Synthesis of Concurrent Garbage Collectors and their Proofs

Cordell Green
Christoph Kreitz
Douglas R Smith
Eric W Smith
Stephen Westfold

Kestrel Institute
Palo Alto, California

www.kestrel.edu

Synthesis of Concurrent Garbage Collectors

Garbage Collection Specification + Derivation Script → **Specware Generator** → Garbage Collector + Correctness Proofs

- Safety
- Security
- Transparency
- Context Assumptions

Inference

Transformation Library

Algorithm theories,
Invariant enforcement,
Datatype refinements,
Concurrency transforms

Simplified Derivation Structure

The initial specification is refined to code by applying a sequence of high-level transformations. Each transformation adds detail.

How do we compose a proof that the code is consistent with the initial specification?

Most of the transformations work by calculation: a sequence of equations from the domain theory are applied.

The calculation is the proof!

Garbage Collection Specification

↓ Algorithm design

Tracing Alg

↓ Optimization

Simplified Alg

↓ Observer Maintenance

Maintain WorkSet

↓ Observer Refinement

Refine Observers

↓ Cotype Synthesis

Synthesize Heap cotype

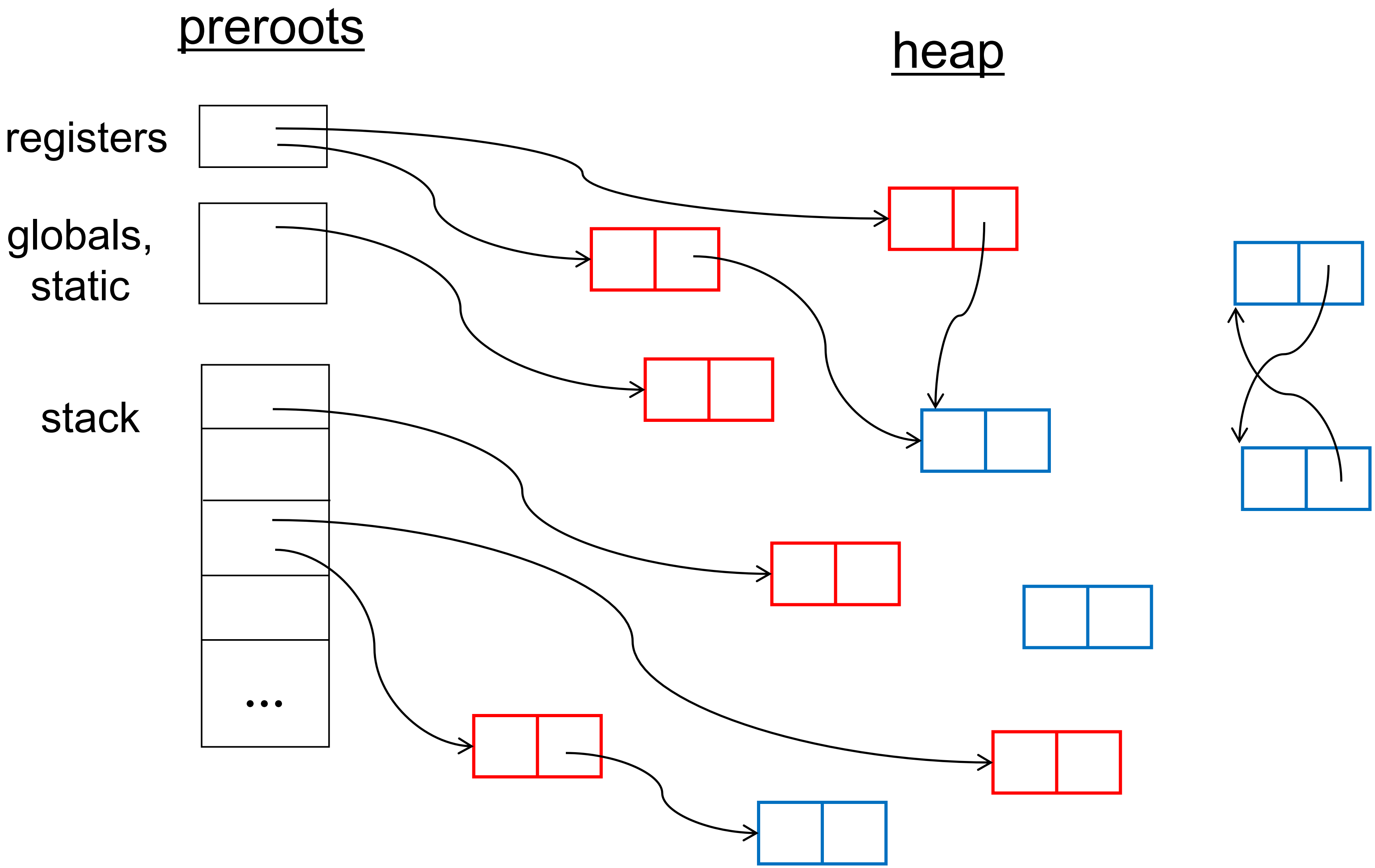
↓ Transformer Synthesis

Synthesize Transformers

