Run-time Systems for High-Assurance Systems Programming

Mark P. Jones, Andrew McCreight,
and Andrew Tolmach

Vision

- A tool-chain for developing robust, reliable, and secure systems software that spans the full range of concerns:
  - From high-level analysis and verification
  - To low-level, performance-sensitive implementation

Focus: Systems Software

Examples: Operating system kernels, Hypervisors, VMMs, Device Drivers, etc…

- **Essential**: key components in any computer system
- **Critical**: failures can compromise higher-level, "secure" application layers

Portland State UNIVERSITY

Current Practices

Industry practices for developing systems software typically rely on fairly low-level languages and tools (e.g., C)

+ Enables programmers to address low-level issues & performance concerns

- Harder to reason formally about the code

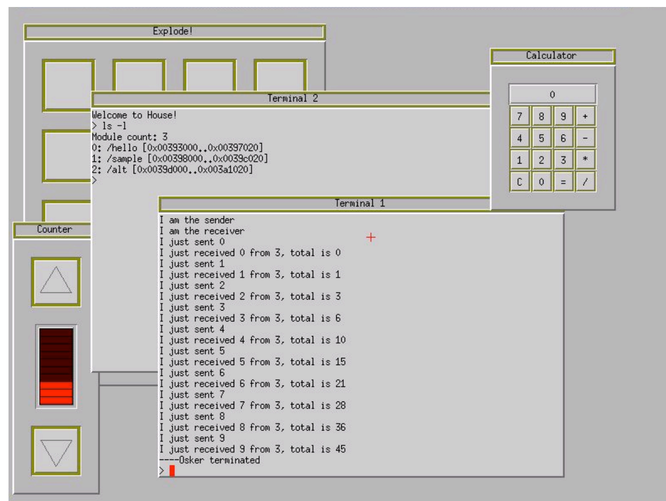
Using a Functional Language ...

- Functional languages can provide:
  - Improved productivity
  - Memory safety
  - Type safety
  - Formal semantics
  - Connections to proof assistants, etc…
- But can you build systems software in a functional language?

Portland State UNIVERSITY

Welcome to our House

- House is a proof-of-concept OS, written in Haskell:
  - Kernel + basic drivers (~5KLOC)
  - Network driver (~2KLOC)
  - GUI (~6KLOC)
  - Apps
  - User programs
- A starting point for the Galois HaLVM



Portland State UNIVERSITY

A Credibility Gap

- House relies on services provided by the "GHC run-time system":
  - a general purpose software component
  - currently around 35-50KLOC of C code

- Any assurance argument that we might make about House requires a corresponding argument about the run-time system

Portland State UNIVERSITY

How to Bridge the Gap

- Reduce code size:
  - Eliminate functionality that we don't need
  - Eliminate accidental/historical complexity

- Implement in a framework that supports formal verification

Portland State UNIVERSITY

Feasibility

Example: Garbage Collection (GC)

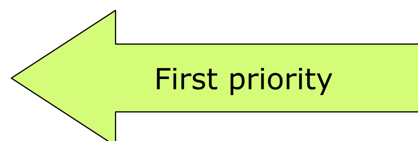
- A simple, verifiable GC can be written in 100-300 lines
- There are fielded, production-quality GC implementations with good performance and support for a rich set of language features in 2KLOC

Portland State UNIVERSITY

HARTS

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

- Design Philosophy:
  - "As simple as possible"
  - Modular
  - Formal verification
- Services:
  - Garbage collection
  - Concurrency
  - Interfacing to untrusted languages



Portland State University

Memory Allocation

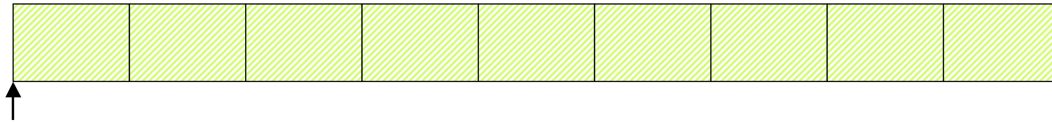
- Languages like C provide run-time libraries for allocating and freeing memory
- It is notoriously difficult to use these functions correctly:
  - Freeing memory too early can result in corruption and crashes
  - Freeing memory too late can result in space leaks and denial of service
- Memory allocation bugs are very hard to find

Portland State University

Garbage Collection

- A mechanism for reclaiming and reusing unused memory automatically
- Many different approaches:
  - Reference Counting, Mark-sweep, Stop-and-copy, Generational, Incremental, Real-time, …
- Programmer doesn't free memory by hand:
  - Less code to write
  - Fewer memory allocation bugs

