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 of the trusted computing base of a large portion of the world's mission-critical software infrastructure, such unreliability is unacceptable [31]. Therefore it is a worthwhile if not mandatory endeavor to provide means by which the quality of such software can be improved – 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. [20, 25, 9]). But also fully automated verifiers exhibit problems. As can be seen e.g. in [13] even the verification of a simplified collector necessitates such a large amount of complex properties that the specification may easily become faulty itself.

These considerations show a first mandatory prerequisite for the development of correct software of realistic size and complexity: not only the software but also its correctness proof need to be *modularized*. However, such a modularization is not enough. Even when it has been successfully verified that all requested properties are fulfilled by the software, it remains open, whether these properties taken together do indeed specify the intended behavior. This is an external judgment that lies outside of any verification system. Evidently, such judgments are easier and more trustworthy, when the properties are few, simple and easy to grasp.

Finally there is a third aspect which needs to be addressed by a development methodology. Garbage collectors – like most software products – come in a plethora of possible variations, each addressing specific quality or efficiency goals. When each of these variations is verified separately, a tremendous duplication of work is generated. On the other hand it is extremely difficult to analyze 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 here a development method, which addresses the aforementioned issues and which has already been successfully applied to complex problems, for example real-world-size planning

and scheduling tasks [5, 27]. The method bases on the concept of *specification refinement*. Two major aspects of this concept are illustrated in Figures 1 and 2.

1.1 Sequential vs. Concurrent Garbage Collection

The very first garbage collectors, which essentially go back to McCarthy's original design [19], 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. [8] and Steele [28] (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 [9, 10], 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* [31]: 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.

1.2 Abstract and Concrete Problems

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. Ideally the abstract problem/solution pairs can even be found in a library like the one of the Specware system [16].

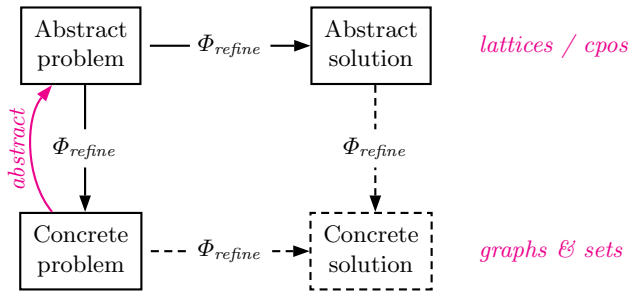


Fig. 1. Abstract and concrete problems and their solutions

For example, in the subsequent sections of this paper 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.

Technically all our problem and solution descriptions are algebraic and coalgebraic specifications (as will be defined more precisely later), 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. (This concept has nowadays been popularized as "automatic code generation from models".) 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 1 also illustrates another aspect of our methodological way of proceeding. When we are confronted with a concrete problem, we try to extract from it a more abstract problem that represents the core of the given task. Even though this looks like additional effort at first sight, it is usually a worthwhile endeavor. First of all, we obtain the desired modularization of the derivation and verification. Secondly, the concentration on the kernel of the problem usually simplifies the finding of the (abstract) solutions. And last but not least we can often come up with variations on the theme that would have been buried under the bulk of details otherwise. As is pointed out in Figure 1 the introduction of the details of the concrete problem can be done almost automatically and thus does not really cause additional work.

This principle of working with an abstract view of the concrete problem can also be found in other approaches, for example in [20, 13]. But there the principle is more implicitly used (in statements such as *Correctness means that each of these procedures faithfully represent the abstract state* [13]), whereas we make the abstraction/concretization into an explicitly available development tool, based on a rigorous notion of morphisms.

1.3 Development by Refinement

Figure 2 illustrates the second essential aspect of our method. We do not work with a single problem/solution pair and their concrete instances. Rather we construct a whole "family tree" (which actually may be a 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.

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 accordingly 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 [26, 16, 22]

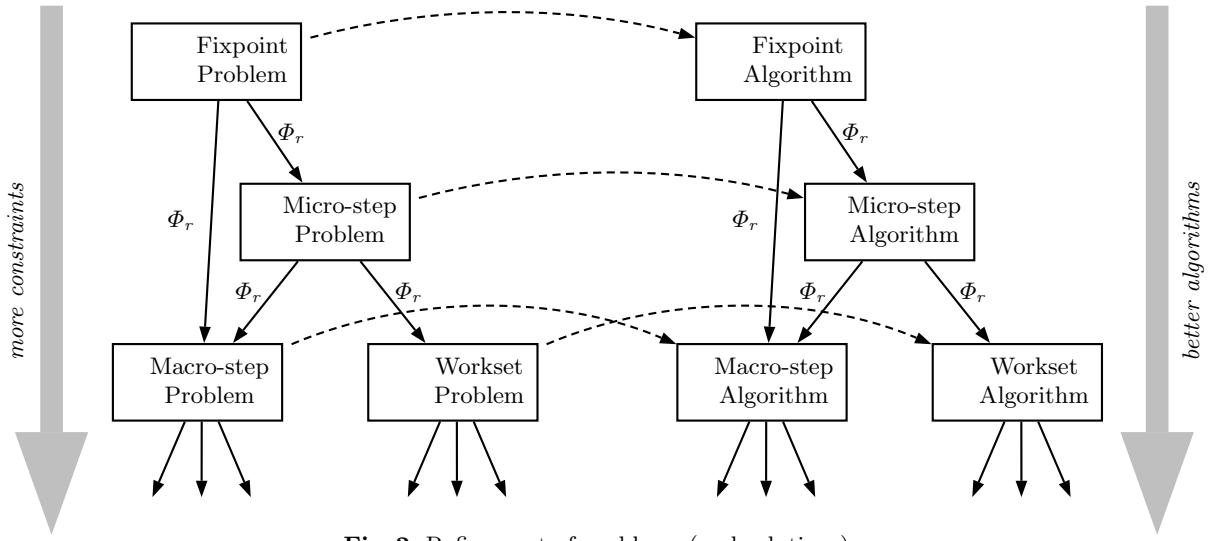


Fig. 2. Refinement of problems (and solutions)

More details about the method sketched above can be found in [26]. The remainder of this paper will make things more precise by presenting concrete examples.

In an earlier paper [22] we have presented one exemplary development of a garbage collector from an initial non-executable specification to an executable implementation. But – as was critically noted in [31] – we did "not explore an algorithm space". Such an exploration is the main purpose of the present paper. This is a similar goal as that of Vechev et al. [31]: 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 [31] is the possibility to combine various "design dimensions" in a very flexible way. By contrast to their approach we study the family tree of specifications and implementations that can be systematically derived using formal refinements. (The interchangeability of some of these refinements actually makes the family tree into a family dag.) So our focus is on the *method* of refinement and its potential tool support and not on garbage collection as such. Moreover, whereas both our earlier paper [22] and the work of Vechev et al. [31] mostly concentrate on one phase of garbage collection – namely the marking phase – the present paper addresses the whole task of garbage collection. In addition, we do not only consider mark-and-sweep collectors but also copying collectors. Last but not least, we base the whole treatment of garbage collection on very fundamental mathematical principles, namely lattices and fixed points.

1.4 Data Reification

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 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 widely automatic in systems like Specware [16], including 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.

1.5 Summary of Results

We present a methodology that allows us to derive a wide variety of garbage collection algorithms in a systematic way. This approach not only modularizes the resulting programs but also the derivation process itself such that the verification is split into small and easy-to-comprehend pieces, allowing considerable reuse of proofs. In more detail, we present the following results:

- We start by presenting a "dynamic" generalization of the well-known fixed-point results of lattice theory.
- This basis is presented as a general specification that covers a whole range of implementations. We call this the "micro-step" approach.
- These fixpoint-based specifications can be refined further to more and more detailed designs, which correspond to the major algorithms found in today's literature.

Even though we cannot present all the algorithms in full detail, we can at least show "in principle", how a whole variety of important and practical algorithms come out from our refinement process. These include above all the (DLG) algorithm of Doligez, Leroy and Gonthier [9] – which sometimes is considered the culmination of concurrent collector development [1] – and its descendants.

