Towards High-Assurance Run-Time Systems

Andrew Tolmach
Andrew McCreight
Tim Chevalier

High-Assurance Systems Programming Project
Portland State University

The Context

- Safety-critical and security-critical software systems cost too much
  - software for certified fielded systems
  - software for the tools used to build certified systems
- Current norm: code in low-level languages
- Certification by inspection doesn't scale
- We need high assurance by construction

Better Languages to the Rescue?

- High-level languages like Java or Haskell prevent many classes of bugs
  - Strong static typing prevents pointer forging
  - Garbage-collected memory prevents "dangling pointer" dereferences
  - Array bounds checking prevents buffer overflow bugs and attacks
- Development is faster and easier too
- Performance is adequate for tools (at least)

A credibility gap

- These safety properties may hold for source programs, but…
- Languages have big compilers and large, complex run-time systems
  - Glasgow Haskell Compiler RTS: 50k+ lines of C
  - Java HotSpot Compiler RTS: 100k+ lines of C++
- Post-hoc certification isn't plausible for all this infrastructure

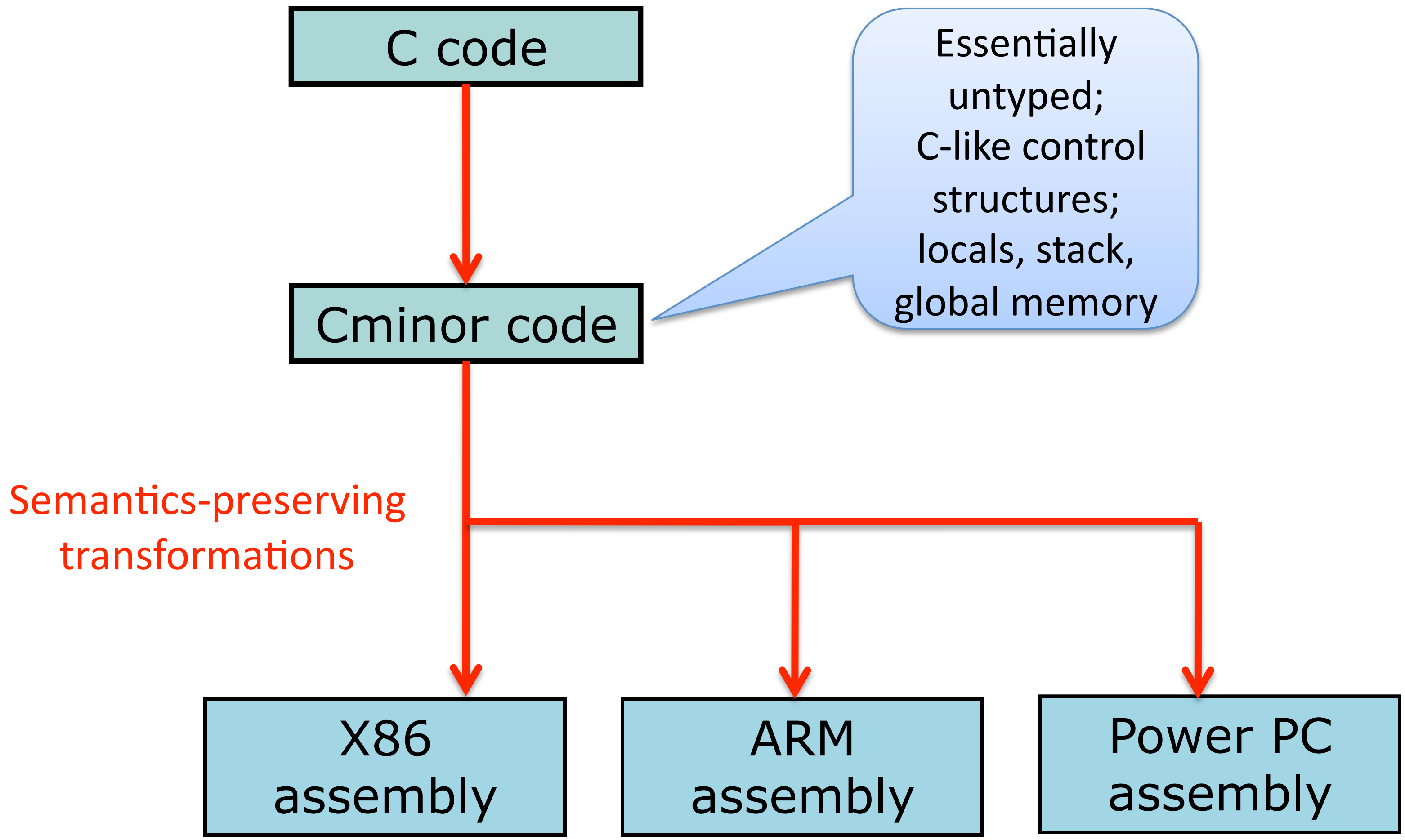
High Assurance Run-Time System

- Designed from scratch using principles for assurance: minimality, simplicity, modularity, mechanized verification
- Goal: credible implementations using scalable assurance techniques
- Essential RTS services:
  - Garbage collection
  - Interfacing to untrusted languages
  - Concurrency

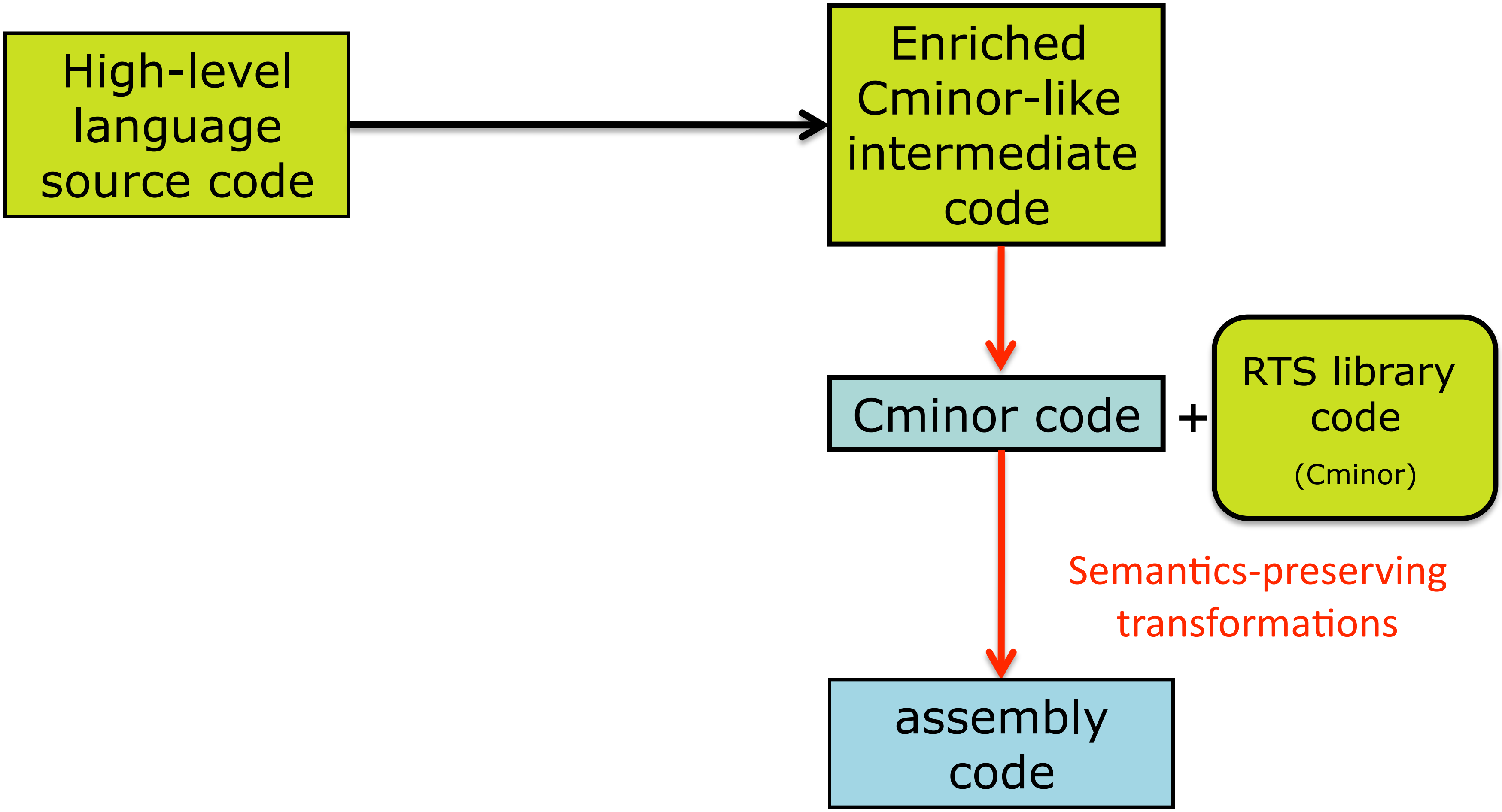
Language-based approach

- Use compiler intermediate languages to package RTS services
- Language formal semantics specify intended behavior of services <u>and</u> clients
- Use semantics-preserving compilation to guarantee behavior of RTS implementation
- Use type systems selectively to help guarantee that client code is well-behaved

