Relaxing Safely: Verified On-the-Fly Garbage Collection for x86-TSO

Peter Gammie, Tony Hosking and Kai Engelhardt

May 27, 2015

Abstract

CIMP extends the small imperative language IMP with synchronous message passing. We use CIMP to model Schism, a state-of-the-art real-time garbage collection scheme for weak memory, and show that it is safe on x86-TSO.

Contents

1 Introduction 2

2 The Schism garbage collector 2
   2.1 Object marking ......................................................... 6
   2.2 Handshakes .............................................................. 8
   2.3 The system process .................................................... 12
   2.4 Mutators ................................................................. 15
   2.5 Garbage collector ...................................................... 17

3 Invariants and Proofs 19
   3.1 Constructors for sets of locations. ............................... 19
   3.2 Hoare triples ............................................................ 20
   3.3 Functions and predicates .............................................. 20
   3.4 Garbage collector locations. ........................................ 24
   3.5 Coarse TSO invariants ............................................... 24
      3.5.1 Locations where the TSO lock is held .................... 25
   3.6 Handshake phases ..................................................... 25
   3.7 Object colours, reference validity, worklist validity .......... 31
   3.8 Mark Object ............................................................ 32
   3.9 The strong-tricolour invariant .................................... 37
   3.10 Invariants .............................................................. 39
   3.11 Lonely mutator assertions ......................................... 40
   3.12 The infamous termination argument. ......................... 40
   3.13 Sweep loop invariants .............................................. 41
1 Introduction

We verify the memory safety of one of the Schism garbage collectors as developed by Pizlo (201x); Pizlo, Ziarek, Maj, Hosking, Blanton, and Vitek (2010) with respect to the x86-TO model (a total store order memory model for modern multicore Intel x86 architectures) developed and validated by Sewell, Sarkar, Owens, Nardelli, and Myreen (2010).

Our development is inspired by the original work on the verification of concurrent mark/sweep collectors by Dijkstra, Lamport, Martin, Scholten, and Steffens (1978), and the more realistic models and proofs of Doligez and Gonthier (1994). We leave a thorough survey of formal garbage collection verification to future work.

We present our model of the garbage collector in §2, the detailed invariants in §3, and the high-level safety results in §4. This bottom-up presentation is how we developed the proof; we have resisted the urge to hide the bodies with a rational reconstruction, partly because we expect the current structure to more readily support extensions and revisions.

This document does not include the formal proofs that the model satisfies these invariants; the curious reader is encouraged to peruse the Isabelle sources.

For details about the modelling language CIMP used in this development, see the separate AFP entry ConcurrentIMP (Gammie 2015).

2 The Schism garbage collector

The following formalises Figures 2.8 (mark-object-fn), 2.9 (load and store but not alloc), and 2.15 (garbage collector) of Pizlo (201x). See also Pizlo et al. (2010).

We additionally need to model TSO memory, the handshakes and compare-and-swap (CAS).

We closely model things where interference is possible and abstract everything else.

NOTE: this model is for TSO only. We elide any details irrelevant for that memory model.

We begin by defining the types of the various parts. Our program locations are labelled with strings for readability. We enumerate the names of the processes in our system. The safety proof treats an arbitrary (unbounded) number of mutators.

\textbf{type-synonym} \textit{location} = \textit{char list}

\textbf{datatype} 'mut process-name = mutator 'mut | gc | sys

The garbage collection process can be in one of the following phases.

\textbf{datatype} gc-phase
The garbage collector instructs mutators to perform certain actions, and blocks until the mutators signal these actions are done. The mutators always respond with their work list (a set of references). The handshake can be of one of the specified types.

**datatype** handshake-type

\[=\] ht-NOOP
| ht-GetRoots
| ht-GetWork

We track how many noop and get_roots handshakes each process has participated in as ghost state. See §2.2.

**datatype** handshake-phase

\[=\] hp-Idle
| hp-IdleInit
| hp-InitMark
| hp-Mark
| hp-IdleMarkSweep

**definition** handshake-step :: handshake-phase ⇒ handshake-phase where

handshake-step ph ≡ case ph of
  hp-Idle       ⇒ hp-IdleInit
| hp-IdleInit   ⇒ hp-InitMark
| hp-InitMark   ⇒ hp-Mark
| hp-Mark       ⇒ hp-IdleMarkSweep
| hp-IdleMarkSweep ⇒ hp-Idle

An object consists of a garbage collection mark and a function that maps its fields to values. A value is either a reference or **NULL**.

'field' is the abstract type of fields. 'ref' is the abstract type of object references. 'mut' is the abstract type of the mutators' names.

For simplicity we assume all objects define all fields and ignore all non-reference payload in objects.

**type-synonym** gc-mark = bool

**record** ('field', 'ref') object =
  obj-mark :: gc-mark
  obj-fields :: 'field ⇒ 'ref option

The TSO store buffers track write actions, represented by ('field', 'ref) mem-write-action.

**datatype** ('field', 'ref) mem-write-action

\[=\] mw-Mark 'ref gc-mark
| mw-Mutate 'ref 'field 'ref option
The following record is the type of all processes’s local states. For the mutators and the garbage collector, consider these to be local variables or registers.

The system’s \( f_A, f_M, \) phase and heap variables are subject to the TSO memory model, as are all heap operations.

\[
\text{record } (\text{\textquote{\text{'field}}, \text{\textquote{\text{'mut}}, \text{\textquote{\text{'ref}}}) local-state = }
\]

— System-specific fields

heap :: \( \text{\textquote{\text{'ref}} \Rightarrow (\text{\textquote{\text{'field}}, \text{\textquote{\text{'ref}}}) object option} \)

— TSO memory state

\[\text{mem-write-buffers :: \text{\textquote{\text{'mut process-name}} \Rightarrow (\text{\textquote{\text{'field}}, \text{\textquote{\text{'ref}}}) mem-write-action list}}\]

\[\text{mem-lock :: \text{\textquote{\text{'mut process-name}} option}}\]

— The state of the handshakes

\[\text{handshake-type :: \text{handshake-type}}\]

\[\text{handshake-pending :: \text{\textquote{\text{'mut}}} \Rightarrow \text{bool}}\]

— Ghost state

\[\text{ghost-handshake-in-sync :: \text{\textquote{\text{'mut}}} \Rightarrow \text{bool}}\]

\[\text{ghost-handshake-phase :: \text{handshake-phase}}\]

— Mutator-specific temporaries

\[\text{new-ref :: \text{\textquote{\text{'ref}} option}}\]

\[\text{roots :: \text{\textquote{\text{'ref}} set}}\]

\[\text{ghost-honorary-root :: \text{\textquote{\text{'ref}} set}}\]

— Garbage collector-specific temporaries

\[\text{field-set :: \text{\textquote{\text{'field set}}} \]

\[\text{mut :: \text{\textquote{\text{'mut}} \}

\[\text{muts :: \text{\textquote{\text{'mut}} set}}\]

— Local variables used by multiple processes

\[\text{fA :: gc-mark} \]

\[\text{fM :: gc-mark} \]

\[\text{cas-mark :: gc-mark option} \]

\[\text{field :: \text{\textquote{\text{'field}}} \]

\[\text{mark :: gc-mark option} \]

\[\text{phase :: gc-phase} \]

\[\text{tmp-ref :: \text{\textquote{\text{'ref}} \]

\[\text{ref :: \text{\textquote{\text{'ref}} option}}\]

\[\text{refs :: \text{\textquote{\text{'ref}} set}}\]

\[\text{W :: \text{\textquote{\text{'ref}} set}}\]

— Ghost state

\[\text{ghost-honorary-grey :: \text{\textquote{\text{'ref}} set}}\]

An action is a request by a mutator or the garbage collector to the system.

\[
\text{datatype } (\text{\textquote{\text{'field}}, \text{\textquote{\text{'ref}}}) \text{ mem-read-action}
\]
= $\text{mr-Ref}$ 'ref 'field
| $\text{mr-Mark}$ 'ref
| $\text{mr-Phase}$
| $\text{mr-fM}$
| $\text{mr-fA}$

datatype ('field, 'mut, 'ref) request-op
= ro-MFENCE
| ro-Read ('field, 'ref) mem-read-action
| ro-Write ('field, 'ref) mem-write-action
| ro-Lock
| ro-Unlock
| ro-Alloc
| ro-Free 'ref
| ro-hs-gc-set-type handshake-type
| ro-hs-gc-set-pending 'mut
| ro-hs-gc-read-pending 'mut
| ro-hs-gc-load-W
| ro-hs-mut-read-type handshake-type
| ro-hs-mut-done 'ref set

abbreviation ReadfM ≡ ro-Read mr-fM
abbreviation ReadMark r ≡ ro-Read (mr-Mark r)
abbreviation ReadPhase ≡ ro-Read mr-Phase
abbreviation ReadRef r f ≡ ro-Read (mr-Ref r f)

abbreviation WritefA m ≡ ro-Write (mw-fA m)
abbreviation WritefM m ≡ ro-Write (mw-fM m)
abbreviation WriteMark r m ≡ ro-Write (mw-Mark r m)
abbreviation WritePhase ph ≡ ro-Write (mw-Phase ph)
abbreviation WriteRef r f r' ≡ ro-Write (mw-Mutate r f r')

type-synonym ('field, 'mut, 'ref) request
= 'mut process-name × ('field, 'mut, 'ref) request-op

datatype ('field, 'ref) response
= mv-Bool bool option
| mv-Mark gc-mark option
| mv-Phase gc-phase
| mv-Ref 'ref option
| mv-Refs 'ref set
| mv-Void

We instantiate CIMP’s types as follows:

type-synonym ('field, 'mut, 'ref) gc-com
= ('field, 'ref) response, location, ('field, 'mut, 'ref) request, ('field, 'mut, 'ref) local-state)
We use one locale per process to define a namespace for definitions local to these processes. Mutator definitions are parametrised by the mutator’s identifier \( m \). We never interpret these locales; we use their contents typically by prefixing their names the locale name. This might be considered an abuse. The attributes depend on locale scoping somewhat, which is a mixed blessing.

If we have more than one mutator then we need to show that mutators do not mutually interfere. To that end we define an extra locale that contains these proofs.

2.1 Object marking

Both the mutators and the garbage collector mark references, which indicates that a reference is live in the current round of collection. This operation is defined in Pizlo (201x, Figure 2.8). These definitions are parameterised by the name of the process.
begin

abbreviation lock :: location ⇒ ('field, 'mut, 'ref) gc-com where
  lock l ≡ \{l\} Request (λs. (p, ro-Lock)) (λ- s. \{s\})
notation lock (\{\} lock)

abbreviation unlock :: location ⇒ ('field, 'mut, 'ref) gc-com where
  unlock l ≡ \{l\} Request (λs. (p, ro-Unlock)) (λ- s. \{s\})
notation unlock (\{\} unlock)

abbreviation
  read-mark :: location ⇒ (('field, 'mut, 'ref) local-state ⇒ 'ref)
    ⇒ ((gc-mark option ⇒ gc-mark option)
    ⇒ (field, 'mut, 'ref) local-state
⇒ (field, 'mut, 'ref) local-state) ⇒ (field, 'mut, 'ref) gc-com
where
  read-mark l r upd ≡ \{l\} Request (λs. (p, ReadMark (r s))) (λmv s. \{ upd (m) s | m. mv = mv-Mark m \})
notation read-mark (\{\} read'-mark)

abbreviation read-fM :: location ⇒ ('field, 'mut, 'ref) gc-com where
  read-fM l ≡ \{l\} Request (λs. (p, ro-Read mr-fM)) (λmv s. \{ s(fM := m) | m. mv = mv-Mark (Some m) \})
notation read-fM (\{\} read'-fM)

abbreviation
  read-phase :: location ⇒ (field, 'mut, 'ref) gc-com
where
  read-phase l ≡ \{l\} Request (λs. (p, ReadPhase)) (λmv s. \{ s(phase := ph) | ph. mv = mv-Phase ph \})
notation read-phase (\{\} read'-phase)

abbreviation write-mark :: location ⇒ (('field, 'mut, 'ref) local-state ⇒ 'ref) ⇒ (('field, 'mut, 'ref) local-state ⇒ bool) ⇒ (field, 'mut, 'ref) gc-com where
  write-mark l r fl ≡ \{l\} Request (λs. (p, WriteMark (r s) (fl s))) (λ- s. \{ s(ghost-honorary-grey := \{r s\} |} \})
notation write-mark (\{\} write'-mark)

abbreviation add-to-W :: location ⇒ (('field, 'mut, 'ref) local-state ⇒ 'ref) ⇒ (field, 'mut, 'ref) gc-com where
  add-to-W l r ≡ \{l\} |λs. s( W := W s \cup \{r s\}, ghost-honorary-grey := \{\} |}
notation add-to-W (\{\} add'-to'-W)

The reference we’re marking is given in ref. If the current process wins the CAS race then the reference is marked and added to the local work list W.

TSO means we cannot avoid having the mark write pending in a store buffer; in other words, we cannot have objects atomically transition from white to grey. The following scheme black-
ens a white object, and then reverts it to grey. The ghost-honorary-grey variable is used to track objects undergoing this transition.

As CIMP provides no support for function calls, we prefix each statement’s label with a string from its callsite.

definition mark-object-fn :: location ⇒ ('field, 'mut, 'ref) gc-com where
  mark-object-fn l ≡
  \{ l @ "-mo-null" | IF not null ref THEN
  \{ l @ "-mo-mark" | read-mark (the ◦ ref) mark-update ;;
  \{ l @ "-mo-fM" | read-fM ;;
  \{ l @ "-mo-mtest" | IF mark neq Some ◦ fM THEN
  \{ l @ "-mo-phase" | read-phase ;;
  \{ l @ "-mo-pptest" | IF phase neq ⟨ph-Idle⟩ THEN
  (∗ CAS: claim object ∗)
  \{ l @ "-mo-co-lock" | lock ;;
  \{ l @ "-mo-co-cmark" | read-mark (the ◦ ref) cas-mark-update ;;
  \{ l @ "-mo-co-ctest" | IF cas-mark eq mark THEN
  \{ l @ "-mo-co-mark" | write-mark (the ◦ ref) fM
  FI ;;
  \{ l @ "-mo-co-unlock" | unlock ;;
  \{ l @ "-mo-co-won" | IF cas-mark eq mark THEN
  \{ l @ "-mo-co-W" | add-to-W (the ◦ ref)
  FI
  FI
  FI
  FIn
end

The worklists (field W) are not subject to TSO. As we later show (§3.7), these are disjoint and hence operations on these are private to each process, with the sole exception of when the GC requests them from the mutators. We describe that mechanism next.

2.2 Handshakes

The garbage collector needs to synchronise with the mutators. In practice this is implemented with some thread synchronisation primitives that include memory fences. The scheme we adopt here has the GC busy waiting. It sets a pending flag for each mutator and then waits for each to respond.