2 Notes on Garbage Collection

Even though our approach starts from very abstract and high-level mathematical concepts – viz. lattices and fixed points (Section 3) – and takes several refinement steps (Section 4) before it ends with some special aspects of garbage collection (Section 5), it is helpful to motivate our main design decisions by having the concrete application of garbage collection in mind. Actually, we perceive three major stages of refinements:

1. We start from a "purely mathematical problem", namely lattices (actually cpos) and fixed points. On this level we derive the core properties that mark off the solution space.
2. Then we proceed to "abstract garbage collection"; that is, we model the problem by graphs and sets. This intermediate stage can on the one hand be easily shown to be an instance of the lattice-based abstraction; but on the other hand it already refers to important aspects of the concrete garbage collection problem. Hence, all algorithmically relevant aspects can be dealt with on this level.
3. The final step introduces the various specialized data structures, write barriers and the like that go to make a realistic garbage collector. (This final step will only be roughly sketched in this paper.)

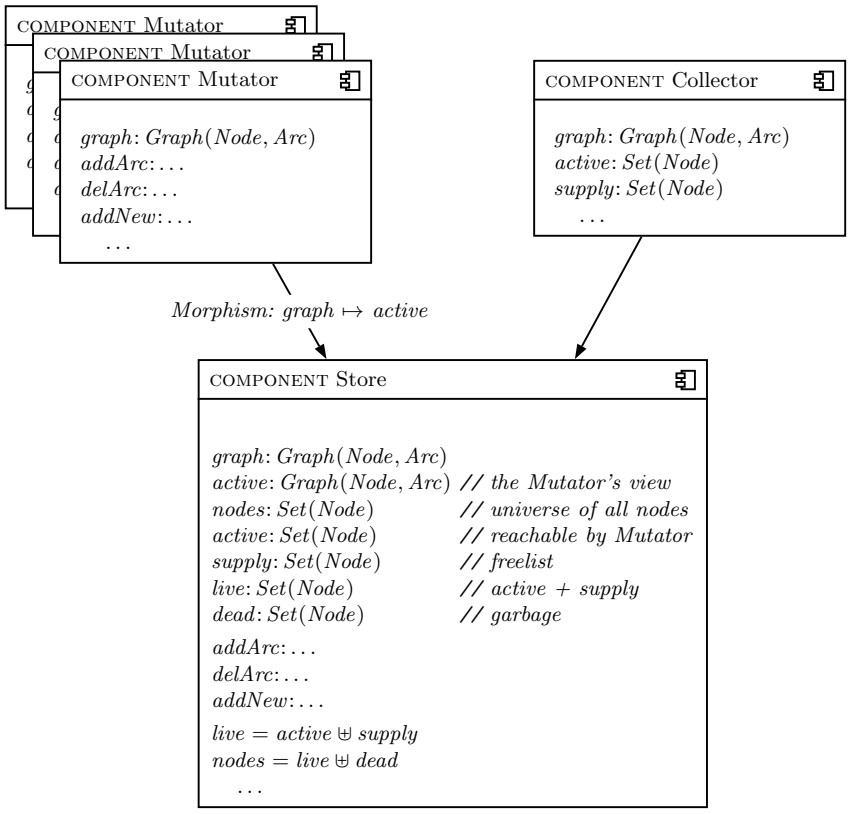


Fig. 3. The system architecture

2.1 Architecture and Basic Terminology

Before we delve into the formal derivation we want to clarify the basic setting and the terminology that we use here. This is best done at the intermediate level of abstraction, where the garbage collection problem is formulated in terms of graphs and sets.

We modularize the problem by way of three kinds of components (using a UML-inspired representation; see Figure 3). 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*. It can essentially perform three primitive operations:[4]

- $addArc(a, b)$: add a new arc between two nodes a and 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").

[4] This considerably simplifies the memory model used in the famous DLG algorithm [9, 10], where the Mutator has eight operations. However, the essence of these operations is captured by our three operations above. (We will come back to this issue in Section 5.)

– *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.[5] But on this abstract level its specification also provides the basic terminology that is needed for talking about garbage collection. In particular we use 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.
– *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 3 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.)

2.2 Fundamental Properties of the Mutator

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 never "escape" the realm given by its graph and the freelist; that is, it can never reclaim dead garbage nodes. In other words, the realm of live nodes (graph + freelist) monotonically* decreases.

2.3 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 3. The Mutator continuously performs its basic operations *addArc*, *delArc* and *addNew*, which – from the Mutators viewpoint – are all considered

[5] This reflects the situation of many modern systems, ranging from functional languages like ML or Haskell to object-oriented languages like C# or Java. In languages like C or C^{++} the situation is more intricate.

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 are existing 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:

- $|\, active| = MemorySize$. That is, the Mutator has used all available memory in its graph. Then nothing can be done!
- $|\, active| < MemorySize$. When $supply = \emptyset$, this means that $dead \neq \emptyset$. This is the situation in which we want to recycle dead 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 4.

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

Fig. 4. 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) $\Box \Diamond \big(supply \longrightarrow supply \uplus recycle(G)\big)$

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.)

2.4 How to Find Dead Nodes

Unfortunately, the specification of *recycle* in Figure 4 is not easily implementable since the dead nodes are not directly recognizable. Since the *dead* nodes are computed by taking the complement of the *live* nodes (i.e. $live = \complement dead = nodesbackslashlive$), the idea comes to mind to work with the complement of *recycle*. This leads to the simple calculation

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

where we introduce a new function $trace(G)$ such that $recycle(G) = \complement\, trace(G)$.

This leads to the refined version of the Collector's specification in Figure 5.

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

Fig. 5. The Collector's task (first refinement)

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!

2.5 An Intuitive Example and a Subtle Bug

We demonstrate the working of a typical garbage collection algorithm by a simple example. Figure 6 illustrates the situation at the beginning of the Collector 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".

Figure 7 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 "hot zone" – the so-called "workset" –, which means that they are marked/copied, but not all outgoing arcs have been handled yet.

Figure 8 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.

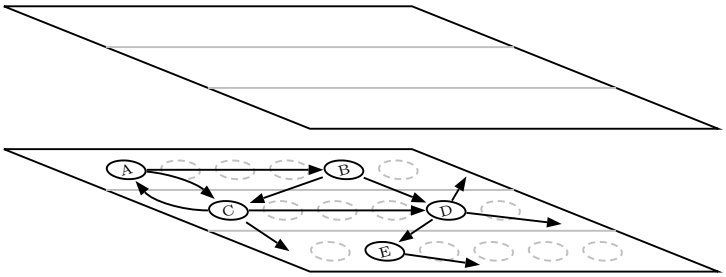


Fig. 6. At the start of the Collector

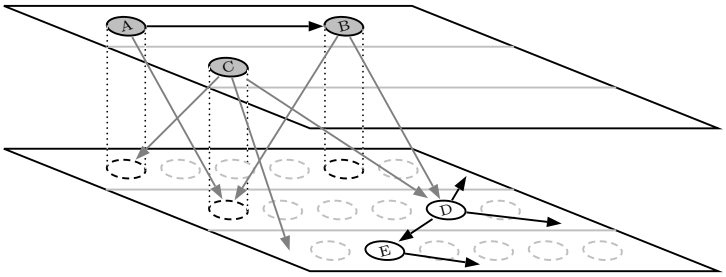


Fig. 7. A snapshot

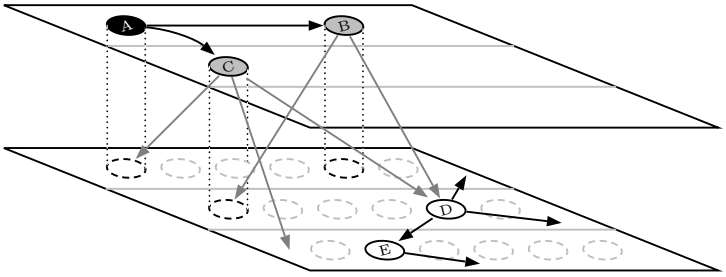


Fig. 8. The next snapshot

Note that we have the invariant property (which will play a major role in the sequel) 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. [8].

A subtle problem: 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 depicted in Figure 9. Since A is no longer in the workset, its connection to E will not be detected. Hence, E is *hidden from the Collector* [31] and therefore will be treated as a dead garbage node – which is a severe bug!