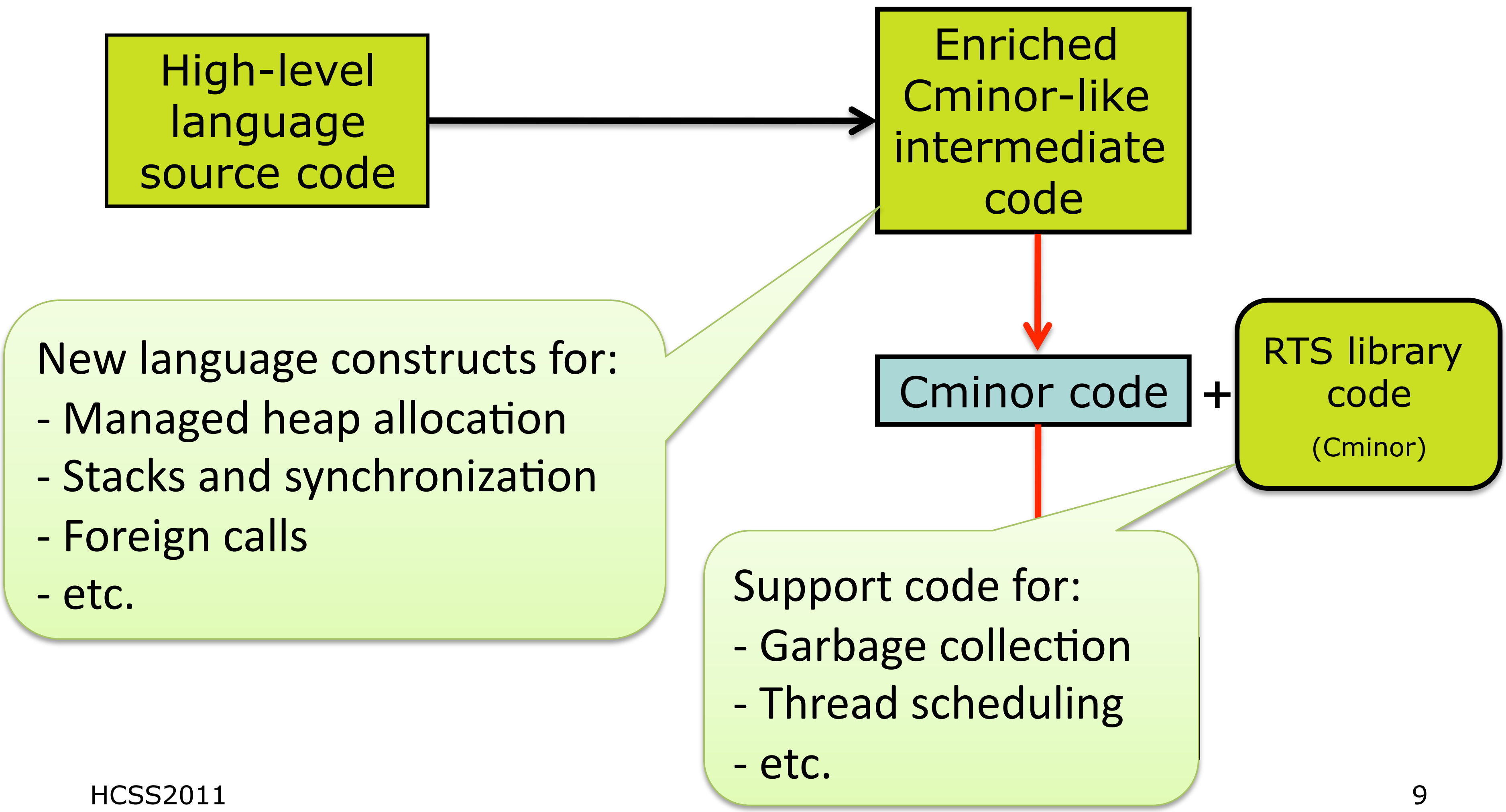
CompCert Architecture



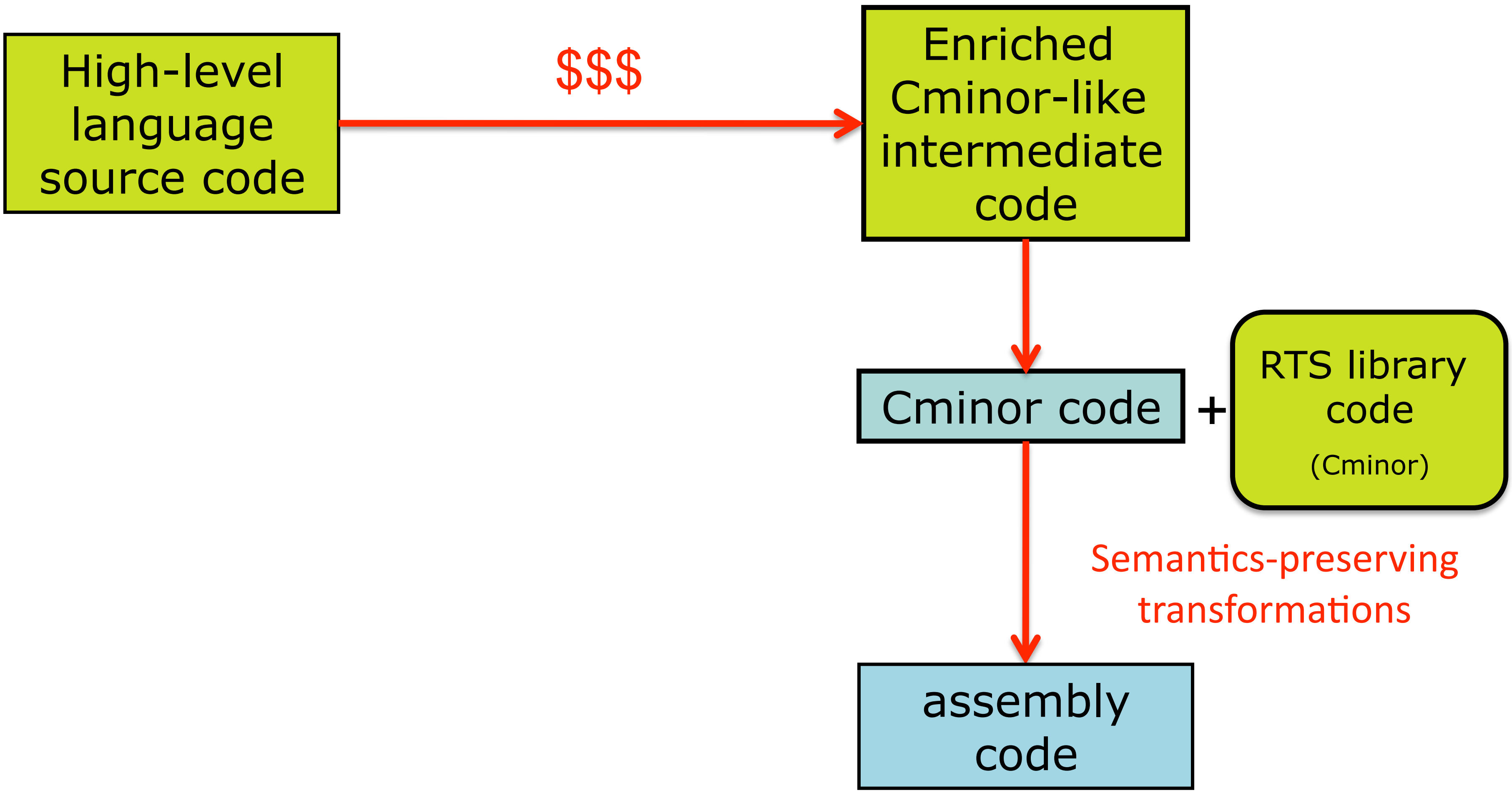
CompCert-based RTS strategy



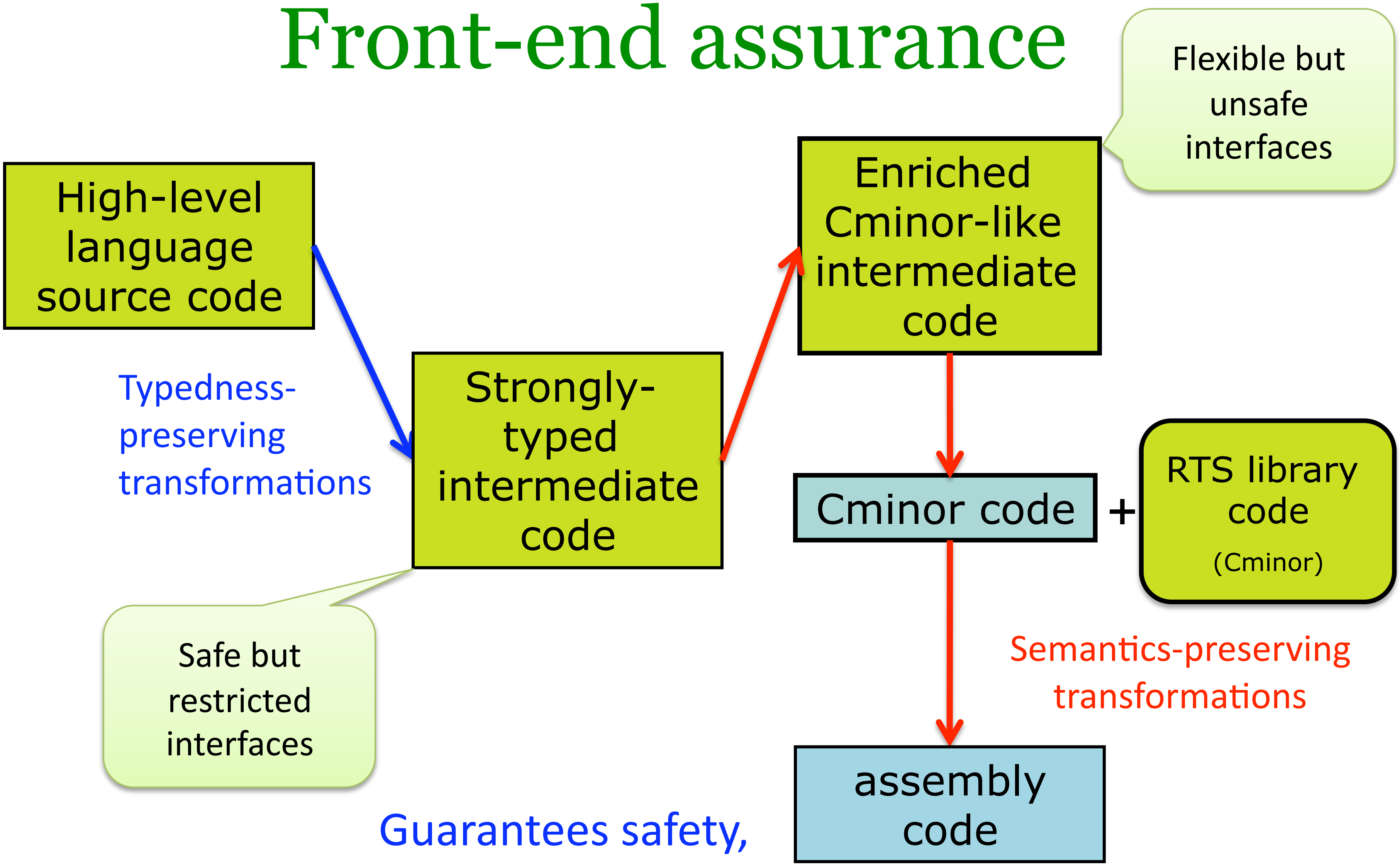
CompCert-based RTS strategy



Front-end assurance



Front-end assurance

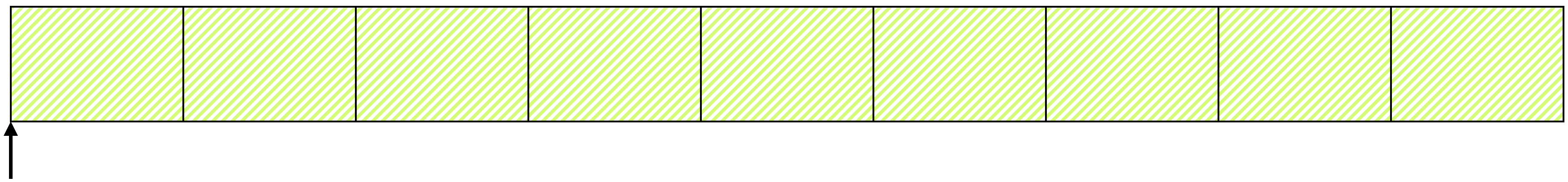


Guarantees safety, not full behavior

Garbage Collection

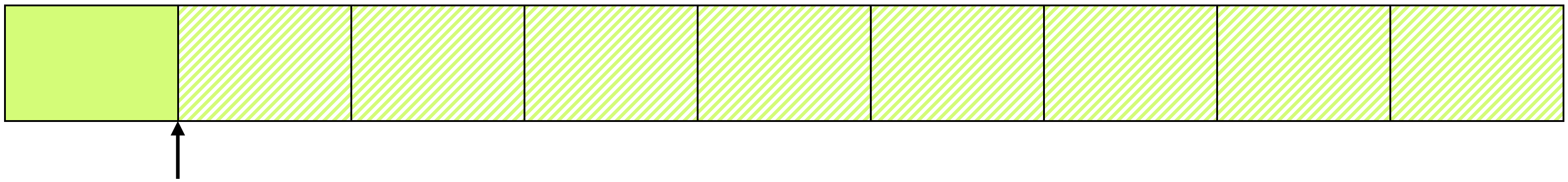
- A mechanism for reclaiming and reusing unused memory automatically
- Programmer never frees memory by hand:
  - Memory never freed too early, so no "dangling pointer" bugs
  - Unreachable memory always freed, so no coding-induced space leaks
- Many different algorithms:
  - Mark-sweep, Stop-and-copy, etc.

Stop-and-copy Garbage Collection



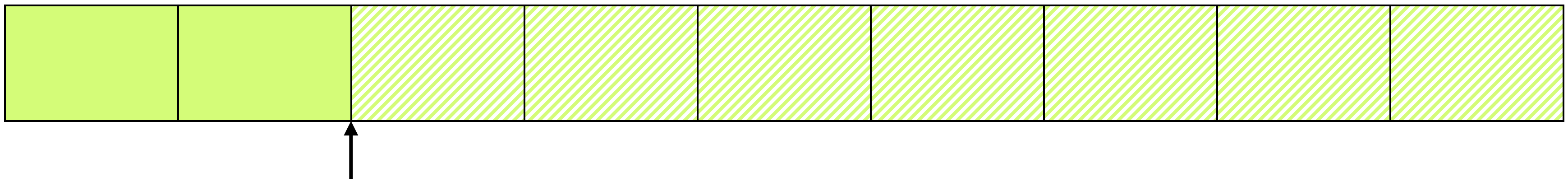
The application program (the "mutator") allocates objects from a contiguous memory "heap"

