Proof Technology for High-Assurance Runtime Systems

Andrew Tolmach, Andrew McCreight,
and the Programatica team

Portland State University

Functional Languages for High-Assurance Applications

- Goal: rely on properties of functional languages to build high-assurance software in cost-effective way
  - Improved productivity through abstraction
  - Memory safety
  - Type safety
  - Formal semantics (maybe!)
  - Easy reasoning about programs (maybe!)
- Especially interested in systems code
  - important, tricky
- Example: the House proof-of-concept OS [ICFP05]

A Credibility Gap

- House relies on services provided by the Glasgow Haskell Compiler (GHC) run-time system
  - currently around 35-50KLOC of complex C code
- Any assurance argument that we might make about House requires a corresponding argument about the run-time system
  - hard or impossible for existing RTS
- Situation is similar for many other high-level languages/implementations, e.g. Java

How to Bridge the Gap

- Reduce code **size**:
  - Eliminate functionality that we don't need
  - Eliminate accidental/historical complexity
- Re-implement in a **safer** language
- Re-implement with new goals
  - **Simplicity**
  - Ease of formal verification
- Stress formal **specification** of intended behavior

HARTS

High-**A**ssurance **RTS** for Haskell, Java, …

Services:

- Garbage collection
- Concurrency
- Interfacing to untrusted languages

First priority

Talk Outline

Motivation for HARTS

Verifying Garbage Collectors

Verifying Imperative Pointer Programs

Verifying Using Deep Embeddings, Separation Logic, and Tactics

Where Do GC Bugs Come From?

- Errors in algorithms
  - Especially for highly-concurrent algorithms
- Errors in GC implementation

Focus for Today

- Errors in mutator
  - Mutator must identify all roots
  - Mutator must respect GC data structures

Formalizing the contract is a critical first step

Portland State University

Principles for Verified GC

- Insist on **machine-checked** proofs
- Verify the actual **implementation**
  - Amortize the cost of verification over all uses
- Engineer a re-usable **framework** for future verifications of similar style
  - Amortize the cost of building the framework over multiple GCs
- **Build on** existing work
  - at INRIA (Leroy *et al*) on certified compilation
  - at Yale (Shao, McCreight, *et al*) on certified GCs

Feasibility

- Very few published machine-checked proofs of GC **implementations**
  - [FluetWang04,McCreight++07,Hawblitzel++07, Myreen08,....?]
- Typically 100-300 lines, and somewhat simplified

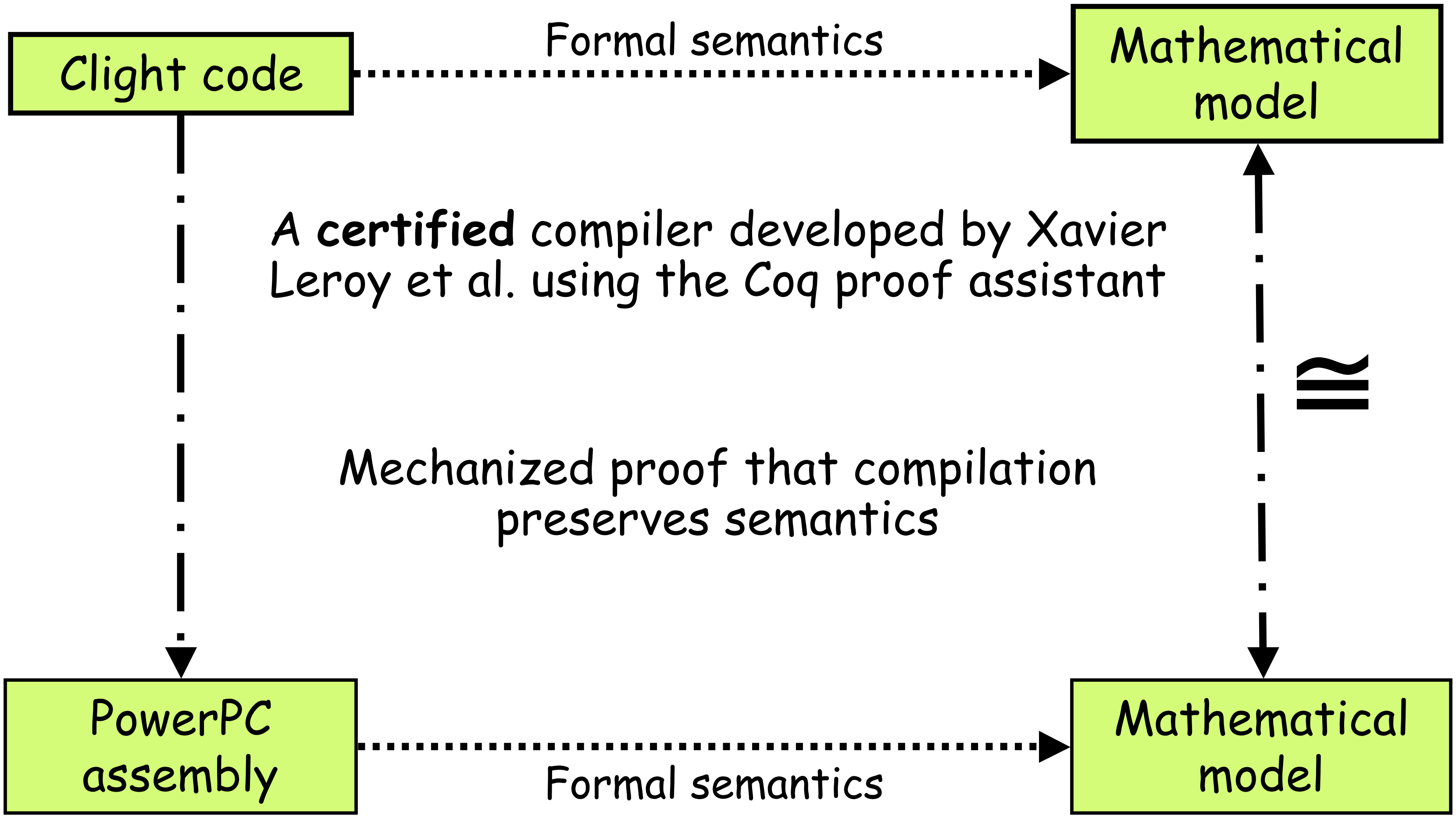
↕ Wanted: a proof methodology that will scale to GC's of this size and complexity

- There are fielded, production-quality GC implementations with good performance and support for a rich set of language features in 2000 LOC

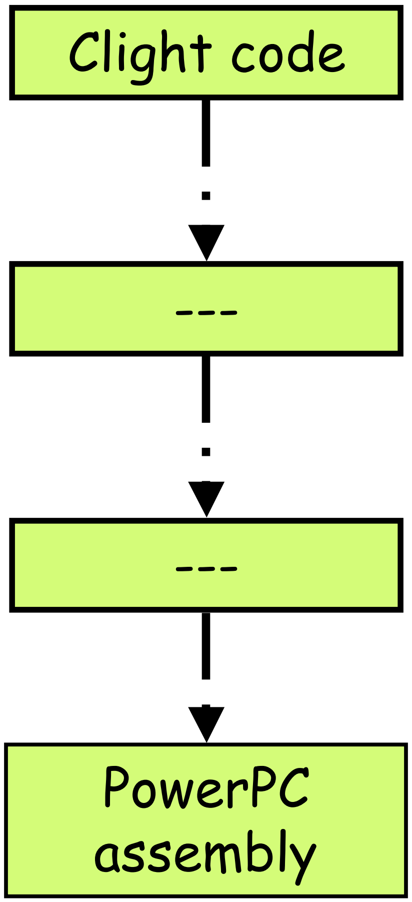
What about types?

