Formal Derivation of Concurrent Garbage Collectors

Dusko Pavlovic[1], Peter Pepper[2], and Douglas R. Smith[1]

[1] Kestrel Institute, Palo Alto, California
{dusko,smith}@kestrel.edu
[2] Technische Universität Berlin and Fraunhofer FIRST, Berlin
pepper@cs.tu-berlin.de

Abstract. Concurrent garbage collectors are notoriously difficult to implement correctly. Previous approaches to the issue of producing correct collectors have mainly been based on posit-and-prove verification or on the application of domain-specific templates and transformations. We show how to derive the upper reaches of a family of concurrent garbage collectors by refinement from a formal specification, emphasizing the application of domain-independent design theories and transformations. A key contribution is an extension to the classical lattice-theoretic fixpoint theorems to account for the dynamics of concurrent mutation and collection.

1 Introduction

Concurrent collectors are extremely complex and error-prone. Since such collectors now form part of the trusted computing base of a large portion of the world's mission-critical software infrastructure, such unreliability is unacceptable [21]. The challenge has been to find a way to provide mathematical assurance of the correctness of concurrent collectors without doing harm to the productivity of the programmers. The latter aspect still is a major obstacle in verification-oriented systems. Interactive theorem provers may need thousands of lines of proof scripts or hundreds of lemmas in order to cope with serious collectors (see e.g. [10, 15, 4]). But also fully automated verifiers exhibit problems. As can be seen in [6], even the verification of a simplified collector necessitates such a large amount of complex properties that the specification may easily become faulty itself. The problem of assurance also has to deal with the fact that garbage collectors come in many variations, each addressing specific quality or efficiency goals. Separate verification of each variation leads to a tremendous duplication of work. On the other hand it is extremely difficult to determine for a slightly modified algorithm, which properties and proofs can remain unchanged, which are superfluous, and which need to be added or redone.

We propose to apply the approach of *specification refinement* as illustrated in Figures 1 and 2. This approach has already been successfully applied to complex problems, such as planning and scheduling tasks [19]. Figure 1 describes the way in which we come from abstract problems to concrete solutions.
(1) Suppose we have an *abstract problem description*, that is, a collection of types, operations and properties that together describe a certain problem.
(2) For this abstract problem we then develop an *abstract solution*, that is, an abstract implementation that fulfills all the requested properties.
(3) When we now have a *concrete problem* that is an instance of our abstract problem (since it meets all its properties), then we can
(4) automatically derive a *concrete solution* by instantiating the abstract solution correspondingly.
The abstract problem/solution pairs can be organized into a taxonomic library [17] in formal development environments such as KIDS [16] and Specware [7]. We will consider the abstract problem of finding fixed points in lattices or cpos and several solutions for this problem. Then we will show that garbage collection is an instance of this abstract problem by considering the concrete graphs and sets as instances of the more abstract lattices. This way our abstract solutions carry over to concrete solutions for the garbage collection problem.

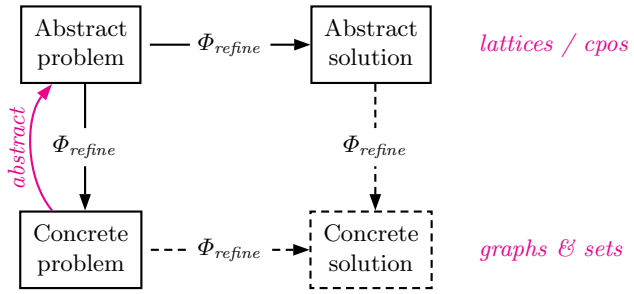


Fig. 1. Abstract and concrete problems and their solutions

Technically, all of our problem and solution descriptions are algebraic and coalgebraic specifications, which are usually underspecified and thus possess many models. "Solutions" are treated as borderline cases of such specifications, which are directly translatable into code of some given programming language. The formal connections between the various specifications are given by certain kinds of refinement morphisms, and the derivation of the concrete solution from the other parts is formally a pushout construction from category theory [3].

Figure 2 illustrates the second essential aspect of our method. We work with a family tree (or dag) of more and more refined problems, each giving rise to more and more refined solutions. On the problem side "refined" essentially means that we have additional properties, on the solution side "refined" essentially means that we have better algorithms, e.g. more efficient, more robust, more concurrent etc.

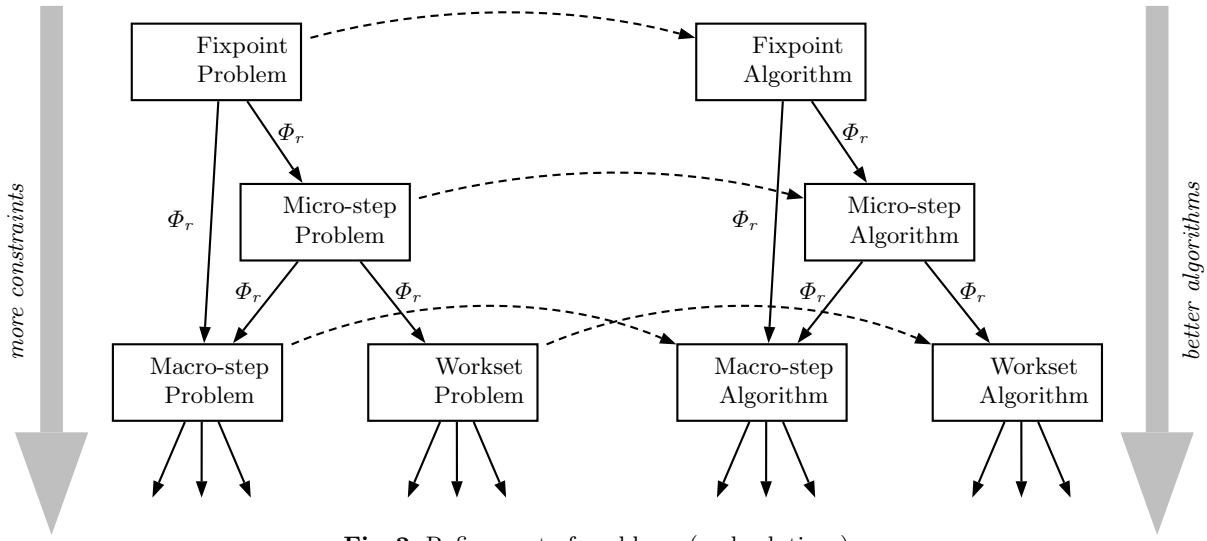


Fig. 2. Refinement of problems (and solutions)

This way of proceeding has the primary advantage that it allows us to reuse verification and development efforts. Suppose that at some point in the tree we want to design a new variation. This is reflected in a new refinement child of the current specification, to which certain properties are added. In the modified new solution we need only prove those properties that have been added; everything else is inherited.

[3] A morphism Φ from specification S to specification T is given by a type-consistent mapping of the type, function, and predicate symbols of S to derived types, functions, and predicates in T. The mapping is a specification morphism if the axioms of S translate to theorems of T. A pushout construction is used to compose specifications. More detail on the category of specifications may be found in [18, 7, 12]

Vechev et al. [21] have a similar goal of presenting a derivational treatment of a family of concurrent garbage collectors. They start from a generic algorithm, which is parameterized by an underspecified function, such that different instantiations of this function lead to different collection algorithms. A primary concern of [21] is the possibility to combine various "design dimensions" in a very flexible way. By contrast, we study the family tree of specifications and implementations that can be systematically derived using formal refinements that are largely problem-independent. We base the whole treatment of garbage collection on fundamental mathematical principles, namely lattices and fixed points. This means that the same design theories can be applied to a wide range of other problems. A key result of our studies was a generalization of classical fixed-point results of lattice theory to handle the dynamics of concurrency; i.e. iterating to a fixpoint with a monotone function that is changing over time.

The final efficiency of most practical garbage collection algorithms depends on the use of clever data representations. Standard techniques range from the classical stacks or queues to bit maps, overlayed pointers, so-called dirty bits, color toggling, concurrent local structures, and so forth. In our approach all these designs fall under the paradigm of data reification morphisms. This means that we can work throughout our developments with high-level abstract data structures such as graphs and sets in order to specify and verify the algorithmic aspects in the clearest possible way. It will be only at the end of the derivation that the high-level data structures are implemented by concrete data structures, which are chosen based on their efficiency in the given context. This step is automated in systems like Specware [7], together with many low-level optimizations. Since this is very technical and can be done almost automatically by advanced systems, we will only touch this part very briefly and sketchy here.

