Programming Concurrent Garbage Collectors With Pushouts and Monads

Dusko Pavlovic, Peter Pepper, Doug Smith

Technische Universität Berlin

and

Kestrel Institute

1. System Diagram
2. Graphs
3. Monads
4. The Mutator
5. The Collector
6. Implementation Problems
7. The Extended Collector
   7.1 Cleaning (Mark)
   7.2 Scavenging (Scan)
8. The Extended Mutator
9. Assessment of the Method

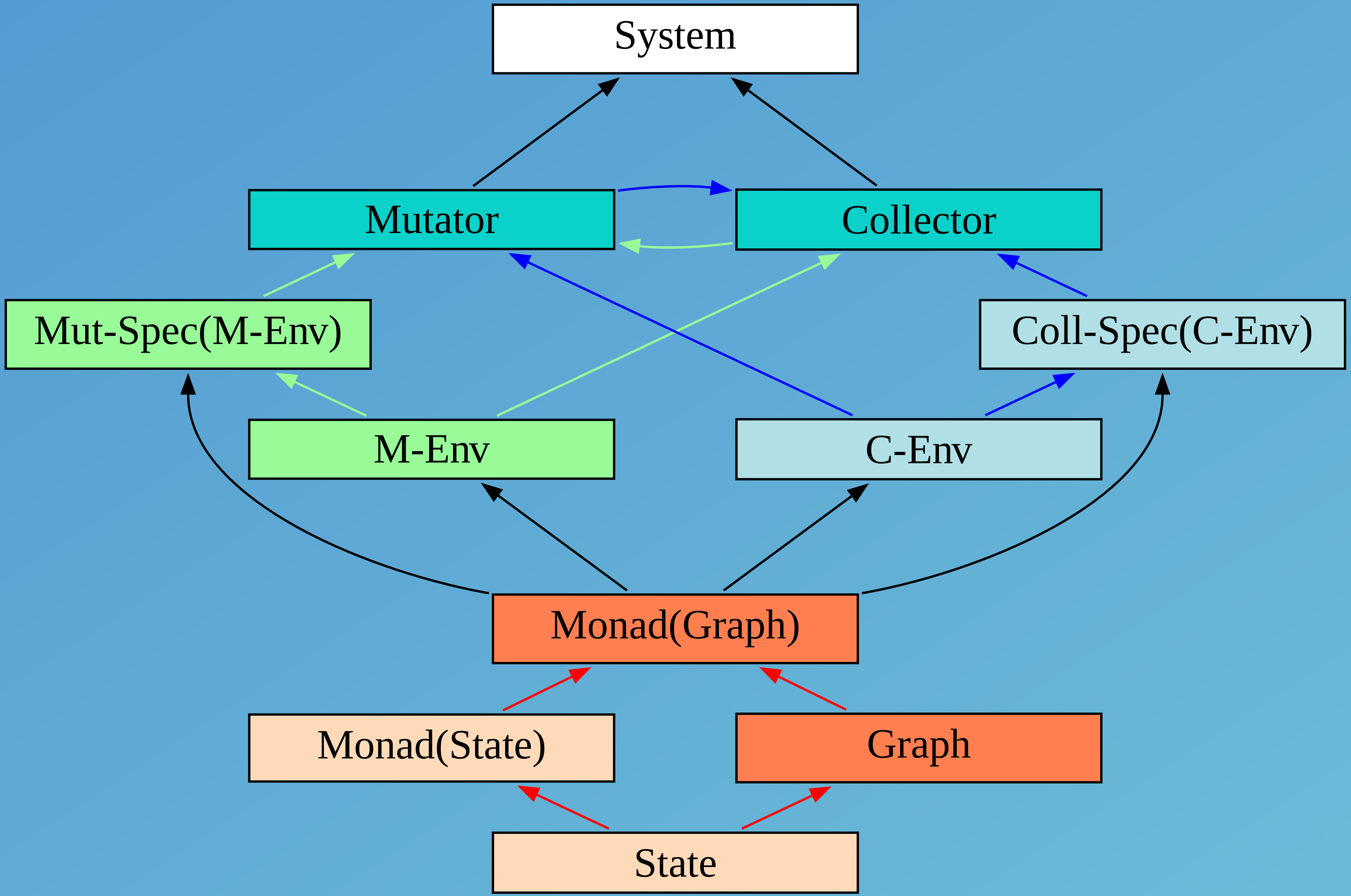
1 The Overall System

☞ The system is a parallel combination of

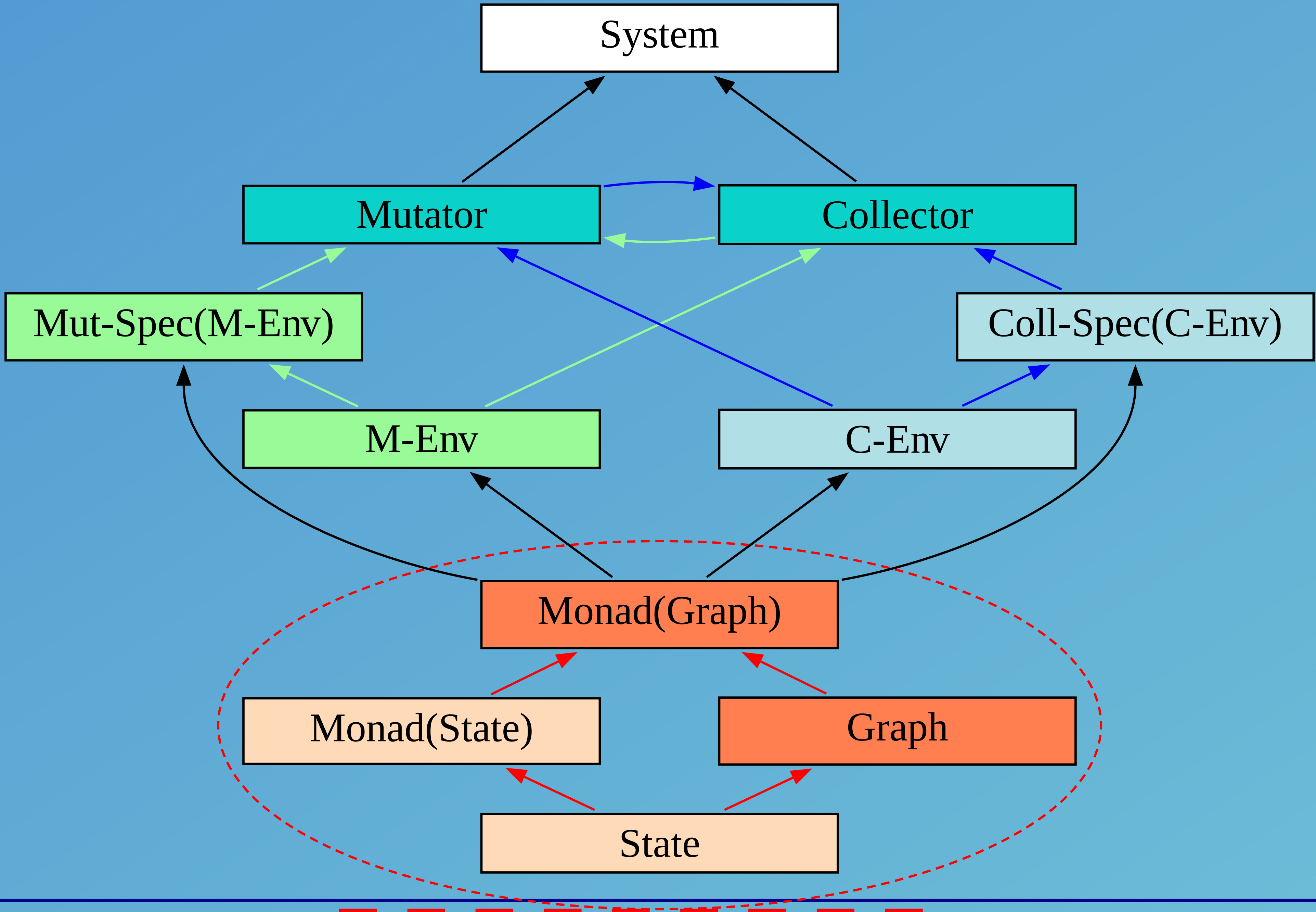
➢ the mutator (the "program")

