Verified Compiler Technology and Separation Logic for Reasoning about Concurrent C Programs

Andrew W. Appel

Princeton University

Research supported by:

Formally verified software

1. Write specification for

   (safety or correctness of) software

2. Prove software meets its specification
   - Must use automated tools to check the proof
   - Must use automated tools to *help* construct the proof

Is this even feasible?

Does it scale? Is it cost-effective?

Progress over past decades

Is this even feasible?

Does it scale?

Is it cost-effective?

Operational Semantics

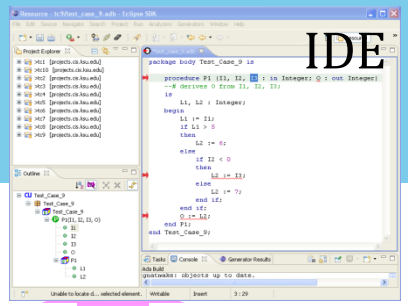
Type systems and program logics

Dependently typed logics

Tactical proof assistants

Faster processors, bigger memories

Software toolchain



Static Analyzer
(or Program Verifier)

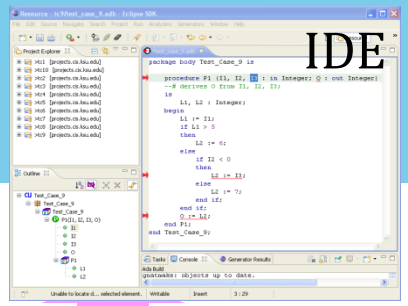
Compiler

Runtime System
(or Operating System)

Want to reason about safety/security/correctness of software at *source-code level*

… but software executes as machine code

Software toolchain



Static Analyzer
(or Program Verifier)

Compiler

Runtime System
(or Operating System)

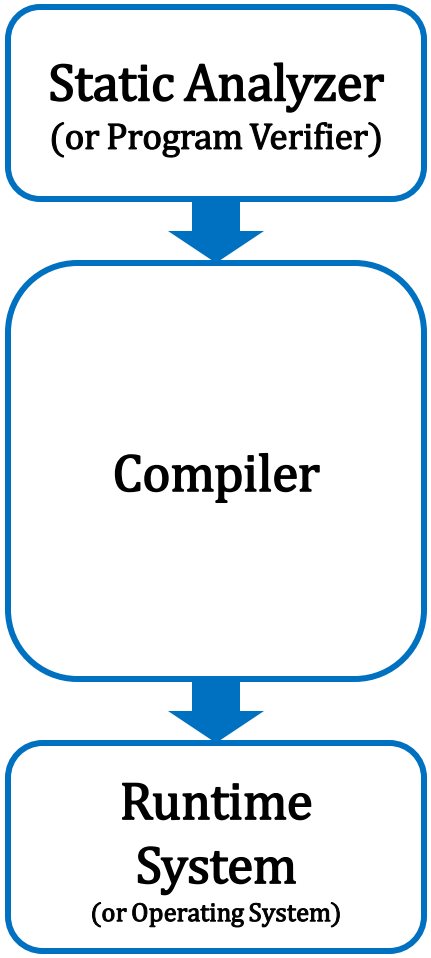
Where adversary is permitted to provided less-trusted (but still analyzed/verified) program modules,

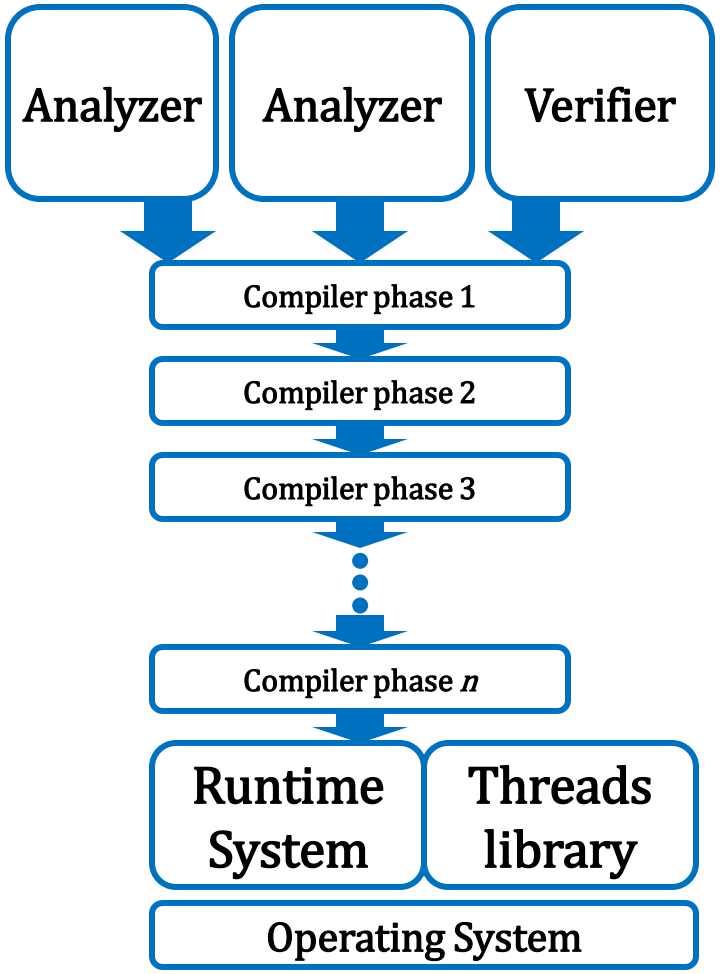
we assume that adversary can exploit bugs in the software tools to sneak attack-code past the verifiers

Therefore we want:
1. proofs that quantify over all possible adversaries,
2. Software toolchain *not* in the trusted base!

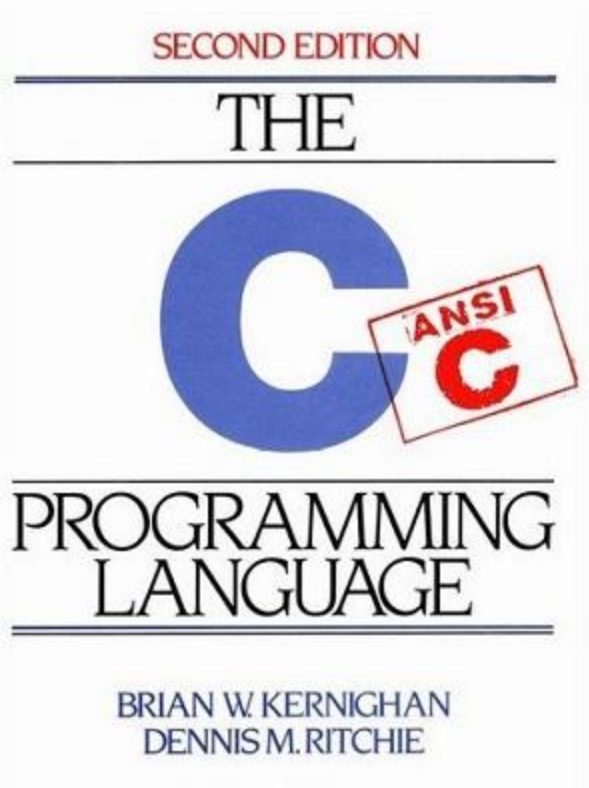
Software toolchain (what it really looks like)

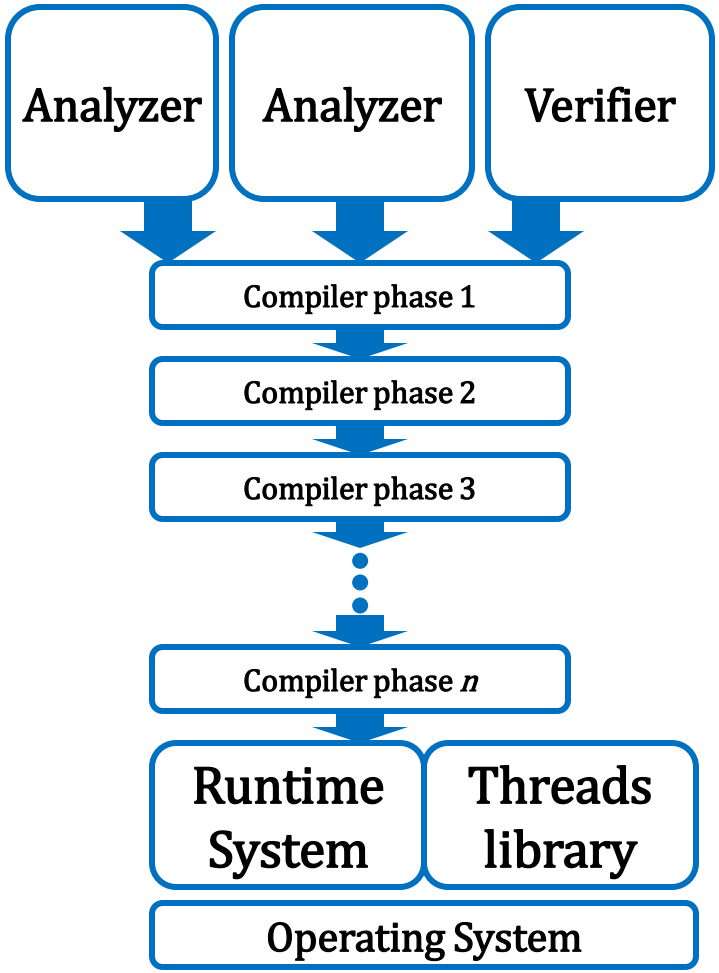
Therefore, we need a modular approach to a verified toolchain