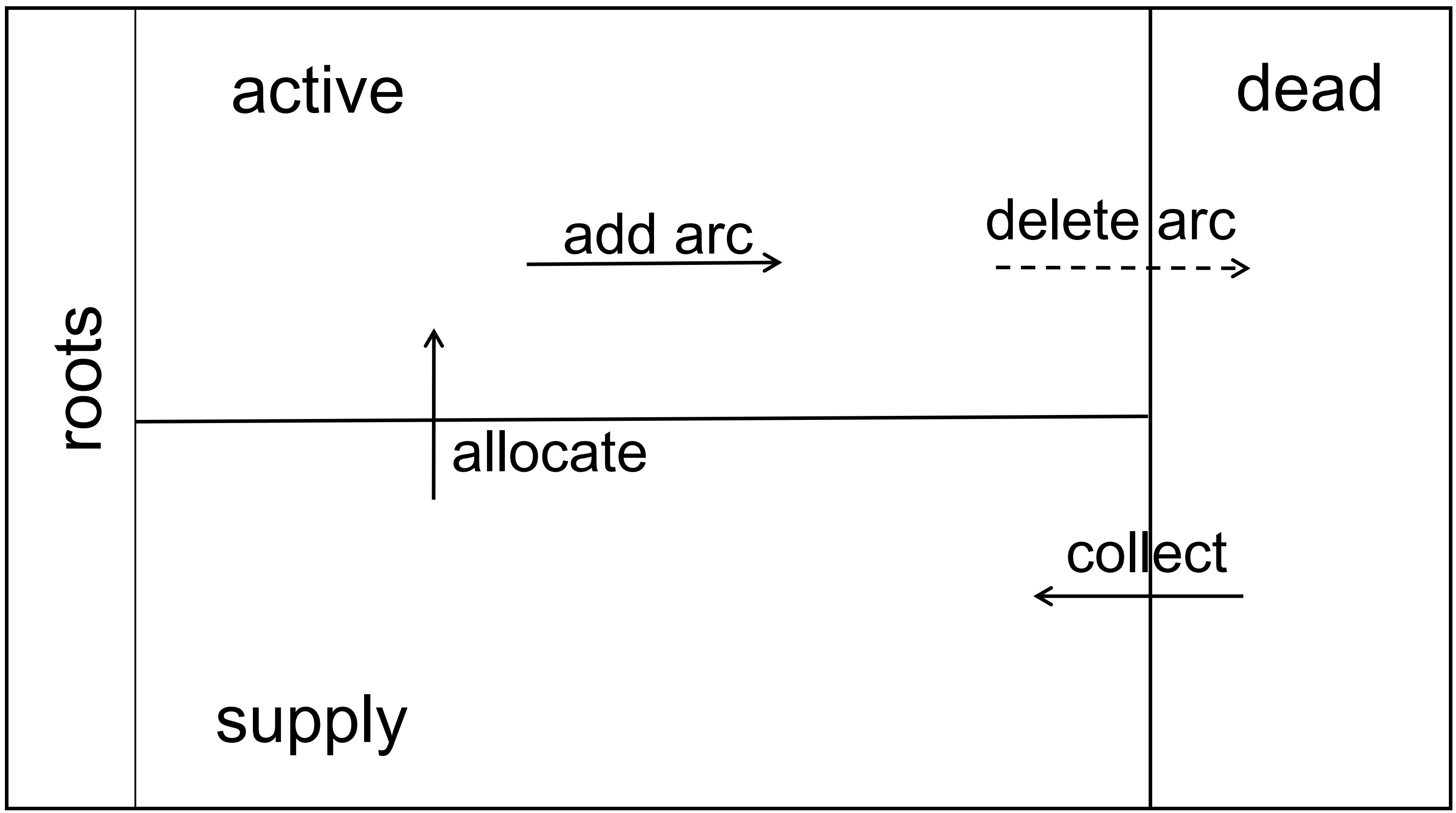
↓ Globalization

Code Generation

Model of Memory as a Rooted Graph

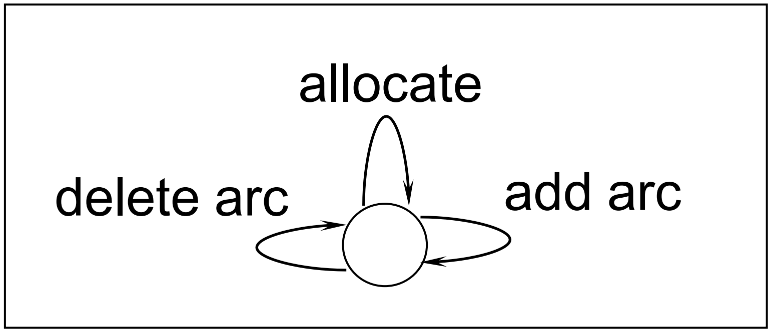


Regions of Memory and Basic Operations

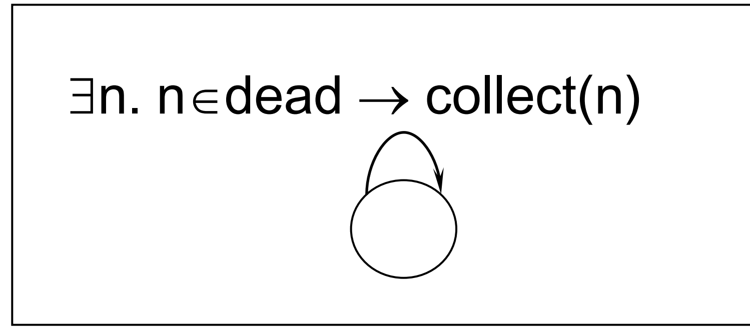


State Machine Models: Mutators + Collector

Mutator



Collector



Mutator is an application that allocates heap nodes, and manipulates arcs (pointers).

Collector identifies dead nodes and recycles them.

A node is dead if there are no paths to it from the roots

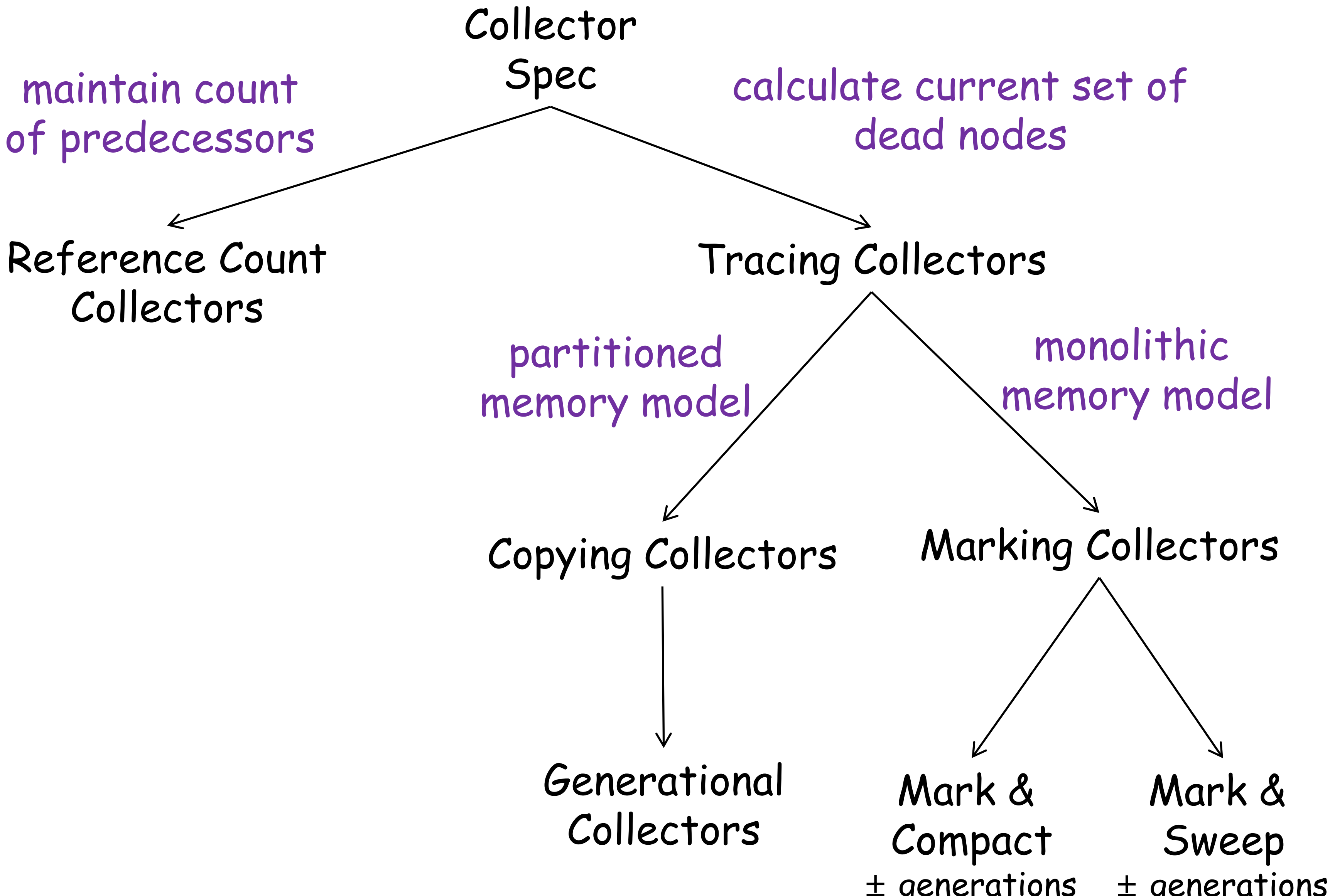
$n \in dead \Leftrightarrow paths(roots, n) = \{\}$

Requirements

Safety: No active nodes are ever collected

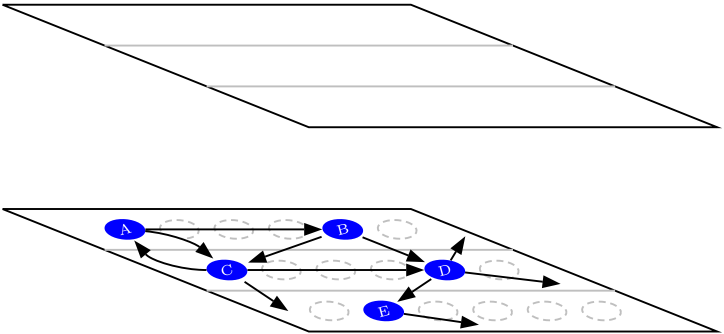
Transparency: Throughput, pause times, footprint, promptness

Deriving Common Garbage Collection Algorithms

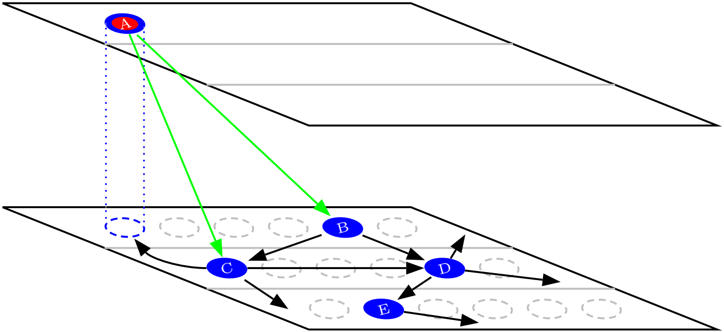


What’s Challenging about Concurrent GC?

Pictorial Intuition: Graph Traversal as Lifting



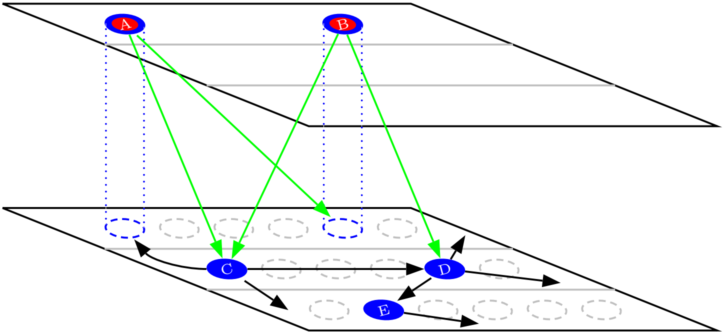
Pictorial Intuition (1)



Lift node A to the upper plane (equivalent "twin nodes")
Node A is active ("hot zone")

Invariant: Downward (green) arrows originate in hot zone

Pictorial Intuition (2)

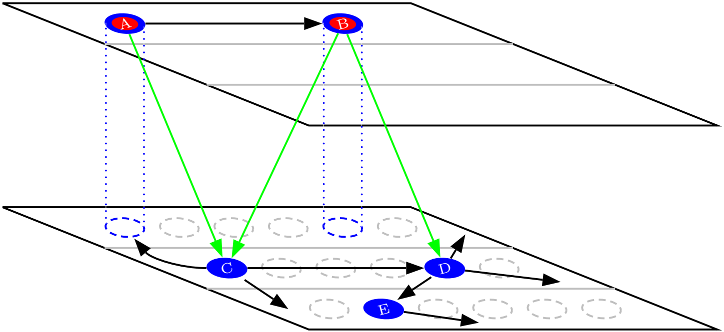


Lift node B to the upper plane

Node B is active ("hot zone")