➢ the collector (the "garbage collector")

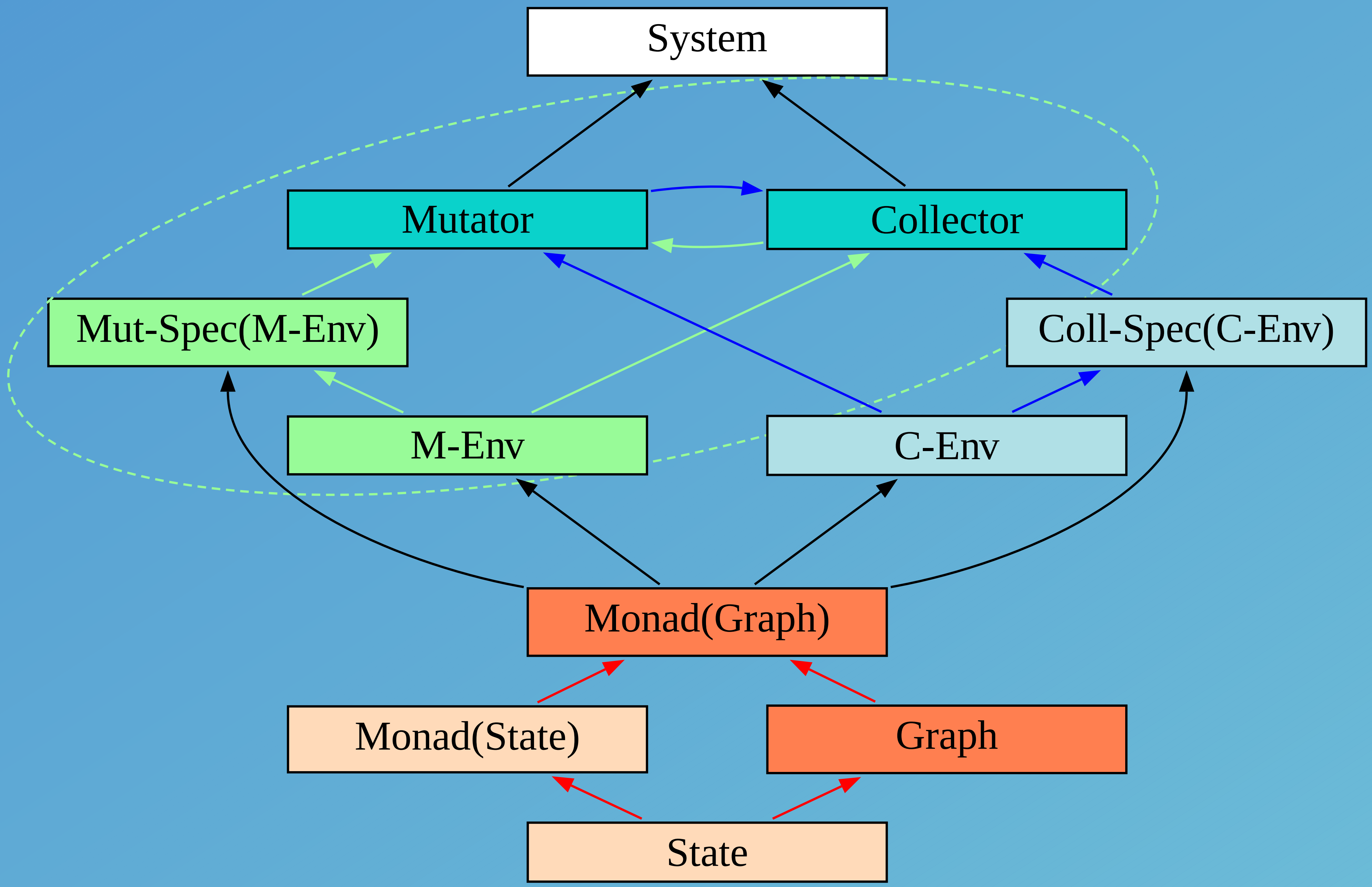
The Overall System



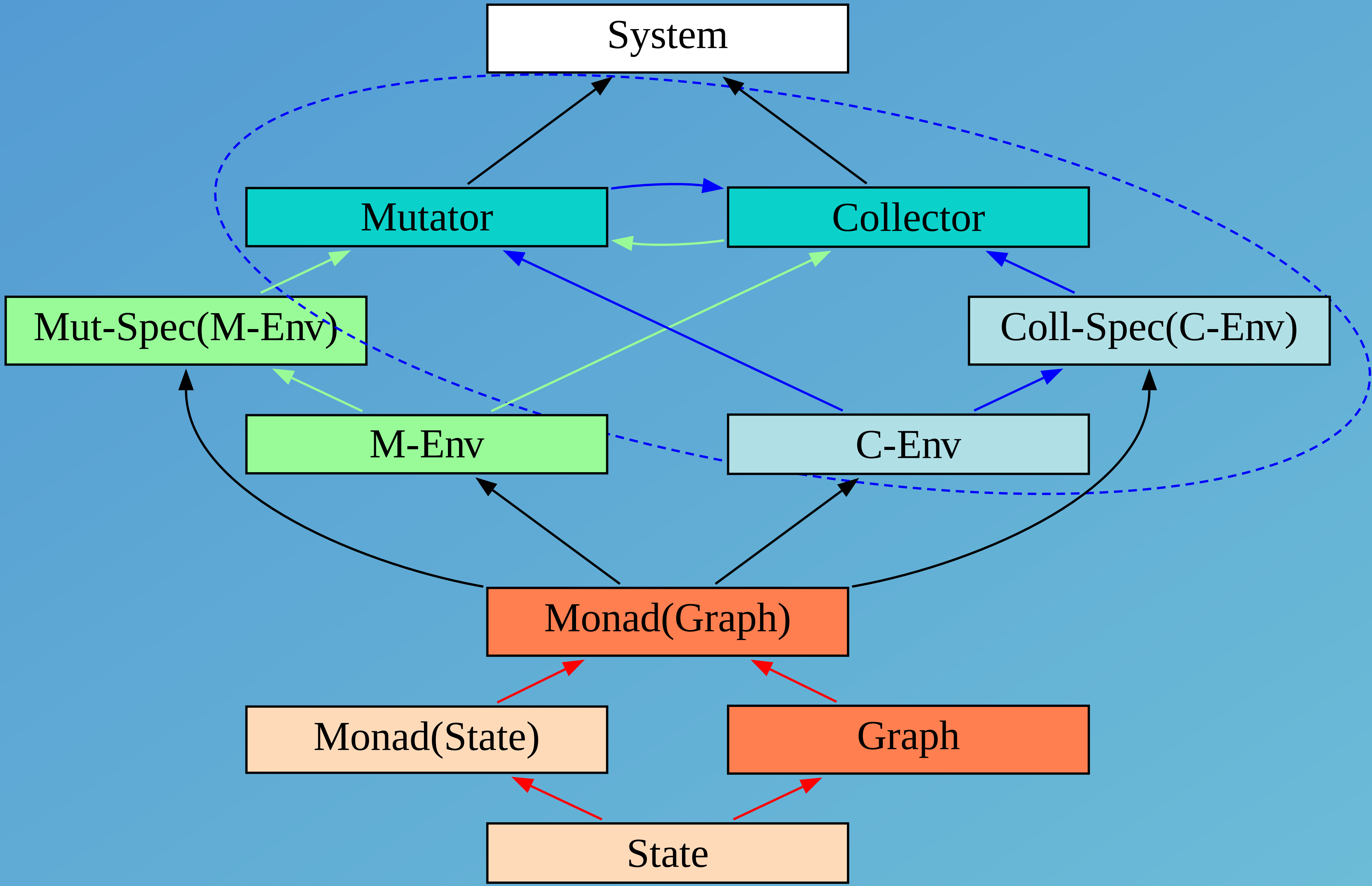
The Overall System



The Overall System



The Overall System



The Overall System Specification

SPEC System =
  IMPORT Mutator, Collector
  FUN run : M[Void] = (mutate ‖ collect)   -- *parallel*

SPEC Mutator = Mutator-Spec(Collector)

SPEC Mutator-Spec(Env : Mutator-Environment) = . . .