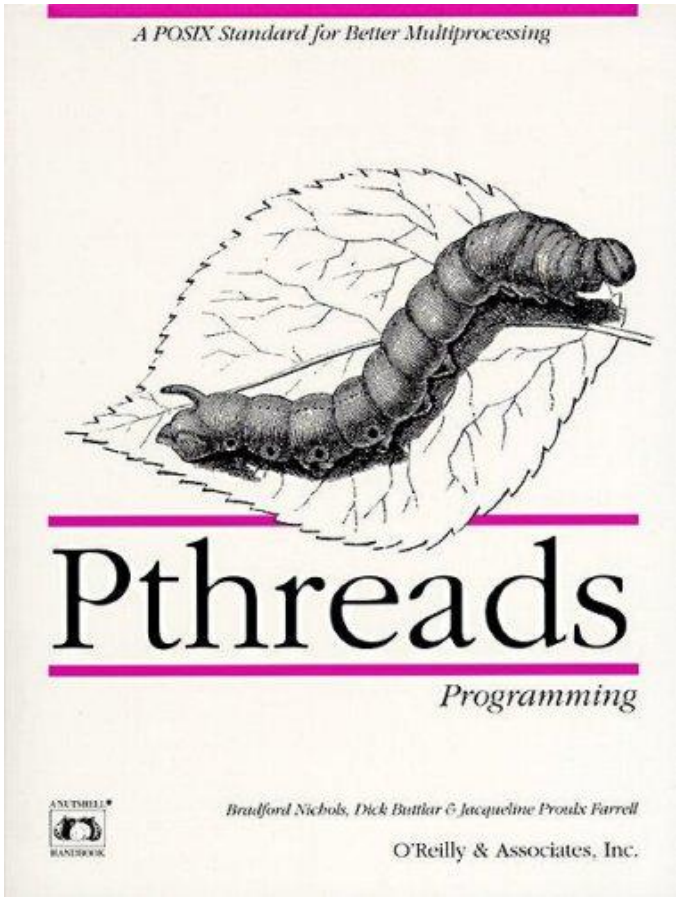


… for a real programming language

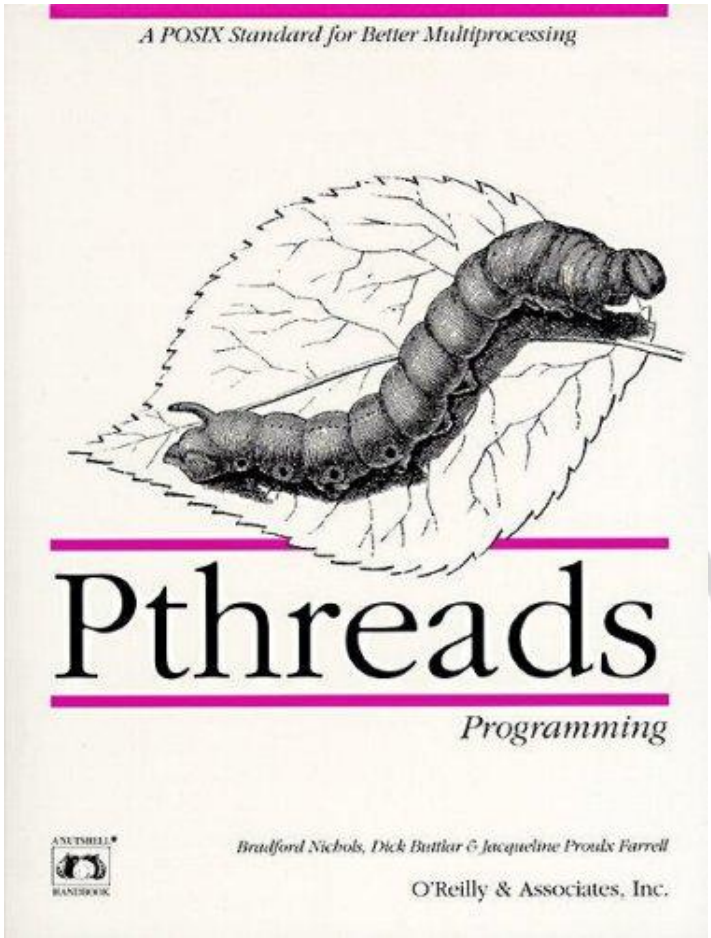


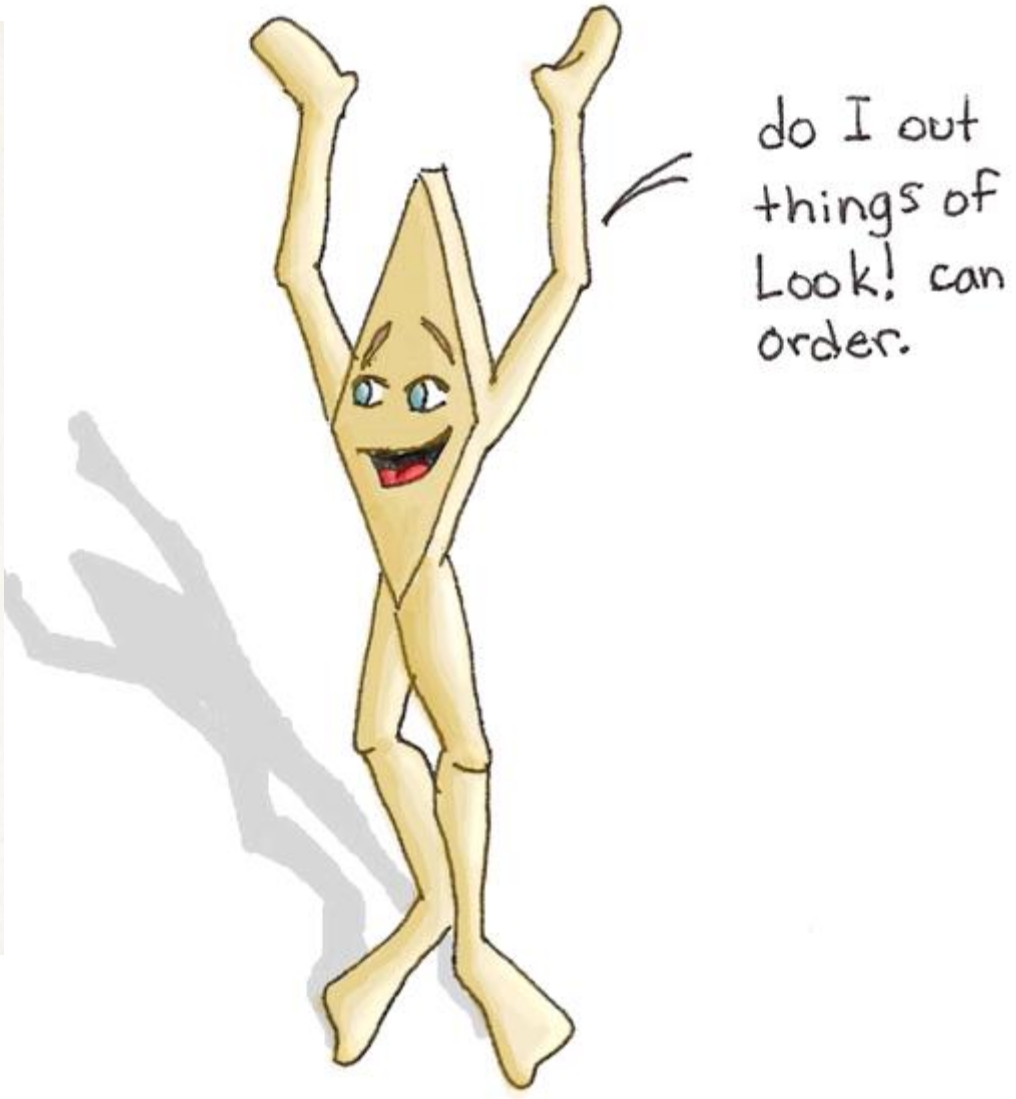


… with a concurrency library

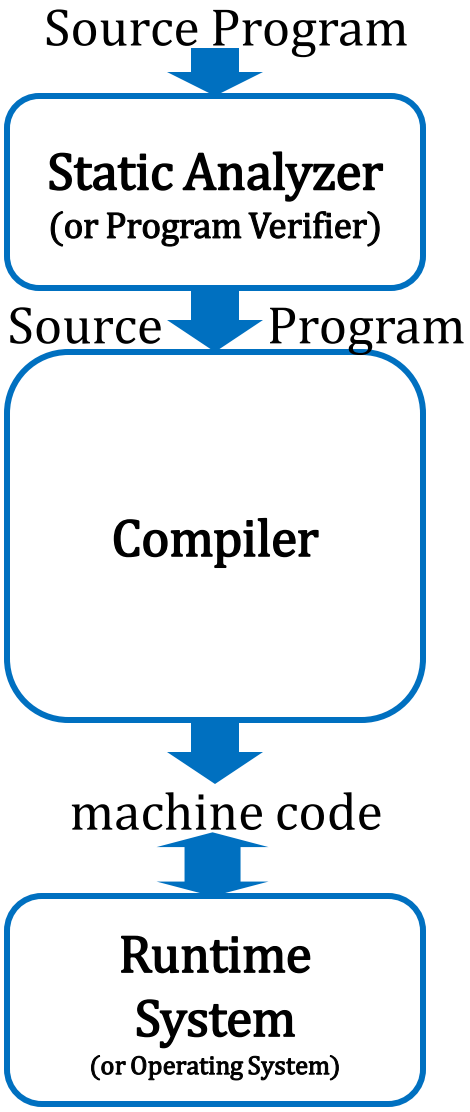


… in a relaxed memory model



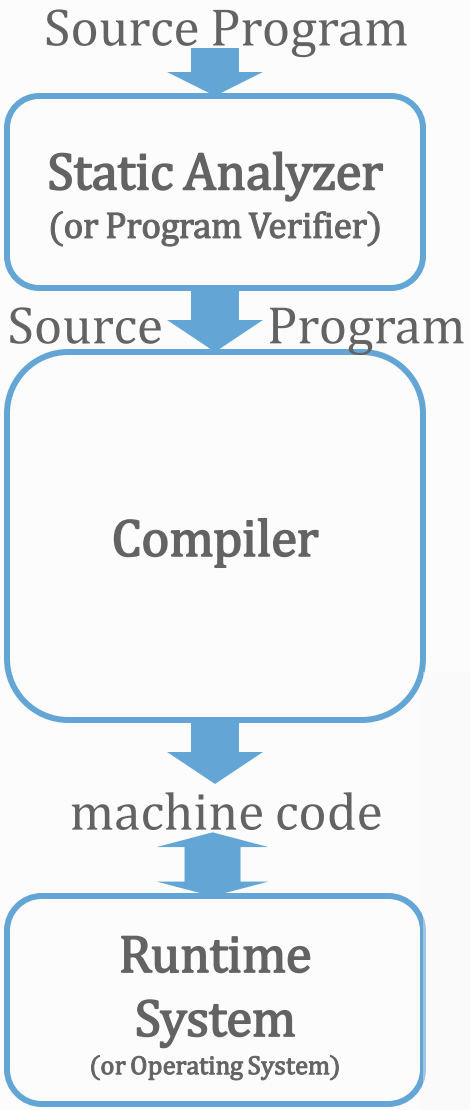


Soundness



"Whatever property the static analysis claims about the source program, that property will *provably* hold on the execution of the object program when running in the runtime system"

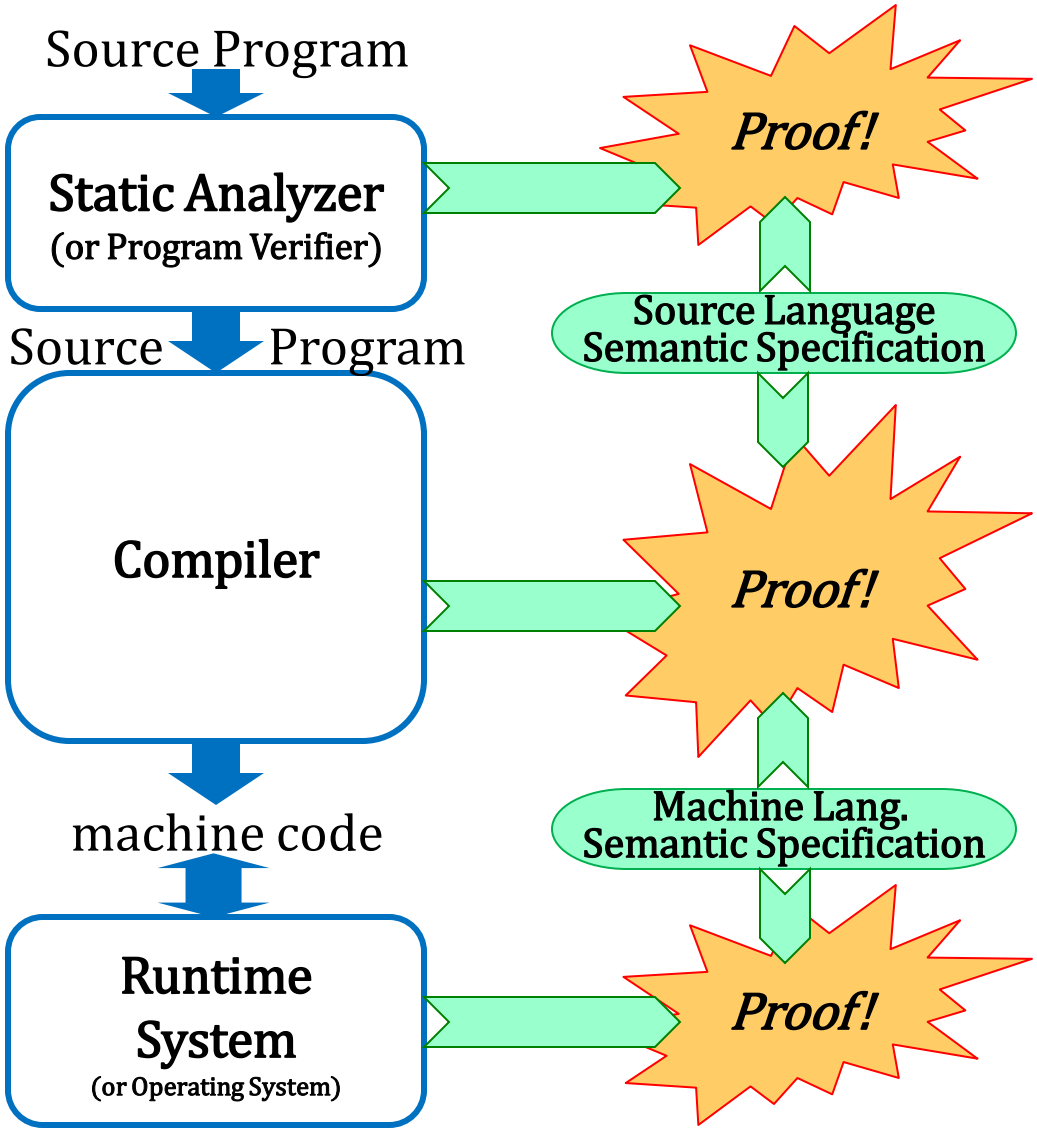
In a proof assistant



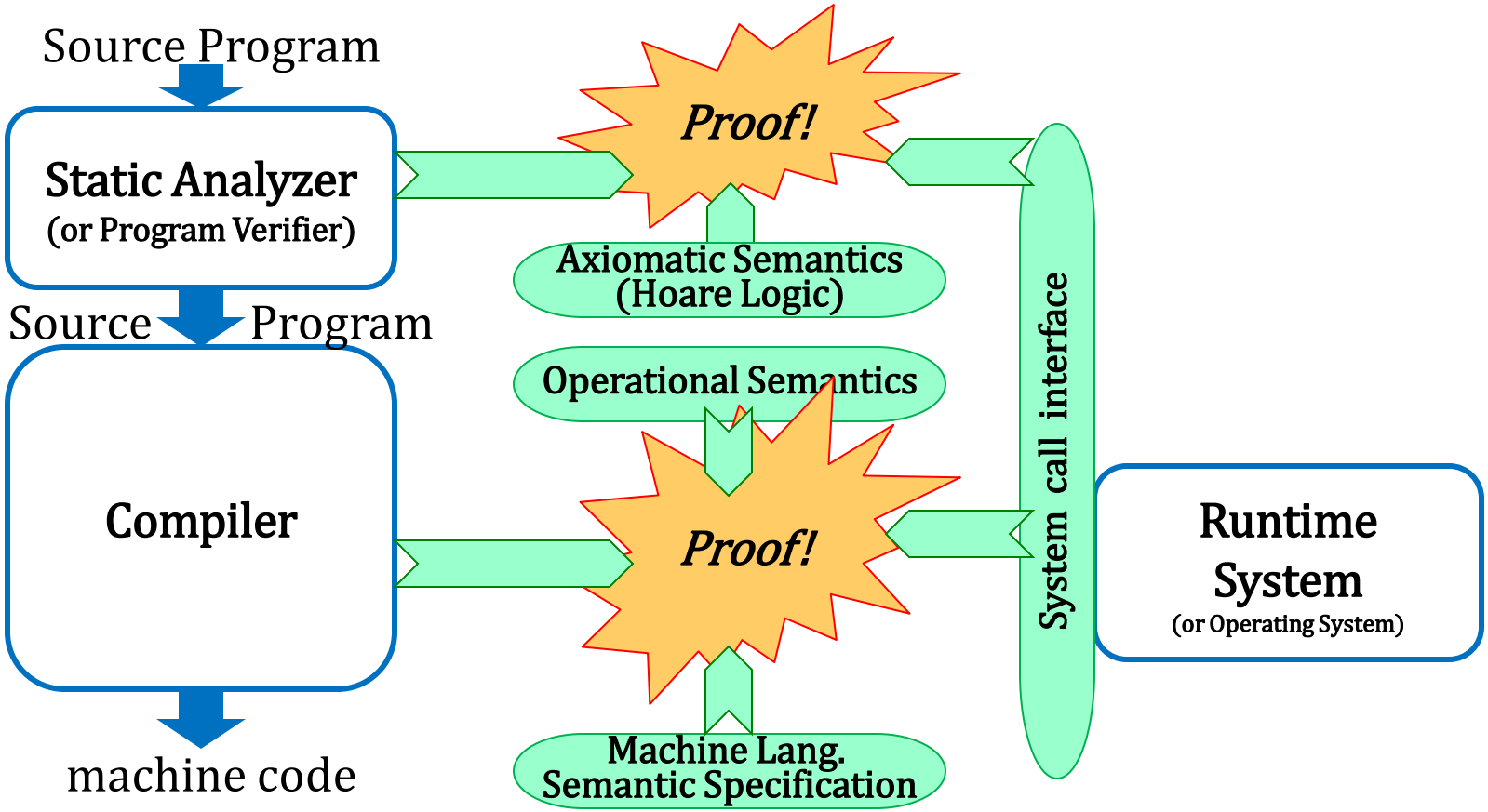
"Whatever property the static analysis claims about the source program, that property will *provably* hold on the execution of the object program when running in the runtime system"

With a machine-checked proof, of course.

Modularity


Source Program
Static Analyzer
(or Program Verifier)
Source Program
Compiler
machine code
Runtime System
(or Operating System)
Proof!
Source Language Semantic Specification
Proof!
Machine Lang. Semantic Specification
Proof!

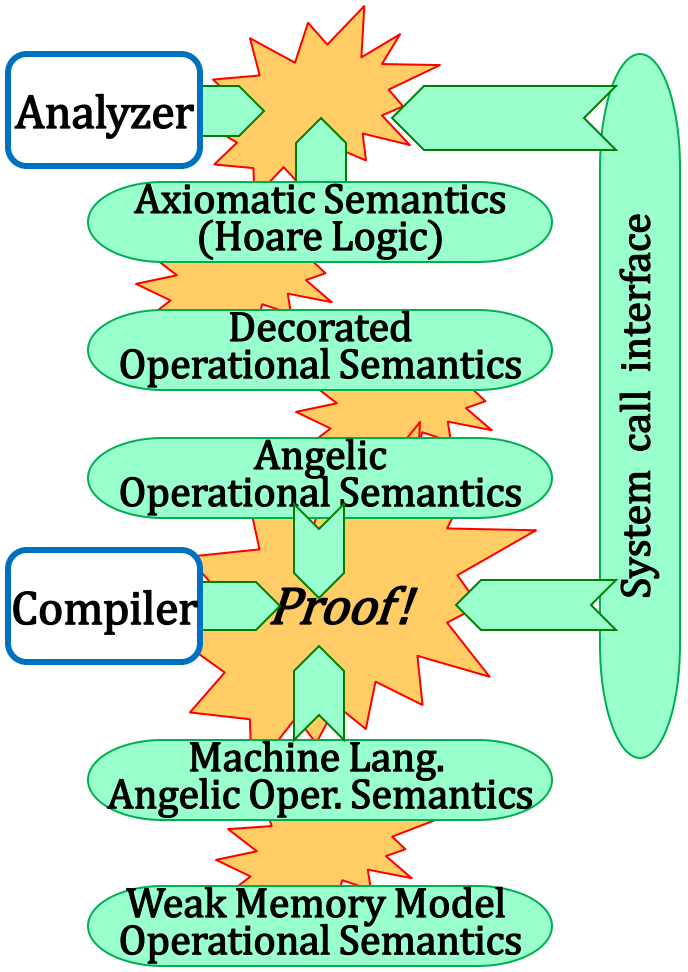
Problems at specification interfaces



Factor semantics to separate core language from external function calls.
Same ext calls from source/target languages.