The system side of the interface collects the responses from the mutators into a single worklist, which acts as a proxy for the garbage collector’s local worklist during get-roots and get-work handshakes. In practise this involves a CAS operation. We carefully model the effect these handshakes have on the process’s TSO buffers.

The system and mutators track handshake phases using ghost state.

abbreviation hp-step :: handshake-type ⇒ handshake-phase ⇒ handshake-phase where
  hp-step ht ≡
  case ht of
ht-NOOP ⇒ handshake-step
| ht-GetRoots ⇒ handshake-step
| ht-GetWork ⇒ id

context sys
begin

definition handshake :: ('field, 'mut, 'ref) gc-com where
  handshake ≡
  "sys-hs-gc-set-type" Response
  (λreq s. { (s[ handshaketype := ht,
    ghost-handshake-in-sync := (False),
    ghost-handshake-phase := hp-step ht (ghost-handshake-phase s) ]},
  mv-Void)
  | ht. req = (gc, ro-hs-gc-set-type ht ) })
\ □ "sys-hs-gc-mut-reqs" Response
  (λreq s. { (s[ handshake-pending := (handshake-pending s)(m := True) ], mv-Void)
  | m. req = (gc, ro-hs-gc-set-pending m ) })
\ □ "sys-hs-gc-done" Response
  (λreq s. { (s, mv-Bool (¬handshake-pending s m))
  | m. req = (gc, ro-hs-gc-read-pending m ) })
\ □ "sys-hs-gc-load-W" Response
  (λreq s. { (s[ W := {} ], mv-Refs (W s))
  | ::unit. req = (gc, ro-hs-gc-load-W ) })
\ □ "sys-hs-mut" Response
  (λreq s. { (s, mv-Void)
  | m. req = (mutator m, ro-hs-mut-read-type (handshake-type s))
  ∧ handshake-pending s m })
\ □ "sys-hs-mut-done" Response
  (λreq s. { (s[ handshake-pending := (handshake-pending s)(m := False),
    W := W s ∪ W',
    ghost-handshake-in-sync := (ghost-handshake-in-sync s)(m := True) ],
  mv-Void)
  | m W’. req = (mutator m, ro-hs-mut-done W’ ) })
end

The mutator’s side of the interface. Also updates the ghost state tracking the handshake state for ht-NOOP and ht-GetRoots but not ht-GetWork.

context mut-m
begin

abbreviation mark-object :: location ⇒ ('field, 'mut, 'ref) gc-com ('- : mark'-object) where
  "l" mark-object ≡ mark-object-fn (mutator m) l

abbreviation mfence :: location ⇒ ('field, 'mut, 'ref) gc-com ('- : MFENCE) where
  "l" MFENCE ≡ "l" Request (λs. (mutator m, ro-MFENCE)) (λ- s. {s})
abbreviation hs-read-type :: location ⇒ handshake-type ⇒ (′field, ′mut, ′ref) gc-com (\hs′-read′-type) where
\hs-read-type ht ≡ \hs \ Request (λs. (mutator m, ro-hs-mut-read-type ht)) (λ- s. \{s\})

abbreviation hs-noop-done :: location ⇒ (′field, ′mut, ′ref) gc-com (\hs′-noop′-done) where
\hs-noop-done ≡ \hs \ Request (λs. (mutator m, ro-hs-mut-done \{\})) (λ- s. \{s\} ghost-handshake-phase := handshake-step)

abbreviation hs-get-roots-done :: location ⇒ (′field, ′mut, ′ref) gc-com (\hs′-get′-roots′-done) where
\hs-get-roots-done ≡ \hs \ Request (λs. (mutator m, ro-hs-mut-done (\{\})) (λ- s. \{s\} W := \{\}, ghost-handshake-phase := handshake-step)

abbreviation hs-get-work-done :: location ⇒ (′field, ′mut, ′ref) gc-com (\hs′-get′-work′-done) where
\hs-get-work-done ≡ \hs \ Request (λs. (mutator m, ro-hs-mut-done (\{\})) (λ- s. \{s\} W := \{\}))

definition handshake :: (′field, ′mut, ′ref) gc-com where
handshake ≡
\hs-noop\hs-noop-mfence\hs-noop-done
\hs-get-roots\hs-get-roots-mfence\hs-get-roots-refs \refs := \refs
\hs-get-roots-loop \WHILE \not \empty \refs \DO
\hs-get-roots-loop-choose-ref \ref := \Some \refs
\hs-get-roots-loop \mark-object
\refs := (\refs \setminus \{\ref\}) \OD
\hs-get-roots-done \hs-get-work\hs-get-work-mfence\hs-get-work-done
\hs-get-roots-loop
\hs-get-work-done

end

The garbage collector’s side of the interface.
abbreviation set-hs-type :: location ⇒ handshake-type ⇒ (field, mut, ref) gc-com (\l\) set'-hs'-type) where
\{l\} set-hs-type ht \equiv \{l\} Request (\lambda.s. (gc, ro-hs-gc-set-type ht)) (λ- s. \{s\})

abbreviation set-hs-pending :: location ⇒ ((field, mut, ref) local-state ⇒ mut) ⇒ (field, mut, ref) gc-com (\l\) set'-hs'-pending) where
\{l\} set-hs-pending m \equiv \{l\} Request (\lambda.s. (gc, ro-hs-gc-set-pending (m s))) (λ- s. \{s\})

definition handshake-init :: location ⇒ handshake-type ⇒ (field, mut, ref) gc-com (\l\) handshake'-init)
where
\{l\} handshake-init req \equiv
\{l\} \text{"-init-type"} \} set-hs-type req ;;
\{l\} \text{"-init-muts"} \} \text{mut} \equiv \text{UNIV} ;;
\{l\} \text{"-init-loop"} \} WHILE not empty muts DO
\{l\} \text{"-init-loop-choose-mut"} \} \text{mut} \equiv \text{mut} ;;
\{l\} \text{"-init-loop-set-pending"} \} set-hs-pending mut ;;
\{l\} \text{"-init-loop-done"} \} \text{mut} \equiv (\text{mut} - \{\text{mut}\})
\text{OD}

definition handshake-done :: location ⇒ (field, mut, ref) gc-com (\l\) handshake'-done)
where
\{l\} handshake-done \equiv
\{l\} \text{"-done-muts"} \} \text{mut} \equiv \text{UNIV} ;;
\{l\} \text{"-done-loop"} \} WHILE not empty muts DO
\{l\} \text{"-done-loop-choose-mut"} \} \text{mut} \equiv \text{mut} ;;
\{l\} \text{"-done-loop-rendezvous"} \} Request
(\lambda.s. (gc, ro-hs-gc-read-pending (mut s)))
(λ\text{mv} s. \{ s\{ \text{mut} \equiv \text{mut} s - \{ \text{mut} \} \text{done. \text{mv} = \text{mv-Bool} \text{ done } \wedge \text{done } } \})\}
\text{OD}

abbreviation load-W :: location ⇒ (field, mut, ref) gc-com (\l\) load'-W) where
\{l\} load-W \equiv \{l\} \text{"-load-W"} Request (\lambda.s. (gc, ro-hs-gc-load-W))
(λ\text{resp} s. \{ s\{ W \equiv W' \} | W'. \text{resp} = \text{mv-Refs} W' \})

abbreviation mfence :: location ⇒ (field, mut, ref) gc-com (\l\) MFENCE) where
\{l\} MFENCE \equiv \{l\} Request (\lambda.s. (gc, ro-MFENCE)) (λ- s. \{s\})

definition handshake-noop :: location ⇒ (field, mut, ref) gc-com (\l\) handshake'-noop)
where
\{l\} handshake-noop \equiv
\{l\} \text{"-noop"} \} MFENCE ;;
\{l\} handshake-init ht-NOOP ;;
2.3 The system process

The system process models the environment in which the garbage collector and mutators execute. We translate the x86-TSO memory model due to Sewell et al. (2010) into a CIMP process. It is a reactive system: it receives requests and returns values, but initiates no communication itself. It can, however, autonomously commit a write pending in a TSO store buffer.

The memory bus can be locked by atomic compare-and-swap (CAS) instructions (and others in general). A processor is not blocked (i.e., it can read from memory) when it holds the lock, or no-one does.

definition not-blocked :: ('field, 'mut, 'ref) local-state ⇒ 'mut process-name ⇒ bool
where
not-blocked s p ≡ case mem-lock s of None ⇒ True | Some p' ⇒ p = p'

We compute the view a processor has of memory by applying all its pending writes.

definition do-write-action :: ('field, 'ref) mem-write-action ⇒ ('field, 'mut, 'ref) local-state ⇒ ('field, 'mut, 'ref) local-state where
do-write-action wact ≡ λs.
case wact of
  mw-Mark r gc-mark ⇒ s(heap := (heap s)(r := map-option (λobj. obj(obj-mark := gc-mark))(heap s r)))
| mw-Mutate f new-r ⇒ s(heap := (heap s)(r := map-option (λobj. obj(obj-fields := (obj-fields obj)(f := new-r))).(heap s r))))
| mw-fM gc-mark ⇒ s(fM := gc-mark)
| mw-fA gc-mark ⇒ s(fA := gc-mark)
\[
\begin{align*}
| \text{mw-Phase gc-phase} & \Rightarrow s(\text{phase} := \text{gc-phase}) \\
\end{align*}
\]

**Definition**

\text{fold-writes} :: \langle \text{field}, \text{ref} \rangle \text{mem-write-action list} \Rightarrow \langle \text{field}, \text{mut}, \text{ref} \rangle \text{local-state} \Rightarrow \langle \text{field}, \text{mut}, \text{ref} \rangle \text{local-state}

**Where**

\text{fold-writes} ws \equiv \text{fold} (\lambda w. \text{op} \circ (\text{do-write-action} w)) ws id

**Abbreviation**

\text{processors-view-of-memory} :: \text{'mut process-name} ⇒ \langle \text{field}, \text{mut}, \text{ref} \rangle \text{local-state} \Rightarrow \langle \text{field}, \text{mut}, \text{ref} \rangle \text{local-state}

**Where**

\text{processors-view-of-memory} p s \equiv \text{fold-writes} (\text{mem-write-buffers} s p) s

**Definition**

\text{do-read-action} :: \langle \text{field}, \text{ref} \rangle \text{mem-read-action} \Rightarrow \langle \text{field}, \text{mut}, \text{ref} \rangle \text{local-state} \Rightarrow \langle \text{field}, \text{ref} \rangle \text{response}

**Where**

\text{do-read-action} ract \equiv \lambda s. \begin{cases} \text{mr-Ref} r f & \Rightarrow \text{mv-Ref} (\text{heap} s r \gg= (\lambda \text{obj. obj-fields obj f})) \\ \text{mr-Mark} r & \Rightarrow \text{mv-Mark} (\text{map-option} \ \text{obj-mark} (\text{heap} s r)) \\ \text{mr-Phase} & \Rightarrow \text{mv-Phase} (\text{phase} s) \\ \text{mr-fM} & \Rightarrow \text{mv-Mark} (\text{Some} (\text{fM} s)) \\ \text{mr-fA} & \Rightarrow \text{mv-Mark} (\text{Some} (\text{fA} s)) \end{cases}

**Definition**

\text{sys-read} :: \text{'mut process-name} \Rightarrow \langle \text{field}, \text{ref} \rangle \text{mem-read-action} \Rightarrow \langle \text{field}, \text{mut}, \text{ref} \rangle \text{local-state} \Rightarrow \langle \text{field}, \text{ref} \rangle \text{response}

**Where**

\text{sys-read} p ract \equiv \text{do-read-action} ract \circ \text{processors-view-of-memory} p

**Context** sys

**Begin**

The semantics of TSO memory following Sewell et al. (2010, §3). This differs from the earlier Owens, Sarkar, and Sewell (2009) by allowing the TSO lock to be taken by a process with a non-empty write buffer. We omit their treatment of registers; these are handled by the local states of the other processes. The system can autonomously take the oldest write in the write buffer for processor \( p \) and commit it to memory, provided \( p \) either holds the lock or no processor does.

**Definition**

\text{mem-TSO} :: \langle \text{field}, \text{mut}, \text{ref} \rangle \text{gc-com}

**Where**
\text{mem-\text{TSO} \equiv}
\begin{align*}
\{"\text{sys-read}"\} & \text{Response } \lambda \text{req } s\{ (s, \text{sys-read } p \, mr \, s) \\
& \quad \{ p \, mr. \, req = (p, \text{ro-Read } mr) \land \text{not-blocked } s \, p \} \\
\cup \{"\text{sys-write}"\} & \text{Response } \lambda \text{req } s\{ (s) \text{-mem-write-buffers} := (\text{mem-write-buffers } s)(p := \text{mem-write-buffers } s \, p \, @ [w]) \}, \text{mv-Void} \\
& \quad \{ p \, w. \, req = (p, \text{ro-Write } w) \} \\
\cup \{"\text{sys-mfence}"\} & \text{Response } \lambda \text{req } s\{ (s) \land \text{mem-write-buffers } s \, p = [] \}, \text{mv-\text{Void}} \\
& \quad \{ p. \, req = (p, \text{ro-MFENCE}) \land \text{mem-write-buffers } s \, p = [] \} \\
\cup \{"\text{sys-lock}"\} & \text{Response } \lambda \text{req } s\{ (s) \text{-mem-lock} := \text{Some } p \}, \text{mv-\text{Void}} \\
& \quad \{ p. \, req = (p, \text{ro-Lock}) \land \text{mem-lock } s = \text{None} \} \\
\cup \{"\text{sys-unlock}"\} & \text{Response } \lambda \text{req } s\{ (s) \text{-mem-lock} := \text{None} \}, \text{mv-\text{Void}} \\
& \quad \{ p. \, req = (p, \text{ro-Unlock}) \land \text{mem-lock } s = \text{Some } p \land \text{mem-write-buffers } s \, p = [] \} \\
\cup \{"\text{sys-dequeue-write-buffer}"\} & \text{LocalOp } \lambda s\{ (\text{do-write-action } w \, s)(\text{mem-write-buffers} := (\text{mem-write-buffers } s)(p := ws)) \} \\
& \quad \{ p \, w \, ws. \, \text{mem-write-buffers } s \, p = w \# ws \land \text{not-blocked } s \, p \land p \neq \text{sys} \}
\end{align*}

We track which references are allocated using the domain of \textit{heap}.

For now we assume that the system process magically allocates and deallocates references. To model this more closely we would need to take care of the underlying machine addresses. We should be able to separate out those issues from GC correctness: the latter should imply that only alloc and free can interfere with each other.

We also arrange for the object to be marked atomically (see §2.4) which morally should be done by the mutator. In practice allocation pools enable this kind of atomicity (wrt the sweep loop in the GC described in §2.5).

