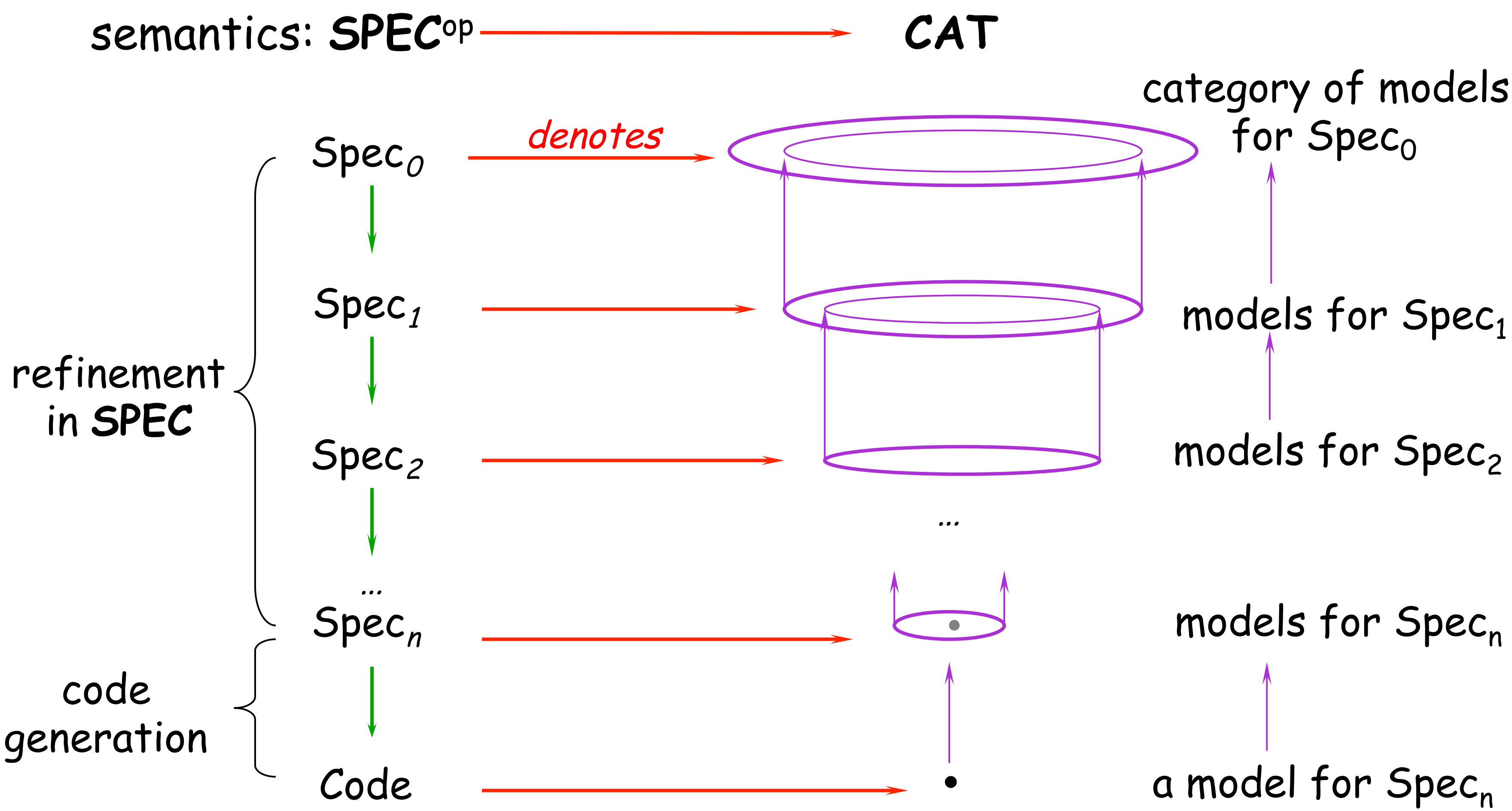
Formal Coalgebraic Specifications and their Refinement

