcpy: semax_body Vprog Gprog f_strcpy strcpy_spec.

Proof.

start_function.

Because this function subscripts the strings, it does *not* treat the strings abstractly, we must unfold cstring and cstringn. unfold cstring,cstringn in *.

forward.

Intros.

```
forward_loop (EX i : Z,
  PROP (0 ≤ i < Zlength s + 1)
  LOCAL (temp _i (Vint (Int.repr i)); temp _dest dest; temp _src src)
  SEP (data_at wsh (tarray tschar n)
          (map Vbyte (sublist 0 i s) ++ list_repeat (Z.to_nat (n - i)) Vundef) dest;
        data_at rsh (tarray tschar (Zlength s + 1)) (map Vbyte (s ++ [Byte.zero])) src)).
```

+

admit.

+

Admitted.

□

Exercise: 3 stars, standard (example_call_strcpy) Prove the correctness of a function that calls *strcpy*.

int example_call_strcpy(void) { char buf10; strcpy(buf,"Hello"); return buf0; }

```
Definition stringlit_1_contents := Hello' ++ [Byte.zero].
```

```
Definition example_call_strcpy_spec :=
 DECLARE _example_call_strcpy
  WITH gv : globals
  PRE [ ]
    PROP ()
    PARAMS() GLOBALS (gv)
    SEP (cstring Ews (Hello' ++ [Byte.zero]) (gv ___stringlit_1))
  POST [ tint ]
    PROP ()
    RETURN (Vint (Int.repr (Z.of_N (Ascii.N_of_ascii "H"%char))))
    SEP (cstring Ews (Hello' ++ [Byte.zero]) (gv ___stringlit_1)).
```

```
Lemma body_example_call_strcpy: semax_body Vprog Gprog
         f_example_call_strcpy example_call_strcpy_spec.
```

Proof.

start_function.

This proof is fairly straightforward. First, figure out what WITH witness to give for forward_call. Hint: look at the SEP clause of the precondition of strcpy_spec, and see how the data_at and cstring conjuncts must match your current assertion.

Second, after the forward_call, don't forget to unfold cstringn; this is necessary because after the call we subscript the _buf array directly. *Admitted.*

□

Exercise: 4 stars, standard, optional (body_strcmp) Lemma body_strcmp: semax_body Vprog Gprog f_strcmp strcmp_spec.

Admitted.

□

Chapter 16

Library VC.Hashfun

16.1 Hashfun: Functional model of hash tables

This C program, *hash.c*, implements a hash table with external chaining. See https://www.cs.princeton.edu/
for an introduction to hash tables.

/* First, access a few standard-library functions */

16.1.1 A functional model

Before we prove the C program correct, we write a functional program that models its behavior as closely as possible. The functional program won't be (average) constant time per access, like the C program, because it takes linear time to get the nth element of a list, while the C program can subscript an array in constant time. But we are not worried about the execution time of the functional program; only that it serve as a model for specifying the C program.

```
Require Import VST.floyd.functional_base.

Instance EqDec_string: EqDec (list byte) := list_eq_dec Byte.eq_dec.

Fixpoint hashfun_aux (h: Z) (s: list byte) : Z :=
 match s with
 | nil ⇒ h
 | c :: s' ⇒
      hashfun_aux ((h × 65599 + Byte.signed c) mod Int.modulus) s'
end.

Definition hashfun (s: list byte) := hashfun_aux 0 s.

Definition hashtable_contents := list (list (list byte × Z)).

Definition N := 109.
Lemma N_eq : N = 109.
Proof. reflexivity. Qed.
Hint Rewrite N_eq : rep_lia.
```

```
Global Opaque N.

Definition empty_table : hashtable_contents :=
  list_repeat (Z.to_nat N) nil.

Fixpoint list_get (s: list byte) (al: list (list byte × Z)) : Z :=
  match al with
 | (k,i) :: al' ⇒ if eq_dec s k then i else list_get s al'
 | nil ⇒ 0
 end.

Fixpoint list_incr (s: list byte) (al: list (list byte × Z))
              : list (list byte × Z) :=
  match al with
 | (k,i) :: al' ⇒ if eq_dec s k
                    then (k, i +1)::al'
                    else (k,i)::list_incr s al'
 | nil ⇒ (s, 1)::nil
 end.

Definition hashtable_get (s: list byte) (contents: hashtable_contents) : Z :=
  list_get s (Znth (hashfun s mod (Zlength contents)) contents).

Definition hashtable_incr (s: list byte) (contents: hashtable_contents)
                    : hashtable_contents :=
  let h := hashfun s mod (Zlength contents)
  in let al := Znth h contents
  in upd_Znth h contents (list_incr s al).
```

Exercise: 2 stars, standard (hashfun_inrange) Lemma hashfun_inrange: ∀ s, 0 ≤ hashfun s ≤ Int.max_unsigned.

```
Proof.
   Admitted.
   □
```

Exercise: 1 star, standard (hashfun_get_unfold) Lemma hashtable_get_unfold:

```
 ∀ sigma (cts: list (list (list byte × Z) × val)),
 hashtable_get sigma (map fst cts) =
  list_get sigma (Znth (hashfun sigma mod (Zlength cts)) (map fst cts)).
Proof.
   Admitted.
   □
```

Exercise: 2 stars, standard (Zlength_hashtable_incr) Lemma Zlength_hashtable_incr:

```
 ∀ sigma cts,
      0 < Zlength cts →
```

```
    Zlength (hashtable_incr sigma cts) = Zlength cts.
Proof.
  Admitted.
Hint Rewrite Zlength_hashtable_incr using list_solve : sublist.
  □
```

Exercise: 3 stars, standard (hashfun_snoc) Lemma Int_repr_eq_mod:

```
  ∀ a, Int.repr (a mod Int.modulus) = Int.repr a.
Proof.
Print Int.eqm. Search Int.eqm. Admitted.
```

Use Int_repr_eq_mod in the proof of hashfun_aux_snoc.

```
Lemma hashfun_aux_snoc:
  ∀ sigma h lo i,
    0 ≤ lo →
    lo ≤ i < Zlength sigma →
  Int.repr (hashfun_aux h (sublist lo (i + 1) sigma)) =
  Int.repr (hashfun_aux h (sublist lo i sigma) × 65599
                                       + Byte.signed (Znth i sigma)).
Proof.
  Admitted.

Lemma hashfun_snoc:
  ∀ sigma i,
    0 ≤ i < Zlength sigma →
  Int.repr (hashfun (sublist 0 (i + 1) sigma)) =
  Int.repr (hashfun (sublist 0 i sigma) × 65599 + Byte.signed (Znth i sigma)).
Proof.
  Admitted.
  □
```

16.1.2 Functional model satisfies the high-level specification

The purpose of a hash table is to implement a finite mapping, (a finite function) from keys to values. We claim that the functional model (empty_table, hashtable_get, hashtable_incr) correctly implements the appropriate operations on the abstract data type of finite functions.

We formalize that statement by defining a Module Type:

```
Module Type COUNT_TABLE.
 Parameter table: Type.
 Parameter key : Type.
 Parameter empty: table.
 Parameter get: key → table → Z.
 Parameter incr: key → table → table.
```

```
 Axiom gempty: ∀ k,
       get k empty = 0.
 Axiom gss: ∀ k t,
      get k (incr k t) = 1+(get k t).
 Axiom gso: ∀ j k t,
      j ≠ k → get j (incr k t) = get j t.
End COUNT_TABLE.
```

This means: in any `Module` that satisfies this `Module Type`, there's a type table of count-tables, and operators empty, get, set that satisfy the axioms gempty, gss, and gso.