Full derivation of practical algorithms requires many more transformations than we can show here. We focus on the crucial initial refinement steps from a formal specification of concurrent garbage collection toward a variety of important and practical algorithms. An expanded version of this paper treats more aspects of contemporary concurrent garbage collectors [13].

2 Notes on Garbage Collection

The very first garbage collectors, which essentially go back to McCarthy's original design [9], were *stop-the-world* collectors. That is, the Mutator was completely laid to sleep, while the Collector did its recycling. This approach leads to potentially very long pauses, which are nowadays considered to be unacceptable.

The idea of having the Collector run concurrently with the Mutator goes back to the seminal papers of Dijkstra et al. [3] and Steele [20], which were followed by many other papers trying to improve the algorithm or its verification. The Doligez-Leroy-Gonthier algorithm (short: DLG) that was developed for the Concurrent CAML Light system [4, 5], is considered an important milestone, since it not only takes many practical complications of real-world collectors into account, but also generalizes from a single Mutator to many Mutators. The transition to concurrent garbage collection necessitates a *trade-off between the precision of the Collector and the degree of concurrency it provides* [21]: the higher the degree of concurrency, the more garbage nodes will be overlooked. However, this is no major concern in practice, since the escaped garbage nodes will be found in the next collection cycle.

We now illustrate the key problem that can arise in concurrent garbage collection. Figure 3 illustrates the situation at the beginning of the collection process by showing a little fragment of the store; solid nodes are reachable from the root A, dashed circles represent dead garbage nodes (the arcs of which are not drawn here for the sake of readability). We use the metaphor of "planes" to illustrate both mark-and-sweep and copying collectors. In the former, "lifting" a node to the upper plane means marking, in the latter it means copying. The picture already hints at a later generalization, where the store is partitioned into "regions".

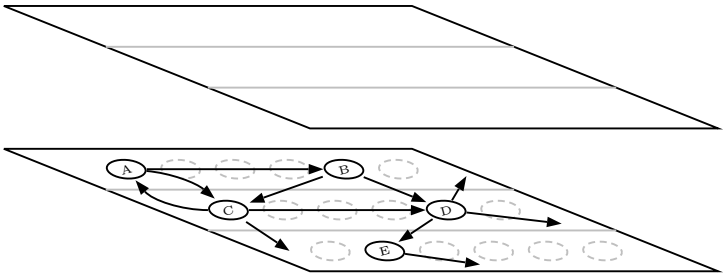


Fig. 3. At the start of the Collector

Figure 4 shows an intermediate snapshot of the algorithm. Some nodes and arcs are already lifted (i.e. marked or copied), others are still not considered. The gray nodes are in the the "workset" which means that they are marked/copied, but not all outgoing arcs have been handled yet.

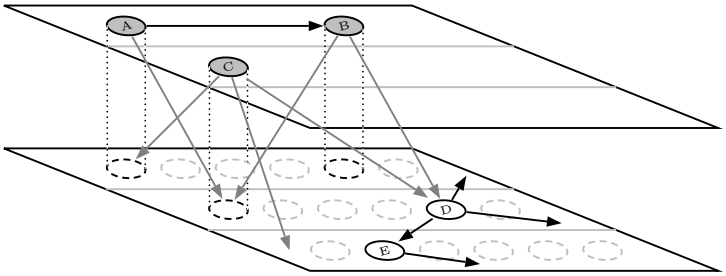


Fig. 4. A snapshot

Figure 5 shows the next snapshot. Now all direct successors of A have been treated. Therefore A is taken out of the workset, which we represent by the color black. Note that we have the invariant property that all downward arrows start in the workset. This corresponds to one of the two main invariants in the original paper of Dijkstra et al. [3].

Now let us assume that in this moment the Mutator intervenes by adding an arc $A \to E$ and then deleting the arc $D \to E$. This leads to the situation in Figure 6. Since A is no longer in the workset, its connection to E will not be detected. Hence, E is hidden from the Collector and therefore will be treated, erroneously, as a dead garbage node.

There are three reasonable ways to cope with this problem (using suitable *write barriers*):

- When performing $addArc(A, E)$, record E. (This is the approach of Dijkstra et al. [3].)
- When performing $addArc(A, E)$, record A. (This is the approach taken by Steele [20].)
- When performing $delArc(D, E)$, record E. (This is the approach taken by Yuasa [22].)

Note also that this bug may appear in an even subtler way *during* the handling of a node in the workset. Consider the node C in Figure 5 and suppose that it has lifted the first two of its three arcs. At this moment the Mutator redirects the first pointer field to, say, E. But a naive Collector will nevertheless take node C out of the workset (color it black) when its final arc has been treated.

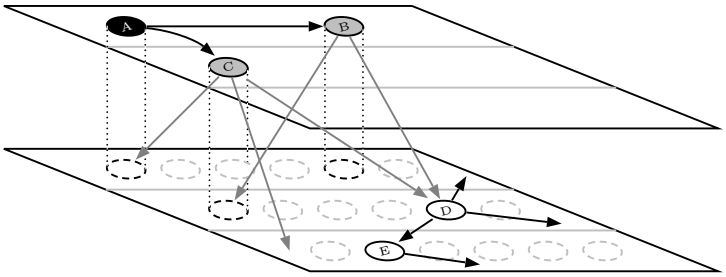


Fig. 5. The next snapshot

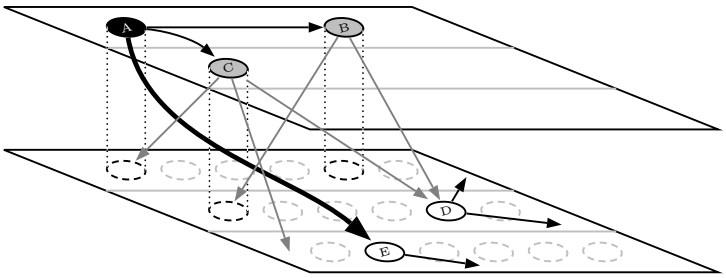


Fig. 6. A subtle error

2.1 Architecture and Basic Terminology

We modularize the problem by way of three kinds of components; see Figure 7. The *Mutators* represent the activities of all programs that use the heap. These activities base on primitive operations that are provided by the component *Store*, which represents the memory management system (as part of the runtime system or operating system). Finally the task of the garbage collection is performed by a component *Collector*.

The *Mutator* operates on a *graph*, which is a data structure of type $Graph(Node, Arc)$. It can essentially perform three primitive operations:[4]

- $addArc(a, b)$: add a new arc node a to node b.
- $delArc(a, b)$: delete the arc between a and b. This may have the effect that b and other nodes reachable from b become unreachable ("garbage").
- $addNew(a)$: allocate a new node b (from the freelist) and attach it by an arc from a. This reflects the fact that in reality *alloc* operations return a pointer, which is stored in some field (variable, register, heap cell) of the Mutator. Hence, the new node is immediately linked to the Mutator's graph.

The *Store* provides the low-level interface to the actual memory-access operations. We distinguish the following sets:

- *active* are those nodes that constitute the Mutator's graph.
- *supply* are the nodes in the freelist. (They become *active* through the operation *addNew*.)
- *live* is a shorthand for the union of the *active* and *supply* nodes.

[4] This considerably simplifies the memory model used in the DLG algorithm [4, 5], where the Mutator has eight operations. However, the essence of these operations is captured by our three operations.