The Overall System Specification

$$\text{SPEC } \mathtt{System} =$$
$$\quad \text{IMPORT } \mathtt{Mutator}, \mathtt{Collector}$$
$$\quad \text{FUN } \mathtt{run} : \mathtt{M}[\mathtt{Void}] = (\ \mathtt{mutate}\ \|\ \mathtt{collect}\ ) \qquad \text{-- } \textit{parallel}$$

$$\text{SPEC } \mathtt{Mutator} = \mathtt{Mutator\text{-}Spec}(\mathtt{Collector})$$

$$\text{SPEC } \mathtt{Collector} = \mathtt{Collector\text{-}Spec}(\mathtt{Mutator})$$

$$\text{SPEC } \mathtt{Mutator\text{-}Spec}(\mathtt{Env} : \mathtt{Mutator\text{-}Environment}) = \ldots$$

$$\text{SPEC } \mathtt{Collector\text{-}Spec}(\mathtt{Env} : \mathtt{Collector\text{-}Environment}) = \ldots$$

2 Graph Specification

Graphs are modelled here by

- ☞ a fixed set of nodes
- ☞ a set of entry points ("roots")
- ☞ a successor function (representing the arcs)
- ☞ a freelist

```
SPEC Graph =
    ...
```

```
SORT Node
  -- the (coalgebraic ) type for graphs
SORT Graph OBSERVED BY roots, sucs, free
FUN roots : Graph → Set Node
FUN sucs : Node → Graph → Set Node
FUN free : Graph → Set Node
```

SORT Node

-- *the (coalgebraic) type for graphs*

SORT Graph OBSERVED BY roots, sucs, free

FUN roots : Graph $\longrightarrow$ Set Node

FUN sucs : Node $\longrightarrow$ Graph $\longrightarrow$ Set Node

FUN free : Graph $\longrightarrow$ Set Node

-- *reachability*

OBS reachable : $(R : \text{Set Node}) \longrightarrow (G : \text{Graph}) \longrightarrow (S' : \text{Set Node})$

POST $S' = \text{LEAST } S.\ (R \subseteq S) \wedge (\cup/(G.\text{sucs}) * S \subseteq S)$

-- *reachable from root*

OBS black : $(G : \text{Graph} \longrightarrow \text{Set Node}) = G.\text{reachable}(G.\text{roots})$

-- *alternative name for freelist*

OBS gray : $(G : \text{Graph} \longrightarrow \text{Set Node}) = G.\text{free}$

-- *totally unreachable nodes (garbage)*

OBS white : $(G : \text{Graph} \longrightarrow \text{Set Node}) = \{ n : \text{Node} \} \setminus (G.\text{black} \cup G.\text{gray})$

AXM 1 : $G.\text{black} \cap G.\text{gray} = \emptyset$

-- *add /delete arc (coinductive definition)*

FUN $\mathtt{add} : (\mathtt{x} : \mathtt{Node}, \mathtt{y} : \mathtt{Node}) \longrightarrow (\mathtt{G} : \mathtt{Graph}) \longrightarrow (\mathtt{G}' : \mathtt{Graph})$

POST $\mathtt{G}'.\mathtt{sucs}(\mathtt{x}) = \mathtt{G}.\mathtt{sucs}(\mathtt{x}) \oplus \mathtt{y}$

FUN $\mathtt{cut} : (\mathtt{x} : \mathtt{Node}, \mathtt{y} : \mathtt{Node}) \longrightarrow (\mathtt{G} : \mathtt{Graph}) \longrightarrow (\mathtt{G}' : \mathtt{Graph})$

POST $\mathtt{G}'.\mathtt{sucs}(\mathtt{x}) = \mathtt{G}.\mathtt{sucs}(\mathtt{x}) \ominus \mathtt{y}$

-- *add / delete arc (coinductive definition)*

$\texttt{FUN}\ \texttt{add} : (\texttt{x} : \texttt{Node}, \texttt{y} : \texttt{Node}) \longrightarrow (\texttt{G} : \texttt{Graph}) \longrightarrow (\texttt{G}' : \texttt{Graph})$

$\quad \texttt{POST}\ \texttt{G}'.\texttt{sucs}(\texttt{x}) = \texttt{G}.\texttt{sucs}(\texttt{x}) \oplus \texttt{y}$

$\texttt{FUN}\ \texttt{cut} : (\texttt{x} : \texttt{Node}, \texttt{y} : \texttt{Node}) \longrightarrow (\texttt{G} : \texttt{Graph}) \longrightarrow (\texttt{G}' : \texttt{Graph})$

$\quad \texttt{POST}\ \texttt{G}'.\texttt{sucs}(\texttt{x}) = \texttt{G}.\texttt{sucs}(\texttt{x}) \ominus \texttt{y}$

-- *obtaining a free node*

$\texttt{FUN}\ \texttt{new} : (\texttt{G} : \texttt{Graph}) \longrightarrow (\texttt{G}' : \texttt{Graph},\ \texttt{n} : \texttt{Node})$

$\quad \texttt{PRE}\ \ \texttt{G}.\texttt{free} \neq \emptyset$

$\quad \texttt{POST}\ \texttt{n} \in \texttt{G}.\texttt{free}$

$\quad\quad \texttt{G}'.\texttt{free} = \texttt{G}.\texttt{free} \ominus \texttt{n}$

$\quad\quad \texttt{G}'.\texttt{sucs}(\texttt{n}) = \emptyset$

-- *recycling a garbage node*

$\texttt{FUN}\ \texttt{recycle} : (\texttt{G} : \texttt{Graph}) \longrightarrow (\texttt{G}' : \texttt{Graph})$

$\quad \texttt{PRE}\ \ \texttt{G}.\texttt{white} \neq \emptyset$

$\quad \texttt{POST}\ \texttt{G}'.\texttt{free} = \texttt{G}.\texttt{free} \oplus \texttt{n}\ \texttt{WHERE}\ \texttt{n} \in \texttt{G}.\texttt{white}$

3 Monads

Modelling of imperative aspects by monads

- ☞ The graph becomes the (hidden) state
- ☞ Parallel execution of mutator/collector is interleaving of monadic operations.