Douglas R Smith
Stephen Westfold

Kestrel Institute
Palo Alto, California

www.kestrel.edu

Specware: Software Development by Refinement



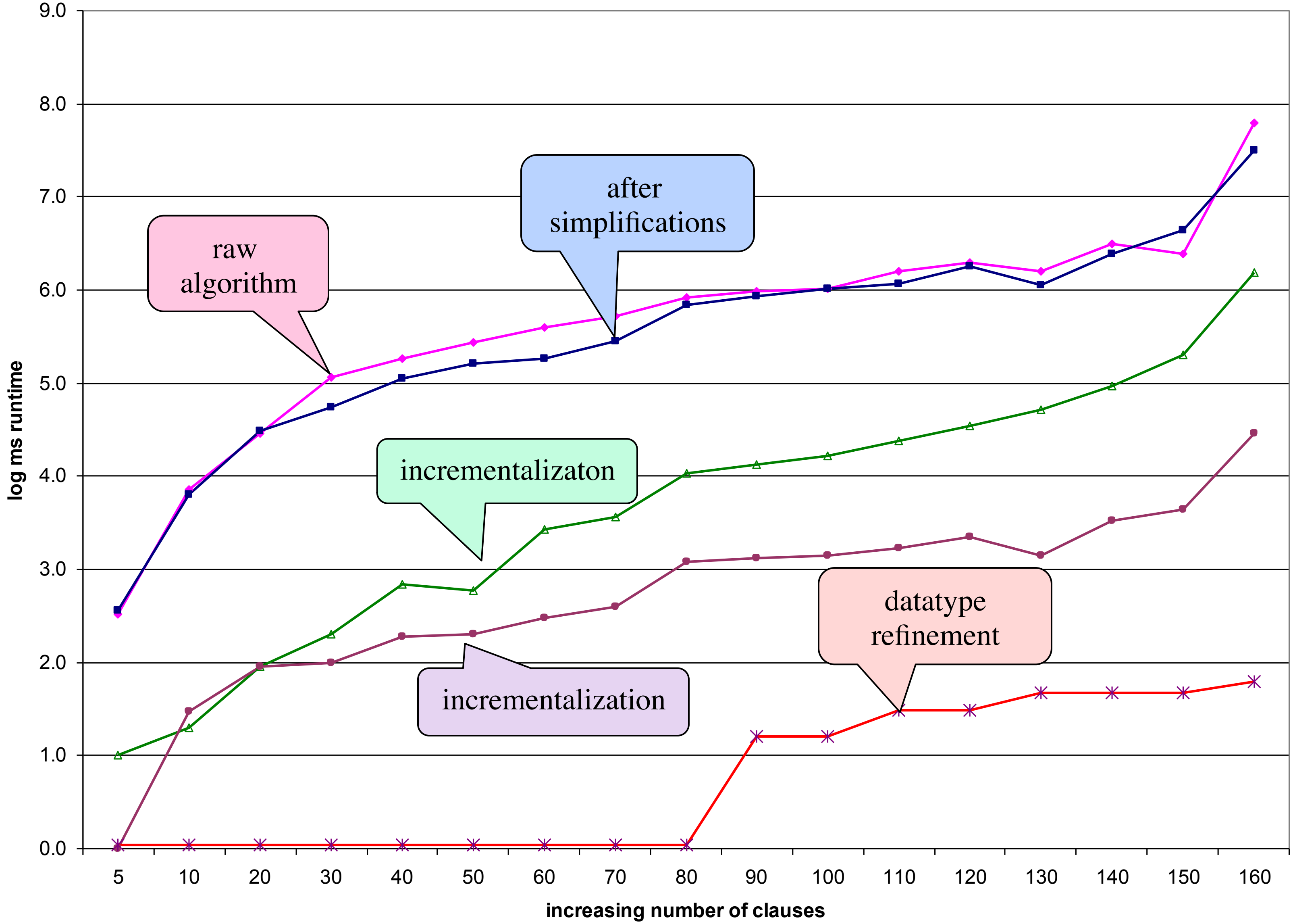
Kestrel

Refinement Sequence for Garbage Collection

C1. Algorithm Design
C2. Simplification
OM1. Observer Maintenance: WS
Mem. rename {Heap +-> Memory}
OR0. Observer Refinement of payload
OR1. Observer Refinement: tgt →tgtIM
OR1a. Observer Refinement: outNodes →outNodesIM
OR2. Observer Refinement: roots → rootsL
OM2. Observer Maintenance: rootCount
OR3. Observer Refinement: nodes → nodesPair
Mut1. Import random mutator
Mut2. Simplify
OR4. Observer Refinement: supply → supplyL
OM3. Observer Maintenance: supplyCount
OR5. Observer Refinement: black → blackCM
OR6. Observer Refinement: WS → WL → WStack
Cot1. FinalizeCoType Memory
Cot2. Define initBlackCM, …
Iso1. Type Isomorphism: Memory ↔ Memory'
DTR1. DataType Refinement: Maps → Vectors
DTR2. DataType Refinement: Stacks → Vectors
DTR3. DataType Refinement: Sets → Lists
G1. Globalize Memory
D. Simplifications
Cgen. Code Generation