Invariant: Downward (green) arrows originate in hot zone

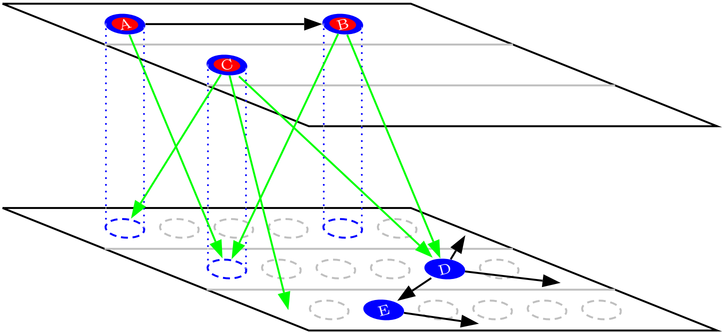
Pictorial Intuition (3)



Adjust downward arc $A \rightarrow B$ to upper plane
Node A remains active

Invariant: Downward (green) arrows originate in hot zone

Pictorial Intuition (4)

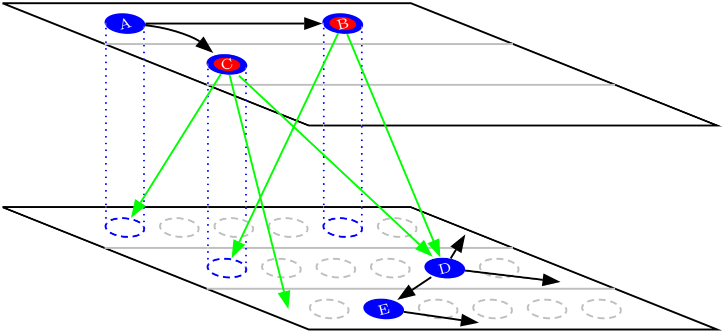


Lift node C to the upper plane

Node C is active ("hot zone")

Invariant: Downward (green) arrows originate in hot zone

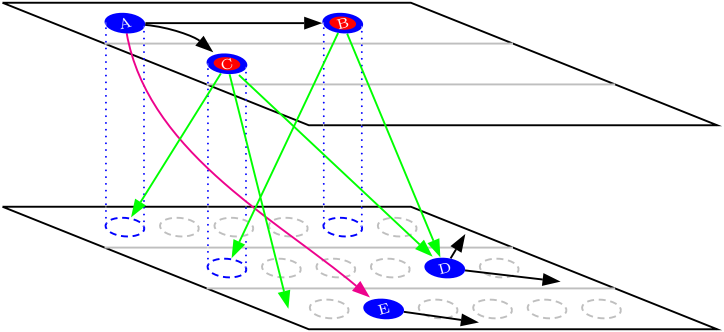
Pictorial Intuition (5)



Adjust downward arc $A \rightarrow C$ to upper plane
Node A becomes inactive

Invariant: Downward (green) arrows originate in hot zone

This leads to the following situation:



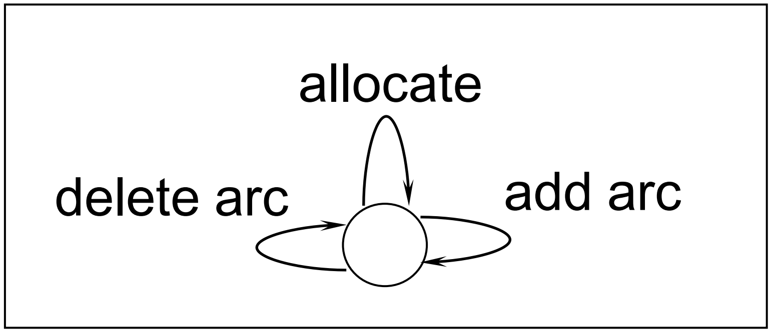
Invariant is violated:

Now there is a downward arrow, which does not originate in the hot zone!

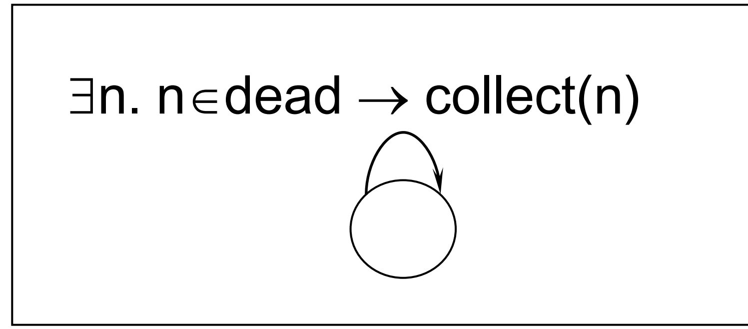
$\rightsquigarrow$ E will be considered unreachable (garbage)

State Machine Models: Mutators + Collector

Mutator



Collector



Mutator is an application that allocates heap nodes, and manipulates arcs (pointers).

Collector identifies dead nodes and recycles them.

A node is dead if there are no paths to it from the roots

$$n \in dead \Leftrightarrow paths(roots, n) = \{\}$$

<u>Requirements</u>

Safety: No active nodes are ever collected

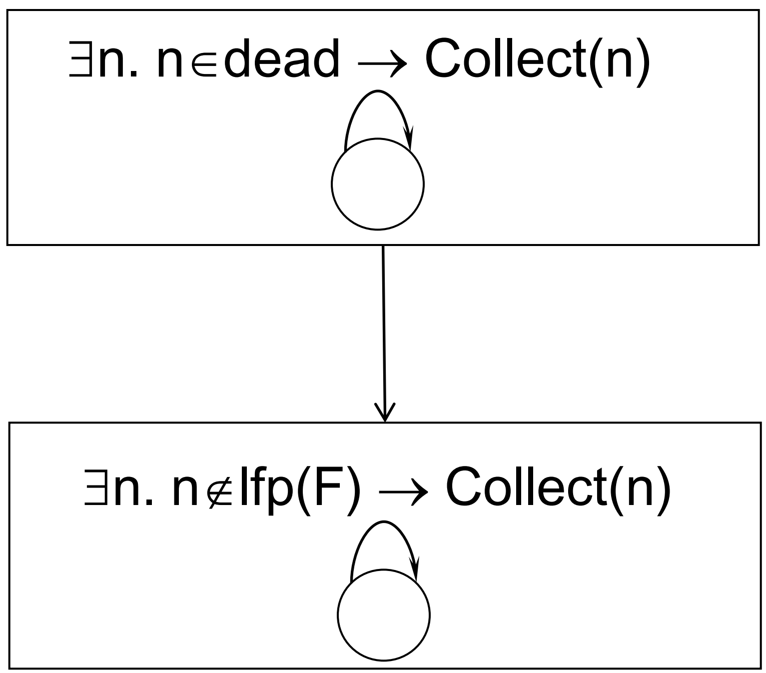
Transparency: Throughput, pause times, footprint, promptness

Tracing Collectors

$n \in dead \Leftrightarrow n \notin live$ where $live = active \cup supply$

$\Leftrightarrow n \notin$ least $S.\ roots \cup sucs(S) \subseteq S$

$\Leftrightarrow n \notin lfp(F)$ where $F(S) = roots \cup sucs(S)$



Tracing Collectors: Kleene

To compute lfp(F), where F is a monotone function in a powerset lattice:

```
S ← {};
while  S ≠ F(S) do
    S ← F(S);
return S
```

instantiating F:

```
S ← {};
while  S ≠ roots ∪ sucs(S) do
    S ← roots ∪ sucs(S) ;
return S
```

generalized proof over complete partial orders (cpo's) based on Tarski, Kleene theorems

Tracing Collectors: Kleene

```
S ← {};
while  S ≠ roots ∪ sucs(S) do
     S ← roots ∪ sucs(S) ;
return S
```

Problem: roots and sucs depend on the state of the heap;

does the algorithm work concurrently?

The Kleene iteration computes where $F(S) = roots \cup sucs(S)$

$\{\}, F(\{\}), F(F(\{\})), \ldots F^n(\{\})$, until convergence

However the memory graph evolves as

$G_0, G_1, G_2, \ldots G_n, \ldots$

so the iteration produces

$\{\}, F_0(\{\}), F_1(F_0(\{\})), F_2(F_1(F_0(\{\}))), \ldots$

what does this sequence converge to?

Tracing Collectors: Kleene

```
S ← {};
while  S ≠ roots ∪ sucs(S) do
    S ← roots ∪ sucs(S) ;
return S
```

Problem: roots and sucs depend on the state of the heap;

does the algorithm work concurrently?

In a MPC10 paper we proved weak conditions under which

$\{\}, F_0(\{\}), F_1(F_0(\{\})), F_2(F_1(F_0(\{\}))), \ldots$

converges to a (non-least) fixpoint of the initial graph G_0.

Effect: When executed concurrently, we can generate a proof that the algorithm returns a subset of dead nodes.

Tracing Collectors

```
S ← {};
 while  S ≠ roots ∪ sucs(S) do
      S ← roots ∪ sucs(S) ;
return S
```

Problem: How to make the iteration efficient?

