Colimits for Concurrent Collectors

Dusko Pavlovic[1], Peter Pepper[2], and Doug Smith[1]

[1] Kestrel Institute ({dusko,smith}@kestrel.edu)
[2] Technische Universität Berlin (pepper@cs.tu-berlin.de)

Abstract. This case study applies techniques of formal program development by specification refinement and composition to the problem of concurrent garbage collection. The specification formalism is mainly based on declarative programming paradigms, the imperative aspect is dealt with by using monads. We also sketch the use of temporal logic in connection with monadic specifications.

1 Introduction

The study of algebraic specification techniques has led to deep insights into this specification paradigm and to very elaborate methods and systems such as the SPECWARE environment at Kestrel Institute [16, 15]. In this approach one can derive complex algorithms by elaborate combinations of specifications. The underlying principles are taken from category theory, in particular morphisms, pushouts, and colimits.

However, these concepts were mostly targeted to purely functional computations. The inclusion of imperative programming started only recently. The even more challenging task of addressing parallel problems in this setting has – to our knowledge – not been seriously undertaken so far.

Today the study of the integration of algebraic/functional and imperative features concentrates mainly on two approaches: *abstract state machines* and *monads*. The specification language SPECWARE contains the former concept in the form of *evolving specifications* [12], whereas functional programming languages such as OPAL or HASKELL usually prefer monads. We choose here the monad-style formalism, since we want to assess its usefulness in the context of specifications.

It is generally agreed that the task of parallel garbage collection provides a sufficiently challenging problem for the evaluation of programming concepts. Therefore we choose this task for our case study. Since there is ample literature on the issue of garbage collection, we refer only to the book [3] and the article [18] for background information.

In the greater part of the paper we work out the example of *concurrent garbage collection* (Sections 2-5). The assessment of the *conceptual principles* that are applied during the development is sketched in Section 6. We base our programs on the concept of *monads*.

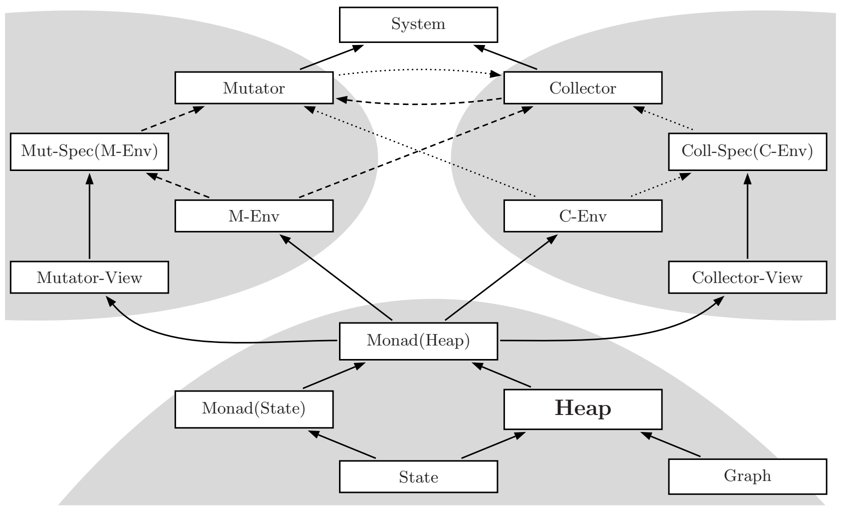


Fig. 1. Sketch of the specification structure

Program 1.1 Outline of the system

```
SPEC System =
  IMPORT Mutator, Collector
  FUN run : M[Void] = ( mutate || collect )

SPEC Mutator = Mutator-Spec(Collector)
SPEC Collector = Collector-Spec(Mutator)

SPEC Mutator-Spec(Mutator-Environment) = ...
SPEC Collector-Spec(Collector-Environment) = ...
```

The overall structure of our derivation is sketched in Figure 1 and its textual representation in Program 1.1. It consists of three major parts: the base specifications, the mutator, and the collector. The basis of the whole program is the specification `Heap`, an extension of `Graph`, which contains all relevant aspects and operations. Together with the parameterized specification `Monad` this yields the central structure `Monad(Heap)` of monadic heaps. The upper half of the diagram shows that the overall `System` is composed of a `Mutator` and a `Collector`. The `Mutator` is an instance of the parameterized specification `Mutator-Spec`, where the `Collector` plays the role of the argument. (This is shown as the dashed subdiagram.) And for the `Collector` the situation is analogous, now with `Mutator` as the argument. (This is shown as the dotted subdiagram.) This mutual recursion is an instance of Lamport's principle of rely/guarantee conditions [5].

The paradigm of program *development* will turn out to be particularly beneficial. For it will be seen that our initial derivation produces a correct solution, albeit based on a non-executable operation (Section 4). Therefore we have to add further complications to both the collector and the mutator. But these additions can be done by *replaying* the original derivation (Section 5).

Notation. The notation is a mixture between SPECWARE [10, 16], OPAL [13] and HASKELL [2], extended by some convenient syntactic sugar. For example, we mimic an object-oriented notation by using the infix function

FUN $\_.\_ : \alpha \times (\alpha \to \beta) \to \beta$
DEF $\mathtt{x.f} = \mathtt{f(x)}$

Functions characterized by the keyword FUN must be implemented, whereas functions characterized by AUX only serve specification purposes and therefore need not be implemented. (To ease reading we write these auxiliary operations in italic.)

The use of a pre-/postcondition style is only syntactic sugar for certain kinds of implicational algebraic equations. For example, the specification of `connect` in Program 2.1 below is equivalent to the axioms

```
AXM G.connect(x, y) NEEDS x ∈ G.nodes ∧ y ∈ G.nodes   -- precondition
AXM G.connect(x, y).sucs(x) = G.sucs(x) ⊕ y            -- essential
AXM G.connect(x, y).sucs(z) = G.sucs(z)   IF x ≠ z     -- may be omitted
AXM G.connect(x, y).nodes = G.nodes                    -- may be omitted
```

In postconditions we often name the arguments and results of functions in a form like `G` and `G′`, respectively. This way we can mimic the specification style of the language Z.

We choose a *coalgebraically* oriented specification style, where types are characterized by their "observations", that is, by their selectors. This will fit more nicely into our later considerations. However, there is currently no generally agreed syntax for coalgebraic descriptions; therefore we use an ad-hoc notation.

The coalgebraic style requires that we specify operations *coinductively.* That is, we must specify the effects of the operation for all selectors ("observers") of the type. In doing so, we frequently encounter a phenomenon that can be seen in the last two of the above axioms. This becomes even more evident if we rewrite them in a higher-order style:

```
AXM sucs(z) ∘ connect(x, y) = sucs(z)   IF x ≠ z
AXM nodes ∘ connect(x, y) = nodes
```

These two axioms simply state that the operation `connect` has *no* effects on the two observers `sucs(z)` and `nodes`. To shorten the presentations we therefore introduce the ***convention*** that *equations for unaffected selectors may be omitted.*

A problem with parameterized specifications. As could already be seen in Fig. 1, a great part of our design is based on parameterized specifications. Here we encounter a subtle difficulty, which we bypass by a small notational convention. In a specification like (see Program 5.5 later on)

```
SPEC Scavenger ( Scavenger-Environment ) =
      ...
   THM* invariant black                        -- only for body
   THM invariant (marked ⊆ white)              -- also for parameter
```

we need to distinguish two kinds of theorems. One kind – in this example *invariant black* – is valid "locally", that is, for the specification `Scavenger`

Program 2.1 Basic specification of graphs

```
  SPEC Graph =                                                              1
    SORT Node                                                               2
    -- the (coalgebraic) type for graphs                                    3
    SORT Graph SELECTORS nodes, sucs                                        4
    FUN nodes : Graph → Set Node                                            5
    FUN sucs : (n : Node) → (G : Graph) → Set Node                          6
        PRE $n \in G.nodes$                                                 7
    -- add / delete arc                                                     8
    FUN connect : (x : Node, y : Node) → (G : Graph) → (G' : Graph)         9
        PRE $x \in G.nodes \wedge y \in G.nodes$                            10
        POST $G'.sucs(x) = G.sucs(x) \oplus y$                              11
    FUN detach : (x : Node, y : Node) → (G : Graph) → (G' : Graph)          12
        PRE $x \in G.nodes \wedge y \in G.nodes$                            13
        POST $G'.sucs(x) = G.sucs(x) \ominus y$                             14
    -- reachability                                                         15
    AUX reachable : (R : Set Node) → (G : Graph) → (S' : Set Node)          16
        PRE $R \subseteq G.nodes$                                           17
        POST $S' = $ LEAST $S.\ (R \subseteq S) \wedge (\cup/(G.sucs) * S \subseteq S)$   18
```

proper. We characterize these properties by writing THM*. The other kind – in this example *invariant* (`marked` ⊆ *white*) has to hold both for the specification `Scavenger` proper *and* for its parameter specification `Scavenger-Environment`. We write these with the usual keyword THM. In Section 6 we will indicate how this notational shorthand is translated into a "clean" algebraic framework.

