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

Peter Gammie, Tony Hosking and Kai Engelhardt

April 18, 2020

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 1
2 The Schism garbage collector 2
2.1 Object marking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2 Handshakes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.3 The system process . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.4 Mutators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.5 Garbage collector . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3 Invariants and Proofs 15
3.1 Constructors for sets of locations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.2 Hoare triples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.3 Functions and predicates . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.4 Garbage collector locations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.5 Coarse TSO invariants . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.5.1 Locations where the TSO lock is held . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.6 Handshake phases . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.7 Object colours, reference validity, worklist validity . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.8 Mark Object . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.9 The strong-tricolour invariant . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.10 Invariants . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.11 Mutator proofs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.12 The infamous termination argument. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.13 Sweep loop invariants . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
4 Top-level safety 43
4.1 Invariants . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.2 Initial conditions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.3 A concrete system state . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
References 46

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-TSO 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.

```plaintext
type-synonym location = char list
datatype 'mut process-name = mutator 'mut | gc | sys
```

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

```plaintext
datatype gc-phase
    = ph-Idle
    | ph-Init
    | ph-Mark
    | ph-Sweep
```

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.

```plaintext
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.

```plaintext
datatype handshake-phase
    = hp-Idle — done 1 noop
    | hp-IdleInit
    | hp-InitMark
    | hp-Mark — done 4 noops
    | hp-IdleMarkSweep — done get roots
```

```plaintext
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
```
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
  | mw-fA gc-mark
  | mw-fM gc-mark
  | mw-Phase gc-phase

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 fA, fM, phase and heap variables are subject to the TSO memory model, as are all heap operations.

record (field, mut, ref) local-state =
  — System-specific fields
  heap :: ref ⇒ (field, ref) object option
  — TSO memory state
  mem-write-buffers :: mut process-name ⇒ (field, ref) mem-write-action list
  mem-lock :: mut process-name option
  — The state of the handshakes
  handshake-type :: handshake-type
  handshake-pending :: mut ⇒ bool
  — Ghost state
  ghost-handshake-in-sync :: mut ⇒ bool
  ghost-handshake-phase :: handshake-phase

  — Mutator-specific temporaries
  new-ref :: ref option
  roots :: ref set
  ghost-honorary-root :: ref set

  — Garbage collector-specific temporaries
  field-set :: field set
  mut :: mut
  muts :: mut set

  — Local variables used by multiple processes
  fA :: gc-mark
  fM :: gc-mark
  cas-mark :: gc-mark option
  field :: field
  mark :: gc-mark option
  phase :: gc-phase
  tmp-ref :: ref
  ref :: ref option
refs :: 'ref set
W :: 'ref set
— Ghost state
ghost-honorary-grey :: 'ref set

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

datatype ('field, 'ref) mem-read-action
  = mr-Ref 'ref 'field
  | mr-Mark 'ref
  | mr-Phase
  | mr-fM
  | 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
  | 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) com

type-synonym ('field, 'mut, 'ref) gc-loc-comp
  = (('field, 'ref) response, location, ('field, 'mut, 'ref) request, ('field, 'mut, 'ref) local-state) loc-comp

type-synonym ('field, 'mut, 'ref) gc-pred-state
  = (('field, 'ref) response, location, 'mut process-name, ('field, 'mut, 'ref) request, ('field, 'mut, 'ref) local-state) pred-state
type-synonym ('field, 'mut, 'ref) gc-pred
  = (('field, 'ref) response, location, 'mut process-name, ('field, 'mut, 'ref) request, ('field, 'mut, 'ref) local-state) gc-event

locale mut-m

define an extra locale that contains these proofs.

If we have more than one mutator then we need to show that mutators do not mutually interfere. To that end we
scoping somewhat, which is a mixed blessing.

by prefixing their names the locale name. This might be considered an abuse. The attributes depend on locale
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.

locale mut-m = fixes m :: 'mut
locale mut-m' = mut-m + fixes m' :: 'mut assumes mm'[iff]: m \neq m'
locale gc
locale sys

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.

countext
  fixes p :: 'mut process-name
begin

abbreviation lock :: location \Rightarrow ('field, 'mut, 'ref) gc-com where
  lock l \equiv \{l\} Request (\lambda s. (p, ro-Lock)) (\lambda- s. \{s\})

notation lock (\{l\} lock)

abbreviation unlock :: location \Rightarrow ('field, 'mut, 'ref) gc-com where
  unlock l \equiv \{l\} Request (\lambda s. (p, ro-Unlock)) (\lambda- s. \{s\})

notation unlock (\{l\} unlock)

abbreviation read-mark :: location \Rightarrow ('field, 'mut, 'ref) local-state \Rightarrow 'ref
  ⇒ ((gc-mark option ⇒ gc-mark option)

notation read-mark ('field, 'mut, 'ref) gc-com where
  read-mark l r upd \equiv \{l\} Request (\lambda s. (p, ReadMark (r s))) (\lambda s. \{ upd (m) s | m. mv = mv-Mark m \})
notation read-mark (\{l\} read'-mark)
The garbage collector needs to synchronize with the mutators. In practice this is implemented with some thread
handshakes.

We describe the mechanism next.

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 synchronize with the mutators. In practice this is implemented with some thread
synchronization 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> :: handshakes-type ⇒ handshakes-phase ⇒ handshakes-phase` **where**

`<hp-step> <ht> ≡`

`case <ht> of`

`  <ht>-NOOP ⇒ handshakes-step`

`  <ht>-GetRoots ⇒ handshakes-step`

`  <ht>-GetWork ⇒ id`

**context** `sys`

**begin**

**definition** `handshakes :: ('field, 'mut, 'ref) gc-com where`

`handshakes ≡`

`{ "sys-hs-gc-set-type" } Response`

`(λ s . { (s){ handshakes-type ::= <ht>,
          handshakes-in-sync ::= (False),
          handshakes-phase ::= <hp-step> <ht> (handshakes-phase s) },
          mv-Void)
           |ht. req = (gc, ro-hs-gc-set-type <ht> ) })`

`{ "sys-hs-gc-mut-reqs" } Response`

`(λ s . { (s){ handshakes-pending ::= (handshakes-pending s)(m ::= True) },
          m. req = (gc, ro-hs-gc-set-pending m) })`

`{ "sys-hs-gc-done" } Response`

`(λ s . { (s, mv-Bool (¬handshakes-pending s m))
          |m. req = (gc, ro-hs-gc-load-W ) })`

`{ "sys-hs-gc-load-W" } Response`

`(λ s . { (s){ W ::= { } }, mv-Refs (W s))
          |::unit. req = (gc, ro-hs-gc-load-W ) })`

`{ "sys-hs-mut" } Response`

`(λ s . { (s, mv-Void)
          |m. req = (mutator m, ro-hs-mut-read-type (handshakes-type s))
          ∧ handshakes-pending s m })`

`{ "sys-hs-mut-done" } Response`

`(λ s . { (s){ handshakes-pending ::= (handshakes-pending s)(m ::= False),
          W ::= W s U W’,
          handshakes-in-sync ::= (handshakes-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 (l- ref) mark'-object where`

`l-mark-object ≡ mark-object-fn (mutator m) l`

**abbreviation** `<mfence> :: location ⇒ ('field, 'mut, 'ref) gc-com (l- ref) MFENCE where`

`l-MFENCE ≡ l-mark-object ≡ Request (λ s. (mutator m, ro-MFENCE)) (λ- s. {s})`

**abbreviation** `<hs-read-type> :: location ⇒ handshakes-type ⇒ ('field, 'mut, 'ref) gc-com (l- ref) hs'-read'-type where`
abbreviation hs-noop-done :: location ⇒ ('field, 'mut, 'ref) gc-com (\{s\} hs'-noop'-done) where
\{l\} hs-noop-done ≡ \{l\} Request (λs. (mutator m, ro-hs-mut-done \{s\}))
(λ- s. \{s\} ghost-handshake-phase := handshake-step (ghost-handshake-phase s) \})

abbreviation hs-get-roots-done :: location ⇒ (('field, 'mut, 'ref) local-state ⇒ 'ref set) ⇒ (('field, 'mut, 'ref) gc-com (\{s\} hs'-get-roots'-done) where
\{l\} hs-get-roots-done wl ≡ \{l\} Request (λs. (mutator m, ro-hs-mut-done (wl s)))
(λ- s. \{s\} W := \{\})

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

definition handshake :: ('field, 'mut, 'ref) gc-com where
handshake ≡
\{"hs-noop\"\} hs-read-type ht-NOOP ;;
\{"hs-noop-mfence\"\} M Fence ;;
\{"hs-noop-done\"\} hs-noop-done
\[ \]
\{"hs-get-roots\"\} hs-read-type ht-GetRoots ;;
\{"hs-get-roots-mfence\"\} M Fence ;;
\{"hs-get-roots-refs\"\} ´refs := ´roots ;;
\{"hs-get-roots-loop\"\} WHILE ¬(EMPTY refs) DO
\{"hs-get-roots-loop-choose-ref\"\} ´ref := Some ´refs ;;
\{"hs-get-roots-loop\"\} mark-object ;;
\{"hs-get-roots-loop-done\"\} ´refs := (´refs − \{the ´ref\})
OD ;;
\{"hs-get-roots-done\"\} hs-get-roots-done W
\[ \]
\{"hs-get-work\"\} hs-read-type ht-GetWork ;;
\{"hs-get-work-mfence\"\} M Fence ;;
\{"hs-get-work-done\"\} hs-get-work-done W

end

The garbage collector’s side of the interface.