Note that the \texttt{abort} in Pizlo (201x, Figure 2.9: Alloc) means the atomic fails and the mutator can revert to activity outside of \texttt{Alloc}, avoiding deadlock.

\textbf{definition}
\begin{align*}
\text{alloc} & \equiv (\text{field}, \text{mut}, \text{ref}) \text{ gc-com} \\
\text{where} & \\
\text{alloc} & \equiv \{"\text{sys-alloc}"\} \text{ Response } \lambda \text{req } s\{ (s) \text{-heap} := (\text{heap } s)(r := \text{Some } fA \text{ s, obj-fields} = \langle \text{None} \rangle) \}, \text{mv-\text{Ref}} \\
& \quad \{ r. \ l \notin \text{dom } \text{(heap } s) \land \text{snd req} = \text{ro-Alloc} \}
\end{align*}

References are freed by removing them from \textit{heap}.

\textbf{definition}
\begin{align*}
\text{free} & \equiv (\text{field}, \text{mut}, \text{ref}) \text{ gc-com} \\
\text{where} & \\
\text{free} & \equiv \{"\text{sys-free}"\} \text{ Response } \lambda \text{req } s\{ (s) \text{-heap} := (\text{heap } s)(r := \text{None})], \text{mv-\text{Void}} \ | r. \ \text{snd req} = \text{ro-Free } r \}
\end{align*}

The top-level system process.

\textbf{definition}
\begin{align*}
\text{com} & \equiv (\text{field}, \text{mut}, \text{ref}) \text{ gc-com}
\end{align*}
where
\[
\text{com} \equiv \begin{align*}
&\text{LOOP DO} \\
&\quad \text{mem-} \text{TSO} \\
&\quad \Box \text{ alloc} \\
&\quad \Box \text{ free} \\
&\quad \Box \text{ handshake} \\
&\text{OD}
\end{align*}
\]
end

2.4 Mutators

The mutators need to cooperate with the garbage collector. In particular, when the garbage collector is not idle the mutators use a write barrier (see §2.1).

The local state for each mutator tracks a working set of references, which abstracts from how the process’s registers and stack are traversed to discover roots.

context mut-m
begin

Allocation is defined in Pizlo (201x, Figure 2.9). See §2.3 for how we abstract it.

abbreviation (in −) mut-alloc :: 'mut ⇒ ('field, 'mut, 'ref) gc-com where
\[
\text{mut-alloc m} \equiv \begin{align*}
&\{\text{"alloc"}\} \text{ Request } (\lambda s. (\text{mutator m}, \text{ro-Alloc})) \\
&\quad (\lambda mv s. \{ s[| \text{ roots} := \text{roots s} \cup \{r\} | | r. \text{mv} = \text{mv-Ref}(\text{Some r}) \})
\end{align*}
\]

abbreviation alloc :: ('field, 'mut, 'ref) gc-com where
\[
\text{alloc} \equiv \text{mut-alloc m}
\]

The mutator can always discard any references it holds.

abbreviation discard :: ('field, 'mut, 'ref) gc-com where
\[
\text{discard} \equiv \begin{align*}
&\{\text{"discard-refs"}\} \text{ LocalOp } (\lambda s. \{ s[| \text{ roots} := \text{roots'} | | \text{roots'}. \text{roots'} \subseteq \text{roots s} \})
\end{align*}
\]

Load and store are defined in Pizlo (201x, Figure 2.9). Dereferencing a reference can increase the set of mutator roots.

abbreviation load :: ('field, 'mut, 'ref) gc-com where
\[
\text{load} \equiv \begin{align*}
&\{\text{"load-choose"}\} \text{ LocalOp } (\lambda s. \{ s[| \text{ tmp-ref} := r, \text{field} := f | | r. f. r \in \text{roots s} \}) ;; \\
&\{\text{"load"}\} \text{ Request } (\lambda s. (\text{mutator m}, \text{ReadRef}(\text{tmp-ref s})(\text{field s}))) \\
&\quad (\lambda mv s. \{ s[| \text{ roots} := \text{roots s} \cup \text{set-option} r | | r. \text{mv} = \text{mv-Ref} r \})
\end{align*}
\]

Storing a reference involves marking both the old and new references, i.e., both insertion and deletion barriers are installed. The deletion barrier preserves the weak tricolour invariant, and the insertion barrier preserves the strong tricolour invariant; see §3.9 for further discussion.
Note that the mutator reads the overwritten reference but does not store it in its roots.

**abbreviation**

\[
\text{mut-\text{deref}} :: \text{location}
\]

\[
\Rightarrow (\text{('field, 'mut, 'ref) local-state} \Rightarrow \text{'ref})
\]

\[
\Rightarrow (\text{('field, 'mut, 'ref) local-state} \Rightarrow \text{'field})
\]

\[
\Rightarrow (\text{('ref option} \Rightarrow \text{'ref option}) \Rightarrow (\text{('field, 'mut, 'ref) local-state} \Rightarrow (\text{('field, 'mut, 'ref) local-state}) \Rightarrow (\text{('field, 'mut, 'ref) gc-com (\{r\} deref})
\]

**where**

\[
\{l\} \text{ deref r f upd} \equiv \{l\} \text{ Request (\lambda s. (mutator m, ReadRef (r s) (f s)))}
\]

\[
(\lambda mv s. \{ \text{ upd (opt-r')} (s\{\mid \text{ghost-honorary-root := set-option opt-r'}}) \mid \text{opt-r'}. mv = mv-Ref opt-r'})
\]

**abbreviation**

\[
\text{write-ref} :: \text{location}
\]

\[
\Rightarrow (\text{('field, 'mut, 'ref) local-state} \Rightarrow \text{'ref})
\]

\[
\Rightarrow (\text{('field, 'mut, 'ref) local-state} \Rightarrow \text{'field})
\]

\[
\Rightarrow (\text{('field, 'mut, 'ref) local-state} \Rightarrow \text{'ref option})
\]

\[
\Rightarrow (\text{'field, 'mut, 'ref) gc-com (\{r\} write-ref})
\]

**where**

\[
\{l\} \text{ write-ref r f r'} \equiv \{l\} \text{ Request (\lambda s. (mutator m, WriteRef (r s) (f s) (r' s))) (\lambda- s. \{s\{\mid \text{ghost-honorary-root := \{}\})})}
\]

**definition**

\[
\text{store} :: (\text{('field, 'mut, 'ref) gc-com}
\]

**where**

\[
\text{store} \equiv
\]

\[
(* \text{Choose vars for: ref} \rightarrow \text{field} := \text{new-ref} *)
\]

\[
\{"\text{store-choose}"\} \text{ LocalOp (\lambda s. \{ s\{ tmp-ref := r, field := f, new-ref := r' \})}
\]

\[
\mid r f r', r \in \text{roots s} \land r' \in \text{Some \{ roots s} \cup \{\text{None} \} \} \};
\]

\[
(* \text{Mark the reference we're about to overwrite. Does not update roots. *})
\]

\[
\{"\text{deref-del}"\} \text{ deref tmp-ref field ref-update ;}
\]

\[
\{"\text{store-del}"\} \text{ mark-object ;}
\]

\[
(* \text{Mark the reference we're about to insert. *})
\]

\[
\{"\text{lop-store-ins}"\} \text{ \'}ref := \text{\'}new-ref ;
\]

\[
\{"\text{store-ins}"\} \text{ mark-object ;}
\]

\[
\{"\text{store-ins}"\} \text{ write-ref tmp-ref field new-ref}
\]

A mutator makes a non-deterministic choice amongst its possible actions. For completeness we allow mutators to issue \text{MFENCE} instructions. We leave \text{CAS} (etc) to future work. Neither has a significant impact on the rest of the development.

**definition**

\[
\text{com} :: (\text{('field, 'mut, 'ref) gc-com}
\]

**where**

\[
\text{com} \equiv
\]
LOOP DO
   "mut local computation" SKIP
   alloc
discard
load
store
"mut mfence" MFENCE
end

2.5 Garbage collector

We abstract the primitive actions of the garbage collector thread.

abbreviation
gc-deref :: location
   ⇒ (('field, 'mut, 'ref) local-state ⇒ 'ref)
   ⇒ (('field, 'mut, 'ref) local-state ⇒ 'field)
   ⇒ (('ref option ⇒ 'ref option) ⇒ ('field, 'mut, 'ref) local-state ⇒ ('field, 'mut, 'ref) local-state) ⇒ ('field, 'mut, 'ref) gc-com
where
gc-deref l r f upd ≡ {l} Request (λs. (gc, ReadRef (r s) (f s))) (λmv s. { upd (r') s |r'. mv = mv-Ref r' })

abbreviation
gc-read-mark :: location
   ⇒ (('field, 'mut, 'ref) local-state ⇒ 'ref)
   ⇒ ((gc-mark option ⇒ gc-mark option) ⇒ ('field, 'mut, 'ref) local-state ⇒ ('field, 'mut, 'ref) local-state) ⇒ ('field, 'mut, 'ref) gc-com
where
gc-read-mark l r upd ≡ {l} Request (λs. (gc, ReadMark (r s))) (λmv s. { upd (m) s |m. mv = mv-Mark m })

syntax
-gc-fassign :: location ⇒ idt ⇒ 'ref ⇒ 'field ⇒ ('field, 'mut, 'ref) gc-com (l- := ` := ` → - [0, 0, 70] 71)
-gc-massign :: location ⇒ idt ⇒ 'ref ⇒ ('field, 'mut, 'ref) gc-com (l- := ` := flag [0, 0] 71)
translations
{l} 'q :: 'r→f => CONST gc-deref l r "f" (update-name q)
l 'm :: 'r→flag => CONST gc-read-mark l r (update-name m)

context gc
begin
abbreviation write-fA :: location ⇒ ('field, 'mut, 'ref) local-state ⇒ gc-mark) ⇒ ('field, 'mut, 'ref) gc-com (\l \r → write-fA) where
\l write-fA f ≡ \l Request (λs. (gc, WritefA (f s))) (λ- s. \{s\})

abbreviation read-fM :: location ⇒ ('field, 'mut, 'ref) gc-com (\l \r → read-fM) where
\l read-fM ≡ \l Request (λs. (gc, ReadfM)) (λmv s. \{ s[fM := m] | m. mv = mv-Mark (Some m) \})

abbreviation write-fM :: location ⇒ ('field, 'mut, 'ref) gc-com (\l \r → write-fM) where
\l write-fM ≡ \l Request (λs. (gc, WriteM (fM s))) (λ- s. \{s\})

abbreviation write-phase :: location ⇒ gc-phase ⇒ ('field, 'mut, 'ref) gc-com (\l \r → write-phase) where
\l write-phase ph ≡ \l Request (λs. (gc, WritePhase ph)) (λ- s. \{s\ phase := ph \})

abbreviation mark-object :: location ⇒ ('field, 'mut, 'ref) gc-com (\l \r → mark-object) where
\l mark-object ≡ mark-object-fn gc l

abbreviation free :: location ⇒ ('field, 'mut, 'ref) local-state ⇒ 'ref) ⇒ ('field, 'mut, 'ref) gc-com (\l \r → free) where
\l free r ≡ \l Request (λs. (gc, ro-Free (r s))) (λ- s. \{s\})

The following CIMP program encodes the garbage collector algorithm proposed in Figure 2.15 of Pizlo (201x).

definition (in gc)
  com :: ('field, 'mut, 'ref) gc-com
where
  com ≡
  LOOP DO
    "idle-noop" handshake-noop ;; (\hp-Idle *)
    "idle-read-fM" read-fM ;;
    "idle-invert-fM" fM := (\¬ fM) ;;
    "idle-write-fM" write-fM ;;
    "idle-flip-noop" handshake-noop ;; (\hp-IdleInit *)
    "idle-phase-init" write-phase ph-Init ;;
    "init-noop" handshake-noop ;; (\hp-InitMark *)
    "init-phase-mark" write-phase ph-Mark ;;
    "mark-read-fM" read-fM ;;
    "mark-write-fA" write-fA fM ;;
    "mark-noop" handshake-noop ;; (\hp-Mark *)

18
"mark-loop-get-roots"); (hp-IdleMarkSweep *)

"mark-loop" WHILE not empty W DO
  "mark-loop-inner" WHILE not empty W DO
    "mark-loop-choose-ref" \tmp-ref \in W ;;
    "mark-loop-fields" \field-set := UNIV ;;
    "mark-loop-object-loop" WHILE not empty \field-set DO
      "mark-loop-choose-field" \field := \field-set ;;
      "mark-loop-deref" \ref := \tmp-ref \rightarrow \field ;;
    OD ;;
    "mark-loop-blacken" W := (W - {\tmp-ref})
  OD ;;
"mark-loop-get-work" handshake-get-work
OD ;;
("sweep")

"mark-end" write-phase ph-Sweep ;;
"sweep-read-fM" read-fM ;;
"sweep.refs" \refs := UNIV ;;
"sweep-loop" WHILE not empty \refs DO
  "sweep-choose-ref" \tmp-ref \in \refs ;;
  "sweep-read-mark" \mark := \tmp-ref \rightarrow flag ;;
  "sweep-check" IF not null \mark and the \circ \mark neq fM THEN
    "sweep-free" free \tmp-ref
  FI ;;
  "sweep-ref-done" \refs := (\refs - {\tmp-ref})
OD ;;
"sweep-idle" write-phase ph-Idle
OD

end

primrec
  gc-pgms :: 'mut process-name \Rightarrow ('field, 'mut, 'ref) gc-com
where
  gc-pgms (mutator m) = mut-m.com m
| gc-pgms gc = gc.com
| gc-pgms sys = sys.com

19
3 Invariants and Proofs

3.1 Constructors for sets of locations.

**abbreviation** prefixed :: location ⇒ location set where
prefixed p ≡ { l . prefixeq p l }

**abbreviation** suffixed :: location ⇒ location set where
suffixed p ≡ { l . suffixeq p l }

3.2 Hoare triples

Specialise CIMP’s pre/post validity to our system.

**definition**
valid-proc :: (′field, ′mut, ′ref) gc-pred ⇒ ′mut process-name ⇒ (′field, ′mut, ′ref) gc-pred ⇒ bool (\{P\} - \{Q\})
where
\{P\} p \{Q\} ≡ ∀ (c, afts) ∈ vcg-fragments (gc-pgms p). (gc-pgms, p, afts |\{P\} c \{Q\})

**abbreviation**
valid-proc-inv-syn :: (′field, ′mut, ′ref) gc-pred ⇒ ′mut process-name ⇒ bool (\{P\} - [100,0])
where
\{P\} p \{P\} \{P\}\{proof\}\{proof\}\{proof\}

As we elide formal proofs in this document, we also omit our specialised proof tactics. These support essentially traditional local correctness and non-interference proofs. Their most interesting aspect is the use of Isabelle’s parallelism to greatly reduce system latency.