A "reference" implementation of COUNT_TABLE

Exercise: 2 stars, standard (FunTable) It's easy to make a slow implementation of COUNT_TABLE, using functions.

```
Module FunTable <: COUNT_TABLE.
 Definition table: Type := nat → Z.
 Definition key : Type := nat.
 Definition empty: table := fun k ⇒ 0.
 Definition get (k: key) (t: table) : Z := t k.
 Definition incr (k: key) (t: table) : table :=
    fun k' ⇒ if Nat.eqb k' k then 1 + t k' else t k'.
 Lemma gempty: ∀ k, get k empty = 0.
   Admitted.
 Lemma gss: ∀ k t, get k (incr k t) = 1+(get k t).
   Admitted.
 Lemma gso: ∀ j k t, j ≠ k → get j (incr k t) = get j t.
   Admitted.
End FunTable.
```

□

Demonstration that hash tables implement COUNT_TABLE

Exercise: 3 stars, standard (IntHashTable) Now we make a "fast" implementation using hash tables. We put "fast" in quotes because, unlike the imperative implementation, the purely functional implementation takes linear time, not constant time, to select the the i'th bucket. That is, Znth i al takes time proportional to i. But that is no problem, because we are not using hashtable_get and hashtable_incr as our real implementation; they are serving as the *functional model* of the fast implementation in C.

```
Module IntHashTable <: COUNT_TABLE.
 Definition hashtable_invariant (cts: hashtable_contents) : Prop :=
  Zlength cts = N ∧
  ∀ i, 0 ≤ i < N →
```

```
                  list_norepet (map fst (Znth i cts))
                  ∧ Forall (fun s ⇒ hashfun s mod N = i) (map fst (Znth i cts)).
 Definition table := sig hashtable_invariant.
 Definition key := list byte.

 Lemma empty_invariant: hashtable_invariant empty_table.
 Proof.
   Admitted.

Lemma incr_invariant: ∀ k cts,
      hashtable_invariant cts → hashtable_invariant (hashtable_incr k cts).
Proof.
   Admitted.

 Definition empty : table := exist _ _ empty_invariant.
 Definition get : key → table → Z :=
                      fun k tbl ⇒ hashtable_get k (proj1_sig tbl).
 Definition incr : key → table → table :=
          fun k tbl ⇒ exist _ _ (incr_invariant k _ (proj2_sig tbl)).

 Theorem gempty: ∀ k, get k empty = 0.
 Proof.
   Admitted.

 Theorem gss: ∀ k t, get k (incr k t) = 1 + (get k t).
 Proof.
   Admitted.

 Theorem gso: ∀ j k t,
      j ≠ k → get j (incr k t) = get j t.
Proof.
   Admitted.
   □

End IntHashTable.
```

Chapter 17

Library VC.Verif_hash

17.1 Verif_hash: Correctness proof of hash.c

In this chapter we will prove that the C program, hash.c, correctly implements the functional model in hashfun.v. That proof, composed with the proof in hashfun.v that the functional model behaves correctly as a key-value map with string keys, demonstrates the correct behavior of hash.c.

The usual boilerplate Require VC.Preface. Require Import VST.floyd.proofauto.
Require Import VST.floyd.library.
Require Import VC.hash.
Instance CompSpecs : **compspecs**. *make_compspecs* prog. Defined.
Definition Vprog : varspecs. *mk_varspecs* prog. Defined.
Require Import VC.hints.

Now we import some VST libraries that don't come standard with VST.floyd.proofauto.

```
Require Import VST.msl.wand_frame.
Require Import VST.msl.iter_sepcon.
Require Import VST.floyd.reassoc_seq.
Require Import VST.floyd.field_at_wand.
```

Now we import the functional model. Require Import VC.Hashfun.

17.2 Function specifications

Imports from the C string library (see Verif_strlib)

```
Definition strcmp_spec :=
 DECLARE _strcmp
  WITH str1 : val, s1 : list byte, str2 : val, s2 : list byte
  PRE [ tptr tschar, tptr tschar ]
```

```
      PROP ()
      PARAMS (str1; str2)
      SEP (cstring Ews s1 str1; cstring Ews s2 str2)
  POST [ tint ]
    EX i : int,
      PROP (if Int.eq_dec i Int.zero then s1 = s2 else s1 ≠ s2)
      RETURN (Vint i)
      SEP (cstring Ews s1 str1; cstring Ews s2 str2).

Definition strcpy_spec :=
 DECLARE _strcpy
  WITH dest : val, n : Z, src : val, s : list byte
  PRE [ tptr tschar, tptr tschar ]
      PROP (Zlength s < n)
      PARAMS (dest; src)
      SEP (data_at_ Ews (tarray tschar n) dest; cstring Ews s src)
  POST [ tptr tschar ]
      PROP ()
      RETURN (dest)
      SEP (cstringn Ews s n dest; cstring Ews s src).

Definition strlen_spec :=
 DECLARE _strlen
  WITH s : list byte, str: val
  PRE [ tptr tschar ]
      PROP ( )
      PARAMS (str)
      SEP (cstring Ews s str)
  POST [ size_t ]
      PROP ()
      RETURN (Vptrofs (Ptrofs.repr (Zlength s)))
      SEP (cstring Ews s str).
```

String functions: copy, hash

```
Definition copy_string_spec : ident × funspec :=
 DECLARE _copy_string
 WITH s: val, sigma : list byte, gv: globals
 PRE [ tptr tschar ]
      PROP ()
      PARAMS (s) GLOBALS(gv)
      SEP (cstring Ews sigma s; mem_mgr gv)
 POST [ tptr tschar ]
      EX p: val,
```

```
      PROP ( ) RETURN (p)
      SEP (cstring Ews sigma s;
           cstring Ews sigma p;
           malloc_token Ews (tarray tschar (Zlength sigma + 1)) p;
           mem_mgr gv).
```

```
Definition hash_spec : ident × funspec :=
  DECLARE _hash
  WITH s: val, contents : list byte
  PRE [ tptr tschar ]
          PROP ()
          PARAMS (s)
          SEP (cstring Ews contents s)
  POST [ tuint ]
        PROP ()
        RETURN (Vint (Int.repr (hashfun contents)))
        SEP (cstring Ews contents s).
```

Data structures for hash table

Some abbreviations for C struct types that we use frequently `Definition tcell := Tstruct _cell noattr.`
`Definition thashtable := Tstruct _hashtable noattr.`

A list_cell has four parts:

- a linked list *cons* cell with a key-pointer *kp*, integer *count*, and pointer to the *next* element of the linked list;
- the key-pointer points to a string (null-terminated array of char) containing the key;
- the cons cell was obtained by *malloc*, and must be freeable by *free*, and so there's a *malloc_token* giving that capability;
- the key-string also has a *malloc-token* so that it can be freed

```
Definition list_cell (key: list byte) (count: Z) (next: val) (p: val): mpred :=
 EX kp: val, cstring Ews key kp
                    × malloc_token Ews (tarray tschar (Zlength key + 1)) kp
                    × data_at Ews tcell (kp,(Vint (Int.repr count), next)) p
                    × malloc_token Ews tcell p.
Definition list_cell_local_facts:
  ∀ key count next p, list_cell key count next p |-- !! isptr p.
Proof. intros. unfold list_cell. Intros kp. entailer!. Qed.
Hint Resolve list_cell_local_facts: saturate_local.
```

```
Definition list_cell_valid_pointer:
  ∀ key count next p, list_cell key count next p |-- valid_pointer p.
Proof. intros. unfold list_cell. Intros kp. entailer!. Qed.
Hint Resolve list_cell_valid_pointer: valid_pointer.
```

Exercise: 1 star, standard (listcell_fold)

```
Lemma listcell_fold: ∀ key kp count p' p,
    cstring Ews key kp
    × malloc_token Ews (tarray tschar (Zlength key + 1)) kp
    × data_at Ews tcell (kp, (Vint (Int.repr count), p')) p
    × malloc_token Ews tcell p
          |-- list_cell key count p' p.
Proof.
   Admitted.
```

☐

```
Fixpoint listrep (sigma: list (list byte × Z)) (x: val) : mpred :=
 match sigma with
 | (s,c)::hs ⇒ EX y: val, list_cell s c y x × listrep hs y
 | nil ⇒
     !! (x = nullval) && emp
 end.
```

Exercise: 2 stars, standard (listrep_hints)

```
Lemma listrep_local_prop: ∀ sigma p, listrep
sigma p |--
        !! (is_pointer_or_null p ∧ (p=nullval ↔ sigma=nil)).
Proof.
   Admitted.
Hint Resolve listrep_local_prop : saturate_local.

Lemma listrep_valid_pointer:
  ∀ sigma p,
   listrep sigma p |-- valid_pointer p.
Proof.
   Admitted.
Hint Resolve listrep_valid_pointer : valid_pointer.
```

☐

```
Lemma listrep_fold: ∀ key count p' p al,
  list_cell key count p' p × listrep al p' |-- listrep ((key,count)::al) p.
Proof. intros. simpl. Exists p'. cancel. Qed.
```

A *listbox* is a pointer to a pointer to a cons cell.

```
Definition listboxrep al r :=
 EX p:val, data_at Ews (tptr tcell) p r × listrep al p.

Definition uncurry {A B C} (f: A → B → C) (xy: A×B) : C :=
  f (fst xy) (snd xy).
```

```
Definition hashtable_rep (contents: hashtable_contents) (p: val) : mpred :=
  EX bl: list (list (list byte × Z) × val),
    !! (contents = map fst bl) &&
    malloc_token Ews thashtable p ×
    field_at Ews thashtable [StructField _buckets] (map snd bl) p
    × iter_sepcon (uncurry listrep) bl.
```

Exercise: 2 stars, standard (hashtable_rep_hints)

```
Lemma hashtable_rep_local_facts:
∀ contents p,
 hashtable_rep contents p |-- !! (isptr p ∧ Zlength contents = N).
   Admitted.
Hint Resolve hashtable_rep_local_facts : saturate_local.

Lemma hashtable_rep_valid_pointer: ∀ contents p,
 hashtable_rep contents p |-- valid_pointer p.
   Admitted.
Hint Resolve hashtable_rep_valid_pointer : valid_pointer.
```

□

Function specifications for hash table

```
Definition new_table_spec : ident × funspec :=
 DECLARE _new_table
 WITH gv: globals
 PRE [ ]
   PROP()
   PARAMS() GLOBALS(gv)
   SEP(mem_mgr gv)
 POST [ tptr thashtable ]
   EX p:val, PROP()
      RETURN (p)
      SEP(hashtable_rep empty_table p; mem_mgr gv).

Definition new_cell_spec : ident × funspec :=
 DECLARE _new_cell
 WITH s: val, key: list byte, count: Z, next: val, gv: globals
 PRE [ tptr tschar, tint, tptr tcell ]
   PROP()
   PARAMS(s; Vint (Int.repr count); next) GLOBALS(gv)
   SEP(cstring Ews key s; mem_mgr gv)
 POST [ tptr tcell ]
   EX p:val, PROP()
      RETURN(p)
      SEP(list_cell key count next p; cstring Ews key s;
```

```
          mem_mgr gv).
Definition get_spec : ident × funspec :=
 DECLARE _get
 WITH p: val, contents: hashtable_contents, s: val, sigma : list byte
 PRE [ tptr (Tstruct _hashtable noattr), tptr tschar ]
    PROP ()
    PARAMS (p; s)
    SEP (hashtable_rep contents p; cstring Ews sigma s)
 POST [ tuint ]
    PROP ( )
    RETURN (Vint (Int.repr (hashtable_get sigma contents)))
    SEP (hashtable_rep contents p; cstring Ews sigma s).
Definition incr_list_spec : ident × funspec :=
 DECLARE _incr_list
 WITH r0: val, al: list (list byte × Z), s: val,
      sigma : list byte, gv: globals
 PRE [ tptr (tptr tcell), tptr tschar ]
    PROP (list_get sigma al < Int.max_unsigned)
    PARAMS (r0; s) GLOBALS(gv)
    SEP (listboxrep al r0; cstring Ews sigma s; mem_mgr gv)
 POST [ tvoid ]
      PROP ( ) RETURN ()
      SEP (listboxrep (list_incr sigma al) r0;
           cstring Ews sigma s; mem_mgr gv).
Definition incr_spec : ident × funspec :=
 DECLARE _incr
 WITH p: val, contents: hashtable_contents, s: val,
      sigma : list byte, gv: globals
 PRE [ tptr (Tstruct _hashtable noattr), tptr tschar ]
    PROP (hashtable_get sigma contents < Int.max_unsigned)
    PARAMS (p; s) GLOBALS(gv)
    SEP (hashtable_rep contents p; cstring Ews sigma s; mem_mgr gv)
 POST [ tvoid ]
      PROP ( ) RETURN ()
      SEP (hashtable_rep (hashtable_incr sigma contents) p;
           cstring Ews sigma s; mem_mgr gv).
```

Putting all the funspecs together

```
Definition Gprog : funspecs :=
        ltac:(with_library prog [
                   strcmp_spec; strcpy_spec; strlen_spec; hash_spec;
                   new_cell_spec; copy_string_spec; get_spec; incr_spec;
```

incr_list_spec
]).

17.3 Proofs of the functions hash, *copy_string*, *new_cell*

Before attempting to prove body_hash, do Verif_strlib at least through body_strlen.

Exercise: 3 stars, standard (body_hash) Lemma body_hash: semax_body Vprog Gprog f_hash hash_spec.

```
Proof.
```

start_function.
unfold cstring in *.

In the PROP part of your loop invariant, you'll want to maintain $0 \leq i \leq$ Zlength *contents*. In the LOCAL part of your loop invariant, try to use something like

temp _c (Vbyte (Znth i (*contents* ++ [*Byte.zero*]))

instead of

temp _c (Znth i (map Vbyte (...)))

The reason is that temp _c (Vint x) or temp _c (Vbyte y) is much easier for Floyd to handle than temp _c X where X is a general formula of type **val**.