context gc

begin

abbreviation set-hs-type :: location ⇒ handshake-type ⇒ ('field, 'mut, 'ref) gc-com (\{s\} set'-hs'-type) where
\{l\} set-hs-type ht ≡ \{l\} Request (λs. (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 (\{s\} set'-hs'-pending) where
\{l\} set-hs-pending m ≡ \{l\} Request (λs. (gc, ro-hs-gc-set-pending (m s))) (λ- s. \{s\})

definition handshake-init :: location ⇒ handshake-type ⇒ ('field, 'mut, 'ref) gc-com (\{s\} handshake'-init) where
\{l\} handshake-init req ≡
\{l @ "-init-type\"\} set-hs-type req ;;
\{l @ "-init-muts\"\} ´muts := UNIV ;;
\{l @ "-init-loop\"\} WHILE ¬(EMPTY muts) DO
\{l @ "-init-loop-choose-mut\"\} ´mut := ´muts ;;

definition
handshake-done :: location ⇒ (′field, ′mut, ′ref) gc-com (¶- handshaké’-done)
where
¶ handshake-done ≡
¶ [l] handshake-done ≡
¶ [l] @ ′-done-muts′] ′muts := UNIV ·;
¶ [l] @ ′-done-loop′] WHILE ¬(EMPTY muts) DO
¶ [l] @ ′-done-loop-choose-mut′] ′mut ∈ ′muts ·;
¶ [l] @ ′-done-loop-rendezvous′] Request
(As. (gc, ro-hs-gc-read-pending (mut s)))
(λmv s. { s\| muts := muts s − { mut s | done. mv = mv-Bool done ∧ done }) })
OD

abbreviation load-W :: location ⇒ (′field, ′mut, ′ref) gc-com (¶- load′-W) where
¶ load-W ≡ [l] [l] @ ′-load-W′] Request (λs. (gc, ro-hs-gc-load-W))
(λresp s. { s\| W := W′ | W′. resp = mv-Refs W′ })

abbreviation mfence :: location ⇒ (′field, ′mut, ′ref) gc-com (¶- MFENCE) where
¶ mfence ≡ [l] [l] Request (λs. (gc, ro-MFENCE)) (λ- s { s })

definition
handshake-noop :: location ⇒ (′field, ′mut, ′ref) gc-com (¶- handshaké′-noop)
where
¶ handshake-noop ≡
¶ [l] handshake-noop ≡
¶ [l] @ ′-mfence′] MFENCE ·;
¶ [l] handshake-init ht-NOOP ·;
¶ [l] handshake-done

definition
handshake-get-roots :: location ⇒ (′field, ′mut, ′ref) gc-com (¶- handshaké′-get′-roots)
where
¶ handshake-get-roots ≡
¶ [l] handshake-get-roots ≡
¶ [l] handshake-init ht-GetRoots ·;
¶ [l] handshake-done ·;
¶ [l] load-W

definition
handshake-get-work :: location ⇒ (′field, ′mut, ′ref) gc-com (¶- handshaké′-get′-work)
where
¶ handshake-get-work ≡
¶ [l] handshake-get-work ≡
¶ [l] handshake-init ht-GetWork ·;
¶ [l] handshake-done ·;
¶ [l] load-W

end

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

\[
\text{do-write-action} :: (\text{'field}, \text{'ref}) \text{ mem-write-action} \Rightarrow (\text{'field}, \text{'mut}, \text{'ref}) \text{ local-state}
\]

where

\[
\text{do-write-action wact} \equiv \lambda s. \text{case wact of}
\]

- \text{mw-Mark r gc-mark} \Rightarrow s[\text{heap} := (\text{heap} s)(r := \text{map-option } (\lambda \text{obj}. \text{obj-fields} := (\text{obj-fields} \text{ obj})(f := \text{mnew-r}))(\text{heap} s r))]

- \text{mw-Mutate r f new-r} \Rightarrow s[\text{heap} := (\text{heap} s)(r := \text{map-option } (\lambda \text{obj}. \text{obj-fields} := (\text{obj-fields} \text{ obj})(f := \text{new-r}))(\text{heap} s r))]

- \text{mw-fM gc-mark} \Rightarrow s[\text{fM} := \text{gc-mark}]

- \text{mw-fA gc-mark} \Rightarrow s[\text{fA} := \text{gc-mark}]

- \text{mw-Phase gc-phase} \Rightarrow s[\text{phase} := \text{gc-phase}]

\[
\text{fold-writes} :: (\text{'field}, \text{'ref}) \text{ mem-write-action list} \Rightarrow (\text{'field}, \text{'mut}, \text{'ref}) \text{ local-state}
\]

where

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

abbreviation

\[
\text{processors-view-of-memory} :: \text{'mut process-name} \Rightarrow (\text{'field}, \text{'mut}, \text{'ref}) \text{ local-state}
\]

where

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

definition

\[
\text{do-read-action} :: (\text{'field}, \text{'ref}) \text{ mem-read-action}
\]

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

\[
\Rightarrow (\text{'field}, \text{'ref}) \text{ response}
\]

where

\[
\text{do-read-action } ract \equiv \lambda s.
\]

\[
\text{case ract of}
\]

- \text{mr-Ref r f} \Rightarrow \text{mv-Ref } (\text{heap} s r \Rightarrow (\lambda \text{obj}. \text{obj-fields} := (\text{obj-fields} \text{ obj})(f := \text{mr-f})}(\text{heap} s r))

- \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))

definition

\[
\text{sys-read} :: \text{'mut process-name}
\]

\[
\Rightarrow (\text{'field}, \text{'ref}) \text{ mem-read-action}
\]

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

\[
\Rightarrow (\text{'field}, \text{'ref}) \text{ response}
\]

where

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

context \text{sys}

\begin{enumerate}
\item 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.