Stop-and-copy Garbage Collection



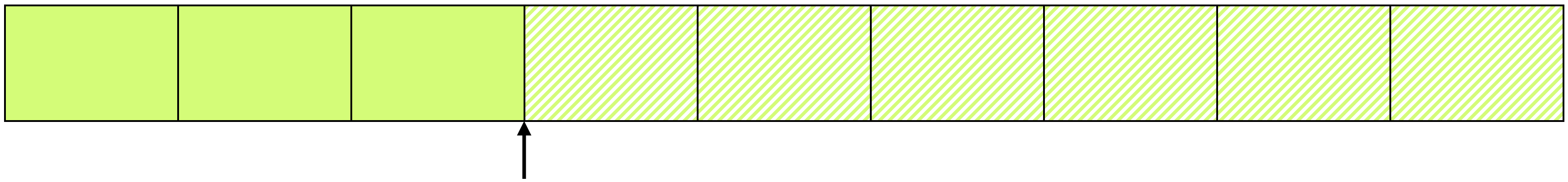
Allocating an object

Stop-and-copy Garbage Collection



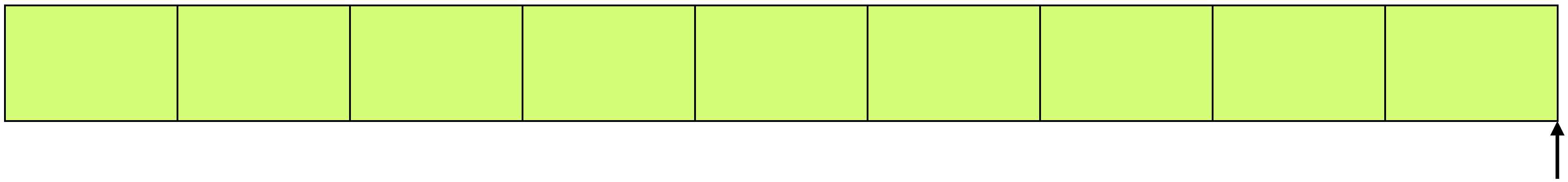
Allocating another object

Stop-and-copy Garbage Collection



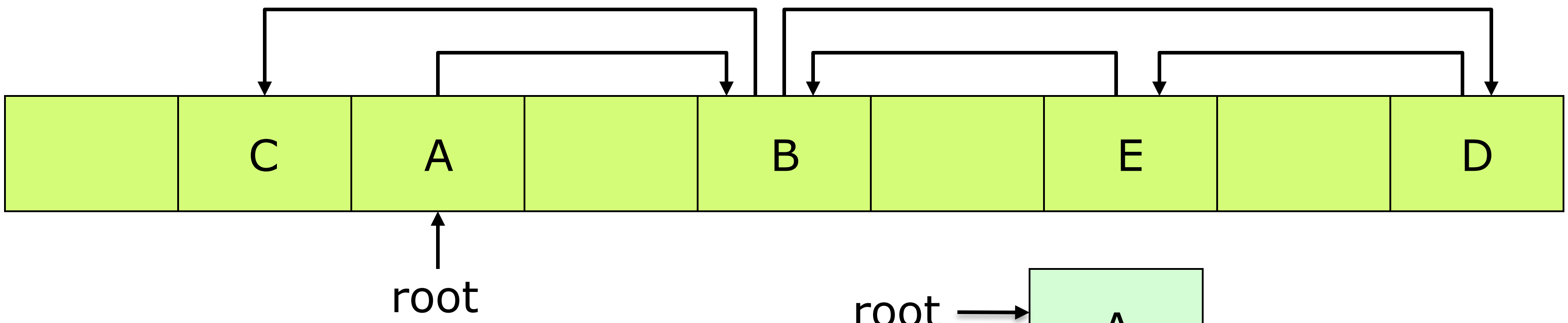
Allocating another object

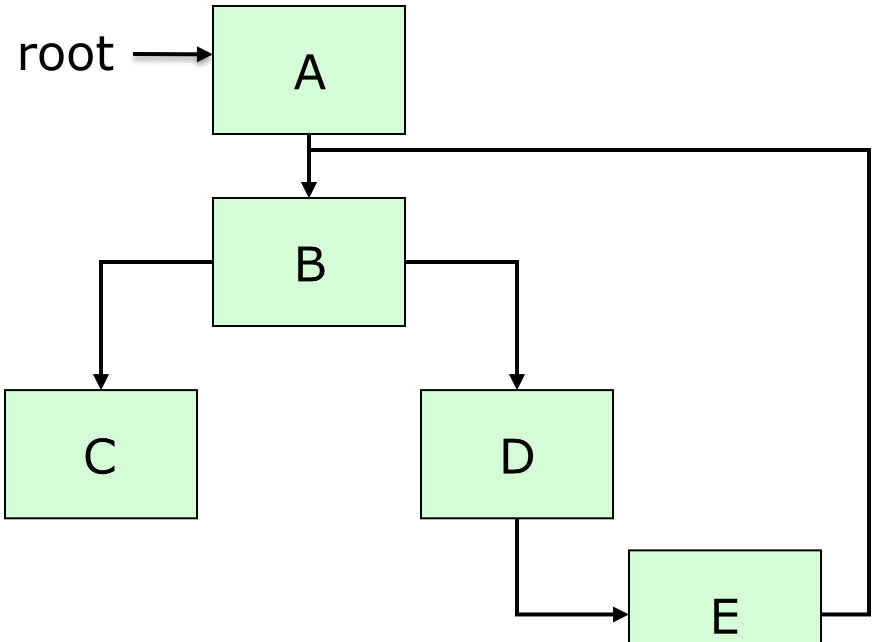
Stop-and-copy Garbage Collection



Eventually, the heap is full of objects!

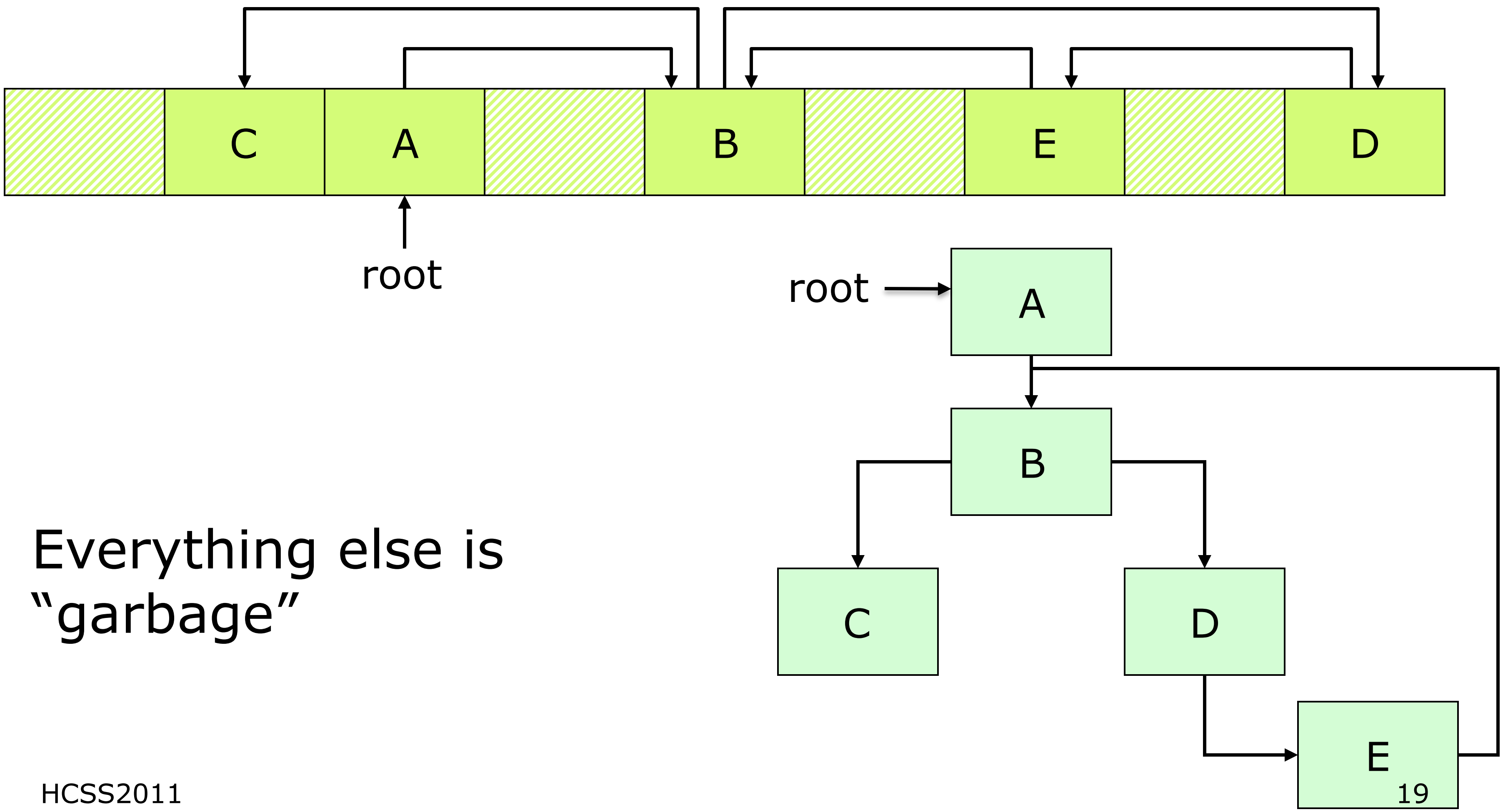
Stop-and-copy Garbage Collection





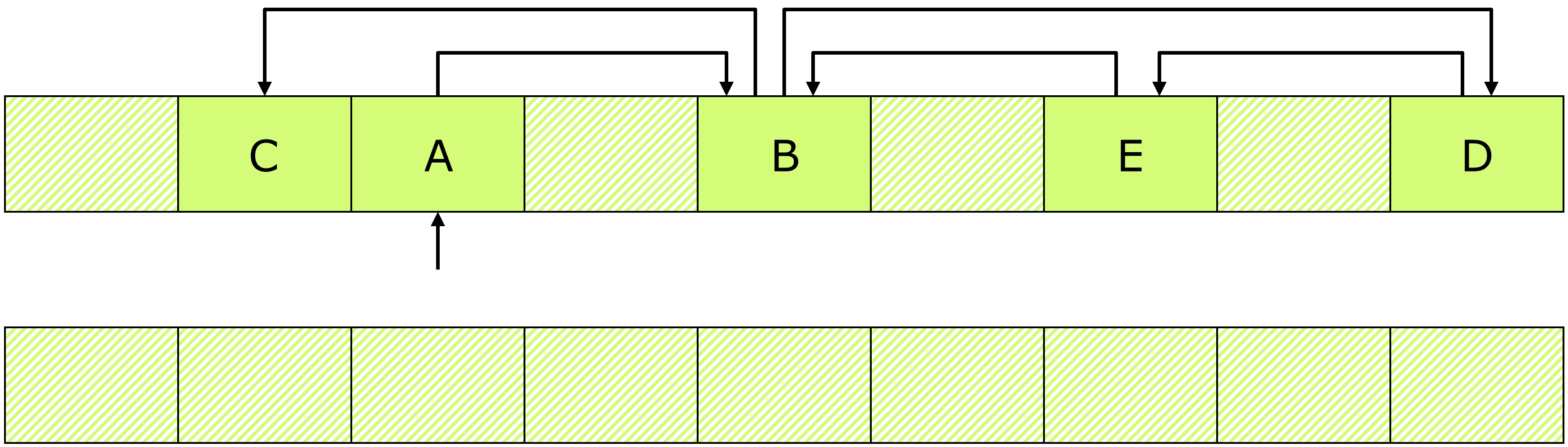
But only some of the objects (the "live" data) are reachable from the mutator's pointers (the "roots")

Stop-and-copy Garbage Collection



Everything else is "garbage"