Stop-and-copy Garbage Collection



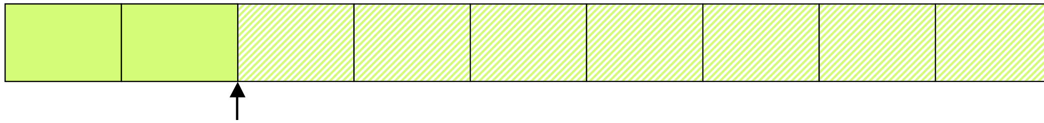
Allocate memory from a "heap"

Stop-and-copy Garbage Collection

Allocate memory from a "heap"

Portland State UNIVERSITY

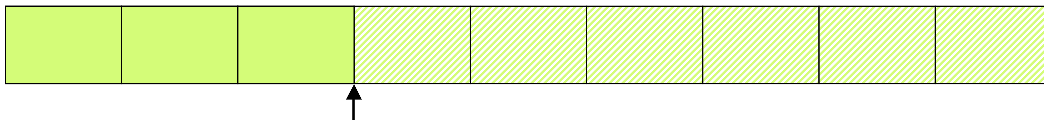
Stop-and-copy Garbage Collection



Allocate memory from a "heap"

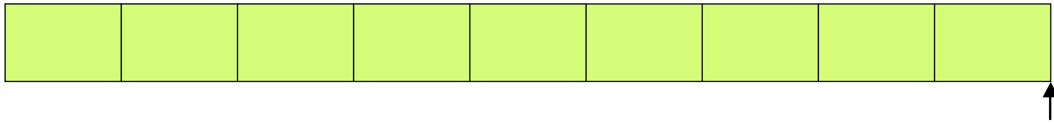
Portland State UNIVERSITY

Stop-and-copy Garbage Collection



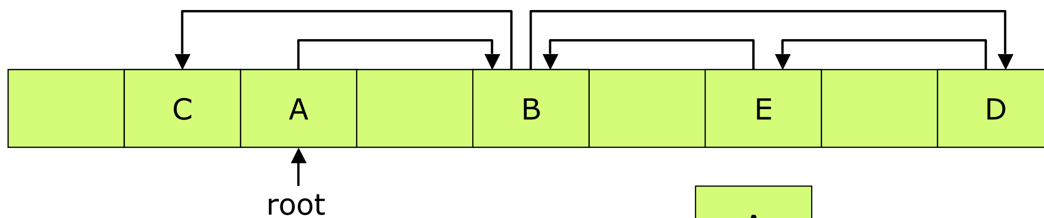
Allocate memory from a "heap"

Stop-and-copy Garbage Collection

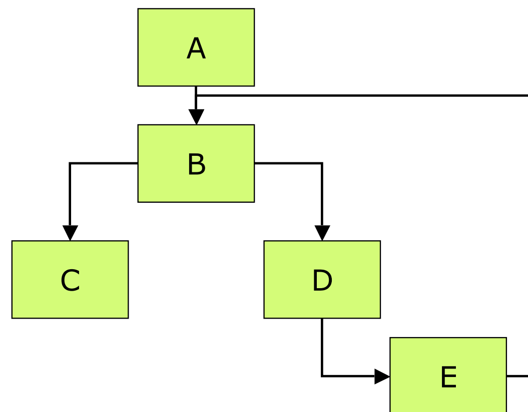


Eventually, the heap is full!