- Long-standing goal: define a strongly-typed language rich enough to express collectors
- Proposals to date are complex
  - and only guarantee safety
- We're following a different path, based on general-purpose provers (e.g. Coq, Isabelle, etc.)
  - Ultimately, approaches may converge
- In any case, type-based approach may still be useful choice for verifying mutator behavior

The Compcert Framework

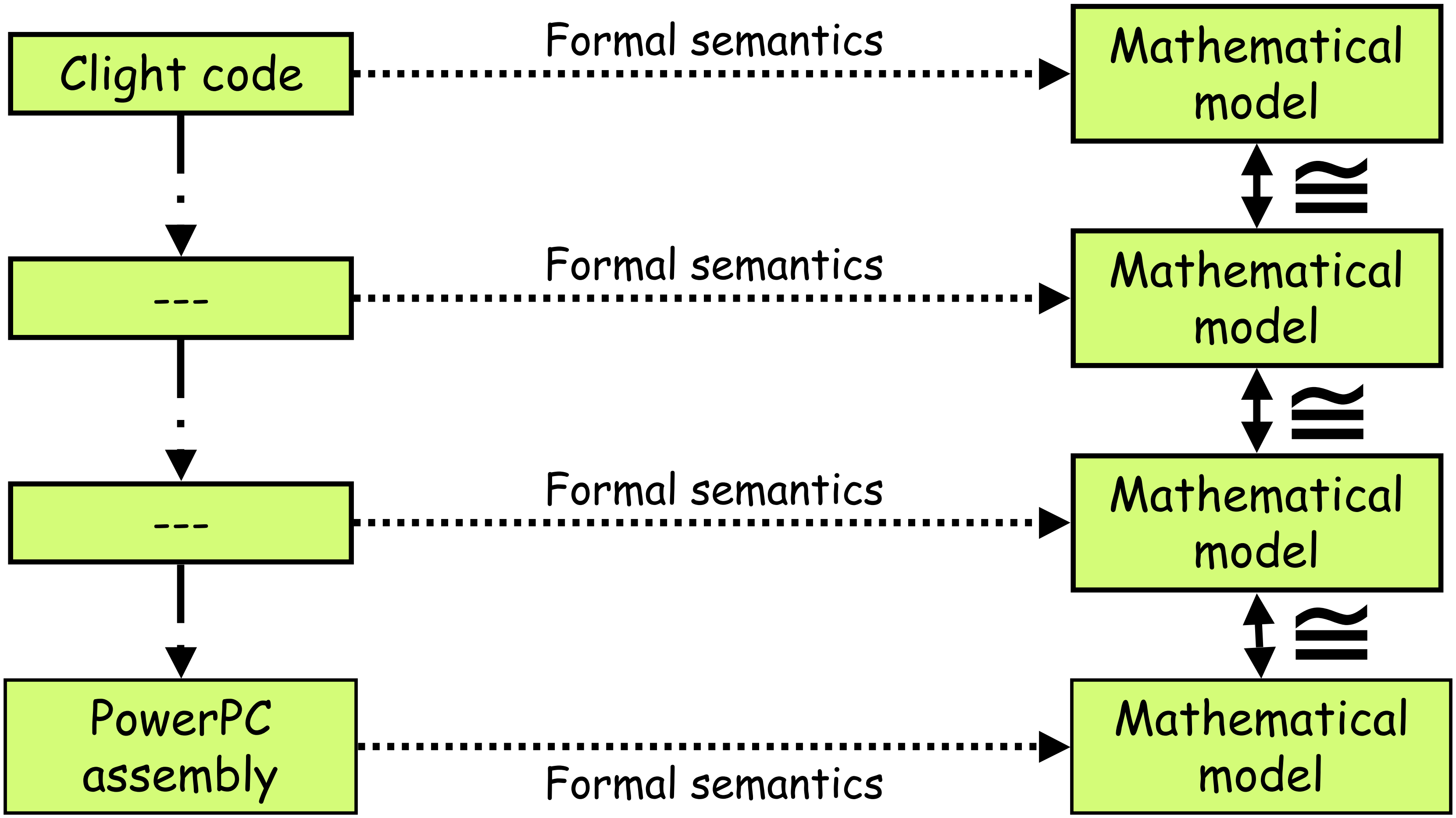


The Compcert Framework

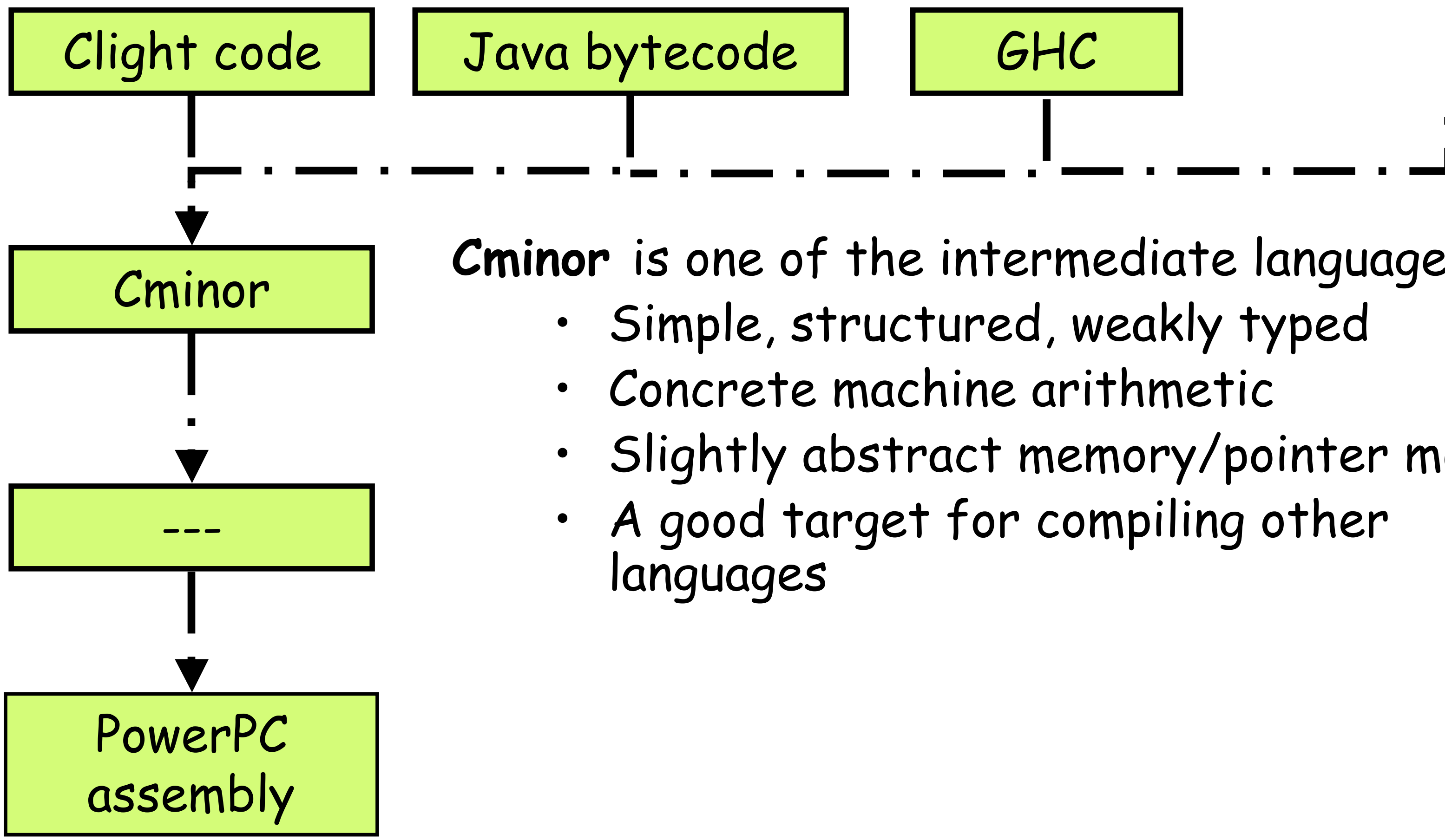


Implemented as a pipeline with multiple stages

The Compcert Framework



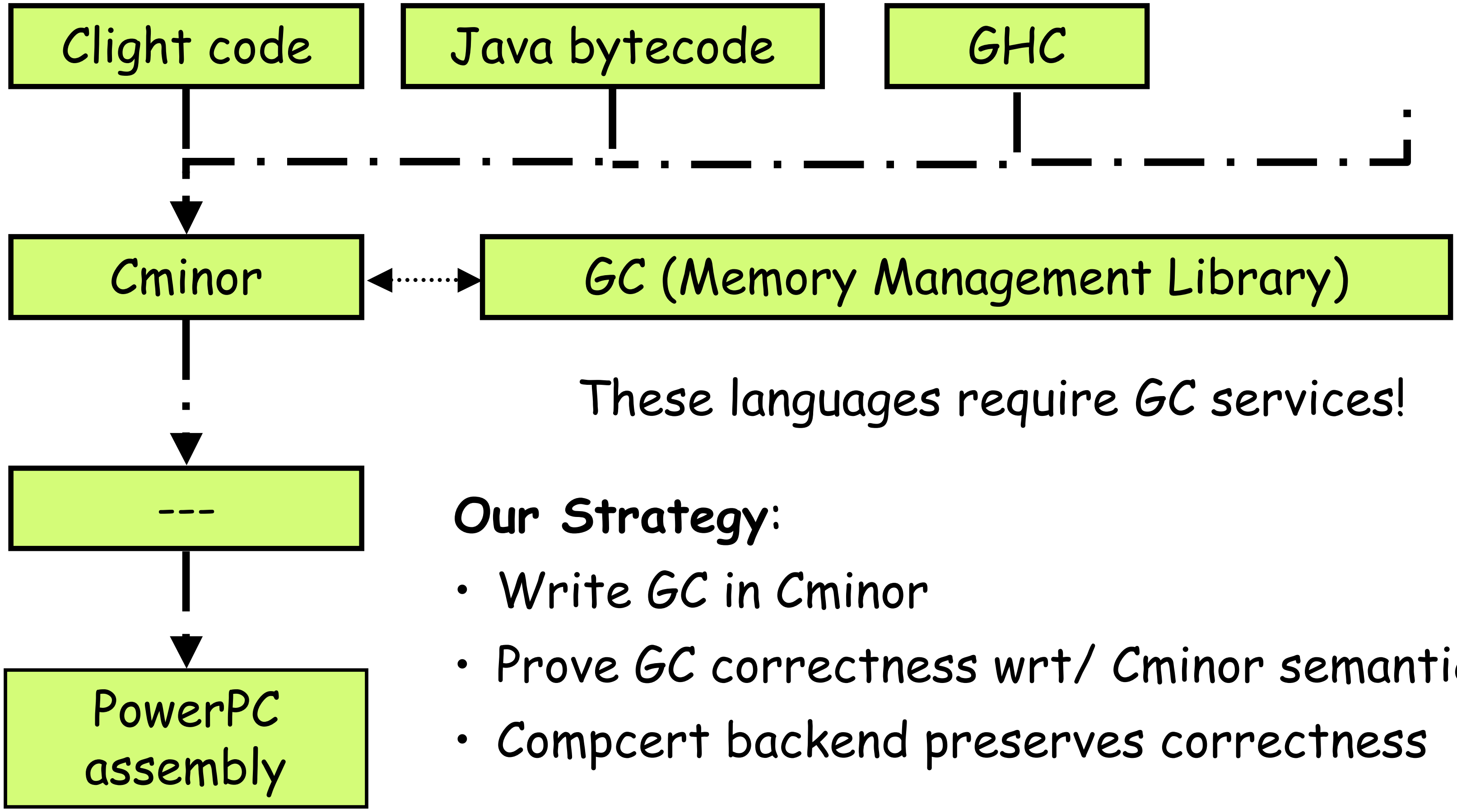
The Compcert Framework



Cminor is one of the intermediate languages

- Simple, structured, weakly typed
- Concrete machine arithmetic
- Slightly abstract memory/pointer model
- A good target for compiling other languages

The Compcert Framework



These languages require GC services!

Our Strategy:

- Write GC in Cminor
- Prove GC correctness wrt/ Cminor semantics
- Compcert backend preserves correctness

Compcert Semantic Framework

- Compcert IL behavior is specified by operational semantics
  - given as Coq inductive relation
  - bad programs just get stuck; no types needed
- Evaluation yields result and **trace** of system calls
- Semantic preservation at each compiler transformation means
  - at program level: result and trace preserved
  - at statement level: effect of statement on state is suitably simulated
  - etc.