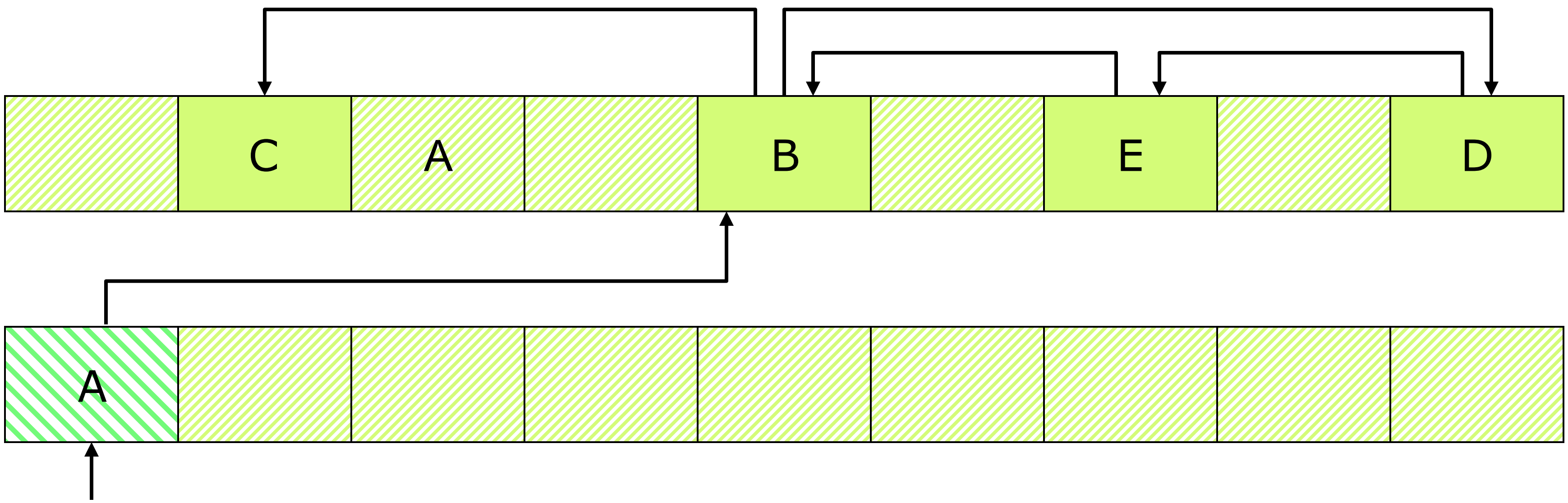
Stop-and-copy Garbage Collection



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

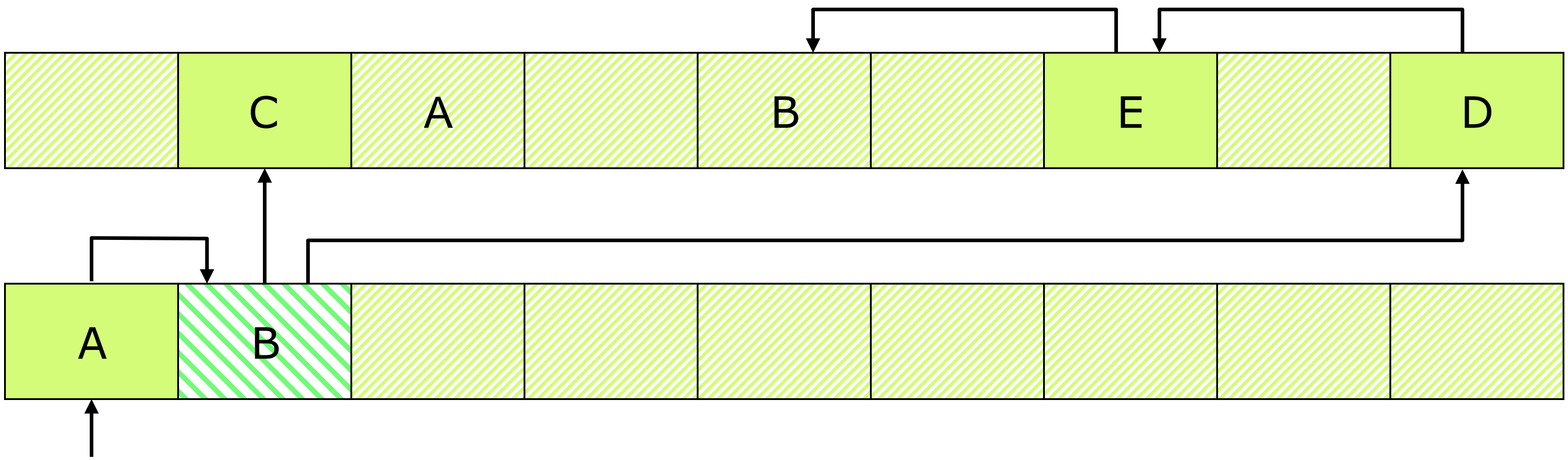
(Algorithm due to Cheney, 1970)

Stop-and-copy Garbage Collection



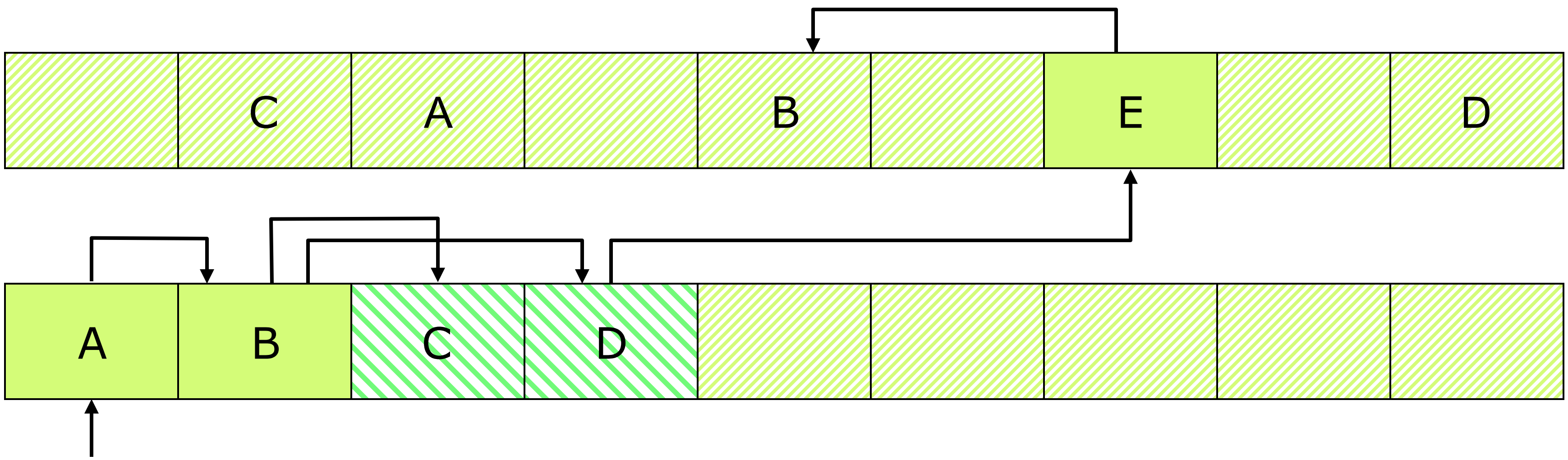
Copy root A into the new heap

Stop-and-copy Garbage Collection



Scavenge A (copy B into the new heap)

Stop-and-copy Garbage Collection

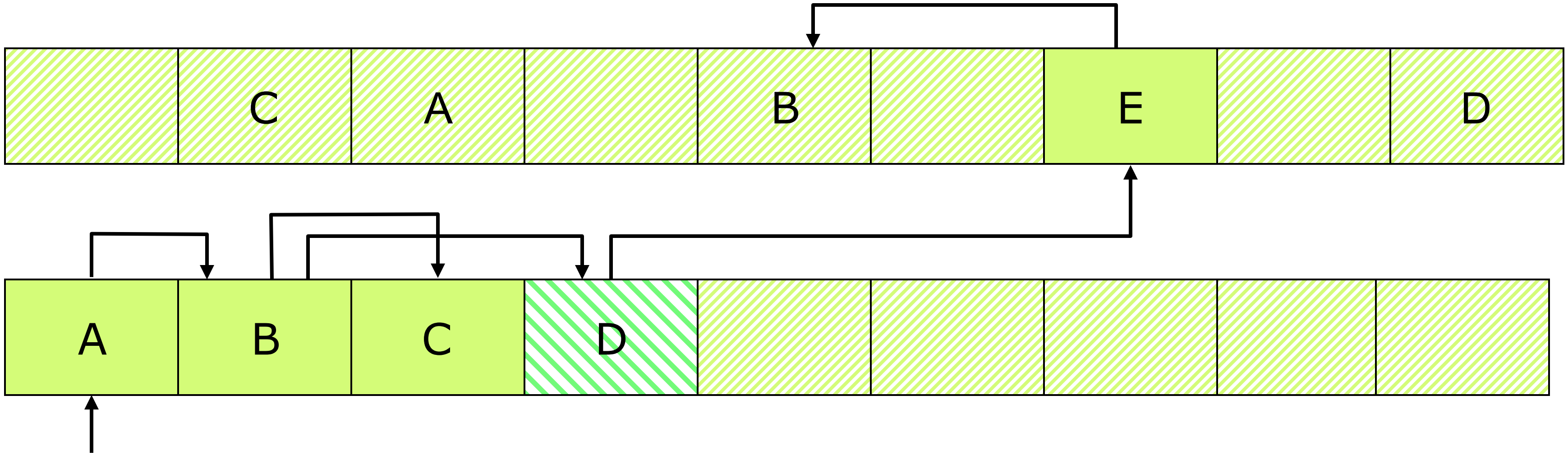


Scavenge B (copy C and D into the new heap)

Hasp Project

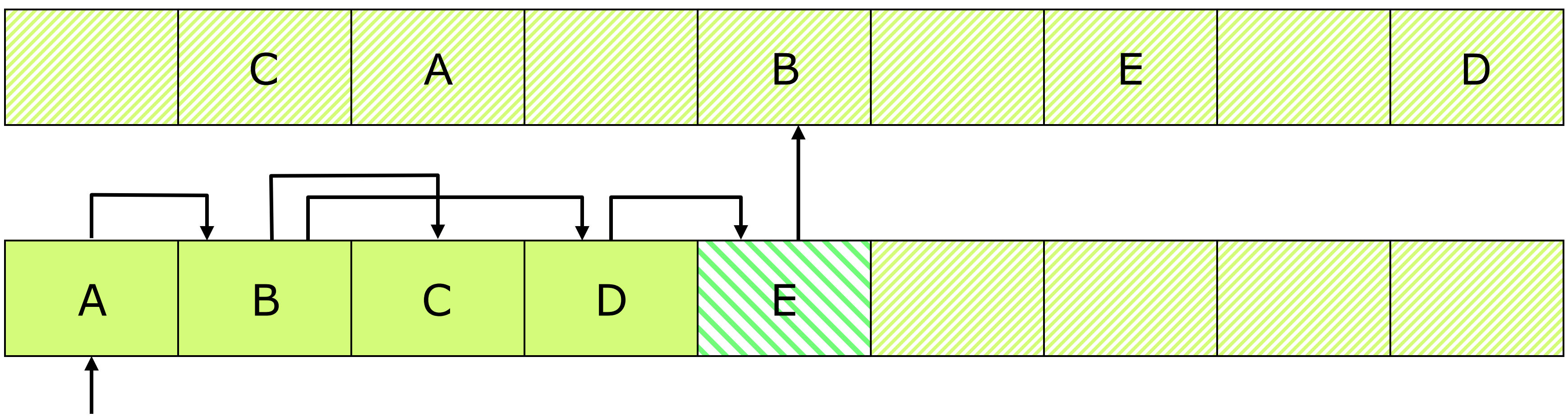
Portland State University

Stop-and-copy Garbage Collection



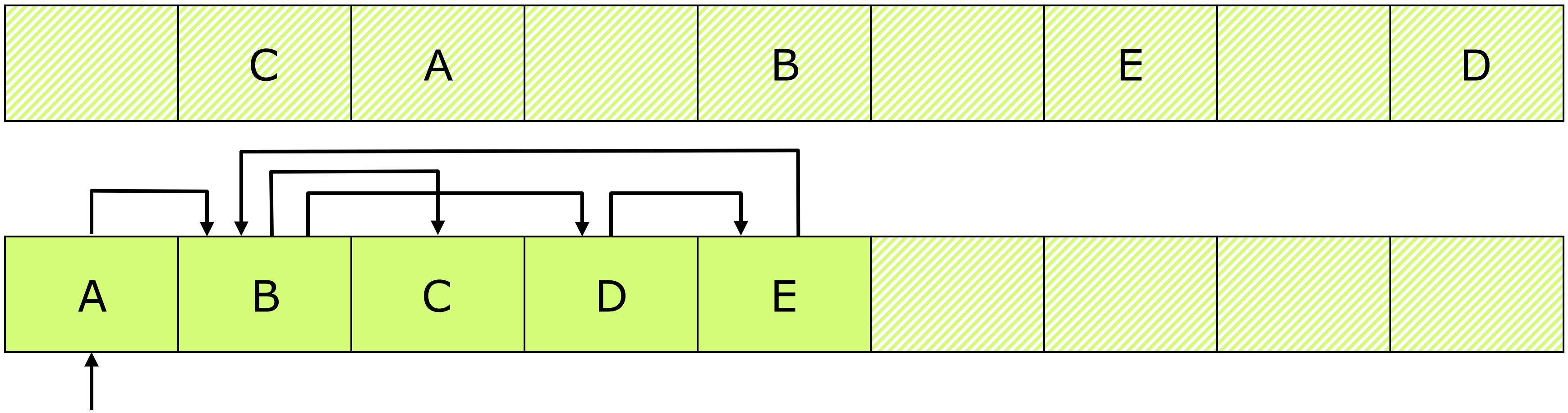
Scavenge C (no objects copied)