2 Specification of Graphs and Heaps

We model the garbage collection problem by special graphs.

2.1 Graphs

The basic graphs are specified in Program 2.1.

Explanation of Program 2.1:

2 The type for the nodes is unconstrained.

3-7 Graphs are defined coalgebraically by two selector functions (observers): `nodes` gives the set of nodes of the graph; `sucs` yields for each node in the graph the set of its successors (and thus implicitly the set of arcs).

8-14 We can add and delete arcs from a graph. Since the type `Graph` is specified coalgebraically, the definitions of `connect` and `detach` – which yield a new graph – have to be given *coinductively.* That is, we have to specify the effects of `connect` and `detach` on the two selector functions. (Note that – according to our conventions – the unaffected selector `nodes` and the unchanged part of `sucs` need not be specified here.)

Program 2.2 Specification of heaps

```
SPEC Heap  = EXTEND Graph BY                                          1
  -- the (coalgebraic ) type for heap                                 2
 SORT Heap = EXTEND Graph BY SELECTORS roots, free                    3
 FUN roots : Heap → Set Node                                          4
 FUN free : Heap → Node                                               5
 -- allocate a node from the gray free list                           6
 FUN new : (H : Heap) → (H′ : Heap, n′ : Node)                        7
      PRE   H.gray ≠ ∅                                                8
      POST  n′ ∈ H.gray                                               9
            H′.gray = H.gray ⊖ n′                                    10
            H′.sucs(n′) = ∅                                          11
 -- recycling a white garbage node                                   12
 FUN recycle : (H : Heap) → (H′ : Heap)                              13
      PRE   H.white ≠ ∅                                              14
      POST  H′.gray = H.gray ⊕ n WHERE n ∈ H.white                   15
 -- reachable from the roots                                         16
 AUX black: (H : Heap) → (B : Graph)                                 17
      POST B.nodes = H.reachable(H.roots)                            18
           n ∈ B.nodes ⇒ B.sucs(n) = H.sucs(n)                       19
 -- alternative name for freelist                                    20
 AUX gray: (H : Heap) → Set Node =  H.reachable({H.free})            21
 -- all accessible nodes (black or gray )                            22
 AUX dark: (H : Heap) → Set Node =  H.black ∪ H.gray                 23
 -- totally unreachable nodes (garbage )                             24
 AUX white: (H : Heap) → Set Node =  H.nodes \ H.dark                25
```

15-18 The set of nodes reachable from a given set R of nodes is the smallest set that contains R and is closed under the successor operation.
This operation is only used for specification purposes and therefore need not be implemented. This is expressed by using the keyword AUX instead of FUN.

2.2 Heaps

Garbage collection operates on the so-called *heap* area of the computer memory. These heaps are modelled here as special graphs. For the specification we use a coloring metaphor by speaking of black, gray and white nodes.

Explanation of Program 2.2:

1 Heap is a specialization of Graph.

2-5 Heaps are defined coalgebraically as subtypes of graphs by adding two further selection functions (observers): `roots` gives the set of entry nodes (from the program) into the heap; `free` is the root of the freelist.

6-11 We can allocate a node from the (gray) freelist. This means that we pick a free node n′ and return this node and the (graph with the) remaining freelist. The allocated node has no successors.

Program 2.3 Useful properties of the heap operations

```
SPEC Heap = EXTEND Graph BY                                              1
...                                                                      2
  -- monotonicity of white                                               3
  THM monotone(white)(connect(x, y))   IF y $\notin$ white               4
  THM monotone(white)(detach)                                            5
  THM monotone(white)(new.$\pi_1$)                                       6
  -- invariance of black                                                 7
  THM invariant(black)(recycle)                                          8
  AUX invariant: (f : Heap $\to \alpha$) $\to$ (ev : Heap $\to$ Heap) $\to$ Bool =   9
      $\forall$H: Heap. f(H) = f(ev(H))                                  10
  AUX monotone: (f : Heap $\to$ Set Node) $\to$ (ev : Heap $\to$ Heap) $\to$ Bool =  11
      $\forall$H: Heap. f(H) $\subseteq$ f(ev(H))                        12
```

`12-15` Recycling simply picks a (white) garbage node and adds it to the freelist. (We can leave the detailed organization of the freelist open.)

`16-25` Following a tradition in the literature we introduce a color metaphor to express the partitioning of the graph into reachable nodes (black), the freelist nodes (gray) and the garbage nodes (white). Since we often address both black and gray nodes, we introduce the name "dark" for them. Again, these operations are only used for specification purposes and need not be implemented.
Note that the operation *black* does not only designate the reachable nodes, but the whole reachable *subgraph*. (The reason is that we will have to guarantee later on the invariance of both the black nodes and arcs.)

2.3 Properties of Heaps

During our derivation we will need a number of properties of the operations of `Heap`. Their motivation will only be seen at the points of their applications. Moreover, they are mostly evident. Therefore we only list a representative selection here (see Program 2.3).

Explanation of Program 2.3:

`1-2` The specification `Heap` entails a number of properties, which are added here.

`3-6` The operations `connect`, `detach` and `new` may at most increase the set of white nodes but never decrease it. This is expressed using an auxiliary function *monotone* (see below). Note that for `connect` this is only true, when the second node `y` is not white.

`7-8` The operation `recycle` leaves the set of black nodes invariant. This is expressed using an auxiliary function *invariant* (see below).

`9-10` The invariance of an observer function `f` under an evolution `ev` simply is $\mathtt{f} = \mathtt{f} \circ \mathtt{ev}$.

`11-12` The monotonicity of an observer function `f` under an evolution `ev` simply is $\texttt{f} \subseteq \texttt{f} \circ \texttt{ev}$.

Later on we will take the liberty of writing e.g. *invariant*(*black*)(`nodes`). In these cases, where the second function does not yield a new heap, the invariance and monotonicity are automatically true. (This extension of the two concepts will save some case distinctions later on.)

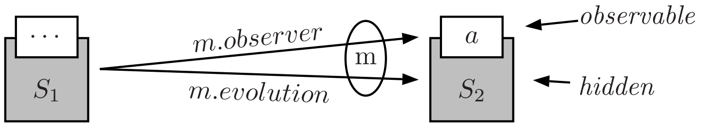
3 A Monad For Heaps

Our ultimate goal is to design two parallel processes that operate on a common heap. This leads into the realm of state-oriented programming. As already mentioned in the introduction, we choose *monads* here to cope with imperative concepts in the context of functional specifications. Their suitability for this purpose will be assessed in Section 6.

3.1 What are Monads?

Following Leibnitz, *monads* are to philosophy what atoms used to be for physics: an entity without further parts [6]. In category theory, *monads* are something closely related to natural transformations [11]. In the world of programming, *monads* are a combination of two long-known concepts, which both go back at least to the 1970's: continuation-based programming (as found e.g. in [20] and implicitly even in [7, 9, 14]) combined with information hiding. The technical details are a little intricate (see the Appendix), but they happen to meet the category-theoretic monad axioms, which justifies the name [19]. Since this is not a paper on monads, we defer their detailed definition to the Appendix and content ourselves here with an informal characterization.

- We use here the so-called state monads; these have an internal (hidden) "*state*", which in our case is the heap.
- A monad `m` essentially is a pair of functions:
  - One function, `m.evolution`, effects a hidden **evolution** of the state.
  - The other function, `m.observer`, allows visible **observations** about the state.



- The type of a monad is denoted as $\texttt{M}[\alpha]$, where α is the type of the observations. (The state is hidden!)

  SORT $\texttt{M}[\alpha]$ SELECTORS *evolution* *observer*
  AUX *evolution*: $\texttt{M}[\alpha] \to (\texttt{State} \to \texttt{State})$
  AUX *observer*: $\texttt{M}[\alpha] \to (\texttt{State} \to \alpha)$

Typical examples are found in the IO-Monad, which encapsulates all the standard input/output commands:

```
SPEC IO = EXTEND Monad(System) RENAME M[α] = Io[α] BY
   -- read a character
   FUN read: Io[Char]
   FUN read.observer =  ≪ content of keyboard register ≫
   FUN read.evolution =  ≪ display keyboard register on screen ≫
...
```

Here the (hidden) evolution even concerns the global state of the operating system. The program can only utilize the visible observation, that is, the character just read.