Late in the proof of the loop body, the lemma hashfun_snoc will be useful. *Admitted.*

□

Exercise: 3 stars, standard (body_copy_string) Lemma body_copy_string: semax_body Vprog Gprog f_copy_string copy_string_spec.

```
Proof.
```

start_function.
assert_PROP (Zlength *sigma* + 1 ≤ Ptrofs.max_unsigned) by *entailer*!.

Admitted.

□

Exercise: 3 stars, standard (body_new_cell) Lemma body_new_cell: semax_body Vprog Gprog f_new_cell new_cell_spec.

```
Proof.
```

Admitted.

□

17.4 Proof of the *new_table* function

17.4.1 Auxiliary lemmas about data-structure predicates

Exercise: 2 stars, standard (iter_sepcon_hints) Lemma iter_sepcon_listrep_local_facts:

```
 ∀ bl, iter_sepcon (uncurry listrep) bl
                          |-- !! Forall is_pointer_or_null (map snd bl).
Proof.
   Admitted.
Hint Resolve iter_sepcon_listrep_local_facts : saturate_local.
```

☐

Exercise: 2 stars, standard (iter_sepcon_split3) Lemma iter_sepcon_split3:

```
  ∀ {A}{d: Inhabitant A} (i: Z) (al: list A) (f: A → mpred),
   0 ≤ i < Zlength al →
  (iter_sepcon f al =
   iter_sepcon f (sublist 0 i al) × f (Znth i al)
    × iter_sepcon f (sublist (i+1) (Zlength al) al))%logic.
Proof.
intros.
rewrite ← (sublist_same 0 (Zlength al) al) at 1 by auto.
   Admitted.
```

☐

Exercise: 3 stars, standard (body_new_table) Lemma body_new_table_helper:

```
 ∀ p,
  data_at Ews thashtable (list_repeat (Z.to_nat N) nullval) p
  |-- field_at Ews thashtable [StructField _buckets]
        (list_repeat (Z.to_nat N) nullval) p ×
          iter_sepcon (uncurry listrep) (list_repeat (Z.to_nat N) ([], nullval)).
Proof.
intros.
unfold_data_at (data_at _ _ _ p).
   Admitted.
Lemma body_new_table: semax_body Vprog Gprog f_new_table new_table_spec.
Proof.
```

The loop invariant in this function describes a partially initialized array. The best way to do that is with something like,

data_at Ews thashtable (list_repeat (Z.to_nat i) nullval ++ list_repeat (Z.to_nat (N-i)) Vundef) p.

Then at some point you'll have to prove something about, data_at Ews thashtable (list_repeat (Z.to_nat (i + 1)) nullval ++ list_repeat (Z.to_nat (N - (i + 1))) Vundef) p

In particular, you'll have to split up

list_repeat (Z.to_nat (i + 1)) nullval

into

list_repeat (Z.to_nat i) nullval ++ list_repeat (Z.to_nat 1) nullval.

The best way to do that is `rewrite` ← `list_repeat_app'`. *Admitted.*

□

17.5 Proof of the get function

Exercise: 2 stars, standard (listrep_traverse) Consider this loop in the get function:

while (p) { if (strcmp(p->key, s)==0) return p->count; p=p->next; }

We will reason about linked-list traversal in separation logic using "Magic Wand as Frame" https://www.cs.princeton.edu/~appel/papers/wand-frame.pdf

When the loop is partway down the linked list, we can view the original list up to the current position as a "linked-list data structure with a hole"; and the current position points to a linked-list data structure that fills the hole. The "data-structure-with-a-hole" we reason about with separating implication, called "magic wand": (hole -* data-structure) which says, if you can conjoin this data-structure-with-a-hole with something-to-fill-the-hole, then you get the original data structure:

hole * (hole -* data-structure) |– data-structure

Before the loop, we have a precondition such as,

(listrep b0 p0; other_stuff)

After a few loop iterations, we have a situation like,

(listrep b p; (listrep b p -* listrep b0 p0); other_stuff)

If the loop reaches $p{=}{=}NULL$, then we have,

(listrep nil nullval; (listrep nil nullval -* listrep b0 p0); other_stuff)

The *listrep_traverse_×* lemmas in this exercise illustrate how to start a traversal, how to perform one iteration of the traversal, and how to finish a traversal.

```
Lemma listrep_traverse_start:
  ∀ p al,
    emp |-- listrep al p -* listrep al p.
  Admitted.

Lemma listrep_traverse_step:
  ∀ al key count p' p,
  list_cell key count p' p |--
     listrep al p' -* listrep ((key, count) :: al) p.
  Admitted.

Lemma listrep_traverse_step_example:
 ∀ kp key count al q p b0 p0,
```

```
  cstring Ews key kp ×
  (listrep ((key, count) :: al) p -* listrep b0 p0) ×
  malloc_token Ews (tarray tschar (Zlength key + 1)) kp ×
  data_at Ews tcell (kp, (Vint (Int.repr count), q)) p ×
  malloc_token Ews tcell p ×
  listrep al q
 |- listrep b0 p0.
Proof.
intros.
```

Hint: use *sep_apply* with the lemmas listcell_fold, listrep_traverse_step, wand_frame_ver, modus_ponens_wand. *Admitted.*

```
Lemma listrep_traverse_finish:
 ∀ al p,
  listrep nil nullval × (listrep nil nullval -* listrep al p)
 |- listrep al p.
```

Admitted.

□

Exercise: 3 stars, standard (body_get) Use the *listrep_traverse_×* lemmas as appropriate. `Lemma body_get: semax_body Vprog Gprog f_get get_spec.`

```
Proof.
start_function.
rename p into table.
pose proof (hashfun_inrange sigma).
unfold abbreviate in MORE_COMMANDS; subst MORE_COMMANDS.
```

This next command would not be part of an ordinary Verifiable C proof, it is here only to guide you through the bigger proof. `apply seq_assoc1; assert_after 1`

```
 (EX cts:list (list (list byte × Z) × val),
  PROP (contents = map fst cts )
  LOCAL (temp _h (Vint (Int.repr (hashfun sigma)));
         temp _table table; temp _s s)
  SEP (cstring Ews sigma s; malloc_token Ews thashtable table;
       field_at Ews thashtable [StructField _buckets] (map snd cts) table;
       iter_sepcon (uncurry listrep) cts))%assert.
{
 admit.
}
Intros cts; subst contents.
forward.
deadvars!.
autorewrite with norm.
```

The previous line's `autorewrite` works only because of hypothesis H. If you `clear` *H*; `autorewrite with` *norm* you'll see that the *Int.modu* is not eliminated.

`rewrite ←` N_eq.

The purpose of this rewrite is to preserve a little bit of abstraction in the proofs. Because N_eq is in the rep_lia Hint database, the *rep_lia* tactic "knows" that N=109.

`assert` (0 ≤ hashfun *sigma* mod N < N).

It is useful to put facts like this above the line, to support *rep_lia* in reasoning about conversions between Z and Int. *admit.*

assert_PROP (Zlength *cts* = N) `as` *H1*.

Put equations like this above the line, to support *list_solve*, *rep_lia*, and other tactics such as entailer! that call them. {

admit.

}

forward.

Where did this proof goal come from? *denote_tc_assert* is a "type-checking assertion", that is, "prove that a C expression evaluates without crashing." In this case, the expression was table→*buckets*[*b*], and we have to prove here that *b* is in bounds of the array, and that the b'th element of the array is, in fact, initialized.