Stop-and-copy Garbage Collection



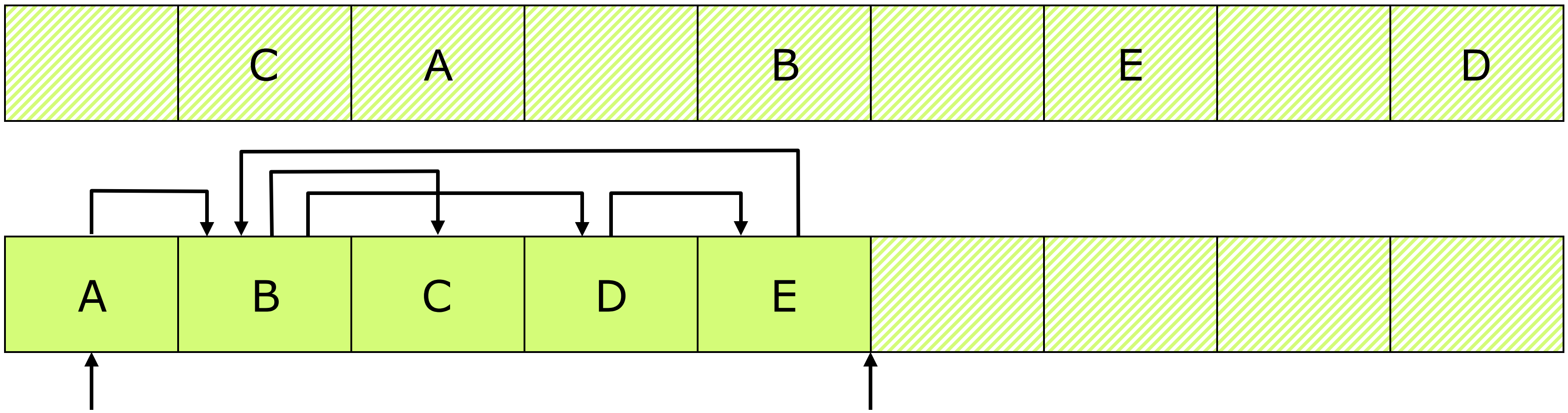
Scavenge D (copy E into the new heap)

Stop-and-copy Garbage Collection



Scavenge E (B is already in the new heap)

Stop-and-copy Garbage Collection



- All live data has been copied to the new heap;
- Structure of the original live data graph has been preserved;
- Unused memory is now contiguous.

Garbage Collectors do have bugs!

- **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

Mozilla **Firefox** Javascript **Garbage Collector Vulnerability**
18 Apr 2008 ... TITLE: Mozilla **Firefox** Javascript **Garbage Collector Vulnerability** SECUNIA ADVISORY ID: SA29787 VERIFY ADVISORY: ...
www.windowsbbs.com › ... › Firefox, Thunderbird & SeaMonkey - Cached

MFSA 2010-25: Re-use of freed object due to scope confusion
1 Apr 2010 ... If **garbage collection** could be triggered at the right time then **Firefox** would later use this freed object. The contest winning exploit only ...
www.mozilla.org/security/announce/2010/mfsa2010-25.html - Cached

Mozilla Foundation Security Advisories
MFSA 2009-08 Mozilla **Firefox** XUL Linked Clones Double Free **Vulnerability** .... MFSA 2006-10 JavaScript **garbage-collection** hazard audit ...
www.mozilla.org/security/announce/ - Cached - Similar
Show more results from www.mozilla.org

RISK - SANS: @RISK: The Consensus Security **Vulnerability** Alert
... 08.17.21 - Mozilla **Firefox**/SeaMonkey JavaScript **Garbage Collector** Memory Corruption .... This control contains remote code execution **vulnerability**. ...
www.sans.org/newsletters/risk/display.php?v=7&i=17 - Cached - Similar

How can we rule out GC bugs?

- Show correctness of GC algorithm and its implementation

Our previously reported work

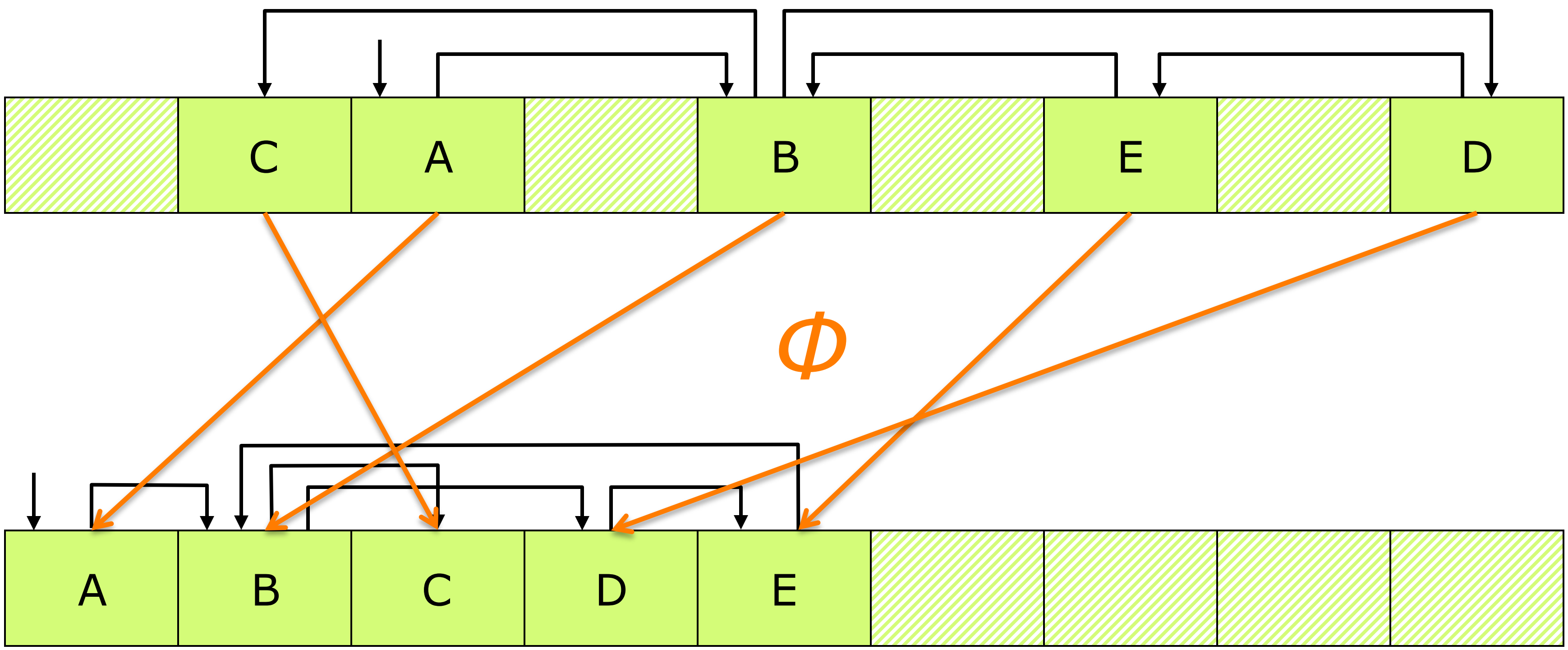
- Show that mutator and collector are correctly integrated:
  - agree about the set of roots and the locations of pointers within objects
  - respect each others' private data structures

Copying Collector Proof

- Have a proof for a simple Cheney-style copying collector implemented in CompCert's Cminor language
- Collector specification is written in separation logic
- Proof relies on reusable tactics and libraries for separation logic reasoning in Coq [McCreight TPHOLS09]
- Comparable to other recent collector proofs

Collector library code
(Cminor)

Cheney collector proof



- Demonstrating isomorphism Φ between old and new object graphs is the key to proving correctness of the GC

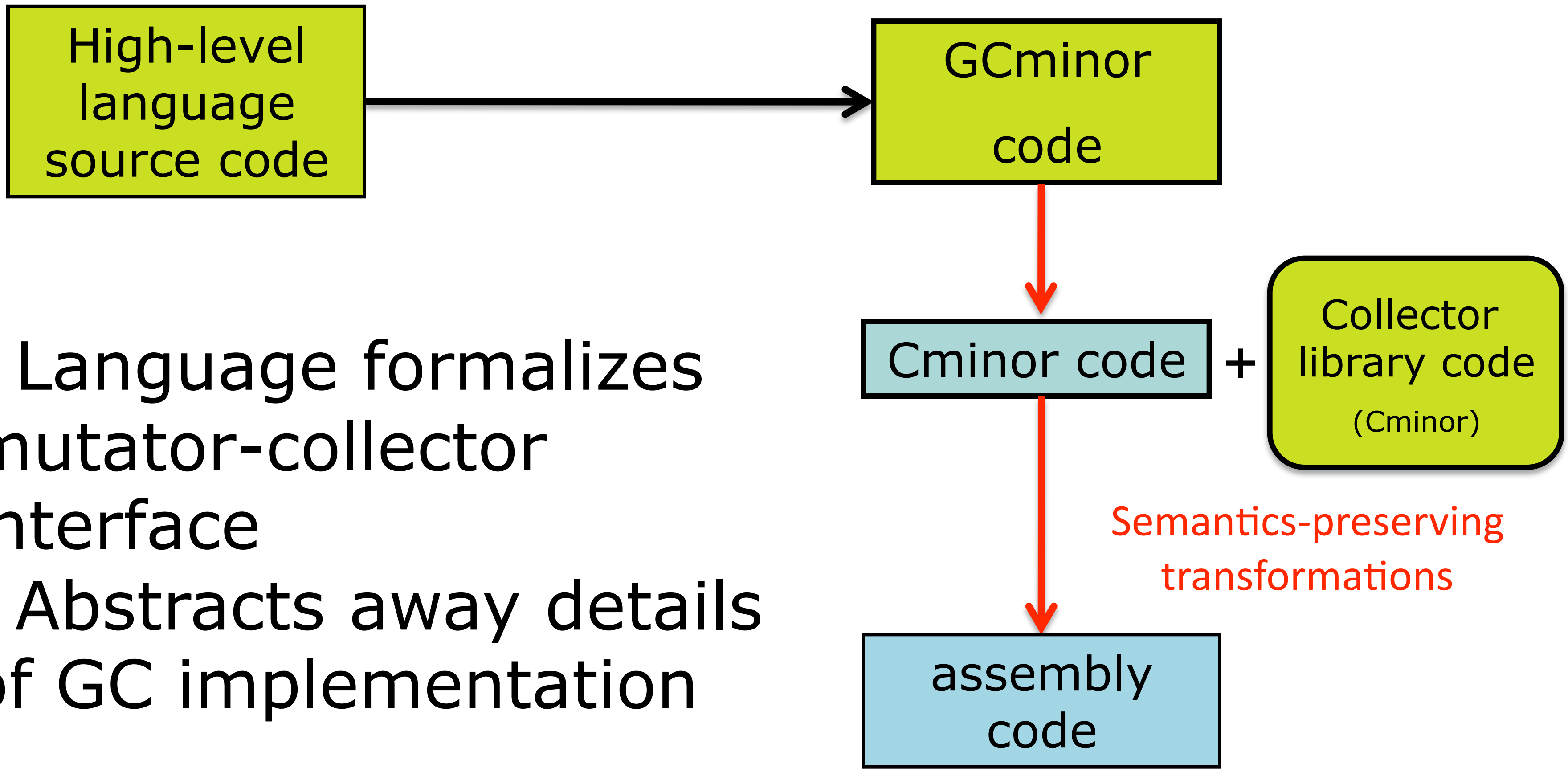
How can we rule out GC bugs?

- Show correctness of GC algorithm and its implementation

Focus of remainder of talk

- Show that mutator and collector are correctly integrated:
  – agree about the set of roots and the locations of pointers within objects
  – respect each others' private data structures

GCminor



- Language formalizes mutator-collector interface
- Abstracts away details of GC implementation

GCminor

- Extends Cminor language with
  - `alloc` primitive to obtain fresh heap objects
    - implicitly invokes GC if necessary
    - contents of objects must be initialized explicitly
  - declarations of GC roots
    - specify which variables contain useful heap pointers
- Object layouts are specified separately as functions
  - size : header → object size
  - isPtr : header → offset → bool

GCminor semantics

- As for existing CompCert languages, GCminor is given a small-step operational semantics
- Each rule describes a valid program step, its impact on the program state, and any externally visible effects

$\sigma, S \xrightarrow{t} \sigma'$

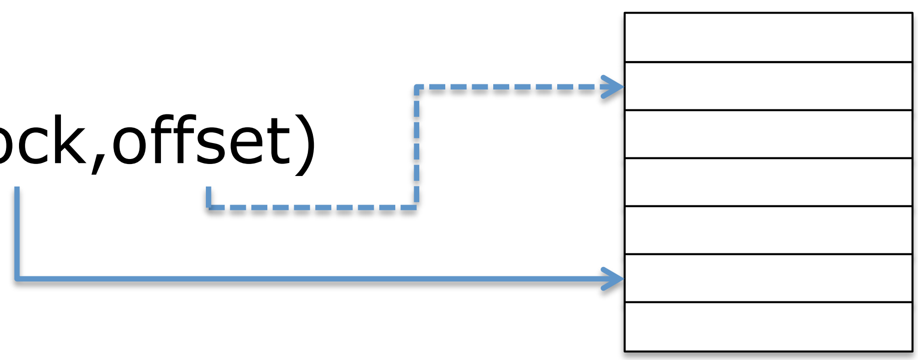
statement S

state σ = heap + local variables + stack + …

trace t = system calls + ...