3.2 Composition of Monads

The fundamental operation on monads is their sequential composition (recall that monads are pairs of functions):

- m$_1$; m$_2$:
  First apply m$_1$: M[α], which leads to some evolution of the internal state (the observable value is ignored). Then m$_2$: M[β] is applied to the new state.

$$\begin{tikzcd}
\begin{pmatrix} \cdots \\ S_0 \end{pmatrix} \arrow[r, bend left=5, "m_1"] \arrow[r] & \begin{pmatrix} a_1 \\ S_1 \end{pmatrix} \arrow[r, bend left=5, "m_2"] \arrow[r] & \begin{pmatrix} a_2 \\ S_2 \end{pmatrix}
\end{tikzcd}$$

- m$_1$; f:
  First apply m$_1$: M[α], which leads to some evolution of the internal state. Then apply the continuation function f : $\alpha \to$ M[β] to the observable value of the new state. This creates the second monad m$_2$: M[β] which is then applied to the new state.

$$\begin{tikzcd}
\begin{pmatrix} \cdots \\ S_0 \end{pmatrix} \arrow[r, bend left=5, "m_1"] \arrow[r] & \begin{pmatrix} a_1 \\ S_1 \end{pmatrix} \arrow[r, bend left=5, "m_2 = f(a_1)"] \arrow[r] & \begin{pmatrix} a_2 \\ S_2 \end{pmatrix}
\end{tikzcd}$$

3.3 Iterators on Monads

Much of the power of monads lies in their flexibility, which makes it easy to add all kinds of useful operations (by contrast to imperative languages, where most of these operators must be predefined by the language). Typical points in case are "exception handlers", "choice operators", "iterators" etc. In this paper we only need two iterators.

```
   -- infinite repetition
FUN forever: (m: M[α]) → M[α] =  m ; forever(m)
   -- repeat as often as possible
FUN iterate: (m: M[α]) → M[α] =
      IF APPLICABLE (m) THEN m ; iterate(m)
                        ELSE nop FI
```

The first operator is used to model continuously running (parallel) systems. The second operator repeats a monadic operation as long as possible. The construct APPLICABLE(`m`) yields false when the monadic operation `m` has a formally stated precondition that does not hold.

3.4 Casting to Monads

Of particular importance for readability is the **automatic casting** of functions to monads. For example, given the function

FUN `round`: `Real` $\to$ `Int`

we automatically associate two monadic operations to it:

FUN `round`: `Real` $\to$ `M`[`Int`]
FUN `round`: `M`[`Real`] $\to$ `M`[`Int`]

Intuitively, the monadic variants apply the original function to the observable values. Functions with more than one argument are treated analogously.

A second casting is even more important in our context. It applies to the hidden internal state. If this state has type `T`, then we lift all functions on `T` to the monad. In our case, where `T` is `Heap`, this means

FUN `f` : `Heap` $\to \alpha$	is lifted to	FUN `f` : `M`[α]
FUN `f` : `Heap` $\to$ `Heap`	is lifted to	FUN `f` : `M`[`Void`]
FUN `f` : `Heap` $\to$ `Heap` $\times \alpha$	is lifted to	FUN `f` : `M`[α]

In the first case, `f` is turned into an observer, and in the second case `f` only effects the corresponding internal state transition. (Since there is nothing to be observed, we need the "empty sort" `Void` here.) The third case has both an observable value and an internal state transition ("side effect"). This lifting also covers situations like `f` : $\alpha \to$ `Heap` $\to \beta$ which is turned into `f` : $\alpha \to$ `M`[β]. And so forth.

The above principles are formally defined in a parameterized specification `Monad(State)`, which is given in the Appendix. In our application we have to instantiate this as `Monad(Heap)`, since the internal state is just the graph.

3.5 Specification of Monadic Heaps

Due to the automatic lifting, the instantiation `Monad(Heap)` provides the following set of operations.

```
FUN nodes: M[Set Node]
FUN sucs: Node → M[Set Node]
FUN connect: Node × Node → M[Void]
FUN detach: Node × Node → M[Void]
FUN roots: M[Set Node]
FUN free: M[Node]
FUN new: M[Node]
FUN recycle: M[Void]
AUX reachable: Set Node → M[Set Node]
AUX black: M[Set Node]
AUX gray: M[Set Node]
AUX dark: M[Set Node]
AUX white: M[Set Node]
```

Most remarkable is the operation `new` which had two results in the original specification and therefore now has an observable value as well as an internal evolution ("side effect").

Along the same lines we can introduce the lifting of the operators *monotone* and *invariant*. Here we have to lift each of the argument functions to monadic form.

```
AUX invariant: (f : M[α]) → (m: M[Void]) → Bool
    POST f.observer = f.observer ∘ m.evolution
AUX invariant: (f : M[α]) → Bool
    POST ∀m: M[α] : invariant(f)(m)
  -- monotonicity is analogous
```

Note that we have added a second form of the operator, which is a shorthand for expressing that `f` is invariant for *all* possible monads. (This is the form that we will mostly employ in the following.)

3.6 Intermediate Assessment of Monads

Given the above definitions, one may get the impression that monads are just a way of mimicking classical imperative programming within the realm of functional programming. Even worse: one can mimic the most problematic feature of imperative programming, namely expressions with side effects.

This is to some extent true. But one can also put this positively: monads allow us to apply imperative-style programming, wherever it is unavoidable (such as input/output, which necessarily operates on the global system state). But we can confine this style to a minimal fragment of our programming, by contrast to imperative languages, where everything has to be programmed state-based.

However, there are more advantages. In monads the side-effects are encapsulated and – above all – typed! In other words, this critical area is supervised by the typing discipline of the compiler (by contrast to imperative languages, where there is just one global hidden state). Moreover, monads are extremely flexible. It is easy to add new and specialized constructs such as iterators or various kinds of exception handlers. By contrast, imperative languages only provide a fixed, predefined set of operators for these issues.

So our claim is that monads not only allow us to extend our algebraic specification techniques to certain kinds of imperative tasks (as will be elaborated in the following case study), but also give us a superior programming model.

4 The Mutator-Collector System – A Simple View

We consider a system consisting of two parallel processes, the *mutator* and the *collector*, which both operate on the same heap (see Program 1.1 in Section 1). The mutator represents the activity of the user's program, which operates on the reachable part of the heap, the collector represents the process that recycles the garbage nodes by adding them to the free list. Both processes work in parallel. The following subsections describe the mutator and the collector in detail.

4.1 The Mutator

The `Mutator` can be almost any program – as long as it only works on the black subgraph. This can be expressed by the specification of Fig. 2, which is an excerpt from the overall specification in Fig. 1. In this diagram we have added

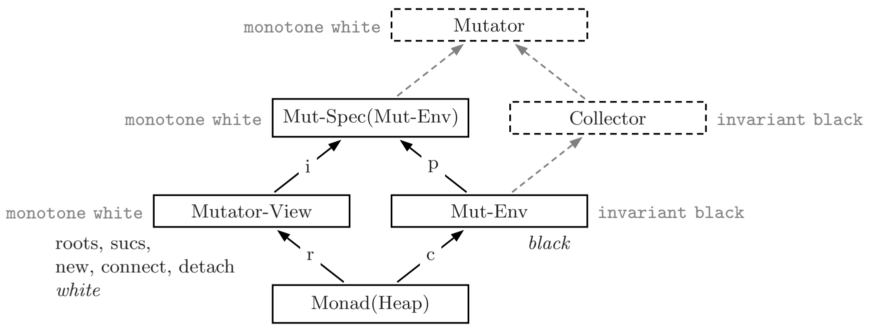


Fig. 2. The mutator's specification

the most relevant aspects of the various specifications and morphisms. In detail:

– The upper diamond represents the instantiation of the parameterized specification `Mut-Spec`(`Mut-Env`) by the argument `Collector`. This requires that the argument `Collector` meets the requirements of the formal parameter `Mut-Env`, in particular the property *`invariant black`*.
– The lower diamond essentially expresses the fact that the mutator and its environment (which will be the collector) have different views of the heap.
– Finally it can be seen that the property *`monotone white`*, which is established by the mutator's view of the heap, is inherited by the mutator's specification and thus also by the mutator itself.