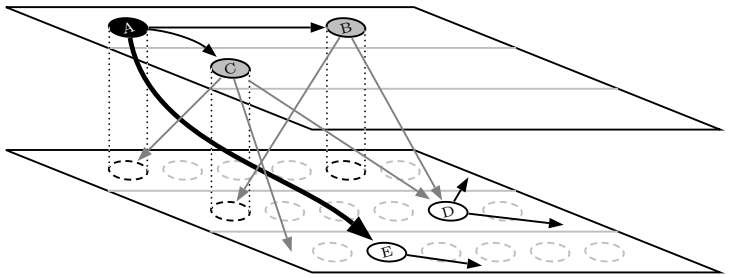


Fig. 9. A subtle error

Any formal method for deriving garbage collection algorithms must ensure that this bug cannot happen! Note that this situation violates the invariant about the downward arrows. And our formal treatment will show that keeping this invariant intact is actually the clue to the derivation of correct garbage collectors.

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. [8].)
- When performing $addArc(A, E)$, record A. (This is the approach taken by Steele [28].)
- When performing $delArc(D, E)$, record E. (This is the approach taken by Yuasa [32].)

Vechev et al. [31] speak of "installation-protected" in the first two cases and of "deletion-protected" in the last case.

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 8 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. – The same bug again!

At this point we will leave the concrete considerations about garbage collectors and pass on to the more abstract viewpoint of fixed points and lattices (or cpos). In the terminology of Figure 1 we follow the upward arrow, that is, we generalize a concrete problem into a more abstract one. Once this is done, we can derive a whole variety of solutions in a strictly top-down fashion.

3 Mathematical Foundation: Fixed Points

In garbage collection one can roughly distinguish two classes of collectors (see Section 1.1):

- *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).

As we will demonstrate in a moment, the traditional stop-the-world collectors correspond on the abstract level to classical fixed-point theory. For the concurrent collectors we need to generalize this classical fixed-point theory to a variant that we baptized "dynamic fixed points".

We briefly review the classical theory before we present our generalization.

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

The best known treatments of the classical fixpoint problem in complete lattices are those of Tarski [29] and Kleene [17]. Before we quote these we present some relevant terminology (assuming that the reader is already familiar with the very basic notions of partial order, join, meet etc.)

- For a set $s = \{ x_0, x_1, x_2, \ldots\}$ of type $Set(A)$ and a function $f\colon A \to A$ we use the overloaded function $f\colon Set(A) \to Set(A)$ by writing $f(s)$ as a shorthand for $\{f(x_0), f(x_1), f(x_2), \ldots\}$. (In functional-programing notation this would be written with the apply-to-all operator as $f * s$.)
- 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. This could be shortly written as $f \circ \sqcup = \sqcup \circ f$ (by using the overloaded versions of the symbol f).
- The function f is *inflationary* in x, if $x \leq f(x)$ holds. Then x is called a *post-fixed point* of f. (Analogously for *pre-fixed points*.)
- 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 $\widehat{f}(x) = \textsc{least}\ u.\ u = f(u) \wedge x \leq u$ 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* $x \leq f(x)$

Proof. The first three properties follow directly from the definition of $\widehat{f}$. The last one can be seen as follows: Denote $u = \widehat{f}(x)$ and $v = \widehat{f}(f(x))$. Then we have by monotonicity

$$x \leq f(x) \;\vdash\; \widehat{f}(x) \leq \widehat{f}(f(x)) \;\vdash\; u \leq v.$$

On the other hand we have

$$x \leq u \;\vdash\; f(x) \leq f(u) = u.$$

Since v is the least value with $v = f(v) \wedge f(x) \leq v$, we have $v \leq u$. *(End of proof)*

Theorem 1 (Tarski). *Let L be a complete lattice and $f : L \to L$ a monotone function on L. Then f has a complete lattice of fixed points. In particular the least fixed point is the meet of all its pre-fixed points and the greatest fixed point is the join of all its post-fixed points.*

Theorem 2 (Kleene). *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.

In the meanwhile it has been shown that the essence of these theorems also holds in the simpler structure of *complete partial orders (cpos)*[6].

More recently Cai and Paige [6] published a number of generalizations that are streamlined towards practical algorithmic implementations of fixpoint computations. We paraphrase their main result here, since we are going to utilize it as a "blueprint" for our subsequent development.

Theorem 3 (Cai-Paige). *Let A be a cpo and $f\colon A \to A$ be a monotone function that is inflationary in r. Let moreover $\{s_0, s_1, s_2, \ldots, s_n\}$ be an arbitrary sequence obeying the conditions*

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

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 3 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 [23]. All practical collectors use a workset that records nodes that await marking.

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

To illustrate these basic results, we derive the overall structure of 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:

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

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

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

This allows us to produce a correct, but naive iterative algorithm to compute dead nodes which is based on Theorem 2.

Program 1 Raw Fixpoint Iteration Program

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

Following Cai and Paige [6], we can construct an 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

$X \setminus g(X)$

$=$ { Definition }

$X \setminus (nodes \setminus (roots \cup \{b \mid b \in sucs(a) \ \ \& \ \ a \in nodes \setminus X\}))$

$=$ $\{$ Using the law $A \setminus (B \cup C) \;=\; (A \setminus B) \setminus C$ $\}$

$X \setminus ((nodes \setminus roots) \setminus \{b \mid b \in sucs(a) \;\&\; a \in nodes \setminus X\})$

$=$ $\{$ Using the law $A \setminus (B \setminus C) \;=\; (A \setminus B) \bigcup (A \cap C)$ $\}$

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

$=$ $\{$ Using the law $\{x|P(x)\} \cap Q \;=\; \{x|P(x) \wedge x \in Q\}$ $\}$

$(X \setminus (nodes \setminus roots)) \bigcup \{b \mid b \in sucs(a) \;\&\; b \in X \;\&\; a \in nodes \setminus X\}$

$=$ $\{$ Again using the law $A \setminus (B \setminus C) \;=\; (A \setminus B) \bigcup (A \cap C)$ (on first term) $\}$

$(X \setminus nodes) \cup (X \cap roots) \bigcup \{b \mid b \in sucs(a) \;\&\; b \in X \;\&\; a \in nodes \setminus X\}$

$=$ $\{$ Simplifying $\}$

$\{\} \cup (X \cap roots) \bigcup \{b \mid b \in sucs(a) \;\&\; b \in X \;\&\; a \in nodes \setminus X\}$

$=$ $\{$ Simplifying$\}$

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

The greatest fixpoint expression can be computed by the workset-based Program 2, which is based on Theorem 3.