- Monads are parameterized by an internal `State`
- Monads provide a polymorphic type $\mathtt{M}[\alpha]$

```
SPEC Monad (TYPE State) =
  TYPE M[α]
```

- Monads are parameterized by an internal `State`
- Monads provide a polymorphic type $\mathtt{M}[\alpha]$

$$\mathrm{SPEC}\ \mathtt{Monad}\ (\mathrm{TYPE}\ \mathtt{State}) =$$
$$\quad \mathrm{TYPE}\ \mathtt{M}[\alpha]$$

The monad type $\mathtt{M}[\alpha]$ consists of two functions

- the evolution of the internal (hidden) state
- the observation of the (visible) associated value

$$\quad \texttt{--}\ \textit{coalgebraic view of } M\,[\alpha]$$
$$\mathrm{OBS}\ \mathtt{evolution} : \mathtt{M}[\alpha] \longrightarrow (\mathtt{State} \rightarrow \mathtt{State})$$
$$\mathrm{OBS}\ \mathtt{observer} : \mathtt{M}[\alpha] \longrightarrow (\mathtt{State} \rightarrow \alpha)$$

We can lift a value to a monad (with no evolution)

$$\mathrm{FUN}\ \mathtt{yield}[\alpha] : \alpha \longrightarrow \mathtt{M}[\alpha]$$
$$\mathrm{DEF}\ (\mathtt{yield}\ \mathtt{a}).\mathtt{evolution} = \mathtt{id}$$
$$\mathrm{DEF}\ (\mathtt{yield}\ \mathtt{a}).\mathtt{observer} = \mathtt{K}\ \mathtt{a}$$

We can lift a value to a monad (with no evolution)

$$\text{FUN } \texttt{yield}[\alpha] : \alpha \longrightarrow \texttt{M}[\alpha]$$
$$\text{DEF } \big(\texttt{yield a}\big).\texttt{evolution} = \texttt{id}$$
$$\text{DEF } \big(\texttt{yield a}\big).\texttt{observer} = \texttt{K a}$$

We can compose two monads

$$\text{FUN } (\_\ ;\ \_)[\alpha, \beta] : \texttt{M}[\alpha] \times \texttt{M}[\beta] \longrightarrow \texttt{M}[\beta]$$
$$\text{DEF } \big(\texttt{m}_1\ ;\ \texttt{m}_2\big).\texttt{evolution} = \texttt{m}_2.\texttt{evolution} \circ \texttt{m}_1.\texttt{evolution}$$
$$\text{DEF } \big(\texttt{m}_1\ ;\ \texttt{m}_2\big).\texttt{observer} = \texttt{m}_2.\texttt{observer} \circ \texttt{m}_1.\texttt{evolution}$$

We can lift a value to a monad (with no evolution)

$$\texttt{FUN}\ \texttt{yield}[\alpha] : \alpha \rightarrow \texttt{M}[\alpha]$$
$$\texttt{DEF}\ (\texttt{yield a}).\texttt{evolution} = \texttt{id}$$
$$\texttt{DEF}\ (\texttt{yield a}).\texttt{observer} = \texttt{K a}$$

We can compose two monads

$$\texttt{FUN}\ (\_ ; \_)[\alpha, \beta] : \texttt{M}[\alpha] \times \texttt{M}[\beta] \rightarrow \texttt{M}[\beta]$$
$$\texttt{DEF}\ (\texttt{m}_1 ; \texttt{m}_2).\texttt{evolution} = \texttt{m}_2.\texttt{evolution} \circ \texttt{m}_1.\texttt{evolution}$$
$$\texttt{DEF}\ (\texttt{m}_1 ; \texttt{m}_2).\texttt{observer} = \texttt{m}_2.\texttt{observer} \circ \texttt{m}_1.\texttt{evolution}$$

We can compose a monad with a continuation

$$\texttt{FUN}\ (\_ ; \_)[\alpha, \beta] : \texttt{M}[\alpha] \times (\alpha \rightarrow \texttt{M}[\beta]) \rightarrow \texttt{M}[\beta]$$
$$\texttt{DEF}\ \texttt{m}_1 ; \texttt{f} = (\texttt{f} \circ \texttt{m}_1.\texttt{observer})\ \texttt{S}\ (\texttt{m}_1.\texttt{evolution})$$

Note: $(\mathrm{h}\ \mathrm{S}\ \mathrm{g})(\mathrm{x}) = (\mathrm{h}\ \mathrm{x})(\mathrm{g}\ \mathrm{x})$

Automatic casting to monadic operations

Let $\mathtt{M}[\alpha]$ be defined by $\mathtt{Monad}(\mathtt{Graph})$:

$\boxed{\text{FUN } \mathtt{f} : \mathtt{Graph} \to \alpha}$ is lifted to

$$\boxed{\begin{array}{l} \text{FUN } \mathtt{f} : \mathtt{M}[\alpha] \\ \text{DEF } \mathtt{f}.\mathtt{observer} = \mathtt{f} \\ \text{DEF } \mathtt{f}.\mathtt{evolution} = \mathtt{id} \end{array}}$$

Automatic casting to monadic operations

Let $\mathtt{M}[\alpha]$ be defined by $\mathtt{Monad}(\mathtt{Graph})$:

$$\text{FUN } \mathtt{f} : \mathtt{Graph} \rightarrow \alpha$$

is lifted to

$$\begin{array}{l} \text{FUN } \mathtt{f} : \mathtt{M}[\alpha] \\ \text{DEF } \mathtt{f}.\mathtt{observer} = \mathtt{f} \\ \text{DEF } \mathtt{f}.\mathtt{evolution} = \mathtt{id} \end{array}$$

$$\text{FUN } \mathtt{f} : \mathtt{Graph} \rightarrow \mathtt{Graph}$$

is lifted to

$$\begin{array}{l} \text{FUN } \mathtt{f} : \mathtt{M}[\mathtt{Void}] \\ \text{DEF } \mathtt{f}.\mathtt{observer} = \mathtt{K}\ \mathtt{nil} \\ \text{DEF } \mathtt{f}.\mathtt{evolution} = \mathtt{f} \end{array}$$

The monadic graph `Monad(Graph)`:

Automatic lifting yields

FUN **roots** : M[Set Node]
FUN **sucs** : Node ⟶ M[Set Arc]
FUN **free** : M[Set Node]
FUN **add** : Node × Node ⟶ M[Void]
FUN **cut** : Node × Node ⟶ M[Void]
FUN **new** : M[Node]
FUN **recycle** : M[Void]
OBS **reachable** : Set Node ⟶ M[Set Node]
OBS **black** : M[Set Node]
OBS **gray** : M[Set Node]
OBS **white** : M[Set Node]

Special Predicates on Monads

$\mathrm{SPEC}\ \mathtt{Monad}\ (\mathrm{TYPE}\ \mathtt{State}) =$

$\quad \ldots$

-- *invariance*

$\mathrm{OBS}\ \mathtt{invariant} : \mathtt{M}[\alpha] \longrightarrow \mathtt{Bool}$

$\mathrm{DEF}\ \mathtt{invariant}(\mathtt{f}) \ = \ \mathtt{preserve}(=)(\mathtt{f})$

-- *monotonicity*

$\mathrm{OBS}\ \mathtt{monotone} : \mathtt{M}[\alpha] \longrightarrow \mathtt{Bool}$

$\mathrm{DEF}\ \mathtt{monotone}(\mathtt{f}) \ = \ \mathtt{preserve}(\preceq)(\mathtt{f})$

-- *preservation of a relation*

$\mathrm{OBS}\ \mathtt{preserve} : (\preceq : \alpha \times \alpha \longrightarrow \mathtt{Bool}) \longrightarrow \mathtt{f} : \mathtt{M}[\alpha] \longrightarrow \mathtt{Bool}$

$\quad \mathrm{PRE}\ \ \mathtt{f.evolution} = \mathtt{id}$

$\quad \mathrm{POST}\ \forall \mathtt{m} : \mathtt{M}[\alpha] : \ \mathtt{f.observer} \preceq \mathtt{f.observer} \circ \mathtt{m.evolution}$

Iterators on Monads

```
SPEC Monad (TYPE State) =
      ...
   -- infinite repetition
  FUN forever : M[Void] → M[Void]
  DEF forever m  =  m ; forever(m)

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

4 The Mutator

Any program that uses only the operations
`roots`, `sucs`, `add`, `cut`, `new`
is an acceptable instance of the mutator.