Program 4.1 The mutator's specification

```
  SPEC Mutator-Spec ( Mutator-Environment ) =                        1
    IMPORT Monad(Mutator-View(Heap))                                 2
    FUN mutate : M[Void]                                             3
    THM* monotone white                                              4
  MORPHISM Mutator-View = ONLY roots, sucs, connect, detach, new     5
```

In the following we list the concrete specification texts for the various parts in Fig. 2.

The *mutator* can be an arbitrary program, as long as it only accesses the heap through the restricted set of operations determined by `Mutator-View`. But it expects that the environment (i.e. the collector) does not tamper with its data structures. Therefore the mutator is characterized by the extremely loose parameterized specification of Program 4.1.

Explanation of Program 4.1:

1 The mutator depends on the proper behavior of its environment, which is given here as a parameter (see Program 4.2).

2 The mutator only has a restricted view of the heap. Therefore it bases on a monad that is built over a suitably restricted version of `Heap` (see line 5).

3 The mutator is an arbitrary program (on the restricted view of the heap).

4 The mutator can at most increase the set of white nodes. (This is a "local" theorem, since it is only valid for the `Mutator-Spec` itself, but not for the parameter `Mutator-Environment`.)

5 The mutator has only access to the graph through the roots. It can follow the arcs (using `sucs`), it can add arcs (but only between reachable nodes), and it can delete (reachable) arcs. That is, all operations of the mutator are confined to the black part of the graph. In addition, it may use the freelist for the allocation of "new" nodes.

Correctness We must prove two things. (1) The theorem *monotone white* has to hold. This follows immediately from the fact that all operations in `Mutator-View` have the property *monotone white* as stated in Program 2.3. (2) The precondition `G.connect(x, y)` NEEDS `y` $\notin$ *white* is also fulfilled. This immediately follows from the fact that the operations `roots`, `sucs` and `new` can only yield black or gray nodes.

The mutator's environment According to Lamport's rely/guarantee principle the proper functioning of virtually any instance of the `Mutator` will depend on an acceptable behavior of the environment: it must never modify the existing black subgraph. Therefore the environment of the mutator must be constrained to a view of the heap monad that guarantees this invariant. This is specified in Program 4.2.

Program 4.2 The mutator's environment

```
SPEC Mutator-Environment =
  CONSTRAIN Monad(Heap) BY
  AXM invariant black
```

In the overall system the mutator itself is defined by instantiating the parameterized specification `Mutator-Spec` by the `Collector` (see Fig. 2 and Program 1.1). The instantiation `Mutator-Spec(Collector)` requires that `Collector` is a valid instance of `Mutator-Environment`. Formally this means that there must exist a specification morphism from the parameter `Mutator-Environment` to the instance `Collector` (see Fig. 2). Hence it will be our task in the next section to design the collector in such a way that it meets this requirement. In other words: the collector has to guarantee the property *`invariant black`*.

Summing up. The above Programs 4.1 – 4.2 provide the loosest possible characterization of the idea of "mutator". Any program that meets these – very weak – constraints is an instance of this specification.

4.2 The Collector (Naive View)

The specification of the collector is analogous to that of the mutator. Its structure is depicted in Fig. 3, which also is an excerpt from the overall specification in Fig. 1. As in the mutator diagram in the previous section we have added the

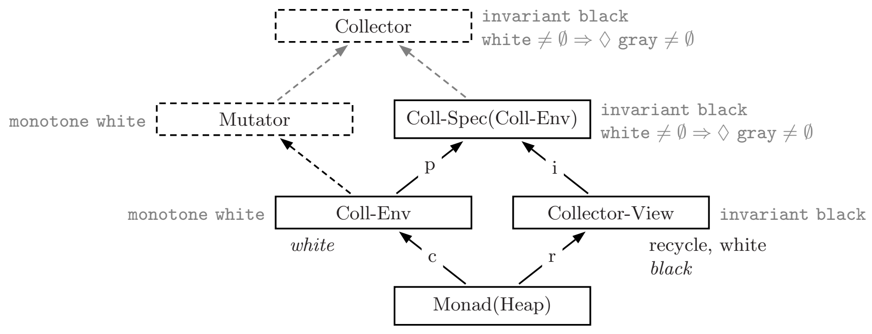


Fig. 3. The collector's specification

most relevant aspects of the various specification boxes and morphism arrows. In detail:

– The upper diamond represents the instantiation of the parameterized specification `Coll-Spec(Coll-Env)` by the argument `Mutator`. This requires that the argument `Mutator` meets the requirements of the formal parameter `Coll-Env`, in particular the property *`monotone white`*.

Program 4.3 The collector's specification

```
SPEC Collector-Spec ( Collector-Environment ) =                    1
  IMPORT Monad(Collector-View(Heap))                               2
  FUN collect : M[Void] = forever(recycle)                         3
  THM* invariant black                                             4
  THM white ≠ ∅ ⇒ ◇ gray ≠ ∅                                       5
MORPHISM Collector-View = ONLY recycle                             6
```

– The lower diamond essentially expresses the fact that the collector and its environment (which will be the mutator) have different views of the heap.
– The property *invariant black*, which is established by the collector's view of the heap, is inherited by the collector's specification and thus also by the collector itself.
– But here we also would like to ensure the *liveness property* that every garbage node will eventually be recycled to the free list. However, this would be an overspecification. Suppose that the timing of the mutator and collector happen to be such that the newest garbage will be continuously recycled and the collector never looks at the old garbage. This is perfectly okay as long as the mutator always obtains a new cell from the freelist, when it needs one. This weaker requirement is captured by the temporal formula `white` $\neq \emptyset \Rightarrow \Diamond$ `gray` $\neq \emptyset$, which states that the gray freelist cannot remain empty, when there are white garbage cells available.

This behavior is captured in Program 4.3. It reflects a naive design, where the collector continuously recycles white nodes.

Explanation of Program 4.3:

1 The collector depends on the proper behavior of its environment, which is given here as a parameter (see below).

2 The collector only has a restricted view of the heap (see below).

3 The collector repeatedly recycles a white (garbage) node to the freelist.

4 The collector does not tamper with the black subgraph. (This is a "local" property.)

5 This liveness property states that the freelist cannot stay forever empty, provided that there are garbage cells available. (Note that this holds "globally", that is, even in the presence of concurrent processes.)

6 The collector is restricted to recycling (white) garbage nodes to the (gray) freelist.

Correctness We have to prove the two theorems. (1) The invariance of the black subgraph follows immediately from the fact that the – only relevant – operation `recycle` has this property as stated in Program 2.3. (2) The proof of the liveness property is also simple: Since we assume a fair merging between the collector and its environment (the mutator), there will always eventually be a `recycle`

Program 4.4 The Collector's Environment

```
SPEC Collector-Environment =
  CONSTRAIN Monad(Heap) BY
  AXM monotone white
```

operation. And when there are garbage cells available, one of them will be added to the freelist. This argument depends on the constraint that the environment does not decrease the set of white nodes.

The collector's environment The collector requires a few constraints for its environment in order to function properly. Essentially the environment must not turn white nodes into black or gray ones. (But it may produce white nodes without interfering with the collector's working.) This is stated formally in Program 4.4.

Summing up. The collector is specified here as a concrete program that continuously recycles white nodes. This meets our intended goals of having a process that assists the mutator without interfering with it. The mutual non-interference is guaranteed by the parameter constraints *monotone white* and *invariant black* that are met by the respective instantiations. Formally:

```
Collector-Spec ⊢ Mutator-Environment
Mutator-Spec   ⊢ Collector-Environment
```

Unfortunately there is a major deficiency in our solution that calls for a much more intricate design, as will be elaborated in the next section.

5 Making the Collector Realistic

Our design so far is correct, but it has a major deficiency: *It is not executable!* The reason lies in the operation `recycle`, on which the collector is based. Recall its definition from Section 2.

```
FUN recycle: (H : Heap) → (H′ : Heap)
    PRE   H.white ≠ ∅
    POST  H′.free = H.free ⊕ n WHERE n ∈ H.white
```

The operation has to pick a white node. But *white* is a non-excutable observer operation! And it is a well-known fact that this operation cannot be made executable easily – in particular, when it is running in parallel to a mutator.

In the following we want to address this issue, that is, we want to make the collector practical. To achieve this goal we need to replace the dependency on the non-executable function *white* by something that is executable and does not sacrifice correctness. In doing so, our major problem is the existence of the concurrently operating mutator.