Program 2 Workset-based Fixpoint Iteration Program

```
1  W ← nodes;
2  while ∃z ∈ ((W ∩ roots) ⋃ {b | b ∈ sucs(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 [21], according to which we incrementally maintain the invariant

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

There are two places in the code that might disrupt the invariant, in lines 1 and 3. We maintain the invariant in line 1 with respect the initialization $W = nodes$ as follows:

Assume: $W = nodes$
Simplify: $WS = (W \cap roots) \bigcup \{b \mid b \in sucs(a) \;\&\; b \in W \;\&\; a \in nodes \setminus W\}$

$(WS = (roots \cap nodes) \bigcup \{b \mid b \in sucs(a) \;\&\; b \in nodes \;\&\; a \in nodes \setminus nodes\}$

$$
\begin{array}{ll}
= & roots \bigcup \{b \mid b \in sucs(a) \ \& \ b \in nodes \ \& \ a \in \{\}\} \\
= & roots \bigcup \{\} \\
= & roots.
\end{array}
$$

and incremental code (wrt the change $W' = W - z$):

Assume: $WS = (W \cap roots) \bigcup \{b \mid b \in sucs(a) \ \& \ b \in W \ \& \ a \in nodes \setminus W\}$
$\quad\quad \& \ W' = W - z$
Simplify: $WS' = (W' \cap roots) \bigcup \{b \mid b \in sucs(a) \ \& \ b \in W' \ \& \ a \in nodes \setminus W'\}$

$W' \cap roots) \bigcup \{b \mid b \in sucs(a) \ \& \ b \in W' \ \& \ a \in nodes \setminus W'\}$

$=$ $\quad$ { Using assumption $W' = W - z$ }

$((W - z) \cap roots) \bigcup \{b \mid b \in sucs(a) \ \& \ b \in (W - z) \ \& \ a \in nodes \setminus (W - z)\}$

$=$ $\quad$ { Simplifying }

$((W \cap roots) - z) \bigcup (\{b \mid b \in sucs(a) \ \& \ b \in W \ \& \ a \in nodes \setminus (W - z)\} - z)$

$=$ $\quad$ { Pulling out common subtraction of z }

$(W \cap roots) \bigcup \{b \mid b \in sucs(a) \ \& \ b \in W \ \& \ a \in nodes \setminus (W - z)\} - z$

$=$ $\quad$ { distribute element membership }

$(W \cap roots) \bigcup \{b \mid b \in sucs(a) \ \& \ b \in W \ \& \ (a \in (nodes \setminus W) \ \vee \ a = z)\} - z$

$=$ $\quad$ { distribute set-former over disjunction }

$$
\begin{array}{l}
(W \cap roots) \\
\quad \bigcup (\{b \mid b \in sucs(a) \ \& \ b \in W \ \& \ a \in (nodes \setminus W)\} \\
\quad \bigcup \{b \mid b \in sucs(a) \ \& \ b \in W \ \& \ a = z)\}) - z
\end{array}
$$

$=$ $\quad$ { fold definition of WS, and simplify}

$WS \bigcup \{b \mid b \in sucs(z) \ \& \ b \in W\}) - z.$

The resulting code is shown in Program 3.

Programs 2 and 3 represent 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, such as finite differencing, simplification, and datatype refinements. For example, the finite set

Program 3 Optimized Fixpoint Iteration Program

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

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.

As our final preparatory step within the realm of the classical fixed-point concepts we mention a central property that is the core of the correctness proof for implementations. If we compute the sequence s_0, s_1, s_2, ..., by a loop, then a Hoare-style verification would need the following invariant.

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

$$\widehat{f}(s_i) = \widehat{f}(r)$$

Proof. This invariance follows directly from monotonicity and the properties of $\widehat{f}$ stated in Lemma 4: $\widehat{f}(s_i) \leq \widehat{f}(s_{i+1}) \leq \widehat{f}(f(s_i)) = \widehat{f}(s_i)$. *(End of proof)*

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 keeps 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, ... 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* $\rightarrow$ *Set*(*Node*) $\rightarrow$ *Set*(*Node*) and

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

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, ...

The foundation 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.

The initial problem formulation Based on this foundation we can now formulate our goal. Recall the specification of the garbage collection task given by *Collector* in Figure 5:

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

$live \subseteq trace(G) \subset nodes$. 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 in Section 2.2: 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.[7]

[7] It may be interesting to compare property ⑤ to the classical non-concurrent situation expressed in Lemma 1. Instantiated to the final value $s = s_n$ Lemma 1 states

$s = \widehat{f}(r)$

.

This equality can be formally rewritten into the two inequalities

$\widehat{f}(r) \leq s \leq \widehat{f}(r)$

.

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. 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:*

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

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

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

We show by induction that $\forall i\colon\ g^i(x) \le \widehat{h}(x)$. Initially we have $g^0(x) = x \le \widehat{h}(x)$ due to the general reflexivity property ③ of the closure. The induction step uses the induction hypothesis and then the premise: $g^{i+1}(x) = g(g^i(x)) \le g(\widehat{h}(x)) \le \widehat{h}(x)$.

By instantiating f_{i+1} for g and f_i for h we immediately obtain $\widehat{f}_{i+1}(x) \le \widehat{f}_i(x)$ by using the axiom ④, when $r \le x$. *(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 (see Theorem 3) 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"*
$s_i < s_{i+1} \le f_i(s_i) \vee s_i = f_i(s_i)$	⑧	*// computation step*
THM $\exists\, n\colon\ \widehat{f}_n(r) \le s_n \le \widehat{f}_0(r)$	⑨	*// to be shown below*
THM $\widehat{f}_0(s_0) \ge \widehat{f}_1(s_1) \ge \ldots \ge \widehat{f}_n(s_n)$	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:

This is weakened in property ⑤ by setting on the left $\widehat{f} \mathrel{\widehat{=}} \widehat{f}_n$ and on the right $\widehat{f} \mathrel{\widehat{=}} \widehat{f}_0$. (Similar kinds of weakenings are also found in Hoare- or Dijkstra-style program developments, when deriving invariants from given pre- or post-conditions.)

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

The right half $s_n \le \widehat{f}_0(r)$ 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*:*

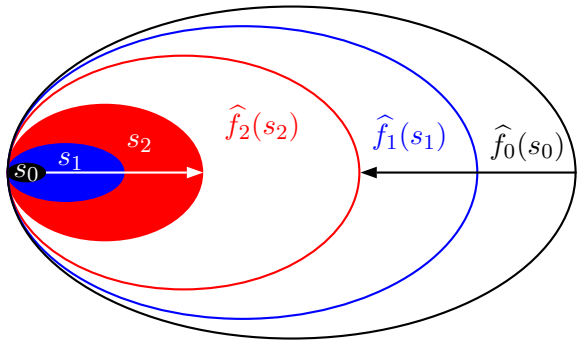
$$\forall i\colon\ \widehat{f}_{i+1}(s_{i+1}) \le \widehat{f}_i(s_i)$$

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

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

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

Lemma 6 may be depicted as follows:



As can be seen here, the approximations s_0, s_1, s_2, ... keep growing, while at the same time their closures $\widehat{f}_0(s_0)$, $\widehat{f}_1(s_1)$, $\widehat{f}_2(s_2)$, ... keep shrinking.

Remark: This situation can also be rephrased as follows: We have a function $F(f_i, s_i)$ that is applied to the elements of two sequences. This function is antitone in the first argument and monotone in the second argument; we have to show that – under the constraints given in our specification – the function still is monotonically increasing.

This essentially concludes the derivation that can reasonably be done on this highly abstract mathematical level of fixed points and lattices. However, in the literature one can find a variant of collectors, the development of which is best prepared on this level of abstraction as well.

3.4 A Side Track: *Snapshot Algorithms*

Consider again the specification *Microstep* in Figure 12, where the goal is described in axiom ⑨ as $\exists\, n\colon\ \widehat{f}_n(r) \le s_n \le \widehat{f}_0(r)$. Evidently computing the value $s_n = \widehat{f}_0(r)$ is an admissible solution.

[8] This approach, which has been used by Yuasa [32] and was refined later by Azatchi et al. [1], is also referred to as *snapshot-at-the-beginning*.

In order to follow this development path we refine the specification *Microstep* in Figure 12 to the specification *Snapshot* in Figure 13 by requiring the additional constraint (11) that needs to be respected by later implementations. Note that the new axiom (11) is the classical invariant that is also used in non-concurrent garbage collectors.

SPEC Snapshot		
EXTEND *MicroStep*		
$\forall\, i\colon\ \widehat{f_0}(s_i) = \widehat{f_0}(r)$	(11)	// *classical invariant* $live_i = live_0$

Fig. 13. The "snapshot approach"