\end{enumerate}
mem-TSO :: ('field, 'mut, 'ref) gc-com

where

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

| \ p \ w. \ req = (p, \text{ro-Write } w) \}
\sqcup \{\text{"sys-mfence"}\} \quad & \text{Response } (\lambda req s. \{ (s, \text{mv-Void} \}) \\
\sqcup \{\text{"sys-lock"}\} \quad & \text{Response } (\lambda req s. \{ (s\| \text{mem-lock} := \text{Some } p \}), \text{mv-Void} \}) \\
\sqcup \{\text{"sys-unlock"}\} \quad & \text{Response } (\lambda req s. \{ (s\| \text{mem-lock} := \text{None } \}, \text{mv-Void} \}) \\
\sqcup \{\text{"sys-dequeue-write-buffer"}\} \quad & \text{LocalOp } (\lambda s. \{ \text{(do-write-action } w \ s)\| \text{mem-write-buffers} := (\text{mem-write-buffers } s)(p := \text{ws}) \})
\end{align*}

\begin{align*}
& | \ 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 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 abort in Pizlo (201x, Figure 2.9: Alloc) means the atomic fails and the mutator can revert to activity outside of Alloc, avoiding deadlock.

**definition**

alloc :: ('field, 'mut, 'ref) gc-com

where

\begin{align*}
\text{alloc} \equiv \{\text{"sys-alloc"}\} \quad & \text{Response } (\lambda req s. \\
& \{ (s\| \text{heap} := (\text{heap } s)(r := \text{Some } ⟨\text{obj-mark} = \text{fA } s, \text{obj-fields} = (\text{None } \rangle) \}), \text{mv-Ref } (\text{Some } r) \}) \\
& | r. \ r \ \notin \ \text{dom } (\text{heap } s) \ \land \ \text{snd req} = \text{ro-Alloc} \}
\end{align*}

References are freed by removing them from heap.

**definition**

free :: ('field, 'mut, 'ref) gc-com

where

\begin{align*}
\text{free} \equiv \{\text{"sys-free"}\} \quad & \text{Response } (\lambda req s. \\
& \{ (s\| \text{heap} := (\text{heap } s)(r := \text{None } \rangle)), \text{mv-Void} \}) \ | r. \ \text{snd req} = \text{ro-Free } r \}
\end{align*}

The top-level system process.

**definition**

com :: ('field, 'mut, 'ref) gc-com

where

\begin{align*}
\text{com} \equiv \text{LOOP DO} \\
& \text{mem-TSO} \\
& \sqcup \text{alloc} \\
& \sqcup \text{free} \\
& \sqcup \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

\[
\text{mut-alloc } m \equiv \\
\{ "alloc" \} \text{ Request } (\lambda s. \text{(mutator } m\text{, ro-Alloc)}) \\
(\lambda m v s. \{ s | \text{roots} := \text{roots } s \cup \{ r \} | r. \text{mv} = \text{mv-Ref } (\text{Some } r) \})
\]

**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

\[
\text{discard } \equiv \\
\{ "discard-refs" \} \text{ LocalOp } (\lambda s. \{ s | \text{roots} := \text{roots } s \})
\]

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

\[
\text{load } \equiv \\
\{ "load-choose" \} \text{ LocalOp } (\lambda s. \{ s | \text{tmp-ref} := r, \text{field} := f \} | r f. \text{r } \in \text{roots } s \}) ;
\{ "load" \} \text{ Request } (\lambda s. \text{(mutator } m\text{, ReadRef } (\text{tmp-ref } s \text{ (field } s))) \\
(\lambda m v s. \{ s | \text{roots} := \text{roots } s \cup \text{set-option } r \}) \\
| r. \text{mv} = \text{mv-Ref } r)
\]

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 the mutator reads the overwritten reference but does not store it in its roots.

**abbreviation**

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

**where**

\[
\text{\{l\} deref } r f \text{ upd } \equiv \text{\{l\} Request } (\lambda s. \text{(mutator } m\text{, ReadRef } (r \text{ s) } (f \text{s))}) \\
(\lambda m v s. \{ \text{upd } (\text{opt-r'}) } (s[\text{ghost-honorary-root} := \text{set-option } \text{opt-r'}]) | \text{opt-r'. mv} = \text{mv-Ref } \text{opt-r'})
\]

**abbreviation**

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

**where**

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

**definition**

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

\[
\text{store} \equiv
\begin{array}{l}
\text{- Choose vars for: } \text{ref} \rightarrow \text{field} := \text{new-ref} \\
\text{\{"store-choose"\}} \text{ LocalOp } (\lambda s. \left\{ \text{tmp-ref} := r, \text{field} := f, \text{new-ref} := r' \right\} \\
| r f r'. r \in \text{roots } s \land r' \in \text{Some } \text{roots } s \cup \{\text{None} \} ) ;;
\end{array}
\]

— Mark the reference we’re about to overwrite. Does not update roots.
\[
\text{\{"deref-del"\}} \text{ deref tmp-ref field ref-update };;
\]
\[
\text{\{"store-del"\}} \text{ mark-object };;
\]

— 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 MFENCE instructions. We leave CAS (etc) to future work. Neither has a significant impact on the rest of the development.

definition

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

where

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

OD

end

2.5 Garbage collector

We abstract the primitive actions of the garbage collector thread.

abbreviation

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

where

\[
\text{gc-deref } l \ r \ f \ \text{upd} \equiv \\{l\} \text{ Request } (\lambda s. (\text{gc, ReadRef } (r s) (f s))) \\
(\lambda mv s. \left\{ \text{ upd } (r') s |r'. \text{ mv} = \text{mv-Ref } r' \right\})
\]

abbreviation

\[
\text{gc-read-mark} :: \text{location} \\
\Rightarrow ((\text{\{field, mut, ref\}} \text{ local-state} \Rightarrow \text{\{ref\}}) \\
\Rightarrow ((\text{\{gc-mark option} \Rightarrow \text{gc-mark option} \Rightarrow ((\text{\{field, mut, ref\}} \text{ local-state} \Rightarrow ((\text{\{field, mut, ref\}} \text{ local-state} \Rightarrow ((\text{\{field, mut, ref\}} \text{ gc-com })
\]

where

\[
\text{gc-read-mark } l \ r \ f \ \text{upd} \equiv \\{l\} \text{ Request } (\lambda s. (\text{gc, ReadMark } (r s))) (\lambda mv s. \left\{ \text{ upd } (m) s |m. \text{ mv} = \text{mv-Mark } m \right\})
\]

syntax

- \text{\{-gc-fassign\}} :: \text{location} \Rightarrow \text{idt} \Rightarrow \text{\{ref\} \Rightarrow \text{\{field\} \Rightarrow (\text{\{field, mut, ref\}} \text{ gc-com } (l\{l\} \text{ } \cdot := \text{\{-\} } \Rightarrow \left[0, 0, 70\right] 71)\}
- \text{\{-gc-massign\}} :: \text{location} \Rightarrow \text{idt} \Rightarrow \text{\{ref\} \Rightarrow (\text{\{field, mut, ref\}} \text{ gc-com } (l\{l\} \text{ } \cdot := \text{\{-\} } \Rightarrow \text{flag } \left[0, 0\right] 71)\}

translations
The following CIMP program encodes the garbage collector algorithm proposed in Figure 2.15 of Pizlo (201x).