The hypothesis H1 that you proved above is generally useful, and particularly in this proof right here. { *admit.*

}

`set` (*h* := hashfun *sigma* mod N) `in` *.

This next line would not be part of an ordinary Verifiable C proof, it is here only to guide you through the bigger proof. eapply semax_pre; [`instantiate` (1:=

```
PROP ( ) LOCAL (temp _p (snd (Znth h cts)); temp _s s)
SEP (cstring Ews sigma s; malloc_token Ews thashtable table;
     field_at Ews thashtable [StructField _buckets] (map snd cts) table;
     iter_sepcon (uncurry listrep) (sublist 0 h cts);
      listrep (fst (Znth h cts)) (snd (Znth h cts));
     iter_sepcon (uncurry listrep) (sublist (h + 1) (Zlength cts) cts)) | ].
```

{ *admit.*

}

`destruct` (Znth *h cts*) `as` [*b0 p0*] *eqn*:*Hbp0*.

`simpl`.

We have reached the while-loop that will walk down the linked list. The pointer *p0* is the pointer to the beginning of a list that represents the sequence *b0*. As the loop progresses, the loop variable *p* will move down the links, pointing to smaller sequences *b*.

We represent the list segment from *p0* to *p* by a magic-wand formula: listrep *b p* -* listrep *b0 p0*. This means a heaplet (portion of memory) that, if you join it with a heaplet representing listrep *b p*, would represent the entire listrep *b0 p0*.

Several of our SEP conjuncts will not be needed until after the loop is done. We can

hide them away in a single SEP-conjunct FRZL *FR1* by doing this command: *freeze FR1* := (*malloc_token* _ _ *table*) (field_at _ _ _ _ *table*)
(iter_sepcon _ _) (iter_sepcon _ _).

The *freeze* tactic "frames out" several conjuncts for a while, until later we *thaw FR1*.

```
forward_while
 (EX b: list (list byte × Z), EX p: val,
    PROP(

        )
    LOCAL (temp _p p; temp _s s)
    SEP (FRZL FR1 ; cstring Ews sigma s


        )).
```

×

admit.

×

admit.

×

As usual in a linked-list traversal, we want to dereference $p \rightarrow$key but we don't have a data_at conjunct for p, we have only listrep b p. So we have to unfold the listrep; but this is useful only if we know that $p \neq$nil. Therefore, case analysis: `destruct` b `as` [| [*key count*] *al*].

This case, where b=nil, is impossible, because (if you have the right loop invariant) certain information in the SEP clause of the precondition is inconsistent with *HRE*: isptr p above the line. To make use of propositional information in the SEP clause, use assert_PROP: {
assert_PROP **False**. {
admit.
}
contradiction.
}

`idtac`.

The structure of the rest of this * bullet goes like this:

- Massage the precondition and get through the forward_call to function *strcmp*.
- Do *forward_if* (*vret≠Int.zero*). The argument you pass to forward_if can be a Prop, it does not need to be a full PROP/LOCAL/SEP, because:
  - One branch of the if never reaches the join point, it returns; and
  - The other branch of the if does not modify any local variables or memory, so the LOCAL and SEP parts of the assertion will be unchanged.

- - Sub-bullet: then-clause. At some point in this proof, you'll need to *thaw FR1*. You'll need to use iter_sepcon_split3. Near the end of the then-clause, you'll have a goal similar (perhaps not identical) to listrep_traverse_step_example.
  - Sub-bullet: else-clause. Fairly short and straightforward proof.
  - Sub-bullet: after the if; another proof goal similar to listrep_traverse_step_example.

admit.
×

Here, you still have a data structure with a hole, represented by (A -* B), and the thing-in-the-hole, represented by A. This is similar to listrep_traverse_finish. *admit.*

Admitted.

□

17.6 Proof of the *incr_list* function

Above, in the proof of the get function, we traverse a linked list without modifying it. The loop invariant looked like, listrep b p × (listrep b p -* listrep $b0$ $p0$).

But the *incr_list* function modifies a linked list, perhaps inserting a new element at the end. Furthermore, the C program's loop variable `struct` *cell* **r is a pointer to a pointer to a list cell. Our separation-logic description of this is listboxrep. `Print` listboxrep.

That is, r is a single-word box containing pointer p, and p is a listrep. Let's examine the loop that we want to verify:

void incr_list (struct cell **r0, char *s) { struct cell *p, **r; for(r=r0; ; r=&p->next) { p = *r; if (!p) { *r = new_cell(s,1,NULL); return; } if (strcmp(p->key, s)==0) {p->count++; return;} } }

We will describe variable r something like this:

PROP() *LOCAL*(temp _r r) *SEP*(data_at Ews (tptr tcell) q r).

That is, pointer to a single word containing q. But when we do $r = \&(p{\rightarrow}next)$ we will have r pointing into the middle of a `struct` *cell* record, at the *next* field. To describe that single field all alone, we use *unfold_data_at* to split

data_at Ews tcell (x,y,q) p

into three separate conjuncts:

field_at Ews tcell StructField _key x p * field_at Ews tcell StructField _count y p * field_at Ews tcell StructField _next q p

and then we must rewrite the third conjunct into

data_at Ews (tptr tcell) q (field_address tcell StructField _next p)

where the (field_address _ _ _) is an "address arithmetic" expression that describes the offset, in bytes, from p to $\&(p{\rightarrow}next)$.

The listboxrep_traverse lemma illustrates the situation at the end of the loop body. Look at the left-hand side of the |– entailment. Variable r points to a single word containing p (it is perhaps the _next field of the previous list cell). Variable p points to a split-up list cell, with fields _key and _count. Where is the _next field of p? We choose not to describe it in this heaplet!

The right-hand-side of this heaplet says, if you provide a heaplet satisfying listboxrep at address $\&p \rightarrow next$ representing the sequence dl, then the combined heaplet satisfies listboxrep at address r representing the sequence (key,$count$)::dl. Furthermore, this is true for *any* sequence dl. That provides the freedom for the program to modify the contents of $p \rightarrow next$, or of any _next field later in the sequence.

Exercise: 3 stars, standard (listboxrep_traverse) Lemma listboxrep_traverse:

```
  ∀ p kp key count r,
     cstring Ews key kp ×
     malloc_token Ews (tarray tschar (Zlength key + 1)) kp ×
     field_at Ews tcell [StructField _key] kp p ×
     field_at Ews tcell [StructField _count] (Vint (Int.repr count)) p ×
     malloc_token Ews tcell p ×
     data_at Ews (tptr tcell) p r
   |-
     ALL dl: list (list byte × Z),
       listboxrep dl (field_address tcell [StructField _next] p)
       -* listboxrep ((key, count) :: dl) r.
Proof.
  intros.
 apply allp_right; intro dl.
 apply → wand_sepcon_adjoint.
```

Sometime during the proof below, you will have data_at Ews tcell ... p that you want to expand into

field_at Ews tcell StructField _key ... p

17.7 field_at Ews tcell StructField _count ... p

17.8 field_at Ews tcell StructField _next ... p].

You can do this with *unfold_data_at* (pattern) where pattern is something like (data_at _ _ _ p) indicating which SEP conjunct you want to expand.