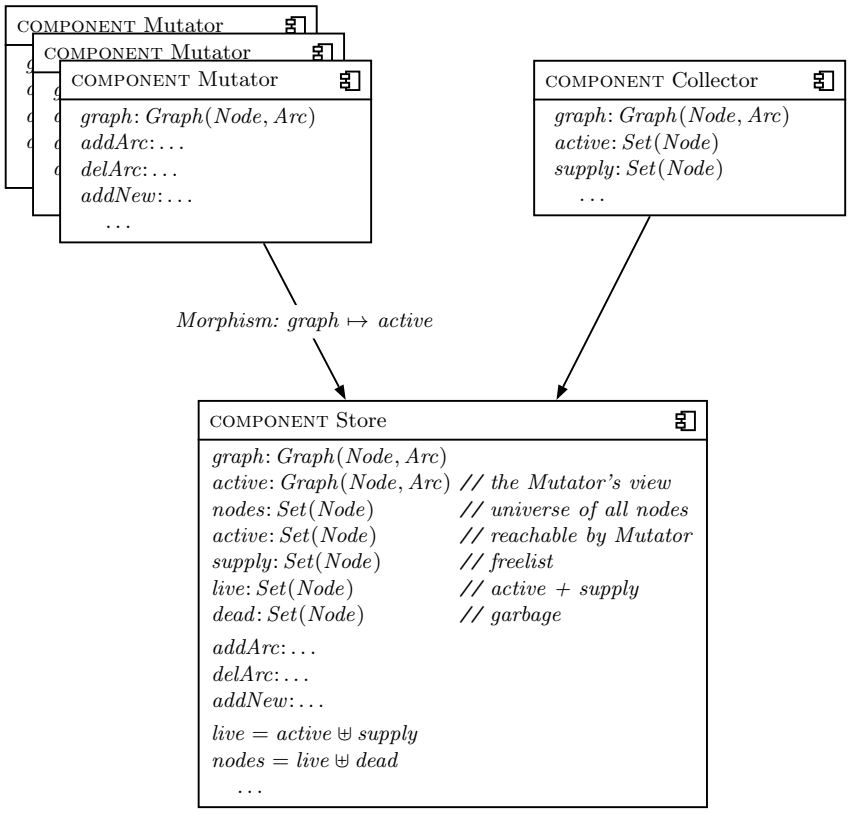


Fig. 7. The system architecture

– *dead* are the garbage nodes that are neither reachable from the Mutator nor in the freelist. (Nodes may become *dead* through the operation *delArc*.)

Note that the specifications in Figure 7 use $A = B \uplus C$ as a shorthand notation for the two properties $A = B \cup C$ and $B \cap C = \emptyset$. They also use overloading of operation names. For example *active* is used both for the subgraph that constitutes the Mutator's view and for the set of nodes in this subgraph. Such overloaded symbols must always be distinguishable from their context. Note also that we frequently refer to the "set" *Arcs* of the arcs of a graph and also to the "set" $sucs(a)$ of all successors of a node a; but these are actually *multisets*, since two nodes may be connected by several arcs. Technically, the cell has several slots that all point to the same cell.

The Mutator's operations *addArc*, *delArc*, *addNew* have an invariant property that is decisive for the working of any kind of garbage collector: *being garbage is a stable property* [1].

Proposition 1 (Antitonicity of Mutator). *A Mutator can only access the nodes in its graph and the freelist; that is, it can never access garbage nodes. In other words, the set of live nodes (graph + freelist) monotonically* decreases.

2.2 The Fundamental Specification of Garbage Collection

Surprisingly often papers on garbage collection refer to an intuitive understanding of what the Collector shall achieve. But in a formal treatment we cannot rely on intuition; rather we have to be absolutely precise

about the goal that we want to achieve. Consider the architecture sketched in Figure 7. The Mutator continuously performs its basic operations *addArc*, *delArc* and *addNew*, which – from the Mutators viewpoint – are all considered to be total functions; i.e. they return a defined value on all inputs. This is trivially so for *addArc* and *delArc*, since their arguments exist in the mutators graph. The problematic operation is *addNew*, since this operation needs an element from the freelist. However, the freelist may be empty (i.e. $supply = \emptyset$). In this situation there are two possibilities:

1. $|\, active| = MemorySize$. That is, the Mutator has used all available memory in its graph. Then nothing can be done!
2. $|\, active| < MemorySize$. When $supply = \emptyset$, this means that $dead \neq \emptyset$. This is the situation in which we want to recycle garbage cells into the freelist. And this is the Collector's reason for existence!

Based on this reasoning, we obtain two basic principles for the Mutator/Collector paradigm.

Assumption 1 (Boundedness of Mutator's graph) $|\, Mutator.graph| < MemorySize$

Under this global assumption the Collector has to ensure that the operation *addNew* is a total function (which may at most be delayed). This can be cast into a temporal-logic formula:

Goal 2 (Specification of Collector) $\Box \Diamond\ supply \neq \emptyset$ *(provided assumption 1 holds)*

This is a liveness property stating that "at any point in time the freelist (may be empty but) will eventually be nonempty." When this condition is violated, that is, $supply = \emptyset$, then it follows by the global Assumption 1 that $dead \neq \emptyset$. Hence the Collector has to find at least *some* dead nodes, which it can then transfer to the freelist. This can be cast into an operation *recycle* with the initial specification given in Figure 8.

SPEC Collector	
$recycle\colon Graph(Node, Arc) \to Set(Node)$	
$\emptyset \subset recycle(G) \subseteq dead$	*if* $dead \neq \emptyset$
$\emptyset = recycle(G)$	*if* $dead = \emptyset$

Fig. 8. The Collector's task

Hence we should design the system's working such that the following property holds (using an ad-hoc notation for transitions).

Goal 3 (Required actions of Collector) $\Box \Diamond \left(supply \longrightarrow supply \uplus recycle(G)\right)$

When Goal 3 is met, then the original Goal 2 is also guaranteed to hold. In other words, the collector has to periodically call *recycle* and add the found subset of the garbage nodes to the freelist.

Note that the above operation can happen at any point in time; we need not wait until the freelist is indeed empty. This observation leaves considerable freedom for optimized implementations which are all correct.

2.3 How to Find Dead Nodes

Unfortunately, the specification of *recycle* in Figure 8 is not easily implementable since the dead nodes are not directly recognizable. Since the *dead* nodes are the complement of the *live* nodes; i.e. $live = \complement\, dead = nodes \backslash live$, the idea comes to mind to work with the complement of *recycle*. This leads to the simple calculation

$$\begin{array}{l} \emptyset \subset \mathit{recycle}(G) \subseteq \mathit{dead} \\ \Leftrightarrow \complement\,\emptyset \supset \complement\,\mathit{recycle}(G) \supseteq \complement\,\mathit{dead} \\ \Leftrightarrow \mathit{nodes} \supset \complement\,\mathit{recycle}(G) \supseteq \mathit{live} \\ \Leftrightarrow \mathit{nodes} \supset \mathit{trace}(G) \supseteq \mathit{live} \end{array}$$

where we introduce a new function $\mathit{trace}(G) = \complement\,\mathit{recycle}(G)$. This leads to the refined version of the Collector's specification in Figure 9. Note that this specification, which will form the starting point for our more detailed derivation, is *formally derived* from the fundamental requirements for garbage collection as expressed in Assumption 1 and Goal 2 above!

SPEC Collector	
$\mathit{recycle}\colon \mathit{Graph}(\mathit{Node}, \mathit{Arc}) \to \mathit{Set}(\mathit{Node})$	
$\mathit{trace}\colon \mathit{Graph}(\mathit{Node}, \mathit{Arc}) \to \mathit{Set}(\mathit{Node})$	
$\mathit{recycle}(G) = \complement\,\mathit{trace}(G)$	
$\mathit{live} \subseteq \mathit{trace}(G) \subset \mathit{nodes}$	*if* $\mathit{dead} \neq \emptyset$
$\mathit{trace}(G) = \mathit{nodes}$	*if* $\mathit{dead} = \emptyset$

Fig. 9. The Collector's task (first refinement)

3 Mathematical Foundation: Fixed Points

In garbage collection one can roughly distinguish two classes of collectors:

- *Stop-the-world collectors*: these are the classical non-concurrent collectors, where the mutators need to be stopped while the collector works.
- *Concurrent collectors*: these are the collectors that allow the mutators to keep working concurrently with the collector (except for very short pauses).

3.1 Classical Fixed Points (Stop-the-world Collectors)

Computing the set of garbage nodes in a stop-the-world collector can be treated as a classical fixpoint computation in a finite powerset lattice. We briefly review the basic concepts and then show how to calculate the overall structure of a marking algorithm.