The emphasis of the following derivation is not so much the topic of concurrent garabge collection as such (this is well-known from the literature). Rather we want to show how the programming can be done by replaying our original design while using appropriately modified (and more complex) specifications. This way we split the overall derivation into two parts: First, we develop an easily understandable but non-executable design; then we extend it to a less understandable, but executable and still correct design. The first part of this process has been demonstrated in the preceding sections, the second part will be shown in the following.

5.1 The Basic Idea

From the literature it is well known that our task cannot be solved without making major changes to the graph and its basic operations: we have to be able to **mark** nodes explicitly. However, we invert the traditional way of proceeding and mark the non-reachable (white) nodes. (As a matter of fact, this is only a change of terminology.)

The underlying idea is illustrated in Fig. 4, which presents a snapshot during the algorithm. On top we have the (black) roots and the (gray) root of the freelist.

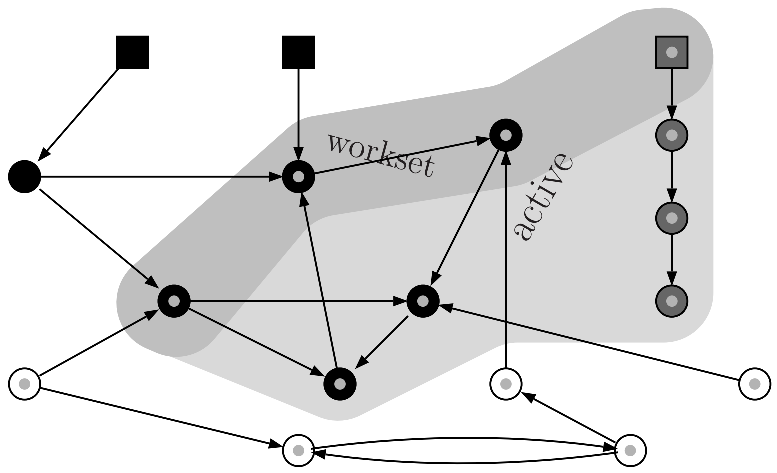


Fig. 4. Cleaning phase (snapshot)

The shaded area represents the *active* nodes, which still need to be visited during the cleaning phase. The dark-shaded subarea represents the current *workset*. Hence, the shaded area of active nodes is defined as the nodes reachable from the workset.

The passive black and gray nodes behind the workset have already been cleaned of their marks, whereas all other nodes – including the white ones – still carry their marks.

Program 5.1 The extended collector

```
SPEC XCollector-Spec ( XCollector-Environment ) =                          1
  IMPORT Cleaner ( XCollector-Environment ) ONLY clean                     2
  IMPORT Scavenger ( XCollector-Environment ) ONLY scavenge                3
  FUN collect : M[Void] = forever(clean ; scavenge)                        4
  THM* invariant black                                                     5
  THM white ≠ ∅ ⇒ ◇ gray ≠ ∅                                               6
```

The *cleaning* phase repeatedly picks a node from the workset, cleans its mark, and adds all its marked successors to the workset. This continues until the workset is empty. Then all remaining marked nodes are white.

At this point we can start the *scavenging* phase, which simply recycles all marked nodes (which are necessarily white) to the freelist.

Of course, in the meanwhile the mutator may have turned some further black nodes into white ones. Therefore the marked nodes will in general only be a subset of the white ones. But this is no problem, since these unmarked white nodes will be caught in the next round of the collector.

Note. In the literature we find two major strategies (see [3]).

– *Snapshot*: At the beginning of each major collector cycle we (conceptually) take a snapshot $\mathtt{W} = \mathit{white}$ of the white nodes. Then we start cleaning the nodes, which finally establishes the property $\mathtt{marked} = \mathtt{W}$. Due to its monotonicity the set white may at most grow during that time, which guarantees our desired goal $\mathtt{marked} = \mathtt{W} \subseteq \mathit{white}$. (This approach is e.g. taken in [21].)

– *Incremental update*: We only guarantee that eventually $\mathtt{marked} \subseteq \mathit{white}$ will be achieved. (This approach has been taken in [1, 17, 4].) We will follow here this approach, essentially in the style of Dijkstra et al. in [1].

The modified collector program This two-phase process is captured in Program 5.1, which is a refinement of the original collector in Program 4.3.

Explanation of Program 5.1:

2-3 The two phases of the new collector are defined in separate specifications (see below). Both rely on the appropriate behavior of the environment.

4 The original naive recycling of white nodes is now replaced by the more elaborate repetition of the two phases `clean` and `scavenge`.

5-6 The new collector must still fulfil the requirements that it does not interfere with the black subgraph and that it will keep the freelist filled as long as possible.

Correctness. We need to show three things. (1) The invariance of the black subgraph is inherited from the specifications of `Cleaner` and `Scavenger`. (2) The liveness property that at least some of the white nodes will eventually make it to the freelist can be shown analogously to our original specification; therefore

Program 5.2 The collector's environment

```
SPEC XCollector-Environment =                                   1
  IMPORT Cleaning-Environment                                   2
  IMPORT Scavenging-Environment                                 3
  AXM monotone white                                            4
  THM invariant marked                                          5
  THM invariant (marked ∩ dark ⊆ active)                        6
```

Program 5.3 The cleaning phase

```
SPEC Cleaner ( Cleaner-Environment ) =                          1
  IMPORT Monad(Cleaning-View(XHeap))                            2
  FUN clean : M[Void] = ( workset ← roots ∪ {free} ; marked ← nodes ;   3
                          iterate(unmark) )                     4
      POST  marked ⊆ white                                      5
  THM* invariant black                                          6
  THM invariant (marked ∩ dark ⊆ active)                        7
MORPHISM Cleaning-View = ONLY nodes, roots, free, marked, workset, unmark   8
```

we omit the proof here. (3) The well-definedness of the program requires that (in line `4`) the precondition of `scavenge` is fulfilled (see Program 5.5). This is ensured by the postcondition of `clean` (see Program 5.3).

The collector's adapted environment The proper working of the collector still depends on properties that must not be violated by the concurrently executing environment (that is, by the mutator). But due to our more intricate design, these requirements are now stronger than before. This is specified in Program 5.2.

Explanation of Program 5.2:

`2-3` The collector's environment (that is, the mutator) must meet the requirements of *both* phases of the collector.

`4` As in the original version we do not want the mutator to recycle white nodes; so it may at most create white nodes.

`5-6` For easier readability we repeat the requirements of the `Cleaner` and the `Scavenger` on the environment here explicitly. They will be explained in the pertinent specifications.

5.2 The Cleaning Phase

The cleaning phase removes the marks from all reachable nodes, including the freelist. This is specified in Program 5.3.

Explanation of Program 5.3:

`2` As all other programs the cleaner has a restricted view of the heap (`XHeap` is specified in Program 5.8).

3 The roots (including that of the freelist) are the initial workset. And all nodes are initially marked. (In an optimized version this would be amalgamated with the preceding scavenging phase.) The notation `marked` $\leftarrow$ `nodes` is a shorthand for expressing that the observer (selector) `marked` now has the value `nodes`.

4 Then we iterate the operation `unmark` until its precondition no longer holds, that is, until the workset has been used up (see Program 5.8 below).

5 When cleaning is finished, only white nodes have marks.

6 The invariance of the black subgraph is guaranteed by the cleaning phase as well.

7 As a central invariant (during the cleaning phase) it is required that all marked non-white nodes will still be visited, that is, are reachable from the workset (which is the meaning of *`active`*).

8 The cleaner only needs operations for initializing the workset and the marked set and for unmarking nodes.

Correctness. We have to prove three things. (1) The invariance of the black subgraph is entailed by the – only relevant – operation `unmark`, since it does not influence the black nodes or edges (see Program 5.8). (2) The property ($\mathtt{marked} \cap \mathit{dark} \subseteq \mathit{active}$) is established by the initial setting of the workspace and the marked set, and is kept invariant by the operation `unmark`. It is also respected by the environment, as stated in Program 5.4. (3) The postcondition of `clean` is indeed established by the definition of the function. This can be shown as follows: The operation `iterate` repeats `unmark` until its precondition is violated. This precondition is $\mathtt{workset} \neq \emptyset$ (see the specification `XHeap` in Program 5.8). Using the definition $\mathit{active} = \mathtt{reachable}(\mathtt{workset})$ and the invariance of $\mathtt{marked} \cap \mathit{dark} \subseteq \mathit{active}$ we can therefore deduce

$$\begin{aligned} &\mathtt{workset} = \emptyset \\ &\vdash\ \mathit{active} = \emptyset \\ &\vdash\ \mathtt{marked} \cap \mathtt{dark} \subseteq \emptyset \\ &\vdash\ \mathtt{marked} \subseteq \mathtt{white} \end{aligned}$$

Note. Strictly speaking, we should also prove the following liveness property: when there are white nodes at the beginning of `clean` then there will be marked nodes at the end of `clean`. This is ensured by the definition of `clean` together with the requirement `invariant marked` on the the environment. In order not to overload the presentation we omit this part from the formal specification.