Values and memory in CompCert

- CompCert semantics uses a simple block-based memory model at all stages in compiler pipeline
  - A block can represent a global data area, a stack frame, a single memory-allocated variable, etc.
- Values in the program state can be
  - integers VInt(n)
  - pointers VPtr(block,offset)

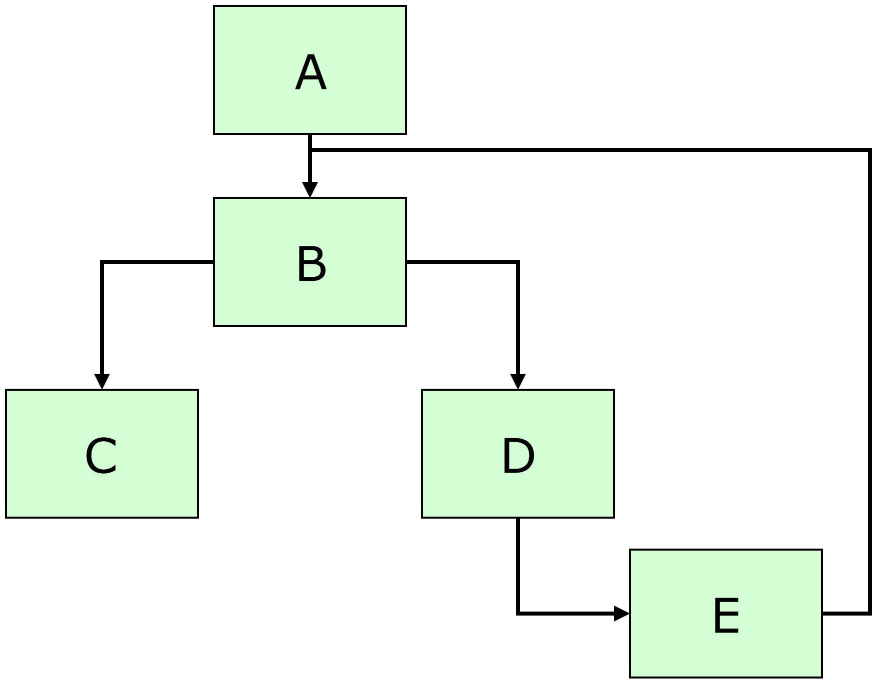


Specifying well-behaved programs

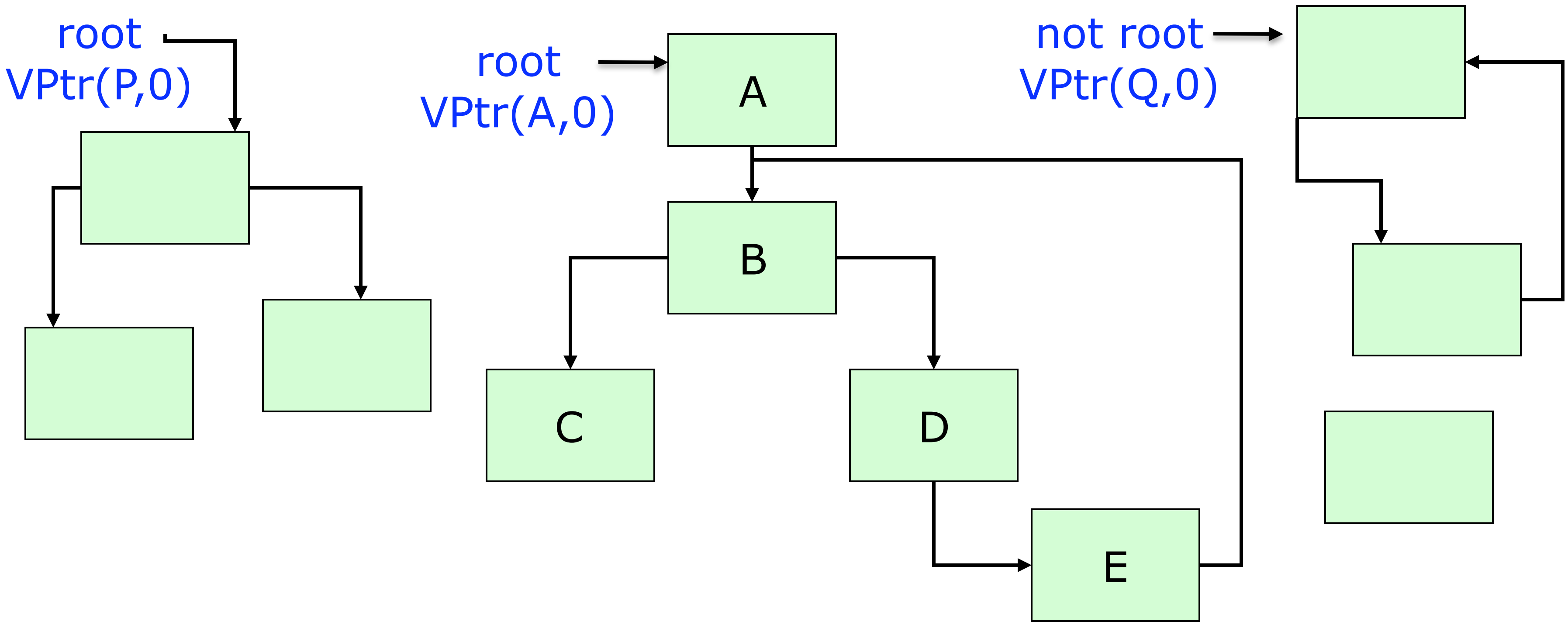
- If no stepping rule applies in a given state, the program is stuck
  - corresponds to an unchecked runtime error
- Example: trying to load memory using a VInt value as if it were a pointer
  - characterizes code that forges pointers
- Well-behaved programs are those that don't get stuck
  - Semantic preservation theorem only applies to these; "Garbage in, garbage out"

GCminor memory semantics

- Each `alloc` creates a fresh separate block
- Heap blocks appear never to go away and never to move!

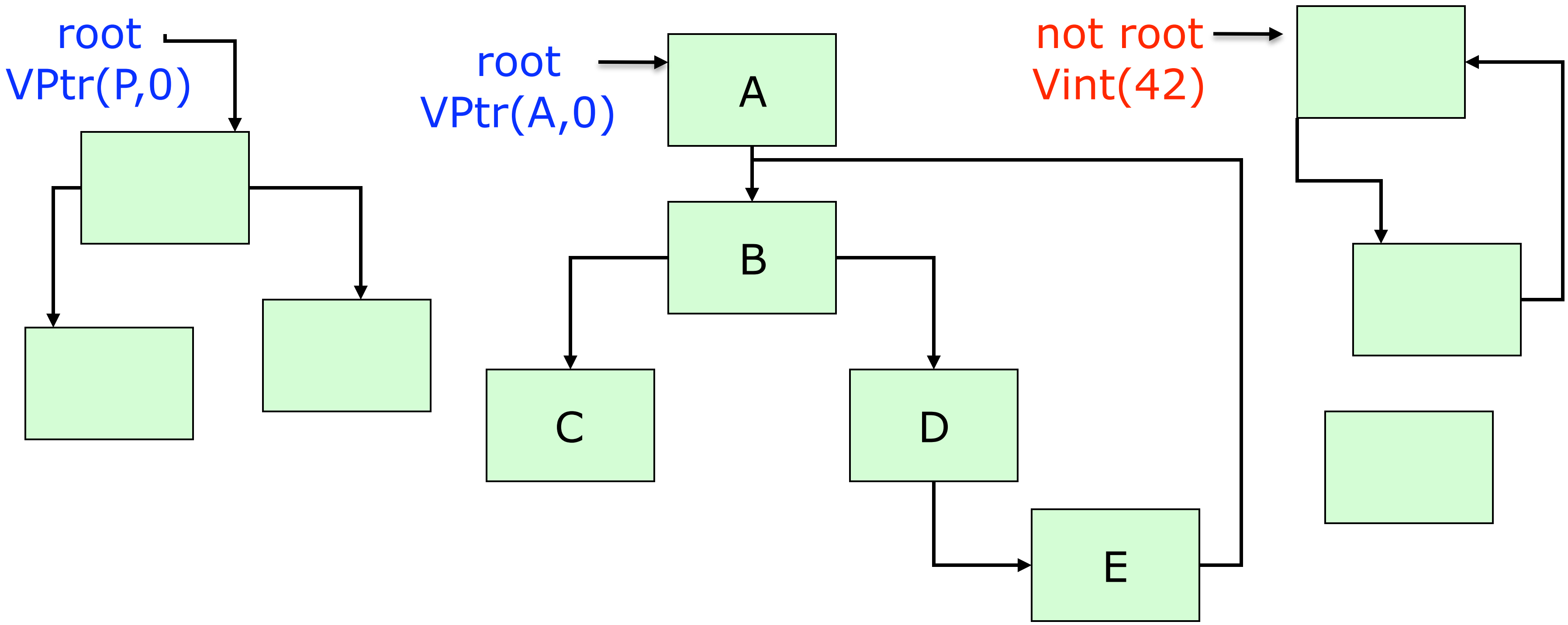


Semantics of root declarations



- Whenever GC might occur, pointers not declared as roots appear to be invalidated

Semantics of root declarations



- Whenever GC might occur, pointers not declared as roots appear to be invalidated
- Any subsequent load attempt will fail

Additional Mutator Specifications

- Semantics is parameterized by a nominal heap size: program gets stuck if live data size exceeds this heap size
- Program also gets stuck if mutator doesn’t initialize object properly before next allocation point

Precise but Flexible Specification

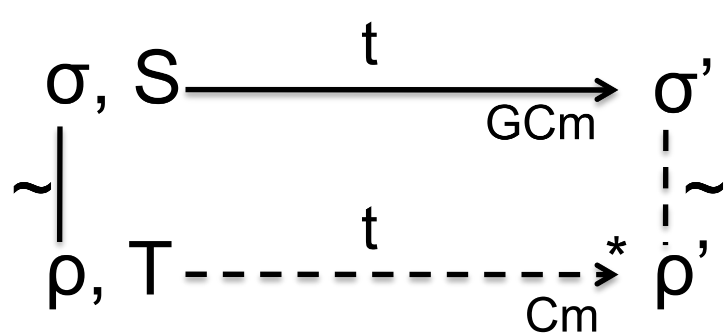
- GCminor semantics forms a specification of how the mutator and GC should interact
  - Non-stuck GCminor programs are well-behaved mutators
  - Any correct implementation of GCminor semantics embodies a well-behaved collector
- Not tied to any particular GC mechanism
  - should work for copying, mark-sweep, and generational collectors

GCminor implementation

- Translate GCminor programs to Cminor; then link in fixed GC library
  - Currently use our simple proven Cheney GC
- Heap = single large global array
- `alloc` primitive becomes library call
- Save and restore live root variables
  - at every function call and allocation site
  - allows GC to scan and update roots
  - "shadow stack" avoids need to change CompCert backend