Chain of proofs in the Verified Toolchain

1. Expressive program logic for source programs
2. Static analyzer infers invariants, emits them expressed in program logic
3. Core static analyzer written in Gallina checks invariants against program
4. Prove static analyzer sound w.r.t. rules of program logic
5. Redesign and refactor operational semantics with rich notion of observables
6. Prove program logic sound w.r.t. **decorated** operational semantics of source
7. Prove erasure theorem: decorated→angelic operational semantics
8. (CompCert) Prove compiler correct w.r.t. angelic op. semantics
9. Prove erasure theorem: angelic→erased operational semantics (extra credit for using weak memory models)
10. Prove that operating system implements its specification ("just" use the same verified toolchain, i.e. with source language)



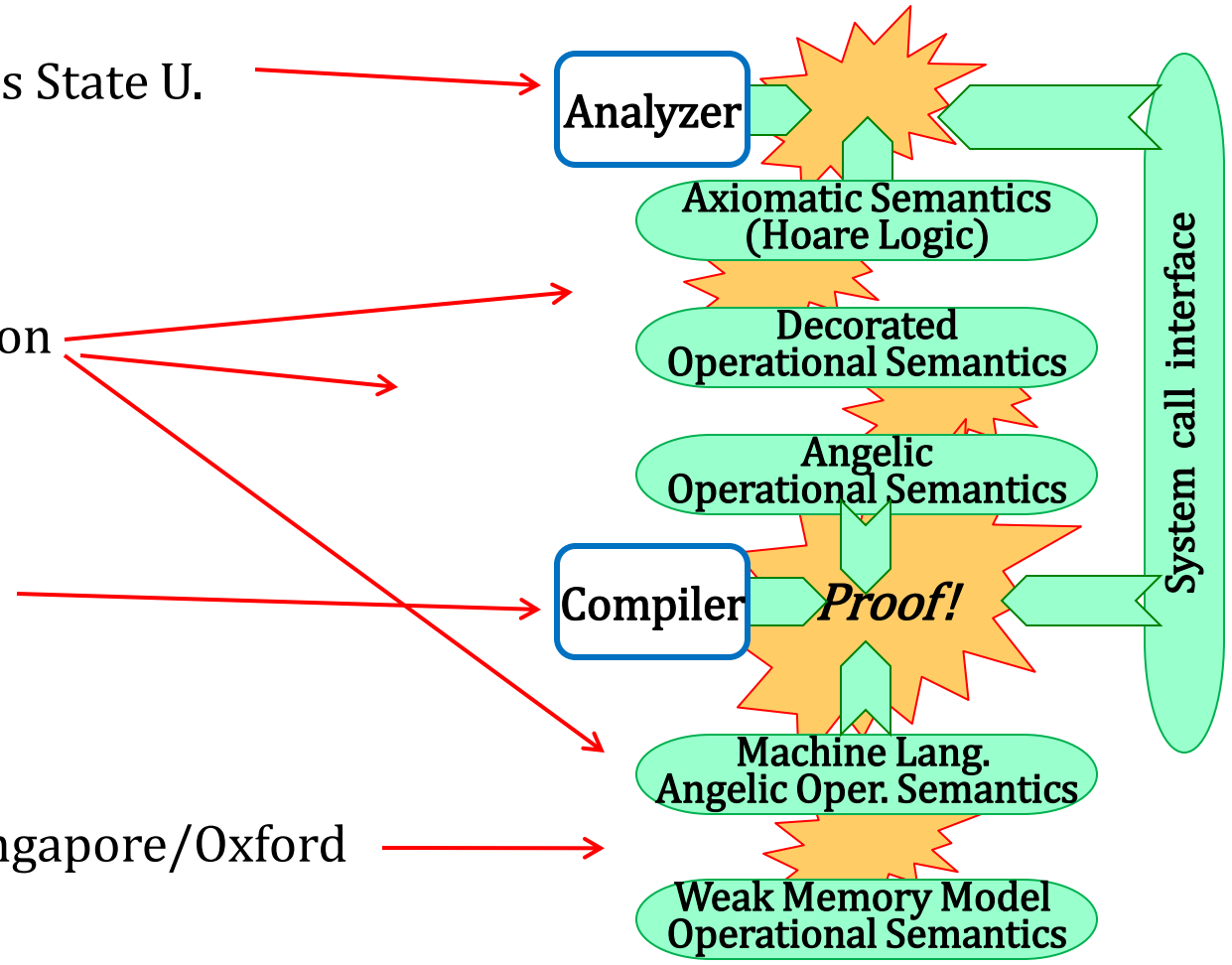
Enough work for 4 research groups

Hatcliff et al., Kansas State U.

Appel et al., Princeton

Leroy et al. , INRIA

Hobor+Alglave, Singapore/Oxford



PROGRAM LOGIC

What program logic?

Want to reason about
C programs with pointer data structures

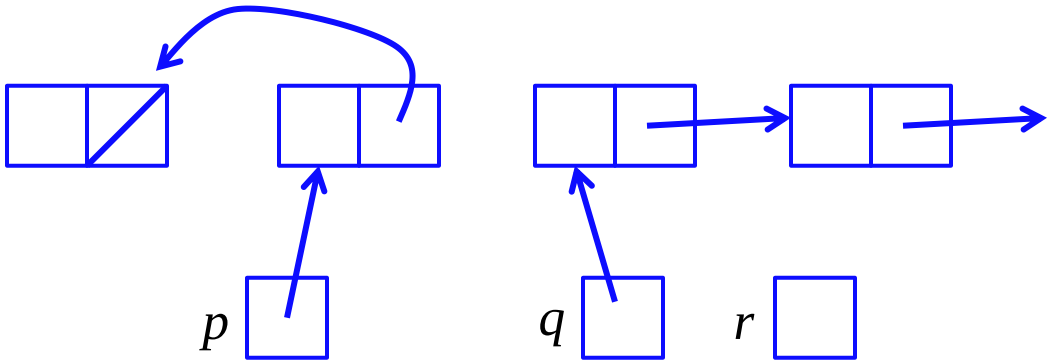
Conventional Hoare logics don't deal well with pointers and aliasing

Solution:

Separation Logic
(O'Hearn and Reynolds 2000)

$p \mapsto 5 \quad * \quad q \mapsto 8$

Separation logic example



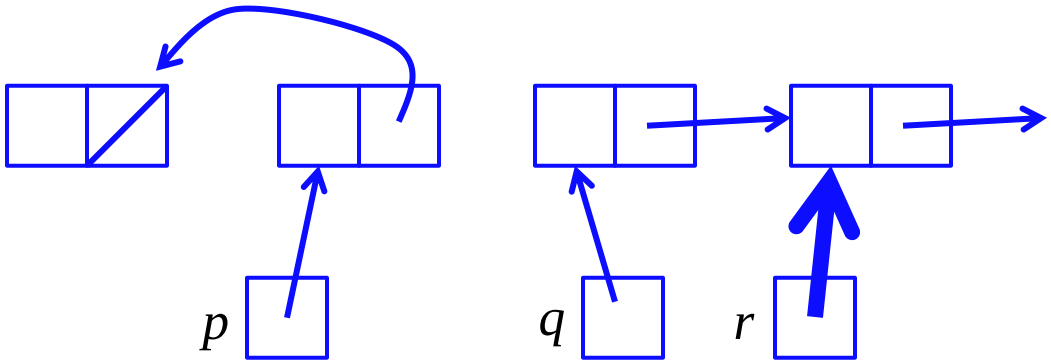
$\text{list}(p) \ * \ \text{list}(q) * q \neq \text{nil}$

```
r := q→next
```

```
q→next := p
```

```
p:= q
```

Separation logic example



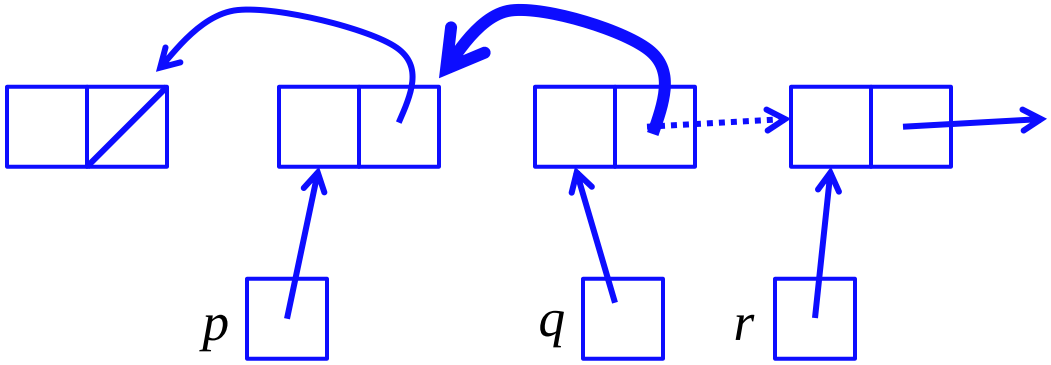
$\text{list}(p) \ * \ \text{list}(q) * q \neq \text{nil}$

```
r := q→next
```

```
q→next := p
```

```
p:= q
```

Separation logic example



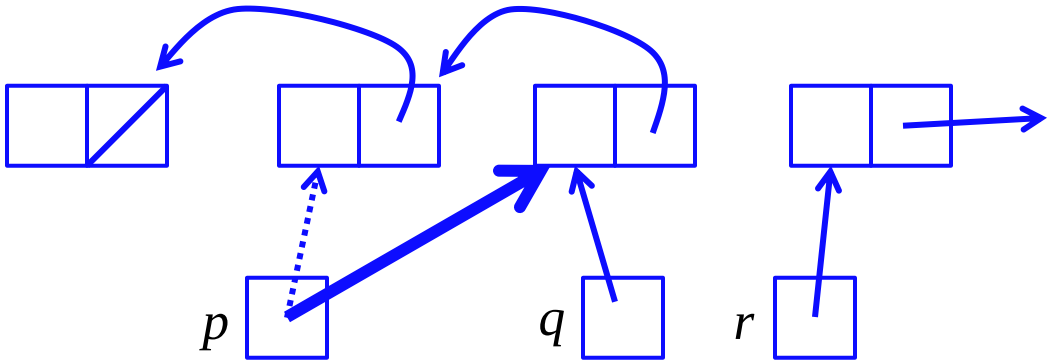
$\text{list}(p) \ * \ \text{list}(q) * q \neq \text{nil}$

```
r := q→next
```

```
q→next := p
```

```
p:= q
```

Separation logic example



$\text{list}(p) \ * \ \text{list}(q) * q \neq \text{nil}$

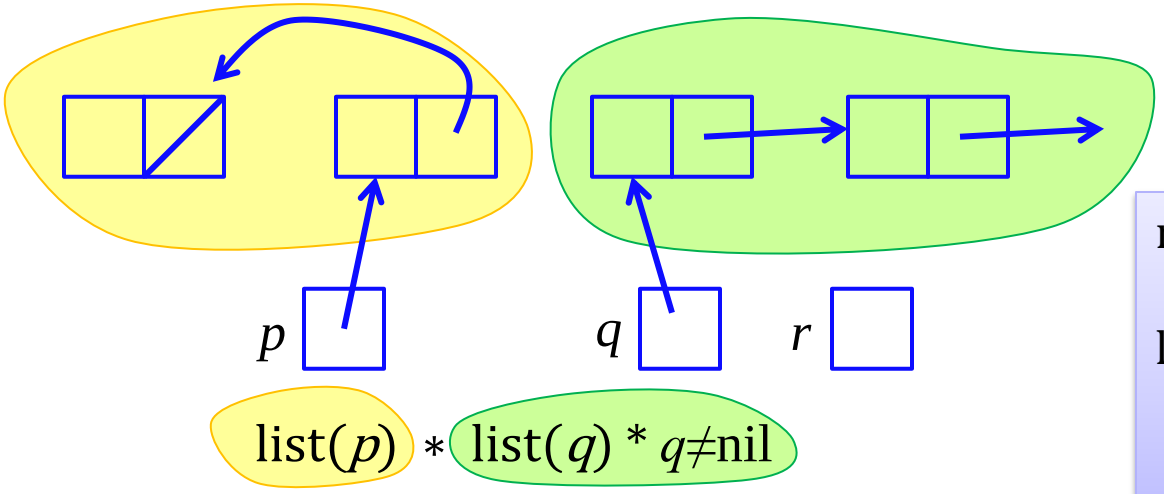
```
r := q→next
```

```
q→next := p
```

```
p:= q
```

$\text{list}(p) \ * \ \text{list}(q)$

Separation logic example

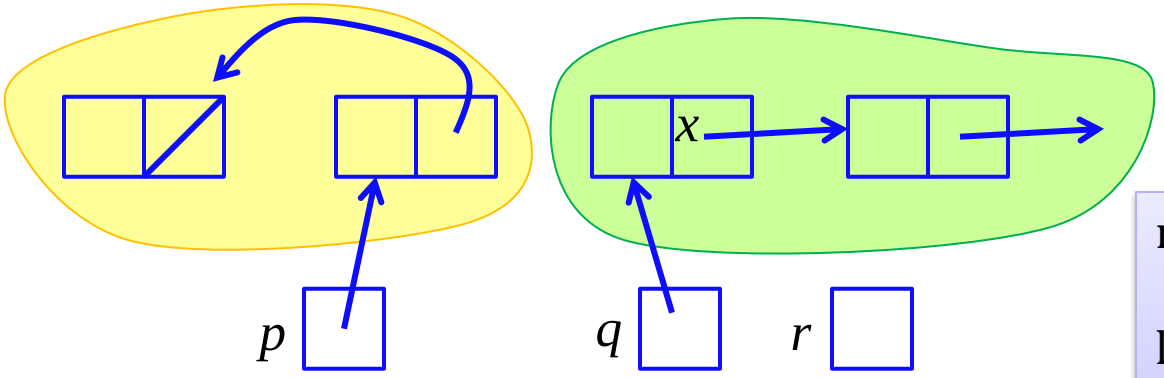


nil = 0

list(*p*) =
(*p* = 0) ∨
(∃*x*. *p* ↦ _ ∗ *p*+1 ↦ *x* ∗ list(*x*))

```
r := q→next

q→next := p

p:= q
```

Separation logic example



$\text{list}(p) \ * \ \text{list}(q) * q \neq \text{nil}$

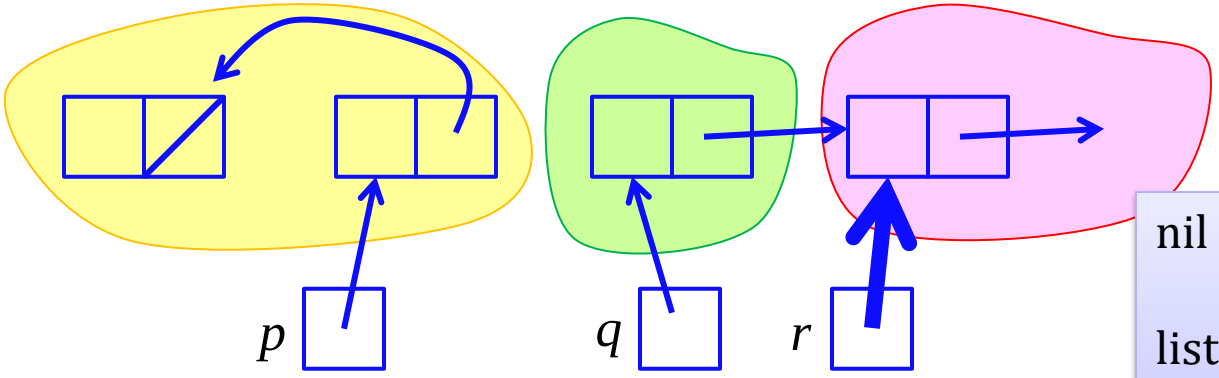
$\text{list}(p) \ * \ \exists x.\ q \mapsto \_ \ * \ q+1 \mapsto x \ * \ \text{list}(x)$

```
r := q→next

q→next := p

p := q
```

$\text{nil} = 0$

$\text{list}(p) =$
$(p = 0) \ \vee$
$(\exists x.\ p \mapsto \_ * p+1 \mapsto x * \text{list}(x))$

Separation logic example



nil = 0

list(p) =
(p =0) ∨
($\exists x.\ p \mapsto \_ * p{+}1 \mapsto x * \text{list}(x)$)