\textbf{mark-loop-mark-deref} \emph{ref} := \emph{tmp-ref} \rightarrow \emph{field} ;
\textbf{mark-loop} \emph{mark-object} ;
\textbf{mark-loop-mark-field-done} \emph{field-set} := (\emph{field-set} \setminus \{\emph{field}\}) 
\textbf{OD} ;
\textbf{mark-loop-blacken} \emph{W} := (\emph{W} \setminus \{\emph{tmp-ref}\}) 
\textbf{OD} ;
\textbf{mark-loop-get-work} \textbf{handshake-get-work} 
\textbf{OD} ;

\begin{itemize}
\item sweep
\end{itemize}
\textbf{mark-end} \textbf{write-phase ph-Sweep} ;
\textbf{sweep-read-fM} \textbf{read-fM} ;
\textbf{sweep-refs} \emph{refs} := \text{UNIV} ;
\textbf{WHILE \neg (EMPTY refs)} \textbf{DO}
\textbf{sweep-loop-choose-ref} \emph{tmp-ref} \in \emph{refs} ;
\textbf{sweep-loop-read-mark} \emph{mark} := \emph{tmp-ref} \rightarrow \text{flag} ;
\textbf{sweep-loop-check} \text{IF \neg (NULL mark) \& the \circ mark \neq fM THEN}
\textbf{sweep-loop-free} \textbf{free tmp-ref}
\textbf{FI} ;
\textbf{sweep-loop-ref-done} \emph{refs} := (\emph{refs} \setminus \{\emph{tmp-ref}\}) 
\textbf{OD} ;
\textbf{sweep-idle} \textbf{write-phase ph-Idle} 
\textbf{OD}

end

\begin{itemize}
\item \textbf{primrec}
\end{itemize}
\textbf{gc-pgms} :: \('\text{mut process-name} \Rightarrow (\text{\emph{field}}, \text{\emph{mut}}, \text{\emph{ref}}) \text{\emph{gc-com}}}
\textbf{where}
\textbf{gc-pgms} (\text{mutator } m) = \text{mut-m.com } m
\text{\mid gc-pgms gc = gc.com}
\text{\mid gc-pgms sys = sys.com}

\section{Invariants and Proofs}

\subsection{Constructors for sets of locations.}
\textbf{abbreviation} \text{prefixed} :: \text{location} \Rightarrow \text{location set} \textbf{where}
\text{prefixed } p \equiv \{l . \text{prefix } p \ l\} 
\textbf{abbreviation} \text{suffixed} :: \text{location} \Rightarrow \text{location set} \textbf{where}
\text{suffixed } p \equiv \{l . \text{suffix } p \ l\} 

\subsection{Hoare triples}
Specialise CIMP's pre/post validity to our system.
\textbf{definition}
\text{valid-proc} :: (\text{\emph{field}}, \text{\emph{mut}}, \text{\emph{ref}}) \text{\emph{gc-pred}} \Rightarrow \text{\emph{mut process-name} \Rightarrow (\text{\emph{field}}, \text{\emph{mut}}, \text{\emph{ref}}) \text{\emph{gc-pred}} \Rightarrow \text{bool} (\| | - \| |))
\textbf{where}
\|P\| p \|Q\| \equiv \forall (c, \text{afts}) \in \text{vcg-fragments} (\text{gc-pgms } p). (\text{gc-pgms}, p, \text{afts} \models |P| c |Q|)
\textbf{abbreviation}
\text{valid-proc-inv-syn} :: (\text{\emph{field}}, \text{\emph{mut}}, \text{\emph{ref}}) \text{\emph{gc-pred}} \Rightarrow \text{\emph{mut process-name} \Rightarrow \text{bool} (\| | - [100.0] 100)}
\textbf{where}
\|P\| p \equiv \|P\| p \|P\|

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.

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)).

\begin{verbatim}
abbreviation is-mw-Mark w \equiv \exists r f l. w = mw-Mark r f l
abbreviation is-mw-Mutate w \equiv \exists r f f'. w = mw-Mutate r f f'
abbreviation is-mw-fA w \equiv \exists f l. w = mw-fA f l
abbreviation is-mw-fM w \equiv \exists f l. w = mw-fM f l
abbreviation is-mw-Phase w \equiv \exists ph. w = mw-Phase ph

abbreviation (input) pred-in-W :: 'ref \Rightarrow 'mut process-name \Rightarrow ('field, 'mut, 'ref) lsts-pred (infix in' W 50) where
  r in-W p \equiv \lambda s. r \in W (s p)

abbreviation (input) pred-in-ghost-honorary-grey :: 'ref \Rightarrow 'mut process-name \Rightarrow ('field, 'mut, 'ref) lsts-pred (infix in' ghost'honorary'grey 50) where
  r in-ghost-honorary-grey p \equiv \lambda s. r \in ghost-honorary-grey (s p)

context gc
begin

abbreviation valid-gc-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 bool
  (\_ \Rightarrow \{\_\}/ \_ \Rightarrow \{\_\})
where
  afts \mid \{P\} \equiv gc-pgms, gc, afts \mid \{P\}

abbreviation valid-gc-inv-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 bool
  (\_ \Rightarrow \{\_\}/\_ \Rightarrow \{\_\}) where
  afts \mid \{P\} \equiv afts \mid \{P\}

end

abbreviation gc-cas-mark s \equiv cas-mark (s gc)
abbreviation gc-fM s \equiv fM (s gc)
abbreviation gc-field s \equiv field (s gc)
abbreviation gc-field-set s \equiv field-set (s gc)
abbreviation gc-mark s \equiv mark (s gc)
abbreviation gc-mut s \equiv mut (s gc)
abbreviation gc-muts s \equiv muts (s gc)
abbreviation gc-phase s \equiv phase (s gc)
abbreviation gc-tmp-ref s \equiv tmp-ref (s gc)
abbreviation gc-ghost-honorary-grey s \equiv ghost-honorary-grey (s gc)
abbreviation gc-ref s \equiv ref (s gc)
abbreviation gc-refs s \equiv refs (s gc)
abbreviation gc-the-ref \equiv \degree gc-ref
abbreviation gc-W s \equiv W (s gc)

abbreviation at-gc :: location \Rightarrow ('field, 'mut, 'ref) lsts-pred \Rightarrow ('field, 'mut, 'ref) gc-pred where
  at-gc l P \equiv at gc l \longrightarrow LSTP P

abbreviation atS-gc :: location set \Rightarrow ('field, 'mut, 'ref) lsts-pred \Rightarrow ('field, 'mut, 'ref) gc-pred where
  atS-gc ls P \equiv atS gc ls \longrightarrow LSTP P
\end{verbatim}
context mut-m
begin

abbreviation valid-mut-syn :: ('field, 'mut, 'ref) gc-loc-comp ⇒ ('field, 'mut, 'ref) gc-pred ⇒ ('field, 'mut, 'ref) gc-com ⇒
    ('field, 'mut, 'ref) gc-pred ⇒ bool
    (| = \{} / - / \{}]
where
    afts |={ P }, c ={ Q } = gc-pgms, mutator m, afts |={ P }, c ={ Q }

abbreviation valid-mut-inv-syn :: ('field, 'mut, 'ref) gc-loc-comp ⇒ ('field, 'mut, 'ref) gc-pred ⇒ ('field, 'mut, 'ref) gc-com ⇒
    ('field, 'mut, 'ref) gc-pred ⇒ bool (- | = \{} / - ) where
    afts |={ P }, c = afts |={ P } c ={ P }

abbreviation at-mut :: location ⇒ ('field, 'mut, 'ref) lsts-pred ⇒ ('field, 'mut, 'ref) gc-pred where
    at-mut l P = at (mutator m) l ─> LSTP P

abbreviation atS-mut :: location set ⇒ ('field, 'mut, 'ref) lsts-pred ⇒ ('field, 'mut, 'ref) gc-pred where
    atS-mut ls P = atS (mutator m) ls ─> LSTP P

abbreviation mut-cas-mark s = cas-mark (s (mutator m))
abbreviation mut-field s = field (s (mutator m))
abbreviation mut-fM s = fM (s (mutator m))
abbreviation mut-ghost-honorary-grey s = ghost-honorary-grey (s (mutator m))