Cheney-style GC code (1)

```
#define NULL_PTR 0

var "freep"[4]
var "toStartp"[4]
var "toEndp"[4]
var "frStartp"[4]
var "frEndp"[4]

"numFields" (x) : int -> int
{ return int32[x]; }

"fieldIsPointer" (x,k) : int -> int -> int
{ return int32[x+4] <= k; }

"memCopy" (src,dst,len) : int -> int -> int -> void
{ var i;
  i = 0;
  while (I < len) {
     int32[dst + 4 * i] = int32[src + 4 * i];
     i = i + 1;
  }
}
```

```
"scanPtrField" (xp,free) : int -> int -> int
{
    var x, len, hdr;

    x = int32[xp];
    if (x == NULL_PTR)
      return free;
    hdr = int32[x - 4];
    if (hdr != NULL_PTR) {
      len = "numFields"(hdr) : int -> int;
      "memCopy"(x - 4, free, len + 1) : int -> int -> int -> void;
      int32[x] = free + 4;
      int32[x - 4] = NULL_PTR;
      free = free + 4 * len + 4;
    }
    int32[xp] = int32[x];
    return free;
}
```

Cheney-style GC code (2)

```
"cheneyCollect" (rootp) : int -> int
{
   var hdr,len,toStart,toEnd,root,free,frStart,frEnd,scan,i,isPtr;

   frStart = int32["toStartp"];   toStart = int32["frStartp"];
   int32["toStartp"] = toStart;  int32["frStartp"] = frStart;
   toEnd = int32["frEndp"];     frEnd = int32["toEndp"];
   int32["toEndp"] = toEnd;    int32["frEndp"] = frEnd;

   free = "scanPtrField"(root, toStart) : int -> int -> int;
   scan = toStart;
   while (scan != free) {
      hdr = int32[scan];
      scan = scan + 4;
      len = "numFields"(hdr) : int -> int;
      i = 0;
     while (I < len) {
          isPtr = "fieldIsPointer"(hdr,i) : int -> int -> int;
          if (isPtr)
            free = "scanPtrField"(scan,free) : int -> int -> int;
          scan = scan + 4;
          i = i + 1;
     }
   }
}
```

```
"cheneyAlloc"(hdr,root) : int -> int -> int
{
   var free,len;

   free = int32["freep"];
   len = "numFields"(hdr) : int -> int;
   len = len * 4;
   if (len == 0)
     return 0;
   if (free + len + 4 >= int32["toEndp"]) {
     free = "cheneyCollect"(root) : int -> int;
     if (free + len + 4 >= int32["toEndp"])
       return 0;
   }
   int32["freep"] = free + len + 4;
   int32[free] = hdr;
   return (free + 4);
}
```

Proving Cminor Programs

- Just a special case of general task: proving properties of imperative pointer-based programs
- A long-standing but newly lively research area
- No single generally-accepted approach
- (NB. Different from Compcert's goal, which is about proving correctness of **transformations** on imperative programs)

Talk Outline

Motivation for HARTS

Verifying Garbage Collectors

Verifying Imperative Pointer Programs

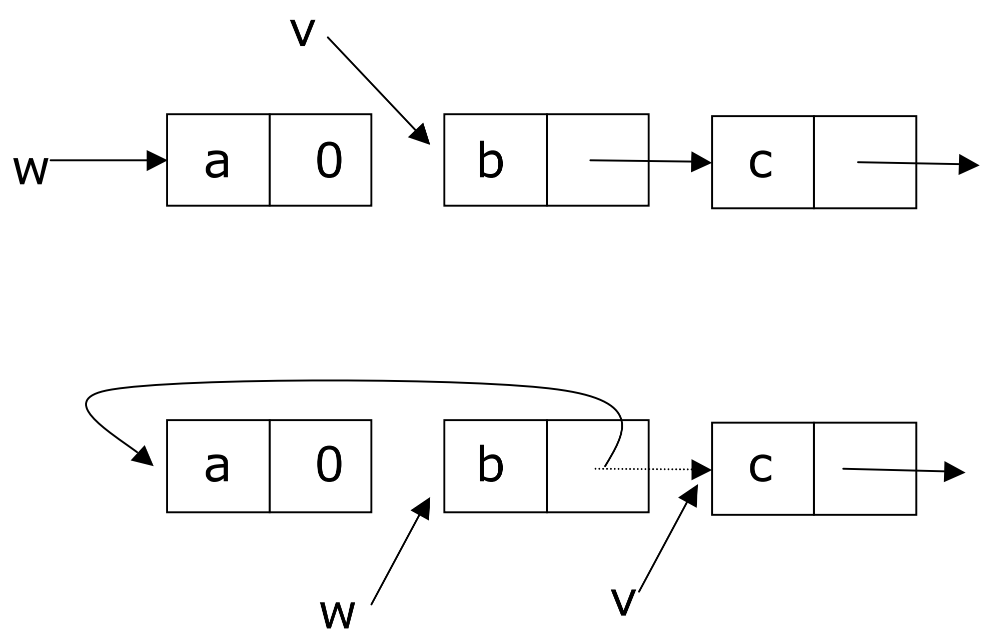
Verifying Using Deep Embeddings, Separation Logic, and Tactics

A naïve investigation

- What's the current state of the art?
- Started examining alternatives in Fall '06
- Caveats:
  - Was on sabbatical at INRIA Rocquencourt
  - Using a theorem prover for the first time
  - National bias towards Coq-based tools
- Case-study examples initially from [Mehta&Nipkow05]
- Assume that bulk of each proof will need to be done using an interactive prover

Example: in-place list reversal

```
"reverse" (v) : int -> int {
  var w,t;
  w = 0;
  while (v != 0) {
     t = int32[v + 4];
     int32[v + 4] = w;
     w = v;
     v = t;
  }
  return w;
}
```



Portland State University

Proving properties of reverse

```
"reverse" (v) : int -> int {
  var w,t;
  w = 0;
  while (v != 0) {
     t = int32[v + 4];
     int32[v + 4] = w;
     w = v;
     v = t;
  }
  return w;
}
```

Precondition: v points to a well-formed acyclic list with cell addresses vs = v,v2,v3, …vn

Loop invariant:

- v and w point to well-formed acyclic lists vs', ws'
- (rev vs') ++ ws' = rev vs
- vs' & ws' are disjoint

Loop termination condition: length of vs decreases at each iteration

Postcondition: return value points to a well-formed acyclic list with cell addresses vn,…,v2,v = rev vs

Not proven: contents of list don't change!