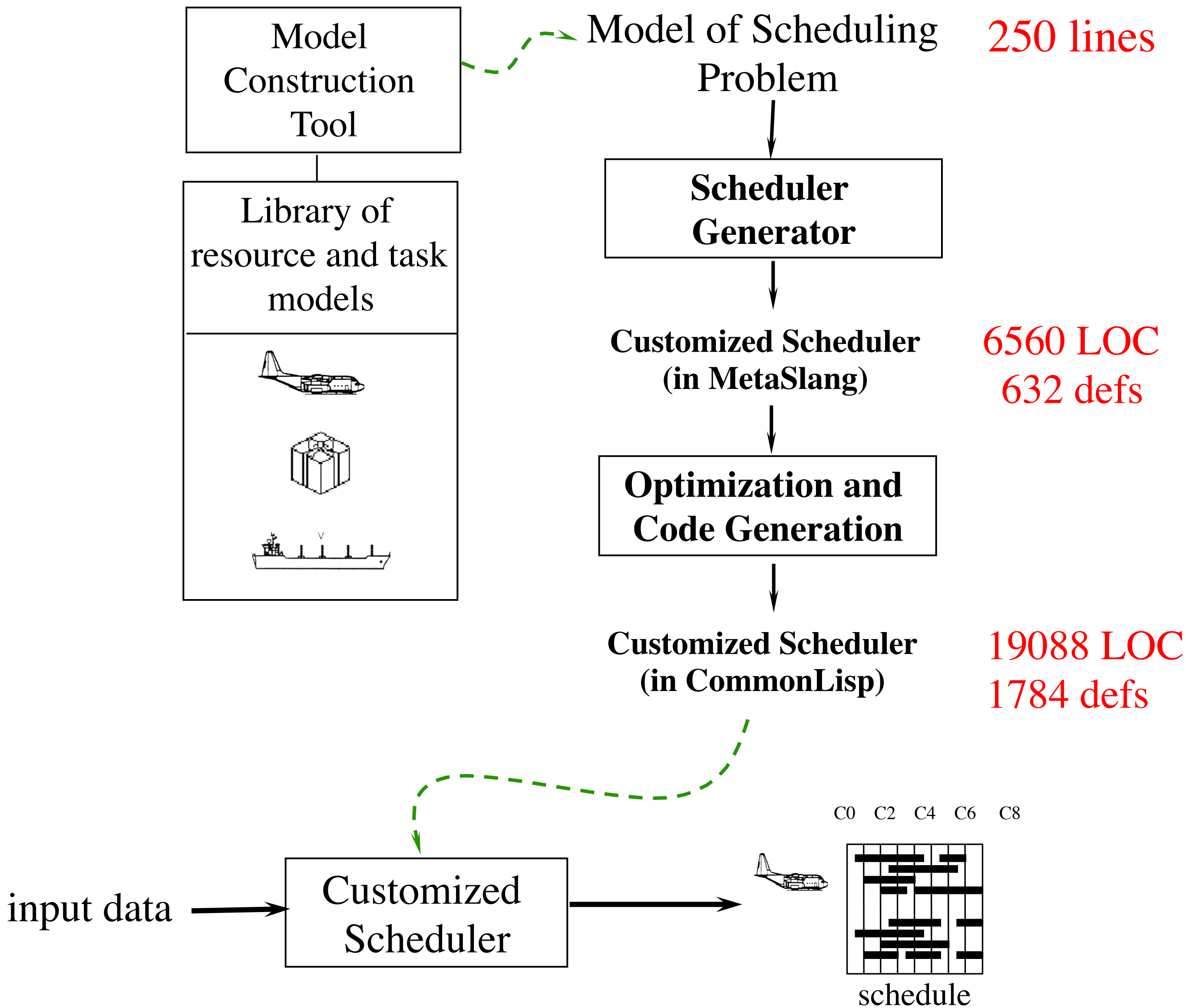
Kestrel

Ablation Study of a SAT Solver Derivation



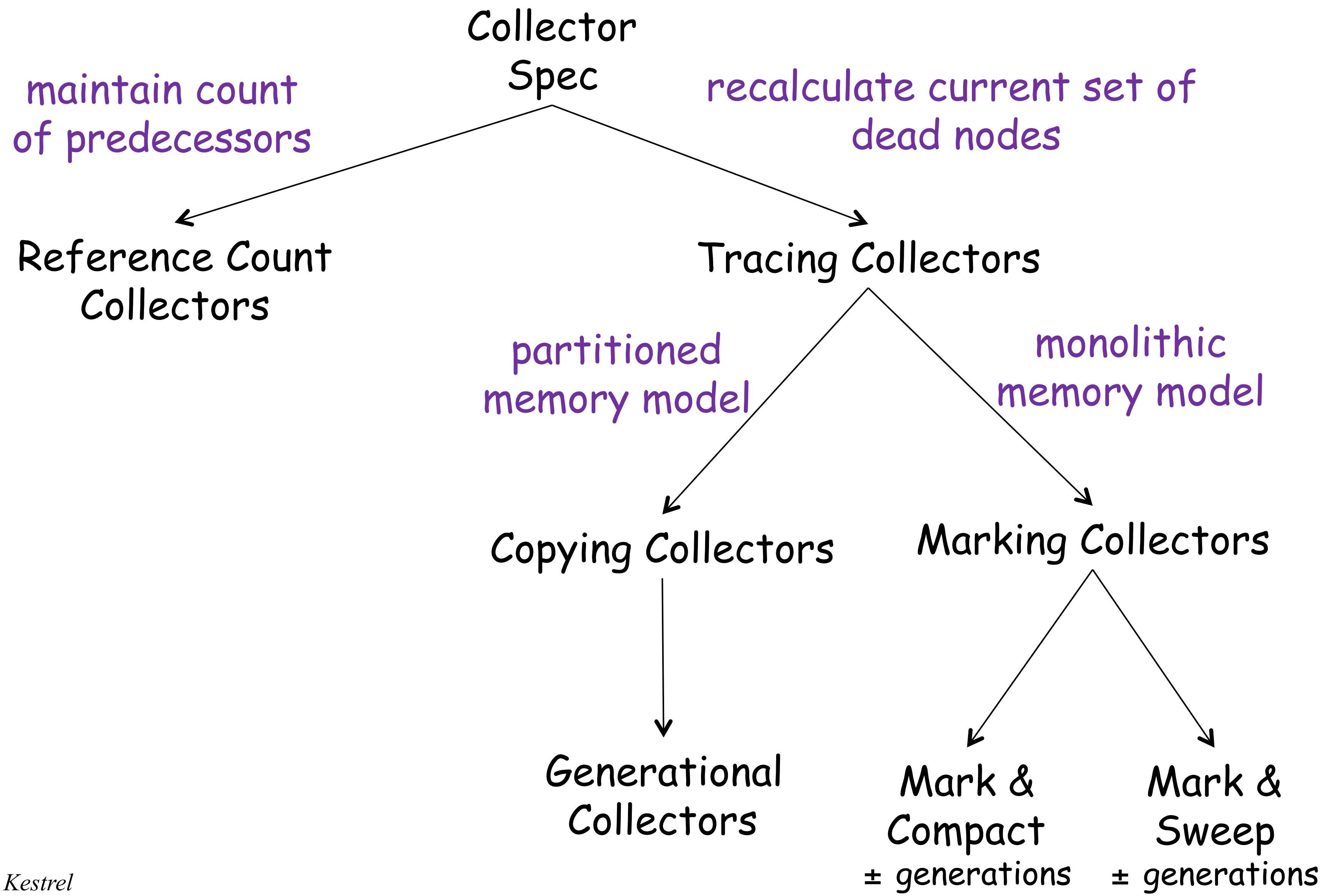
Kestrel

Planware: Synthesis of High Performance Schedulers



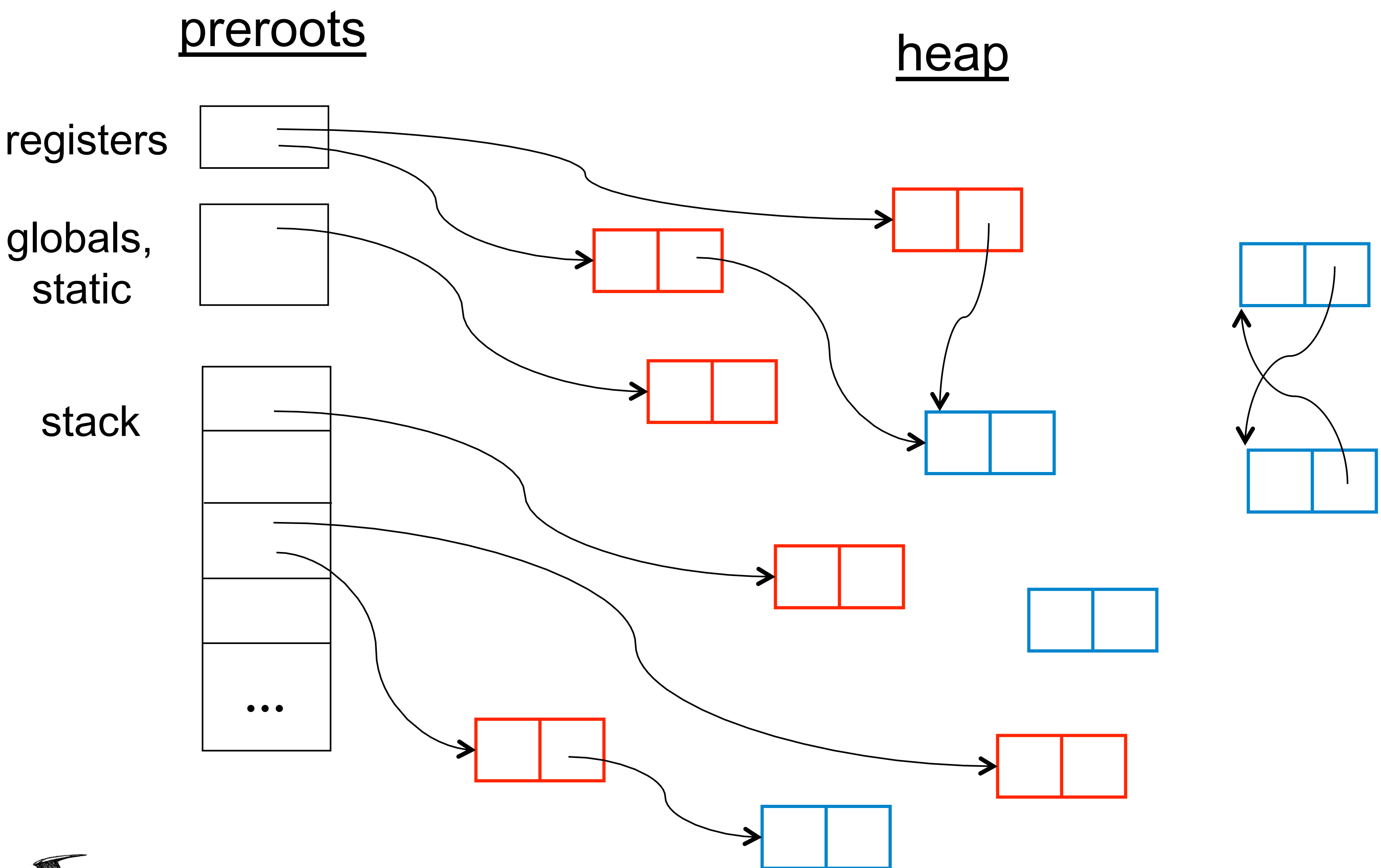
Kestrel

Deriving Common Garbage Collection Algorithms



Kestrel

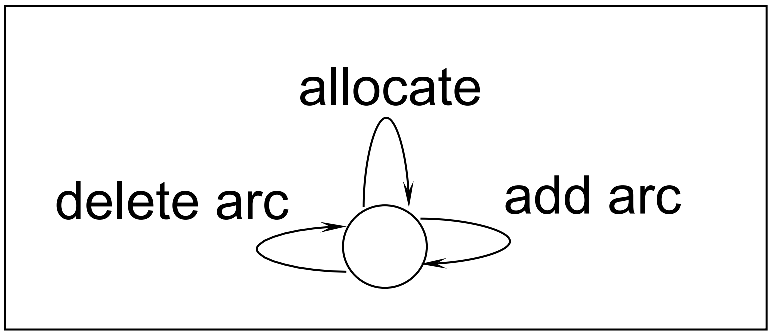
Model of Memory as a Rooted Graph



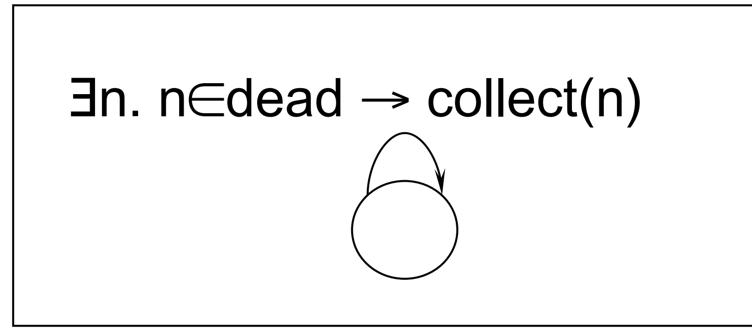
Kestrel

State Machine Models: Mutator + 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

Kestrel

Algebra and Coalgebra

an algebra is a morphism $F\ A \rightarrow A$

where F is a (commonly polynomial) functor

F provides the signature of operations of the algebra

a coalgebra is a morphism $A \rightarrow F\ A$

where F is a (commonly linear) functor

coalgebras provides a unifying treatment of

- dynamical systems
- automata
- transition systems