After that, you will want to rewrite by field_at_data_at ...

Check (field_at_data_at Ews tcell [StructField _next]). Eval simpl in (nested_field_type tcell [StructField _next]).

Admitted.

□

Exercise: 4 stars, standard (body_incr_list) Lemma body_incr_list: semax_body Vprog Gprog f_incr_list incr_list_spec.
Proof.

This proof uses "magic wand as frame" to traverse *and update* a (linked list) data structure. This pattern is a bit more complex than the wand-as-frame pattern used in body_get, which did not update the data structure. You will still use "data-structure-with-a-hole" and "what-is-in-the-hole"; but now the "data-structure-with-a-hole" must be able to accept the *future* hole-filler, not the one that is in the hole right now.

The key lemmas to use are, *wand_refl_cancel_right*, *wand_frame_elim'*, and wand_frame_ver. When using wand_frame_ver, you will find listboxrep_traverse to be useful. *Admitted.*

□

17.9 field_compatible

Let's discuss how to address individual fields of structs, or individual slots of arrays.

First, data_at *sh* (Tstruct _ _) is equivalent to the conjunction of individual field_at predicates for each of the fields. (Something similar holds for arrays.) Lemma example_split_struct:

```
 ∀ p (x y z: val),
      data_at Ews tcell (x,(y,z)) p
    = (field_at Ews tcell [StructField _key] x p
      × field_at Ews tcell [StructField _count] y p
      × field_at Ews tcell [StructField _next] z p)%logic.
Proof.
intros.
unfold_data_at (data_at _ _ _ p).
rewrite sepcon_assoc.
reflexivity.
Qed.
```

Second, field_at *sh t gfs z p* is equivalent to data_at *sh* (tptr *t*) *z* (field_address *t gfs p*), that is, field_address is a way to describe the offset (in bytes) from the base of a struct to the address of a field of that struct (or similarly to an element of an array).

```
Lemma example_field_at_data_at:
 ∀ p (z: val),
    field_at Ews tcell [StructField _next] z p =
    data_at Ews (tptr tcell) z
     (field_address tcell [StructField _next] p).
Proof.
intros.
 rewrite field_at_data_at.
 reflexivity.
Qed.
```

The _next field of struct *cell* is the third field, after two integer fields. If there is no padding between those fields (which is the case here), then the distance from the base of the struct to the base of the _next field should be 2*sizeof(tint).

```
Lemma example_field_at_data_at':
 ∀ p (z: val),
   field_at Ews tcell [StructField _next] z p |-
   data_at Ews (tptr tcell) z
    (offset_val (2 × sizeof tint) p).
Proof.
intros.
 rewrite field_at_data_at.
 unfold field_address.
 if_tac; simpl; auto.
 cancel.
 entailer!.
 destruct H0 as [H0 _].
 contradiction H0.
Qed.
```

But the converse does not hold: Lemma example_field_at_data_at'':

```
 ∀ p (z: val),
   data_at Ews (tptr tcell) z
    (offset_val (2 × sizeof tint) p)
 |- field_at Ews tcell [StructField _next] z p.
Proof.
 intros.
 rewrite field_at_data_at.
 simpl.
 unfold field_address.
 rewrite if_true.
 simpl.
 cancel.
Abort.
```

If we assume an extra premise, we can prove this, however: Lemma example_field_at_data_at''':

```
 ∀ p (z: val),
  field_compatible tcell [StructField _next] p →
   data_at Ews (tptr tcell) z
    (offset_val (2 × sizeof tint) p)
 |- field_at Ews tcell [StructField _next] z p.
Proof.
 intros.
 rewrite field_at_data_at.
 simpl.
```

```
 unfold field_address.
 rewrite if_true by assumption.
 simpl.
 cancel.
Qed.
```

Why is that? The answer is in the definition of field_address: `Print field_address.`

```
Print field_compatible.
```

_count is a field of `struct` *cell* `Goal (legal_nested_field tcell [StructField _count]).`

```
compute; auto 10.
Qed.
```

_s is not a field of `struct` *cell* `Goal (~ legal_nested_field tcell [StructField _s]).`

```
compute. intuition congruence.
Qed.
```

0 is a legal subscript of `struct` *cell* $\times a[109]$ `Goal (legal_nested_field (tarray (tptr tcell) 109) [ArraySubsc 0]).`

```
compute. intuition congruence.
Qed.
```

108 is a legal subscript of `struct` *cell* $\times a[109]$ `Goal (legal_nested_field (tarray (tptr tcell) 109) [ArraySubsc 108]).`

```
compute. intuition congruence.
Qed.
```

109 is not a legal subscript of `struct` *cell* $\times a[109]$ `Goal (~ legal_nested_field (tarray (tptr tcell) 109) [ArraySubsc 109]).`

```
compute. intuition congruence.
Qed.
```

Sometimes in the C language we want a pointer *just at the end* of an array. That is, given `struct` *cell* $\times a[109]$ we want the pointer value $\&a[109]$. This is legal, even though this slot of the array does not exist.

For this we want a variant of legal_nested_field that permits "just at the end": `Check legal_nested_field0.`

108 is an an addressible subscript of `struct` *cell* $\times a[109]$ `Goal (legal_nested_field0 (tarray (tptr tcell) 109) [ArraySubsc 108]).`

```
compute. intuition congruence.
Qed.
```

109 is an an addressible subscript of `struct` *cell* $\times a[109]$ `Goal (legal_nested_field0 (tarray (tptr tcell) 109) [ArraySubsc 109]).`

```
compute. intuition congruence.
Qed.
```

110 is not an addressible subscript of `struct` *cell* $\times a[109]$ `Goal (~ legal_nested_field (tarray (tptr tcell) 109) [ArraySubsc 110]).`

`compute. intuition congruence.`
`Qed.`

Based on these two notions,

- legal_nested_field (loadable/storable field) and
- legal_nested_field0 (addressible field),

we have, respectively field_compatible and field_compatible0.

`Print` field_compatible0.

17.9.1 Where does field_compatible come from?

Let's look again at this lemma: `Check` example_field_at_data_at'''.

We can apply this lemma if field_compatible... is above the line. How can we get that hypothesis above the line? Often the entailer or entailer! does this automatically, deriving it from data_at or field_at facts in the SEP clause of your left-hand side.

17.10 Proof of the incr function

```
void incr_list (struct cell **r0, char *s);

void incr (struct hashtable *table, char *s) {
  unsigned int h = hash(s);
  unsigned int b = h % N;
  incr_list (& table->buckets[b], s);
}
```

The difficult part here is the function-argument, & table$\rightarrow$*buckets*[*b*] . The precondition of the *incr_list* function requires just a single pointer-to-pointer-to-cell, but we have an entire array of 109 pointers-to-cell.

We start with table, a pointer to a struct containing one field that's an array of 109 elements. For calling *incr_list*, we need to split that into two separate data structures:

- the single-element array at table$\rightarrow$*buckets*$+b$
- all the rest of the data structure, including the other fields of `struct` *hashtable* (if there were any) and the elements 0..b-1 and b+1..108 of the array.

The wand_slice_array lemma can do this:

`Check` wand_slice_array.

Here (array_with_hole sh t lo hi n al p) means `Print` array_with_hole.