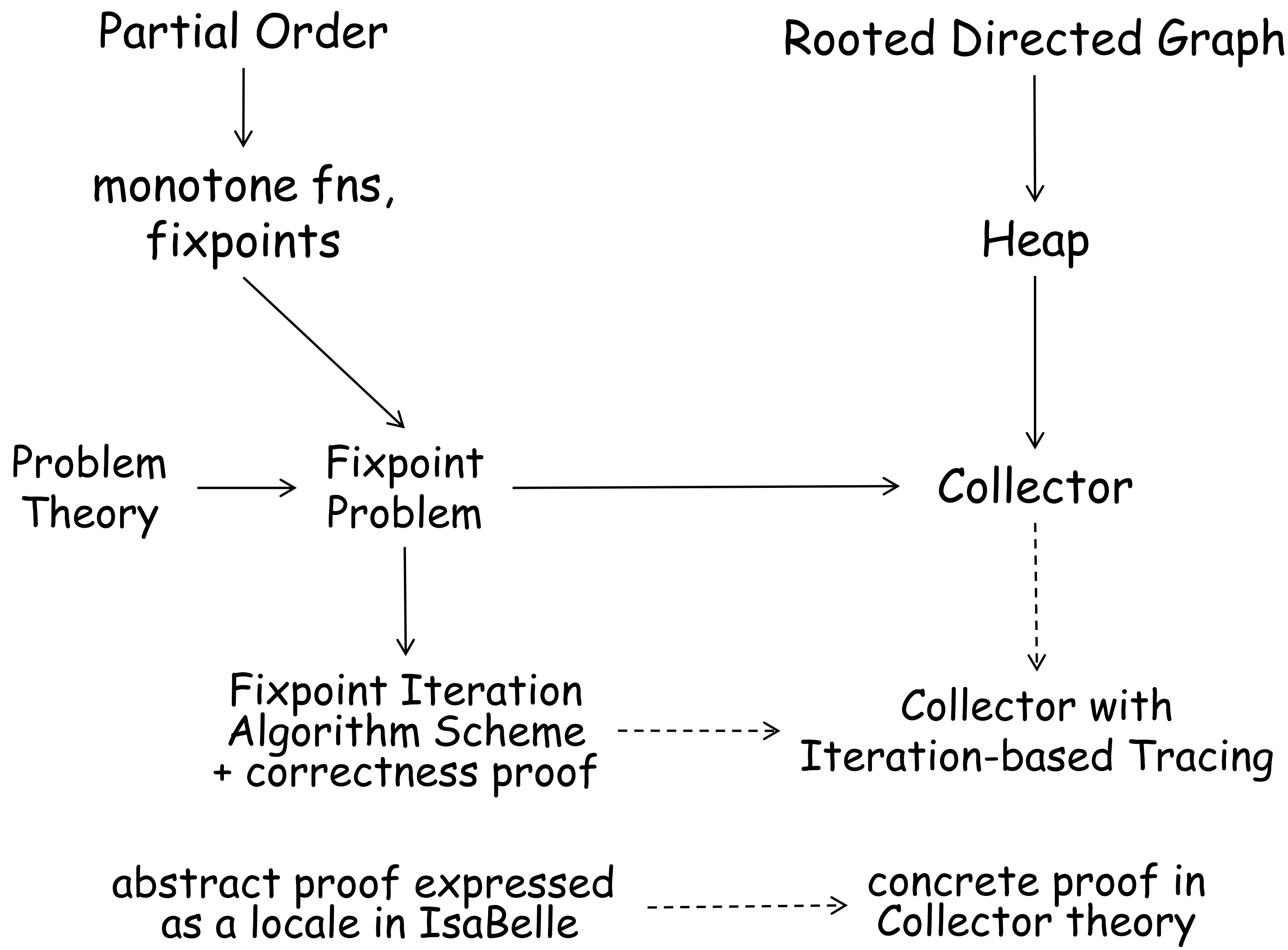
→ introduce a workset

$$WS = F(S) \setminus S$$
$$= (roots \cup sucs(S)) \setminus S$$

```
S ← {};
 while  ∃z∈ (roots ∪ sucs(S))\S do
     S ← S ∪ {z};
return S
```

proof based on Cai/Paige theorems

Derivation Structure



Simplified Derivation Structure

The initial specification is refined to code by applying a sequence of high-level transformations. Each transformation adds detail.

How do we compose a proof that the code is consistent with the initial specification?

Most of the transformations work by calculation: a sequence of equations from the domain theory are applied.

The calculation is the proof!

Garbage Collection Specification

↓ Algorithm design

Tracing Alg

↓ Optimization

Simplified Alg

↓ Observer Maintenance

Maintain WorkSet

↓ Observer Refinement

Refine Observers

↓ Cotype Synthesis

Synthesize Heap cotype

↓ Transformer Synthesis

Synthesize Transformers

↓ Globalization

Code Generation

Specifying Algebraic Types

An algebraic type is defined by constructors

- well-founded
- new functions defined inductively over constructors

type List a = nil | cons a (List a)

List is defined using constructors nil and cons

op length: List a $\rightarrow$ Nat
length nil = 0
length (cons a lst) = 1 + length lst

length is defined inductively in terms of its value over the constructors

Specifying Coalgebraic Types (aka cotypes)

A coalgebraic type is characterized by observers

- not well-founded: may be circular or infinite
- transformers specified coinductively by effect on observers

```
type Graph
op   nodes: Graph → Set Node
op   sucs  : Graph  → Node → Set Node
```

Graph is specified using observers nodes and sucs

```
op  addArc(G:Graph) (x:Node, y:Node) :
             {G':Graph |   nodes G' = nodes G
                         & sucs G' x  = (sucs G x) + y }
```

addArc is specified coinductively in terms of its effect on the observers

Coalgebraic Specifications

- Algebraic types used for ordinary data (boolean, Nat, List)
- Coalgebraic type used for heaps
- Observers obs: Heap $\rightarrow$ *A*
  - basic/undefined
  - defined but maintained
  - defined but computed
- Transformers t: Heap $\rightarrow$ Heap
  - preconditions
  - postconditions: coinductive constraints on observations

Tracing Collectors:

Instantiated Small-Step Fixpoint Iteration

$S \leftarrow \{\}$

while $\exists z \in (\text{roots}(G) \cup \text{sucs}(G)(S)) \setminus S$ do

  $S \leftarrow S \cup \{z\}$

return S

to optimize the algorithm, we introduce a new observer:

$$\boxed{\text{WS } G = (\text{roots } G \cup \text{sucs}(G)(S)) \setminus S}$$

Maintaining Observers

Observer Maintenance Transform (aka Finite Differencing)

- given a defined observer

  WS (G:Graph):Set *A* = e *G*

- for each transformer t, add definition to postcondition:

  t(*G*:Graph | WS *G* = e *G*):

  {*G*':Graph | … && WS *G*' = e *G*' }

- simplify

Maintaining Observers

type Graph
op nodes: Graph $\to$ Set Node
op outArcs : Graph $\to$ Node $\to$ Set Node
op roots : Graph $\to$ Set Node
op S : Graph $\to$ Set Node
op WS(G:Graph):Set Node = (roots $G \cup$ outArcs G (S G)) \ (S G)

op addArc(G:Graph) (x:Node, y:Node) :
{G′:Graph | nodes G' = nodes G
$\wedge$ outArcs G' x = (outArcs G x) + ($x \to y$)
$\wedge$ WS G' = WS $G \cup \{y \mid x \in S\ G \wedge y \notin S\ G\}$}

design-time calculation:

$$
\begin{aligned}
\text{WS } G' &= (\text{roots } G' \cup \text{outArcs } G' (S\ G)) \setminus (S\ G)\\
&= (\text{roots } G \cup \text{outArcs } (G \cup \{x \to y\}) (S\ G)) \setminus (S\ G)\\
&= (\text{roots } G \cup \text{outArcs } G\ S) \setminus (S\ G) \cup \{y \mid x \in (S\ G)\} \setminus (S\ G)\\
&= \text{WS } G \cup \{y \mid x \in (S\ G) \wedge y \notin (S\ G)\}
\end{aligned}
$$

Tracing Collectors

after all design-time calculations to enforce the invariant:

invariant $WS = (roots \cup outArcs(S)) \setminus S$

atomic $\langle\ S \leftarrow \{\}\ ||\ WS \leftarrow roots\ \rangle$

while $\exists z \in WS$ do

    atomic $\langle\ S \leftarrow S \cup \{z\}\ ||\ WS \leftarrow WS \cup outArcs(z) \setminus S - z\ \rangle$

return S

atomic $\langle\ addArc(x,y)\ ||\ WS \leftarrow WS \cup \{y \mid x \in S \wedge y \notin S\}\ \rangle$

this is essence of the coarse-grain Dijkstra et al. "on-the-fly" collector

Observer Refinement

- refine an existing observer WS (G:Graph):Set *A*

  by a new observer WL (G:Graph):List *A*

  WS G = List_to_Set (WL G)

  where List_to_Set is a homomorphism
- replace all occurrences of WS by List_to_Set∘WL
- simplify