$\text{list}(p) * \text{list}(q) * q \neq \text{nil}$

$\text{list}(p) * \exists x.\ q \mapsto \_ * q{+}1 \mapsto x * \text{list}(x)$

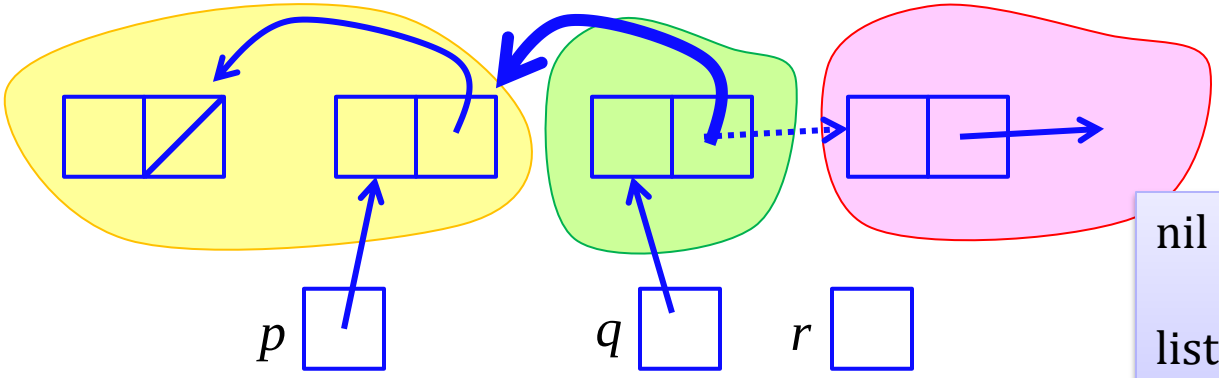
`r := q→next`

$\text{list}(p) * q \mapsto \_ * q{+}1 \mapsto r * \text{list}(r)$

`q→next := p`

`p:= q`

Separation logic example



nil = 0

list(p) =
(p = 0) ∨
($\exists x.\ p \mapsto \_ * p{+}1 \mapsto x * \text{list}(x)$)

$\text{list}(p) * \text{list}(q) * q \neq \text{nil}$

$\text{list}(p) * \exists x.\ q \mapsto \_ * q{+}1 \mapsto x * \text{list}(x)$

`r := q→next`

$\text{list}(p) * q \mapsto \_ * q{+}1 \mapsto r * \text{list}(r)$

`q→next := p`

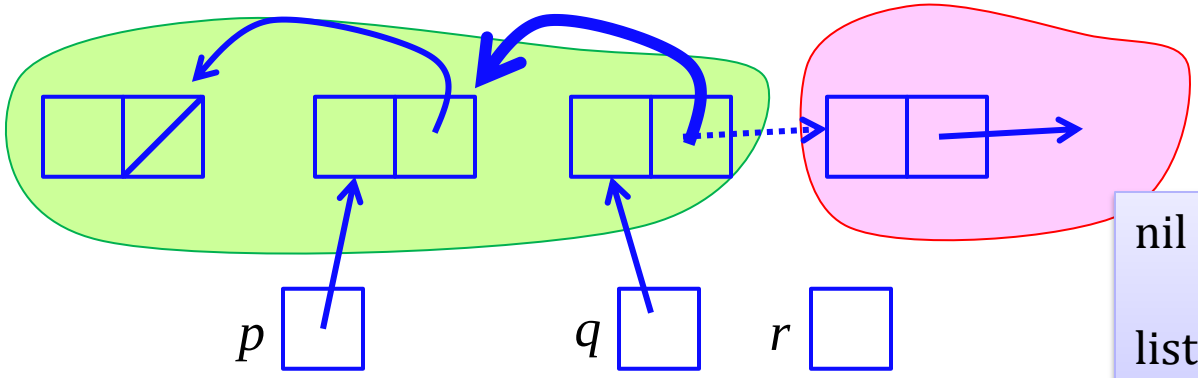
$\text{list}(p) * q \mapsto \_ * q{+}1 \mapsto p * \text{list}(r)$

$\text{list}(q) * \text{list}(r)$

`p := q`

$\text{list}(p) * \text{list}(r)$

Separation logic example



nil = 0

list(p) =
(p = 0) ∨
($\exists x.\ p \mapsto \_ * p{+}1 \mapsto x * \text{list}(x)$)

$\text{list}(p) * \text{list}(q) * q \neq \text{nil}$

$\text{list}(p) * \exists x.\ q \mapsto \_ * q{+}1 \mapsto x * \text{list}(x)$

`r := q→next`

$\text{list}(p) * q \mapsto \_ * q{+}1 \mapsto r * \text{list}(r)$

`q→next := p`

$\text{list}(p) * q \mapsto \_ * q{+}1 \mapsto p * \text{list}(r)$

$\text{list}(q) * \text{list}(r)$

`p := q`

$\text{list}(p) * \text{list}(r)$

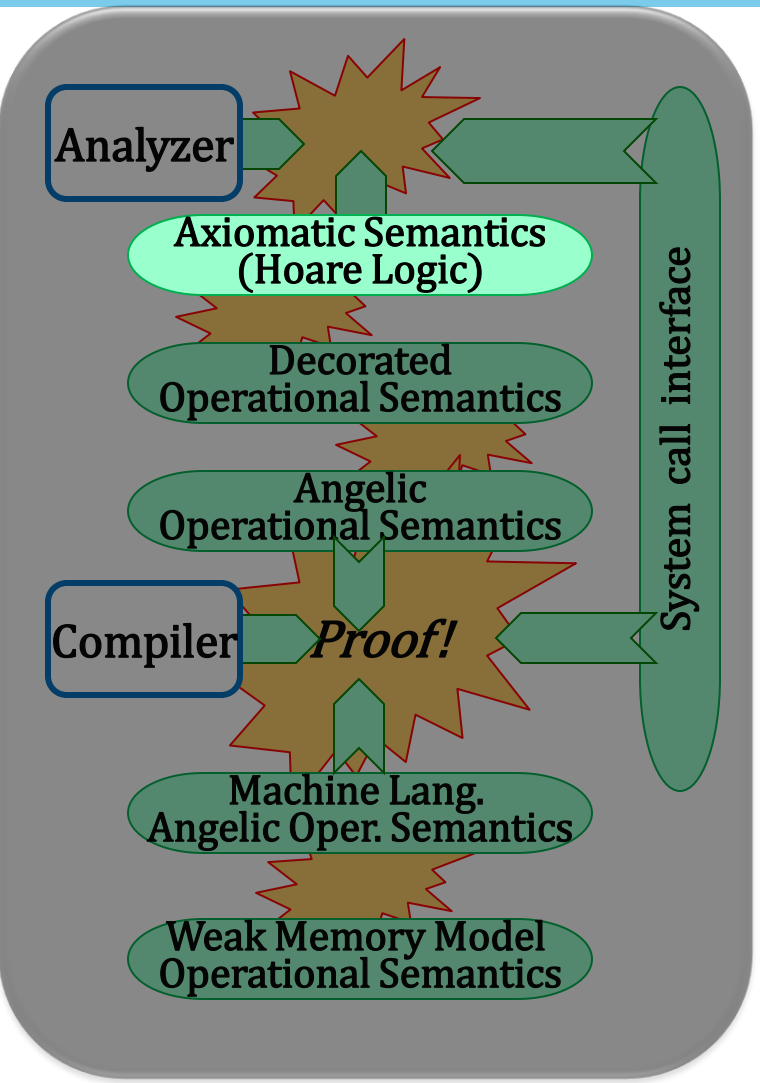
Why separation logic?

Everything that Hoare Logic can do, plus:

Pointer update, aliasing,
modular local reasoning

Keeps track of "who owns what"
(in the static reasoning, not via run-time permissions)

Higher-order impredicative concurrent separation logic



Program-variable values: $v \Downarrow 7$

Memory values: $p \mapsto 5$

Separation: $P * Q$

And, or $P \wedge Q$ $P \vee Q$

Recursion $\boldsymbol{\mu}\, x.\, P$

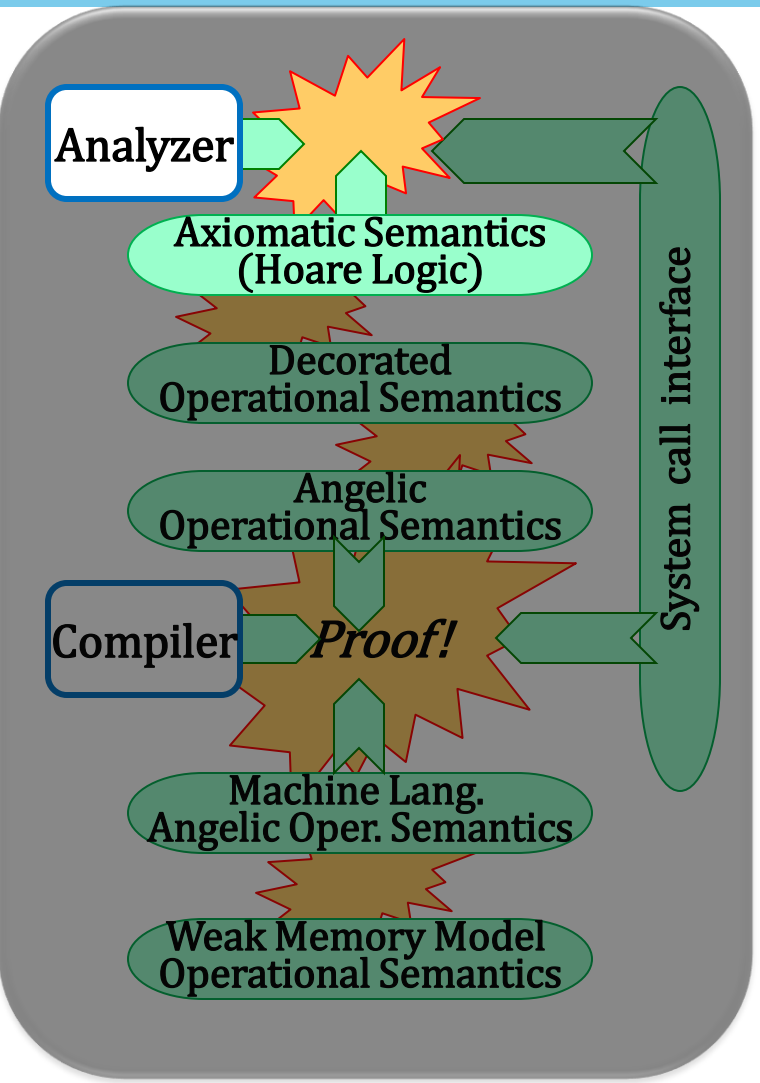
Impredicative quantification $\forall x.P$ $\exists x.P$

Function pointers $f : P \to Q$

Lock resource invariants $\ell \rightsquigarrow R$

PROVING CORRECTNESS OF THE TOOLCHAIN

Correctness of the static analyzer



Emit program invariants expressed in the program logic

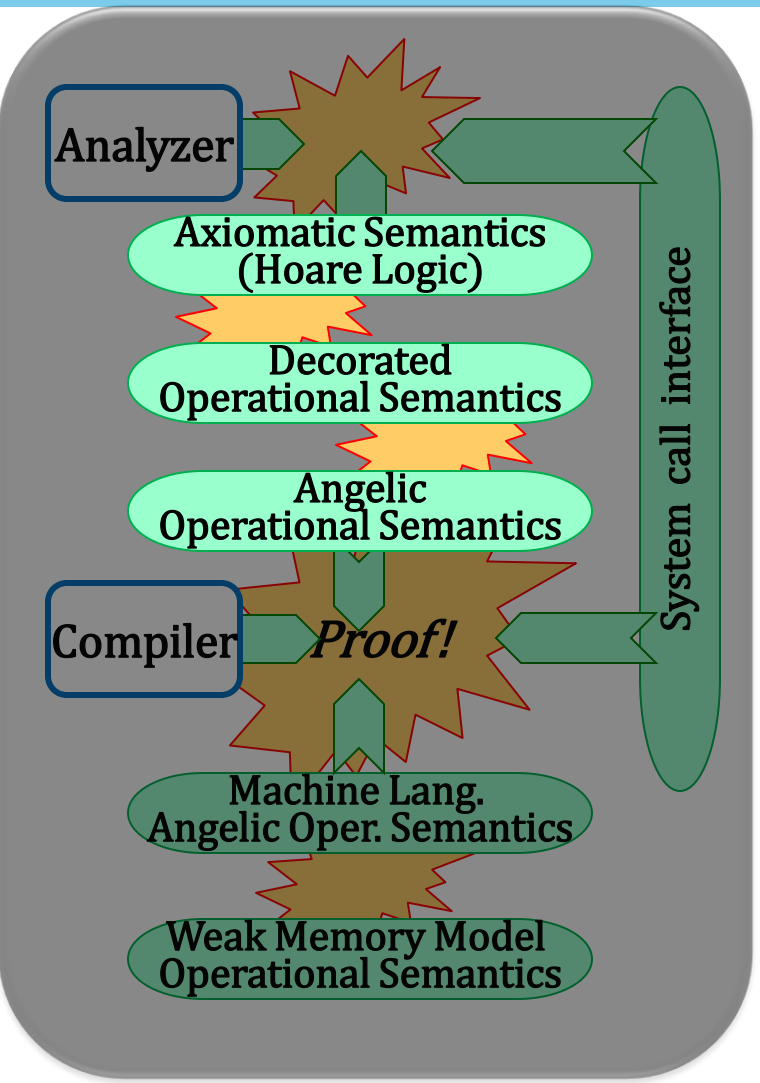
Write Gallina program to recheck invariants against the program

Prove Gallina program correct w.r.t. inference rules of the program logic

Extract Caml program from Coq

Example: W. Manksy undergraduate thesis, Princeton 2008, for analyzer of Gotsman et al., PLDI 2007

Soundness of program logic w.r.t. Op.Sem.



Program-variable values: $v \Downarrow 7$

Memory values: $p \mapsto 5$

Separation: $P * Q$

And, or $P \wedge Q$ $P \vee Q$

Recursion $\boldsymbol{\mu}\, x.\, P$

Impredicative quantification $\forall x.P$ $\exists x.P$

Function pointers $f : P \to Q$

Lock resource invariants $\ell \rightsquigarrow R$

Semantic soundness

Semantic methods of the 20th century were not quite capable of handling all these features simultaneously

Step indexing (Appel & McAllester, TOPLAS 2001)
Step indexing + indirection(Ahmed,Appel,Virga, LICS 2002)
Step indexing + impredicativity (Ahmed PhD thesis 2004)
Very Modal Model (Appel,Melliès,Richards,Vouillon POPL 2007)
Indirection Theory (Hobor,Dockins,Appel, POPL 2010)

$v \Downarrow 7$

$p \mapsto 5$

$P * Q$

$P \wedge Q$

$\mu x.\ P$

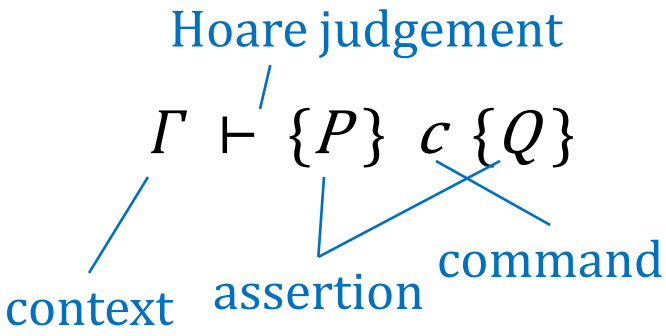
Impredicativity → $\exists x.P$

Indirection → $f : P \to Q$

Indirection → $\ell \rightsquigarrow R$

But also perhaps:
Domains and ultrametric spaces (Birkedal *et al.,* 2009,2010)

Soundness theorem



Axiomatic Semantics (program logic)

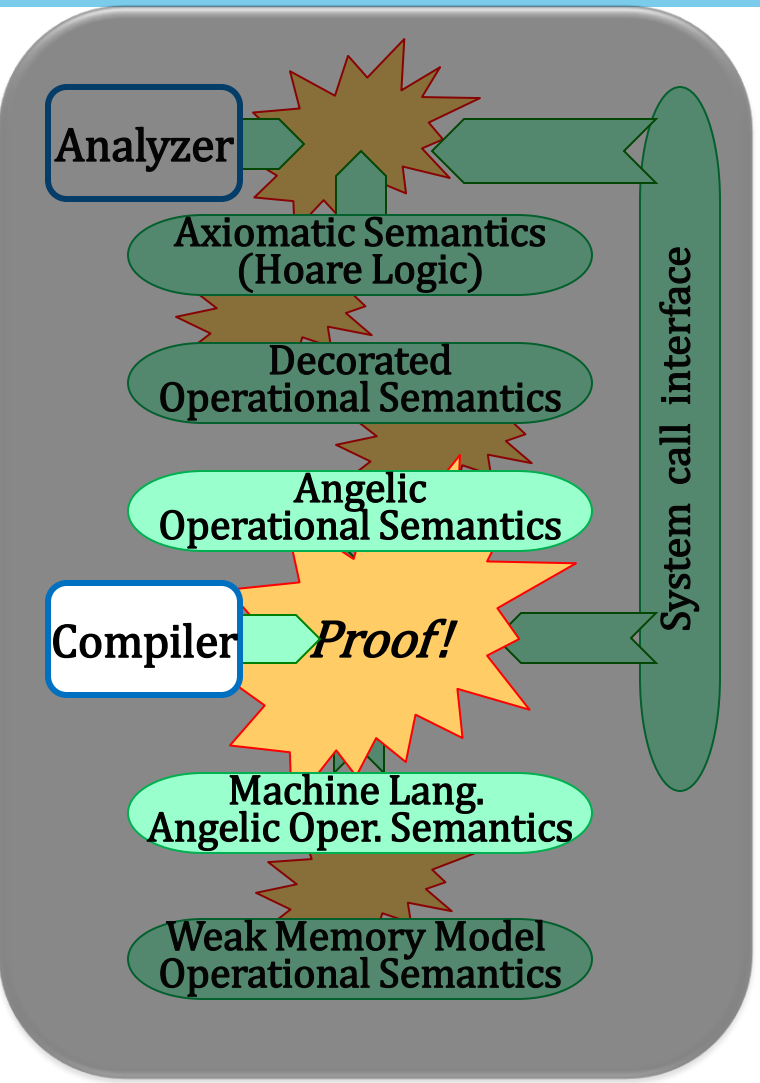
When you run a program, it obeys its Hoare triple

$\Gamma \vdash \sigma \mapsto \sigma'$

Operational Semantics

$P(\Gamma,\sigma) \rightarrow Q(\Gamma,\sigma')$ Soundness (vastly oversimplified)

Verified optimizing compiler



Previous machine-verified compilers

Journal of Automated Reasoning 5:461-492, 1989.