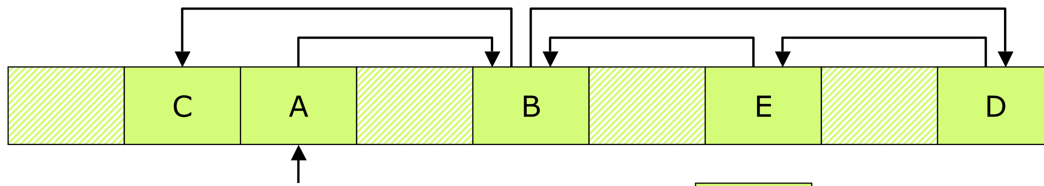
Stop-and-copy Garbage Collection

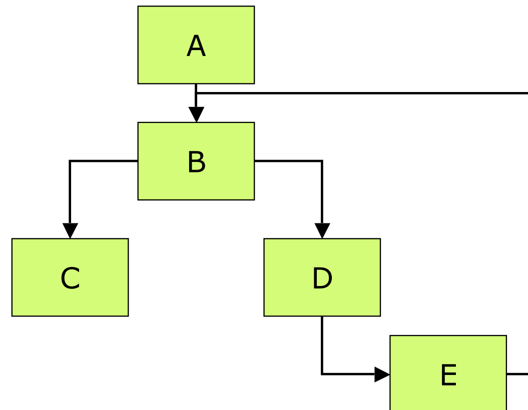


But we may only be using
a part of this …



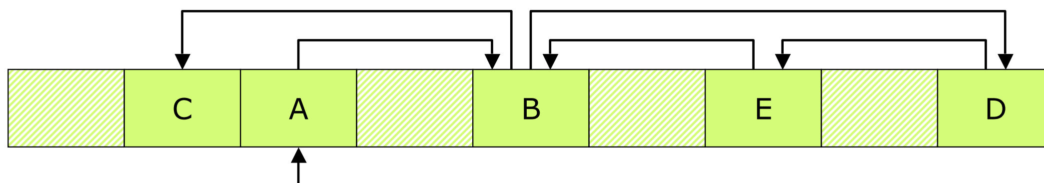
Stop-and-copy Garbage Collection





Everything else is "garbage"

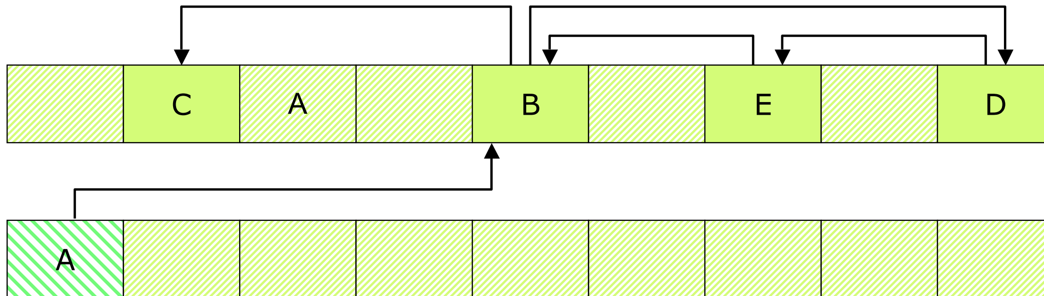
Stop-and-copy Garbage Collection



Assume that we have a second block of memory that we can use as a new heap

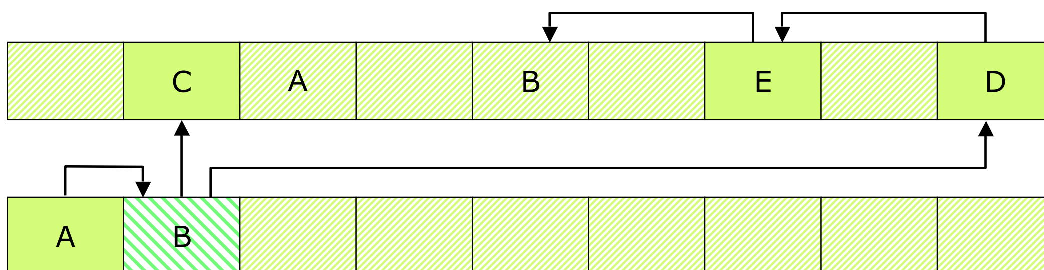
(Algorithm due to Cheney, 1970)

Stop-and-copy Garbage Collection



Copy A into the new heap

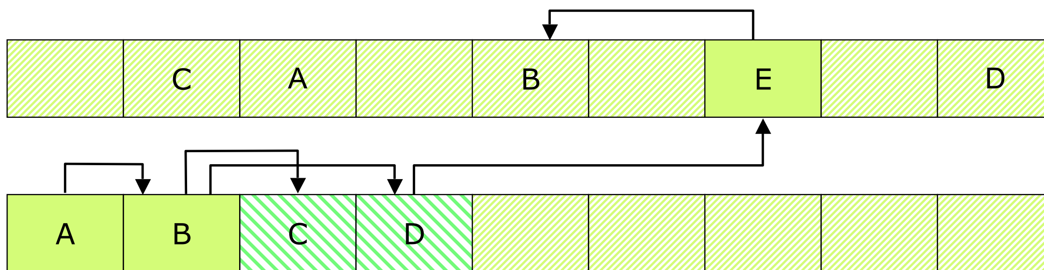
Stop-and-copy Garbage Collection



Scavenge A (copy B into the new heap)