Three Coq-based Alternatives

- **Caduceus+Why -> Coq**
- Monadic shallow embedding + extraction
- Deep embedding + separation logic + tactics

Caduceus+Why [Filliatre+]

- Verification Condition (VC) generation from annotated imperative programs (C,Java,...)
  - function pre- and post- conditions
  - loop invariants, "variants" (termination measures)
  - assertions
- Targets many backend provers
  - both fully automated (Ergo,...) and proof assistants (Coq,...)
- No mechanized proof that VC extraction is correct

Example: specifying 'reverse'

- I'll skip the actual specification notation…
- By the time we've translated to Coq, our notion of a well-formed pointer list amounts to this:

```
Inductive Plist : Sto -> Ptr -> Ptr list -> Prop :=
| PlistNil : forall s, Plist s 0 nil
| PlistCons: forall s p ps, p <> 0 ->
        Plist s (s(p+4)) ps -> Plist s p (p::ps)
end.
```

- Note that the store is quite explicit

Invariant for 'reverse'

- Here's a suitable loop invariant:

```
Definition rev_inv (s:Sto) (v:Ptr) (vs: list Ptr)
                   (w:Ptr) (ws: list Ptr) (xs: list Ptr) :=
  Plist s v vs /\ Plist s w ws /\ disjoint vs ws /\
  rev vs ++ ws = rev xs.
```

- We must maintain explicit disjointness information in rev_inv, and via lemmas like this:

```
Lemma List_NoDup: forall s x xs,
   List s x xs -> NoDup xs.
```

- Can also use Bornat-style field-separation axioms

Example 'reverse' VC

- Here's the VC corresponding to maintenance of the loop invariant and "variant"

```
Lemma loop_ok :
  forall s0 v0 vs0,  Plist s0 v0 vs0 ->
  forall s v vs w ws,
  rev_inv s v vs w ws vs0 ->
  v <> null ->
  forall v', v' = load s (next v) ->
  forall s', s' = update s (next v) w ->
  rev_inv s' v' (tail vs) v (v::ws) vs0 /\
     length s' v' < length s v.
```

- Note that imperative operations on local variables are all gone

Assessment of Caduceus

+ Function and loop specs are (mostly) natural
+ Termination handling is separable -- very nice
+ Proof size reasonable (~ 138 lines for reverse)
- Coq translations of specs and VC's are **much** uglier than I've shown
- Very hard to connect VC's mentally to code positions/paths
- VC's can be **huge** and repetitive
  - e.g. 25 line in-place merge algorithm from [Mehta&Nipkow05] generated 6900 lines of VC's!

Many of these problems are "just" engineering issues

+ team is working on them
- but their focus is on fully automated paths

Three Coq-based Alternatives

- Caduceus+Why -> Coq
- **Monadic shallow embedding + extraction**
- Deep embedding + separation logic + tactics

Coq proofs for Coq functions

- The easiest subject for a Coq proof is a Coq program
  - i.e., a function written in the **C**alculus of **I**nductive **C**onstructions (CIC) itself
- Can then use Coq's **extraction** facility to get corresponding executable code in OCaml, etc.
  - Same properties should hold
  - Remaining proof obligation: extraction is correct...
- But CIC programs must be **pure** (and "obviously" **terminating**) and can be higher-order...

Monadic Shallow Embeddings

How can we adopt this approach to imperative pointer code?

Answer : Code programs using an abstract **state monad**! (And keep code first-order)

This gives a **shallow embedding**: our imperative program is represented by its denotation in CIC.

Must adjust extraction to get imperative operations instead of monadic encoding...

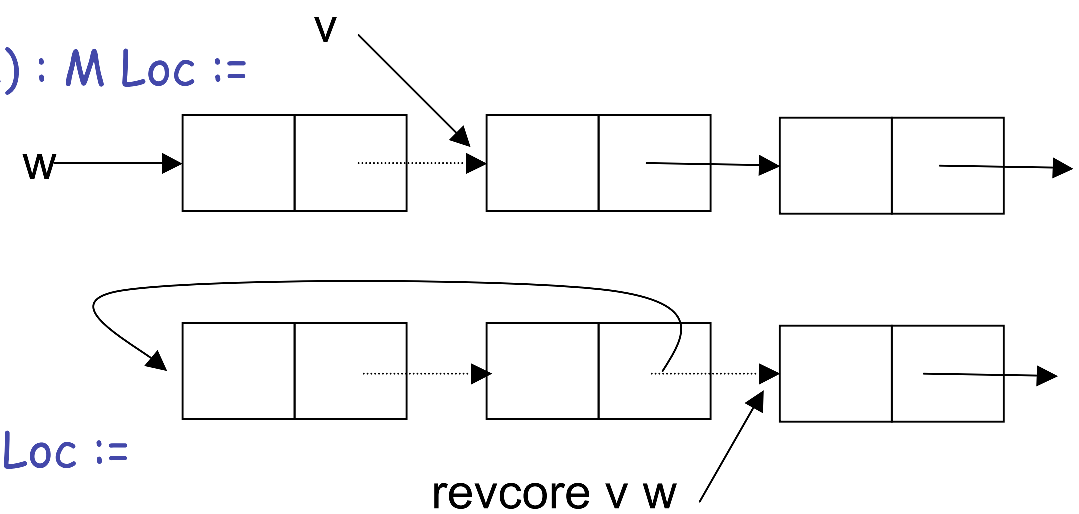
...or connect to imperative code another way

Defining the Store Monad

```
Definition Sto := Loc -> Val.
Definition update (s:Sto) (l:Loc) (v:Val) : Sto :=
  fun l0 => if eq_loc_dec l l0 then v else s l0.

Definition M (A:Set) := Sto -> Sto*A.
Definition Return (A:Set) (e:A) : M A := fun s => (s,e).
Definition Bind (A B:Set) (m : M A) (k : A -> M B) : M B :=
  fun s =>  let (s',a)  = m s in k a s'.

Definition Put (l:Loc) (v:Val): M unit :=  fun s => (update s l v,u).
Definition Get (l:Loc) : M  Val := fun s => (s,s l).

Definition run (A:Set) (s:Sto) (m: M A) : Sto*A :=  m s.
```

Monadic CIC example: 'reverse'

```
(* We pull this out to make a convenient spot
to state the  "loop" invariant.*)
Definition revcore (v:Loc) (w:Loc) : M Loc :=
  Get (tl v) >>= fun t =>
  Put (tl v) w >>
  Return t.
```




```
Fixpoint rev1 (v:Loc) (w:Loc) : M Loc :=
  if eq_loc_dec v null then
    Return w
  else
    revcore v w >>= fun t =>
    rev1 t v.

Definition revinplace (v : Loc) : M Loc := rev1 v 0.
```

Specs & proof for 'reverse'

- Specification is essentially similar to Caduceus style
- Proof (~ 80 lines) is also similar in substance, but code appears explicitly in hypotheses
  - We can "step through" it if we wish
- Proof "opens up" monadic abstraction, making heap state explicit
- Code is already functional, so no mutable local variables to worry about

What about Termination?

- All CIC functions must be "obviously" terminating
- So as written just now, rev1 wasn't valid Coq
- Recent Coq extensions use dependent types to allow termination obligations to be treated separately
  - Can get partial correctness by just admitting obligation
  - Proof terms can get messy: dependent types don't mix well with monadic abstraction
- Alternatively, we can add a decreasing measure as extra, artificial argument

Larger example: mark&sweep GC

Extremely simple heap model:

- two-word cons cells, each with one-word header (containing marked flag)
- all reachable cell contents are valid pointers (possibly null) -- no other values!