```
SPEC Mutator-Spec ( Env : Mutator-Environment ) =
  IMPORT Monad(Graph) ONLY roots, sucs, add, cut, new
  FUN mutate : M[Void]
  THM monotone white
```

The mutator guarantees that the (unreachable) white nodes remain white.

The mutator relies on the proper behaviour of its environment

```
SPEC Mutator-Environment =
  EXTEND Monad(Graph) BY
  AXM 1: invariant black
  AXM 2: ∀ n ∈ black: invariant sucs(n)
```

The environment must not change the black part of the graph, that is,

- not change the black nodes
- not change the arcs between black nodes

5 The Collector (naive view)

The collector
continuously recycles
white (unreachable) nodes.

```
SPEC Collector-Spec ( Env : Collector-Environment ) =
  IMPORT Monad(Graph) ONLY recycle, white
  FUN collect : M[Void] = forever(recycle)
  THM invariant black
  THM ∀n ∈ black : invariant sucs(n)
```

The collector guarantees that the
black part of the graph remains untouched

The collector relies on the proper behaviour of its environment

```
SPEC Collector-Environment =
  EXTEND Monad(Graph) BY
  AXM monotone white
```

The environment must not make white nodes reachable

Correctness:

Collector and mutator meet each others rely conditions

$$\begin{array}{lcl}\texttt{Collector-Spec} & \vdash & \texttt{Mutator-Environment} \\ \texttt{Mutator-Spec} & \vdash & \texttt{Collector-Environment}\end{array}$$

Correctness:

Collector and mutator meet each others rely conditions

$$\begin{array}{lcl} \texttt{Collector-Spec} & \vdash & \texttt{Mutator-Environment} \\ \texttt{Mutator-Spec} & \vdash & \texttt{Collector-Environment} \end{array}$$

Moreover:

Each unreachable node will eventually be in the freelist

$$\texttt{n} \in \texttt{white} \;\Rightarrow\; \Diamond\, \texttt{n} \in \texttt{gray}$$

Correctness:

Collector and mutator meet each others rely conditions

$$\begin{array}{lcl} \texttt{Collector-Spec} & \vdash & \texttt{Mutator-Environment} \\ \texttt{Mutator-Spec} & \vdash & \texttt{Collector-Environment} \end{array}$$

Moreover:

Each unreachable node will eventually be in the freelist

$$\texttt{n} \in \texttt{white} \Rightarrow \Diamond\, \texttt{n} \in \texttt{gray}$$

However:

The collector depends on the non-implementable function white.

6 Implementation

Idea:

Mark (a subset of) the unreachable nodes

GOAL $\texttt{marked} \subseteq \texttt{white}$