- For a set $s = \{ x_0, x_1, x_2, \ldots\}$ of type $\mathit{Set}(A)$ and a function $f\colon A \to A$ we use the overloaded function $f\colon \mathit{Set}(A) \to \mathit{Set}(A)$ by writing $f(s)$ as a shorthand for $\{f(x_0), f(x_1), f(x_2), \ldots\}$.
- A function $f\colon A \to A$ is *monotone*, if $x \leq y \Rightarrow f(x) \leq f(y)$ holds.
- The function f is *continuous*, if $f(\sqcup\{x_0, x_1, x_2, \ldots\}) = \sqcup\{f(x_0), f(x_1), f(x_2), \ldots\}$ holds.
- The function f is *inflationary* in x, if $x \leq f(x)$ holds.
- The element x is called a *fixed point* of f, if $x = f(x)$ holds; x is the *least fixed point*, if $x \leq y$ for any other fixed point y of f.
- The element x is called a *fixed point* of f *relative* to r, if $x = f(x) \wedge r \leq x$ holds.
- By $\widehat{f}(x) = \text{LEAST } u.\ u = f(u) \wedge x \leq u$ we denote the reflexive-transitive *closure* of f (when it exists); i.e. the function that yields the least fixed point of f relative to x.

Lemma 4 (Properties of the closure $\widehat{f}$). *The closure $\widehat{f}(x)$ has a number of properties that we will utilize frequently:*

- $x \leq \widehat{f}(x)$ *(inflationary);*
- $\widehat{f}(\widehat{f}(x)) = \widehat{f}(x)$ *(idempotent);*
- $f(\widehat{f}(x)) = \widehat{f}(x)$ *(fixpoint);*
- $\widehat{f}(f(x)) = \widehat{f}(x)$ *if* $x \leq f(x)$

Theorem 1 (Kleene[8]). *For a* continuous *function f the least fixed point x is obtained as the least upper bound of the Kleene chain:*

$$x = \sqcup\{\, \bot, f(\bot), f^2(\bot), f^3(\bot), \ldots\}$$

where $\bot$ is the bottom element of the lattice.

It has been shown that the essence of these theorems also holds in the simpler structure of *complete partial orders (cpos)*[5]. Cai and Paige [2] present a number of generalizations of Theorem 1 that are streamlined towards practical algorithmic implementations of fixpoint computations.

Theorem 2 (Cai-Paige). *Let A be a cpo and $f\colon A \to A$ be a monotone function that is inflationary in r. Let $\{s_0, s_1, s_2, \ldots, s_n\}$ be an arbitrary sequence obeying the conditions*

$$\begin{gathered} r = s_0 \\ s_i < s_{i+1} \leq f(s_i) \quad \text{for } i < n \\ s_n = f(s_n) \end{gathered}$$

then s_n is the least fixed point *of f relative to r. Conversely, when the least fixed point is finitely computable, then the sequence will lead to such an s_n.*

Theorem 2 provides a natural abstraction from workset-based iterative algorithms, which maintain a workset of change items. At each iteration, a change item is selected and used to generate the next element of the iteration sequence. The incremental changes tend to be small and localized, hence this is called the *micro-step* approach, and the Kleene chain the *macro-step* approach [14]. All practical collectors use a workset that records nodes that await marking.

Corollary 1 (Invariance of closure). *The elements of the set $\{\, s_0 < s_1 < s_2 < \ldots < s_n \,\}$ all have the same closure: $\widehat{f}(s_i) = \widehat{f}(r)$.*

Using these basic results, we derive the overall structure of a marking algorithm for a stop-the-world collector. The essence of it is the iterative algorithm for finding garbage nodes to recycle.

Letting *roots* denote the roots of the active graph together with the head of the *supply* list, we have $live = \widehat{f}(roots)$ where $f(R) = \{b \mid b \in G.sucs(a) \ \ \&\ \ a \in R\}$; in words, the *active* nodes are the closure of the roots under the successor function in the current graph G.

To derive an algorithm for computing the *dead* nodes, we calculate as follows:

[5] A *cpo* is a partial order in which every directed subset has a supremum.

$$
\begin{array}{lll}
& \mathit{dead} & \\
= & \complement\, \mathit{live} & \text{definition} \\
= & \complement\, \widehat{f}(\mathit{roots}) & \text{definition} \\
= & \check{g}(\mathit{roots}) & \text{using the law } \complement\, \widehat{h}(R) \;=\; \check{i}(R) \text{ where } i(x) = \complement h(\complement\, x)
\end{array}
$$

where $\check{g}(R)$ is the greatest fixpoint of the monotone function

$$g(x) = \mathit{nodes} \setminus (\mathit{roots} \cup \{b \mid b \in \mathit{sucs}(a) \;\&\; a \in \mathit{nodes} \setminus x\}).$$

This allows us to produce a correct, but naive iterative algorithm to compute dead nodes via a Kleene chain.

Program 1 Raw Fixpoint Iteration Algorithm

```
1  W ← h.nodes;
2  while W ≠ g(W) do W ← g(W)
3  return W
```

Following Cai and Paige [2], we can construct a more efficient fixpoint iteration algorithm using a workset defined by

$$WS \;=\; X \setminus g(X).$$

Although this workset definition is created by instantiating a problem-independent scheme, it has an intuitive meaning: the workset is the set of nodes whose parents have been "marked" as live, but who themselves have not yet been marked. The workset expression can be simplified as follows

$$
\begin{array}{ll}
& X \setminus g(X) \\
= & \quad \{ \text{ Definition } \} \\
& X \setminus (\mathit{nodes} \setminus (\mathit{roots} \cup \{b \mid b \in \mathit{sucs}(a) \;\&\; a \in \mathit{nodes} \setminus X\})) \\
= & \quad \{ \text{ Using the law } A \setminus (B \cup C) \;=\; (A \setminus B) \setminus C \;\} \\
& X \setminus ((\mathit{nodes} \setminus \mathit{roots}) \setminus \{b \mid b \in \mathit{sucs}(a) \;\&\; a \in \mathit{nodes} \setminus X\}) \\
= & \quad \{ \text{ Using the law } A \setminus (B \setminus C) \;=\; (A \setminus B) \bigcup (A \cap C) \;\} \\
& (X \setminus (\mathit{nodes} \setminus \mathit{roots})) \bigcup (\{b \mid b \in \mathit{sucs}(a) \;\&\; a \in \mathit{nodes} \setminus X\} \cap X) \\
= & \quad \{ \text{ Using the law } \{x|P(x)\} \cap Q \;=\; \{x|P(x) \wedge x \in Q\} \;\} \\
& (X \setminus (\mathit{nodes} \setminus \mathit{roots})) \bigcup \{b \mid b \in \mathit{sucs}(a) \;\&\; b \in X \;\&\; a \in \mathit{nodes} \setminus X\} \\
= & \quad \{ \text{ Again using the law } A \setminus (B \setminus C) \;=\; (A \setminus B) \bigcup (A \cap C) \text{ (on first term) } \} \\
& (X \setminus \mathit{nodes}) \cup (X \cap \mathit{roots}) \bigcup \{b \mid b \in \mathit{sucs}(a) \;\&\; b \in X \;\&\; a \in \mathit{nodes} \setminus X\} \\
= & \quad \{ \text{ Simplifying } \} \\
& \{\} \cup (X \cap \mathit{roots}) \bigcup \{b \mid b \in \mathit{sucs}(a) \;\&\; b \in X \;\&\; a \in \mathit{nodes} \setminus X\}
\end{array}
$$

$=$ { Simplifying}

$$(X \cap roots) \bigcup \{b \mid b \in succs(a) \ \& \ b \in X \ \& \ a \in nodes \setminus X\}.$$

The greatest fixpoint expression can be computed by the workset-based Program 2 justified by Theorem 2.

Program 2 Workset-based Fixpoint Iteration Program

```
1  W ← nodes;
2  while ∃z ∈ ((W ∩ roots) ⋃ {b | b ∈ succs(a) & b ∈ W & a ∈ nodes \ W})
3        W ← W − z
4  return W
```

To improve the performance of this algorithm, we apply the Finite Differencing transformation [11] and incrementally maintain the invariant

$$WS = (W \cap roots) \bigcup \{b \mid b \in succs(a) \ \& \ b \in W \ \& \ a \in nodes \setminus W\}.$$

The calculations to enforce the invariant (detailed in [13]) result in the code is shown in Program 3, where concurrent assignment is used to update both W and the workset WS.