Extremely simple collector:

- single free list, linked through left children
- assume unbounded recursion stack, but...

To keep Coq happy, recursive mark routine has an extra depth parameter that bounds traversal

- (could be used to index an explicit mark stack)

Proofs for mark&sweep

- We specify and prove a strong correctness result for the collector
  - includes both safety and progress results
- Proof is ~ 2100 lines
- Side note: bounded marking has a much more complicated invariant than unbounded marking!
- Not a very realistic collector
  - No headers (beause fixed size, everything is a pointer)
  - Heap addresses are modeled as natural numbers

Imperative Code Extraction

- Can hack a post-processor for existing Coq extraction mechanism that converts **explicitly** monadic code to **implicitly** monadic code.
- Cleaner approach: get Coq team to support extraction to imperative languages directly
- But is the extraction process itself trustworthy anyhow?
  - There is a pencil&paper proof…
  - …and ongoing work to formalize this within Coq
- Basic idea: model the extraction target language within Coq using ASTs and an operational semantics
  - a **deep embedding**
  - **prove** shallow and deep embeddings are equivalent

Monadic CIC Assessment

+ Flexible proof organization & style
+ Good integration of programs and proofs
+ Pleasant (functional!) coding style
- Termination is a persistent problem
- Don't know how to mix monads with proof techniques based on dependent types
- Need a lot more engineering to automate and verify connection between CIC and imperative code

Three Coq-based Alternatives

- Caduceus+Why -> Coq
- Monadic shallow embedding + extraction
- **Deep embedding + separation logic + tactics**

Just use Deep Embeddings?

McCreight, Shao et al. (working at Yale) have produced impressive GC proofs on a deeply-embedded MIPS-like machine code

Appel & Blazy (working at INRIA) have suggested doing program proofs directly on a deep embedding of CMinor

Proofs require a **program logic** describing the target language's behavior

These authors also use **separation logic**

- avoid need for much explicit separation reasoning in proofs

Strong need for specialized **tactics** to work with these encoded logics

Initial Assessment : Mixed

+++ Proofs apply directly to the imperative program representation (and to Compcert certified compiler chain)

--- Working directly with the semantic evaluation relation is hard!

- Yale work took many graduate-student-years
- Specialized tactics seem essential
- But tactics are hard to develop and maintain (e.g. Appel&Blazy's don't quite work yet)…
- …and they are fragile, leaving you at the mercy of the expert tactic author!

Three Coq-based Alternatives

- Caduceus+Why -> Coq
- Monadic shallow embedding + extraction
- Deep embedding + separation logic + tactics

Overall assessment:

- All have promise
- None quite works
- Not clear which is best bet

But we had to move forward somehow…

Talk Outline

Motivation for HARTS

Verifying Garbage Collectors

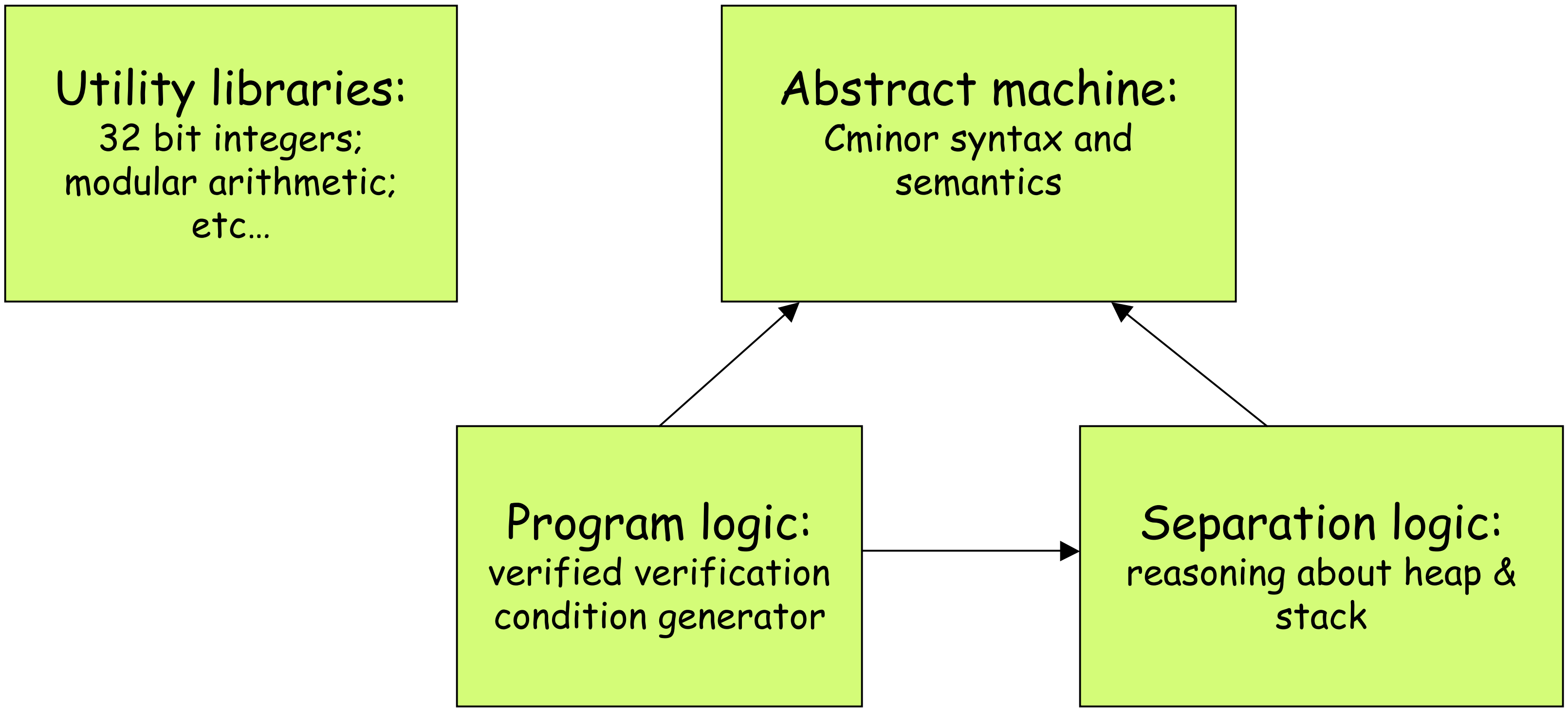
Verifying Imperative Pointer Programs

Verifying Using Deep Embeddings, Separation Logic, and Tactics

HARTS project approach

- Hired Andrew McCreight!
- Using a deep embedding of Cminor
- Using separation logic
- Building a substantial tactic framework
- Have already used it to prove a Cheney-style collector
    - Fairly realistic features
        - especially: true machine arithmetic
    - Fairly high level of automation

Framework Overview



Everything is implemented in the Coq proof assistant

Separation Logic

- Logic for reasoning about heaps [Reynolds, O'Hearn]

- Key predicates:
- P * Q Heap is split into two disjoint parts
  P holds on one part, Q on the other
- $x \mapsto v$ Holds on a heap containing only address x that contains value v

- Neatly encapsulates complexities of reasoning about pointer-based programming (aliasing, etc.)