Portland State University

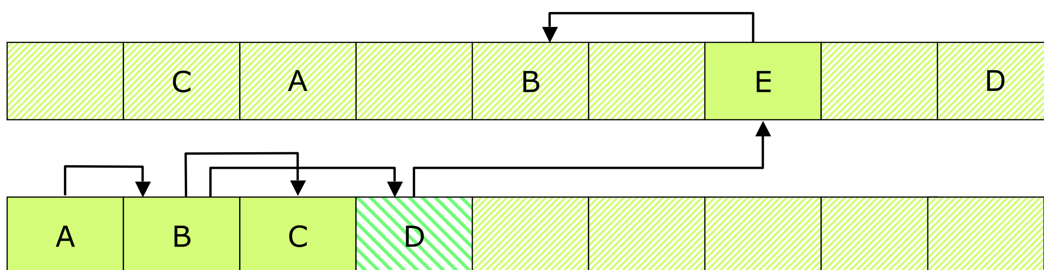
Stop-and-copy Garbage Collection



Scavenge B (copy C and D into the new heap)

Portland State University

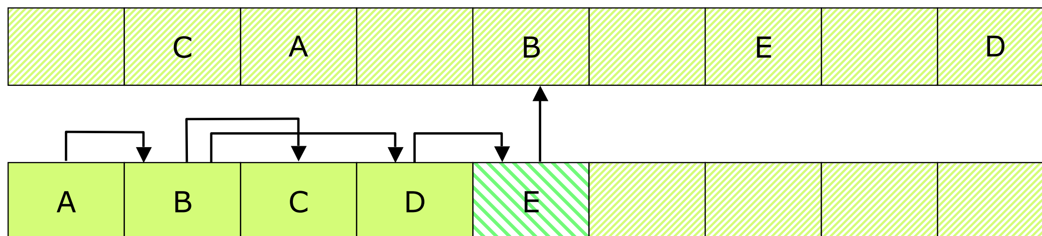
Stop-and-copy Garbage Collection



Scavenge C (no objects copied)

Portland State UNIVERSITY

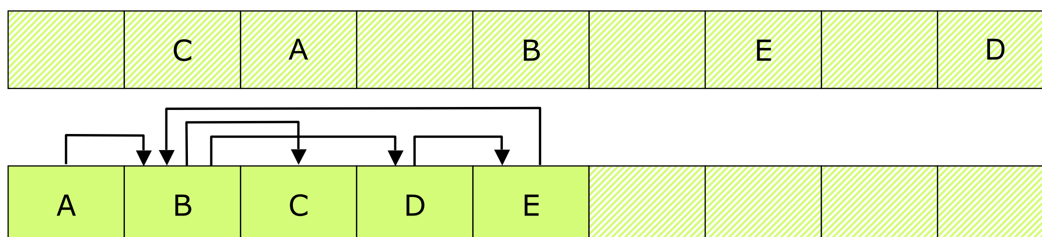
Stop-and-copy Garbage Collection



Scavenge D (copy E into the new heap)

Portland State UNIVERSITY

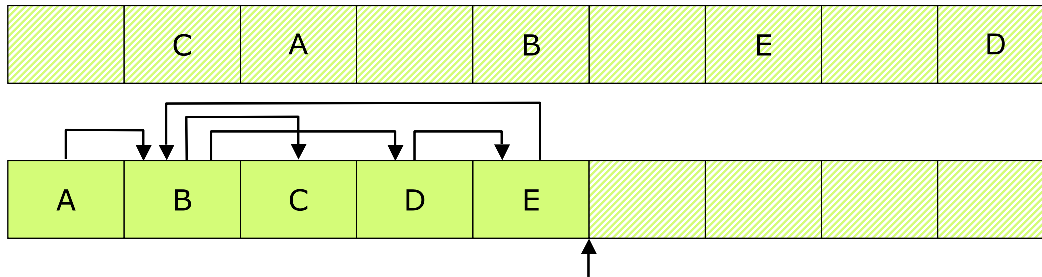
Stop-and-copy Garbage Collection



Scavenge E (B is already in the new heap)