This says that data_at *sh* (tarray *t* *n*) *al* *p* can be split up into two pieces:

- 1. the slice from index *lo* to index *hi*-1,
- 2. everything else.

Later in the proof of body_incr, you need to handle the function-argument, & (table→*buckets*[*b*]) , where variable _b has the value *h*. CompCert will calculate this as table + (sizeof **int**)**h*, which is to say,

offset_val (sizeof (tptr tcell) * h) table

But we want to express that as [field_address0 (tarray (tptr tcell) N) [ArraySubsc *h*] (field_address thashtable [StructField _buckets] table).

As discussed above in *the* [field_compatible] *section, to prove a* field_address *one must know that the address* [table] is `field`-*compatible* with [thashtable], *and that the address* [table→*buckets*] is `field`-*compatible* with *the* [ArraySubsc *h*] `field`. *That's all proved* in *the following lemma*:

```
Lemma body_incr_field_address_lemma:
  ∀ (table: val) (h : Z),
  0 ≤ h < N →
  field_compatible (tarray (tptr tcell) N) []
    (field_address thashtable [StructField _buckets] table) →
  field_compatible (tptr tcell) []
    (field_address0 (tarray (tptr tcell) N)
        [ArraySubsc h]
        (field_address thashtable [StructField _buckets] table)) →
  offset_val (sizeof (tptr tcell) × h) table =
  field_address0 (tarray (tptr tcell) N) [ArraySubsc h]
    (field_address thashtable [StructField _buckets] table).
Proof.
  intros.
```

The Hint database allows `auto with` field_compatible to make use of the field_compatible facts above the line. `rewrite` field_address0_offset `by auto with` *field_compatible.*

```
  rewrite field_address_offset by auto with field_compatible.
  autorewrite with norm. auto.
Qed.
```

Exercise: 4 stars, standard (body_incr) Lemma body_incr: semax_body Vprog Gprog f_incr incr_spec.

```
Proof.
start_function.
rename p into table.
rename H into Hmax.
assert_PROP (isptr table) as Htable by entailer!.
```

The next two lines would not be part of an ordinary Verifiable C proof; they are here only to guide you through the bigger proof. `subst` *MORE_COMMANDS*; `unfold ab-`

```
breviate; match goal with ⊢ semax _ _ (Ssequence ?c1 (Ssequence ?c2 ?c3)) _ ⇒ apply
(semax_unfold_seq (Ssequence (Ssequence c1 c2) c3)); [ reflexivity | ] end.
pose (j := EX cts: list (list (list byte × Z) × val), PROP (contents = map fst cts; 0 ≤
hashfun sigma mod N < N; Zlength cts = N) LOCAL (temp _b (Vint (Int.repr (hashfun sigma
mod N))); temp _h (Vint (Int.repr (hashfun sigma))); temp _table table; temp _s s; gvars gv)
SEP (cstring Ews sigma s; malloc_token Ews thashtable table; data_at Ews (tarray (tptr tcell)
N) (map snd cts) (field_address thashtable [StructField _buckets] table); iter_sepcon (uncurry
listrep) cts; mem_mgr gv)); apply semax_seq' with j; subst j; abbreviate_semax.
{
 admit.
 }

Intros cts.
subst contents.
unfold hashtable_get in Hmax.
rewrite Zlength_map, H1 in Hmax.
set (h := hashfun sigma mod N) in *.
erewrite (wand_slice_array h (h+1) N _ (tptr tcell))
  by first [rep_lia | list_solve ].
```

For the remainder of the proof, here are some useful lemmas: sublist_len_1, sublist_same, *sublist_map*, *data_at_singleton_array_eq*, iter_sepcon_split3, *iter_sepcon_app*, sublist_split, field_at_data_at, *wand_slice_array_tptr_tcell*

Admitted.

□

Chapter 18

Library VC.Postscript

18.1 Postscript: Postcript and bibliography

18.2 Looking back

You've now seen how to verify programs that use many of C's data structures (arrays, pointers, structs, integers) and control structures (functions, if-then-else, loops), and abstraction structures (data abstraction, module interfaces). The final exercise (hash tables) demonstrated all of these at once, in addition to a key concept: layering the proof using a functional model, so that proofs about the properties of the functional model (Hashfun) cleanly separate from the implementation proof (Verif_hash) showing that the C program refines the functional model.

18.3 Looking forward

If you want to verify some C programs on your own, you may take inspiration from some of these Verifiable C proofs:

- SHA-2 cryptographic hashing *Appel* 2015 (in Bib.v)
- HMAC cryptographic authentication *Beringer* 2015 (in Bib.v)
- HMAC-DRBG cryptographic random number generation *Ye* 2017 (in Bib.v)
- Concurrent messaging system *Mansky* 2017 (in Bib.v)
- Generational copying garbage collector *Wang* 2019 (in Bib.v)
- Bins-based malloc/free system *Appel and Naumann* 2020 (in Bib.v)
- AES encryption, B+Trees, Tries of B+ trees (unpublished master's or undergraduate theses).

18.3.1 Small examples

VST is distributed with a *progs* directory of many small C programs that demonstrate different features and methods of Verifiable C. If your VST installation is in the standard place, you can find this as a subdirectory of '*coqc* `-where`'/*user-contrib*/*VST*.

18.3.2 Modules

All the verifications in this volume are single-file C programs. Real software engineering in C is done with modules in .c files and interfaces in .h files. Since 2019, VST has a module system; the early version is described in *Beringer* 2019 (in Bib.v) and the newer version is demonstrated in progs/VSUpile. The description in *Beringer* 2019 (in Bib.v) mostly matches the proofs in progs/VSUpile, but the proofs handle data abstraction in a more advanced way than is described in the paper.

18.3.3 Input/output

To prove I/O using our Verifiable C logic, we treat the state of the outside world as a resource in the SEP clause, alongside (but separating from) in-memory conjuncts. Proved examples are in *progs/io.c*, *progs/io_mem.c*, and their proof files *progs/verif_io.v*, *progs/verif_io_mem.v*. Research papers describing these concepts include *Koh* 2020 (in Bib.v) and *Mansky* 2020 (in Bib.v).

18.4 Looking around

VST is not the only program verification system. How should you decide which system to use?

18.4.1 Static analyzers

There are many *static analyzers* – too numerous to list here – that attempt to check *safety* of your program: will it crash? Will it access an array out of bounds, or dereference an uninitialized pointer? Static analyzers can be useful in software engineering, but they have two major limitations:

- They don't verify *functional correctness* – that is, does your program produce the right answer, or interact with the right behavior?
- They are incomplete. For example, proving that $a[i]$ is an in-bounds array access requires proving that $0 \leq i < \mathsf{N}$. Sometimes a static analyzer can deduce that from the program flow, but in general it is a functional correctness property that may require sophisticated invariants to prove.

18.4.2 Functional correctness verifiers – functional languages

A good way to write proved-correct software is to program in a pure functional program logic, and use higher-order logic to prove correctness. For example:

- Write pure functional programs in Coq, prove them correct in Coq, then extract them from Coq to OCaml or Haskell. See Volume 3 of Software Foundations: Verified Functional Algorithms.
- In HOL systems (Higher-order Logic) such as Isabelle/HOL, you can do the same thing: write functional programs, prove them correct, extract, compile.
- You can write Haskell programs, compile with *ghc*, and import into Coq for verification using hs-to-coq *Spector-Zabusky* 2018 (in Bib.v).
- ACL2 is an older system, that uses a first-order logic. That's less expressive, you don't get polymorphism or quantification, but there have been many successful applications of ACL2 in industry.