A Mechanically Verified Language Implementation

J STROTHER MOORE

Computational Logic, Inc., 1717 West Sixth Street, Suite 290, Austin, TX 78703, U.S.A.

(using ACL2 theorem prover)

Electronic Notes in Theoretical Computer Science 82 No. 2 (2003)

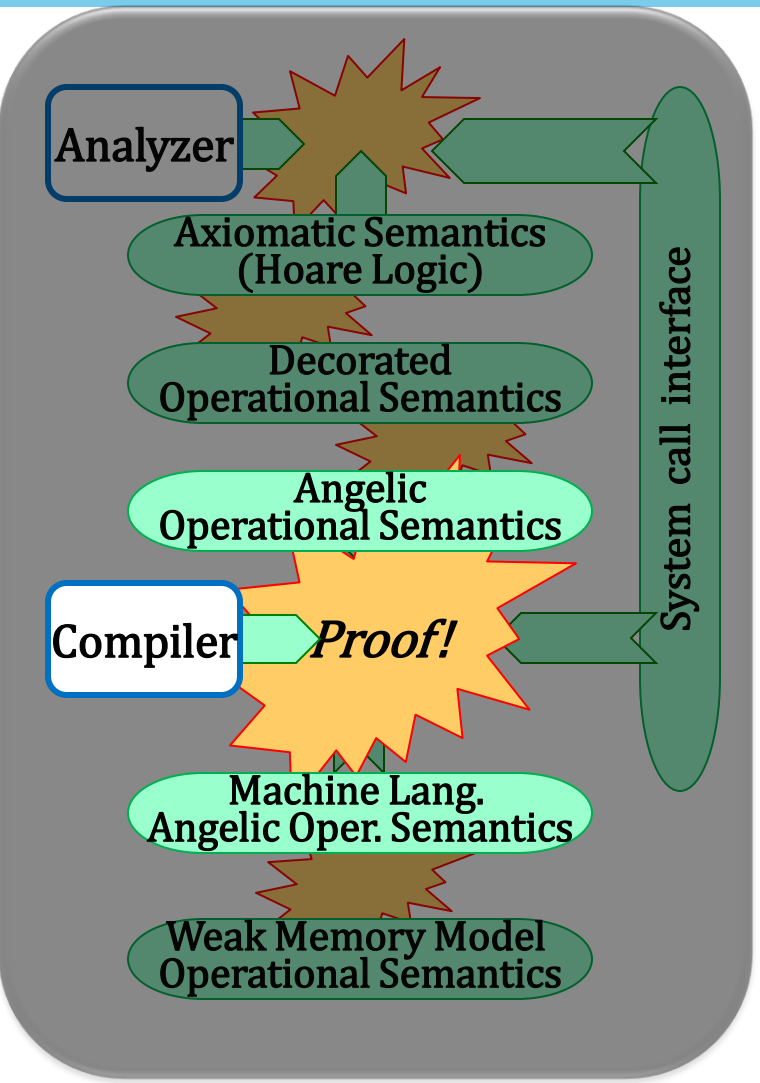
Extracting a formally verified, fully executable compiler from a proof assistant

Stefan Berghofer, Martin Strecker

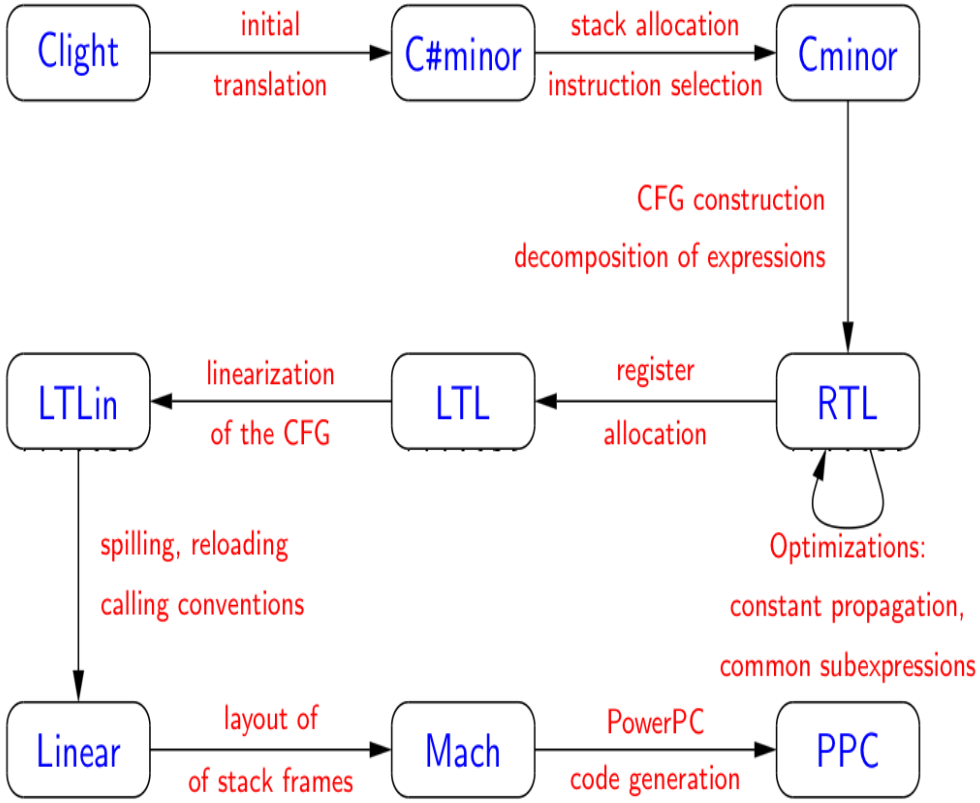
Technische Universität München, Fakultät für Informatik, D-85748 Garching

(using Isabelle/HOL theorem prover)

CompCert



Verified optimizing C compiler
Xavier Leroy, et al. 2006-11
INRIA Paris-Rocquencourt



Operational Semantics (C minor)

$$\Gamma, \psi \vdash \Omega, m, \rho, \kappa \mapsto \Omega', m', \rho', \kappa'$$

Operational Semantics (C minor)

```
Inductive step' (ge: genv) (m: mem) (rho: locals) (ctl: control):
              forall (s: stmt) (m': mem) (rho': locals) (ctl': control), Prop :=
| step_skip:
      step' ge m rho ctl Sskip m rho ctl
| step_assign: forall id e v rho'
      (Hexp: (w_eval_expr e v) (cfilter ge m rho))
      (Hrho': rho' = IdMap.set id (pfullshare,v) rho),
      step' ge m rho ctl (Sassign id e) m rho' ctl
| step_store: forall ch (a e: expr) v v2 m1
      (Ha: (w_eval_expr a v) (cfilter ge m rho))
      (He: (w_eval_expr e v2) (cfilter ge m rho))
      (Hstore: storev ch m v v2 = Some m1),
       step' ge m rho ctl (Sstore ch a e) m1 rho ctl
| step_ifthenelse: forall a (s1 s2: stmt) v1 vb (s': stmt)
      (Ha: (w_eval_expr a v1) (cfilter ge m rh
      (Hb: bool_of_valf v1 = Some vb)
      (Hs': s' = match vb with true => s1 | fal
      step' ge m rho ctl (Sifthenelse a s1 s2) m rho (Kseq s' ctl)
| step_seq: forall s1 s2,
      step' ge m rho ctl (Sseq s1 s2)  m rho (Kseq s1 (Kseq s2 ctl))
| step_loop:  forall s1,
      step' ge m rho ctl (Sloop s1) m rho (Kseq s1 (Kseq (Sloop s1) ctl))
| step_call: forall lid sig a bl P vf vargs f
     (Ha: (w_eval_expr a vf) (cfilter ge m rho))
     (Hbl: (w_eval_exprlist bl vargs) (cfilter ge m rho))
     (Hfind: Globalenvs.Genv.find_funct ge vf = Some (Internal f))
     (Hsig: f.(fn_sig).(sig_args) = sig)
     m' m'' sp' rho'
     (Halloc: alloc m 0 f.(fn_stackspace) = (m',sp'))
     (Hage: age1_mem m' = Some m'')
     (Hrho': rho' = (set_locals f.(fn_vars)
                    (set_params ((Vptr sp' Int.zero)::vargs)
                                (stack_pointer :: f.(fn_params))))),
     step' ge m rho ctl (Scall lid sig a bl P) m'' rho' (Kseq (fn_body f)
                (Kcall lid f rho ctl))
```

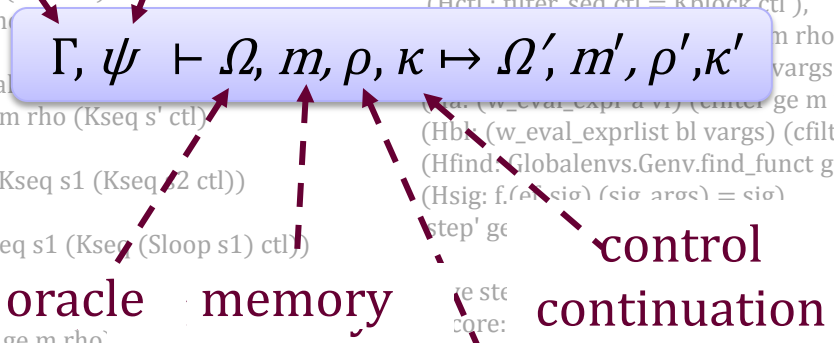
$$\Gamma, \psi \vdash \Omega, m, \rho, \kappa \mapsto \Omega', m', \rho', \kappa'$$

```
| step_return: forall sig al lv lid f sp e' rho' m' ctl' sh
      (Hal: (w_eval_exprlist (opt2list al) (opt2list lv)) (cfilter ge m rho))
      (Hfilter: filter_block_seq ctl = Kcall lid f e' ctl')
      (Hsig: f.(fn_sig).(sig_res) = sig)
      (Hrho: env_return e' lid lv = rho')
      (Hsp: rho ! stack_pointer = Some (sh, Vptr sp Int.zero))
      (Hfree: free m (sp,0) (f.(fn_stackspace)) = Some m'),
      step' ge m rho ctl (Sreturn sig al) m' rho' ctl'
| step_block: forall s',
      step' ge m rho ctl (Sblock s') m rho (Kseq s' (Kblock ctl))
| step_exit_0: forall ctl'
       (Hctl': filter_seq ctl = Kblock ctl'),
      step' ge m rho ctl (Sexit 0) m rho ctl'
| step_exit_S: forall n ctl'
       (Hctl': filter_seq ctl = Kblock ctl'),
                                          m rho (Kseq (Sexit n) ctl')
                                          vargs f
                                          ge m rho))
      (Hbl: (w_eval_exprlist bl vargs) (cfilter ge m rho))
      (Hfind: Globalenvs.Genv.find_funct ge vf = Some (External f))
      (Hsig: f.(ef_sig).(sig_args) = sig),
       step' ge m rho ctl (Scall lid sig a bl P) m rho (Kext lid f vargs P ctl).

Inductive step (ge: genv): forall (st st' : state), Prop :=
 | step_core: forall ora m q m' q'
        (Hcs: corestep ge m q  m' q'),
        step ge (State ora m q) (State ora m' q')
 | step_external_Some: forall f vargs P rho' ora m q ora' m' q'
        (Hat: at_external q = Some (f,vargs,P))
        (Hcons: consult f ora vargs m P (Some (ora', m', rho')))
        (Hafter: after_external rho' q = Some q'),
        step ge (State ora m q) (State ora' m' q')
 | step_external_None:  forall ora m q f vargs P
        (Hat: at_external q = Some (f,vargs,P))
        (Hcons: consult f ora vargs m P None),
        step ge (State ora m q) (State ora m q).
```

Operational Semantics (C minor)

```
Inductive step' (ge: genv) (m: mem) (rho: locals) (ctl: control):
              forall (s: stmt) (m': mem) (rho': locals) (ctl': control), Prop :=
| step_skip:
      step' ge m rho ctl Sskip m rho ctl
| step_assign: forall id e v rho'
      (Hexp: (w_eval_expr e v) (cfilter ge m rho))
      (Hrho': rho' = IdMap.set id (pfullshare,v) rho)
      step' ge m rho c                    ho' ctl
| step_store: forall ch
      (Ha: (w_eval_ex                     rho))
      (He: (w_eval_ex                     rho))
      (Hstore: storev ch m v v2 = Some m1)
       step' ge m rho ctl (Sstore ch a e) m1 rho ctl
| step_ifthenelse: forall a (s1 s2: stmt) v1 vb (s': stmt)
      (Ha: (w_eval_expr a v1) (cfilter ge m rh
      (Hb: bool_of_valf v1 = Some vb)
      (Hs': s' = match vb with true => s1 | fal
      step' ge m rho ctl (Sifthenelse a s1 s2) m rho (Kseq s' ctl)
| step_seq: forall s1 s2,
      step' ge m rho ctl (Sseq s1 s2)  m rho (Kseq s1 (Kseq s2 ctl))
| step_loop:  forall s1,
      step' ge m rho ctl (Sloop s1) m rho (Kseq s1 (Kseq (Sloop s1) ctl))
| step_call: forall lid sig a bl P vf vargs f
      (Ha: (w_eval_expr a vf) (cfilter ge m rh
      (Hbl: (w_eval_exprlist bl vargs) (cfilter ge m rho)
      (Hfind: Globalenvs.Genv.find_funct ge vf = Some (Internal f))
      (Hsig: f.(fn_sig).(sig_args) = sig)
      m' m'' sp' rho'
      (Halloc: alloc m 0 f.(fn_stackspace) = (m',sp'))
      (Hage: age1_mem m' = Some m'')
      (Hrho': rho' = (set_locals f.(fn_vars)
                     (set_params ((Vptr sp' Int.zero)::vargs)
                                 (stack_pointer :: f.(fn_params))))),
      step' ge m rho ctl (Scall lid sig a bl P) m'' rho' (Kseq (fn_body f)
                  (Kcall lid f rho ctl))
```

```
| step_return: forall sig al lv lid f sp e' rho' m' ctl' sh
      (Hal: (w_eval_exprlist (opt2list al) (opt2list lv)) (cfilter ge m rho))
      (Hfilter: filter_block_seq ctl = Kcall lid f e' ctl')
      (Hsig: f.(fn_sig).(sig_res) = sig)
      (Hrho: env_return e' lid lv = rho')
      (Hsp: rho ! stack_pointer = Some (sh, Vptr sp Int.zero))
      (Hfree: free m (sp,0) (f.(fn_stackspace)) = Some m'),
      step' ge m rho ctl (Sreturn sig al) m' rho' ctl'
| step_block: forall s',
      step' ge m rho ctl (Sblock s') m rho (Kseq s' (Kblock ctl))
| step_exit_0: forall ctl'
       (Hctl': filter_seq ctl = Kblock ctl'),
      step' ge m rho ctl (Sexit 0) m rho ctl'
| step_exit_S: forall n ctl'
       (Hctl': filter_seq ctl = Kblock ctl'),
                                          m rho (Kseq (Sexit n) ctl')
                                          vargs f
                                          ge m rho))
      (Hbl: (w_eval_exprlist bl vargs) (cfilter ge m rho))
      (Hfind: Globalenvs.Genv.find_funct ge vf = Some (External f))
      (Hsig: f.(ef_sig) (sig_args) = sig)
      step' ge                                    n rho (Kext lid f vargs P ctl).

                                                  e), Prop :=
                                          ora m' q')
| ste                                     P rho' ora m q ora' m' q'
                                          ,vargs,P))
                                          P (Some (ora', m', rho')))
      (Hafter: after_external rho' q = Some q'),
      step ge (State ora m q) (State ora' m' q')
| step_external_None:  forall ora m q f vargs P
      (Hat: at_external q = Some (f,vargs,P))
      (Hcons: consult f ora vargs m P None),
      step ge (State ora m q) (State ora m q).
```