Portland State UNIVERSITY

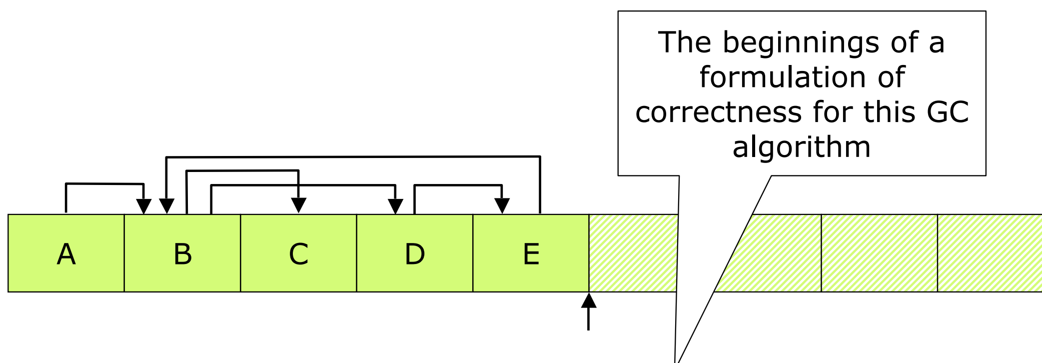
Stop-and-copy Garbage Collection



- All live data has been copied to the new heap;
- The original data structure has been preserved;
- Unused memory reorganized in a single block.

Portland State UNIVERSITY

Stop-and-copy Garbage Collection



- All live data has been copied to the new heap;
- The original data structure has been preserved;
- Unused memory reorganized in a single block.

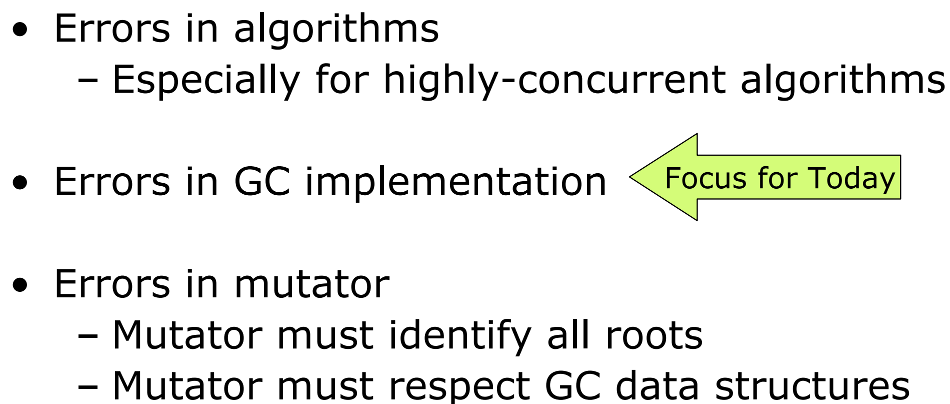
Garbage Collectors have Bugs Too!

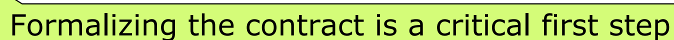
- GC bugs show up regularly in public bug and vulnerability databases
- **Example**: Widely used browsers (IE, Firefox, Safari), have all suffered from JavaScript engine GC bugs that can lead to:
  - browser crashes
  - denial of service attacks
  - execution of arbitrary code

Portland State UNIVERSITY

Where Do GC Bugs Come From?

- Errors in algorithms
  - Especially for highly-concurrent algorithms
- Errors in GC implementation
- Errors in mutator
  - Mutator must identify all roots
  - Mutator must respect GC data structures





Portland State UNIVERSITY

Principles for Verified GC

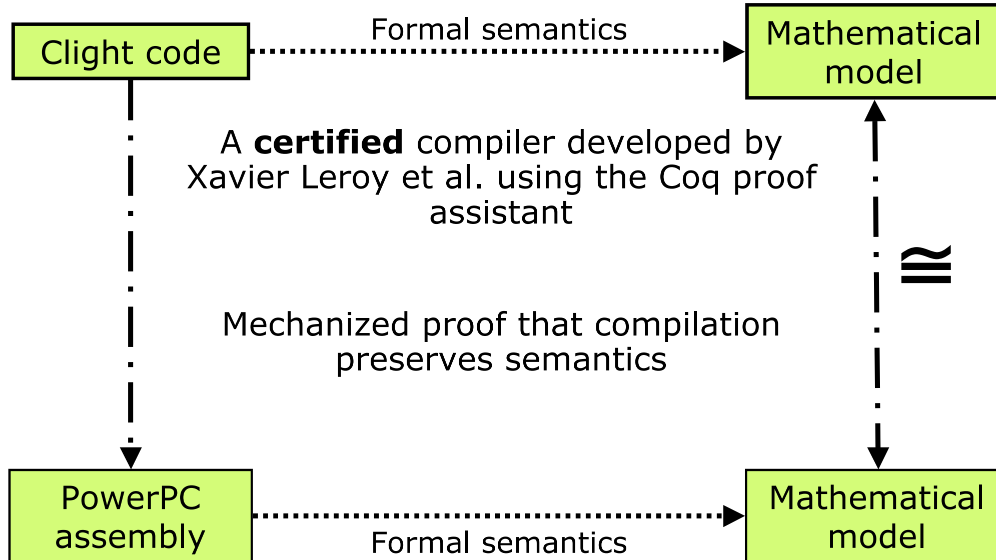
- Insist on machine-checked proofs
- Verify the actual implementation
- Amortize the cost of verification over all uses
- Engineer a re-usable framework for future verifications of similar style
- Amortize the cost of building the framework over multiple GCs
- Leverage existing work
  - INRIA (Leroy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