Refining Observers

type Graph

axiom $\quad WS\ G = List2Set\ WL\ G$

op addArc(G:Graph) (x:Node, y:Node) :

$$\{G'{:}Graph \mid \quad nodes\ G' = nodes\ G$$
$$\wedge\ outArcs\ G'\ x\ = (outArcs\ G\ x) + (x \rightarrow y)$$
$$\wedge\ WL\ G' = WL\ G \mathbin{++} [y \mid x \in S\ \&\ y \notin S]\ \}$$

design-time calculation:

$$\begin{array}{ll}
 & WS\ G' = WS\ G \cup \{y \mid x \in S \wedge y \notin S\} \\
\Leftrightarrow & L2S\ WL\ G' = L2S\ WL\ G \cup \{y \mid x \in S \wedge y \notin S\} \\
\Leftrightarrow & L2S\ WL\ G' = L2S\ WL\ G \cup L2S\ [y \mid x \in S \wedge y \notin S] \\
\Leftrightarrow & L2S\ WL\ G' = L2S\ (WL\ G \mathbin{++} [y \mid x \in S \wedge y \notin S]) \\
\Leftarrow & WL\ G' = \qquad WL\ G \mathbin{++} [y \mid x \in S \wedge y \notin S]
\end{array}$$

Generating Proof Scripts

For example, a refinement based on this calculation from the derivation of a Mark & Sweep garbage collector:

Sequence of Rewrites	Justification for Each Step
initialState x0	
= FHeap x0 {}	unfolding initialState
= roots x0 ∪ allOutNodes x0 {}	unfolding FHeap
= roots x0 ∪ {}	rule allOutNodes_of_emptyset
= roots x0	rule right_unit_of_union

automatically generates this Isabelle/Isar proof script :

```
theorem initialState_refine_def:
  "(initialState x0) = (roots x0)"
proof -
  have " (initialState x0)
                    = FHeap x0 {}"                        by (unfold initialState_def, rule HOL.refl)
  also have     "... = (roots x0 ∪ allOutNodes x0 {})"    by (unfold FHeap_def, rule HOL.refl)
  also have     "... = (roots x0 ∪ {})"                   by (rule_tac f="λy . (?term ∪ y)" in arg_cong,
                                                              rule allOutNodes_of_empty_set)
  also have     "... = (roots x0)"                        by (rule union_right_unit)
  finally show ?thesis .
qed
```

The proof script discharges the proof obligation of the refinement

Simplified Derivation Structure

The initial specification is refined to code by applying a sequence of high-level transformations. Each transformation adds detail.

How do we compose a proof that the code is consistent with the initial specification?

Most of the transformations work by calculation: a sequence of equations from the domain theory are applied.

The calculation is the proof!

Garbage Collection Specification

↓ **Algorithm design**

Tracing Alg

↓ **Optimization**

Simplified Alg

↓ **Finite Differencing**

Maintain WorkSet

↓ **Type Refinement**

Refine Observers

↓ **Cotype Synthesis**

Synthesize Heap cotype

↓ **Transformer Synthesis**

Synthesize Transformers

↓ **Globalization**

Code Generation

Cotype Synthesis – Extract a Final Model

Given these undefined or maintained observers

```
nodesL    : Memory -> List Node
rootsL    : Memory -> List Node
supplyL   : Memory -> List Node
WL        : Memory -> List Node

blackCM   : Memory -> Map(Node,Boolean)
sucsIM    : Memory -> Map(Node,Map(Index, Arc))

srcM      : Memory -> Map(Arc,Node)
tgtM      : Memory -> Map(Arc,Node)
```

Cotype Synthesis – Extract a Final Model

reify all undefined or maintained observers into a product

```
type Memory= { nodesL    : List Node,
               rootsL    : List Node,
               supplyL   : List Node,
               WL        : List Node,
               blackCM   : Map(Node, Boolean),
               sucsIM    : Map(Node, Map(Index, Arc)),
               srcM      : Map(Arc, Node),
               tgtM      : Map(Arc, Node)
             }
type Node
type Arc
```

Transformer Synthesis

- replace the constraints in transformer postconditions by concurrent updates of the cotype
- simplify

```
type Memory { nodesL : List Node,
                  WL : List Node,
              arcMap : Map(Arc, Node * Node)
              ... }

op  swingArc(G:Memory) (x:Node, i:Index, y:Node| ok?(x,y)) : Memory =
    G <<  {arcMap = update (G.arcMap).(x,i)  (x,y)
           WL     = G.WL ++ [y | x∈G.S & y ∉ G.S] }
```

Globalization

- add global variable of the cotype
  var M: Memory
- eliminate the cotype (Memory) in all functions
  - parameter (at most one)
  - return type
- replace local refs to state by global refs

```
type Memory= {nodesL : List Node,
                    sucsM :  map(Node, List Node),
                       WL : List Node }
var M:Memory

op  addArc(x:Node, y:Node) : Unit =
    (    M.sucsM x := (M.sucsM x) + y
    ||         M.WL := M.WL ++ [n | m∈S & n ∉ S]  )
```

Summary

- coalgebraic specification and refinement techniques
- Basic specification and refinement support in Specware
- Platform-independent derivations of concurrent M&S
- New transformations:
  - dynamic fixpoint iteration
  - observer maintenance
  - observer refinement
  - cotype definition
  - globalization

Next steps:

- Output checkable proofs
- Copying Collectors
- Code generation to multithreaded C and CommonLisp

References

Dusko Pavlovic, Peter Pepper, and Douglas R. Smith, Colimits for Concurrent Collectors, in *Verification: Theory and Practice* (Z. Manna Festschrift), Springer LNCS 2772, 2003, 568-597.

Dusko Pavlovic, Peter Pepper, and Douglas R. Smith, Formal Derivation of Concurrent Garbage Collectors, in *Mathematics of Program Construction 2010* (MPC10), Springer LNCS 6120, July 2010, 353-376.

Extras

Derivation Structure

Fixpoint Problem → GC_0 safety, security, transparency reqts

Fixpoint Problem ↓ Fixpoint Iteration Algorithm Scheme

GC_0 ↓ Algorithm Design: Workset-based Fixpoint Iteration

po

Fixpoint Iteration Algorithm Scheme → GC_1

GC_1 ↓ Maintain invariants; Optimize the code; Introduce synchronization

GC_2

GC_2 ↓ Datatype Refinement

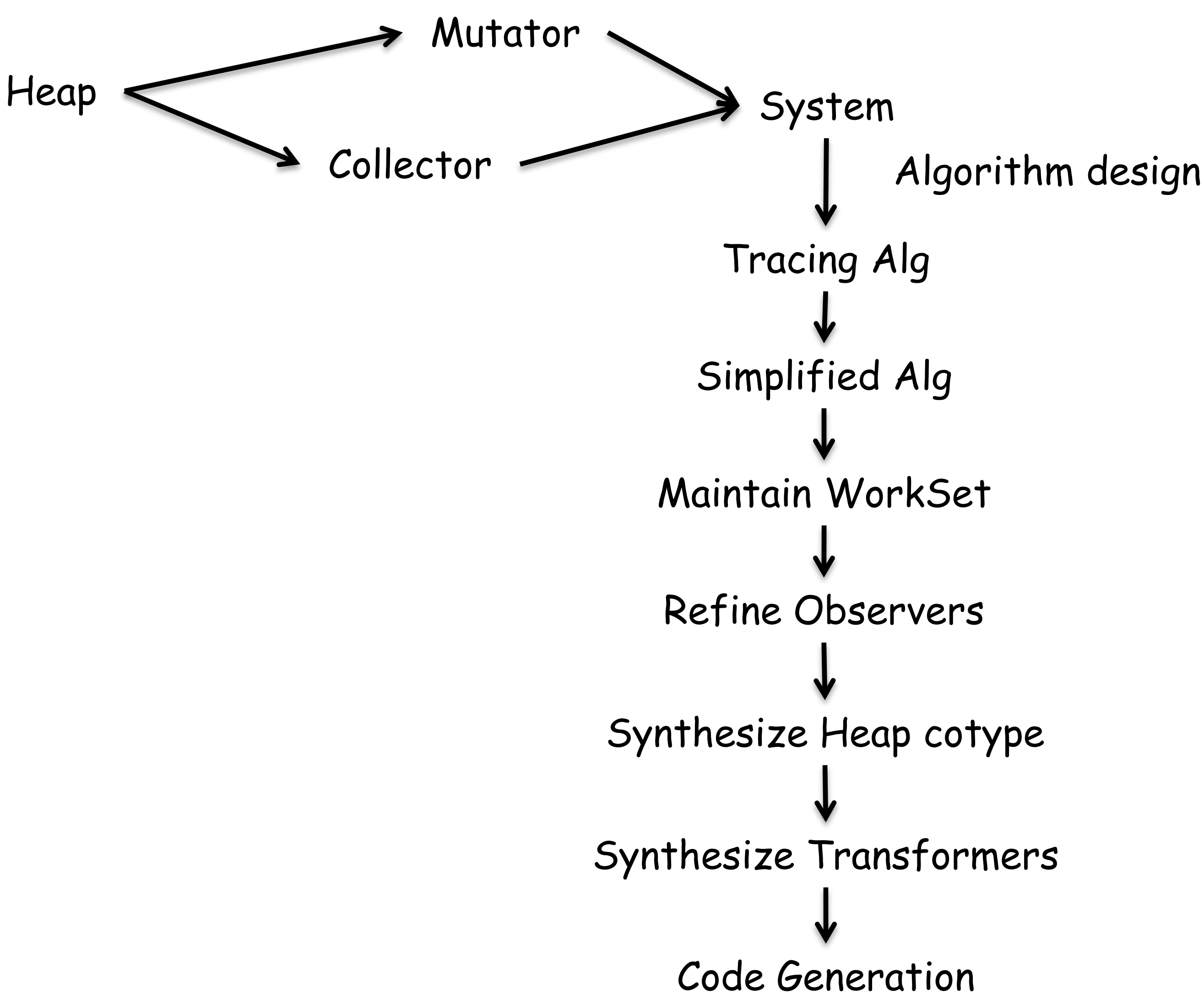
- Graph ↦ Monolithic Memory
- Node ↦ Block = Addr×Header×Body
- Arc ↦ Block×Offset×Block