Example: Linked Lists

- Relating list values to in-memory representation:

```
Inductive Plist : val -> list val -> mem -> Prop :=
| Plist_nil : Plist null_ptr nil m
| Plist_cons : forall x xs t m,
    (lexists v, x ↦ v * ((x+4) ↦ t) * Plist t xs) m ->
     Plist x (x::xs) m.
```

- Separating conjunction enforces that elements are disjoint (and hence lists are acyclic)

Separation Logic Tactics

- Simplification: sle/sli

  ((B * true) * (emp * D) * true) m

  → (B * D * true) m

- Re-arrangement: assocPerm [3, [4, 1], 2]

  $(A^{1} * B^{2} * C^{3} * D^{4})\ m \rightarrow$

  $(C_{3} * (D_{4} * A_{1}) * B_{2})\ m$

- Matching:

  Hypothesis: (A * B * C * D) m

  Goal: (B * C * A * D) m

  searchMatch solves this immediately

Program Logic

- Hoare-style reasoning using pre- and post-conditions
  - Similar to program logic of [Appel&Blazy07]
- **Verified** verification condition generation
  - Generator calculates a VC for each statement
  - Generated VC proven consistent with original operational semantics

Verification Conditions

precondition of next statement

initial state

- Example: vc (x := e) Q s

  $= \exists\, v.\ e \xrightarrow{s} v$

  $\wedge\ Q(s\{x{:=}v\})$

- Extra predicate arguments are added for return, call, and jump
- Infrastructure provides tools for helping to prove VCs automatically

VC Proof Tactics

- Automatically analyze the VC
  - Break down a complex expression into substeps
  - Look for hypothesis to solve a single step
    - e.g. if loading from x, do we know what x contains?
  - Often need to manually transform a hypothesis
    - e.g. to apply elimination rules for data structures like Plist
- Branch splitting
  - Analyze the result of the branch
    - e.g. if test is (x >=4), then in true branch we know x is defined and $x \geq 4$

Proof Example: List Reverse

Lemma reverseOk : fdefOk reversePre reversePost reverseDef.

Pre-condition:

```
Definition
 reversePre is args:=
   lexists i, !(args=i::nil) *
                         plist i is.
```

Post-condition:

```
Definition
 reversePost is result :=
     plist result (rev is).
```

Loop Invariant:

```
Definition inv is (s:cstate) :=
  exists w, exists v,
    (vfEqv (xv :: xw :: xt :: nil) ((xw,w) :: (xv, v) :: nil) (cvfOf s) /\
      (lexists vl, lexists wl,
        plist v vl * plist w wl * !(rev vl ++ wl = rev is)) (cmemOf s)).
```

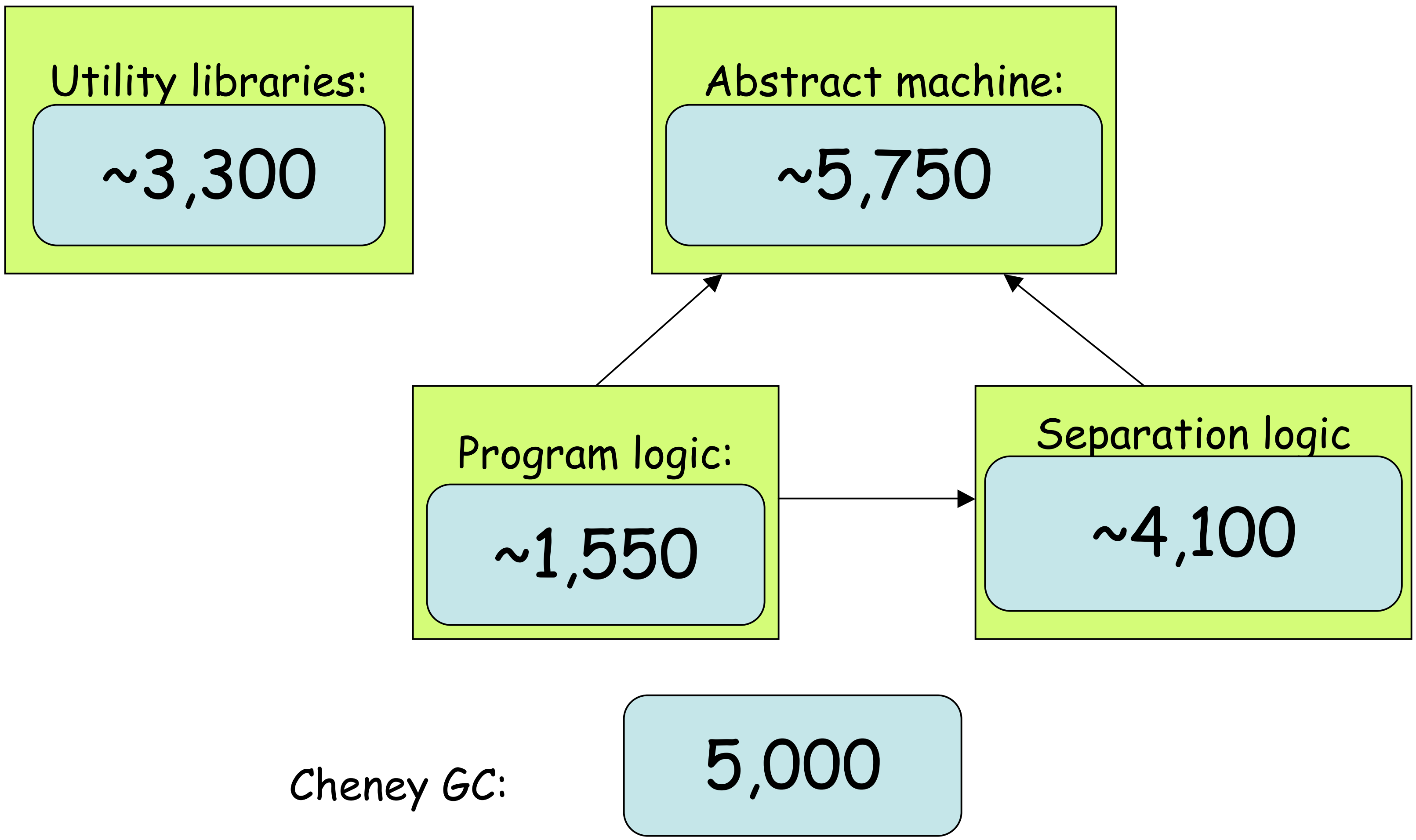
Proof Details:

DEMO!!