Portland State UNIVERSITY

Typed Garbage Collection?

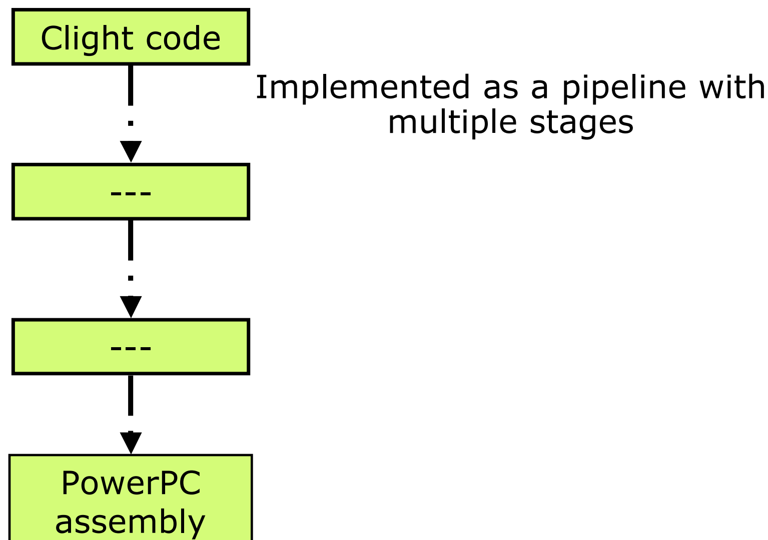
- Long-standing goal: define a type system rich enough to express a GC
  - Proposals to date are complex and only guarantee safety
- We propose a different path using general-purpose provers (e.g. Coq, Isabelle, etc.)
- Type-based approach may still be good choice for mutator