Mark&Sweep Collector

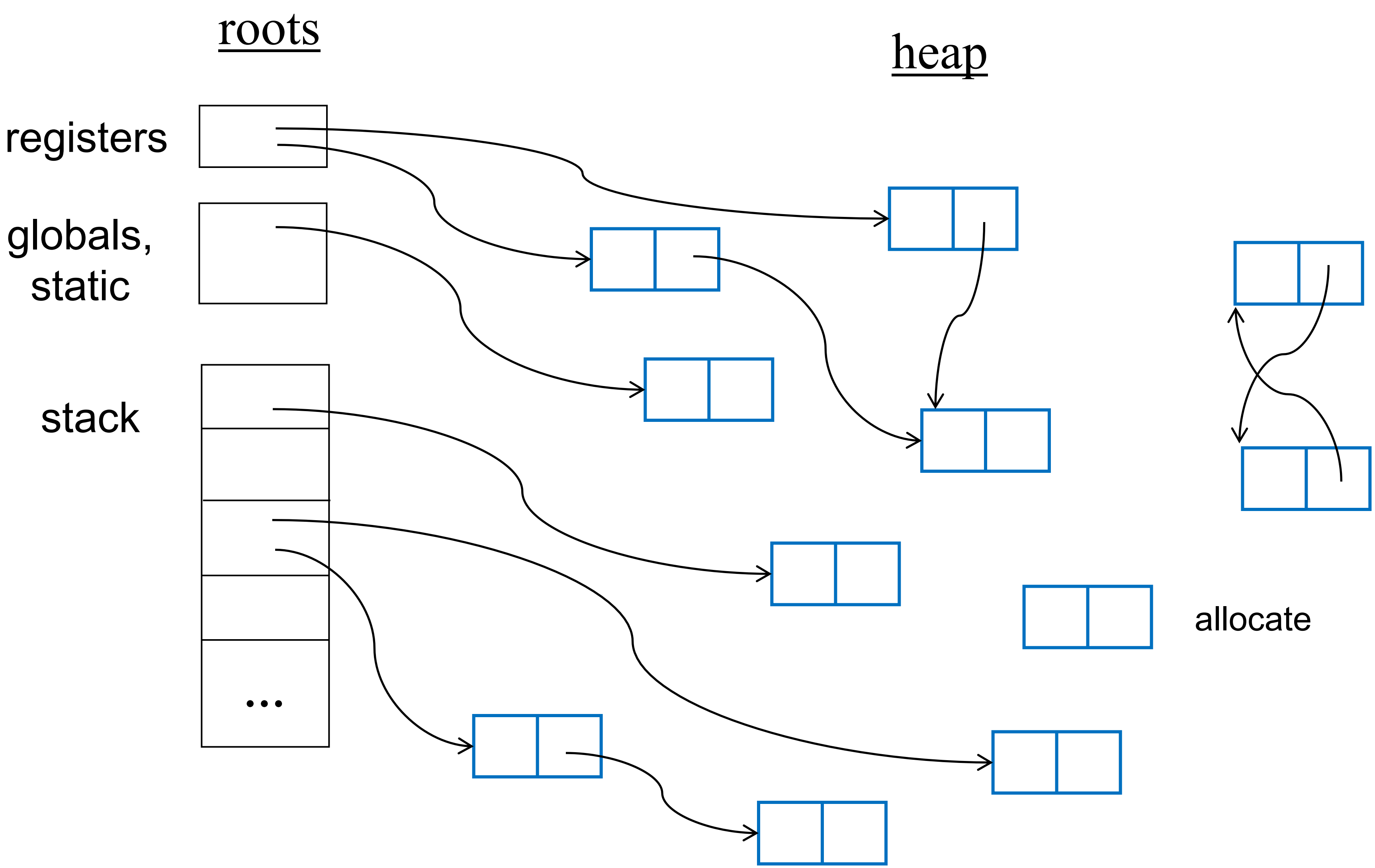
Mark&Sweep Collector ↓ Code Generation

code

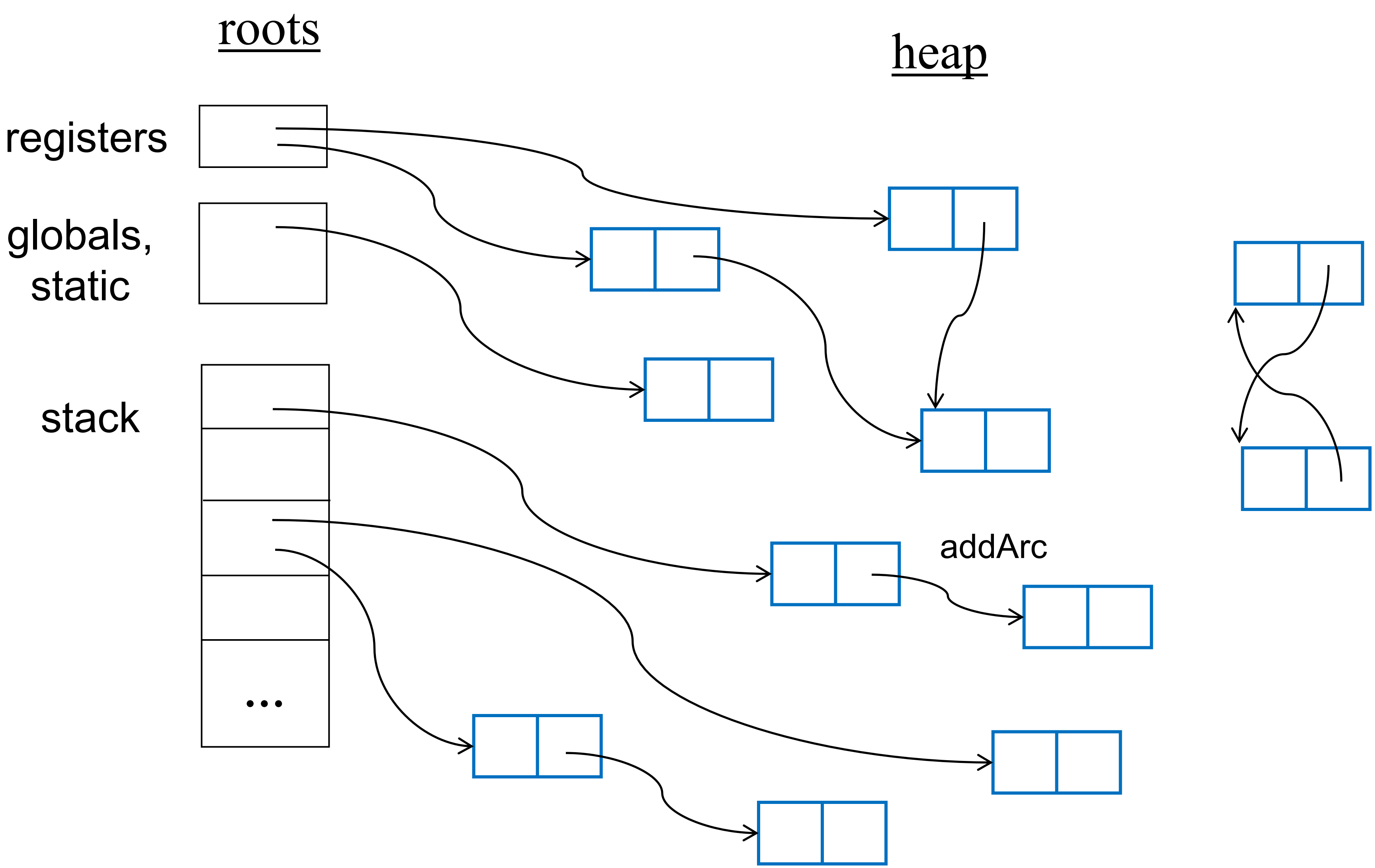
Simplified Derivation Structure



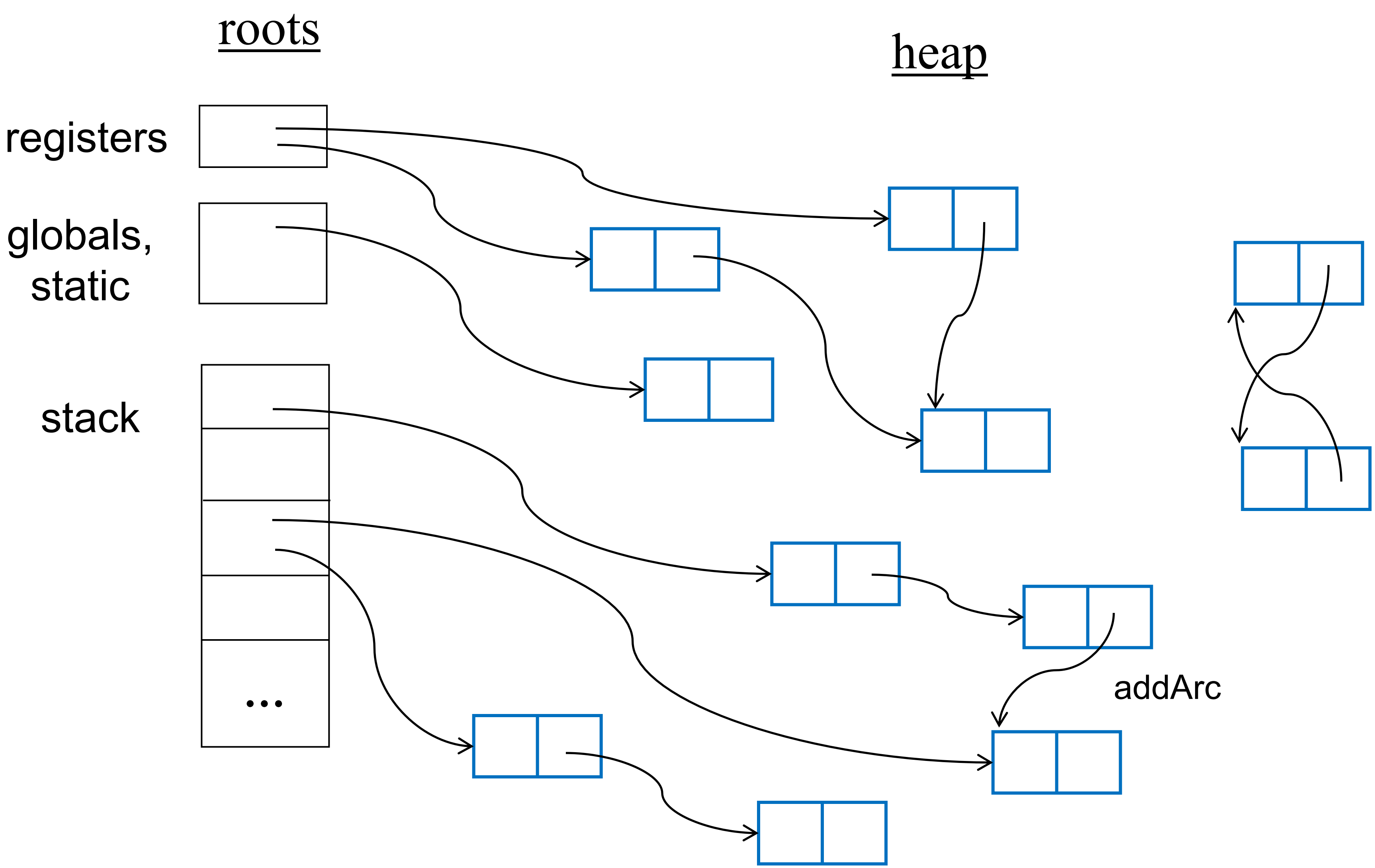
Model of Memory as a Rooted Graph



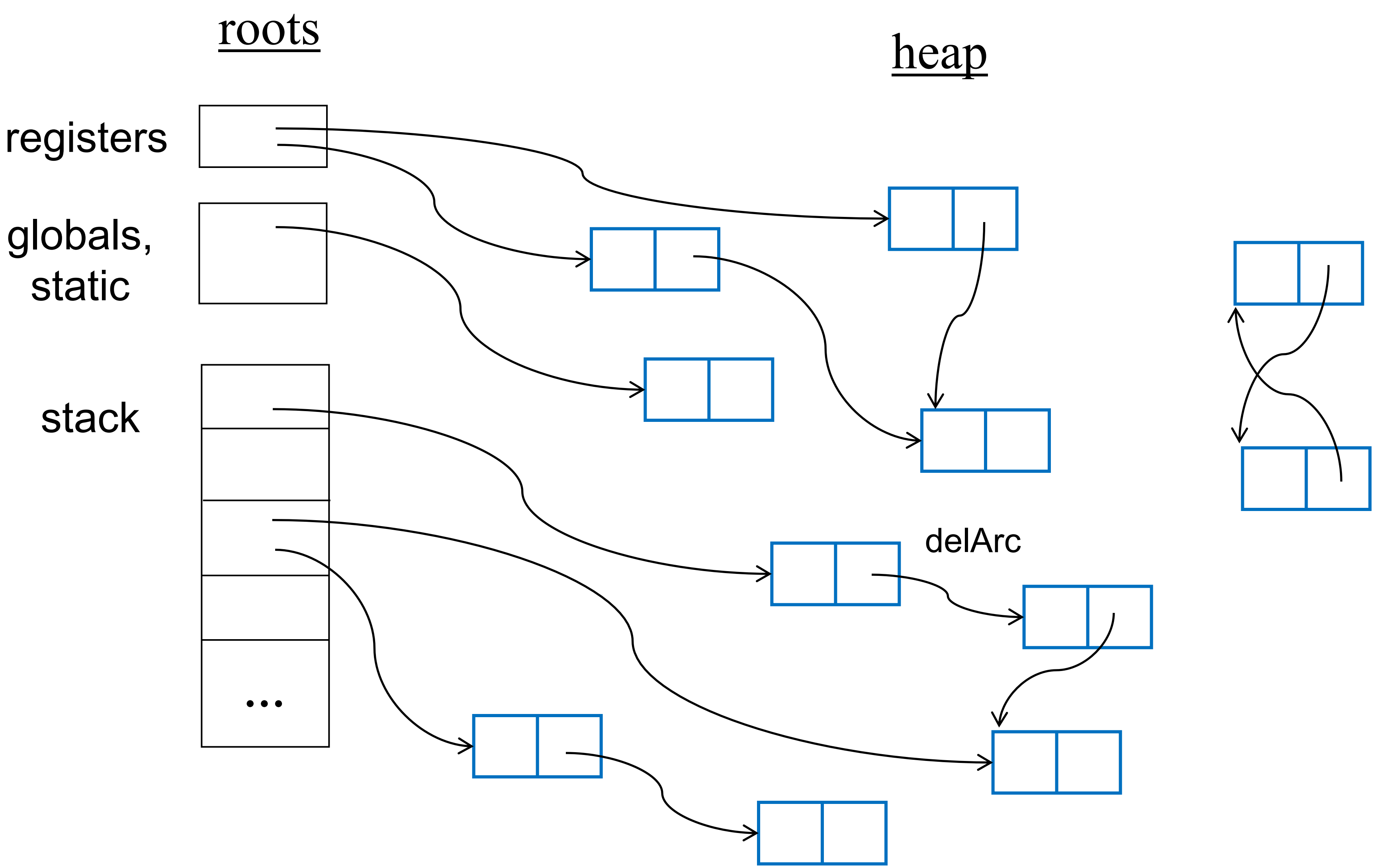
Model of Memory as a Rooted Graph



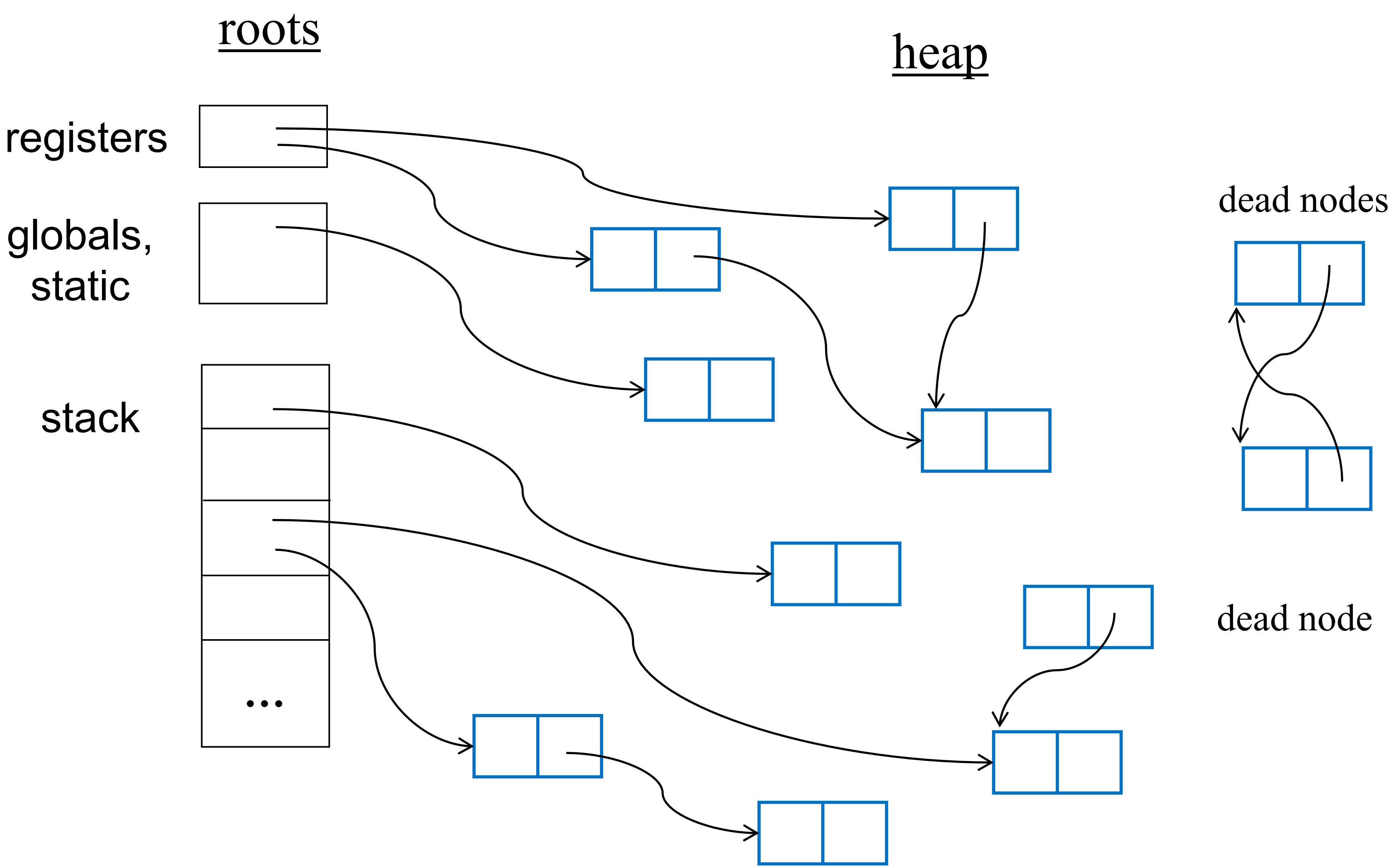
Model of Memory as a Rooted Graph



Model of Memory as a Rooted Graph



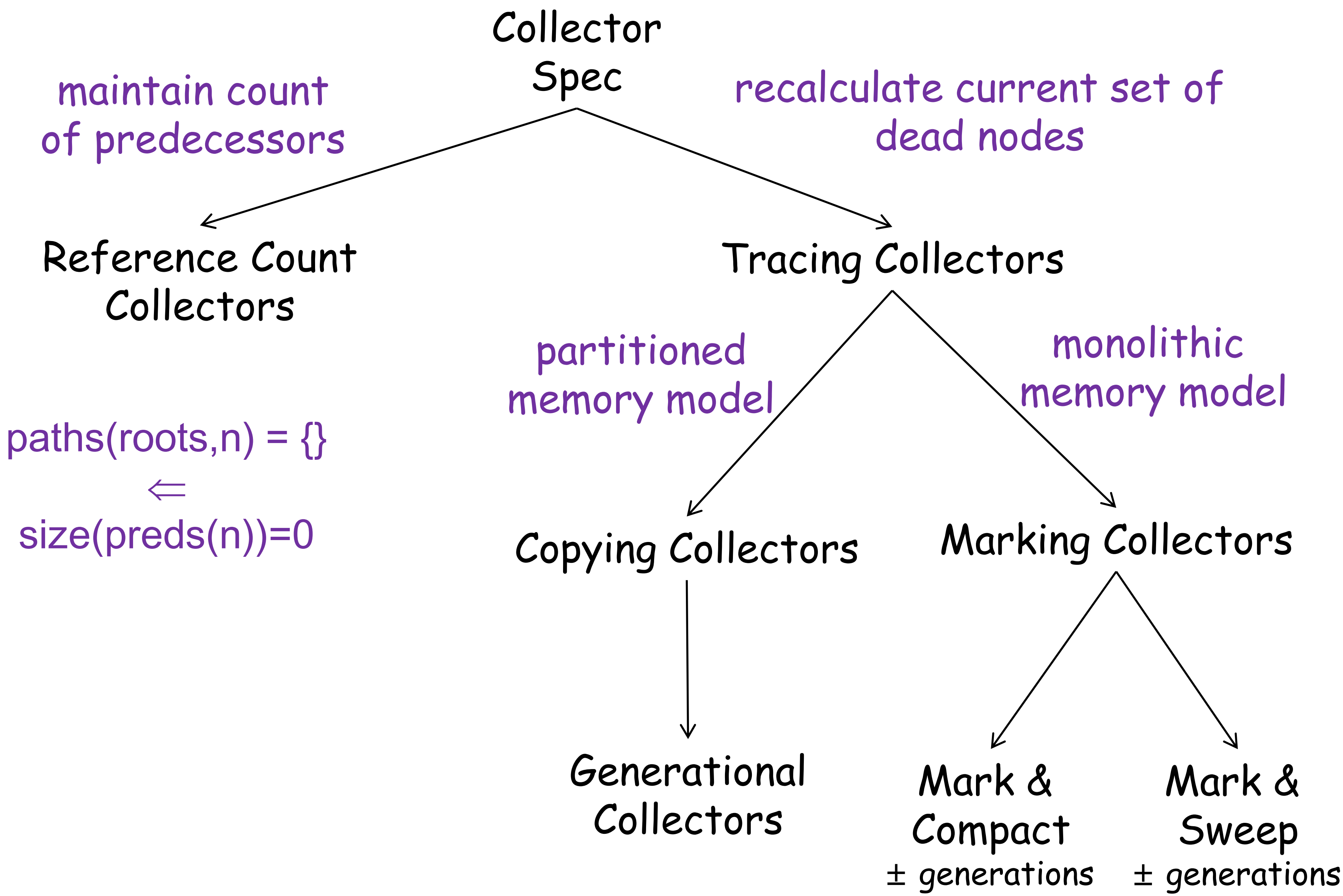
Model of Memory as a Rooted Graph



Generating Proof Scripts: Status

- We have worked out the general proof script pattern, using examples from the synthesis of Garbage Collectors (DARPA CRASH program).
- We are currently implementing a mechanism to translate transformation steps into proof script steps: For each kind of transformation step, we have developed a general "meta-rule" for how to generate its corresponding proof script step.
- Prototype development in process – to be presented at HCSS.
- The proof scripts closely reflect the transformation steps by a one-to-one relationship; search by Isabelle is avoided.
- The proof scripts are meant for machine checkability, but are surprisingly readable!