Program 3 Optimized Fixpoint Iteration Algorithm

```
invariant WS = (W ∩ roots) ⋃ {b | b ∈ succs(a) & b ∈ W & a ∈ nodes \ W}      1
W, WS := nodes, roots;                                                       2
while ∃z ∈ WS do                                                             3
      W, WS := W − z, WS ⋃ {b | b ∈ succs(z) & b ∈ W}) − z                    4
output W.                                                                    5
```

Program 3 represents the abstract structure of most marking algorithms. Our point is that its derivation, and further steps toward implementation, are carried out by generic, problem-independent transformations, supported by domain-specific simplifications, as above. Further progress toward a detailed implementation requires a variety of other transformations, including finite differencing, simplification, and datatype refinements. For example, the finite set W may be implemented by a characteristic function, which in turn is refined to a bit array, or concurrent data structures for local buffers or work-stealing queues.

3.2 Fixed Points in Dynamic Settings (Concurrent Collectors)

The classical fixed-point considerations work with a fixed monotone function f. In the garbage collection application this is justified as long as the graph, on which the collector works, remains fixed during the collector's activities. But as soon as the mutator is working in parallel with the collector, the graph keeps changing, while the collector is active. This can be modeled by considering a sequence of graphs $G_0, G_1, G_2, \ldots$ and by making the function f dependent on these graphs: $f(G_0)(\ldots), f(G_1)(\ldots), f(G_2)(\ldots), \ldots,$ where f: $Graph \to Set(Node) \to Set(Node)$ and

$$f(G)(S) = S \bigcup \{b \mid a \in S \ \& \ b \in G.succs(a)\}.$$

Intuitively, f extends a given set of nodes with the set of their successors in the graph. To ease readability we omit the explicit reference to the graphs and simply write f_0, f_1, f_2, ... Using this notational liberty the specification of the underlying foundation is stated in Figure 10: the f_i are monotone ① and inflationary in r ②. Moreover the closure-forming operator $\widehat{f}$ is defined by ③.

SPEC Foundation		
EXTEND *Cpo*(A)		*// A is a cpo (alternatively: lattice)*
$f_0, f_1, f_2, \ldots : A \to A$		*// sequence of functions*
$\widehat{\_} : A \to A \to A \to A$		*// $\widehat{f}$ is reflexive-transitive closure of f*
$r : A$		*// "root"*
$x \leq y \Rightarrow f_i(x) \leq f_i(y)$	①	*// all f_i are monotone*
$r \leq f_i(r)$	②	*// all f_i are inflationary in r*
$\widehat{f}(x) =$ LEAST s: $x \leq s \wedge s = f(s)$	③	*// closure (computes least fixed point)*

Fig. 10. Initial Specification

Based on this foundation we can now formulate our goal. Recall the specification of the garbage collection task given by *Collector* in Figure 9: $live \subseteq trace(G) \subset nodes$. This translates into our dynamic setting as $live_n \subseteq s \subset nodes$. We add as a working hypothesis that the set $live_0$ serves as an upper bound that we will need to guarantee in our dynamic algorithm: $live_n \subseteq s \subseteq live_0 \subset nodes$. The set $live_0$ is sometimes called the "snapshot-at-the-beginning" [1]. Since in our abstract setting $live_n$ corresponds to the closure $\widehat{f_n}(r)$ and $live_0$ corresponds to the closure $\widehat{f_0}(r)$, we immediately obtain the abstract formulation ⑤ of our problem statement (Figure 11).

SPEC Fixpoint-Problem		
EXTEND *Foundation*		
$r \leq x \Rightarrow f_{i+1}(\widehat{f_i}(x)) \leq \widehat{f_i}(x)$	④	*// garbage can only grow*
THM $\exists\, n, s$: $\widehat{f_n}(r) \leq s \leq \widehat{f_0}(r)$	⑤	*// $live_n \leq s \leq live_0$*
THM $r \leq x \Rightarrow \widehat{f_0}(x) \geq \widehat{f_1}(x) \geq \widehat{f_2}(x) \geq \ldots$	⑥	*// Lemma 5*

Fig. 11. Initial Specification

Axiom ④ is the abstract counterpart of the fundamental Proposition 1: the set of live nodes is monotonically decreasing over time, or, dually, garbage increases monotonically. For proof-technical reasons we have to conditionalize this property to any set x containing the roots r.)

Note that the existential formula ⑤ is trivially provable by setting $n = 0$ and $s = \widehat{f_0}(r)$. Actually the property ⑥ (see Lemma 5 below) shows that such an s exists for any n. However, our actual task will be to come up with a *constructive algorithm* that yields such an n and s.

For the specification *FixpointProblem* we can prove the property ⑥ (i.e. Lemma 5) that will be needed later on. This monotonic decreasing of the closure is in accordance with our intuitive perception of the Mutator's activities. The operation *delArc* may lead to fewer live nodes. And the operations *addArc* and *addNew* do not change the set of live nodes (since the freelist is part of the live nodes).

Lemma 5 (Antitonicity of closure). *The closures are monotonically decreasing:*

$$\textit{For} \quad r \leq x \quad \textit{we have} \quad \widehat{f}_0(x) \geq \widehat{f}_1(x) \geq \widehat{f}_2(x) \geq \ldots \qquad (6)$$

Proof: We use a more general formulation of this lemma: For monotone g and h we have the property

$$\forall x\colon\ g(\widehat{h}(x)) \leq \widehat{h}(x) \Rightarrow \widehat{g}(x) \leq \widehat{h}(x)$$

We show by induction that $\forall i\colon\ g^i(x) \leq \widehat{h}(x)$. Initially we have $g^0(x) = x \leq \widehat{h}(x)$ due to the general reflexivity property (3) of the closure. The induction step uses the induction hypothesis and then the premise: $g^{i+1}(x) = g(g^i(x)) \leq g(\widehat{h}(x)) \leq \widehat{h}(x)$. By instantiating f_{i+1} for g and f_i for h we immediately obtain $\widehat{f}_{i+1}(x) \leq \widehat{f}_i(x)$ by using the axiom (4), when $r \leq x$. *(End of proof)*

3.3 The *Microstep* Refinement

In order to get closer to constructive solutions we perform our first essential refinement. Generalizing the idea of Cai and Paige in Theorem 2, we add further properties to our specification, resulting in the new specification of Figure 12. Note that we now use some member s_n of the sequence $s_0, s_1, s_2, \ldots$ as a witness for the existentially quantified s.

SPEC Micro-Step		
EXTEND *FixpointProblem*		
$s_0, s_1, s_2, \ldots : A$		*// sequence of approximations*
$s_0 = r$	(7)	*// start with "root"*
$s_i < s_{i+1} \leq f_i(s_i) \vee s_i = f_i(s_i)$	(8)	*// computation step*
THM $\exists\, n\colon\ \widehat{f}_n(r) \leq s_n \leq \widehat{f}_0(r)$	(9)	*// to be shown below*
THM $\widehat{f}_0(s_0) \geq \widehat{f}_1(s_1) \geq \ldots \geq \widehat{f}_n(s_n)$	(10)	*// Lemma 6 below*

Fig. 12. The "micro-step approach"

Proof of property (9) : In a finite lattice the s_i cannot grow forever. Therefore there must be a fixpoint $s_n = f_n(s_n)$ due to axiom (8). Then the left half of the proof of (9) follows trivially from monotonicity:

$\forall i\colon\ r \leq s_i$	*// axiom (7) and (8)*
$\vdash\ \ r \leq s_n = f_n(s_n)$	*// s_n is fixpoint*
$\vdash\ \ \widehat{f}_n(r) \leq \widehat{f}_n(f_n(s_n)) = \widehat{f}_n(s_n) = s_n$	*// properties of $\widehat{f}_n$ Lemma 4*

The right half $s_n \leq \widehat{f}_0(r)$ is a direct consequence of the following Lemma 6. *(End of proof)*

Lemma 6 (Decreasing Closures). *As a variation of Lemma 5 we can show property (10): the closures are **de**creasing, even when applied to the **in**creasing s_i:*

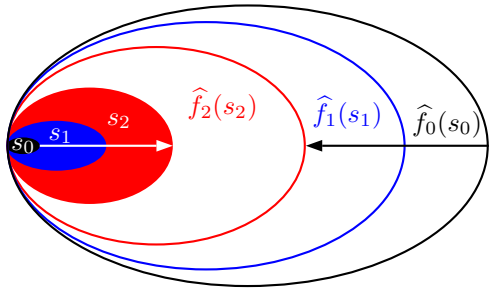
$$\forall i\colon\ \widehat{f}_{i+1}(s_{i+1}) \leq \widehat{f}_i(s_i)$$

Proof: On the basis of Lemma 5 (property (6) in Figure 11) the proof follows directly from axiom (8) by monotonicity:

$s_{i+1} \leq f_i(s_i)$ // *axiom* (8)
$\vdash \quad \widehat{f}_{i+1}(s_{i+1}) \leq \widehat{f}_{i+1}(f_i(s_i)) \leq \widehat{f}_i(f_i(s_i)) = \widehat{f}_i(s_i)$ // *monotonicity of* $\widehat{f}_{i+1}$; (6)

Note that (6) is applicable here, since – due to (8) – $r \leq f_i(s_i)$ holds. *(End of proof)*

Lemma 6 may be depicted as follows:



where the approximations s_0, s_1, s_2, ... keep growing, while their closures $\widehat{f}_0(s_0)$, $\widehat{f}_1(s_1)$, $\widehat{f}_2(s_2)$, ... keep shrinking.