\(⟨ML\rangle\{proof\}\{proof\}\{proof\}\{proof\}\{proof\}\{proof\}\{proof\}\{proof\}\{ML\}\)

3.3 Functions and predicates

We define a pile of predicates and accessor functions for the process’s local states. One might hope that a more sophisticated approach would automate all of this (cf Schirmer and Wenzel (2009)).

**abbreviation**
is-mw-Mark w ≡ ∃ r fl. w = mw-Mark r fl

**abbreviation**
is-mw-Mutate w ≡ ∃ f r r′. w = mw-Mutate r f r′

**abbreviation**
is-mw-fA w ≡ ∃ fl. w = mw-fA fl

**abbreviation**
is-mw-fM w ≡ ∃ fl. w = mw-fM fl

**abbreviation**
is-mw-Phase w ≡ ∃ ph. w = mw-Phase ph

**abbreviation**
(infix) pred-in-W :: ′ref ⇒ ′mut process-name ⇒ (′field, ′mut, ′ref) lsts-pred
(infix in-\text{\textquotesingle}W\text{\textquotesingle}\ 50) where
r in-W p ≡ λs. r ∈ W (s p)

**abbreviation**
(infix) pred-in-ghost-honorary-grey :: ′ref ⇒ ′mut process-name ⇒ (′field, ′mut, ′ref) lsts-pred
(infix in-′\text{\textquotesingle}ghost\text{\textquotesingle}-honorary\text{\textquotesingle}\text{-grey}\ 50) where
\[
\text{r in-ghost-honorary-grey } p \equiv \lambda s. \ r \in \text{ghost-honorary-grey } (s \ p)
\]

**context gc**

**begin**

**abbreviation**

\[\text{valid-gc-syn :: ('}\text{field}', 'mut', 'ref') gc-loc-comp \Rightarrow ('}\text{field}', 'mut', 'ref') gc-pred \Rightarrow ('}\text{field}', 'mut', 'ref') gc-com \Rightarrow ('}\text{field}', 'mut', 'ref') gc-pred \Rightarrow \text{bool} \equiv (\text{- } \text{|} / \text{- } \text{|})\]

**where**

\[ afts \models \{P\} c \{Q\} \equiv \text{gc-ghosts, gc, afts} \models \{P\} c \{Q\} \]

**abbreviation**\[\text{valid-gc-inv-syn :: ('}\text{field}', 'mut', 'ref') gc-loc-comp \Rightarrow ('}\text{field}', 'mut', 'ref') gc-pred \Rightarrow ('}\text{field}', 'mut', 'ref') gc-com \Rightarrow \text{bool} \equiv (\text{- } \text{|} / \text{- } \text{|})\]

**where**

\[ afts \models \{P\} c \equiv afts \models \{P\} c \{P\} \]

**end**

**abbreviation**\[\text{gc-cas-mark } s \equiv \text{cas-mark } (s \ gc)\]

**abbreviation**\[\text{gc-fM } s \equiv \text{fM } (s \ gc)\]

**abbreviation**\[\text{gc-field } s \equiv \text{field } (s \ gc)\]

**abbreviation**\[\text{gc-mark } s \equiv \text{mark } (s \ gc)\]

**abbreviation**\[\text{gc-mut } s \equiv \text{mut } (s \ gc)\]

**abbreviation**\[\text{gc-muts } s \equiv \text{muts } (s \ gc)\]

**abbreviation**\[\text{gc-phase } s \equiv \text{phase } (s \ gc)\]

**abbreviation**\[\text{gc-tmp-ref } s \equiv \text{tmp-ref } (s \ gc)\]

**abbreviation**\[\text{gc-ghost-honorary-grey } s \equiv \text{ghost-honorary-grey } (s \ gc)\]

**abbreviation**\[\text{gc-ref } s \equiv \text{ref } (s \ gc)\]

**abbreviation**\[\text{gc-refs } s \equiv \text{refs } (s \ gc)\]

**abbreviation**\[\text{gc-the-ref } \equiv \text{the } \circ \text{ gc-ref}\]

**abbreviation**\[\text{gc-W } s \equiv \text{W } (s \ gc)\]

**abbreviation**\[\text{at-gc :: location } \Rightarrow ('}\text{field}', 'mut', 'ref') lsts-pred \Rightarrow ('}\text{field}', 'mut', 'ref') gc-pred \equiv \text{at-gc } l \ P \equiv \text{at gc } l \ \text{imp LSTP } P\]

**abbreviation**\[\text{atS-gc :: location set } \Rightarrow ('}\text{field}', 'mut', 'ref') lsts-pred \Rightarrow ('}\text{field}', 'mut', 'ref') gc-pred \text{ where } \text{atS-gc } ls \ P \equiv \text{atS gc } ls \ \text{imp LSTP } P\]

**context mut-m**

**begin**

**abbreviation**\[\text{valid-mut-syn :: ('}\text{field}', 'mut', 'ref') gc-loc-comp \Rightarrow ('}\text{field}', 'mut', 'ref') gc-pred \Rightarrow ('}\text{field}', 'mut', 'ref') gc-com \Rightarrow ('}\text{field}', 'mut', 'ref') gc-pred \Rightarrow \text{bool}\]
\[(\cdot \models \{\cdot\} / \cdot)\]

where
\[
afts \models \{P\} c \{Q\} \equiv \text{gc-pgms}, \text{mutator } m, \ afts \models \{P\} c \{Q\}
\]

\textbf{abbreviation} valid-mut-inv-syn :: ('field, 'mut, 'ref) gc-loc-comp \Rightarrow ('field, 'mut, 'ref) gc-pred \Rightarrow ('field, 'mut, 'ref) gc-com \Rightarrow \text{bool} \ (\cdot \models \{\cdot\} / \cdot) \\\text{where}
\[
afts \models \{P\} c \equiv \ afts \models \{P\} c \{P\}
\]

\textbf{abbreviation} at-mut :: location \Rightarrow ('field, 'mut, 'ref) lsts-pred \Rightarrow ('field, 'mut, 'ref) gc-pred \\\text{where}
\[
\text{at-mut } l \ P \equiv \text{at} (\text{mutator } m) \ l \ \text{imp} \ LSTP \ P
\]

\textbf{abbreviation} atS-mut :: location set \Rightarrow ('field, 'mut, 'ref) lsts-pred \Rightarrow ('field, 'mut, 'ref) gc-pred \\\text{where}
\[
\text{atS-mut } ls \ P \equiv \text{atS} (\text{mutator } m) \ ls \ \text{imp} \ LSTP \ P
\]

\textbf{abbreviation} mut-cas-mark s \equiv \text{cas-mark} (s \ (\text{mutator } m))
\textbf{abbreviation} mut-field s \equiv \text{field} (s \ (\text{mutator } m))
\textbf{abbreviation} mut-fM s \equiv \text{fM} (s \ (\text{mutator } m))
\textbf{abbreviation} mut-ghost-honorary-grey s \equiv \text{ghost-honorary-grey} (s \ (\text{mutator } m))
\textbf{abbreviation} mut-ghost-handshake-phase s \equiv \text{ghost-handshake-phase} (s \ (\text{mutator } m))
\textbf{abbreviation} mut-ghost-honorary-root s \equiv \text{ghost-honorary-root} (s \ (\text{mutator } m))
\textbf{abbreviation} mut-mark s \equiv \text{mark} (s \ (\text{mutator } m))
\textbf{abbreviation} mut-new-ref s \equiv \text{new-ref} (s \ (\text{mutator } m))
\textbf{abbreviation} mut-phase s \equiv \text{phase} (s \ (\text{mutator } m))
\textbf{abbreviation} mut-ref s \equiv \text{ref} (s \ (\text{mutator } m))
\textbf{abbreviation} mut-tmp-ref s \equiv \text{tmp-ref} (s \ (\text{mutator } m))
\textbf{abbreviation} mut-the-new-ref \equiv \text{the} \circ \text{mut-new-ref}
\textbf{abbreviation} mut-the-ref \equiv \text{the} \circ \text{mut-ref}
\textbf{abbreviation} mut-refs s \equiv \text{refs} (s \ (\text{mutator } m))
\textbf{abbreviation} mut-roots s \equiv \text{roots} (s \ (\text{mutator } m))
\textbf{abbreviation} mut-W s \equiv \text{W} (s \ (\text{mutator } m))

\textbf{end}

\textbf{context} sys

\textbf{begin}

\textbf{abbreviation}
\[
\text{valid-sys-syn} :: ('field, 'mut, 'ref) gc-loc-comp \Rightarrow ('field, 'mut, 'ref) gc-pred \Rightarrow ('field, 'mut, 'ref) gc-com \Rightarrow ('field, 'mut, 'ref) gc-pred \Rightarrow \text{bool} \ (\cdot \models \{\cdot\} / \cdot) \\\text{where}
\]
\[
afts \models \{P\} c \{Q\} \equiv \text{gc-pgms}, \text{sys}, \ afts \models \{P\} c \{Q\}
\]

\textbf{abbreviation} valid-sys-inv-syn :: ('field, 'mut, 'ref) gc-loc-comp \Rightarrow ('field, 'mut, 'ref) gc-pred \Rightarrow ('field, 'mut, 'ref) gc-com \Rightarrow \text{bool} \ (\cdot \models \{\cdot\} / \cdot) \\\text{where}
$$\text{afts} \models \{P\} \ c \equiv \ \text{afts} \models \{P\} \ c \ \{P\}$$

end

abbreviation sys-heap :: ('field, 'mut, 'ref) lsts ⇒ 'ref ⇒ ('field, 'ref) object option where
sys-heap s ≡ heap (s sys)

abbreviation sys-fA s ≡ fA (s sys)
abbreviation sys-fM s ≡ fM (s sys)
abbreviation sys-ghost-honorary-grey s ≡ ghost-honorary-grey (s sys)
abbreviation sys-ghost-handshake-in-sync m s ≡ ghost-handshake-in-sync (s sys) m
abbreviation sys-ghost-handshake-phase s ≡ ghost-handshake-phase (s sys)
abbreviation sys-handshake-pending m s ≡ handshake-pending (s sys) m
abbreviation sys-handshake-type s ≡ handshake-type (s sys)
abbreviation sys-mem-write-buffers p s ≡ mem-write-buffers (s sys)
abbreviation sys-mem-lock s ≡ mem-lock (s sys)
abbreviation sys-phase s ≡ phase (s sys)
abbreviation sys-W s ≡ W (s sys)

abbreviation atS-sys :: location set ⇒ ('field, 'mut, 'ref) object option where
atS-sys ls P ≡ atS sys ls imp LSTP P

Projections on TSO buffers.

abbreviation (input) tso-unlocked s ≡ mem-lock (s sys) = None
abbreviation (input) tso-locked-by p s ≡ mem-lock (s sys) = Some p

abbreviation (input) tso-pending p P s ≡ filter P (mem-write-buffers (s sys)) p
abbreviation (input) tso-pending-write p w s ≡ w ∈ set (mem-write-buffers (s sys)) p

abbreviation (input) tso-pending-fA p ≡ tso-pending p is-mw-fA
abbreviation (input) tso-pending-fM p ≡ tso-pending p is-mw-fM
abbreviation (input) tso-pending-mark p ≡ tso-pending p is-mw-Mark
abbreviation (input) tso-pending-mutate p ≡ tso-pending p is-mw-Mutate
abbreviation (input) tso-pending-phase p ≡ tso-pending p is-mw-Phase

abbreviation (input) tso-no-pending-marks ≡ ALLS p. list-null (tso-pending-mark p)

A somewhat-useful abstraction of the heap, following l4.verified, which asserts that there is an object at the given reference with the given property.

definition obj-at :: ('field, 'ref) object ⇒ bool ⇒ 'ref ⇒ ('field, 'mut, 'ref) lsts-pred where
obj-at P r ≡ λs. case sys-heap s r of None ⇒ False | Some obj ⇒ P obj

abbreviation (input) valid-ref :: 'ref ⇒ ('field, 'mut, 'ref) lsts-pred where
valid-ref r ≡ obj-at (True) r

definition valid-null-ref :: 'ref option ⇒ ('field, 'mut, 'ref) lsts-pred where
valid-null-ref \( r \equiv \text{case } r \text{ of } \text{None } \Rightarrow \langle \text{True} \rangle \mid \text{Some } r' \Rightarrow \text{valid-ref } r' \)

\text{abbreviation} pred-points-to :: \('ref \Rightarrow \text{('field, 'mut, 'ref)}\) \text{lsts-pred (infix points'-to 51)} \text{ where} \\
x \text{points-to } y \equiv \lambda s. \text{obj-at } (\lambda \text{obj. } y \in \text{ran } (\text{obj-fields obj})) x s

We use Isabelle’s standard transitive-reflexive closure to define reachability through the heap.

\text{abbreviation} pred-reaches :: \('ref \Rightarrow \text{('field, 'mut, 'ref)}\) \text{lsts-pred (infix reaches 51)} \text{ where} \\
x \text{reaches } y \equiv \lambda s. (\lambda x y. (x \text{points-to } y) s) \ast x y

The predicate \text{obj-at-field-on-heap} asserts that if there is an object at \( r.f \) on the heap, then it satisfies \( P \).

\text{definition} obj-at-field-on-heap :: \('ref \Rightarrow \text{bool}\) \Rightarrow \text{'field} \Rightarrow \text{'field} \Rightarrow \text{'field} \Rightarrow \text{bool}\) \text{lsts-pred} \text{ where} \\
\text{case } \text{Option.map-option } \text{obj-fields } (\text{sys-heap } s r) \text{ of} \\
\text{None } \Rightarrow \text{False} \\
\mid \text{Some } fs \Rightarrow (\text{case } fs f \text{ of } \text{None } \Rightarrow \text{True} \\
\mid \text{Some } r' \Rightarrow P r')

3.4 Garbage collector locations.

\text{definition} idle-locs :: \text{location set} \text{ where} \\
idle-locs \equiv \text{prefixed } "\text{idle}" 

\text{definition} init-locs :: \text{location set} \text{ where} \\
init-locs \equiv \text{prefixed } "\text{init}" 

\text{definition} mark-locs :: \text{location set} \text{ where} \\
mark-locs \equiv \text{prefixed } "\text{mark}" 

\text{definition} mark-loop-locs :: \text{location set} \text{ where} \\
mark-loop-locs \equiv \text{prefixed } "\text{mark-loop}" 

\text{definition} sweep-locs :: \text{location set} \text{ where} \\
sweep-locs \equiv \text{prefixed } "\text{sweep" (ML)} \langle \text{proof}\rangle \langle \text{proof}\rangle \langle \text{proof}\rangle \langle \text{proof}\rangle \langle \text{proof}\rangle \langle \text{proof}\rangle \langle \text{proof}\rangle \langle \text{proof}\rangle \langle \text{proof}\rangle 

3.5 Coarse TSO invariants

Very coarse invariants about what processes write, and when they hold the TSO lock.