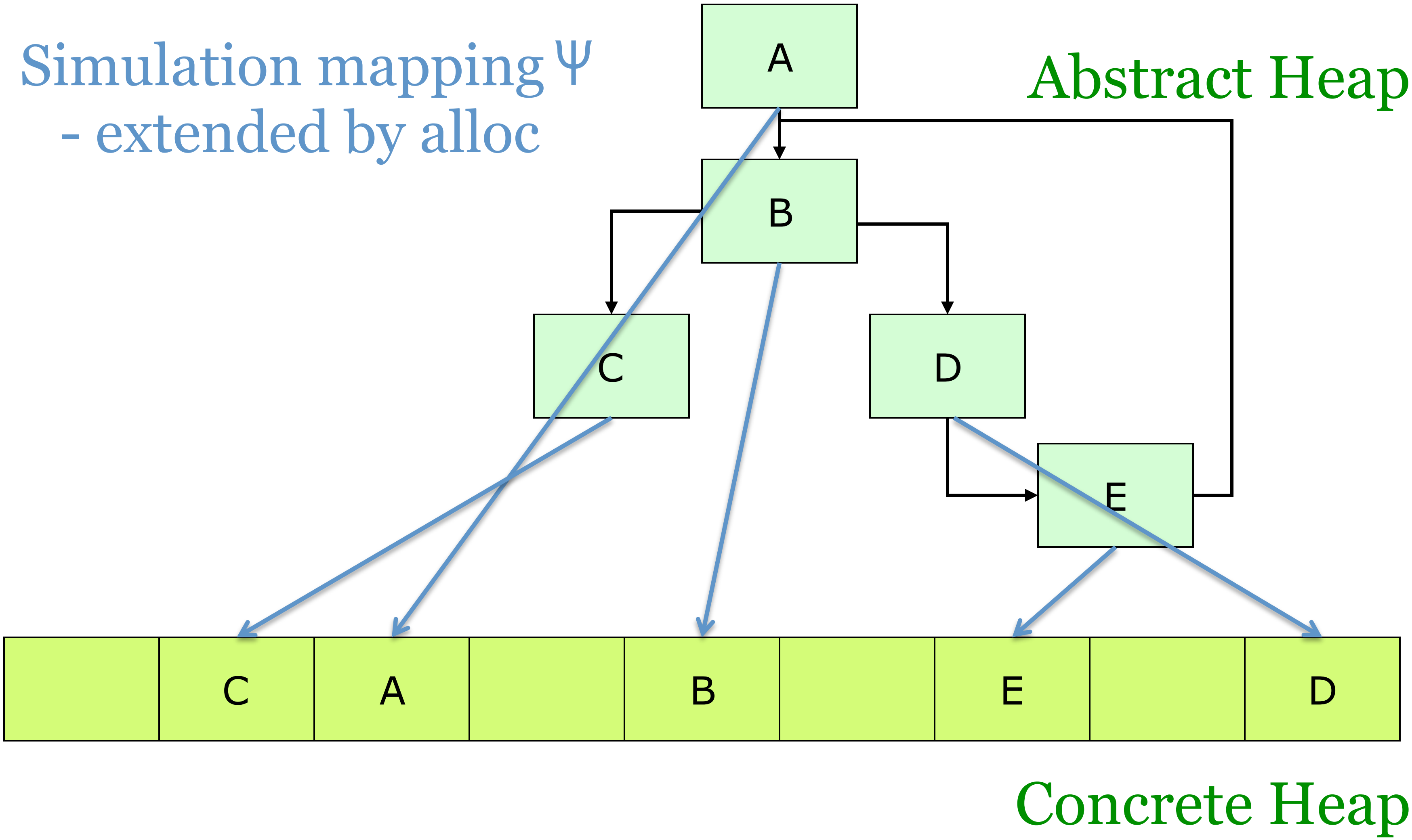
Preservation Lemma

- We define a simulation relation
  - GCminor state σ ~ Cminor state ρ
  - Maps abstract heap to concrete heap and root variables to shadow stack
- Key lemma:



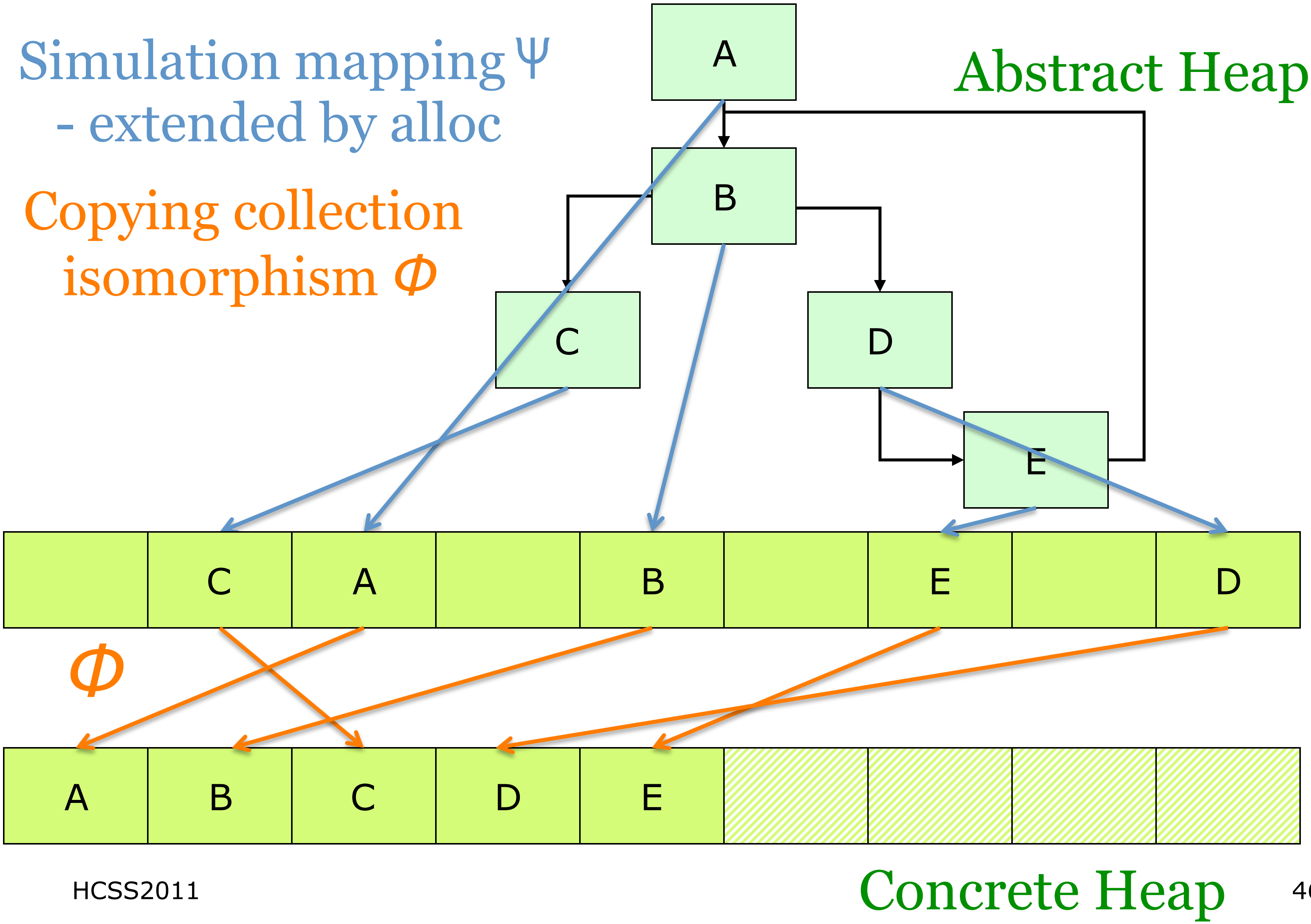
where T_{Cm}= translation of S_{GCm}

Simulation mapping Ψ - extended by alloc



Simulation mapping Ψ
- extended by alloc

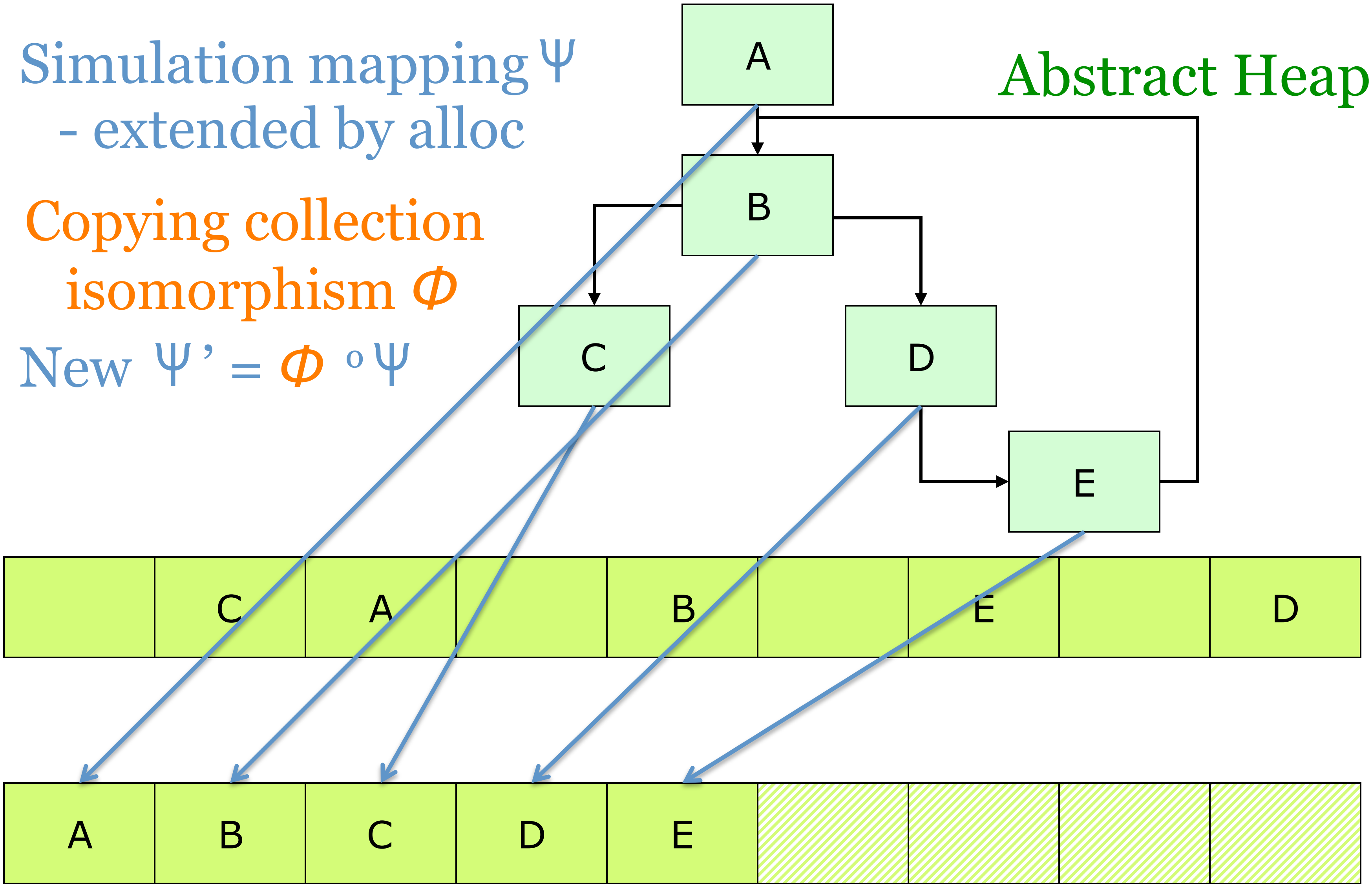
Copying collection isomorphism Φ



Simulation mapping Ψ
- extended by alloc

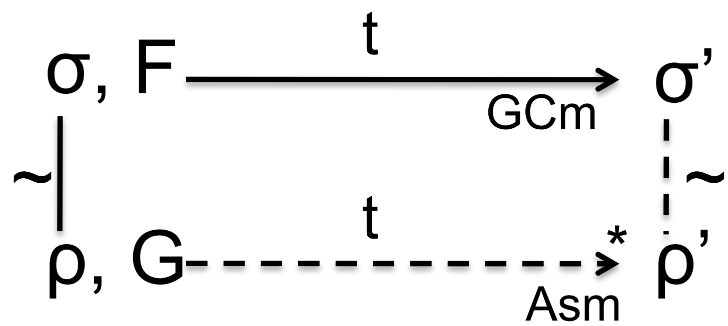
Copying collection
isomorphism *Φ*

New Ψ’ = *Φ* ∘ Ψ



Overall Semantics Preservation

- Theorem:



where G_{Asm}= final translation of function F_{GCm}

Pf: Iterate Lemma + existing CompCert pfs

- Corollary: If program P_{GCm} does not get stuck, then neither does translated program Q_{Asm} and P & Q behave the same