Operational Semantics (expressions)

```
Definition eval_unop (op: unary_operation) (arg: val) : option val :=
 match op, arg with
 | Ocast8unsigned, _ => Some (Val.cast8unsigned arg)
 | Ocast8signed, _ => Some (Val.cast8signed arg)
 | Ocast16unsigned, _ => Some (Val.cast16unsigned arg)
 | Ocast16signed, _ => Some (Val.cast16signed arg)
 | Onegint, Vint n1 => Some (Vint (Int.neg n1))
 | Onotbool, Vint n1 => Some (Val.of_bool (Int.eq n1 Int.zero))
 | Onotbool, Vptr b1 n1 => Some Vfalse
 | Onotint, Vint n1 => Some (Vint (Int.not n1))
 | Onegf, Vfloat f1 => Some (Vfloat (Float.neg f1))
 | Oabsf, Vfloat f1 => Some (Vfloat (Float.abs f1))
 | Osingleoffloat, _ => Some (Val.singleoffloat arg)
 | Ointoffloat, Vfloat f1 => Some (Vint (Float.intoffloat f1))
 | Ofloatofint, Vint n1 => Some (Vfloat (Float.floatofint n1))
 | Ofloatofintu, Vint n1 => Some (Vfloat (Float.floatofintu n1))
 | _, _ => None
 end.

Definition eval_compare_null (c: comparison) (n: int) : option val :=
 if Int.eq n Int.zero
 then match c with Ceq => Some Vfalse | Cne => Some Vtrue | _ => None end
 else None.

Definition eval_binop  (op: binary_operation) (arg1 arg2: val) (* (m: mem) *) : option val :=
 match op, arg1, arg2 with
 | Oadd, Vint n1, Vint n2 => Some (Vint (Int.add n1 n2))
 | Oadd, Vint n1, Vptr b2 n2 => Some (Vptr b2 (Int.add n2 n1))
 | Oadd, Vptr b1 n1, Vint n2 => Some (Vptr b1 (Int.add n1 n2))
 | Osub, Vint n1, Vint n2 => Some (Vint (Int.sub n1 n2))
 | Osub, Vptr b1 n1, Vint n2 => Some (Vptr b1 (Int.sub n1 n2))
 | Osub, Vptr b1 n1, Vptr b2 n2 =>   if eq_block b1 b2 then Some (Vint (Int.sub n1 n2)) else None
 | Omul, Vint n1, Vint n2 => Some (Vint (Int.mul n1 n2))
 | Odiv, Vint n1, Vint n2 =>   if Int.eq n2 Int.zero then None else Some (Vint (Int.divs n1 n2))
 | Odivu, Vint n1, Vint n2 =>   if Int.eq n2 Int.zero then None else Some (Vint (Int.divu n1 n2))
 | Omod, Vint n1, Vint n2 =>   if Int.eq n2 Int.zero then None else Some (Vint (Int.mods n1 n2))
 | Omodu, Vint n1, Vint n2 =>   if Int.eq n2 Int.zero then None else Some (Vint (Int.modu n1 n2))
 | Oand, Vint n1, Vint n2 => Some (Vint (Int.and n1 n2))
 | Oor, Vint n1, Vint n2 => Some (Vint (Int.or n1 n2))
 | Oxor, Vint n1, Vint n2 => Some (Vint (Int.xor n1 n2))
 | Oshl, Vint n1, Vint n2 =>   if Int.ltu n2 (Int.repr 32) then Some (Vint (Int.shl n1 n2)) else None
 | Oshr, Vint n1, Vint n2 =>   if Int.ltu n2 (Int.repr 32) then Some (Vint (Int.shr n1 n2)) else None
 | Oshru, Vint n1, Vint n2 =>   if Int.ltu n2 (Int.repr 32) then Some (Vint (Int.shru n1 n2)) else None
  | Oaddf, Vfloat f1, Vfloat f2 => Some (Vfloat (Float.add f1 f2))
 | Osubf, Vfloat f1, Vfloat f2 => Some (Vfloat (Float.sub f1 f2))
 | Omulf, Vfloat f1, Vfloat f2 => Some (Vfloat (Float.mul f1 f2))
 | Odivf, Vfloat f1, Vfloat f2 => Some (Vfloat (Float.div f1 f2))
 | Ocmp c, Vint n1, Vint n2 =>   Some (Val.of_bool(Int.cmp c n1 n2))
 | Ocmp c, Vptr b1 n1, Vptr b2 n2 =>   if eq_block b1 b2 then Some(Val.of_bool(Int.cmp c n1 n2)) else None
 | Ocmp c, Vptr b1 n1, Vint n2 => eval_compare_null c n2
 | Ocmp c, Vint n1, Vptr b2 n2 => eval_compare_null c n1
 | Ocmpu c, Vint n1, Vint n2 =>   Some (Val.of_bool(Int.cmpu c n1 n2))
 | Ocmpf c, Vfloat f1, Vfloat f2 =>   Some (Val.of_bool (Float.cmp c f1 f2))
 | _, _, _ => None
 end.
```

```
Inductive w_eval_expr:  expr -> val -> pred world :=
 | w_eval_Evar: forall id sh v w,
    env_get (w_rho w) id = Some (sh,v) ->
    w_eval_expr (Evar id) v w
 | w_eval_Eval: forall v w,
    w_eval_expr (Eval v) v w
 | w_eval_Eaddrsymbol: forall id ofs sh b i v w,
     env_get (w_ge w) id = Some (sh,Vptr b i) ->
     v =  Vptr b (Int.add i ofs) ->
    w_eval_expr (Eaddrsymbol id ofs) v w
 | w_eval_Eunop: forall op a1 v1 v w,
    w_eval_expr a1 v1 w ->
    eval_unop op v1 = Some v ->
    w_eval_expr (Eunop op a1) v w
 | w_eval_Ebinop: forall op a1 a2 v1 v2 v w,
    w_eval_expr a1 v1 w ->
    w_eval_expr a2 v2 w ->
   eval_binop op v1 v2 = Some v ->
    w_eval_expr (Ebinop op a1 a2) v w
 | w_eval_Eload: forall chunk addr b ofs v w,
    w_eval_expr addr (Vptr b ofs) w->
    CoreMem.core_load chunk (b, Int.signed ofs) v (w_m w) ->
    w_eval_expr (Eload chunk addr) v w
 | w_eval_Econdition: forall a1 a2 a3 v1 b1 v2 w,
    w_eval_expr a1 v1 w ->
    Val.bool_of_val v1 b1 ->
    w_eval_expr (if b1 then a2 else a3) v2 w ->
    w_eval_expr (Econdition a1 a2 a3) v2 w.

Inductive w_eval_exprlist: list expr -> list val -> pred world :=
 | w_eval_Enil: forall w,
    w_eval_exprlist nil nil w
 | w_eval_Econs: forall a1 al v1 vl w,
    w_eval_expr a1 v1 w -> w_eval_exprlist al vl w ->
    w_eval_exprlist (a1 :: al) (v1 :: vl) w.
```

Operational Semantics (memory) excerpt

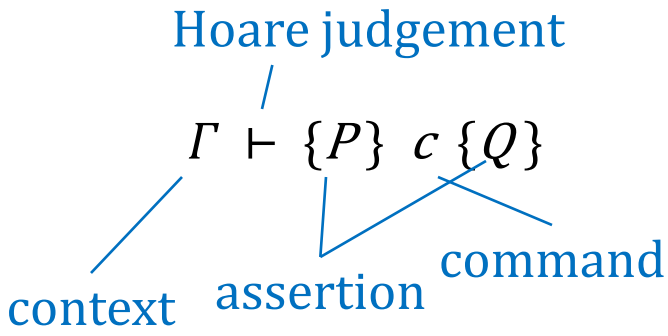
Blazy + Leroy, plus some recent changes

```
Require Import plain_compcert.
Require Import Address.

Module Type MEM.

Parameter mem : Type.

Parameter nextblock: mem -> block.
Axiom nextblock_pos: forall (m: mem), nextblock m > 0.

(** The initial store. *)

Parameter empty:  forall (approx_level: nat),  mem.

Definition nullptr: block := 0.

Definition mem_access : Type := mem -> address -> Prop.

Parameter mem_readable: mem_access.
Parameter mem_writable: mem_access.
Parameter mem_empty:  mem_access.


Axiom mem_writable_readable:
  forall m loc, mem_writable m loc -> mem_readable m loc.

Axiom mem_readable_not_empty:
  forall m loc, mem_readable m loc -> ~mem_empty m loc.

Definition access_block (acc: mem_access) (m: mem) (loc: address) (n: Z) : Prop :=
  forall i, 0 <= i < n -> acc m (fst loc, snd loc + i).

Axiom access_imp: forall (acc1 acc2 : mem_access),
    (forall m loc, acc1 m loc -> acc2 m loc) ->
    forall m loc n,
     access_block acc1 m loc n -> access_block acc2 m loc n.

Axiom access_imp_not: forall (acc1 acc2 : mem_access),
    (forall m loc, acc1 m loc -> ~acc2 m loc) ->
    forall m loc n,
     access_block acc1 m loc n -> access_block (fun m a => ~acc2 m a) m loc n.

Hint Resolve mem_writable_readable mem_readable_not_empty : mem.

Hint Resolve (access_imp _ _ mem_writable_readable) : mem.
Hint Resolve (access_imp_not _ _ mem_readable_not_empty) : mem.

Axiom mem_readable_dec:  forall m loc, decide (mem_readable m loc).
Axiom mem_writable_dec:  forall m loc, decide (mem_writable m loc).

Axiom access_block_dec: forall acc,
    (forall m loc, decide (acc m loc)) ->
    forall m loc n, decide (access_block acc m loc n).
Implicit Arguments access_block_dec [acc].

(** Properties of empty *)
Axiom mem_empty_empty:  forall lev loc, mem_empty (empty lev) loc.

Axiom empty_nextblock:  forall lev, nextblock (empty lev) = (1:block).

(** Allocation of a fresh block with the given bounds.  Return an updated
  memory state and the address of the fresh block, which initially contains
  undefined cells.  Note that allocation never fails: we model an
  infinite memory. *)

Parameter alloc:  forall (m: mem) (lo hi : Z), (mem*block).

(** Freeing a block.  Return the updated memory state where the given
  block address has been invalidated: future reads and writes to this
  address will fail.  Note that we make no attempt to return the block
  to an allocation pool: the given block address will never be allocated
  later.

  Unlike in original CompCert, free is given not just a block, but a start
  address and a size.  All the memory in that range must be fully owned
  (writable).  If not, then the free will fail.  Thus, free now returns
  an (option mem), not a plain mem.  *)

Parameter free': forall (m: mem) (loc: address) (n: Z)
    (WR: access_block mem_writable m loc n),  mem .

Definition free (m: mem) (loc: address) (n: Z) : option mem :=
  match access_block_dec mem_writable_dec m loc n with
  | left WR => Some (free' m loc n WR)
  | right _ => None
  end.

(** Freeing of a list of blocks. *)

Definition free_list (m:mem) (l:list (address*Z)) :=
  List.fold_right (fun bn mx => match bn , mx with
                                | (base, n) , Some m => free m base n
                                | _ , None => None
                                end)
      (Some m) l.

(** A block address is valid if it CANNOT be the result of any future "alloc".
  A block remains valid even after being freed. *)

Definition valid_block (m: mem)  (b: block) :=
  b < nextblock m.

(** In CompCert [valid_access m chunk b ofs] meant that a memory access (load or store)
    of the given chunk is possible in [m] at address [b, ofs].
  This was used to test validity of address arithmetic.  In Concurrent Separation
  Logic, this method will not work.

  In this version, one can obtain something similar:

  Definition valid_access m chunk b ofs :=
       forall i, 0 <= i < size_chunk ch -> ~mem_empty m (b, ofs+i).
  Definition valid_pointer m b ofs :=
      ~mem_empty m (b,ofs).

  However, there would still be problems using this to test validity of
  address arithmetic.  Therefore, although this Module Type contains
  several axioms containing the words "valid_access" and "valid_pointer",
  these axioms are probably not useful in a concurrent shared-memory context. *)

(** [load chunk m (b,ofs)] perform a read in memory state [m], at address
  [b] and offset [ofs].  [None] is returned if the address is invalid
  or the memory access is out of bounds.
    For people who like programming with dependent types, a version
  load' is provided, where you can provide a proof that the address is loadable,
  and you get a total function.  *)

Parameter load': forall(chunk: memory_chunk) (m: mem) (loc: address)
        (RD: access_block mem_readable m loc (size_chunk chunk)),
        val.

Definition load (chunk: memory_chunk) (m: mem) (loc: address) : option val :=
  match access_block_dec mem_readable_dec m loc (size_chunk chunk)
  with left RD => Some (load' chunk m loc RD)
     | right _ => None
  end.

(** [loadv chunk m addr] is similar, but the address and offset are given
  as a single value [addr], which must be a pointer value. *)

Definition loadv (chunk: memory_chunk) (m: mem) (addr: val) :=
  match addr with
  | Vptr b ofs => load chunk m (b, Int.signed ofs)
  | _ => None
  end.

(** [store chunk m (b,ofs) v] performs a write in memory state [m].
  Value [v] is stored at address [b] and offset [ofs].
  Return the updated memory store, or [None] if the address is invalid
  or the memory access is out of bounds.
    For people who like programming with dependent types, a version
  store' is provided, where you can provide a proof that the address is storable,
  and you get a total function.  *)

Parameter store': forall (ch: memory_chunk) (m: mem) (loc: address) (v: val)
    (WR: access_block mem_writable m loc (size_chunk ch)),  mem .

Definition store (chunk: memory_chunk) (m: mem) (loc: address) (v: val) :
            option mem :=
  match access_block_dec mem_writable_dec m loc (size_chunk chunk) with
    | left WR =>  Some (store' chunk m loc v WR)
    | right _ => None
  end.

Axiom mem_writable_store: forall ch loc v m,
      access_block mem_writable m loc (size_chunk ch) ->
      exists m', store ch m loc v = Some m'.

Hint Resolve mem_writable_store : mem.

(** [storev chunk m addr v] is similar, but the address and offset are given
  as a single value [addr], which must be a pointer value. *)

Definition storev (chunk: memory_chunk) (m: mem) (addr v: val) : option mem :=
  match addr with
  | Vptr b ofs => store chunk m (b, Int.signed ofs) v
  | _ => None
  end.
```

```
(** Build a block filled with the given initialization data. *)
Definition size_init_data (id: init_data) : Z :=
  match id with
  | Init_int8 _ => 1
  | Init_int16 _ => 2
  | Init_int32 _ => 4
  | Init_float32 _ => 4
  | Init_float64 _ => 8
  | Init_space n => Zmax n 0
  | Init_pointer _ => 4
  end.

Definition size_init_data_list (id: list init_data): Z :=
  List.fold_right (fun id sz => size_init_data id + sz) 0 id.

Axiom size_init_data_list_pos:
  forall id, size_init_data_list id >= 0.

Parameter  alloc_init_data: forall (m: mem)  (id: list init_data),  (mem * block).

(** Definitions and axiom to characterize the result of alloc_init_data *)

Inductive init_val_type :=
  | InitValData: memory_chunk -> val -> init_val_type
  | InitValSpace: Z -> init_val_type.

Definition init_val (id: init_data) : init_val_type :=
  match id with
  | Init_int8 n => InitValData Mint8unsigned (Vint n)
  | Init_int16 n => InitValData Mint16unsigned (Vint n)
  | Init_int32 n => InitValData Mint32 (Vint n)
  | Init_float32 f => InitValData Mfloat32 (Vfloat f)
  | Init_float64 f => InitValData Mfloat64 (Vfloat f)
  | Init_space n => InitValSpace (Zmax n 0)
  | Init_pointer _ => InitValSpace 4
  end.

Definition init_data1 (m: mem) (loc: address) (a: init_data) : option mem :=
  match init_val a with
  | InitValData ch v => store ch m loc v
  | InitValSpace n =>  Some m
  end.

Fixpoint init_data_block (b: block)(pos: Z) (id: list init_data) (m: mem)  {struct id} :
                          option mem  :=
  match id  with
  | nil => Some m
  | a :: id' => match init_data1 m (b,pos) a with
                | Some m' => init_data_block b (pos+size_init_data a) id' m'
                | None => None
                end
  end.

Axiom alloc_init_data_e:
  forall id m m' b,
    alloc_init_data  m id = (m',b)  ->
    b = nextblock  m /\
      (let (m1,b) := alloc m 0 (size_init_data_list id)
       in init_data_block b 0 id m1) = Some m'.

Axiom alloc_init_data_e2:
  forall id id1 a id2 m m' ofs b,
    alloc_init_data  m id = (m',b) ->
    id =  (id1++a::id2) ->
    ofs = size_init_data_list id - size_init_data_list (a::id2) ->
    match init_val a with
    | InitValData ch v => load ch m' (nextblock  m, ofs) = Some (Val.load_result ch v)
    | InitValSpace n => access_block mem_writable m' (b, ofs) n
    end.

(* The memory state [m] after a store of value [v] at offset [ofs]
   in block [b]. *)

(** * Properties of the memory operations *)

(** ** Properties related to block validity *)

Axiom valid_not_valid_diff:
  forall m b b', valid_block m b -> ~(valid_block m b') -> b <> b'.

Hint Resolve valid_not_valid_diff : mem.

(** ** Properties related to [load] *)

Axiom mem_readable_load: forall ch m loc,
          access_block mem_readable m loc (size_chunk ch) ->
          exists v, load ch m loc = Some v.
```