\text{abbreviation} gc-writes :: \('field, 'ref) \text{mem-write-action} \Rightarrow \text{bool} \text{ where} \\
gc-writes w \equiv \text{case } w \text{ of } \text{mw-Mark } - - \Rightarrow \text{True} \mid \text{mw-Phase } - \Rightarrow \text{True} \mid \text{mw-fM } - \Rightarrow \text{True} \mid \text{mw-fA } - \Rightarrow \text{True} \mid - \Rightarrow \text{False} 

\text{abbreviation} mut-writes :: \('field, 'ref) \text{mem-write-action} \Rightarrow \text{bool} \text{ where} \\
mut-writes w \equiv \text{case } w \text{ of } \text{mw-Mutate } - - \Rightarrow \text{True} \mid \text{mw-Mark } - \Rightarrow \text{True} \mid - \Rightarrow \text{False}
definition tso-writes-inv :: (field, mut, ref) lsts-pred where
\[
\text{tso-writes-inv} \equiv \\
(\text{ALLS } w. \ tso-pending-write gc w \imp \langle gc-writes w \rangle) \\
\text{and } (\text{ALLS } m w. \ tso-pending-write (mutator m) w \imp \langle \text{mut-writes } w \rangle) \\
\text{proof} \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \\
\]

3.5.1 Locations where the TSO lock is held

The GC holds the TSO lock only during the CAS in mark-object.

definition gc-tso-lock-locs :: location set where
\[
\text{gc-tso-lock-locs} \equiv \bigcup l \in \{ "mo-co-cmark", "mo-co-ctest", "mo-co-mark", "mo-co-unlock" \}. \\
\text{suffixed } l \\
\langle \text{ML} \rangle \\
\]
definition (in gc) tso-lock-invL :: (field, mut, ref) gc-pred where
\[
[\text{inv}] : \text{tso-lock-invL} \equiv \\
\text{atS-gc gc-tso-lock-locs } \langle \text{tso-locked-by gc} \rangle \\
\text{and } \text{atS-gc } \langle \text{not tso-locked-by gc} \rangle \\
\text{proof} \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \\
\]

A mutator holds the TSO lock only during the CASs in mark-object.

definition mut-tso-lock-locs \equiv \\
\bigcup l \in \{ "mo-co-cmark", "mo-co-ctest", "mo-co-mark", "mo-co-unlock" \}. \\
\text{suffixed } l \\
\langle \text{ML} \rangle \\
\]
definition (in mut-m) tso-lock-invL :: (field, mut, ref) gc-pred where
\[
[\text{inv}] : \text{tso-lock-invL} \equiv \\
\text{atS-mut mut-tso-lock-locs } \langle \text{tso-locked-by (mutator m)} \rangle \\
\text{and } \text{atS-mut } \langle \text{not tso-locked-by (mutator m)} \rangle \\
\text{proof} \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \\
\]

3.6 Handshake phases

The mutators can be at most one step behind the garbage collector (and system). If any
mutator is behind then the GC is stalled on a pending handshake. Unfortunately this is a
complicated by needing to consider the handshake type due to get-work. This relation is very
precise.

definition hp-step-rel :: (bool \times handshake-type \times handshake-phase \times handshake-phase) set
where
\[
\text{hp-step-rel} \equiv \\
\{ \text{True} \} \times \{ (ht-NOOP, hp, hp) \mid hp, hp \in \{ hp-Idle, hp-IdleInit, hp-InitMark, hp-Mark \} \} \\
\cup \{ (ht-GetRoots, hp-IdleMarkSweep, hp-IdleMarkSweep) \\
, (ht-GetWork, hp-IdleMarkSweep, hp-IdleMarkSweep) \} \} \\
\cup \{ \text{False} \} \times \{ (ht-NOOP, hp-Idle, hp-IdleMarkSweep) \\
, (ht-NOOP, hp-IdleInit, hp-Idle) \\
, (ht-NOOP, hp-InitMark, hp-IdleInit) \\
, (ht-NOOP, hp-Mark, hp-InitMark) \} \\
\]

25
\begin{verbatim}

, (ht-GetRoots, hp-IdleMarkSweep, hp-Mark)
, (ht-GetWork, hp-IdleMarkSweep, hp-IdleMarkSweep) }

\textbf{definition} handshake-phase-inv :: ('field, 'mut, 'ref) lsts-pred where
\begin{align*}
\text{handshake-phase-inv} & \equiv \text{ALLS m.} \\
& \quad (\text{sys-ghost-handshake-in-sync m} \otimes \text{sys-handshake-type} \\
& \quad \otimes \text{sys-ghost-handshake-phase} \otimes \text{mut-m.mut-ghost-handshake-phase m}) \in \langle \text{hp-step-rel} \rangle
\end{align*}

Connect \text{sys-ghost-handshake-phase} with locations in the GC.

\textbf{definition} hp-Idle-locs \equiv
\begin{align*}
& \text{(prefixed "idle-noop" } - \{ "idle-noop-mfence", "idle-noop-init-type" \}) \\
& \cup \{ "idle-read-fM", "idle-invert-fM", "idle-write-fM", "idle-flip-noop-mfence", "idle-flip-noop-init-type" \}
\end{align*}
\langle ML \rangle

\textbf{definition} hp-IdleInit-locs \equiv
\begin{align*}
& \text{(prefixed "idle-flip-noop" } - \{ "idle-flip-noop-mfence", "idle-flip-noop-init-type" \}) \\
& \cup \{ "idle-phase-init", "init-noop-mfence", "init-noop-init-type" \}
\end{align*}
\langle ML \rangle

\textbf{definition} hp-InitMark-locs \equiv
\begin{align*}
& \text{(prefixed "init-noop" } - \{ "init-noop-mfence", "init-noop-init-type" \}) \\
& \cup \{ "init-phase-mark", "mark-read-fM", "mark-write-fA", "mark-noop-mfence", "mark-noop-init-type" \}
\end{align*}
\langle ML \rangle

\textbf{definition} hp-IdleMarkSweep-locs \equiv
\begin{align*}
& \{ "idle-noop-mfence", "idle-noop-init-type", "mark-end" \} \\
& \cup \text{sweep-locs} \\
& \cup (\text{mark-loop-locs } - \{ "mark-loop-get-roots-init-type" \})
\end{align*}
\langle ML \rangle

\textbf{definition} hp-Mark-locs \equiv
\begin{align*}
& \text{(prefixed "mark-noop" } - \{ "mark-noop-mfence", "mark-noop-init-type" \}) \\
& \cup \{ "mark-loop-get-roots-init-type" \}
\end{align*}
\langle ML \rangle

\textbf{abbreviation}
\begin{align*}
\text{hs-noop-prefixes} & \equiv \{ "idle-noop", "idle-flip-noop", "init-noop", "mark-noop" \}
\end{align*}

\textbf{definition} hs-noop-locs \equiv
\begin{align*}
& \bigcup l \in \text{hs-noop-prefixes}. \text{prefixed } l - \{ \text{suffixed "-noop-mfence" } \cup \text{suffixed "-noop-init-type"} \}
\end{align*}
\langle ML \rangle

\textbf{definition} hs-get-roots-locs \equiv
\begin{align*}
& \text{prefixed "mark-loop-get-roots" } - \{ "mark-loop-get-roots-init-type" \}
\end{align*}
\end{verbatim}
\[ML\]

**definition** hs-get-work-locs ≡

prefixed "mark-loop-get-work" − \{"mark-loop-get-work-init-type"\}

\[ML\]

**abbreviation** hs-prefixes ≡

hs-noop-prefixes ∪ \{"mark-loop-get-roots", "mark-loop-get-work" \}

**definition** hs-init-loop-locs ≡ \( \bigcup l \in \text{hs-prefixes}. \text{prefixed} (l @ ".-init-loop") \)

\[ML\]

**definition** hs-done-loop-locs ≡ \( \bigcup l \in \text{hs-prefixes}. \text{prefixed} (l @ ".-done-loop") \)

\[ML\]

**definition** hs-done-locs ≡ \( \bigcup l \in \text{hs-prefixes}. \text{prefixed} (l @ ".-done") \)

\[ML\]

**definition** hs-none-pending-locs ≡ \(- (\text{hs-init-loop-locs} \cup \text{hs-done-locs}) \)

\[ML\]

**definition** hs-in-sync-locs ≡

\(- (\bigcup l \in \text{hs-prefixes}. \text{prefixed} (l @ ".-init")) \cup \text{hs-done-locs} \)

\bigcup l \in \text{hs-prefixes}. \{l @ ".-init-type"\}

\[ML\]

**definition** hs-out-of-sync-locs ≡

\( \bigcup l \in \text{hs-prefixes}. \{l @ ".-init-muts"\} \)

\[ML\]

**definition** hs-mut-in-muts-locs ≡

\( \bigcup l \in \text{hs-prefixes}. \{l @ ".-init-loop-set-pending", l @ ".-init-loop-done"\} \)

\[ML\]

**definition** hs-init-loop-done-locs ≡

\( \bigcup l \in \text{hs-prefixes}. \{l @ ".-init-loop-done"\} \)

\[ML\]

**definition** hs-init-loop-not-done-locs ≡

hs-init-loop-locs − \( \bigcup l \in \text{hs-prefixes}. \{l @ ".-init-loop-done"\} \)

\[ML\]

**definition** (in gc) handshake-invL :: ("field", "mut", "ref") gc-pred where

[inv]: handshake-invL ≡

\( \text{atS-gc hs-noop-locs} \quad (\text{sys-handshake-type eq \{\text{ht-NOOP}\}}) \)

and \( \text{atS-gc hs-get-roots-locs} \quad (\text{sys-handshake-type eq \{\text{ht-GetRoots}\}}) \)

and \( \text{atS-gc hs-get-work-locs} \quad (\text{sys-handshake-type eq \{\text{ht-GetWork}\}}) \)
and atS-gc hs-mut-in-muts-locs (gc-mut in gc-muts)
and atS-gc hs-init-loop-locs (ALLS m. not ⟨m⟩ in gc-muts imp sys-handshake-pending m
or sys-ghost-handshake-in-sync m)
and atS-gc hs-init-loop-not-done-locs (ALLS m. ⟨m⟩ in gc-muts imp not sys-handshake-pending m
and not sys-ghost-handshake-in-sync m)
and atS-gc hs-init-loop-done-locs ((sys-handshake-pending ⊃ gc-mut
or sys-ghost-handshake-in-sync ⊃ gc-mut)
and (ALLS m. ⟨m⟩ in gc-muts and ⟨m⟩ neq gc-mut
imp not sys-handshake-pending m
and not sys-ghost-handshake-in-sync m))
and atS-gc hs-done-locs (ALLS m. sys-handshake-pending m or sys-ghost-handshake-in-sync m)
and atS-gc hs-done-loop-locs (ALLS m. not ⟨m⟩ in gc-muts imp not sys-handshake-pending m)
and atS-gc hs-none-pending-locs (ALLS m. not sys-handshake-pending m)
and atS-gc hs-in-sync-locs (ALLS m. sys-ghost-handshake-in-sync m)
and atS-gc hs-out-of-sync-locs (ALLS m. not sys-handshake-pending m
and not sys-ghost-handshake-in-sync m)
and atS-gc hp-Idle-locs (sys-ghost-handshake-phase eq ⟨hp-Idle⟩)
and atS-gc hp-IdleInit-locs (sys-ghost-handshake-phase eq ⟨hp-IdleInit⟩)
and atS-gc hp-InitMark-locs (sys-ghost-handshake-phase eq ⟨hp-InitMark⟩)
and atS-gc hp-IdleMarkSweep-locs (sys-ghost-handshake-phase eq ⟨hp-IdleMarkSweep⟩)
and atS-gc hp-Mark-locs (sys-ghost-handshake-phase eq ⟨hp-Mark⟩)

Local handshake phase invariant for the mutators.

definition mut-no-pending-mutates-locs ≡
prefixed "hs-noop" ⊃ { "hs-noop", "hs-noop-mfence" }
∪ (prefixed "hs-get-roots" ⊃ { "hs-get-roots", "hs-get-roots-mfence" })
∪ (prefixed "hs-get-work" ⊃ { "hs-get-work", "hs-get-work-mfence" })

ML

definition (in mut-m) handshake-invL :: ('field, 'mut, 'ref) gc-pred where
[inv]: handshake-invL ≡
atS-mut (prefixed "hs-noop") (sys-handshake-type eq ⟨ht-NOOP⟩ and sys-handshake-pending m)
and atS-mut (prefixed "hs-get-roots") (sys-handshake-type eq ⟨ht-GetRoots⟩ and sys-handshake-pending m)
and atS-mut (prefixed "hs-get-work") (sys-handshake-type eq ⟨ht-GetWork⟩ and sys-handshake-pending m)
and atS-mut mut-no-pending-mutates-locs (list-null (tso-pending-mutate (mutator m)))

Relate sys-ghost-handshake-phase, gc-phase, sys-phase and writes to the phase in the GC’s
TSO buffer.
The first relation treats the case when the GC’s TSO buffer does not contain any writes to the phase.
The second relation exhibits the data race on the phase variable: we need to precisely track the possible states of the GC’s TSO buffer.

**Definition**

handshake-phase-rel :: handshake-phase ⇒ bool ⇒ gc-phase ⇒ bool

handshake-phase-rel hp in-sync ph ≡

  case hp of
    hp-Idle ⇒ ph = ph-Idle
    | hp-IdleInit ⇒ ph = ph-Idle ∨ (in-sync ∧ ph = ph-Init)
    | hp-InitMark ⇒ ph = ph-Init ∨ (in-sync ∧ ph = ph-Mark)
    | hp-Mark ⇒ ph = ph-Mark
    | hp-IdleMarkSweep ⇒ ph = ph-Mark ∨ (in-sync ∧ ph ∈ { ph-Idle, ph-Sweep })

**Definition**

phase-rel :: (bool × handshake-phase × gc-phase × gc-phase × (′field, ′ref) mem-write-action list) set

phase-rel ≡

  \{ (in-sync, hp, ph, ph, [], ) | in-sync hp ph. handshake-phase-rel hp in-sync ph \} ∪ \{ {True} × \{ (hp-IdleInit, ph-Init, ph-Idle, [mw-Phase ph-Init]), (hp-InitMark, ph-Mark, ph-Init, [mw-Phase ph-Mark]), (hp-IdleMarkSweep, ph-Sweep, ph-Mark, [mw-Phase ph-Sweep]), (hp-IdleMarkSweep, ph-Idle, ph-Mark, [mw-Phase ph-Sweep, mw-Phase ph-Idle]), (hp-IdleMarkSweep, ph-Idle, ph-Sweep, [mw-Phase ph-Idle]) \} \}

**Definition**

phase-rel-inv :: (′field, ′mut, ′ref) lsts-pred

phase-rel-inv ≡ (ALLS m. sys-ghost-handshake-in-sync m) ⊗ sys-ghost-handshake-phase ⊗ gc-phase ⊗ sys-phase ⊗ tso-pending-phase gc in ⟨phase-rel⟩⟨proof⟩⟨proof⟩⟨proof⟩

Tie the garbage collector’s control location to the value of gc-phase.