How can this kind of computation be achieved in practice? This is demonstrated by Azatchi et al. [1] based on a technique that has been developed by some of the authors for a concurrent reference-counting collector [18]. Starting from the fictitious idea of making a virtual snapshot by cloning the complete original heap, it is then shown that this copying can be done lazily such that only those nodes are actually copied that are critical.

We only sketch this idea here abstractly in our framework: We introduce a structure $clone_i$ and an operator ∇ such that $(G_i \nabla clone_i) \simeq G_0$; i.e. $x \notin clone_i \Rightarrow G_i.sucs(x) = G_0.sucs(x)$ and $x \in clone_i \Rightarrow clone_i.sucs(x) = G_0.sucs(x)$. The computation of the sequence s_0, s_1, s_2, ... is then done based on $(G_i \nabla clone_i)$ such that we effectively always apply f_0.

It remains to determine the structure $clone_i$. There are solutions of varying granularity, but the most reasonable choice appears to be the following: Whenever the Mutator executes one of the operations $addArc(a, b)$, $delArc(a, b)$ or $addNew(a)$, it puts the old a including all its outgoing arcs into *clone*. (This amounts to making a copy of the heap cell.) In practice, the efficiency of this approach is considerably improved by observing that the cloning need only be done during a very short phase of the collection cycle. The most complex aspect of this approach is the computation of the pointers into the heap that come from the local fields (stack, registers) of the Mutator; it has to be ensured that during this phase no simultaneously changed pointers get lost. In [1, 18] this is performed by a technique called "snooping": essentially all pointers that are changed during this phase are treated as roots. (We will come back to this in Section 5.)

For the technical details of this approach we refer to [1]. Suffice it to say that almost all of our subsequent refinements – which we start from the specification *MicroStep* – could also descend from *Snapshot*. Technically, we could combine *Snapshot* with our various refinements of *MicroStep* by forming a pushout.

Remark: By showing that such an effectively working implementation exists, we have implicitly shown that the specification *Snapshot* is consistent with the specification *MicroStep*; that is, the refinement is admissible.

[8] Remember that the closure computes the live nodes; axiom (9) therefore means $live_n \subseteq s_n \subseteq live_0$, which can be solved by $s_n = live_0$. In other words, we compute the nodes that were live, when the Collector started.

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 correspondences:

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

- 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.)

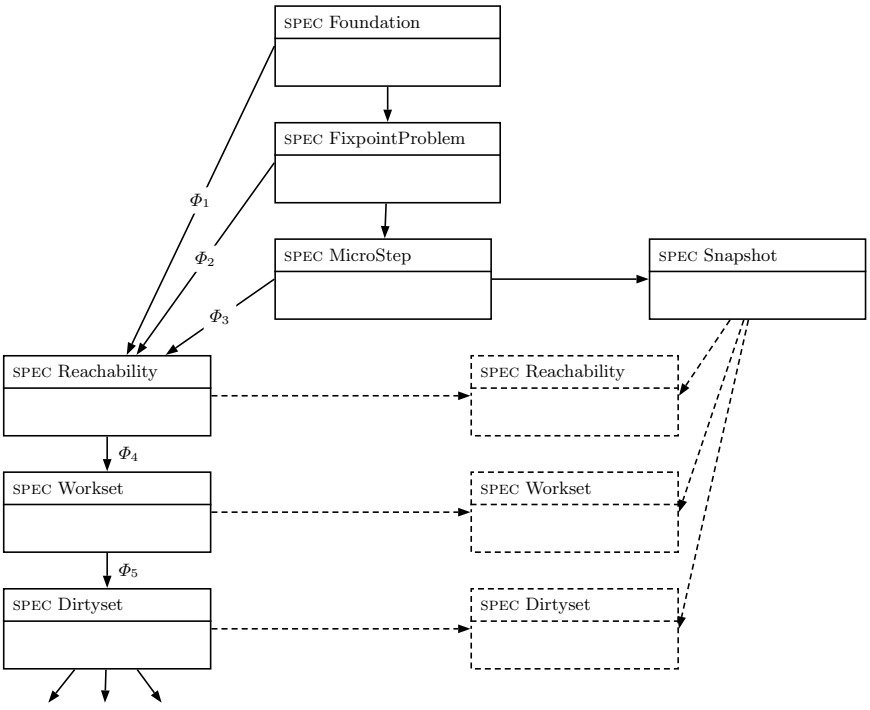


Fig. 14. Roadmap of refinements

Figure 14 illustrates the road map through our essential refinements. The upper half shows the refinements that have been performed in the previous Section 3 on the abstract mathematical

level of lattices and fixed points. The lower half shows the refinements on the semi-abstract level of graphs and sets that will be presented in this section. Finally, the right side of the diagram points out that all further developments could also be combined (by way of pushouts) with the sidetrack of the snapshot approach of Section 3.4.

Lemma 7 (Morphism abstract → concrete). *Under the above morphism Φ all axioms of the abstract specifications Foundation, FixpointProblem and MicroStep hold for the more concrete specifications of graphs and sets (see Figure 14).*

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.

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

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

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

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

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, ... 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 also referred to as "wavefront"), which has already been illustrated Program 2, and in the examples in Section 2.5. This refinement is given in Figure 15.

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. 15. 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 9 in Section 2.5 violates the axiom ⑫. Therefore any further refinement of the specification *Workset* cannot exhibit this error. In other words: If we derive all our implementations as offsprings of the specification *Workset* in Figure 15, 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 9 in Section 2.5. 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 2.5, 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 a number of 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 16. 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. 16. 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 17 (where the shorthand notation $\ldots$ USING x WITH $p(x)$ entails that the property only has to hold when such an x exists).

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. [8] and repeatedly scan the heap, while applying *step* to all gray nodes that are encountered. This has

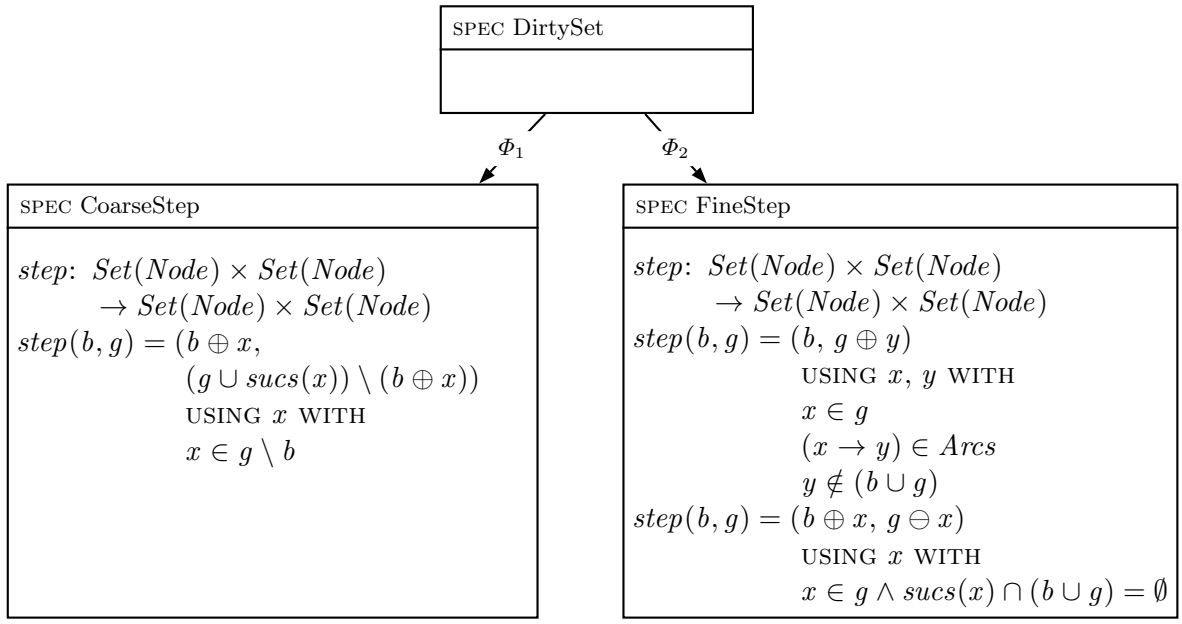


Fig. 17. Step functions of different granularities

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 [9, 10]). 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 18. (But we refrain from coding all the technical details.)

It should be emphasized that the refinements Ψ_1, Ψ_2, Ψ_3, Ψ_4 of Figure 18 are independent of the refinements Φ_1, Φ_2 of Figure 17. 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 1.2. In a system like Specware [16] such pushouts are performed automatically.

It should be noted that axiom ⑮ in Figure 16 requires at least one scan in order to perform the cleanup $\widehat{f}_n(d_n)$ of the dirty nodes after the main marking phase is completed.

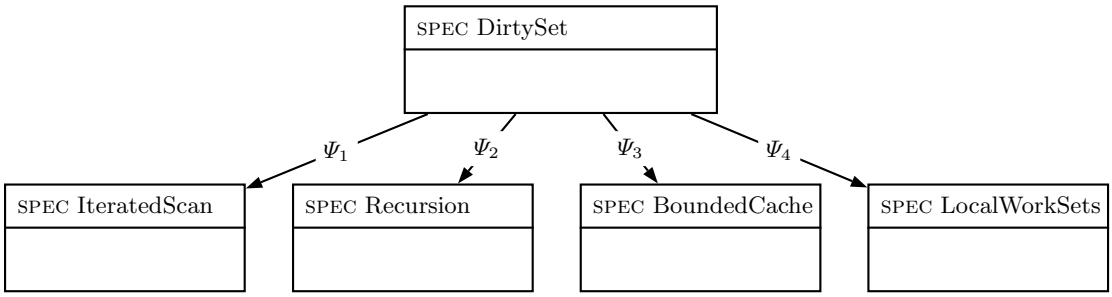


Fig. 18. Design choices for finding the gray nodes

The Collector may also compute (parts of) $\widehat{f_i}(d_i)$ at any given point in time concurrently with the Mutators. This may make the finally remaining dirty set d_n smaller and thus speed up the cleanup operation, which shortens the necessary pauses for the Mutators. These interim computations are harmless as long as the termination of the Collector's main marking phase does not depend on the set d_i becoming empty.

4.4 Heap Partitioning (Generations, Cards, Pages etc.)

The necessary scanning of the "dirty nodes" described in Section 4.3 above motivates a refinement that actually has a whole variety of different applications. In other words, the following abstract refinement is the parent of a number of further refinements that aim at solving different kinds of problems.

In the following we briefly sketch, how such alternative refinements fit into our abstract framework. In Sections 6.3 and 6.4 we will discuss some concrete applications of this paradigm, in particular

- *generational* garbage collectors;
- *dirty cards*;
- *dirty pages*.

We integrate these techniques into our approach by means of *superimposing* a further structuring on the graph. This has already been hinted at in Figures 6–9 in Section 2.5, where the planes are further partitioned into areas.

Definition 1 (Graph partitioning). *A* partitioning *of the graph is given by splitting the set of nodes into disjoint subsets:*

$$Nodes = N_1 \uplus \ldots \uplus N_k$$

The subsets N_i are called cards *(following the terminology of e.g. [24]).*

These cards can be used to optimize the computation of the dirty nodes without compromising the correctness of the algorithm. To this end we need to introduce the constraint shown in the specification *DirtyCards* in Figure 19.

SPEC DirtyCards		
EXTEND *Dirtyset*		
TYPE $Card = Set(Node)$		
$N_1, \ldots, N_k$: $Card$		// *cards*
$Nodes = N_1 \uplus \ldots \uplus N_k$		// *cards partition the node set*
$dirty$: $Card \to Bool$		// *"dirty" property of cards*
$d_i \subseteq \bigcup \{ N_j : \ dirty(N_j) \ \}$	(16)	// *constraining dirty nodes*

Fig. 19. Introducing "dirty" cards

The axiom (16) establishes the constraint that the dirty nodes can only lie on dirty cards. (This constraint has to be obeyed by the Mutator.)

The axiom (15) in Figure 16 requires the cleanup computation of $\widehat{f}_n(d_n)$ after the main marking phase has been completed. For this cleanup the new axiom (16) entails a considerable speedup, since only a subset of the cards N_j need to be scanned.