The Compcert Framework

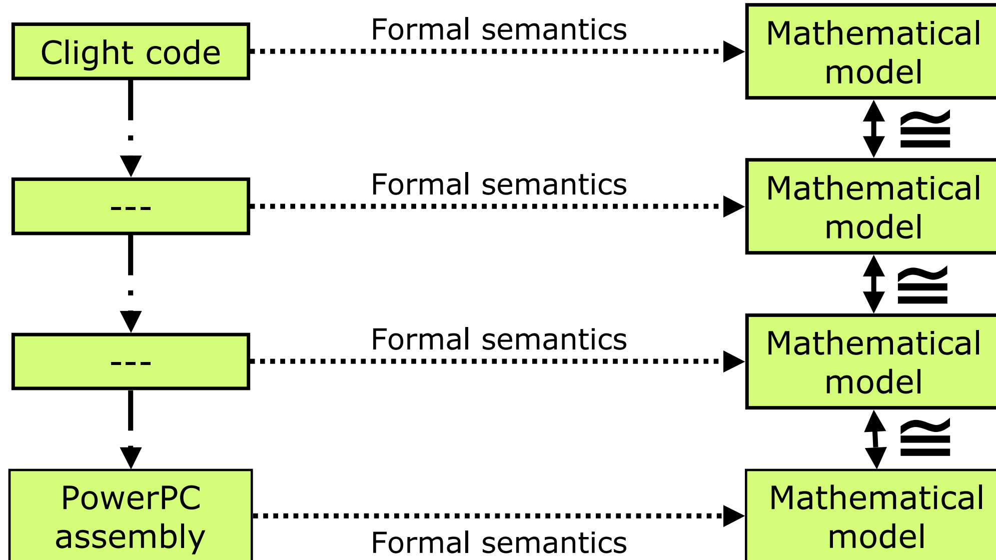


Portland State UNIVERSITY

The Compcert Framework

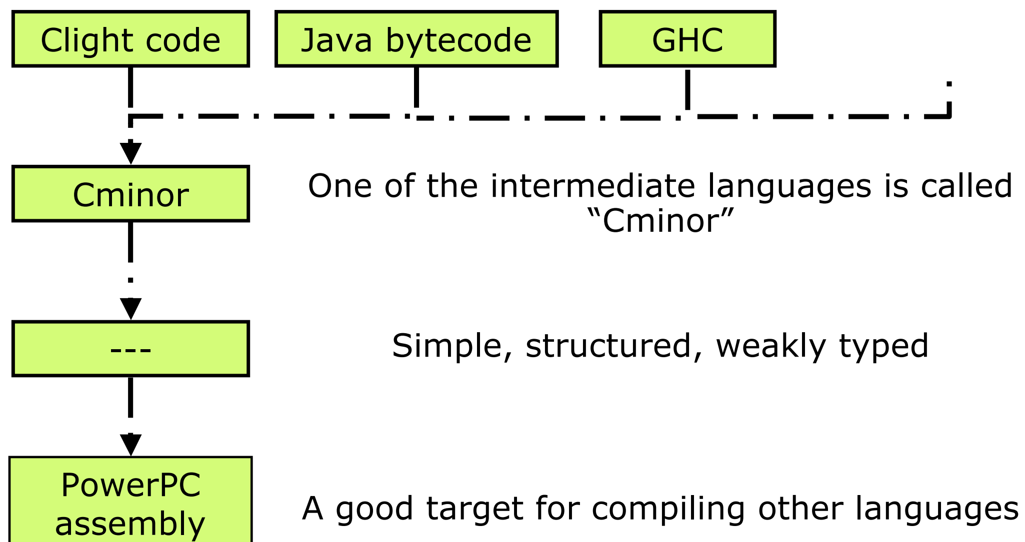


The Compcert Framework

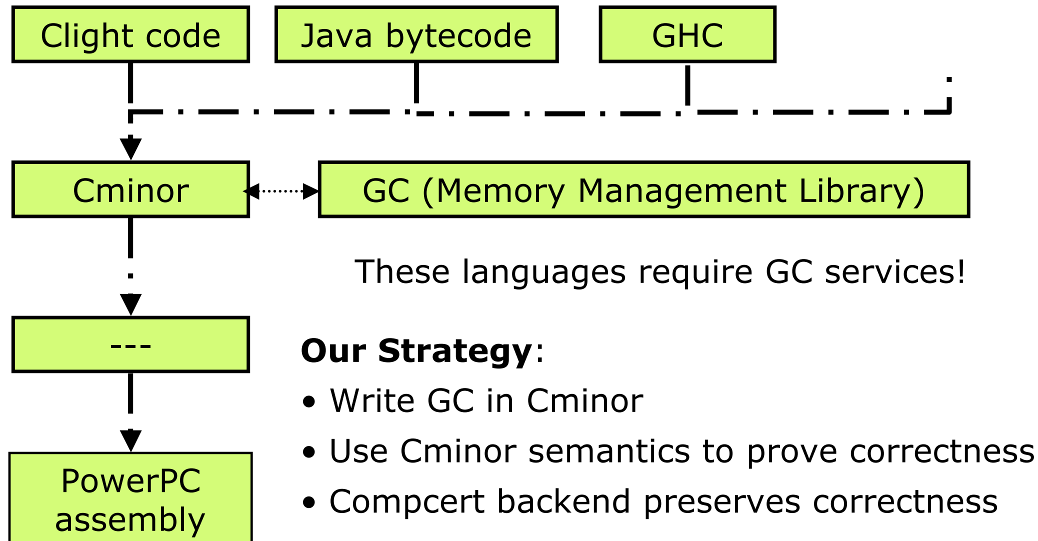


Portland State UNIVERSITY

The Compcert Framework



The Compcert Framework



Stop-and-copy 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;
}

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

    frStart = int32["toStartp"];
    toStart = int32["frStartp"];
```

Portland State UNIVERSITY

Stop-and-copy GC code (2)