The cleaner's environment The invariants, on which the cleaner is based, must also be respected by its environment. This is specified in Program 5.4.

5.3 The Scavenging Phase

The scavanging phase relies on the precondition that all marked nodes are white, which has been established by the cleaning phase. Therefore it can safely recycle

Program 5.4 The cleaner's environment

```
SPEC Cleaner-Environment =
  CONSTRAIN Monad(XHeap) BY
  AXM invariant marked
  AXM invariant (marked ∩ dark ⊆ active)
```

Program 5.5 The scavenging phase

```
SPEC Scavenger ( Scavenger-Environment ) =            1
  IMPORT Monad(Scavenging-View(XHeap))                2
  FUN scavenge : M[Void] = iterate(recycle)           3
      PRE marked ⊆ white                              4
  THM* invariant black                                5
  THM invariant (marked ⊆ white)                      6

MORPHISM Scavenging-View = ONLY recycle               7
```

all marked nodes, since this meets the original requirement that white nodes be recycled.

Explanation of Program 5.5:

2	As usual, the scavenger has a restricted view of the heap.
3-4	Scavenging recycles all marked nodes (which are guaranteed to be white due to the precondition).
5	As usual, the invariance of the black subgraph needs to be respected.
6	During the scavenging phase we must ensure that the marked nodes remain white.
7	The scavenger only needs the operation `recycle`.

Correctness. (1) The invariance of the black subgraph follows trivially from the fact that `recycle` only changes the marking and the freelist, but never an edge (see Program 5.8). (2) That the marked nodes are always a subset of the white nodes can be seen as follows. The operation `recycle` only makes white nodes gray when it simultaneously unmarks them (see Program 5.8). And the environment (see Program 5.6) does not change the markings and does not decrease the set of white nodes.

Program 5.6 The scavenger's environment

```
SPEC Scavenger-Environment =
  CONSTRAIN Monad(XHeap) BY
  AXM monotone white
  AXM invariant marked
```

Program 5.7 The adapted mutator

```
SPEC XMutator-Spec ( Mutator-Environment ) =
  IMPORT Monad(Mutator-View(XHeap))
  FUN mutate : M[Void]
  THM* monotone white
  THM* invariant marked
  THM* invariant (marked ∩ dark ⊆ active)
```

5.4 Adapting the Mutator

In our original version the mutator was not even aware of the existence of the collector. This is, unfortunately, no longer true in our new design. As can be seen in the specification `XCollector-Environment` in Program 5.2 above, there are now considerably more intricate requirements that are needed for the proper working of the collector.

There are indeed situations where the mutator can interfere with the collector. This can be seen from the example in Figure 5. A demonic mutator could

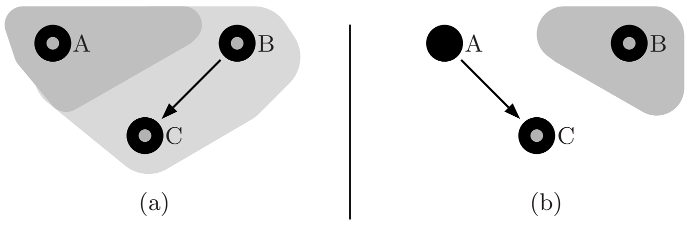


Fig. 5. A demonic mutator

alternate the arc between the two situations (a) and (b). When the node `A` is in the workset and currently considered during situation (a) and later on node `B` is considered during situation (b), then the cleaner will miss the black node `C` and leave it marked. This violates the invariant that all marked dark nodes are active and thus the correctness of the algorithm. Therefore the mutator has to ensure that this invariant is kept intact. (This has been formally required in the specification `XCollector-Environment` in Program 5.2 and therefore could not go undetected in the sequel.)

These stronger constraints in `XCollector-Environment` force us to establish corresponding theorems in the specification of the mutator in order to allow the instantiation of the parameterized specification. Evidently, the deletion of edges does not cause problems. But the addition has to take the marking into consideration. Therefore we have to base the mutator on an adapted view of the heap.

Explanation of Program 5.7:

2 The extended mutator has the same view on the heap as the original mutator (but some operations will be modified as defined in the specification `XHeap` in Program 5.8).

4-6 These are the properties requested by the specification of the collector's environment.

Correctness. As usual, all theorems are essentially inherited from the corresponding properties of the basic operations of `XHeap` in Program 5.8. (1) The monotonicity of the white nodes follows as in the original version. (2) None of the available operations changes the marking. (3) For this property all operations except `connect` are harmless. For `connect` we therefore have to modify its original definition (see `XHeap` in Program 5.8): when `y` is marked and dark, we have to ensure that it is active. (Without this modification it could happen that `x` has already been cleaned and is no longer active; this would leave `y` non-active as well.)

Note that by this new definition `y` may be added to the workset by the mutator, while the collector is in its scavenging phase. Then `y` will enter the next cleaning phase already as a member of the workset. But this is harmless.

5.5 Adapting the Original Heap Specification

The need for marking affects the heap fundamentally. Therefore we have to extend our specification accordingly, naming it `XHeap`.

Explanation of Program 5.8:

1 `XHeap` extends the original specification `Heap` by a few operations and properties. But it also replaces the original operations `recycle` and `connect` by modified definitions.

3 We add a set of marked nodes to the graph.

4 We also add a workset of nodes to the graph.

5-6 The nodes reachable from the workset are "active".

7-12 Unmarking takes a node from the workset, unmarks it, and adds all marked successors to the workset.

13-18 Recycling now takes a marked node (which – during the algorithm – is known to be white) and adds it to the freelist.

19-22 The addition of arcs has to be redesigned according to Fig. 5. If we connect a marked node to an unmarked node, we have to ensure that the marked node is active. This is most easily achieved by adding it to the workset.

As a result of these extensions the new sort `XHeap.Heap` is a subtype of the old sort `Heap.Heap`. (In object-oriented terminology this is referred to as inheritance.)

Program 5.8 The extended heap

```
SPEC XHeap = EXTEND Heap EXCEPT recycle, connect BY                      1
  SORT Heap EXTEND BY SELECTORS marked, workset                          2
  FUN marked : Heap → Set Node                                           3
  FUN workset : Heap → Set Node                                          4
  -- reachable from workset is called active                             5
  AUX active : (H : Heap) → Set Node = (H.reachable)(H.workset)          6
  -- unmarking has to be provided                                        7
  FUN unmark : (H : Heap) → (H′ : Heap)                                  8
      PRE   H.workset ≠ ∅                                                9
      POST  LET n ∈ H.workset IN                                         10
            H′.workset = H.workset ∪ (H.sucs(n) ∩ H.marked) ⊖ n          11
            H′.marked = H.marked ⊖ n                                     12
  -- recycling needs to be modified                                      13
  FUN recycle : (H : Heap) → (H′ : Heap)                                 14
      PRE   H.marked ≠ ∅                                                 15
      POST  LET n ∈ H.marked IN                                          16
            H′.marked = H.marked ⊖ n                                     17
            H′.gray = H.gray ⊕ n                                         18
  -- connecting need to be modified                                      19
  FUN connect : (x : Node, y : Node) → (G : Heap) → (G′ : Heap)          20
      POST G′.sucs(x) = G.sucs(x) ⊕ y                                    21
           x ∉ G.marked ∧ y ∈ G.marked  ⇒  G′.workset = G.workset ⊕ y    22
```

5.6 Properties of XHeap

As a result of our modifications and extensions, the operations of `XHeap` now have all the properties that we presupposed during our development. Since most of them are evident, we only list a representative selection in Program 5.9.

Explanation of Program 5.8:

`1` We add some useful properties to our specification `XHeap`.

`3-4` The new operations needed for the collector also respect the invariance of the black subgraph.

`5` Recycling respects the main invariant of the scavenging phase.