This essentially concludes the derivation that can reasonably be done on the abstract mathematical level of fixed points and lattices.

4 Garbage Collection in Dynamic Graphs

We now take specific properties of garbage collection into account, but still on the semi-abstract level of sets and graphs. First we note that our specification of garbage collection using sets and set inclusion is a trivial instance of the lattice-oriented specification in the previous section. Therefore all results carry over to the concrete problem. The morphism is essentially defined by the following map:

$$\Phi = \begin{bmatrix} A & \mapsto & Set(Node) \\ \leq & \mapsto & \subseteq \\ f_i(s) & \mapsto & f(G_i)(s) = s \cup G_i.succs(s) = s \cup \bigcup_{a \in s} G_i.succs(a) \\ r & \mapsto & G_0.roots \end{bmatrix}$$

- The basis now is a sequence of graphs G_0, G_1, G_2, ... which are due to the activities of the Mutator.
- The function $f(G_i)(s) = s \cup \bigcup_{a \in s} G_i.succs(a)$ adds to the set s all its direct successors. (We will retain the shorthand notation $f_i = f(G_i)$ in the following.)

Figure 13 illustrates the road map through our essential refinements. The left half shows the refinements that have been performed in the previous Section 3 on the abstract mathematical level of lattices and fixed points. The right half shows the refinements on the semi-abstract level of graphs and sets that will be presented in this section.

Lemma 7 (Morphism abstract → concrete). *Under the morphism Φ, all axioms of the abstract specifications Foundation, FixpointProblem and MicroStep hold for the more concrete specifications of graphs and sets (see Figure 13).*

Proof: We show the three morphism properties Φ_1, Φ_2, Φ_3 in turn.

Φ_1: The proof is trivial, since the monotonicity axiom (1) is a direct consequence of the definition of $\Phi(f_i)$. Axiom (3) is just a definition.

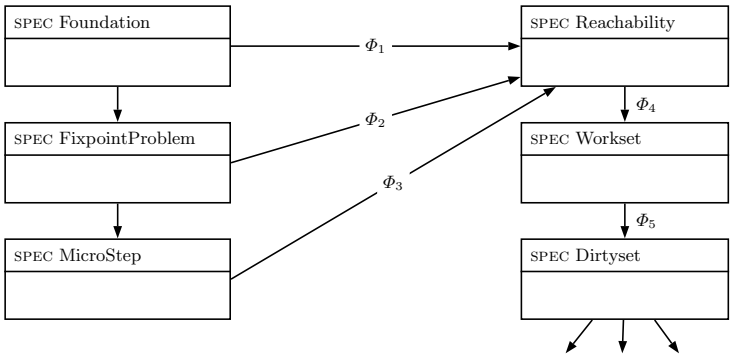


Fig. 13. Roadmap of refinements

Φ_2: To foster intuition, we first consider the special case $x = r$: the morphism translates:

$$
\begin{array}{lll}
\textcircled{4} \overset{\Phi}{\mapsto} & f_{i+1}\big(\widehat{f_i}(r)\big) \subseteq \widehat{f_i}(r) & \\
& \Leftrightarrow & \text{// } \widehat{f_i}(r) = \mathit{live}_i,\ \mathit{def.\ of}\ \Phi(f_i) \\
& \big(\mathit{live}_i \cup \bigcup_{a \in \mathit{live}_i} G_{i+1}.\mathit{sucs}(a)\big) \subseteq \mathit{live}_i & \\
& \Leftrightarrow & \text{// } (A_1 \cup ... \cup A_n) \subseteq B \Leftrightarrow \forall i : A_i \subseteq B \\
& \forall a \in \mathit{live}_i \colon G_{i+1}.\mathit{sucs}(a) \subseteq \mathit{live}_i &
\end{array}
$$

In order to prove this last property, i.e. $\forall a \in \mathit{live}_i \colon G_{i+1}.\mathit{sucs}(a) \subseteq \mathit{live}_i$, we must consider all nodes $a \in \mathit{live}_i$ and all (sequences of) actions that the Mutator can use to effect the transition $G_i \rightsquigarrow G_{i+1}$. We distinguish the two possibilities for $a \in \mathit{live}_i$:

(1) $a \in G_i.\mathit{freelist}$: Then there are two subcases (which base on the reasonable constraint that nodes in the freelist and newly created nodes do not have "wild" outgoing pointers):

(1a) $a \in G_{i+1}.\mathit{freelist}$, *then* $G_{i+1}.\mathit{sucs}(a) \subseteq G_{i+1}.\mathit{freelist} \subseteq G_i.\mathit{freelist} \subseteq \mathit{live}_i$
(1b) $a \in G_{i+1}.\mathit{active}$ (*caused by addNew*), *then* $G_{i+1}.\mathit{sucs}(a) = \emptyset$; *now* (2) *applies*

(2) $a \in G_i.\mathit{active}$: Then there are three subcases for $b \in G_{i+1}.\mathit{sucs}(a)$:

(2a) $(a \to b) \in G_i.\mathit{arcs} \ \vdash\ b \in G_i.\mathit{active} \subseteq \mathit{live}_i$
(2b) $(a \to b)$ *created by* $\mathit{addArc}(a, b) \ \vdash\ b \in G_i.\mathit{active} \subseteq \mathit{live}_i$
(2c) $(a \to b)$ *created by* $\mathit{addNew}(a) \ \vdash\ b \in G_i.\mathit{freelist} \subseteq \mathit{live}_i$

If we start this line of reasoning not from the roots r but from a superset $x \supseteq r$, then we need to consider supersets $\widehat{l_i} \supseteq \mathit{live}_i$ (where the hat shall indicate that these sets are closed under reachability) and prove $\forall a \in \widehat{l_i} \colon G_{i+1}.\mathit{sucs}(a) \subseteq \widehat{l_i}$. Evidently the reasoning in (1) and (2) applies here as well. But now there is a third case:

(3) $a \in G_i.\mathit{dead}$. In this case there is no operation of the Mutator that could change the successors of a (since all operations require $a \in \mathit{active}$). Hence $G_{i+1}.\mathit{sucs}(a) = G_i.\mathit{sucs}(a)$. Due to the closure property we have $a \in \widehat{l_i} \Rightarrow G_i.\mathit{sucs}(a) \subseteq \widehat{l_i}$. The above equality then entails also $G_{i+1}.\mathit{sucs}(a) \subseteq \widehat{l_i}$.

Φ_3: The morphism Φ translates the axioms ⑦ and ⑧ into

$$s_i \subseteq s_i \cup \bigcup_{a \in s_i} G_i.\mathit{sucs}(a)$$

This is trivially fulfilled such that the constraint on the choice of s_{i+1} is well-defined. *(End of proof)*

When considering the last specification *Micro-Step* in Figure 12 then we have basically shown that any sequence $s_0, s_1, s_2, \ldots$ that fulfills the constraints ⑦ and ⑧ solves our task. But we have not yet given a *constructive* algorithm for building such a sequence. In the next refinement steps Φ_4 and Φ_5 we will proceed further towards such a constructive implementation (actually to a whole collection of implementation variants) by adding more and more constraints to our specification. Each of these refinements constitutes a design decision that narrows down the set of remaining implementations.