Algorithm:

- Start with all nodes marked
- Clean the reachable nodes
- Scavenge the marked nodes

We introduce the workset of currently considered nodes

$$\text{FUN } \mathtt{red} : \mathtt{Graph} \to \mathtt{Set\ Node} \quad \texttt{--}\ \textit{a coalgebraic observer}$$
$$\text{OBS } \mathtt{pink} : (\mathtt{G} : \mathtt{Graph} \to \mathtt{Set\ Node}) = \mathtt{G.reachable(G.red)}$$

- red is the wavefront of currently considered nodes
- pink are all nodes that still need to be visited.

We introduce the workset of currently considered nodes

$$\mathrm{FUN}\ \texttt{red} : \texttt{Graph} \to \texttt{Set Node} \quad \text{-- } \textit{a coalgebraic observer}$$
$$\mathrm{OBS}\ \texttt{pink} : (\texttt{G} : \texttt{Graph} \to \texttt{Set Node}) = \texttt{G.reachable(G.red)}$$

- ☞ red is the wavefront of currently considered nodes
- ☞ pink are all nodes that still need to be visited.

Central invariant:

$$\texttt{marked} \cap (\texttt{black} \cup \texttt{gray}) \subseteq \texttt{pink}$$

(The reachable nodes that are still marked will still be visited)

We introduce the workset of currently considered nodes

$$\mathrm{FUN}\ \texttt{red} : \texttt{Graph} \rightarrow \texttt{Set Node} \quad \texttt{--}\ \textit{a coalgebraic observer}$$
$$\mathrm{OBS}\ \texttt{pink} : (\texttt{G} : \texttt{Graph} \rightarrow \texttt{Set Node}) = \texttt{G.reachable}(\texttt{G.red})$$

- red is the wavefront of currently considered nodes
- pink are all nodes that still need to be visited.

Central invariant:

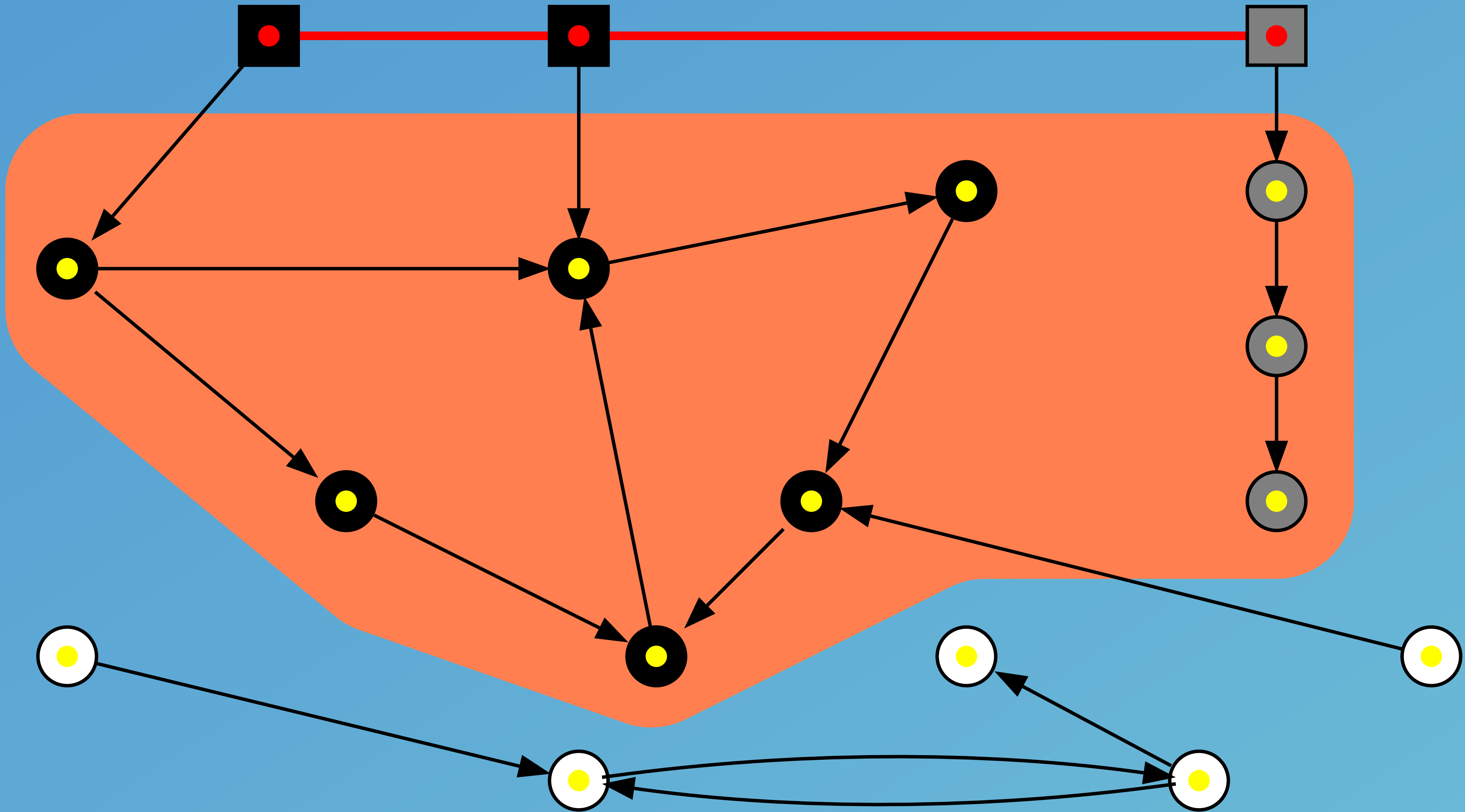
$$\texttt{marked} \cap (\texttt{black} \cup \texttt{gray}) \subseteq \texttt{pink}$$

(The reachable nodes that are still marked will still be visited)

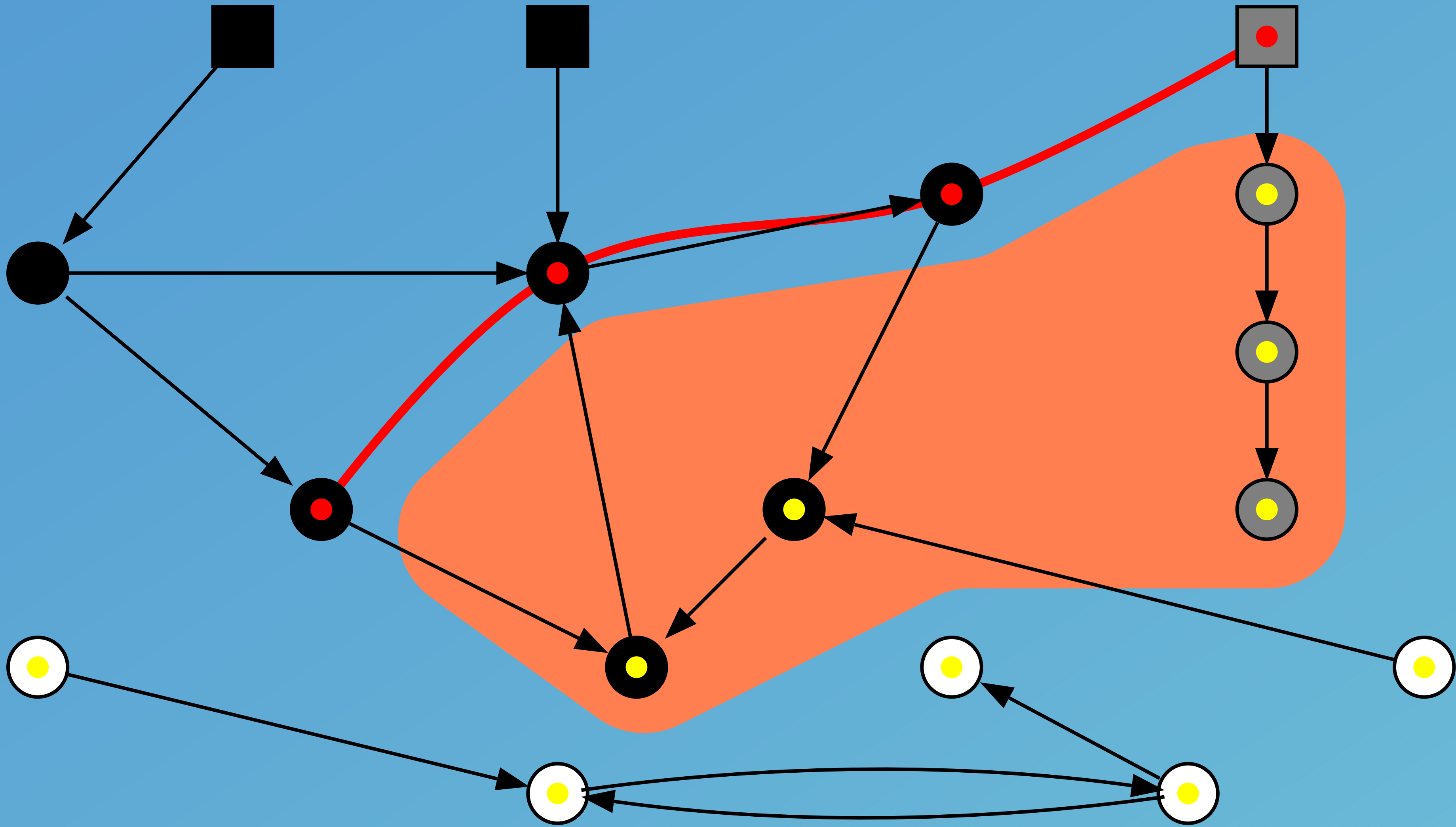
Conclusion

$$\texttt{red} = \emptyset \quad \vdash \quad \texttt{pink} = \emptyset \quad \vdash \quad \texttt{marked} \subseteq \texttt{white}$$

Initial situation



Intermediate snapshot



7 The Extended Collector

- ☞ The collector alternates between two phases:

7 The Extended Collector

☞ The collector alternates between two phases:

➢ cleaning
(removing the marks of the reachable nodes)

7 The Extended Collector

☞ The collector alternates between two phases:

➢ cleaning
(removing the marks of the reachable nodes)

➢ scavenging
(adding the remaining marked nodes to the freelist)