Program 5.9 Properties of the extended heap

```
SPEC XHeap = EXTEND Heap EXCEPT recycle, connect BY                      1
...                                                                      2
  THM invariant(black)(unmark)                                           3
  THM invariant(black)(recycle)                                          4
  THM invariant(marked ⊆ white)(recycle)                                 5
  THM invariant(marked)(connect)                                         6
  THM monotone(white)(connect)                                           7
...                                                                      8
```

6-7 The new `connect` operation respects the fundamental requirements of the collector.

5.7 Variations on the Theme

Line 22 of Program 5.8 deserves further discussion, since it is the place where the various algorithms in the literature differ (see [3]).

- The most conservative approach is taken by Dijkstra et al. in [1]. They add `y` to the workset even when it is not marked. That is, line 22 has no longer a precondition and thus reads
  $\mathtt{G}'.\mathtt{workset} = \mathtt{G.workset} \oplus \mathtt{y}$
- While Dijkstra's algortihm pushes the wave-front of the workset forward, Steele [17] pushes it back a little by adding the source node `x` to the workset instead of the target node `y`. Here, line 22 reads
  $\mathtt{x} \notin \mathtt{G.marked} \wedge \mathtt{y} \in \mathtt{G.marked} \Rightarrow \mathtt{G}'.\mathtt{workset} = \mathtt{G.workset} \oplus \mathtt{x}$
  Termination is a little trickier here, but this variant is likely to catch more garbage cells in each cycle. (Note: In Steele's algorithm the mutator only makes the node active when the collector is in its cleaning phase. So it must also be aware of the collector's current phase.)
- If we wanted to mimic the snapshot algorithm of Yuasa [21], we had to change the operation `detach(x, y)` as well. Since `y` was in the black set, when we made the snapshot `W`, we have to clean it as well. That is, `y` has to be added to the workset.

6 Conclusion: Assessment of Methodology

The underlying principle of our derivation is the development of programs by refinement and composition of specifications as it has been worked out thoroughly in the Specware methodology and tools. The fundamental notion here is that

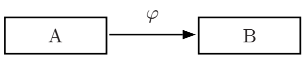

$$\begin{tikzcd} \boxed{A} \arrow[r, "\varphi"] & \boxed{B} \end{tikzcd}$$

Fig. 6. Morphism

of a **morphism** φ from a specifcation `A` to a specification `B` (see Figure 6). Such a morphism essentially has two aspects:

1. *Syntactic part*: Each sort and function symbol of `A` has to be mapped to a corresponding sort and function symbol of `B` such that the proper typing of the functions is preserved.
2. *Semantic part*: Every property of `A` must be retained in `B` (modulo the syntactic renaming). In other words, `B` is more constrained than `A`.

It should be kept in mind that the stronger constraints in `B` entail that `B` has fewer models.

Parameterized Specifications Parameterized specifications and their instantiations can be represented by diagrams of the form depicted in Figure 7. The for-

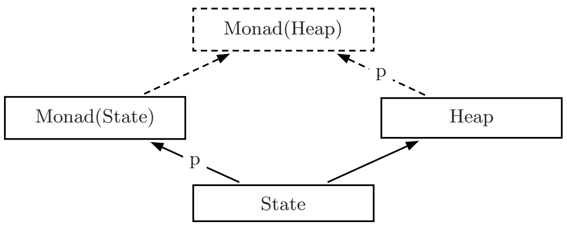


Fig. 7. Instantiation of parameterized specifications by colimit

mal parameter `State` is included in the parameterized specification `Monad(State)`. This inclusion is depicted by the special arrow $\xrightarrow{p}$. If another specification – here `Heap` – shall be used as an argument, then it must meet all requirements of the parameter (up to renaming). In other words, there must be a morphism from the parameter `State` to the argument `Heap`. For such a diagram with two morphism we can form the **colimit**, which automatically builds the instantiated specification `Monad(Graph)`.

However, there was a complication in several of our examples, which led us to the shorthand notation THM* for the "locally" valid theorems. This corresponds to the situation of Fig. 8 (where the occurences of `M-Spec` on the left and right refer to the same specification). Consider the situation in the middle.

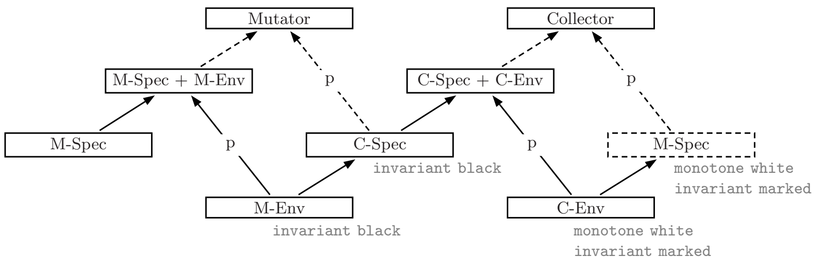


Fig. 8. The role of "local" theorems

The specification `C-Spec` fulfills the property `invariant black`, the specification `C-Env` fulfills the properties `monotone white` and `invariant marked`. But their combined specification `C-Spec` + `C-Env` possesses neither property.

To ease readability we did not want to introduce too many little specifications that are only needed for technical reasons. Therefore we preferred to write the diagram of Fig. 8 in the compactified form of the diagram in Fig. 9. Here we amalgamate e.g. `C-Spec` and `C-Spec` + `C-Env` into a single parameterized

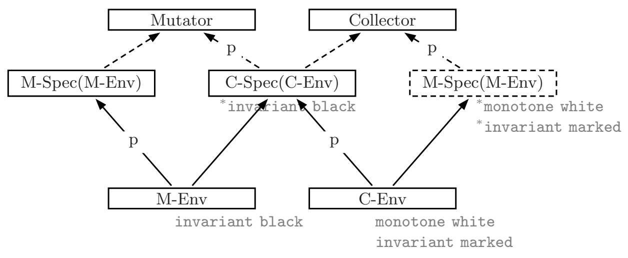


Fig. 9. Compact form of Fig. 8

specification `C-Spec(C-Env)`. Analogously for `M-Spec`. The price to be paid is that we need to introduce special syntax for those properties that in Fig. 8 were encapsulated in `C-Spec`.

"Replaying" In the second part of our case study we had to modify our original specification `Heap` into the specification `XHeap`. This induces e.g. the subdiagram in Fig. 10. The morphism from `Heap` to `XHeap` induces the other morphisms as

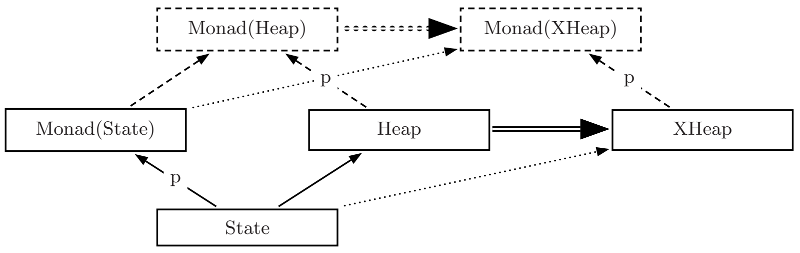


Fig. 10. Instantiation of parameterized specifications by colimit

well. This way we obtain a "copy" of the whole development, now based on `XHeap` instead of `Heap`.

References

1. E. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Comm. ACM*, 21(11):965–975, Nov. 1978.
2. P. Hudak. *The Haskell School of Expression*. Cambridge University Press, 2000.
3. R. Jones and R. Lins. *Garbage Collection*. John Wiley and Sons, 1996.
4. H. Kung and S. Song. An efficient parallel garbage collection system and its corectness proof. In *IEEE Symp. on Foundations of Comp. Sc.*, pages 120–131. IEEE Press, 1977.
5. L. Lamport. Simple approach to specifying concurrent systems. *Comm. ACM*, 32(1):32–45, 1989.