5 Dynamic Root Sets

In the previous sections we have performed the transition from classical fixed points in static graphs (relating to stop-the-world collectors) to dynamic fixed points in changing graphs (relating to concurrent collectors). However, we still utilize the inherent assumption that the Collector starts from a *fixed root set* r. Alas, this assumption – which is also contained in the original papers by Dijkstra, Steele and others [8, 28] – can not be maintained in practice due to the following reasons (see e.g. [9, 10, 1]):

1. The local data of the Mutator (registers and stack) are indeed *local*; that is, the Collector has *no access* to them!
2. The synchronization between the Collector and the Mutators requires *write barriers*. The corresponding overhead may be tolerable for heap accesses, but it is certainly out of the question for local stack or register operations.

As a consequence of the first observation the Mutators have to participate actively in the garbage collection process, at least during the start phase of each collection cycle. And the second observation rules out certain solutions due to their unacceptable overhead. So the challenge is to maximize concurrency in the presence of these constraints.

In the following we will show how these issues fit into our overall framework.

5.1 Modeling Local Data

In accordance with our earlier levels of abstraction we devise the following modeling for the Mutator-local data: all local data of a Mutator (registers and stack) are considered as one large

node, which we call the pre-root ρ_m. The potentially quite large set $sucs(\rho_m)$ then represents all pointers out of the local stack and registers into the heap.[9]

From now on let us assume that there are q Mutators $M_1, \ldots, M_q$ with pre-roots $\rho_1, \ldots, \rho_q$. The global variables are represented by the pre-root ρ_0; they are accessible by the Collector.

Definition 2 (Pre-roots; local roots). *Each Mutator possesses a* pre-root ρ_m*. The successors of this pre-root are referred to as the* Mutator's local roots*:* $r_m = sucs(\rho_m)$*.*

Moreover, the global variables *(which are accessible to the Collector) are represented by the pre-root* ρ_0 *and the corresponding roots by* $r_0 = sucs(\rho_0)$*.*

The set of all pre-roots is denoted as $\rho = \{\rho_0, \rho_1, \ldots, \rho_q\}$*. The set of all roots is defined as* $r = sucs(\rho) = r_0 \cup \bigcup_{m \in Mut} r_m$*.*

The set r introduced in Definition 2 above essentially plays the role of the start value s_0 in the specification *MicroStep* of Figure 12. Hence, we might rephrase the central property ⑨ of this specification (after applying the morphism Φ from Section 4):

$$\exists n\colon \widehat{f}_n(r) \subseteq s_n \subseteq \widehat{f}_0(r) \quad \text{WHERE } r = sucs(\rho) = sucs(\rho_0) \cup \bigcup_{m \in Mut} sucs(\rho_m).$$

However, due to the subtle difficulties that are caused by the concurrent activities of Collector and Mutators we should retreat to a more fundamental rephrasing of our overall task.

5.2 Computation of the Roots

The abstract modeling introduced in Definition 2 at the beginning of this section allows us to retain all the other modeling aspects of the preceding sections. In particular the concept of varying graphs $G_0, G_1, G_2, \ldots$ and the thus induced functions $f_0, f_1, f_2, \ldots$ can be applied to the computation of the root set r without changes.

However, for reasons that will become clear in a moment, we need to make one change. Since the local registers and stacks of the Mutators are not part of the heap, they must not undergo the mark and sweep or the copying process. In our mathematical modeling we therefore no longer consider the reflexive-transitive closure $\widehat{f}$ but only the non-reflexive transitive closure, which we denote as $\widetilde{f}$. As a consequence, the function f should not be inflationary either. Hence, the morphism Φ at the beginning of Section 4 now sets $f_i(s) = \bigcup_{a \in s} G_i.sucs(a)$. Consequently, the axiom ⑧ in the specification *MicroStep* has to be changed to

$$s_i < s_{i+1} \leq s_i \sqcup f_i(s_i) \vee f_i(s_i) \leq s_i \qquad ⑧$$

With these changes (for which all previous proofs work unchanged except for minor notational adaptations) we can now reformulate the garbage collection task slightly differently (see Figure 20, where the numbers are the same as in the original specifications).

A major difference between this specification *MicroStep′* and the original specification *MicroStep* is the omission of the axiom ⑦, which determines the start value s_0. This start value is no

[9] Here it pays that we model pointers more abstractly as a (multi)set of arcs. This is much more concise and elegant than speaking about "objects" and their "slots" for pointers, as it is usually done in the literature.

SPEC Micro-Step′		
$\widetilde{f}(x) = \text{LEAST } s\text{: } f(x) \leq s \wedge s = f(s)$	③	// *transitive closure*
$s_0, s_1, s_2, \ldots : A$		// *sequence of approximations*
$s_i < s_{i+1} \leq s_i \sqcup f_i(s_i) \vee f_i(s_i) \leq s_i$	⑧	// *computation step*
$\text{THM } \exists\, n\text{: } \widetilde{f}_n(\rho) \leq s_n \leq \widetilde{f}_0(\rho)$	⑨	// $live_n \subseteq s_n \subseteq live_0$

Fig. 20. The "micro-step approach"

longer a constant, but now has to be computed from the pre-roots. This computation is slightly intricate, since the graph is undergoing continuous changes. To be more precise, we have the following scenario:

- There are q mutators $M_1, \ldots, M_q$.
- When Mutator M_i computes its local roots r_i from its pre-root ρ_i, then the graph is in some stage G_j.
- During the root computation the mutator stops its other activities. And the other mutators cannot access the mutator's local data. Hence the outgoing arcs from the local data into the heap remain unchanged throughout the local root computation. Hence we obtain $r_i = f_i(\rho_i) = G_j\,.\,succs(\rho_i)$.[10]

Based on these observations, the set $s_0 = r$ that is computed by the mutators essentially is

$$r = f_0(\rho_0) \cup f_1(\rho_1) \ldots \cup f_q(\rho_q) \qquad \text{// too naive}$$

However, this naive approach doesn't work all the time as we will show in the following.

A problem. Looking at our central correctness property ⑨ we would wish that the equality $\widetilde{f}(\rho) = \widehat{f}(r)$ holds. Alas, this is not necessarily the case. To see this, consider the example of Figure 21 (adapted from [9]).

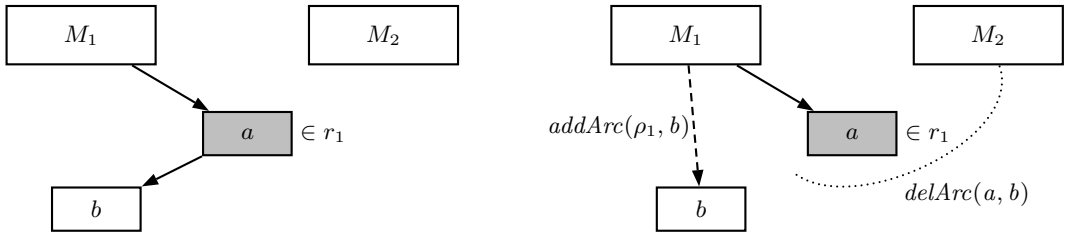


Fig. 21. A potential error

[10] Note that this does *not* mean that the graph remains invariant throughout the computation of the local root set r_i. On the contrary, the other mutators will usually change the graph continuously. But these changes do not affect the specific set $G_j\,.\,succs(\rho_i)$.

Suppose M_1 has computed its (only) local root a, i.e. $r_1 = \{a\}$ (left side of Figure 21). Then it resumes its normal activities, which happen to load a pointer to b into a local variable or register; this is modeled as $addArc(\rho_1, b)$ on the right side of Figure 21. When the Collector now starts its recycling activities, we have $b \notin r_1$, even though $b \in sucs(\rho_1)$. So the Collector starts from a wrong root set.

It is easily seen that this can indeed be disastrous: suppose that before the start of the Collector some Mutator M_2 (or M_1 itself) deletes the pointer from a to b (right side of Figure 21). Then b will indeed be considered garbage, even though it is still reachable from M_1.

This is the same problem as the one that we had already encountered in Figure 9 of Section 2.5. However, there is a difference: the problematic pointers now are not caused by heap operations but by operations on the local variables and registers. For reasons of efficiency we do not want to slow down these local operations by wrapping them into read or write barriers.[11] This would be particularly unpleasant, since the protection is only needed during an extremely short phase (namely the root marking), while the overhead would be hindering permanently.

In [9] further problems are illustrated that could occur, when the Mutator is e.g. interrupted between the $addArc(\rho_1, b)$ and the $delArc(a, b)$ operation for such a long time that the Collector performs a whole collection cycle.