abbreviation mut-ghost-handshake-phase s = ghost-handshake-phase (s (mutator m))
abbreviation mut-ghost-honorary-root s = ghost-honorary-root (s (mutator m))
abbreviation mut-mark s = mark (s (mutator m))
abbreviation mut-new-ref s = new-ref (s (mutator m))
abbreviation mut-phase s = phase (s (mutator m))
abbreviation mut-ref s = ref (s (mutator m))
abbreviation mut-tmp-ref s = tmp-ref (s (mutator m))
abbreviation mut-the-new-ref = the o mut-new-ref
abbreviation mut-the-ref = the o mut-ref
abbreviation mut-refs s = refs (s (mutator m))
abbreviation mut-roots s = roots (s (mutator m))
abbreviation mut-W s = W (s (mutator m))
end

context sys
begin

abbreviation valid-sys-syn :: ('field, 'mut, 'ref) gc-loc-comp ⇒ ('field, 'mut, 'ref) gc-pred ⇒ ('field, 'mut, 'ref) gc-com ⇒
    ('field, 'mut, 'ref) gc-pred ⇒ bool
    (| = \{} / - / \{}]
where
    afts |={ P }, c ={ Q } = gc-pgms, sys, afts |={ P }, c ={ Q }

abbreviation valid-sys-inv-syn :: ('field, 'mut, 'ref) gc-loc-comp ⇒ ('field, 'mut, 'ref) gc-pred ⇒ ('field, 'mut, 'ref) gc-com ⇒
    ('field, 'mut, 'ref) gc-pred ⇒ bool (- | = \{} / - ) where
    afts |={ P }, c = afts |={ P } c ={ P }
end

abbreviation sys-heap :: ('field, 'mut, 'ref) lsts ⇒ 'ref ⇒ ('field, 'ref) object option where
    sys-heap s = heap (s sys)
abbreviation \texttt{sys-fA} \( s \equiv fA \ (s \text{ sys}) \)
abbreviation \texttt{sys-fM} \( s \equiv fM \ (s \text{ sys}) \)
abbreviation \texttt{sys-ghost-honorary-grey} \( s \equiv \text{ghost-honorary-grey} \ (s \text{ sys}) \)
abbreviation \texttt{sys-ghost-handshake-in-sync} \( m \equiv \text{ghost-handshake-in-sync} \ (s \text{ sys}) \)
abbreviation \texttt{sys-ghost-handshake-phase} \( s \equiv \text{ghost-handshake-phase} \ (s \text{ sys}) \)
abbreviation \texttt{sys-handshake-pending} \( m \equiv \text{handshake-pending} \ (s \text{ sys}) \)
abbreviation \texttt{sys-handshake-type} \( s \equiv \text{handshake-type} \ (s \text{ sys}) \)
abbreviation \texttt{sys-mem-lock} \( s \equiv \text{mem-lock} \ (s \text{ sys}) \)
abbreviation \texttt{sys-mem-write-buffers} \( p \equiv \text{mem-write-buffers} \ (s \text{ sys}) \)
abbreviation \texttt{sys-handshake-pending} \( m \equiv \text{handshake-pending} \ (s \text{ sys}) \)
abbreviation \texttt{sys-ghost-handshake-phase} \( s \equiv \text{ghost-handshake-phase} \ (s \text{ sys}) \)
abbreviation \texttt{sys-ghost-handshake-in-sync} \( m \equiv \text{ghost-handshake-in-sync} \ (s \text{ sys}) \)
abbreviation \texttt{sys-phase} \( s \equiv \text{phase} \ (s \text{ sys}) \)
abbreviation \texttt{sys-W} \( s \equiv W \ (s \text{ sys}) \)
abbreviation \texttt{atS-sys} \( l \equiv P \)
abbreviation \texttt{sys-ghost-handshake-phase} \( s \equiv \text{ghost-handshake-phase} \ (s \text{ sys}) \)
abbreviation \texttt{sys-ghost-handshake-in-sync} \( m \equiv \text{ghost-handshake-in-sync} \ (s \text{ sys}) \)
abbreviation \texttt{sys-ghost-honorary-grey} \( s \equiv \text{ghost-honorary-grey} \ (s \text{ sys}) \)
abbreviation \texttt{sys-mem-lock} \( s \equiv \text{mem-lock} \ (s \text{ sys}) \)
abbreviation \texttt{sys-phase} \( s \equiv \text{phase} \ (s \text{ sys}) \)
abbreviation \texttt{sys-W} \( s \equiv W \ (s \text{ sys}) \)

abbreviation \texttt{atS-sys} \( :: \) location set \( \Rightarrow \) (\text{field}, \text{mut}, \text{ref}) lsts-pred \( \Rightarrow \) (\text{field}, \text{mut}, \text{ref}) gc-pred where
\texttt{atS-sys}\( l \equiv atS\ (s\ sys)\)
\texttt{lSTP}\( P \) Projections on TSO buffers.

abbreviation (input) \texttt{tso-unlocked} \( s \equiv \text{mem-lock} \ (s \text{ sys}) \) = None
abbreviation (input) \texttt{tso-locked-by} \( p \) \( s \equiv \text{mem-lock} \ (s \text{ sys}) \) = Some \( p \)

abbreviation (input) \texttt{tso-pending} \( p \) \( P \) \( s \equiv \text{filter} \ (s \text{ sys}) \) \( p \)
abbreviation (input) \texttt{tso-pending-write} \( p \) \( w \) \( s \equiv w \in \text{set} \ (s \text{ sys}) \) \( p \)

abbreviation (input) \texttt{tso-pending-fA} \( p \equiv \text{tso-pending} \ (s \text{ sys}) \) \( p \)
abbreviation (input) \texttt{tso-pending-fM} \( p \equiv \text{tso-pending} \ (s \text{ sys}) \) \( p \)
abbreviation (input) \texttt{tso-pending-mark} \( p \equiv \text{tso-pending} \ (s \text{ sys}) \) \( p \)
abbreviation (input) \texttt{tso-pending-mutate} \( p \equiv \text{tso-pending} \ (s \text{ sys}) \) \( p \)
abbreviation (input) \texttt{tso-pending-phase} \( p \equiv \text{tso-pending} \ (s \text{ sys}) \) \( p \)

abbreviation (input) \texttt{tso-no-pending-mark} \( m \equiv \forall \ p. \ L\text{IST-NULL} \ (s\ sys) \)

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 \texttt{obj-at} :: (\text{field}, \text{mut}, \text{ref}) object \Rightarrow \text{bool} \Rightarrow \text{ref} \Rightarrow (\text{field}, \text{mut}, \text{ref}) lsts-pred where
\texttt{obj-at}\( P \) \( r \equiv \lambda s. \ \text{case} \ \text{sys-heap} \ s \asto \ r \ \Rightarrow \ \text{False} \ \lor \ \text{Some} \ obj \Rightarrow P \ obj \)

definition \texttt{valid-ref} :: \text{ref} \Rightarrow (\text{field}, \text{mut}, \text{ref}) lsts-pred where
\texttt{valid-ref}\( r \equiv \text{obj-at} \ (\text{True}) \ r \)

definition \texttt{valid-null-ref} :: \text{ref} \ \text{option} \Rightarrow (\text{field}, \text{mut}, \text{ref}) lsts-pred where
\texttt{valid-null-ref}\( r \equiv \text{case} \ r \asto \ \text{None} \Rightarrow (\text{True}) \ \lor \ \text{Some} \ r' \Rightarrow \text{valid-ref} \ r' \)

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

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

abbreviation \texttt{pred-reaches} :: \text{ref} \Rightarrow \text{ref} \Rightarrow (\text{field}, \text{mut}, \text{ref}) lsts-pred \ (\text{infix reaches 51}) \ where
\texttt{pred-reaches}\( x \asto \ y \equiv \lambda s. \ (\\lambda x. \ (x \pto \ y) \ s)^* \ x \ y \)

The predicate \texttt{obj-at-field-on-heap} asserts that \texttt{obj-at} \ (\lambda s. \text{True}) \ r and if \( f \) is a field of the object referred to by \( r \) then it satisfies \( P \).