6. G. W. Leibnitz. *Vernunftprinzipien der Natur und der Gnade – Monadologie*. Felix meiner Verlag, 1982.
7. Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.
8. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, 1992.
9. J. McCarthy. Towards a mathematical science of computation. In C. Popplewell, editor, *Information Processing 6*, pages 21–28. North-Holland, 1963.
10. J. McDonald and J. Anton. Specware – producing software correct by construction. Technical Report KES.U.01.4, Kestrel Institute, Palo Alto, March 2001.
11. E. Moggi. Notions of computation and monad. *Information and Computing*, 93:55–92, 1991.
12. D. Pavlovic and D. R. Smith. Composition and refinement of behavioral specifications. In *Proceedings of Sixteenth International Conference on Automated Software Engineering*, pages 157–165. IEEE Computer Society Press, 2001.
13. P. Pepper. *Funktionale Programmierung in Opal, ML, Haskell und Gofer*. Springer Verlag, 2002.
14. J. Reynolds. On the relation between direct and continuation semantics. In J. Loeckx, editor, *Proc. 2nd Coll. on Automata, Languages, and programming*, Lecture Notes in Computer Science 14, pages 141–156. Springer Verlag, 1974.
15. D. Smith. Designware: Software development by refinement. In *Proc. Eighth Int. Conf. on Category Theory and Computer Science, Edinburgh*, Sept. 1999.
16. Specware. Documentation `www.specware.org/doc.html`. 2002.
17. G. Steele. Multiprocessing compactifying garbage collection. *Comm. ACM*, 18(9):495–508, Sep. 1975.
18. G. Tel, R. Tan, and J. van Leeuwen. The derivation of graph marking algorithms from distributed termination detection protocols. *Science of Comp.Progr.*, (10):107–137, 1988.
19. P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.
20. M. Wand. Continuation-based program transformation strategies. *J.ACM*, 27(1):164–180, 1980.
21. T. Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, 11(3):181–190, 1990.

A Appendix: More on Monads

This is not a paper on monads. But to make it self-contained, we have to specify our notion of monads (which slightly varies from the classical views given in the literature.) Monads are a combination of continuation-based programming with the principle of information hiding. This is directly reflected in our specification in Program A.1.

Explanation of Program A.1:

1 Monads are based on some internal type that shall be hidden. This type is often referred to as the "state". Since there are many possibilities for concrete kinds of "state", we parameterize the whole specification by this type.

Program A.1 Specification of monads

```
SPEC Monad (TYPE State) =                                              1
  -- The (generic )monad type                                          2
  SORT M[$\alpha$]SELECTORS evolution observer                         3
  -- coalgebraic view                                                  4
  AUX evolution: M[$\alpha$] $\to$ (State $\to$ State)                 5
  AUX observer: M[$\alpha$] $\to$ (State $\to \alpha$)                 6
  -- yield a value                                                     7
  FUN yield[$\alpha$] : $\alpha \to$ M[$\alpha$]                       8
  DEF (yield a).evolution = id                                         9
  DEF (yield a).observer = K a                                         10
  -- composition of monads                                             11
  FUN ( _ ; _ )[$\alpha, \beta$] : M[$\alpha$] $\times$ M[$\beta$] $\to$ M[$\beta$]        12
  DEF ($m_1$ ; $m_2$).evolution = $m_2$.evolution $\circ$ $m_1$.evolution                 13
  DEF ($m_1$ ; $m_2$).observer = $m_2$.observer $\circ$ $m_1$.evolution                   14
  -- composition of monad with continuation                            15
  FUN ( _ ; _ )[$\alpha, \beta$] : M[$\alpha$] $\times$ ($\alpha \to$ M[$\beta$]) $\to$ M[$\beta$]   16
  DEF $m_1$ ; f = (f $\circ$ $m_1$.observer) S ($m_1$.evolution)                        17
  -- evaluation of the monad                                           18
  FUN eval[$\alpha$] : M[$\alpha$] $\to$ State $\to \alpha$            19
  DEF eval(m)(init) = (m.observer)(init)                               20
```

3 Monads provide a generic type `M`[α], where (each instance of) α represents the observable values. For example, in the input/output monad `readInt` would be of type `M[Int]`.

5-6 The type `M`[α] is coalgebraically characterized by two functions. *evolution* yields a function for state transitions and *observer* yields a function for observing the states.

8-10 The first fundamental operation on monads is `yield(a)`, which leaves the state unchanged and makes `a` the next observable value. (The operator `K` is the constant combinator `K a b` = `a`.)

12-13 The second fundamental operation on monads is their sequential composition. The combined evolution effected by (m_1 ; m_2) is the composition of their individual evolutions. And the combined observer is the observer of m_2 after the evolution of m_1.

16-17 One often does not have a fixed second monad m_2 in the sequential composition; rather one constructs it from the observable value `a` of the first monad m_1 with the help of a continuation function `f`; that is, m_2 = `f(a)`. This can be expressed very concisely with the help of the `S`-combinator (`g S h`)`x` = (`g x`)(`h x`).

19-20 Evaluation of a monad `m` in some initial state `init` simply is the application of `m`'s observer function to `init`. Since `m` usually is a long sequence of monadic operations, `eval` yields the observation at the end of this sequence.

A.1 Lifting to Monads: Atomic Monads

In Section 3 we have presented various liftings for "normal" operations to monadic operations. Now we want to show the formal definitions: For example

FUN `round`: `Real` → `Int`

has the two monadic liftings

FUN `round`: (`a`: `Real`) → `M[Int]` = `yield(round a)`

FUN `round`: (`m`: `M[Real]`) → `M[Int]` = `yield(round m.`*observer*`)`

The second kind of automatic lifting for `Monad(Heap)` applies to functions on `Heap`. Note the overloading!

```
FUN f : Heap → α       is lifted to   FUN  f : M[α]
                                      DEF f.observer = λH:Heap . f(H)
                                      DEF f.evolution = id
FUN f : Heap → Heap    is lifted to   FUN  f : M[Void]
                                      DEF f.observer = K nil
                                      DEF f.evolution = λH:Heap . f(H)
```

Note that this lifting also covers situations like `f` : α → `Heap` → β which is turned into `f` : α → `M`[β]. And so forth.

This lifting provides an important notion that we will need in the following to make some other notions precise: A monad `m` that has been obtained through lifting is called an ***atomic monad***. This includes `yield` (which is the lifting of (`K a`)). In other words, the atomic monads are all those that are not obtained by composition operators such as ';' (and others that we have not mentioned in this paper).

A.2 Temporal Logic for Monads

It may be elucidating to review the situation in the light of **temporal logic** [8]. In a liberal notation, where the nexttime operator '$\bigcirc$' can also be applied to non-boolean values, the preservation property may be written as

$$\Box\,(\mathtt{f}_{\mathtt{observer}} = \bigcirc\,\mathtt{f}_{\mathtt{observer}})$$

However, this kind of formula is not quite true. The reason is subtle. The temporal operator $\Box$ implicitly quantifies over all states of the computation, whereas our formula contains an additional quantification over all monadic operations. Now recall that most of our specifications declare "local" monadic types that are subtypes of the overall type `M`[α]. This induces an implcit filter for the quantification; that is, we only refer to certain monadic operations. Consequently, the states are restricted to those appearing immediately before and after these selected monads.

Explanation of Program A.2:

`3-6` Invariance preserves equality under the observation `f`.

`7-10` Monotonicity preserves the order under the observation `f`. (For simplicity we have restricted the operator to the only ordering that we need in our paper, namely the subset relation.)

Program A.2 Temporal operators for monads

```
SPEC Monad (TYPE State) =
    ...
  -- invariance
  AUX □ : (p : M[Bool]) → Bool
      PRE   p.evolution = id
      POST  ∀m : M[α] :  p.observer ∘ m.evolution = true
  -- monotonicity
  AUX ○ : (f : M[α]) → M[α]
      PRE   f.evolution = id
      POST  ∀m : M[α] :  f.observer = f.observer ∘ m.evolution
```

This phenomenon is well-known in temporal logic and has been addressed in various ways, the simplest of which is the use of predicates at $\pi \Rightarrow \ldots$, where the at operator refers to the control points in the program. We feel that the typing discipline provided by the monads is a more elegant solution to this problem than the reference to the good old program counter. But this relationship needs further investigation.

A.3 Concurrency for Monads

Finally it remains to consider the parallel operator.

FUN _||_ : M[α] $\times$ M[α] $\rightarrow$ M[α]

A full technical definition goes beyond the scope of this paper (it would include a discussion of the associativity of ';' and of the notion of atomic monadic operation). But the idea is straightforward: we may consider the parallel composition as the interleaving of the two monads, where the sequential composition operator ';' determines the interleaving points.

As a matter of fact, the monadic view provides a very clean approach to interleaving. In A $\|$ B each of the two processes (monads) A and B ultimately is a *word* over the atomic monadic operations (in our application add, cut, new etc.), where ';' acts as the concatenation operator. Therefore computational interleaving is indeed modelled as the interleaving of the two words.

In connection with our approach of composing specifications we obtain another nice feature. Recall that the two monads A and B usually are words over different subtypes of M[α], which has the aforementioned advantages for verification. The interleaving then yields a word over the full type.