18.4.3 Functional correctness verifiers – imperative languages

Functional programming has its limitations: it requires a garbage collector, usually written in an imperative language, and how did you prove that correct? In functional languages it is often harder to build (and reason about) low-latency code, or to access low-level primitives. For these and other reasons, some software is still written in low-level imperative languages such as C.

There are verifiers for high-level imperative languages (that require a garbage collector). For example, Dafny *Leino* 2010 (in Bib.v) is a language and tool for Hoare-logic style verification. It's relatively simple to learn and elegant to use.

ML with mutable references and arrays is also a high-level imperative language. CFML is a system for reasoning about imperative ML programs using separation logic in Coq *Chargueraud* 2010 (in Bib.v), soon to be described in another volume of *Software Foundations*. CakeML is a system for proving (and correctly compiling) ML programs in HOL *Kumar* 2014 (in Bib.v).

18.4.4 Functional correctness verifiers – C

For the canonical *low-level imperative language* – C – there are several systems:

- Frama-C (https://frama-c.com/)
- VeriFast *Jacobs* 2011 (in Bib.v)
- VST (the subject of this volume).

Unlike VST, where (as you have seen) the proof script is separate from the program, in both Frama-C and VeriFast you prove the program by inserting assertions and invariants into the program. the tools complete the verification automatically – or else, point out which assertions have failed, so you can adjust your assertions and invariants, and try again.

Each of these three systems has an assertion language, in which you express your function specifications, assertions, and invariants.

- In VST, as you have seen, that language is a separation language (PROP/LOCAL/SEP) embedded in Coq, so that the PROP, LOCAL, and SEP clauses can all make use of the full expressive power of the Calculus of Inductive Constructions (CiC). You have seen a simple example of the expressive power of this approach, where we can use ordinary Coq proofs in Hashfun, and directly connect them to separation-logic proofs in Verif_hash.
- Frama-C uses a weaker assertion language, expressed in C syntax. That's a much weaker logic to reason in, and it doesn't directly connect to a general-purpose logic (and proof assistant) like Coq. Also, since Frama-C is not a separation logic, it can be difficult to reason about data structures.
- VeriFast uses a capable Dafny-like logic – even more capable, since it is separation logic, not just Hoare logic. That means you can reason about data structures. There's no artificial limitation to a C-like syntax in the assertion language. Unfortunately, VeriFast's assertion language is not connected to a general-purpose logic (and proof assistant); that means you can do refinement proofs (this C program implements that functional model), but it's not so easy to reason about properties of your functional models.

18.4.5 Foundational soundness

Formal reasoning about source programs is a good thing – but once you've proved your source program correct, how do you know that is compiled correctly? That is,

- the compiler must not have bugs, and
- the compiler must agree with your verifier on every detail of the semantics of the source language.

Of all the systems described above, only VST and CakeML can make this connection end-to-end, with machine-checked proofs.

18.5 Conclusion

This volume has given you a taste of formal verification for a low-level imperative language. Since C was not designed with verification in mind, it has many sharp corners and idiosyncrasies, which the verification tool cannot always hide from you. But C is a *lingua franca* in which you can express a wide variety of programming paradigms, and Verifiable C is expressive enough to allow to verify them. We wish you success in your future software verification efforts.

Chapter 19

Library VC.Bib

19.1 Bib: Bibliography

19.2 Resources cited in this volume

Appel 2014 Program Logics for Certified Compilers, by Andrew W. Appel with Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. Cambridge University Press, 2014.

Appel 2015 Verification of a Cryptographic Primitive: SHA-256, by Andrew W. Appel, ACM Transactions on Programming Languages and Systems 37(2) 7:1-7:31, April 2015. https://www.cs.princeton.edu/~appel/papers/verif-sha-2.pdf

Appel and Naumann 2020 Verified sequential malloc/free, by Andrew W. Appel and David A. Naumann, in 2020 ACM SIGPLAN International Symposium on Memory Management, June 2020. https://www.cs.princeton.edu/~appel/papers/memmgr.pdf

Beringer 2015 Verified Correctness and Security of OpenSSL HMAC, by Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 24th USENIX Security Symposium, pages 207-221, August 2015. https://www.cs.princeton.edu/~appel/papers/verified-hmac.pdf

Beringer 2019 Abstraction and Subsumption in Modular Verification of C Programs, by Lennart Beringer and Andrew W. Appel. In: Maurice H. ter Beek and Annabelle McIver: Formal Methods – the next 30 years. Proceedings of the 23rd International Symposium on Formal Methods (FM'19), pages 573-590, Springer, 2019. https://www.cs.princeton.edu/~appel/papers/fu

Chargueraud 2010 Program verification through characteristic formulae, by Arthur Chargueraud, in Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, pp. 321-332, 2010. https://dl.acm.org/doi/pdf/10.1145/1863543.1863590

Jacobs 2011 VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java, by Bart Jacobs, Jan Smans, Pieter Philippaerts, Frederic Vogels, Willem Penninckx, and Frank Piessens. In Proc. NASA Formal Methods (NFM) 2011. https://people.cs.kuleuven.be/~bart.jacobs/nfm2

Koh 2020 From C To Interaction Trees: specifying, verifying and testing a networked server, by Nicolas Koh, Yao Li, Yishuai Li, Li-Yao Xia, Lennart Beringer, Wolf Honore, William Mansky, Benjamin C Pierce, and Steve Zdancewic. CPP 2019 Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, Pages 234-248, ACM, 2019 https://dl.acm.org/citation.cfm?doid=3293880.3294106

Kumar 2014 CakeML: a verified implementation of ML, by Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens, in POPL'14, ACM SIGPLAN Notices 49, no. 1 (2014): 179-191.

Leino 2010 Dafny: An automatic program verifier for functional correctness, by K. Rustan M. Leino, International Conference on Logic for Programming Artificial Intelligence and Reasoning, pp. 348-370, 2010.

Mansky 2017 A verified messaging system, by William Mansky, Andrew W. Appel, and Aleksey Nogin. OOPSLA'17: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, October 2017. PACM/PL volume 1, issue OOPSLA, paper 87, 2017.

Mansky 2020 Connecting Higher-Order Separation Logic to a First-Order Outside World, by William Mansky, Wolf Honoré, and Andrew W. Appel, ESOP 2020: European Symposium on Programming, April 2020. https://www.cs.princeton.edu/~appel/papers/connecting-esop.pdf

Spector-Zabusky 2018 Total Haskell is reasonable Coq, by Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. CPP 2018: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, January 2018. https://doi.org/10.1145/3167092

Wang 2019 Certifying Graph-Manipulating C Programs via Localizations within Data Structures, by Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. OOPSLA: Conference on Object-Oriented Programming Systems, Languages, and Applications; PACM/PL volume 3, issue OOPSLA, 2019. https://dl.acm.org/doi/pdf/10.1145/3360597

Ye 2017 Verified Correctness and Security of mbedTLS HMAC-DRBG, by Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel, CCS'17: ACM Conference on Computer and Communications Security, October 2017. https://www.cs.princeton.edu/~appel/papers/verified-hmac-drbg.pdf