definition \texttt{obj-at-field-on-heap} :: (\text{ref} \Rightarrow \text{bool}) \Rightarrow \text{ref} \Rightarrow \text{ref} \Rightarrow (\text{field}, \text{mut}, \text{ref}) lsts-pred where
\texttt{obj-at-field-on-heap}\( P \) \( f \equiv \lambda s. \ \text{case} \ \text{Option.map-option obj-fields} \ (\text{sys-heap} \ s \ r) \asto
\text{None} \Rightarrow \text{False}
\text{Some} \ f \Rightarrow (\text{case} \ f \asto \text{of} \ \text{None} \Rightarrow \text{True}
\text{Some} \ r' \Rightarrow P \ r') \)
3.4 Garbage collector locations

locset-definition idle-locs = prefixed "idle"
locset-definition init-locs = prefixed "init"
locset-definition mark-locs = prefixed "mark"
locset-definition mark-loop-locs = prefixed "mark-loop"
locset-definition sweep-locs = prefixed "sweep"

3.5 Coarse TSO invariants

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

abbreviation gc-writes :: (field, ref) mem-write-action \Rightarrow bool where
\[ gc\text{-}writes w \equiv \begin{cases} True & \text{if } mw\text{-Mark } - - \Rightarrow True \text{ or } mw\text{-Phase } - - \Rightarrow True \text{ or } mw\text{-fM } - \Rightarrow True \text{ or } mw\text{-fA } - \Rightarrow True \\ False & \text{otherwise} \end{cases} \]

abbreviation mut-writes :: (field, ref) mem-write-action \Rightarrow bool where
\[ mut\text{-}writes w \equiv \begin{cases} True & \text{if } mw\text{-Mutate } - - - - \Rightarrow True \text{ or } mw\text{-Mark } - - \Rightarrow True \\ False & \text{otherwise} \end{cases} \]

definition tso-writes-inv :: (field, mut, ref) lsts-pred where
\[ tso\text{-}writes-inv \equiv (\forall w. tso\text{-}pending\text{-}write gc w \Rightarrow (gc\text{-}writes w)) \land (\forall m w. tso\text{-}pending\text{-}write (mutator m) w \Rightarrow (mut\text{-}writes w)) \]

3.5.1 Locations where the TSO lock is held

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

locset-definition gc-tso-lock-locs :: location set where
\[ gc\text{-}tso\text{-}lock\text{-}locs \equiv \bigcup \{ \text{"mo-co-cmark", } \text{"mo-co-ctest", } \text{"mo-co-mark", } \text{"mo-co-unlock" } \}. \text{ suffixed } l \]

inv-definition (in gc) tso-lock-invL :: (field, mut, ref) gc-pred where
\[ tso\text{-}lock-invL \equiv \begin{cases} atS\text{-}gc gc\text{-}tso\text{-}lock\text{-}locs (tso\text{-}locked\text{-}by gc) \\ \land atS\text{-}gc (tso\text{-}tso\text{-}lock\text{-}locs) (\neg (tso\text{-}locked\text{-}by gc)) \end{cases} \]

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

locset-definition mut-tso-lock-locs =
\[ \bigcup \{ \text{"mo-co-cmark", } \text{"mo-co-ctest", } \text{"mo-co-mark", } \text{"mo-co-unlock" } \}. \text{ suffixed } l \]

inv-definition (in mut-m) tso-lock-invL :: (field, mut, ref) gc-pred where
\[ tso\text{-}lock-invL = \begin{cases} atS\text{-}mut mut\text{-}tso\text{-}lock\text{-}locs (tso\text{-}locked\text{-}by (mutator m)) \\ \land atS\text{-}mut (tso\text{-}tso\text{-}lock\text{-}locs) (\neg (tso\text{-}locked\text{-}by (mutator m))) \end{cases} \]

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
\[ hp\text{-}step-rel \equiv \begin{cases} \{ True \} \times \{ (ht\text{-}NOOP, hp, hp) \mid hp, hp \in \{ hp\text{-}Idle, hp\text{-}IdleInit, hp\text{-}InitMark, hp\text{-}Mark \} \} \\ \cup \{ (ht\text{-}GetRoots, hp\text{-}IdleMarkSweep, hp\text{-}IdleMarkSweep) \\ \land (ht\text{-}GetWork, hp\text{-}IdleMarkSweep, hp\text{-}IdleMarkSweep) \} \} \\ \cup \{ False \} \times \{ (ht\text{-}NOOP, hp\text{-}Idle, hp\text{-}IdleMarkSweep) \\ \land (ht\text{-}NOOP, hp\text{-}IdleInit, hp\text{-}Idle) \\ \land (ht\text{-}NOOP, hp\text{-}InitMark, hp\text{-}IdleInit) \} \]
\begin{definition}
handshake-phase-inv :: ('field, 'mut, 'ref) lsts-pred
\begin{align*}
\text{handshake-phase-inv} &= (\forall 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 \{\text{hp-step-rel}\} \\
&\quad \land (\text{sys-handshake-pending } m \rightarrow \neg (\text{sys-ghost-handshake-in-sync } m)))
\end{align*}
\end{definition}

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

\begin{locset-definition}
\text{hp-Idle-locs} = \\
\quad (\text{prefixed } "\text{idle-noop}" - \{ "\text{idle-noop-mfence}, "\text{idle-noop-init-type}" \}) \\
\quad \cup \{ "\text{idle-read-fM}, "\text{idle-invert-fM}, "\text{idle-write-fM}, "\text{idle-flip-noop-mfence}, "\text{idle-flip-noop-init-type}" \}
\end{locset-definition}

\begin{locset-definition}
\text{hp-IdleInit-locs} = \\
\quad (\text{prefixed } "\text{init-noop}" - \{ "\text{init-noop-mfence}, "\text{init-noop-init-type}" \}) \\
\quad \cup \{ "\text{init-phase-mark}, "\text{mark-read-fM}, "\text{mark-write-fA}, "\text{mark-noop-mfence}, "\text{mark-noop-init-type}" \}
\end{locset-definition}

\begin{locset-definition}
\text{hp-IdleMarkSweep-locs} = \\
\quad \{ "\text{idle-noop-mfence}, "\text{idle-noop-init-type}, "\text{mark-end}" \} \\
\quad \cup \text{sweep-locs} \\
\quad \cup (\text{mark-loop-locs} - \{ "\text{mark-loop-get-roots-init-type}" \})
\end{locset-definition}

\begin{locset-definition}
\text{hp-Mark-locs} = \\
\quad (\text{prefixed } "\text{mark-noop}" - \{ "\text{mark-noop-mfence}, "\text{mark-noop-init-type}" \}) \\
\quad \cup \{ "\text{mark-loop-get-roots-init-type}" \}
\end{locset-definition}

abbreviation
\text{hs-noop-prefixes} \equiv \{ "\text{idle-noop}, "\text{idle-flip-noop}, "\text{init-noop}, "\text{mark-noop}" \}

\begin{locset-definition}
\text{hs-noop-locs} = \\
(\bigcup l \in \text{hs-noop-prefixes}. \text{prefixed } l - (\text{suffixed } "\text{-noop-mfence}" \cup \text{suffixed } "\text{-noop-init-type}"))
\end{locset-definition}

\begin{locset-definition}
\text{hs-get-roots-locs} = \\
\quad \text{prefixed } "\text{mark-loop-get-roots}" - \{"\text{mark-loop-get-roots-init-type}" \}