**Definition**

no-pending-phase-locs :: location set

no-pending-phase-locs ≡

  \{ idle-locs − \{ "idle-noop-mfence" \} \} ∪ \{ init-locs − \{ "init-noop-mfence" \} \} ∪ \{ mark-locs − \{ "mark-read-fM", "mark-write-fA", "mark-noop-mfence" \} \}

(ML)

**Definition**

(in gc) phase-invL :: (′field, ′mut, ′ref) gc-pred

phase-invL ≡

  atS-gc idle-locs (gc-phase eq ⟨ph-Idle⟩)
and atS-gc init-locs (gc-phase eq ⟨ph-Init⟩)
and atS-gc mark-locs (gc-phase eq ⟨ph-Mark⟩)
and atS-gc sweep-locs (gc-phase eq ⟨ph-Sweep⟩)
and atS-gc no-pending-phase-locs (list-null (tso-pending-phase gc))⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩(ML)

Validity of sys-fM wrt gc-fM and the handshake phase. Effectively we use gc-fM as ghost state. We also include the TSO lock to rule out the GC having any pending marks during the hp-Idle handshake phase.
definition \( fM\text{-rel} :: (\text{bool} \times \text{handshake-phase} \times \text{gc-mark} \times \text{gc-mark} \times (\text{field}, \text{ref}) \text{mem-write-action list} \times \text{bool}) \text{ set where} \)

\[ fM\text{-rel} = \]
\[ \{ (\text{in-sync}, \text{hp}, fM, fM, [], l) \mid fM \text{ hp in-sync } l. \text{ hp = hp-Idle } \rightarrow \text{ in-sync } \} \]
\[ \cup \{ (\text{in-sync}, \text{hp-Idle}, fM, fM', [], l) \mid fM fM' \text{ in-sync } l. \text{ in-sync } \} \]
\[ \cup \{ (\text{in-sync}, \text{hp-Idle}, \neg\text{fM}, fM, [\text{mw-fM } (\neg fM)], \text{False}) \mid fM \text{ in-sync. in-sync } \} \]

definition \( fM\text{-rel-inv} :: (\text{field}, \text{mut}, \text{ref}) \text{lsts-pred where} \)

\[ fM\text{-rel-inv} \equiv (\text{ALLS } m. \text{sys-ghost-handshake-in-sync } m) \otimes \text{sys-ghost-handshake-phase} \otimes \text{gc-fM} \otimes \text{sys-fM} \otimes \text{tso-pending-fM gc} \otimes (\text{sys-mem-lock eq } (\text{Some gc})) \text{ in } (fM\text{-rel}) \]

definition \( fA\text{-rel} :: (\text{bool} \times \text{handshake-phase} \times \text{gc-mark} \times \text{gc-mark} \times (\text{field}, \text{ref}) \text{mem-write-action list} \times \text{bool}) \text{ set where} \)

\[ fA\text{-rel} = \]
\[ \{ (\text{in-sync}, \text{hp-Idle}, fA, fM, []) \mid fA fM \text{ in-sync. in-sync } \rightarrow fA = fM \} \]
\[ \cup \{ (\text{in-sync}, \text{hp-IdleInit}, fA, \neg fA, []) \mid fA \text{ in-sync. True } \} \]
\[ \cup \{ (\text{in-sync}, \text{hp-InitMark}, fA, \neg fA, [\text{mw-fA } (\neg fA)]) \mid fA \text{ in-sync. in-sync } \} \]
\[ \cup \{ (\text{in-sync}, \text{hp-InitMark}, fA, fM, []) \mid fA fM \text{ in-sync. in-sync } \rightarrow fA \neq fM \} \]
\[ \cup \{ (\text{in-sync}, \text{hp-Mark}, fA, fA, []) \mid fA \text{ in-sync. True } \} \]
\[ \cup \{ (\text{in-sync}, \text{hp-IdleMarkSweep}, fA, fA, []) \mid fA \text{ in-sync. True } \} \]

definition \( fA\text{-rel-inv} :: (\text{field}, \text{mut}, \text{ref}) \text{lsts-pred where} \)

\[ fA\text{-rel-inv} \equiv (\text{ALLS } m. \text{sys-ghost-handshake-in-sync } m) \otimes \text{sys-ghost-handshake-phase} \otimes \text{sys-fA} \otimes \text{gc-fM} \otimes \text{tso-pending-fA gc} \text{ in } (fA\text{-rel}) \]

definition \( fM\text{-eq-locs} :: \text{location set where} \)

\[ fM\text{-eq-locs} \equiv (\neg \{ "\text{idle-write-fM}", "\text{idle-flip-noop-mfence}" \}) \]

\( \langle ML \rangle \)

definition \( fM\text{-tso-empty-locs} :: \text{location set where} \)

\[ fM\text{-tso-empty-locs} \equiv (\neg \{ "\text{idle-flip-noop-mfence}" \}) \]

\( \langle ML \rangle \)

definition \( fA\text{-tso-empty-locs} :: \text{location set where} \)

\[ fA\text{-tso-empty-locs} \equiv (\neg \{ "\text{mark-noop-mfence}" \}) \]

\( \langle ML \rangle \)

definition \( fA\text{-eq-locs} :: \text{location set where} \)

\[ fA\text{-eq-locs} \equiv \{ "\text{idle-read-fM}" , "\text{idle-invert-fM}" \}
\[ \cup \text{prefixed } "\text{idle-noop}" \]
\[ \cup (\text{mark-locs} \setminus \{ "\text{mark-read-fM}" , "\text{mark-write-fA}" , "\text{mark-noop-mfence}" \}) \]
\[ \cup \text{sweep-locs} \]

\( \langle ML \rangle \)

definition \( fA\text{-neq-locs} :: \text{location set where} \)

\[ fA\text{-neq-locs} \equiv \{ "\text{idle-phase-init}" , "\text{idle-write-fM}" , "\text{mark-read-fM}" , "\text{mark-write-fA}" \}
\[ \cup \text{prefixed } "\text{idle-flip-noop}" \]

\( \langle ML \rangle \)
∪ init-locs

(ML)

definition (in gc) fM-fA-invL :: ('field, 'mut, 'ref) gc-pred where
[inv]: fM-fA-invL ≡ 
  atS-gc fM-eq-locs (sys-fM eq gc-fM)
  and at-gc "idle-write-fM" (sys-fM neq gc-fM)
  and at-gc "idle-flip-noop-mfence" (sys-fM neq gc-fM imp (not list-null (tso-pending-fM gc)))
  and atS-gc fM-tso-empty-locs (list-null (tso-pending-fM gc))
  and atS-gc fA-eq-locs (sys-fA eq gc-fM)
  and atS-gc fA-neq-locs (sys-fA neq gc-fM)
  and at-gc "mark-noop-mfence" (sys-fA neq gc-fM imp (not list-null (tso-pending-fA gc)))
  and atS-gc fA-tso-empty-locs (list-null (tso-pending-fA gc))

3.7 Object colours, reference validity, worklist validity

We adopt the classical tricolour scheme for object colours due to Dijkstra et al. (1978), but tweak it somewhat in the presence of worklists and TSO. Intuitively:

White potential garbage, not yet reached

Grey reached, presumed live, a source of possible new references (work)

Black reached, presumed live, not a source of new references

In this particular setting we use the following interpretation:

White: not marked

Grey: on a worklist

Black: marked and not on a worklist

Note that this allows the colours to overlap: an object being marked may be white (on the heap) and in ghost-honorary-grey for some process, i.e. grey.

abbreviation marked :: 'ref ⇒ ('field, 'mut, 'ref) lsts-pred where
marked r s ≡ obj-at (λobj. obj-mark obj = sys-fM s) r s

abbreviation white :: 'ref ⇒ ('field, 'mut, 'ref) lsts-pred where
white r s ≡ obj-at (λobj. obj-mark obj = (¬sys-fM s)) r s

definition WL :: 'mut process-name ⇒ ('field, 'mut, 'ref) lsts ⇒ 'ref set where
WL p ≡ λs. W (s p) ∪ ghost-honorary-grey (s p)

definition grey :: 'ref ⇒ ('field, 'mut, 'ref) lsts-pred where
grey r ≡ EXS p. (r) in WL p
We show that if a mutator can load a reference into its roots (its working set of references), then there is an object in the heap at that reference.

In this particular collector, we can think of grey references and pending TSO heap mutations as extra mutator roots; in particular the GC holds no roots itself but marks everything reachable from its worklist, and so we need to know these objects exist. By the strong tricolour invariant (§3.9), black objects point to black or grey objects, and so we do not need to treat these specially.

The worklists track the grey objects. The following invariant asserts that grey objects are marked on the heap except for a few steps near the end of mark-object-fn, the processes’ worklists and ghost-honorary-greys are disjoint, and that pending marks are sensible. The safety of the collector does not to depend on disjointness; we include it as proof that the single-threading of grey objects in the implementation is sound.
3.8 Mark Object

Local invariants for \textit{mark-object-fn}. Invoking this code in phases where \textit{sys-fM} is constant marks the reference in \textit{ref}. When \textit{sys-fM} could vary this code is not called. The two cases are distinguished by \textit{p-ph-enabled}.

Each use needs to provide extra facts to justify validity of references, etc. We do not include a post-condition for \textit{mark-object-fn} here as it is different at each call site.

\begin{verbatim}
locale mark-object =
  fixes p :: 'mut process-name
  fixes l :: location
  fixes p-ph-enabled :: ('field, 'mut, 'ref) lsts-pred
  assumes p-ph-enabled-eq-imp: eq-imp (λ(_:unit) s. s p) p-ph-enabled
begin

abbreviation (input) p-cas-mark s ≡ cas-mark (s p)
abbreviation (input) p-mark s ≡ mark (s p)
abbreviation (input) p-fM s ≡ fM (s p)
abbreviation (input) p-ghost-handshake-phase s ≡ ghost-handshake-phase (s p)
abbreviation (input) p-ghost-honorary-grey s ≡ ghost-honorary-grey (s p)
abbreviation (input) p-ghost-handshake-in-sync s ≡ ghost-handshake-in-sync (s p)
abbreviation (input) p-phase s ≡ phase (s p)
abbreviation (input) p-ref s ≡ ref (s p)
abbreviation (input) p-the-ref ≡ the ◦ p-ref
abbreviation (input) p-W s ≡ W (s p)

abbreviation at-p :: location ⇒ ('field, 'mut, 'ref) lsts-pred ⇒ ('field, 'mut, 'ref) gc-pred
where
  at-p l' P ≡ at p (l @ l') imp LSTP P
abbreviation (input) p-en-cond P ≡ p-ph-enabled imp P

abbreviation (input) p-valid-ref ≡ not null p-ref and valid-ref ▷ p-the-ref
abbreviation (input) p-tso-no-pending-mark ≡ list-null (tso-pending-mark p)
abbreviation (input) p-tso-no-pending-mutate ≡ list-null (tso-pending-mutate p)

abbreviation (input)
  p-valid-W-inv ≡ ((p-cas-mark neq p-mark or p-tso-no-pending-mark) imp marked ▷ p-the-ref)
  and (tso-pending-mark p in (λs. {[]} [mw-Mark (p-the-ref s) (p-fM s)]))

abbreviation (input)
  p-mark-inv ≡ not null p-mark
  and ((λs. obj-at (λobj. Some (obj-mark obj) = p-mark s) (p-the-ref s) s)
       or marked ▷ p-the-ref)

abbreviation (input)
  p-cas-mark-inv ≡ (λs. obj-at (λobj. Some (obj-mark obj) = p-cas-mark s) (p-the-ref s) s)
\end{verbatim}
abbreviation (input) \( p\text{-valid-fM} \equiv p\text{-FM }\equiv \text{sys-fM} \)

abbreviation (input)
\( p\text{-ghg-eq-ref} \equiv p\text{-ghost-honorary-grey }\equiv \text{pred-singleton }\circ\ p\text{-ref} \)

abbreviation (input)
\( p\text{-ghg-inv} \equiv \text{If } p\text{-cas-mark }\equiv p\text{-mark }\text{Then } p\text{-ghg-eq-ref }\text{Else empty } p\text{-ghost-honorary-grey} \)

definition \( \text{mark-object-invL }:: \left( ^{\prime}\text{field}, ^{\prime}\text{mut}, ^{\prime}\text{ref}\right) \) gc-pred where
\( \text{mark-object-invL }\equiv \)
\( \text{at-p }^{\prime}\text{-mo-null} \) \( \equiv (\text{True}) \)
\( \text{and at-p }^{\prime}\text{-mo-mark} \) \( \equiv (p\text{-valid-ref}) \)
\( \text{and at-p }^{\prime}\text{-mo-fM} \) \( \equiv (p\text{-valid-ref }\) and \( p\text{-en-cond }\) (p-mark-inv))
\( \text{and at-p }^{\prime}\text{-mo-mtest} \) \( \equiv (p\text{-valid-ref }\) and \( p\text{-en-cond }\) (p-mark-inv and p-valid-fM))
\( \text{and at-p }^{\prime}\text{-mo-phase} \) \( \equiv (p\text{-valid-ref }\) and \( p\text{-mark }\) neq Some \( \circ\ p\text{-FM} \) and \( p\text{-en-cond} \)
\( (p\text{-mark-inv and p-valid-fM}) \)
\( \text{and at-p }^{\prime}\text{-mo-ptest} \) \( \equiv (p\text{-valid-ref }\) and \( p\text{-mark }\) neq Some \( \circ\ p\text{-FM} \) and \( p\text{-en-cond} \)
\( (p\text{-mark-inv and p-valid-fM}) \)
\( \text{and at-p }^{\prime}\text{-mo-co-lock} \) \( \equiv (p\text{-valid-ref }\) and \( p\text{-mark }\) neq Some \( \circ\ p\text{-FM} \) and \( p\text{-mark }\) neq Some \( \circ\ p\text{-FM} \) and \( p\text{-mark }\) neq Some \( )\).
\( \text{and at-p }^{\prime}\text{-mo-co-cmark} \) \( \equiv (p\text{-valid-ref }\) and \( p\text{-mark }\) neq Some \( \circ\ p\text{-FM} \) and \( p\text{-mark }\) neq Some \( )\).
\( \text{and at-p }^{\prime}\text{-mo-co-ctest} \) \( \equiv (p\text{-valid-ref }\) and \( p\text{-mark }\) neq Some \( \circ\ p\text{-FM} \) and \( p\text{-mark }\) neq Some \( )\).
\( \text{and at-p }^{\prime}\text{-mo-co-mark} \) \( \equiv (p\text{-cas-mark }\) eq \( p\text{-mark }\) and \( p\text{-valid-ref }\) and \( p\text{-valid-fM }\) and white\( \triangleright p\text{-the-ref }\) and \( p\text{-tso-no-pending-mark} \)
\( \text{and at-p }^{\prime}\text{-mo-co-unlock} \) \( \equiv (p\text{-ghg-inv }\) and \( p\text{-valid-ref }\) and \( p\text{-valid-fM }\) and \( p\text{-valid-W-inv} \)
\( \text{and at-p }^{\prime}\text{-mo-co-won} \) \( \equiv (p\text{-ghg-inv }\) and \( p\text{-valid-ref }\) and \( p\text{-valid-fM }\) and \( p\text{-mark }\) neq Some \( \circ\ p\text{-FM} \) and \( p\text{-mark }\) neq Some \( \circ\ p\text{-FM} \) and \( p\text{-mark }\) neq Some \( )\).
\( \text{and at-p }^{\prime}\text{-mo-co-won} \) \( \equiv (p\text{-ghg-eq-ref }\) and \( p\text{-valid-ref }\) and \( p\text{-valid-fM }\) and \( p\text{-mark }\) neq Some \( )\).
\( \langle \text{proof} \rangle \)
end