4.1 Worksets ("Wavefront")

As a first step towards more constructive descriptions we return to the standard idea of worksets (sometimes referred to as "wavefront"), which has already been illustrated Program 2, and in the examples in Section 2. This refinement is given in Figure 14.

SPEC Workset		
EXTEND *MicroStep*		
$b_0, b_1, b_2, \ldots : A$		*// completely treated ("black")*
$w_0, w_1, w_2, \ldots : A$		*// partially treated ("workset" or "gray")*
$s_i = (b_i \uplus w_i)$		*// partitioning into black and gray*
$\widehat{f_i}(s_i) = b_i \cup \widehat{f_i}(w_i)$	⑫	*// additional constraint*
THM $w_n = \emptyset \Rightarrow \widehat{f_n}(s_n) = b_n$	⑬	*// termination condition*

Fig. 14. The workset approach

The partitioning $s_i = (b_i \uplus w_i)$ arises naturally from the definition of the workset, as in Program 2. But the additional axiom ⑫ is a major constraint! It essentially states that the closure $\widehat{f_i}(s_i)$ of the current approximation s_i shall be primarily dependent on the closure of the workset w_i. This reduces the design space of the remaining implementations considerably – but from a practical viewpoint this is no problem, since we only exclude inefficient solutions. The theorem ⑬ stated in the specification provides a termination condition for the later implementations that is far more efficient than our original termination criterion $f_n(s_n) = s_n$.

An important observation: It is easily seen that the subtle error situation illustrated in Figure 6 in Section 2 violates the axiom ⑫. Therefore any further refinement of the specification *Workset* cannot exhibit this error. In other words: if we derive an implementation by refinement from the specification *Workset* in Figure 14, then we are certain that the bug cannot occur!

A major problem: Unfortunately, just introducing sufficient constraints for excluding error situations is not enough. Consider the situation of Figure 6 in Section 2. We have to ensure that the Mutator cannot perform the two operations $addArc(A, E)$ and $delArc(D, E)$ without somehow keeping the axiom ⑫ intact. This necessitates for the first time that the Mutator cooperates with the Collector, thus introducing constraints for the Mutator. Even though these constraints may be hidden in the component *Store*, they do have an implicit influence on the Mutator's working.

As has already been pointed out in Section 2, there are three principal possibilities to resolve this problem:

- One can stop the Mutator until the Collector has finished (Section 3.1).
- One can put A or E into the workset, when $addArc(A, E)$ is executed.
- One can put E into the workset, when $delArc(D, E)$ is executed.

Each of these solutions keeps the axiom (12) intact, but they have problems. Stopping the Mutator is unacceptable, since this destroys the very idea of having Mutator and Collector work concurrently. In both of the other cases the Mutator adds elements to the workset, while the Collector is taking them out of the workset. Naive implementations of this specification would not guarantee termination.

In the following we will present several refinements for solving this problem. These refinements are the high-level formal counterparts of solutions that can be found in the literature and in realistic production systems for the JVM and .Net.

4.2 Dirty Nodes

One can alleviate the stop times for the Mutator by splitting the workset into two sets, one being the original workset of the Collector, the other assembling the critical nodes from the Mutator. This is shown in Figure 15. The new axiom (14) is similar to (12) using the partitioning $w_i = (g_i \uplus d_i)$.

SPEC Dirtyset		
EXTEND *Workset*		
$g_0,\ g_1,\ g_2,\ \ldots : A$		*// partially treated by Collector ("gray")*
$d_0,\ d_1,\ d_2,\ \ldots : A$		*// introduced by Mutator ("dirty")*
$s_i = (b_i \uplus g_i \uplus d_i)$		*// partitioning into black, gray and dirty*
$\widehat{f}_i(s_i) = b_i \cup \widehat{f}_i(g_i) \cup \widehat{f}_i(d_i)$	(14)	*// closure condition*
THM $g_n = \emptyset \Rightarrow \widehat{f}_n(s_n) = b_n \cup \widehat{f}_n(d_n)$	(15)	*// intermediate termination condition*

Fig. 15. Introducing "dirty" nodes

This specification can be implemented by a Collector that successively treats the gray nodes in g_i until this set becomes empty (which can be guaranteed). But – by contrast to the earlier algorithms – this does not yet mean that all live nodes have been found. As the theorem (15) shows we still have to compute $\widehat{f}_i(d_i)$. But this additional calculation tends to be short in practice, and the Mutator can be stopped during its execution. Consequently, correctness has been retained and termination has been ensured.

The Mutator now adds "critical" nodes to the "dirty" set d_i. In order to keep the set d_i as small as possible one does not add all potentially critical nodes to it: as follows from axiom (14), black or gray nodes need not be put into d_i. And since d_i is a set, nodes need not be put into it repeatedly. Actually, when the Mutator executes $addArc(a, b)$ with $a \notin s_i$ ("a is still before the wavefront"), then axiom (14) would allows us the choice of putting a into d_i or not (similarly for b. Commonly, a is simply added to d_i.

4.3 Implementing the Step $s_i \mapsto s_{i+1}$

So far all our specifications only impose the constraint (8) (see *MicroStep* in Figure 12) on their implementations, that is:

$$s_i < s_{i+1} \leq f_i(s_i) \quad \vee \quad s_i = f_i(s_i)$$

The actual computation of the step $s_i \mapsto s_{i+1}$ has to be implemented by some function *step*. For this function we can have different degrees of granularity:

– In a *coarse-grained* implementation we pick some node x from the gray workset and add all its non-black successors to the workset. Then we color x black.

This variant is simpler to implement and verify, but it entails a long atomic operation. The corresponding write barrier slows down the standard working of the Mutators.
– In a *fine-grained* implementation we treat the individual pointer fields within the current (gray) node x one-by-one. In our abstract setting this means that we work with the individual arcs.
This makes the write barrier shorter and thus increases concurrency, but the implementation and its correctness proof become more intricate.

On our abstract level we treat this design choice by way of two different refinements. This is depicted in Figure 16 (where the shorthand notation $\ldots$ USING x WITH $p(x)$ entails that the property only has to hold when such an x exists).

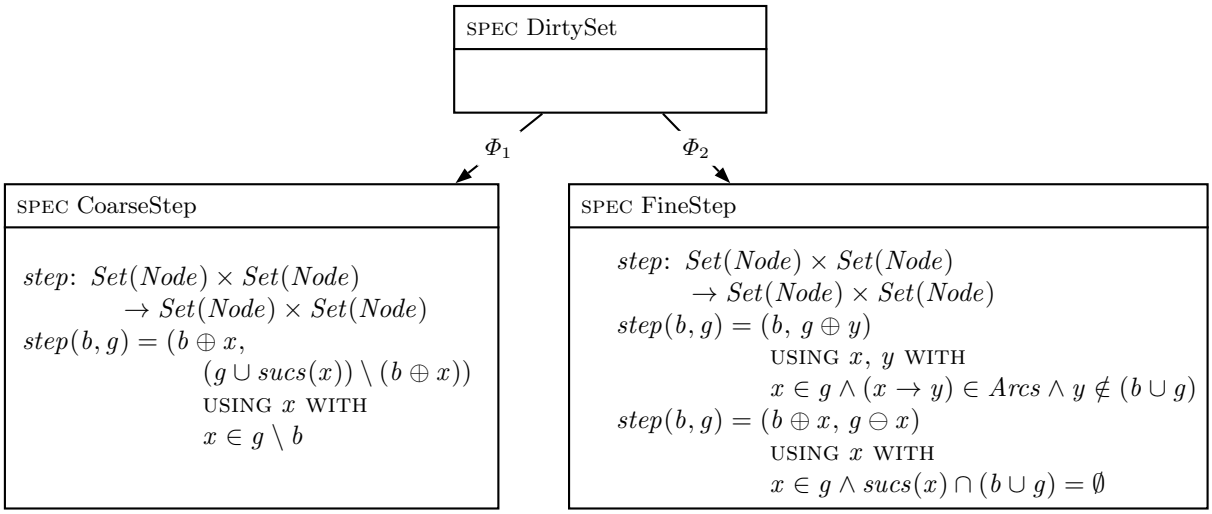


Fig. 16. Step functions of different granularities

A note of caution. If we apply the morphism Φ introduced at the beginning of Section 4 directly, the strict inclusion $s_i < s_{i+1}$ of axiom ⑧ would not be provable. Therefore we must interpret

$$(b, g) < (b', g') \quad \overset{\Phi}{\mapsto} \quad b \subset b' \vee (b = b' \wedge g \subset g').$$