- Main proof: ~ 45 lines
- Similar length and complexity as for our proof of the same result using shallow embedding
- Program logic and Separation logic **tactics** make this possible.

```
Lemma reverseOk : fdefOk P0 reverseTy reversePre reversePost
                         reverseDef.
Proof.
  fdefBegin.
  unfold reversePre. intros is args m sp Hp. sle Hp. subst args.
  split. reflexivity.
  intros vf VFE.  simpl in VFE.
  vcSteps.

  exists (inv is). split.

  (* establish loop invariant *)
  unfold inv.
  exists null_ptr. exists x. split; auto.
  exists is. exists (@nil val).
  simpl. sli.
  auto with datatypes.

  (* loop entry *)
  clear VFE Hp.
  intros.  destruct s'. destruct H as [w [v0 [VFE Hp]]].
  vcSteps.
  branchStep.

  (* true branch: establish postcondition *)
  sli. unfold reversePost.
  subst v0. sle Hp.
  srewrite plist_null in Hp. sle Hp.
  subst x0. simpl in H. subst x1.
  apply Hp.

  (* false case: do loop body *)
  sle Hp.
  srewrite plist_non_null in Hp; [sle Hp | auto].
  vcSteps.
  (* bottom of loop body: re-establish invariant *)
  rewrite H0 in H; simpl in H; rewrite app_ass in H; simpl in H.
  unfold inv.
  exists v0. exists x3. split.
  vfEqvSolver.
  exists (tail x0). exists (v0::x1).
  simpl; sli.
  searchMatch.

  (* undefined case : impossible *)
  subst v0; sle Hp.
  srewrite plist_not_undef in Hp. sle Hp. auto.
Qed.
```

Infrastructure Line Counts



Cheney-style GC Proof Spec

Lemma cheneyCollectorOk :
fdefOk cheneyCollectorPre cheneyCollectorPost cheneyCollectorDef.

```
Definition cheneyCollectorPre objs fields cmap (rootp:addr) root C cl
  (frStart frEnd toStart toEnd:addr) (vv:list val) :=
  let objsAddrs := objs_addrs objs cl cmap in
   !(vv = (rootp:val)::nil /\
     (root = null_ptr \/ ptr_In root objs) /\
     contiguous frStart objsAddrs /\
     (Z_of_nat (AS.cardinal objsAddrs) < indexBound)%Z) **
  rootp |-> root **
  clDescrs C cmap ** gcInfo toStart toEnd frStart frEnd **
  okObjHp C cmap objs objs cl fields **
  buffer toStart (AS.cardinal objsAddrs).
```

Pre-condition

```
Definition cheneyCollectorPost (objs:AS.t) (fields:addr->list val) cmap
  rootp root C (cl:addr->addr) (frStart frEnd toStart toEnd:addr) (v:val) :=
  lexists M, lexists phi,
   let objs' := AASetMap.map phi M in
   let cl' := seq (inv M phi) cl in
   let fields' := seq (inv M phi) fields in
   let objsAddrs := objs_addrs objs cl cmap in
   let objs'Addrs := objs_addrs objs' cl' cmap in
   let free := toStart + 4 * AS.cardinal objs'Addrs in
    !(map_inj M phi /\
      (forall x, AS.In x M -> vaReachable cmap cl fields root x) /\
      (root = null_ptr \/ ptr_In root M) /\
      AS.Subset M objs /\
      contiguous toStart objs'Addrs /\
      v = free) **
    rootp |-> fwd_ptr phi root **
    okObjHp C cmap objs' objs' cl' (fwd_objs_fields cmap cl' phi fields') **
    buffer frStart (AS.cardinal objsAddrs) **
    clDescrs C cmap ** gcInfo frStart frEnd toStart toEnd **
    buffer free (AS.cardinal objsAddrs - AS.cardinal objs'Addrs).
```

Post-condition

```
#define NULL_PTR 0

var "freep"[4]
var "toStartp"[4]
var "toEndp"[4]
var "frStartp"[4]
var "frEndp"[4]

"numFields" (x) : int -> int
{ return int32[x]; }

"fieldIsPointer" (x,k) : int -> int -> int
{ return int32[x+4] <= k; }

"memCopy" (src,dst,len) : int -> int -> int -> void
{ var i;
  i = 0;
  while (I < len) {
     int32[dst + 4 * i] = int32[src + 4 * i];
     i = i + 1;
  }
}

"scanPtrField" (xp,free) : int -> int -> int
{
    var x, len, hdr;

    x = int32[xp];
    if (x == NULL_PTR)
      return free;
    hdr = int32[x - 4];
    if (hdr != NULL_PTR) {
      len = "numFields"(hdr) : int -> int;
      "memCopy"(x - 4, free, len + 1) : int -> int -> int -> void;
      int32[x] = free + 4;
      int32[x - 4] = NULL_PTR;
      free = free + 4 * len + 4;
    }
    int32[xp] = int32[x];
    return free;
}

"cheneyCollect" (rootp) : int -> int  {
    var hdr,len,toStart,toEnd,root,free,frStart,frEnd,scan,i,isPtr;

    frStart = int32["toStartp"];
    toStart = int32["frStartp"];
    int32["toStartp"] = toStart;
    int32["frStartp"] = frStart;
    toEnd = int32["frEndp"];
    frEnd = int32["toEndp"];
    int32["toEndp"] = toEnd;
    int32["frEndp"] = frEnd;

    free = "scanPtrField"(root, toStart) : int -> int -> int;
    scan = toStart;
    while (scan != free) {
       hdr = int32[scan];
       scan = scan + 4;
       len = "numFields"(hdr) : int -> int;
       i = 0;
      while (I < len) {
           isPtr = "fieldIsPointer"(hdr,i) : int -> int -> int;
           if (isPtr)
             free = "scanPtrField"(scan,free) : int -> int -> int;
           scan = scan + 4;
           i = i + 1;
      }
    }
}

"cheneyAlloc"(hdr,root) : int -> int -> int
{
    var free,len;

    free = int32["freep"];
    len = "numFields"(hdr) : int -> int;
    len = len * 4;
    if (len == 0)
      return 0;
    if (free + len + 4 >= int32["toEndp"]) {
      free = "cheneyCollect"(root) : int -> int;
      if (free + len + 4 >= int32["toEndp"])
        return 0;
    }
    int32["freep"] = free + len + 4;
    int32[free] = hdr;
    return (free + 4);
}
```

Definition

GC Achievements to Date

- We've proved correctness of a realistic GC implementation written in Cminor
- Advances on our (McCreight's) previous work:
  - **Uses true machine arithmetic**
  - Supports arbitrary record sizes
  - Supports precise pointer information
- Next steps: Must ensure that mutator keeps to its part of the GC contract …
- Next steps: Proof of generational collector

Conclusions

- Assurance of programs written in high-level languages requires assurance of underlying run-time systems
- Tools and techniques for reasoning about run-time system code are still young and little tested
- Results described today:
    - A verified implementation of realistic GC
    - A general verification infrastructure for GCs and other code that manipulates the heap
    - Essential use of tactics to automate reasoning

- An enabling step towards the use of high-level languages for high-assurance applications.