```
    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;
     }
    }
}
```

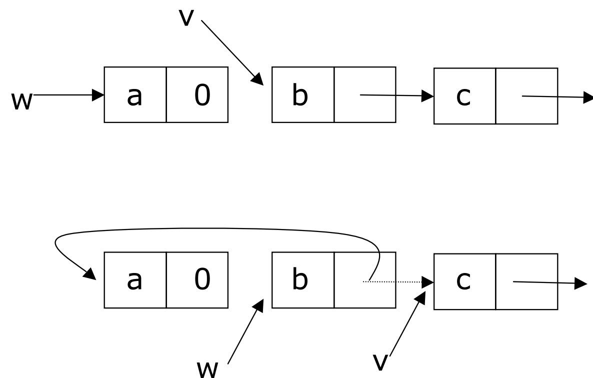
```
"cheneyAlloc"(hdr,root) : int -> int -> int
{
  var free,len;

  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);
}
```

Portland State UNIVERSITY

Example: in-place list reversal

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



Portland State University

Proving Cminor Programs

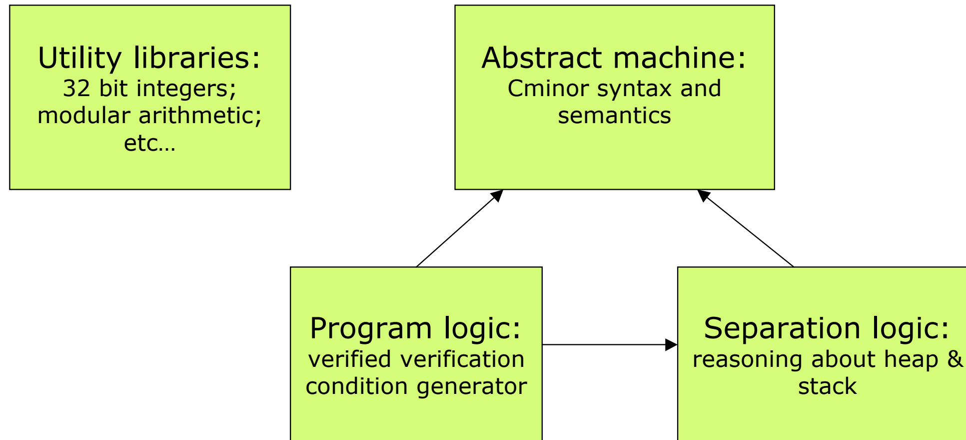
- Proving correctness of imperative pointer-based programs is a hot research topic
- (Different from Compcert's goal, which is about proving compiler correctness)
- Our current direction:
  - Construct proofs on a **deep embedding** of the Cminor semantics in Coq
  - Use **separation logic** to describe the heap

Portland State University

Shallow vs. Deep Embeddings

- In a shallow embedding, the program is represented by a term in the prover's logic
  + Can reason directly using full power of prover
  – Hard to formalize connection with actual code
- In a deep embedding, the program is represented as a syntactic object (AST):
  + Compcert gives operational semantics for this already
  – But reasoning requires lots of tactic support

Framework Overview



Everything is implemented in the Coq proof assistant

Portland State
UNIVERSITY

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.)

Portland State University

Example: Linked Lists

- Relating list values to in-memory representation:

```
Fixpoint plist (x:val) (xs:list val) :=
 match xs with
 | nil        => !(x = null_ptr)
 | (x'::ys) =>
     !(x = x') * lexists v, lexists y,
                    (x ↦ v) * ((x+4) ↦ y) * plist y ys
end.
```

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

Portland State University

Separation Logic Implementation

- Definitions and properties of predicates
  - Commutativity, associativity, etc.

- Tactics are critical:
  - Simplification
  - Rearranging
  - Matching

Portland State UNIVERSITY

Program Logic

- Hoare-style reasoning using pre- and post-conditions
- Similar to prog logic of [Appel Blazy 07]
- Verified verification condition generation
  - Generator calculates a verification condition (VC) for each statement
  - Resulting semantics consistent with original Cminor semantics

Portland State UNIVERSITY

Verification Conditions

precondition of next statement

initial state

- Example: vc (x := e) Q s
  $= \exists$ v. s $\vdash$ e $\rightarrow$ 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

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)).
```

Portland State
UNIVERSITY

Proof Details:

- Main proof: ~ 45 lines
- Comparable length to our proof of the same result using a shallow-embedding of Cminor semantics
- 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

Utility libraries: ~3,300

Abstract machine: ~5,750

Program logic: ~1,550

Separation logic ~4,100

Garbage Collector: 5,000

Portland State UNIVERSITY

Stop-and-copy GC Proof

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

Definition

```
#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);
}
```

Key Points

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

Conclusions

- Assurance of programs written in high-level languages requires assurance of underlying run-time systems
- 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.