```
SPEC XCollector-Spec ( Env : XCollector-Environment ) =
  IMPORT Monad(XGraph)

  IMPORT Cleaner ONLY clean
  IMPORT Scavenger ONLY scavenge

  FUN collect : M[Void] = forever(clean ; scavenge)

  THM invariant black
  THM ∀n ∈ black : invariant sucs(n)
```

☞ The black subgraph has to remain untouched!

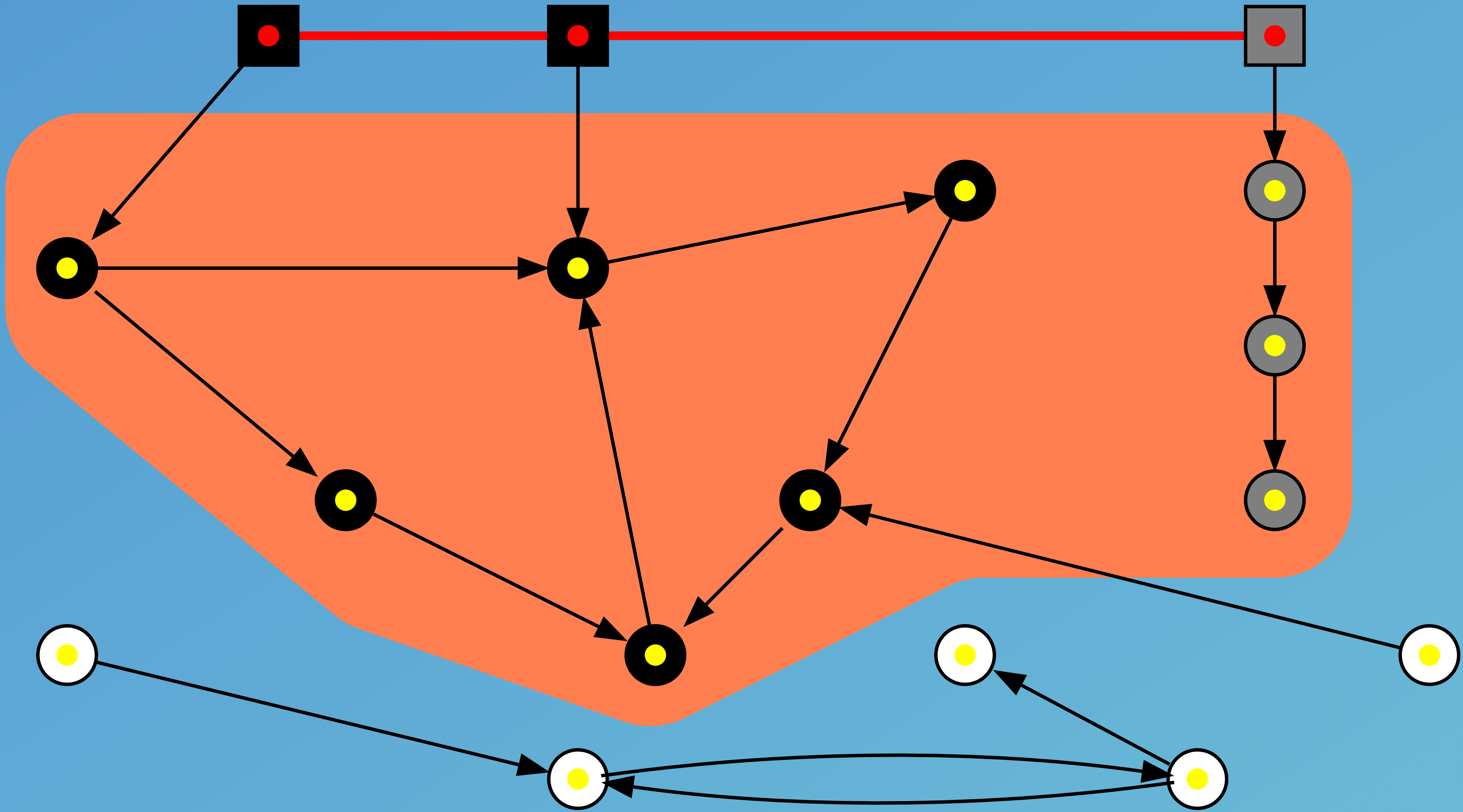
7.1 The Cleaning Phase ("unmark")

- Cleaning unmarks reachable nodes as long as possible
- It bases on an extended spec of `Graph`

$$
\begin{array}{l}
\mathrm{SPEC}\ \mathtt{Cleaner}\ =\\
\quad \mathrm{IMPORT}\ \mathtt{Monad}(\mathtt{Cleaning\text{-}View\text{-}of\text{-}XGraph})\\
\\
\quad \mathrm{FUN}\ \mathtt{clean} : \mathtt{M}[\mathtt{Void}]\\
\qquad\qquad \mathrm{PRE}\quad \mathtt{marked} = \{\mathtt{n} : \mathtt{Node}\}\\
\qquad\qquad \mathrm{POST}\ \mathtt{marked} \subseteq \mathtt{white}\\
\quad \mathrm{DEF}\ \mathtt{clean} = (\ \mathtt{red} \leftarrow \mathtt{roots} \cup \mathtt{free}\ ;\ \mathtt{iterate}(\mathtt{unmark})\ )\\
\\
\quad \mathrm{THM}\ \mathtt{invariant\ black}\\
\quad \mathrm{THM}\ \forall \mathtt{n} \in \mathtt{black} : \mathtt{invariant\ sucs}(\mathtt{n})
\end{array}
$$

- The black subgraph remains untouched!

Initial situation



Unmarking a red node makes all its marked successors red (push red frontier forward)

SPEC $\texttt{Cleaning-View-of-XGraph}$ $=$

EXTEND $\texttt{XGraph}$ ONLY $\texttt{red}, \texttt{sucs}$ BY

-- *unmark some node in the workset*

FUN $\texttt{unmark} : (\texttt{G} : \texttt{Graph}) \rightarrow (\texttt{G}' : \texttt{Graph})$

PRE $\texttt{G.red} \neq \emptyset$

POST LET $\texttt{n} \in \texttt{G.red}$ IN

$\texttt{G}'\texttt{.red} = \texttt{G.red} \cup (\texttt{G.sucs}(\texttt{n}) \cap \texttt{G.marked}) \ominus \texttt{n}$

$\texttt{G}'\texttt{.marked} = \texttt{G.marked} \ominus \texttt{n}$

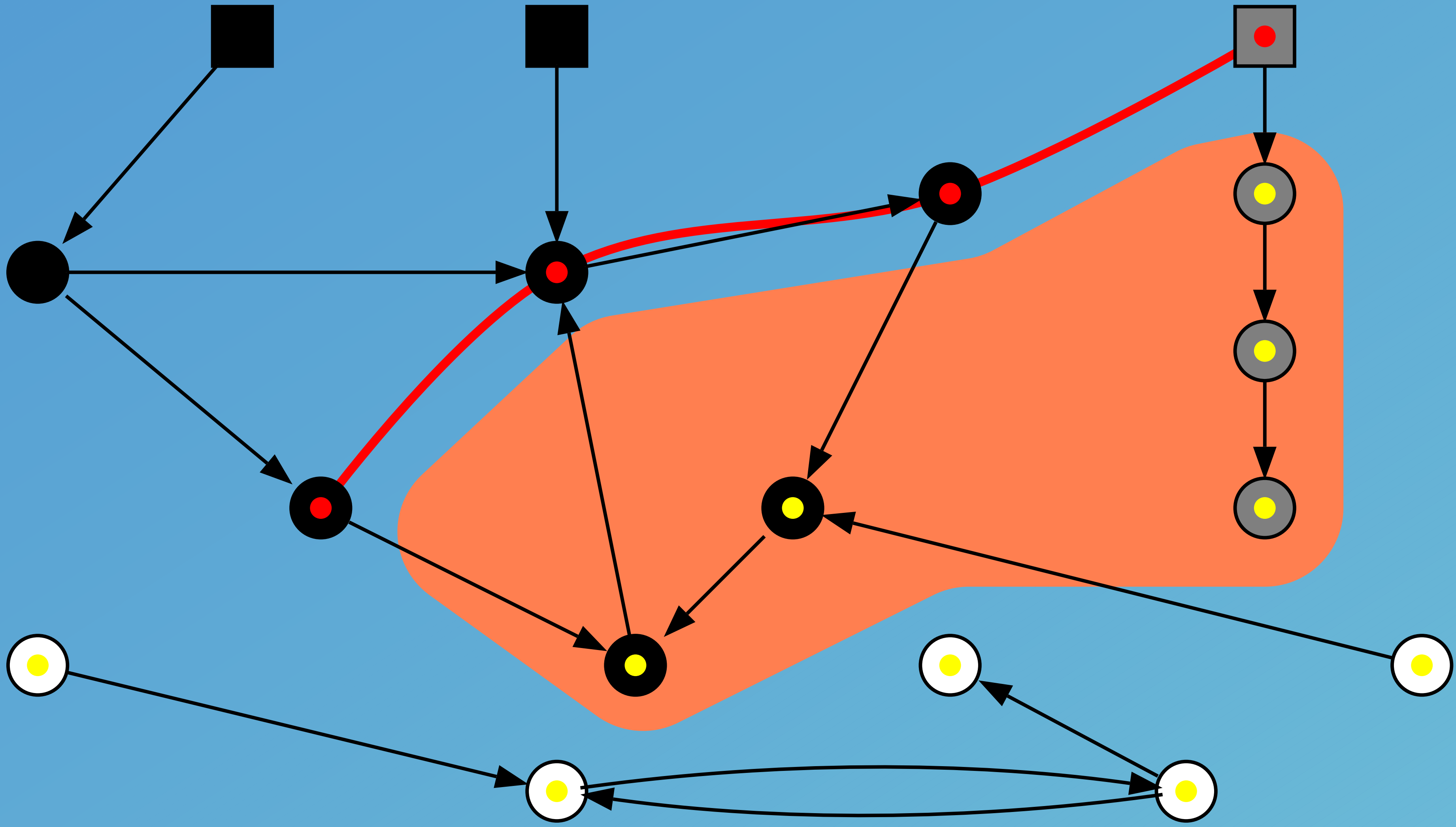
-- *the central invariant*

THM 1 : $\texttt{G.marked} \cap \texttt{G.dark} \subseteq \texttt{G.pink}$

-- *a helpful lemma*

THM 2 : $\texttt{G.red} = \emptyset \Rightarrow \texttt{G.marked} \subseteq \texttt{G.white}$

Intermediate snapshot



7.2 The Scavenging Phase ("Scan")

The scavenger recycles marked nodes as long as possible