Towards a solution. We enforce the invariant

$$\widetilde{f}_i(\rho) \subseteq \widehat{f}_i(s_i).$$

Then when the computation terminates with the fixpoint $s_n = f_n(s_n)$, the fixpoint properties immediately entail

$$\widetilde{f}_n(\rho) \subseteq s_n.$$

This can be rephrased as

$$live_n \subseteq s_n.$$

which is the main correctness criterion for the collector (as was stated in property ⑨ of Figure 12).

When looking at the critical situation of Figure 21, we can see that there are three perceivable solutions for establishing this requirement:

1. We can stop all other activities of the mutators during the local-root computation.
   This is very safe, but it clearly has the disadvantage of causing long pauses. In [9] it is shown how this solution can be achieved using three handshakes that ensure that the Collector and all Mutators are in sync.
2. We could request that the operation $addArc(\rho_1, b)$ puts b into the set s_i.
   This is a correct solution, but it has the disadvantage that we need a *read barrier* for loading heap pointers into local variables or registers. Even though this read barrier will be a single if-test during most of the time, it does add overhead.

[11] In some systems there is not even enough information to distinguish pointers from other values such that an abundance of operations would have to be engulfed into this protective overhead of barriers.

3. We could request that the operation $delArc(a, b)$ puts b into the set s_i.
   This is a correct solution, which needs a *write barrier* on heap operations. This is not so bad, since this write barrier already exists for handling the other problems encountered in the previous sections. Still, there are disadvantages: marking nodes at the very moment of their deletion will create floating garbage in the majority of cases. Moreover, the operation $delArc(a, b)$ has to touch the node a, since it changes one of its slots; but the marking additionally has to touch another node, namely b. This adds to the overhead.

We should mention that a fourth kind of solution is possible in the "snapshot-at-the-beginning" approaches (see Section 3.4). This is demonstrated in the so-called "sliding-views" approach of [1].

4. The operation $delArc(a, b)$ produces a clone of the node a.
   The advantage here is that less floating garbage is generated and that only the object a needs to be touched by the write barrier. But one has the overhead of the storing and managing the snapshot.

6 Real-world Considerations

It is well known that realistic garbage collectors – in particular concurrent or parallel ones – 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)?

Evidently we do not have the space here to address these questions in detail. But we should at least indicate the path towards their solution within our method. Therefore we list here some of the technical features contained in realistic product-level collectors and show how they fit into our framework.

6.1 The Mutators' Capabilities

In Section 2.1 we have limited the Mutators' behavior to the three operations *addArc*, *delArc* and *addNew*. By contrast, Doligez et al. [9, 10] use eight operations, some of which are modified in other approaches [11]. We may group their operations as follows:

- *move, load*: local data transfers (stack, registers) and read access to heap cells;
- *reserve, create*: obtain space from freelist; create cell in that space;
- *fill, update*: write into a new/existing cell;

– *cooperate, mark*: synchronize with Collector; mark local roots.

This design breaks the usual *allocate* operation into the three separate operations *reserve, create, fill*. Moreover it treats *fill* differently from *update*, since a new object is guaranteed to be still unknown to other Mutators and therefore can be handled with less synchronization overhead. (But in the approach of [11] this distinction is abolished.) This diversity and granularity is important for concrete discussions about issues such as Mutator-local mini heaps and other implementation details. But for our modeling approach and its refinement and correctness considerations our three basic operations cover the essential aspects. Technically speaking, the eight operations of Doligez et al. can be obtained by refining our low-level models even further.

Moreover, Doligez et al. [9, 10] and many of their successor papers use a notation like `heap[x,i]` to refer to the i-th slot in the object x in the heap. Our model achieves the same effect with a more mathematical attitude by considering individual arcs $(a \to b)$. In other words, $(a \to b_1)$, $\ldots$, $(a \to b_k)$ model the slots of the object a. Our notation allows a more flexible treatment of the question of whether objects are uniform or can have varying numbers of successor slots.

6.2 Availability of a Runtime System

Section 2.1 introduces an architecture with a component *Store* that represents the memory management used in modern runtime systems such as .Net or JVM, based on old ideas from the realm of functional languages such as ML or Haskell. Such a component provides only indirect access to the heap such that it is relatively easy to integrate read or write barriers, handshakes and other organizational means. The vast majority of the newer papers therefore target these kinds of architectures.

In "uncooperative languages" such as C or C^{++} things are more intricate and it is not surprising that only a few papers address their demands, e.g. [3]. The difficulties caused by such uncooperative languages are overcome by using the capabilities of the underlying operating system such as the page table to introduce concepts like *dirty pages*. Moreover, one has to deal with lots of floating garbage, since – due to the lack of typing information – many non-pointer values have to be treated conservatively as if they were pointers. Last but not least there are also more intricate synchronization issues between the Collector and the Mutators. Benchmarks indicate that the overhead in such uncooperative languages is considerably higher than that in cooperative languages.

6.3 Generational Collectors

In Section 4.4 we have shown that we can superimpose the graph with an additional structuring such that the set of nodes is partitioned into subsets: $\mathit{Nodes} = N_1 \uplus \ldots \uplus N_k$. Such a partitioning is fully compatible with our correctness considerations and the pertinent refinements:

Generational garbage collectors partition the nodes according to their "age", where the age usually reflects the number of collection cycles that the node has survived.

Whereas in traditional garbage collectors the nodes are physically moved to another area, this is no longer possible in concurrent collectors due to the large overhead that the pointer tracking

and synchronization would require. Therefore one usually only defines the generations logically. A non-moving solution for concurrent collectors is presented by Domani et al. [12] basing on earlier work of Demers et al. [7].

Printezis and Detlefs [24] describe a concurrent generational collector that has been implemented as part of the SUN Research JVM. The different generations often use different collectors. For the young generation, where nodes tend to die fast, copying collectors are the technique of choice, whereas in older generations a mark-and-sweep approach works better [24].

The most critical issue in generational collectors is the treatment of old-to-young references, since the whole point of generations is NOT to touch the elder generations in most collection cycles. To cope with these references (that are caused by the Mutators' activities) one usually employs the dirty-cards or dirty-pages techniques that we discuss in the section.

6.4 "Dirty Areas" (Cards, Pages, ...)

As has been pointed out in Section 4.4 the disturbances caused by the Mutators can be encapsulated into a concept of "dirty areas", which allows a compromise between accuracy and efficiency.

- *Cards* are often used to speed up the scanning by constraining it to memory areas that actually may need scanning. This is done by using a *dirty bit* for each card, on which a mutator has performed some update: when the dirty bit is set, the card needs to be scanned. Cards and dirty bits are used (possibly under a different name) in connection with generational garbage collectors [24] but also to cope with problems caused by multiple mutators. They can be based on very efficient write barriers. For example, the SUN ResearchVM [24] uses the two-instruction write barrier proposed in [30].