But there are still further implementation decisions to be made. Both *CoarseStep* and *FineStep* specify (at least partly) how the *step* operation deals with the selected gray node. But this still leaves one important design decision open: *How are the gray nodes selected?* In the literature we find several approaches to this task:

1. *Iterated scanning.* One may proceed as in the original paper by Dijkstra et al. [3] and repeatedly scan the heap, while applying *step* to all gray nodes that are encountered. This has the advantage of not needing any additional space, but it may lead to many scans over the whole heap, in the worst case $\mathcal{O}(N^2)$ times, and is not considered practical.
2. Alternatively one performs the classical recursive graph traversal, which may equivalently be realized by an iteration with a workset managed as a stack. This allows all the well-known variations, ranging from a stack for depth-first traversal to a queue for breadth-first traversal. In any case the time cost is in the order $\mathcal{O}(|live|)$, since only the live nodes need to be scanned. However, there also is a worst-case need for $\mathcal{O}(|live|)$ space – and space is a scarce resource in the context of garbage collection.
3. One may compromise between the two extremes and approximate the workset by a data structure of bounded size (called a *cache* in [4, 5]). When this cache overflows one has to sacrifice further scan rounds.

4. When there are multiple mutators, for efficiency it is necessary to have local worksets working concurrently.

These design choices are illustrated in Figure 17, but we refrain from coding all the technical details.

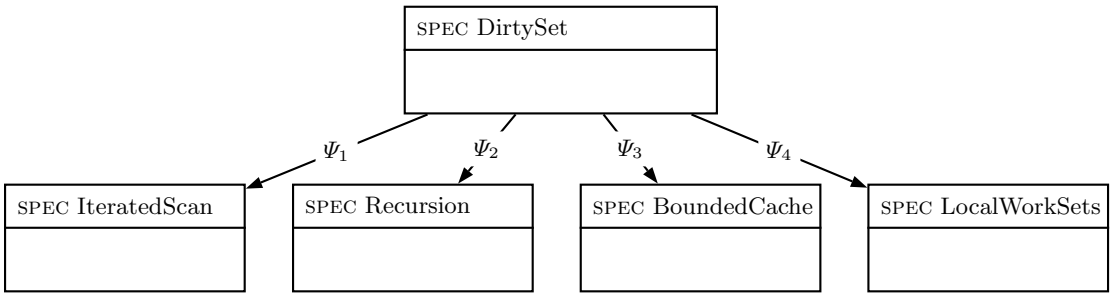


Fig. 17. Design choices for finding the gray nodes

It should be emphasized that the refinements $\Psi_1, \Psi_2, \Psi_3, \Psi_4$ of Figure 17 are independent of the refinements Φ_1, Φ_2 of Figure 16. This means that we can combine them in any way we like. The combination of some Φ_i with some Ψ_j is formally achieved by a pushout construction as already mentioned in Section 1. In a system like Specware [7] such pushouts are performed automatically.

Further refinements to handle generational garbage collectors, dirty cards, dirty pages and related techniques for scanning the dirty nodes are sketched in [13].

5 Conclusion

It is well known that realistic garbage collectors exhibit a huge amount of technical details that are ultimately responsible for the size and complexity of the verification efforts. The pertinent issues cover a wide range of questions such as:

- What are the exact read and write barriers?
- How do we treat the references in the global variables, the stacks and the registers?
- Where do we put the marker bits (in mark-and-sweep collectors) or the forward pointers (in copying collectors)?

Due to space limitations we have omitted discussion of these and other topics, which may however be found in the extended version of this paper [13], particularly the issue of computing the dynamically changing set of roots.

We have shown how the main design concepts in contemporary concurrent collectors can be derived from a common formal specification. The algorithmic basis of the concurrent collectors required the development of some novel generalizations of classical fixpoint iteration theory. We hope to find a wide variety of applications for the generalized theory, as there has been for the classical theory. This is of interest since the reuse of abstract design knowledge across application domains is a key factor in the economics of formal derivation technology. Alternative refinements from the basic algorithm lead to a family tree of concurrent collectors, with shared ancestors corresponding to shared design knowledge. While our presentation style has been pedagogical, the next step is to develop the derivation tree in a formal derivation system, such as Specware.

Acknowledgment. We are grateful to Erez Petrank and Chris Hawblitzel, with whom one of us (pp) enjoyed intensive discussions at Microsoft Research. Their profound knowledge on the challenges of practical real-world garbage collectors motivated us to push our original high-level and abstract treatment further towards concrete and detailed technical aspects – although we realize that we may still be on a very abstract level in the eyes of true practitioners. We would also like to thank Bernd Finkbeiner and the MPC reviewers for their helpful comments.

References

1. Hezi Azatchi, Yossi Levanoni, Harez Paz, and Erez Petrank. An on-the-fly mark and sweep garabage collector based on sliding views. In *OOPSLA'03, Anaheim CA*, 2003.
2. Jiazhen Cai and Robert Paige. Program Derivation by Fixed Point Computation. *Science of Computer Programming*, 11(3):197–261, April 1989.
3. Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Comm. ACM*, 21(11):965–975, November 1978.
4. Damien Dolingez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL'94, Portland Oregon*, ACM SIGPLAN Notices, pages 70–83. ACM Press, January 1994.
5. Damien Dolingez and Xavier Leroy. A concurrent generational garbage collector for a mulit-threaded implementation of ml. In *POPL'93, New York, NY*, ACM SIGPLAN Notices, pages 113–123. ACM Press, 1993.
6. Chris Hawblitzel and Erez Petrank. Automated verification of practical garbage collectors. In *POPL'09, Savannah, Georgia*, pages 113–123, October 2009.
7. Kestrel Institute, 3260 Hillview Ave., Palo Alto, CA 94304 USA. *Specware System and documentation*, 2003. http://www.specware.org/.
8. Stephen Kleene. *Introduction to Metamathematics*. American Mathematical Society Press, 1956.
9. John MacCarthy. Recursive functions of symbolic expressions and their computation by machine. *Comm. ACM*, 3(4):184–195, 1960.
10. Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. In *PLDI'07, San Diego*, 2007.
11. Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages*, 4(3):402–454, July 1982.
12. Dusko Pavlovic, Peter Pepper, and Doug Smith. Colimits for concurrent collectors. In N. Dershovitz, editor, *Verification: Theory and practice, essays dedicated to Zohar Manna*, volume 2772 of *Lecture Notes in Computer Science*, pages 568–597. Springer Verlag, 2003.
13. Dusko Pavlovic, Peter Pepper, and Doug Smith. Formal derivation of concurrent garbage collectors. Technical Report TR-2010-1, Kestrel Institute, February 2010. ftp://ftp.kestrel.edu/pub/papers/smith/PPS-2010.pdf.
14. Peter Pepper and Petra Hofstedt. *Funktionale Programmierung*. Springer Verlag, 2006.
15. David M. Russinoff. A mechanically verified incremental garbage collectors. *Formal Aspects of Computing*, 6:359–390, 1994.
16. Douglas R. Smith. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering*, 16(9):1024–1043, 1990.
17. Douglas R. Smith. Toward a classification approach to design. In *Proceedings of Algebraic Methodology and Software Technology (AMAST)*, volume LNCS 1101, pages 62–84. Springer-Verlag, 1996.
18. Douglas R. Smith. Designware: Software development by refinement. In M. Hoffman, D. Pavlovic, and P. Rosolini, editors, *Proceedings of the Eighth International Conference on Category Theory and Computer Science*, pages 355–370, 1999.
19. Douglas R. Smith, Eduardo A. Parra, and Stephen J. Westfold. Synthesis of planning and scheduling software. In A. Tate, editor, *Advanced Planning Technology*, pages 226–234. AAAI Press, Menlo Park, 1996.
20. G. L. Steele. Multiprocessing compactifying garbage collection. *Comm. ACM*, 18(9):495–508, Sep. 1975.
21. Martin T. Vechev, Eran Yahav, and David F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. In *PLDI 06, Ottawa, Canada*. ACM Press, 2006.
22. T. Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, March 1990.