  Pf: Iterate Thm + determinacy of Asm

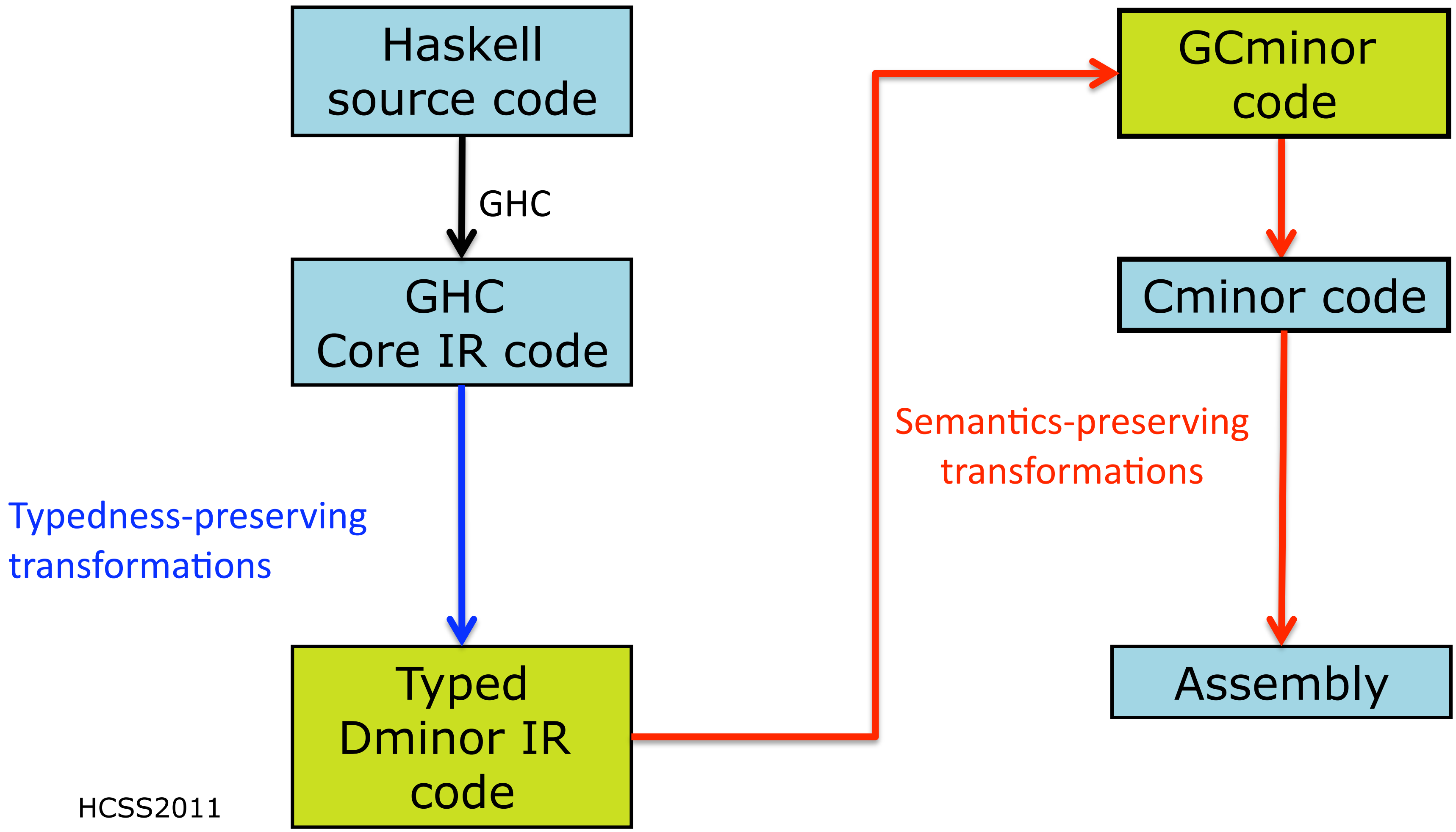
Assessing the Semantics

- We get completeness of the GC as well as soundness…
- …but only for programs that obey a maximum live memory bound
- More generally, front ends need to guarantee that GCminor code doesn't get stuck…
- … type systems can help
- We get guarantees only for observable behavior of whole programs

Case Study : Haskell front end

- Proof-of-concept that exercises GCminor
- Feedback on interface design and performance for client
- Built on Glasgow Haskell Compiler: real source language
- Limited set of primitives
  - no foreign functions, exceptions, concurrency
  - compiles good part of std. benchmark suite
- Modest expectations for performance

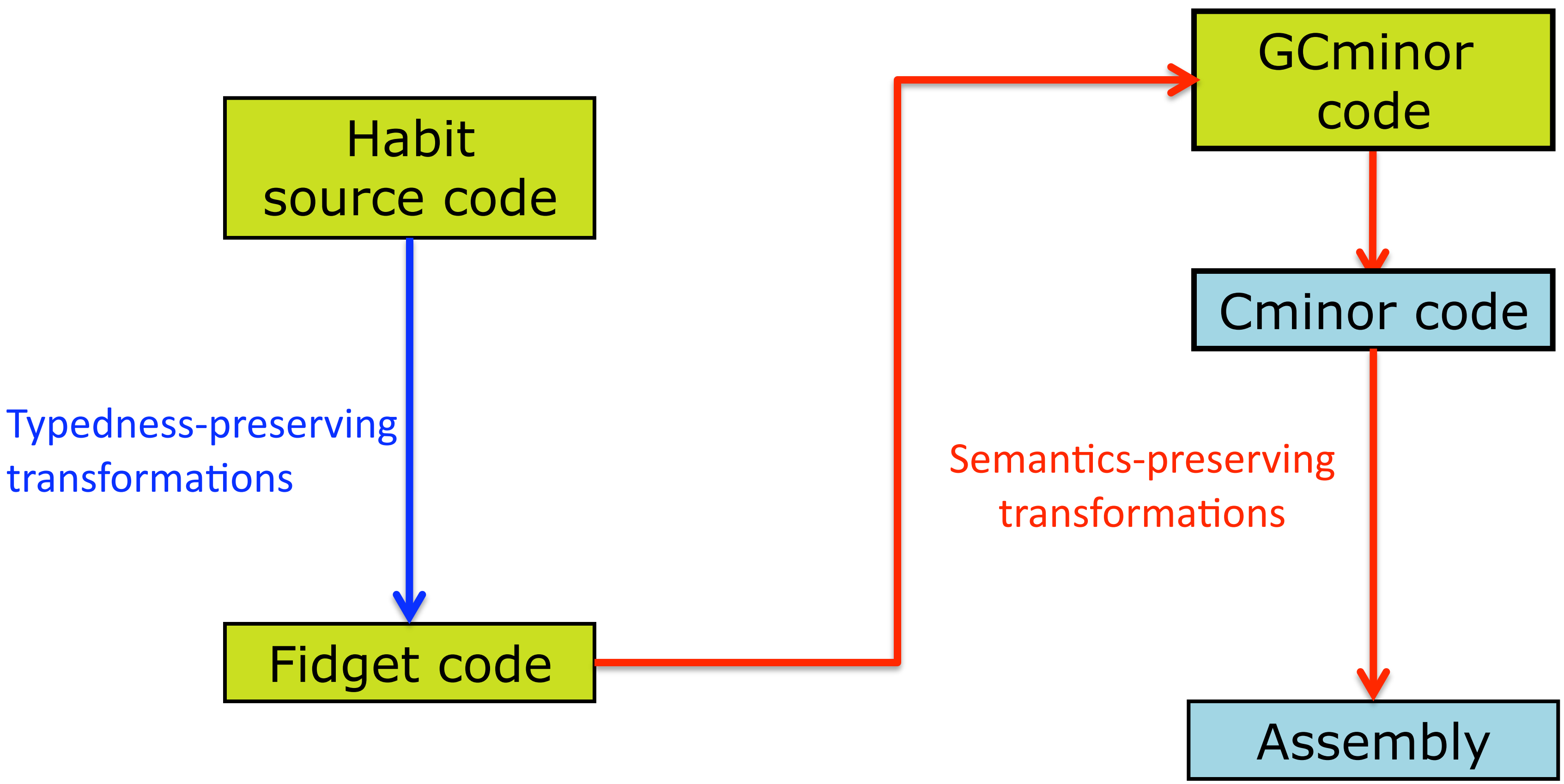
Haskell Case Study Architecture



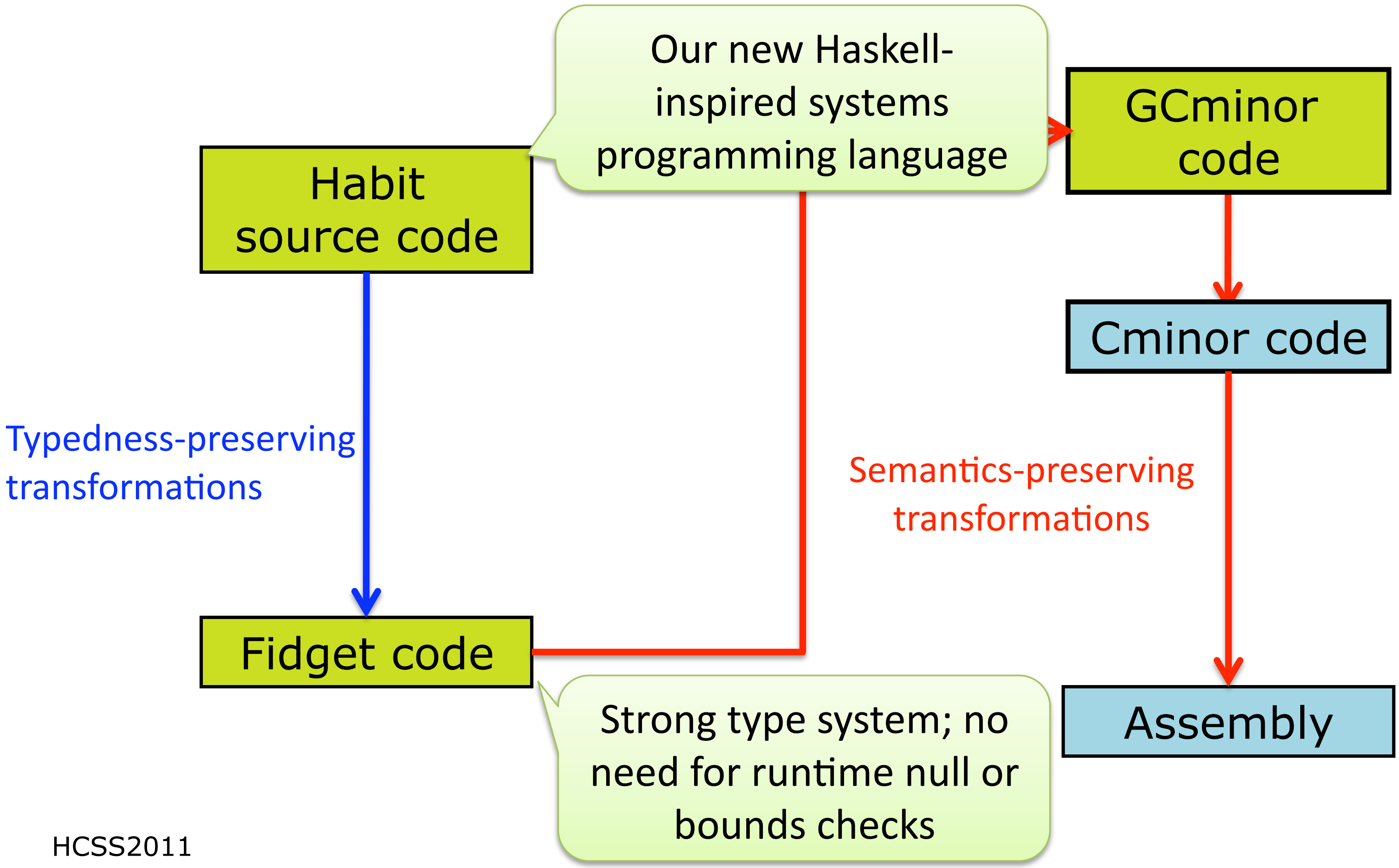
Assurance Argument

- Semantics preservation proof for whole front-end would be huge effort
- Much simpler to prove only safety of the front-end using types
- New Dminor IR bridges between typed and untyped worlds
- As an experiment, we kept type system very minimal, so much of safety argument relies on run-time checks

Current work: Habit front-end



Current work: Habit front-end



Future Challenges

- Extending RTS to support privileged hardware
  - e.g. MMU control for secure inter-language ops
  - will require novel intermediate languages
- Incorporating non-determinism
  - e.g. pre-emptive multithreading, multicores
  - breaks CompCert's forward simulation approach
- More realistic collectors; more front ends
  - need to raise level of Coq proof automation