\end{locset-definition}

\begin{locset-definition}
\text{hs-get-work-locs} = \\
\quad \text{prefixed } "\text{mark-loop-get-work}" - \{"\text{mark-loop-get-work-init-type}" \}
\end{locset-definition}

abbreviation \text{hs-prefixes} \equiv \\
\text{hs-noop-prefixes} \cup \{ "\text{mark-loop-get-roots}, "\text{mark-loop-get-work}" \}

\begin{locset-definition}
\text{hs-init-loop-locs} = (\bigcup l \in \text{hs-prefixes}. \text{prefixed } (l @ "\text{-init-loop}"))
\end{locset-definition}

\begin{locset-definition}
\text{hs-done-loop-locs} = (\bigcup l \in \text{hs-prefixes}. \text{prefixed } (l @ "\text{-done-loop}"))
\end{locset-definition}

\begin{locset-definition}
\text{hs-done-locs} = (\bigcup l \in \text{hs-prefixes}. \text{prefixed } (l @ "\text{-done}"))
\end{locset-definition}

\begin{locset-definition}
\text{hs-none-pending-locs} = - (\text{hs-init-loop-locs} \cup \text{hs-done-locs})
\end{locset-definition}

\begin{locset-definition}
\text{hs-in-sync-locs} = \\
\quad - (\bigcup l \in \text{hs-prefixes}. \text{prefixed } (l @ "\text{-init}")) \cup \text{hs-done-locs}) \\
\quad \cup (\bigcup l \in \text{hs-prefixes}. \{l @ "\text{-init-type}"\})}
\end{locset-definition}
locset-definition $hs$-out-of-sync-locs =
$(\bigcup l \in \textnormal{hs-prefixes}. \{l \oplus "\textnormal{-init-muts}"\})$

locset-definition $hs$-mut-in-muts-locs =
$(\bigcup l \in \textnormal{hs-prefixes}. \{l \oplus "\textnormal{-init-loop-set-pending"}, l \oplus "\textnormal{-init-loop-done}"\})$

locset-definition $hs$-init-loop-done-locs =
$(\bigcup l \in \textnormal{hs-prefixes}. \{l \oplus "\textnormal{-init-loop-done}"\})$

locset-definition $hs$-init-loop-not-done-locs =
$(\textnormal{hs-init-loop-locs} - (\bigcup l \in \textnormal{hs-prefixes}. \{l \oplus "\textnormal{-init-loop-done}"\}))$

inv-definition (in gc) handshake-invL :: ("field", "mut", "ref") gc-pred where
handshake-invL =
(atS-gc hs-noop-locs (sys-handshake-type = ⟨ht-NOOP⟩)
∧ atS-gc hs-get-roots-locs (sys-handshake-type = ⟨ht-GetRoots⟩)
∧ atS-gc hs-get-work-locs (sys-handshake-type = ⟨ht-GetWork⟩)
∧ atS-gc hs-mut-in-muts-locs (gc-mut ∈ gc-muts)
∧ atS-gc hs-init-loop-locs (∀ m. −((m) ∈ gc-muts) → sys-handshake-pending m
∧ (sys-handshake-type = ⟨ht-NOOP⟩)
∧ atS-gc hs-init-loop-done-locs (∀ m. (m) ∈ gc-muts → ¬(sys-handshake-pending m)
∧ ¬(sys-handshake-type = ⟨ht-NOOP⟩)
∧ atS-gc hs-init-loop-not-done-locs (∀ m. (m) ∈ gc-muts → (sys-handshake-pending m)
∧ (sys-handshake-type = ⟨ht-NOOP⟩)
∧ atS-gc hs-done-locs (∀ m. sys-handshake-pending m ∨ sys-handshake-type = ⟨ht-NOOP⟩)
∧ atS-gc hs-done-loop-locs (∀ m. −((m) ∈ gc-muts) → (sys-handshake-pending m)
∧ (sys-handshake-type = ⟨ht-NOOP⟩)
∧ atS-gc hs-none-pending-locs (∀ m. −(sys-handshake-pending m))
∧ atS-gc hs-in-sync-locs (∀ m. sys-ghost-handshake-in-sync m)
∧ atS-gc hs-out-of-sync-locs (∀ m. −(sys-handshake-pending m)
∧ (sys-ghost-handshake-type = ⟨hp-Idle⟩)
∧ atS-gc hp-Idle-locs (sys-ghost-handshake-phase = ⟨hp-Idle⟩)
∧ atS-gc hp-IdleInit-locs (sys-ghost-handshake-phase = ⟨hp-IdleInit⟩)
∧ atS-gc hp-InitMark-locs (sys-ghost-handshake-phase = ⟨hp-InitMark⟩)
∧ atS-gc hp-IdleMark Sweep-locs (sys-ghost-handshake-phase = ⟨hp-IdleMarkSweep⟩)
∧ atS-gc hp-Mark-locs (sys-ghost-handshake-phase = ⟨hp-Mark⟩))$

Local handshake phase invariant for the mutators.

locset-definition mut-no-pending-mutates-locs =
(preinned "hs-noop" − \{"hs-noop", "hs-noop-mfence"\})
∪ (preinned "hs-get-roots" − \{"hs-get-roots", "hs-get-roots-mfence"\})
∪ (preinned "hs-get-work" − \{"hs-get-work", "hs-get-work-mfence"\})

inv-definition (in mut-m) handshake-invL :: ("field", "mut", "ref") gc-pred where
handshake-invL =
(atS-mut (preinned "hs-noop") (sys-handshake-type = ⟨ht-NOOP⟩)
∧ atS-mut (preinned "hs-get-roots") (sys-handshake-type = ⟨ht-GetRoots⟩)
∧ atS-mut (preinned "hs-get-work") (sys-handshake-type = ⟨ht-GetWork⟩)
∧ atS-mut no-pending-mutate-locs (LIST-NULL (iso-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 where

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 where

phase-rel ≡

\{ (in-sync, hp, ph, []) | in-sync hp ph. handshake-phase-rel hp in-sync ph \}
\cup \{ {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 where

\*phase-rel-inv* = (\(∀ m. \ sys-ghost-handshake-in-sync m\) \otimes \ sys-ghost-handshake-phase \otimes gc-phase \otimes sys-phase \otimes tso-pending-phase gc ∈ \{phase-rel\})

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

**locset-definition** no-pending-phase-locs :: location set where

no-pending-phase-locs ≡

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

**inv-definition** (in gc) phase-invL :: (field, mut, ref) gc-pred where

\*phase-invL* =

\( \text{atS-gc idle-locs} \quad \text{gc-phase} = \langle \text{ph-Idle} \rangle \)
\& \( \text{atS-gc init-locs} \quad \text{gc-phase} = \langle \text{ph-Init} \rangle \)
\& \( \text{atS-gc mark-locs} \quad \text{gc-phase} = \langle \text{ph-Mark} \rangle \)
\& \( \text{atS-gc sweep-locs} \quad \text{gc-phase} = \langle \text{ph-Sweep} \rangle \)
\& \( \text{atS-gc no-pending-phase-locs} \ \text{(LIST-NULL (tso-pending-phase gc))} \)

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

\*fM-rel* =

\{ (in-sync, hp, fM, fM, [], l) | fM hp in-sync l. hp = hp-Idle → ¬in-sync \}
\cup \{ (in-sync, hp-Idle, fM, fM', [], l) | fM fM' in-sync l. in-sync \}
\cup \{ (in-sync, hp-Idle, ¬fM, fM, [mw-fM (¬fM)], False) | fM in-sync. in-sync \}

**definition** fM-rel-inv :: (field, mut, ref) lsts-pred where

\*fM-rel-inv* = (\(∀ m. \ sys-ghost-handshake-in-sync m\) \otimes \ sys-ghost-handshake-phase \otimes gc-fM \otimes sys-fM \otimes tso-pending-fM gc \otimes \ sys-mem-lock = \langle \text{Some gc} \rangle \) ∈ \{fM-rel\})

**definition** fA-rel :: (bool × handshake-phase × gc-mark × gc-mark × (field, ref) mem-write-action list) set where

\*fA-rel* =