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
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   outArcs  : Graph → Node → Set Arc
```

Graph is specified using observers nodes and outArcs

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

addArc is specified in terms of its effect on the observers

Kestrel

Coalgebraic Specifications

- Algebraic types used for ordinary data (boolean, Nat, List)
- Coalgebraic types used for state (heaps, objects), streams
- Observers obs: State → *A*
  - basic/undefined
  - defined but maintained
  - defined but computed
- Transformers t: State → State
  - preconditions
  - postconditions: coinductive constraints on observations

Coalgebraic Refinement

- Coalgebraic types remain undefined until the last step
- Observer transformation:
  - Introduction
  - Maintenance (of a definition)
  - Refinement (to a more concrete observer)
- Transformer refinement
  - postconditions are strengthened
  - synthesized to a definition at the last step

Tracing Collectors:
Instantiated Small-Step Fixpoint Iteration

$S \leftarrow \{\}$

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

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

return S

to optimize the algorithm, we introduce a new observer:

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

Kestrel

Maintaining Observers

Observer Maintenance Transform (aka Incrementalization, Finite Differencing)

- given a defined observer

$$WS\ (G{:}Graph){:}Set\ A = \ e\ G$$

- for each transformer t, add definition to postcondition:

$$t(G{:}Graph\ \ | \qquad WS\ G = e\ G\ ):$$

$$\{G'{:}Graph \mid \ldots \wedge WS\ G' = e\ G'\ \}$$

- simplify

Maintaining Observers

type Graph
op nodes: Graph $\rightarrow$ Set Node
op outArcs : Graph $\rightarrow$ Node $\rightarrow$ Set Node
op roots : Graph $\rightarrow$ Set Node
op S : Graph $\rightarrow$ 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$\rightarrow$y)
$\wedge$ WS G′ = WS G $\cup$ {y | x$\in$S G $\wedge$ y$\notin$S G}}

design-time calculation:

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

Kestrel

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

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

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

would also automatically generate 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

Kestrel

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.

Towards A Unified Mathematical Theory for Garbage Collection,
Peter Pepper and Douglas R. Smith, in preparation

Coalgebraic Specification and Refinement
Peter Pepper, Douglas R. Smith, and Stephen J. Westfold, in prep.