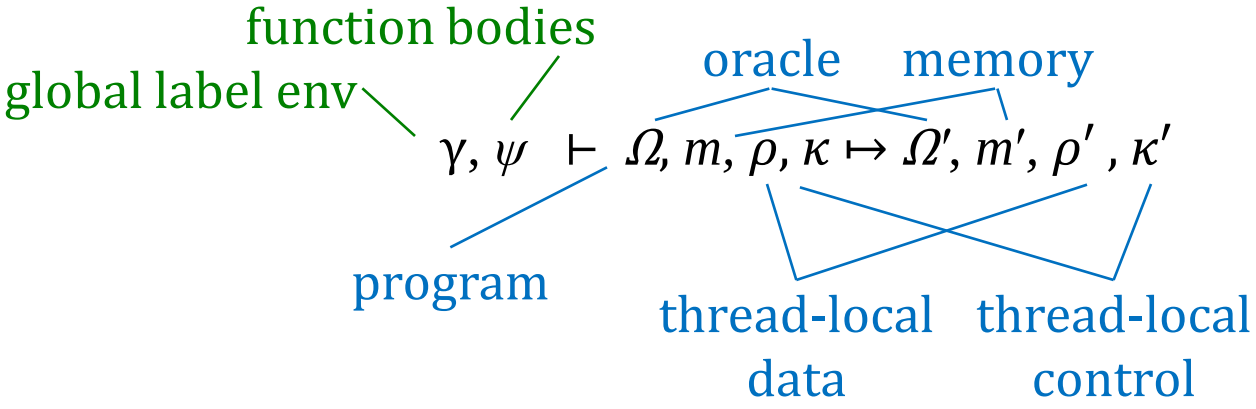
```
Axiom load_mem_readable: forall ch m loc v,
            load ch m loc = Some v ->
            access_block mem_readable m loc (size_chunk ch).

Hint Resolve mem_readable_load load_mem_readable : mem.

Axiom load_result_characterization: forall ch m loc v,
          load ch m loc = Some v ->
          exists v', v = Val.load_result ch v'.

(** ** Properties related to [store] *)

Axiom nextblock_store:
  forall chunk m1 loc v m2
    (STORE: store chunk m1 loc v = Some m2),
  nextblock  m2 = nextblock  m1.

Axiom store_valid_block_1:
  forall chunk m1 loc v m2
    (STORE: store chunk m1 loc v = Some m2),
  forall b', valid_block m1 b' -> valid_block  m2 b'.

Axiom store_valid_block_2:
  forall chunk m1 loc v m2
    (STORE: store chunk m1 loc v = Some m2),
  forall b', valid_block  m2 b' -> valid_block  m1 b'.

Hint Resolve store_valid_block_1 store_valid_block_2: mem.

(** The next 9 axioms replace the old store_valid_access123 lemmas *)

Axiom store_empty1:
  forall chunk m1 loc v m2
    (STORE: store chunk m1 loc v = Some m2),
  forall loc',
   mem_empty  m1 loc' -> mem_empty  m2 loc'.

Axiom store_empty1':
  forall chunk m1 loc v m2
    (STORE: store chunk m1 loc v = Some m2),
  forall loc' n,
   access_block  mem_empty  m1 loc' n ->
   access_block  mem_empty  m2 loc' n.
Axiom store_empty2:
  forall chunk m1 loc v m2
    (STORE: store chunk m1 loc v = Some m2),
  forall loc',
   mem_empty  m2 loc' -> mem_empty  m1 loc'.

Axiom store_empty2':
  forall chunk m1 loc v m2
    (STORE: store chunk m1 loc v = Some m2),
  forall loc' n,
   access_block  mem_empty  m2 loc' n ->
   access_block  mem_empty  m1 loc' n.

Axiom store_writable1:
  forall chunk m1 loc v m2
    (STORE: store chunk m1 loc v = Some m2),
  forall loc',
   mem_writable  m1 loc' -> mem_writable  m2 loc'.

Axiom store_writable1':
  forall chunk m1 loc v m2
    (STORE: store chunk m1 loc v = Some m2),
  forall loc' n,
   access_block  mem_writable  m1 loc' n ->
   access_block  mem_writable  m2 loc' n.

Axiom store_writable2:
  forall chunk m1 loc v m2
    (STORE: store chunk m1 loc v = Some m2),
  forall loc',
   mem_writable  m2 loc' -> mem_writable  m1 loc'.

Axiom store_writable2':
  forall chunk m1 loc v m2
    (STORE: store chunk m1 loc v = Some m2),
  forall loc' n,
   access_block  mem_writable  m2 loc' n ->
   access_block  mem_writable  m1 loc' n.

Axiom store_readable1:
  forall chunk m1 loc v m2
    (STORE: store chunk m1 loc v = Some m2),
  forall loc',
   mem_readable  m1 loc' -> mem_readable  m2 loc'.
```

Soundness theorem



Axiomatic Semantics (program logic)

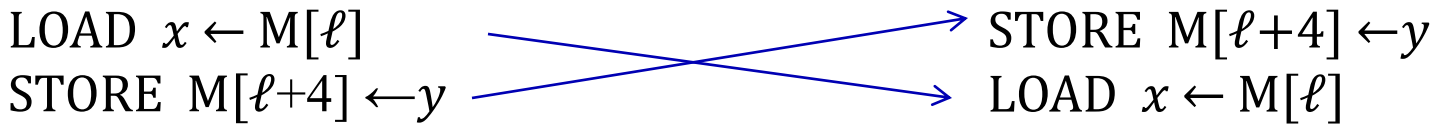
When you run a program, it obeys its Hoare triple



Operational Semantics

CONCURRENCY

Can the compiler hoist this store?



Pure dataflow considerations permit it—the addresses are different

But what if it is involved in a race condition?

Solution: Use semaphores [Dijkstra 1965] to avoid race conditions.

Use Concurrent Separation Logic [O'Hearn 2004, Hobor 2008] to reason about semaphores.

CompCert Leroy et al. 2006-11

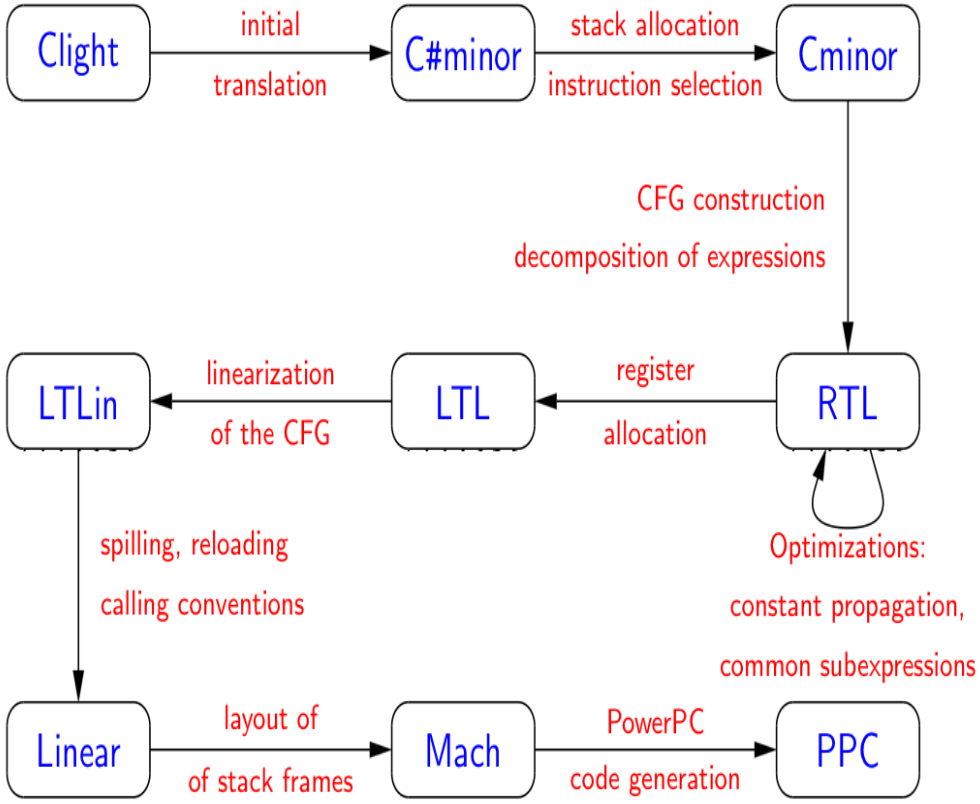
Problem: CompCert is proved correct with respect to

the wrong specification!

(Sequential, not concurrent; System-call model too weak; etc.; etc.;)

Solution: Adjust the specification, keep (almost) the same proof.

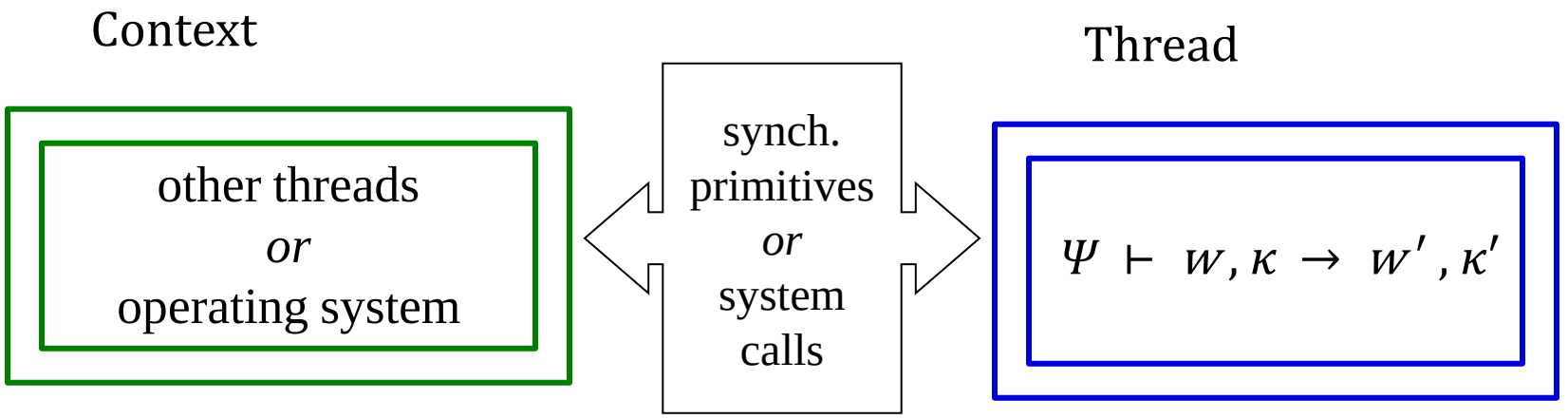
Verified optimizing C compiler



Design decision

- Many kinds of reasoning are facilitated by the absence of data races
- The operational semantics will enforce race-freedom **by keeping track of "who owns what",** and get stuck at any possibility of a data race
- We'll use the program logic to prove "who owns what" (and thus, race-freedom) before running the operational semantics

Interaction of a thread with its context



Permissions can be changed (only) by "external" calls

We attach "permissions" (none, read, write) to each memory location in each thread

What program logic?

Want to reason about (concurrent) shared-memory C programs

Optimizing compiler, which (as Boehm warns) may be unsound for concurrency if one is not careful

Solution:

Concurrent Separation Logic (O'Hearn 2004)

but with first-class locks and threads (Gotsman '07, Hobor '08)

Enforces: no race conditions in source program

Preserve this guarantee through compiler phases!