- *Pages* taken from the virtual-page mechanism of the underlying OS together with *dirty bits* can be used for uncooperative languages like C and C^{++}, where no runtime system exists that nicely separates the application programs from the memory management [3]. However, as is reported in [14], the use of cards is more efficient than the use of page-protection-based barriers.

6.5 Privacy of Local Data; Local Heaps

One of the primary goals of many concurrent-collector designs is to keep the stop times of the mutators to a minimum (in practice within an order of magnitude of 2ms; see e.g. [1]). Here the greatest barrier is the global synchronization, when the mutators derive the local roots from their pre-roots (local registers and stack). *The longest time that a thread waits for garbage collection is the time for it to mark the objects directly reachable from its stack* [11].

Based on the observation that the major percentage of heap dynamics comes from short-lived and small objects, the global synchronization can be made less frequent by providing every mutator with its own local "mini heap" [9, 13]. Then the mutator only needs to interact with the global heap in order to acquire a new mini heap, when the old one is full, or in order to store large objects. Otherwise it uses a very simple allocation scheme in its local heap, e.g. using a

so-called "bumper-pointer" technique [13]. This design, which is used e.g. in the IBM JVM, also has the advantage of being cache-friendly [2, 4, 15].

6.6 Detailed Memory Management

There is a plethora of little details to be considered in real-world garbage collectors, of which the following list gives but a small selection.

- *Object size.* The original garbage collectors by McCarthy [19] or Dijkstra [8] are based on the assumption of fixed-size heap cells. But in reality these cells come in all sizes and internal layouts. There all kinds of solutions for this problems such as using freelists of different sizes, splitting large objects into smaller pieces, using Mutator-local heaps for small objects and so forth. The size information is either stored with the object or inferred from the object's class. And so forth.
- Coalescing. The sweeping phase should try to combine consecutive free junks into one large free junk in order to alleviate the fragmentation problem. This is easy in non-concurrent collectors, but is more complex in concurrent collectors, since now the Collector and the Mutators compete for the same resource, namely for the removal of cells from the *freelist* [24].
- *Marker and pointer management.* All the algorithms use various kinds of markers – for example the colors black, gray, white or the dirty bits – and various pointers – for example the forward pointers in copying collectors or the clone pointers in the "sliding-views" approaches. A typical technique for handling such problems is e.g. described in [13], where the vtable pointer, which is the first word of every object, is overwritten for the forward pointer needed in the copying collector. These pointers can be distinguished, since they point into disjoint (and known) storage areas.
- *Sets, markers and bitmaps.* Many markers actually represent the membership of the node in a certain set. This can either be implemented by a marker bit in each object or by a global set representation using a bitmap.
- *Coping with "no-information".* Most garbage collection approaches nowadays address the JVM or DotNet, where all memory accesses of application programs are indirect, since they are handled by a memory manager. Therefore all the garbage collection activity can be bundled in the runtime system. Old systems based on C or C^{++} do not exhibit this luxury. There one may at best use work-arounds such as employing the virtual-paging mechanism of the underlying OS by making pages "dirty", whenever they are written to [3]. Another problem here is that many integer values have to treated as if they were pointers, this way producing a lot of floating garbage.

7 Conclusion

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.

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. Katherine Barabash, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, Yoav Ossia, Avi Owshanko, and Erez Petrank. A parallel, incremental mostly concurrent garbage collector for servers. *ACM Transactions on Programming Languages*, 27(6):1097–1146, Nov 2005.
3. Hans Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *SIGPLAN Notices*, 26(6):157–164, 1991.
4. Sam Borman. Sensible sanitation – understanding the ibm java garbage collector, part i: Object allocation. In *IBM developer works*, August 2002.
5. Mark H. Burstein and Douglas R. Smith. ITAS: A portable interactive transportation scheduling tool using a search engine generated from formal specifications. In *Proceedings of AIPS-96*, Edinburgh, UK, May 1996.
6. Jiazhen Cai and Robert Paige. Program Derivation by Fixed Point Computation. *Science of Computer Programming*, 11(3):197–261, April 1989.
7. Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel G. Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. POPL'90. *ACM SIGPLAN Notices*, pages 261–269, January 1990.
8. 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.
9. 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.
10. 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.
11. Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Eliot E. Salant, Katherine Barabash, Itai Lahan, Yossi Levanoni, Erez Petrank, and Igor Yanorer. Implementing an on-the-fly garbage collector for java. In *ISMM '00: Proceedings of the 2nd international symposium on Memory management*, pages 155–166, New York, NY, USA, 2000. ACM.
12. Tamar Domani, Elliot K. Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for java. In *PLDI'00*, 2000.
13. Chris Hawblitzel and Erez Petrank. Automated verification of practical garbage collectors. In *POPL'09, Savannah, Georgia*, pages 113–123, October 2009.
14. A. L. Hosking and J. Eliot B. Moss. Protection traps and alternatives for memory management of an object-oriented language. In Barbara Liskov, editor, *Proc. 14th Symposium on Operatig System Principles, New York*, pages 106–119. ACM Press, Decemeber 1993.
15. Haim Kermany and Erez Petrank. The compressor: Concurrent, incremental and parallel compaction. In *PLDI'06, Ottawa*, pages 354–363. ACM Press, 2006.
16. Kestrel Institute, 3260 Hillview Ave., Palo Alto, CA 94304 USA. *Specware System and documentation*, 2003. http://www.specware.org/.
17. Stephen Kleene. *Introduction to Metamathematics*. American Mathematical Society Press, 1956.

18. Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for java. oopsla 2001. *ACM SIGPLAN Notices*, 36(10):367–380, 2001.
19. John MacCarthy. Recursive functions of symbolic expressions and their computation by machine. *Comm. ACM*, 3(4):184–195, 1960.
20. 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.
21. Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages*, 4(3):402–454, July 1982.
22. 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.
23. Peter Pepper and Petra Hofstedt. *Funktionale Programmierung*. Springer Verlag, 2006.
24. Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. Technical Report SMLI TR-2000-88, SUN Microsystems, June 2000.
25. David M. Russinoff. A mechanically verified incremental garbage collectors. *Formal Aspects of Computing*, 6:359–390, 1994.
26. 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.
27. 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.
28. G. L. Steele. Multiprocessing compactifying garbage collection. *Comm. ACM*, 18(9):495–508, Sep. 1975.
29. Alfred Tarski. A lattice-theoretical fixpoint theorem and its applicatons. *Pacific J. Math.*, 5(2):285–309, 1955.
30. Hölzle U. A fast write barrier for generationl garbage collectors. In *OOPSLA'93 Workshop on Memory Management and Garbage Collection, Washington DC*, 1993.
31. 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.
32. T. Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, March 1990.