The uses of \( \text{mark-object-fn }\) in the GC and during the root marking are straightforward.

interpretation \( \text{gc-mark}!: \text{mark-object gc }"\text{mark-loop}" (True) \langle \text{proof} \rangle \)
lemmas \( \text{gc-mark-mark-object-invL-def2[inv]} = \text{gc-mark.mark-object-invL-def[simplified]} \)

interpretation \( \text{mut-get-roots}!: \text{mark-object mutator m }"\text{hs-get-roots-loop}" (True) \text{ for m} \langle \text{proof} \rangle \)
lemmas \( \text{mut-get-roots-mark-object-invL-def2[inv]} = \text{mut-get-roots.mark-object-invL-def[simplified]} \)

The most interesting cases are the two asynchronous uses of \( \text{mark-object-fn }\) in the mutators: we need something that holds even before we read the phase. In particular we need to avoid interference by an \( fM \) flip.

interpretation \( \text{mut-store-del}!: \text{mark-object mutator m }"\text{store-del}" \text{mut-m.mut-ghost-handshake-phase m neq }\langle \text{hp-Idle} \rangle \text{ for m} \langle \text{proof} \rangle \)
lemmas \text{mut-store-del-mark-object-invL-def2}[\text{inv}] = \text{mut-store-del.mark-object-invL-def}[\text{simplified}]

interpretation \text{mut-store-ins!} : \text{mark-object mutator m "store-ins" mut-m.mut-ghost-handshake-phase m neq \langle hp-Idle \rangle} \text{ for m \langle proof \rangle}

lemmas \text{mut-store-ins-mark-object-invL-def2}[\text{inv}] = \text{mut-store-ins.mark-object-invL-def}[\text{simplified}]

Local invariant for the mutator’s uses of \text{mark-object}.

\text{definition} \text{mut-hs-get-roots-loop-locs} \equiv
   \text{prefixed "hs-get-roots-loop" (ML)}

\text{definition} \text{mut-hs-get-roots-loop-mo-locs} \equiv
   \text{prefixed "hs-get-roots-loop-mo" \cup \{"hs-get-roots-loop-done"\} (ML)}

\text{abbreviation} \text{mut-async-mark-object-prefixes} \equiv \{"store-del", "store-ins"\}

\text{definition} \text{mut-hs-not-hp-Idle-locs} \equiv
   \bigcup \text{pref} \in \text{mut-async-mark-object-prefixes}.
   \bigcup l \in \{"mo-co-lock", "mo-co-cmark", "mo-co-ctest", "mo-co-mark", "mo-co-unlock", "mo-co-won", "mo-co-W"\}. \{\text{pref} @ "." @ l\} \text{(ML)}

\text{definition} \text{mut-async-mo-ptest-locs} \equiv
   \bigcup \text{pref} \in \text{mut-async-mark-object-prefixes}. \{\text{pref} @ ".-mo-ptest"\} \text{(ML)}

\text{definition} \text{mut-mo-ptest-locs} \equiv
   \bigcup \text{pref} \in \text{mut-async-mark-object-prefixes}. \{\text{pref} @ ".-mo-ptest"\} \text{(ML)}

\text{definition} \text{mut-mo-valid-ref-locs} \equiv
   \text{prefixed "store-del" \cup \text{prefixed "store-ins" \cup \{"deref-del", "lop-store-ins"\}}} \text{(ML)\langle proof \rangle\langle proof \rangle}

This local invariant for the mutators illustrates the handshake structure: we can rely on the insertion barrier earlier than on the deletion barrier. Both need to be installed before \text{get-roots} to ensure we preserve the strong tricolour invariant. All black objects at that point are allocated: we need to know that the insertion barrier is installed to preserve it. This limits when \text{fA} can be set.

It is interesting to contrast the two barriers. Intuitively a mutator can locally guarantee that it, in the relevant phases, will insert only marked references. Less often can it be sure that the reference it is overwriting is marked. We also need to consider writes pending in TSO buffers.

\text{definition} \text{ghost-honorary-grey-empty-locs :: location set where}
   \text{ghost-honorary-grey-empty-locs} \equiv
   \bigcup \langle\bigcup \{"mark-loop", "hs-get-roots-loop", "store-del", "store-ins"\}\rangle.
\[ \bigcup l \in \{ "mo-co-unlock", "mo-co-won", "mo-co-W" \}. \{ \text{pref } @ "." @ l \} \]

\[(ML)\]

**definition (in mut-m)** mark-object-invL :: ('field, 'mut, 'ref) gc-pred where

[inv]: mark-object-invL \equiv

\[\begin{align*}
\text{atS-mut} \ & \text{mut-hs-get-roots-loop-locs} \\
& \quad \text{(mut-refs subseteq mut-roots and (ALLS r. \langle r \rangle in mut-roots diff mut-refs imp marked r))}
\end{align*}\]

and \(\text{atS-mut} \ \text{mut-hs-get-roots-loop-mo-loc-code} \)

\[\begin{align*}
& \quad \text{(not null mut-ref and mut-the-ref in mut-roots)} \\
& \quad \text{and at-mut "hs-get-roots-loop-done"} \\
& \quad \text{(marked \( \triangleright \) mut-the-ref)} \\
\end{align*}\]

and \(\text{at-mut "hs-get-roots-loop-mo-ptest" (mut-phase neq \langle ph-Idle \rangle)}\)

and \(\text{at-mut "hs-get-roots-done" (ALLS r. \langle r \rangle in mut-roots imp marked r)}\)

and \(\text{atS-mut} \ \text{mut-mo-valid-ref-locs} \)

\[\begin{align*}
& \quad \text{(not null mut-new-ref imp marked in mut-roots)} \\
& \quad \text{and (mut-temp-ref in mut-roots)} \\
& \quad \text{and at-mut "store-del-mo-null" (not null mut-ref imp mut-the-ref in mut-ghost-honorary-root)} \\
& \quad \text{and atS-mut (prefixed "store-del" - \{"store-del-mo-null"\}) (mut-the-ref in mut-ghost-honorary-root)} \\
& \quad \text{and atS-mut (prefixed "store-ins") (mut-temp-ref eq mut-new-ref)} \\
& \quad \text{and atS-mut } \\text{(suffix "mo-ptest") (mut-phase eq \langle ph-Idle \rangle) imp mut-ghost-handshake-phase neq \langle hp-Idle \rangle)} \\
& \quad \text{and atS-mut } \text{mut-not-hp-Idle-loc-code} \\
& \quad \text{(mut-ghost-handshake-phase neq \langle hp-Idle \rangle)}
\end{align*}\]

\[\begin{align*}
& \quad \text{and atS-mut \text{mut-mo-ptest-loc-code} in } \langle \{ hp-Idle, hp-IdleInit \} \rangle \\
& \quad \text{or (mut-ghost-handshake-phase eq \langle hp-IdleMarkSweep \rangle \text{ and sys-phase eq } \langle ph-Idle \rangle) } \\
\end{align*}\]

\[\begin{align*}
& \quad \text{and atS-mut } \text{ghost-honorary-grey-empty-loc-code} \text{ (empty mut-ghost-honorary-grey)}
\end{align*}\]

(* insertion barrier *)

and at-mut "store-ins" (mut-ghost-handshake-phase in \{hp-InitMark, hp-Mark\})

\[\begin{align*}
& \quad \text{or (mut-ghost-handshake-phase eq \langle hp-IdleMarkSweep \rangle \text{ and sys-phase eq } \langle ph-Idle \rangle) )} \\
& \quad \text{and not null mut-new-ref} \\
& \quad \text{imp marked \( \triangleright \) mut-the-new-ref} \\
\end{align*}\]

(* deletion barrier *)

and atS-mut (prefixed "store-del-mo" ) ("lop-store-ins")

\[\begin{align*}
& \quad \text{(mut-ghost-handshake-phase eq \langle hp-Mark \rangle \text{ or (mut-ghost-handshake-phase eq } \langle hp-IdleMarkSweep \rangle)} \\
& \quad \text{and sys-phase eq \langle ph-Idle \rangle) } \\
\end{align*}\]

\[\begin{align*}
& \quad \text{and } (\lambda s. \forall \text{ opt-r' } \neg \text{tso-pending-write } \text{(mutator m) (mw-Mutate} \\
& \quad (\text{mut-temp-ref s) (mut-field s) opt-r'} s) \\
& \quad \text{imp (\lambda s. obj-at-field-on-heap (\lambda r. \text{mut-ref s = Some r } \triangleright} \\
& \quad \text{marked r s) (mut-temp-ref s) (mut-field s) s))}
\end{align*}\]
and at-mut "lop-store-ins" (mut-ghost-handshake-phase eq ⟨hp-Mark⟩ or (mut-ghost-handshake-phase eq ⟨hp-IdleMarkSweep⟩) and not null mut-ref
imp marked ⊢ mut-the-ref )
and atS-mut (prefixed "store-ins") (mut-ghost-handshake-phase eq ⟨hp-Mark⟩ or (mut-ghost-handshake-phase eq ⟨hp-IdleMarkSweep⟩) and sys-phase neq ⟨ph-Idle⟩)
and sys-phase neq ⟨ph-Idle⟩)

The GC’s use of mark-object-fn is correct. When we take grey tmp-ref to black, all of the objects it points to are marked, ergo the new black does not point to white, and so we preserve the strong tricolour invariant.

definition (in gc) obj-fields-marked-inv :: ('field, 'mut, 'ref) lsts-pred where
obj-fields-marked-inv ≡
ALLS f. (f) in (− gc-field-set) imp (λs. obj-at-field-on-heap (λr. marked r s) (gc-tmp-ref s) f s))
proof

definition obj-fields-marked-locs :: location set where
obj-fields-marked-locs ≡
{ "mark-loop-mark-object-loop", "mark-loop-mark-choose-field", "mark-loop-mark-deref", "mark-loop-mark-field-done", "mark-loop-blacken" }
∪ prefixed "mark-loop-mo"
⟨ML⟩

definition (in gc) obj-fields-marked-invL :: ('field, 'mut, 'ref) gc-pred where
[inv]: obj-fields-marked-invL ≡
atS-gc obj-fields-marked-locs (obj-fields-marked-inv and gc-tmp-ref in gc-W)
and atS-gc (prefixed "mark-loop-mo" ∪ { "mark-loop-mark-field-done" }) (λs. obj-at-field-on-heap (λr. gc-ref s = Some r ∨ marked r s) gc-tmp-ref s) gc-field s)
and atS-gc (prefixed "mark-loop-mo") (ALLS y. not null gc-ref and (λs. ((gc-the-ref s) reaches y) s) imp valid-ref y)
and at-gc "mark-loop-fields" (gc-temp-ref in gc-W)
and at-gc "mark-loop-mark-field-done" (not null gc-ref imp marked ⊢ gc-the-ref)
and at-gc "mark-loop-blacken" (empty gc-field-set)
and atS-gc ghost-honorary-grey-empty-locs (empty gc-ghost-honorary-grey) proof

3.9 The strong-tricolour invariant

As the GC algorithm uses both insertion and deletion barriers, it preserves the strong tricolour-invariant:
abbreviation points-to-white :: 'ref ⇒ 'ref ⇒ ('field, 'mut, 'ref) lsts-pred (infix points'-to'-white 51) where  
x points-to-white y ≡ x points-to y and white y

definition strong-tricolour-inv :: ('field, 'mut, 'ref) lsts-pred where  
strong-tricolour-inv ≡ ALLS b w. black b imp not b points-to-white w

Intuitively this invariant says that there are no pointers from completely processed objects to the unexplored space; i.e., the grey references properly separate the two. In contrast the weak tricolour invariant allows such pointers, provided there is a grey reference that protects the unexplored object.

definition has-white-path-to :: 'ref ⇒ 'ref ⇒ ('field, 'mut, 'ref) lsts-pred (infix has'-white'-path'-to 51) where  
x has-white-path-to y ≡ λs. (λx y. (x points-to-white y) s)** x y

definition grey-protects-white :: 'ref ⇒ 'ref ⇒ ('field, 'mut, 'ref) lsts-pred (infix grey'-protects'-white 51) where  
g grey-protects-white w ≡ grey g and g has-white-path-to w

definition weak-tricolour-inv :: ('field, 'mut, 'ref) lsts-pred where  
weak-tricolour-inv ≡  
ALLS b w. black b and b points-to-white w imp (EXS g. g grey-protects-white w)

lemma strong-tricolour-inv s =⇒ weak-tricolour-inv s (proof)
The key invariant that the mutators establish as they perform get-roots: they protect their white-reachable references with grey objects.

definition in-snapshot :: 'ref ⇒ ('field, 'mut, 'ref) lsts-pred where  
in-snapshot r ≡ black r or (EXS g. g grey-protects-white r)

definition (in mut-m) reachable-snapshot-inv :: ('field, 'mut, 'ref) lsts-pred where  
reachable-snapshot-inv ≡ ALLS r. reachable r imp in-snapshot r

Note that it is not easy to specify precisely when the snapshot (of objects the GC will retain) is taken due to the raggedness of the initialisation.