```
SPEC Scavenger =
  IMPORT Monad(Scavenging-View-of-XGraph)

  FUN scavenge : M[Void]
      PRE marked ⊆ white
  DEF scavenge = iterate(recycle)

  THM invariant black
  THM ∀n ∈ black : invariant sucs(n)
```

☞ The black subgraph remains untouched!

The scavenger needs a primitive operation for recycling

SPEC $\mathtt{Scavanging\text{-}View\text{-}of\text{-}XGraph} =$

EXTEND $\mathtt{XGraph}$ ONLY $\mathtt{marked}$, $\mathtt{free}$ BY

-- *modify recycle*

FUN $\mathtt{recycle} : (\mathtt{G} : \mathtt{Graph}) \longrightarrow (\mathtt{G}' : \mathtt{Graph})$

PRE $\mathtt{G.marked} \neq \emptyset$

POST LET $\mathtt{n} \in \mathtt{G.marked}$ IN

$\mathtt{G}'.\mathtt{marked} = \mathtt{G.marked} \ominus \mathtt{n}$

$\mathtt{G}'.\mathtt{free} = \mathtt{G.free} \oplus \mathtt{n}$

-- *the central invariant*

AXM $1 : \mathtt{G.marked} \subseteq \mathtt{G.white}$

Adaptions for the Collector's Environment

☞ For retaining correctness the collector now relies on three properties of its environment

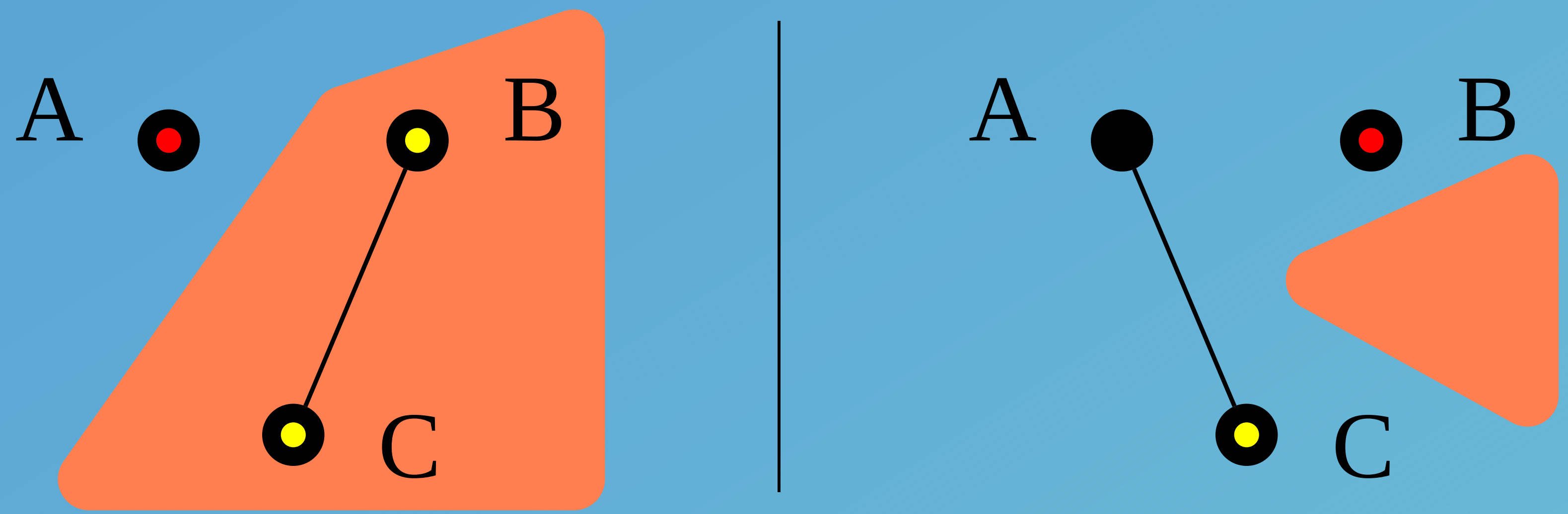
$$
\begin{array}{l}
\text{SPEC } \mathtt{XCollector\text{-}Environment} = \\
\quad \text{EXTEND } \mathtt{Monad}(\mathtt{XGraph}) \\
\quad \text{AXM } 1: \mathtt{G.marked} \cap \mathtt{G.dark} \subseteq \mathtt{G.pink} \\
\quad \text{AXM } 2: \mathtt{monotone\ white} \\
\quad \text{AXM } 3: \mathtt{invariant\ marked}
\end{array}
$$

(Note: $\mathtt{G.dark} = \mathtt{G.black} \cup \mathtt{G.gray}$)

8 The Extended Mutator

☞ The mutator now has to cooperate

Counterexample: A demonic mutator



```
SPEC XMutator-Spec ( Env : Mutator-Environment ) =
  IMPORT Monad(Mutator-View-of-XGraph)

  FUN mutate : M[Void]

  THM 1 : G.marked ∩ G.dark ⊆ G.pink
  THM 2 : monotone white
  THM 3 : invariant marked
```

The mutator guarantees:

1. all reachable marked nodes will still be visited
2. unreachable nodes remain unreachable
3. the marking remains untouched

The mutator needs a modified `add` operation

The mutator needs a modified add operation

- ☞ Whenever an arc to a (still) marked node is created, this node is coloured red (needs visiting)

The mutator needs a modified add operation

- ☞ Whenever an arc to a (still) marked node is created, this node is coloured red (needs visiting)

$$\text{SPEC } \mathtt{Mutator\text{-}View\text{-}of\text{-}XGraph} =$$

$$\quad \text{EXTEND } \mathtt{XGraph} \text{ ONLY } \mathtt{roots}, \mathtt{sucs}, \mathtt{cut}, \mathtt{new} \text{ BY}$$

-- *modify add*

$$\text{FUN } \mathtt{add} : (\mathtt{x} : \mathtt{Node}, \mathtt{y} : \mathtt{Node}) \to (\mathtt{G} : \mathtt{Graph}) \to (\mathtt{G}' : \mathtt{Graph})$$

$$\quad \text{POST } \mathtt{G}'.\mathtt{sucs}(\mathtt{x}) = \mathtt{G}.\mathtt{sucs}(\mathtt{x}) \oplus \mathtt{y}$$

$$\quad \mathtt{x} \notin \mathtt{G}.\mathtt{marked} \wedge \mathtt{y} \in \mathtt{G}.\mathtt{marked} \Rightarrow \mathtt{G}'.\mathtt{red} = \mathtt{G}.\mathtt{red} \oplus \mathtt{y}$$

9 Assessment of the Method

☞ Monads provide a sound and smooth way of integrating imperative concurrent programs with declarative specifications

9 Assessment of the Method

☞ Monads provide a sound and smooth way of integrating imperative concurrent programs with declarative specifications

☞ Selective import plays a central role in the formulation and verification of properties:
It restricts quantification to selected operations.

(this avoids the clumsy $\Box\,(\mathtt{at}\ \pi \Rightarrow \ldots)$ known from temporal logic.)

Parameterized specifications and restricted imports in the Mutator specification