CSL rules for locks

Hobor, Appel, Zappa Nardelli 2008
Aquinas Hobor PhD thesis 2008

$\{\ell \hookrightarrow R\}$ acquire ℓ $\{R * \ell \hookrightarrow R\}$

$\{R * \ell \hookrightarrow R\}$ release ℓ $\{\ell \hookrightarrow R\}$

Operational semantics keeps track of ownership (permissions)

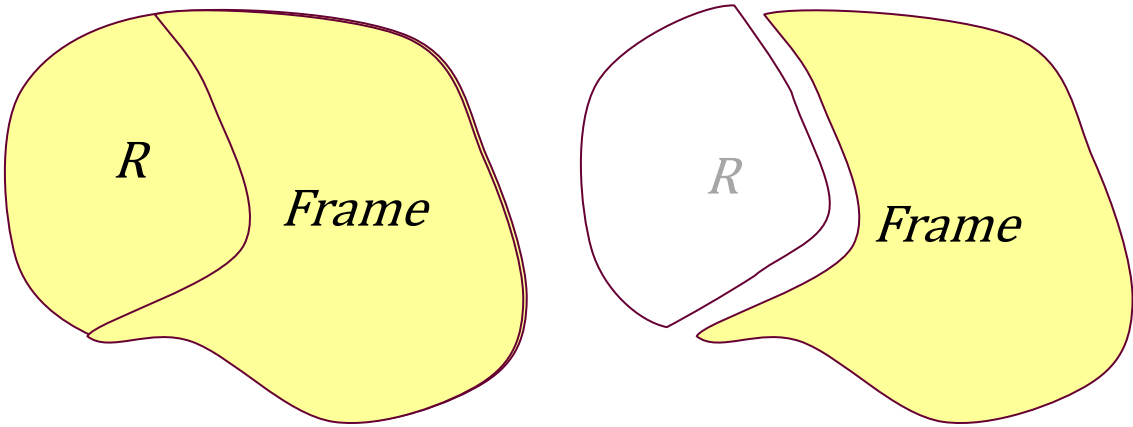
$\therefore$ Operational semantic state needs to contain binding $\ell \hookrightarrow R$

*Such bindings are a component of the **decorated** operational semantics*

Releasing a lock

Hobor, Appel, Zappa Nardelli 2008
Aquinas Hobor PhD thesis 2008

$$\frac{\{R^* \ell \rightsquigarrow R\}\ \text{release}\ \ell\ \{\ell \rightsquigarrow R\}}{\{R\ ^* \ell \rightsquigarrow R^* \text{Frame}\}\ \text{release}\ \ell\ \{\ell \rightsquigarrow R^* \text{Frame}\}}$$

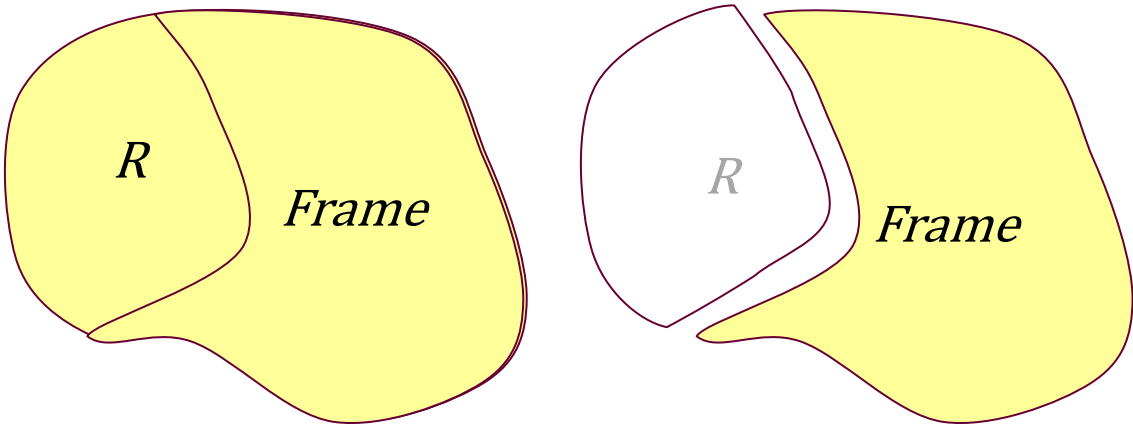


Hobor, Appel, Zappa Nardelli 2008
Aquinas Hobor PhD thesis 2008

Operational semantic step

How does the operational semantics know how much of the heap to give away when releasing lock ℓ ?

$$\frac{\{R * \ell \rightsquigarrow R\}\ \text{release}\ \ell\ \{\ell \rightsquigarrow R\}}{\{R * \ell \rightsquigarrow R * \text{Frame}\}\ \text{release}\ \ell\ \{\ell \rightsquigarrow R * \text{Frame}\}}$$



Answer: The oracle Ω contains a map from locations ℓ to resource-invariants R

$$\Gamma, \psi \vdash \Omega, m, \rho, \kappa \mapsto \Omega', m', \rho', \kappa'$$

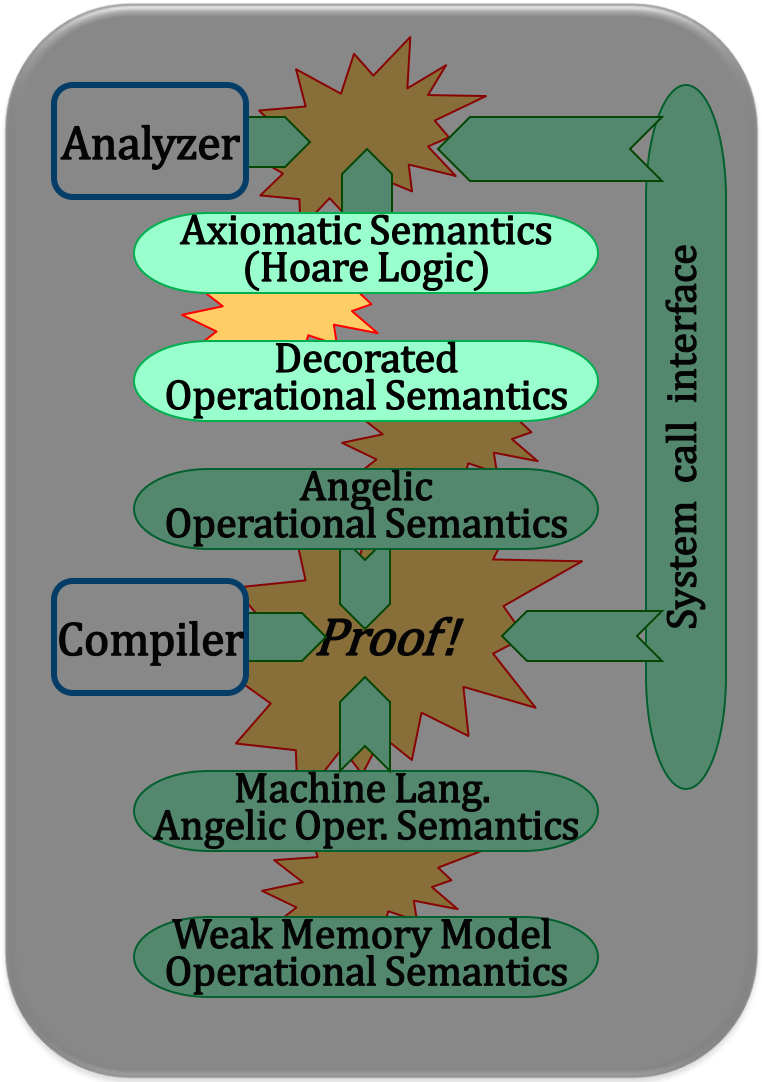
END –TO – END VERIFICATION

Indirection theory

Hobor, Dockins, Appel 2010

- To handle predicates inside states, we use ***Indirection Theory*** (step indexing)

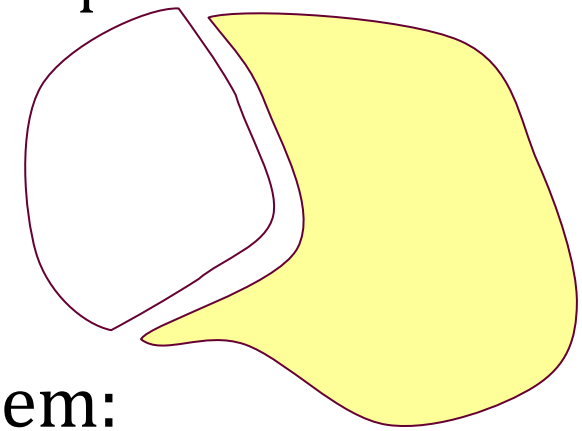
- This is not convenient for bisimulation proofs (especially if different # of steps in source/target pgm.)



Angelic semantics

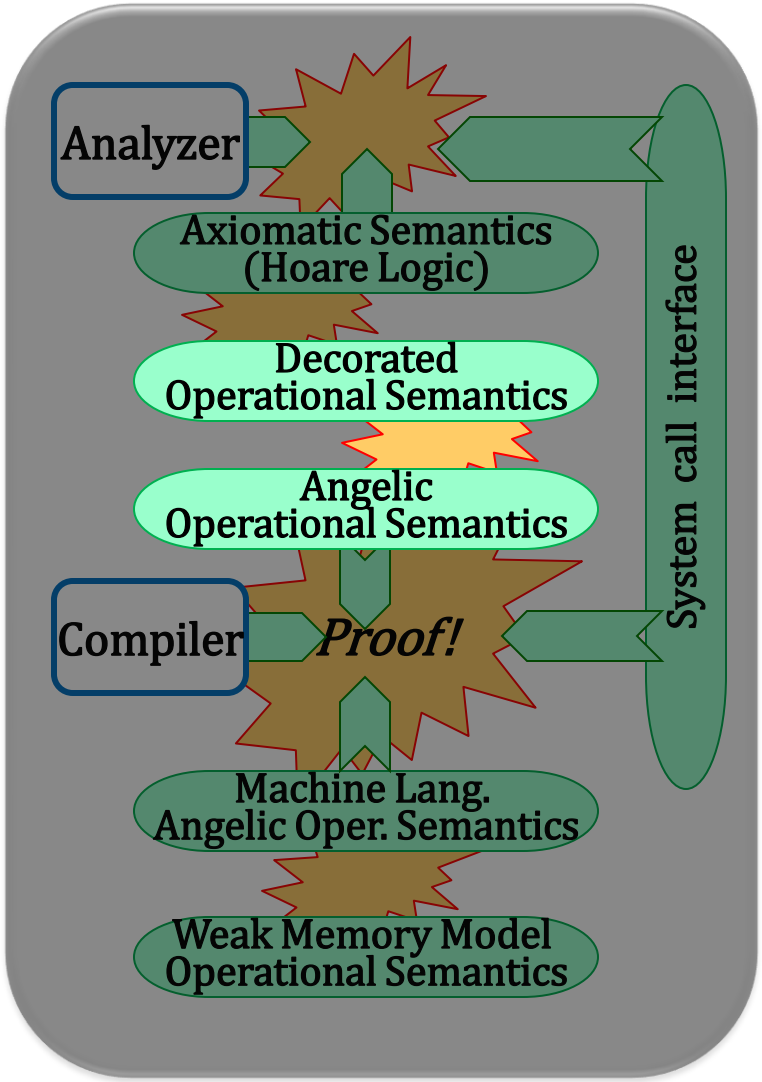
Appel, ESOP 2011 *(right now!)*
Robert Dockins PhD thesis 2012
Dockins et al. POPL 2012 ??

Remove predicates from states; use "angel" oracle to tell what portion of heap is given up at lock-release

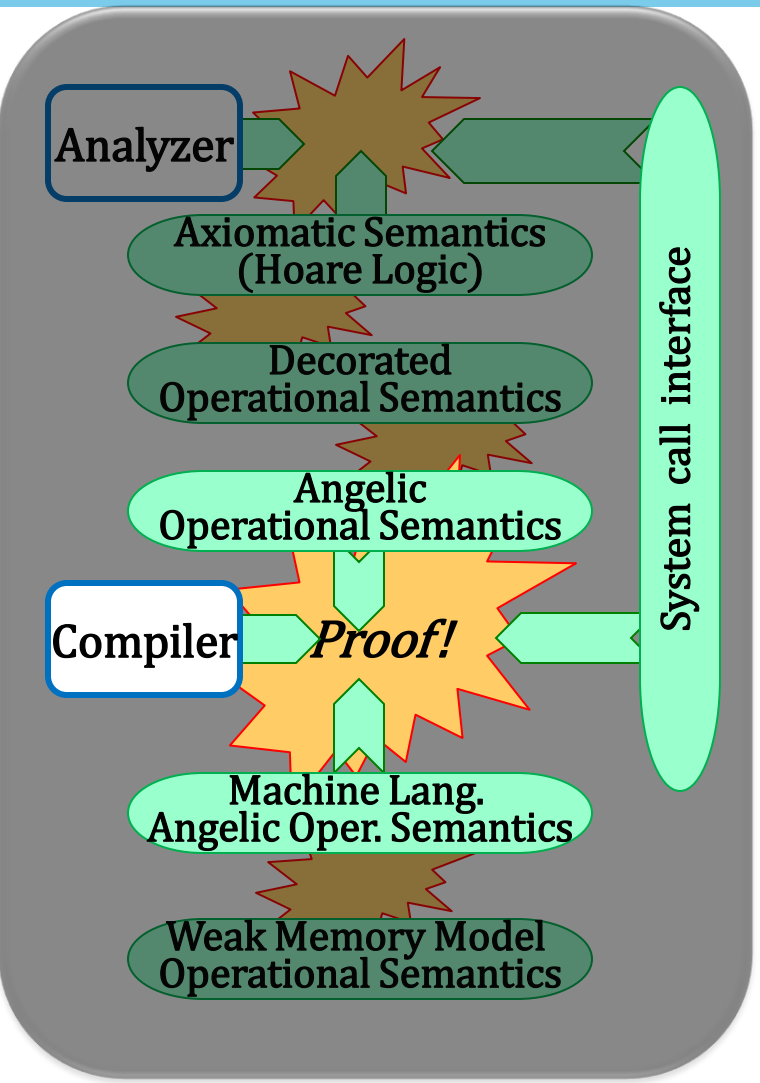


Theorem:

If decorated op.sem. safely executes, then ∃ angel that gives same sequence of subheaps



Compiler correctness



Theorem:

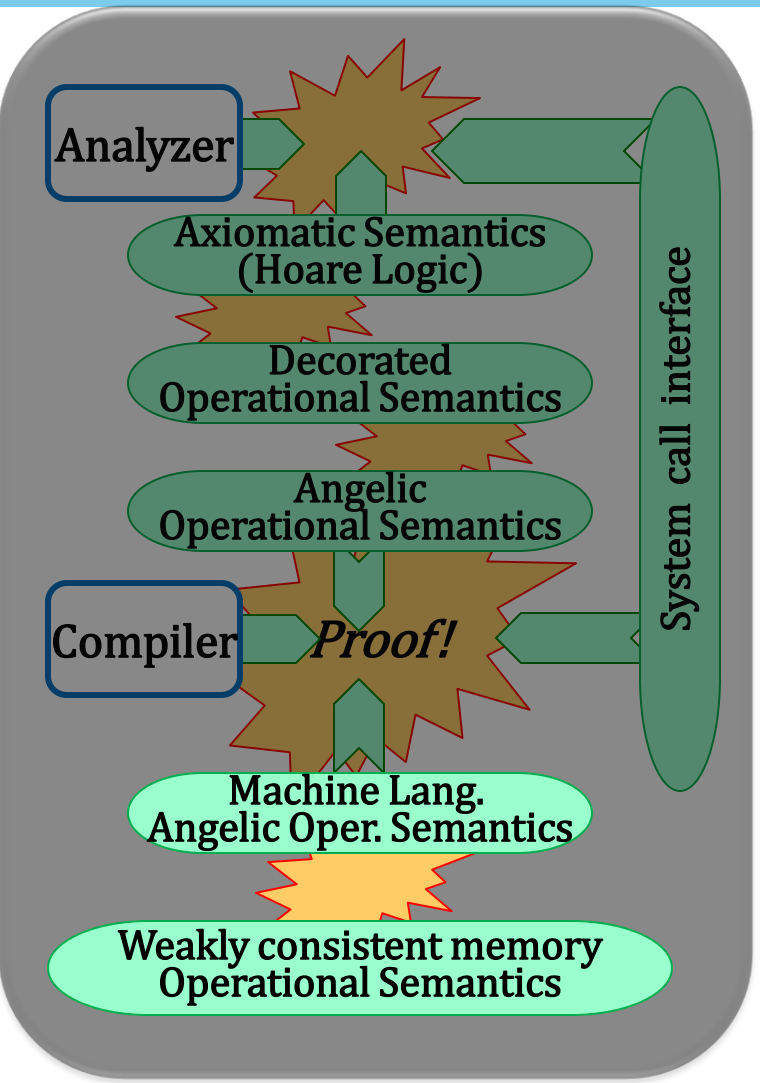
If

source-language program has noncrashing observable behavior *B*,

then

compiled program has same observable behavior *B*

Relaxed memory models



Toolchain proves and preserves this guarantee:

no race conditions

Therefore,

executions on relaxed memory models will have the correct observable behavior

Proof: Hobor & Alglave, in progress

END