- Anticipate that 90+% of proofs in the garbage collector derivations can be automatically co-generated with the refinements.
- In a calculational derivation, the *calculation is the proof!*

Deriving Common Garbage Collection Algorithms



Cotype Synthesis – Extract a Final Model

Given these undefined or maintained observers

```
nodesL    : Memory -> List Node
rootsL    : Memory -> List Node
supplyL   : Memory -> List Node
WL        : Memory -> List Node

blackCM    : Memory -> Map(Node,Boolean)
outArcsIM: Memory -> Map(Node,Map(Index, Arc))

srcM      : Memory -> Map(Arc,Node)
tgtM      : Memory -> Map(Arc,Node)
```

Cotype Synthesis – Extract a Final Model

reify all undefined or maintained observers into a product

```
type Memory= { nodesL     : List Node,
               rootsL     : List Node,
               supplyL    : List Node,
               WL         : List Node,
               blackCM    : Map(Node, Boolean),
               outArcsIM  : Map(Node, Map(Index, Arc)),
               srcM       : Map(Arc, Node),
               tgtM       : Map(Arc, Node)
             }
type Node
type Arc
```

Tracing Collectors: Workset

$$
\begin{array}{l}
S \leftarrow \{\}; \\
\text{while } \exists z \in (\text{roots} \cup \text{sucs}(S)) \backslash S \text{ do} \\
\quad S \leftarrow S \cup \{z\}; \\
\text{return } S
\end{array}
$$

$\rightarrow$ enforce the invariant $WS = (\text{roots} \cup \text{sucs}(S)) \setminus S$

$$\text{G:Graph} \xrightarrow[\;WS := WS \cup \{n \mid m \in S \wedge n \notin S\}\;]{\text{addArc}(m,\, n)} \text{G':Graph}$$

design-time calculation:

$$
\begin{aligned}
WS' &= (\text{roots} \cup G'.\text{sucs}(S)) \setminus S \\
&= (\text{roots} \cup (G \cup \{m \rightarrow n\}).\text{sucs}(S)) \setminus S \\
&= (\text{roots} \cup G.\text{sucs}(S)) \setminus S \cup \{n \mid m \in S\} \setminus S \\
&= WS \cup \{n \mid m \in S \wedge n \notin S\}
\end{aligned}
$$

essentially the Dijkstra et al. on-the-fly concurrent collector

Tracing Collectors

after all calculations to enforce the invariant:

invariant $WS = (roots \cup sucs(S)) \setminus S$

atomic[$S \leftarrow \{\} \;||\; WS \leftarrow roots$]

while $\exists z \in WS$ do

    atomic[$S \leftarrow S \cup \{z\} \;||\; WS \leftarrow WS \cup sucs(z) \setminus S - z$]

return S

atomic[addArc(m,n) || $WS \leftarrow WS \cup \{n \mid m \in S \wedge n \notin S\}$]