In some phases we need to know that the insertion and deletion barriers are installed, in order to preserve the snapshot. These can ignore TSO effects as marks hit system memory in a timely way.

abbreviation marked-insertion :: ('field, 'ref) mem-write-action ⇒ ('field, 'mut, 'ref) lsts-pred where  
marked-insertion w ≡ λs. case w of mw-Mutate r f (Some r') ⇒ marked r' s | - ⇒ True

definition (in mut-m) marked-insertions :: ('field, 'mut, 'ref) lsts-pred where  
marked-insertions ≡ ALLS w. tso-pending-write (mutator m) w imp marked-insertion w

abbreviation marked-deletion :: ('field, 'ref) mem-write-action ⇒ ('field, 'mut, 'ref) lsts-pred where
marked-deletion \( w \equiv \lambda s. \text{case } w \text{ of } \text{mw-Mutate } f \text{ opt-}r' \Rightarrow \text{obj-at-field-on-heap } (\lambda r'. \text{marked } r' \text{ s}) \ r \ s \ | \ - \Rightarrow \text{True} \)

**definition** (in \( \text{mut-m} \)) marked-deletions :: (\text{\}'field, 'mut, 'ref\') lsts-pred where
marked-deletions \( \equiv \text{ALLS } w. \text{tso-pending-write } (\text{mutator } m) \ w \text{ imp marked-deletion } w \)

Finally, in some phases the heap is somewhat monochrome.

**definition** black-heap :: (\text{\}'field, 'mut, 'ref\') lsts-pred where
black-heap \( \equiv \text{ALLS } r. \text{black } r \text{ or not valid-ref } r \)

**definition** white-heap :: (\text{\}'field, 'mut, 'ref\') lsts-pred where
white-heap \( \equiv \text{ALLS } r. \text{white } r \text{ or not valid-ref } r \)

**definition** no-black-refs :: (\text{\}'field, 'mut, 'ref\') lsts-pred where
no-black-refs \( \equiv \text{ALLS } r. \text{not black } r \)

**definition** no-grey-refs :: (\text{\}'field, 'mut, 'ref\') lsts-pred where
no-grey-refs \( \equiv \text{ALLS } r. \text{not grey } r(\text{\textlangle proof\textrangle}\langle proof\textrangle\langle proof\textrangle\langle proof\textrangle\langle proof\textrangle\langle proof\textrangle\langle proof\textrangle\langle proof\textrangle\langle proof\textrangle\langle proof\textrangle\langle proof\textrangle\langle proof\textrangle\langle proof\textrangle) \)

### 3.10 Invariants

We need phase invariants in terms of both mut-ghost-handshake-phase and sys-ghost-handshake-phase which respectively track what the mutators and GC know by virtue of the synchronisation structure of the system.

Read the following as “when mutator \( m \) is past the specified handshake, and has yet to reach the next one, ... holds.”

**primrec** (in \( \text{mut-m} \)) mutator-phase-inv-aux :: handshake-phase \( \Rightarrow (\text{\}'field, 'mut, 'ref\') lsts-pred \)
where
mutator-phase-inv-aux \( hp-\text{Idle} \) \( \equiv (\text{True}) \)
| mutator-phase-inv-aux \( hp-\text{IdleInit} \) \( \equiv \text{no-black-refs} \)
| mutator-phase-inv-aux \( hp-\text{InitMark} \) \( \equiv \text{marked-insertions} \)
| mutator-phase-inv-aux \( hp-\text{Mark} \) \( \equiv (\text{marked-insertions and marked-deletions}) \)
| mutator-phase-inv-aux \( hp-\text{IdleMarkSweep} \) \( \equiv (\text{marked-insertions and marked-deletions and reachable-snapshot-inv}) \)

**abbreviation** (in \( \text{mut-m} \)) mutator-phase-inv :: (\text{\}'field, 'mut, 'ref\') lsts-pred where
mutator-phase-inv \( s \equiv \text{mutator-phase-inv-aux } (\text{mut-ghost-handshake-phase } s) \ s \)

**abbreviation** mutators-phase-inv :: (\text{\}'field, 'mut, 'ref\') lsts-pred where
mutators-phase-inv \( \equiv \text{ALLS } m. \text{mut-m.mutator-phase-inv } m \)

This is what the GC guarantees. Read this as “when the GC is at or past the specified handshake, ... holds.”

**primrec** sys-phase-inv-aux :: handshake-phase \( \Rightarrow (\text{\}'field, 'mut, 'ref\') lsts-pred \)
where
sys-phase-inv-aux \( hp-\text{Idle} \) \( \equiv (\text{If } \text{sys-fA eq sys-fM } \text{Then black-heap Else white-heap}) \)
and no-grey-refs \)

39
| sys-phase-inv-aux hp-IdleInit = no-black-refs |
| sys-phase-inv-aux hp-InitMark = (sys-fA neq sys-fM imp no-black-refs) |
| sys-phase-inv-aux hp-Mark = ⟨True⟩ |
| sys-phase-inv-aux hp-IdleMarkSweep = ( (sys-phase eq ⟨ph-Idle⟩ or tso-pending-write gc (mw-Phase ph-Idle)) imp no-grey-refs ) |

abbreviation sys-phase-inv :: ('field, 'mut, 'ref) lsts-pred where
sys-phase-inv s ≡ sys-phase-inv-aux (sys-ghost-handshake-phase s) s⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩

3.11 Lonely mutator assertions

The second assertion is key: after the "init-noop" handshake, we need to know that there are no pending white insertions (mutations that insert unmarked references) for the deletion barrier to work.

definition ghost-honorary-root-empty-locs :: location set where
ghost-honorary-root-empty-locs ≡
− (prefixed "store-del" ∪ {"lop-store-ins"} ∪ prefixed "store-ins")

⟨ML⟩

definition (in mut-m) load-invL :: ('field, 'mut, 'ref) gc-pred where
[inv]: load-invL ≡
at-mut "load" (mut-trm-ref in mut-roots)
and at-mut "hs-noop-done" (list-null (tso-pending-mutate (mutator m)))
and atS-mut ghost-honorary-root-empty-locs (empty mut-ghost-honorary-root)⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩

3.12 The infamous termination argument.

We need to know that if the GC does not receive any further work to do at get-roots and get-work, then there are no grey objects left. Essentially this encodes the stability property that grey objects must exist for mutators to create grey objects.

Note that this is not invariant across the scan: it is possible for the GC to hold all the grey references. The two handshakes transform the GC's local knowledge that it has no more work to do into a global property, or gives it more work.

definition (in mut-m) gc-W-empty-mut-inv :: ('field, 'mut, 'ref) lsts-pred where
gc-W-empty-mut-inv ≡
(empty sys-W and sys-ghost-handshake-in-sync m and not empty (WL (mutator m)))
imp (EXS m'. not (sys-ghost-handshake-in-sync m') and not empty (WL (mutator m')))

definition (in −) gc-W-empty-locs :: location set where
gc-W-empty-locs ≡
idle-locs ∪ init-locs ∪ sweep-locs ∪ { "mark-read-fM", "mark-write-fA", "mark-end" }
∪ prefixed "mark-noop"
∪ prefixed "mark-loop-get-roots"
∪ prefixed "mark-loop-get-work"
⟨ML⟩
4.1 Invariants

definition (in gc) invsL :: (‘field, ‘mut, ‘ref) gc-pred where
invsL ≡
  fM-fA-invL
and gc-mark.mark-object-invL
and gc-W-empty-invL
and handshake-invL
and obj-fields-marked-invL
and phase-invL
and sweep-loop-invL

3.13 Sweep loop invariants

definition sweep-loop-locs ≡ prefixed "sweep-loop"
(ML)

4 Top-level safety

4.1 Invariants

definition (in gc) invsL :: (‘field, ‘mut, ‘ref) gc-pred where
invsL ≡
  fM-fA-invL
and gc-mark.mark-object-invL
and gc-W-empty-invL
and handshake-invL
and obj-fields-marked-invL
and phase-invL
and sweep-loop-invL
and tso-lock-invL
and LSTP (fA-rel-inv and fM-rel-inv)

**definition** (in mut-m) invsL :: ('field, 'mut, 'ref) gc-pred where
invsL ≡
  load-invL
and mark-object-invL
and mut-get-roots.mark-object-invL m
and mut-store-ins.mark-object-invL m
and mut-store-del.mark-object-invL m
and handshake-invL
and tso-lock-invL
and LSTP mutator-phase-inv

**definition** invs :: ('field, 'mut, 'ref) lsts-pred where
invs ≡
  handshake-phase-inv
and phase-rel-inv
and strong-tricolour-inv
and sys-phase-inv
and tso-writes-inv
and valid-refs-inv
and valid-W-inv

**definition** I :: ('field, 'mut, 'ref) gc-pred where
I ≡
  gc.invsL
and (ALLS m. mut-m.invsL m)
and LSTP invs⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩

### 4.2 Initial conditions

We ask that the GC and system initially agree on some things:

- All objects on the heap are marked (have their flags equal to sys-fM, and there are no grey references, i.e. the heap is uniformly black.
- The GC and system have the same values for fA, fM, etc. and the phase is Idle.
- No process holds the TSO lock and all write buffers are empty.
- All root-reachable references are backed by objects.

Note that these are merely sufficient initial conditions and can be weakened.

locale gc-system =
  fixes initial-mark :: gc-mark
begin
definition gc-initial-state :: (field, mut, ref) lst-pred where
gc-initial-state \ s \equiv
  \ fM \ s = \text{initial-mark}
  \land \ phase \ s = \text{ph-Idle}
  \land \ ghost-honorary-grey \ s = \{\}
  \land \ W \ s = \{\}

definition mut-initial-state :: (field, mut, ref) lst-pred where
mut-initial-state \ s \equiv
  ghost-handshake-phase \ s = \text{hp-IdleMarkSweep}
  \land \ ghost-honorary-grey \ s = \{\}
  \land \ ghost-honorary-root \ s = \{\}
  \land \ W \ s = \{\}

definition sys-initial-state :: (field, mut, ref) lst-pred where
sys-initial-state \ s \equiv
  (\forall \ m. \neg \text{handshake-pending} \ s \ m \land \text{ghost-handshake-in-sync} \ s \ m)
  \land \ ghost-handshake-phase \ s = \text{hp-IdleMarkSweep} \land \text{handshake-type} \ s = \text{ht-GetRoots}
  \land \ obj-mark \ ' \ ran \ (heap \ s) \subseteq \{\text{initial-mark}\}
  \land \ fA \ s = \text{initial-mark}
  \land \ fM \ s = \text{initial-mark}
  \land \ phase \ s = \text{ph-Idle}
  \land \ ghost-honorary-grey \ s = \{\}
  \land \ W \ s = \{\}
  \land (\forall \ p. \text{mem-write-buffers} \ s \ p = [])
  \land \text{mem-lock} \ s = \text{None}

abbreviation
root-reachable \ y \equiv \ EXS \ m \ x. (x) \ in \ mut-m.mut-roots \ m \ and \ x \ reaches \ y

definition valid-refs :: (field, mut, ref) lsts-pred where
valid-refs \equiv \ ALLS \ y. \ root-reachable \ y \implies \ valid-ref \ y

definition gc-system-init :: (field, mut, ref) lsts-pred where
\ gc-system-init \equiv
  (\lambda s. \ gc-initial-state \ (s gc))
  \land (\lambda s. \forall \ m. \ mut-initial-state \ (s (mutator \ m)))
  \land (\lambda s. \ sys-initial-state \ (s sys))
  \land \ valid-refs

The system consists of the programs and these constraints on the initial state.

abbreviation gc-system :: (field, mut, ref) gc-system where
gc-system \equiv \ gc-pgms, gc-system-init)

theorem inv: \ s \in \ reachable-states \ gc-system \implies \ I \ (mkP \ s)

Our headline safety result follows directly.
corollary safety:
s ∈ reachable-states gc-system \implies valid-refs (mkP s) \downarrow \langle proof \rangle
end
The GC is correct for the remaining fixed-but-arbitrary initial conditions.

interpretation gc-system-interpretation! : gc-system undefined \langle proof \rangle

4.3 A concrete system state

We demonstrate that our definitions are not vacuous by exhibiting a concrete initial state that satisfies the initial conditions. We use Isabelle’s notation for types of a given size.

theory Concrete-heap
imports
  ~/src/HOL/Library/Saturated
  ../Proofs
begin

  type-synonym field = 3
  type-synonym mut = 2
  type-synonym ref = 5

  type-synonym concrete-local-state = (field, mut, ref) local-state
  type-synonym clsts = (field, mut, ref) lsts

  abbreviation mut-common-init-state :: concrete-local-state where

  context gc-system
begin

  abbreviation sys-init-heap :: ref ⇒ (field, ref) object option where
    sys-init-heap ≡ 
      [ 0 ⇒ (| obj-mark = initial-mark, 
          obj-fields = [ 0 \mapsto 5 ] |),
       1 ⇒ (| obj-mark = initial-mark, 
          obj-fields = Map.empty |),
       2 ⇒ (| obj-mark = initial-mark, 
          obj-fields = Map.empty |),
       3 ⇒ (| obj-mark = initial-mark, 
          obj-fields = [ 0 \mapsto 1, 1 \mapsto 2 ] |),
       4 ⇒ (| obj-mark = initial-mark, 
          obj-fields = [ 1 \mapsto 0 ] |),
       5 ⇒ (| obj-mark = initial-mark, 
          obj-fields = Map.empty |) ]
abbreviation mut-init-state0 :: concrete-local-state where
mut-init-state0 ≡ mut-common-init-state (| roots := \{1, 2, 3\} )

abbreviation mut-init-state1 :: concrete-local-state where
mut-init-state1 ≡ mut-common-init-state (| roots := \{3\} )

abbreviation mut-init-state2 :: concrete-local-state where
mut-init-state2 ≡ mut-common-init-state (| roots := \{2, 5\} )

end

end
c

c

context gc-system

begin

abbreviation sys-init-state :: concrete-local-state where
sys-init-state ≡ undefined (| fA := initial-mark,
                          fM := initial-mark,
                          heap := sys-init-heap,
                          handshake-pending := \{False\},
                          handshake-type := ht-GetRoots,
                          mem-lock := None,
                          mem-write-buffers := \{\[]\},
                          phase := ph-Idle,
                          W := \{\},
                          ghost-honorary-grey := \{\},
                          ghost-handshake-in-sync := \{True\},
                          ghost-handshake-phase := hp-IdleMarkSweep )

abbreviation gc-init-state :: concrete-local-state where
gc-init-state ≡ undefined (| fM := initial-mark,
                           fA := initial-mark,
                           phase := ph-Idle,
                           W := \{\},
                           ghost-honorary-grey := \{\} )

primrec lookup :: (’k × ’v) list ⇒ ’v ⇒ ’k ⇒ ’v where
lookup [] v0 k = v0
| lookup (kv # kvs) v0 k = (if fst kv = k then snd kv else lookup kvs v0 k)

abbreviation muts-init-states :: (mut × concrete-local-state) list where
muts-init-states ≡ [ (0, mut-init-state0), (1, mut-init-state1), (2, mut-init-state2) ]
abbreviation init-state :: clsts where
init-state ≡ λp. case p of
gc ⇒ gc-init-state
| sys ⇒ sys-init-state
| mutator m ⇒ lookup muts-init-states mut-common-init-state m

lemma
gc-system-init init-state⟨proof⟩
end

References


F. Pizlo. Fragmentation Tolerant Real Time Garbage Collection. PhD thesis, Purdue University, 201x.