\{ (in-sync, hp-Idle, fA, fM, []) | fA fM in-sync. ¬in-sync → fA = fM \}
\cup \{ (in-sync, hp-IdleInit, fA, ¬fA, []) | fA in-sync. True \}
\cup \{ (in-sync, hp-InitMark, fA, ¬fA, [mw-fA (¬fA)]) | fA in-sync. in-sync \}
\[ \begin{align*}
& \cup \{ \text{in-sync, hp-InitMark}, \ fA, \ fM, [\] } | fA \ fM \ \text{in-sync.} \ \neg\text{in-sync } \rightarrow fA \neq fM \} \\
& \cup \{ \text{in-sync, hp-Mark}, \ fA, \ fA, [] \} | fA \ \text{in-sync.} \ \text{True} \} \\
& \cup \{ \text{in-sync, hp-IdleMarkSweep, fA, fA, []} | fA \ \text{in-sync.} \ \text{True} \} \\
\end{align*} \]

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

\( fA\text{-rel-inv} = ((\forall m. \ \text{sys-ghost-handshake-in-sync} \ m) \ \otimes \ \text{sys-ghost-handshake-phase} \ \otimes \ \text{sys-fA} \ \otimes \ \text{gc-fM} \ \otimes \ \text{tso-pending-fA} \ gc \in (fA\text{-rel})) \)

**Locset Definition** \( fM\text{-eq-locs} = (- \ {"idle-write-fM", "idle-flip-noop-mfence"}) \)

**Locset Definition** \( fM\text{-tso-empty-locs} = (- \ {"idle-flip-noop-mfence"}) \)

**Locset Definition** \( fA\text{-tso-empty-locs} = (- \ {"mark-noop-mfence"}) \)

**Locset Definition** \( fA\text{-eq-locs} \equiv \{ "idle-read-fM", "idle-invert-fM" \} \)

**Locset Definition** \( fA\text{-neq-locs} \equiv \{ "idle-phase-init", "mark-read-fM", "mark-write-fA" \} \)

**Inv-definition** \( (\text{in gc}) fM\text{-fA-invL} :: (\text{field, mut, ref}) \) \( gc\text{-pred} \) where

\( fM\text{-fA-invL} = \)

\( (\text{atS-gc fM-eq-locs} \ (\text{sys-fM} = gc\text{-fM}) \land \text{at-gc "idle-write-fM"} \ (\text{sys-fM} \neq gc\text{-fM}) \land \text{at-gc "idle-flip-noop-mfence"} \ (\text{sys-fM} \neq gc\text{-fM} \rightarrow (\neg (\text{LIST-NUL}(\text{tso-pending-fM} \ gc)))) \land \text{atS-gc fM-tso-empty-locs} \ (\text{LIST-NUL}(\text{tso-pending-fM} \ gc)) \land \text{atS-gc fA-eq-locs} \ (\text{sys-fA} = gc\text{-fM}) \land \text{atS-gc fA-neq-locs} \ (\text{sys-fA} \neq gc\text{-fM}) \land \text{at-gc "mark-noop-mfence"} \ (\text{sys-fA} \neq gc\text{-fM} \rightarrow (\neg (\text{LIST-NUL}(\text{tso-pending-fA} \ gc)))) \land \text{atS-gc fA-tso-empty-locs} \ (\text{LIST-NUL}(\text{tso-pending-fA} \ gc)))) \)

### 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** \( \text{marked} :: \text{ref} \Rightarrow (\text{field, mut, ref}) \) \( \text{lsts-pred} \) where

\( \text{marked} \ r \ s \equiv \text{obj-at} (\lambda \text{obj}. \ \text{obj-mark obj} = \text{sys-fM} \ s) \ r \ s \)

**Abbreviation** \( \text{white} :: \text{ref} \Rightarrow (\text{field, mut, ref}) \) \( \text{lsts-pred} \) where
white r s \equiv \text{obj-at (}\lambda \text{obj. obj-mark obj = } (\neg \text{sys-fM s})\text{)} r s

definition WL :: 'mut process-name \Rightarrow ('field, 'mut, 'ref) lsts \Rightarrow 'ref set where
WL p \equiv \lambda s. \ W (s p) \cup \text{ghost-honorary-grey (s p)}

definition grey :: 'ref \Rightarrow ('field, 'mut, 'ref) lsts-pred where
grey r \equiv \exists p. (r) \in WL p

definition black :: 'ref \Rightarrow ('field, 'mut, 'ref) lsts-pred where
black r \equiv \text{marked r} \land \neg (\text{grey r})

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.

abbreviation write-refs :: ('field, 'ref) mem-write-action \Rightarrow 'ref set where
write-refs w \equiv \text{case w of mw-Mutate r f r' \Rightarrow \{r\} \cup Option.set-option r'} |. \text{-} \Rightarrow \{\}

definition (in mut-m) tso-write-refs :: ('field, 'mut, 'ref) lsts \Rightarrow 'ref set where
tso-write-refs = (\lambda s. \bigcup w \in \text{set (sys-mem-write-buffers (mutator m) s)}). \text{write-refs w}

definition (in mut-m) reachable :: 'ref \Rightarrow ('field, 'mut, 'ref) lsts-pred where
reachable y = (\exists x. (x) \in \text{mut-roots} \cup \text{mut-ghost-honorary-root} \cup \text{tso-write-refs}
\land x \text{ reaches y})

definition grey-reachable :: 'ref \Rightarrow ('field, 'mut, 'ref) lsts-pred where
grey-reachable y = (\exists g. \text{grey g} \land g \text{ reaches y})

definition valid-refs-inv :: ('field, 'mut, 'ref) lsts-pred where
valid-refs-inv = (\forall x. (\exists m. (\text{mut-m, reachable m x}) \lor \text{grey-reachable x}) \rightarrow \text{valid-ref x})

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 depend on disjointness; we include it as proof that the single-threading of grey objects in the implementation is sound.

definition valid-W-inv :: ('field, 'mut, 'ref) lsts-pred where
valid-W-inv = (\forall p q r fl.
(r in-W p \lor (\text{sys-mem-lock} \neq (\text{Some p}) \land r \text{ in-ghost-honorary-grey p}) \rightarrow \text{marked r})
\land ((p \neq q) \rightarrow \neg((r) \in WL p \land (r) \in WL q))
\land (\neg(r \text{ in-ghost-honorary-grey p} \land r \text{ in-W q}))
\land (\text{EMPTY sys-ghost-honorary-grey})
\land (\text{tso-pending-write p (mw-Mark r fl})
\rightarrow (\lfloor fl \rfloor = \text{sys-fM}
\land r \text{ in-ghost-honorary-grey p}
\land \text{tso-locked-by p}
\land \text{white r}
\land \text{tso-pending-mark p = (\lfloor [mw-Mark r fl] \rfloor)})

3.8 Mark Object

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

Each use needs to provide extra facts to justify validity of references, etc. We do not include a post-condition for mark-object-fn here as it is different at each call site.
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') → LSTP P

abbreviation (input) p-en-cond P ≡ p-ph-enabled → P
abbreviation (input) p-valid-ref ≡ ¬(NULL p-ref) ∨ 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 ≠ p-mark ∨ p-tso-no-pending-mark) → marked $ p-the-ref)
  ∧ (tso-pending-mark p ∈ (λs. {[], [mw-Mark (p-the-ref s) (p-fM s)]}))

abbreviation (input)
  p-mark-inv ≡ ¬(NULL p-mark)
  ∧ ((λs. obj-at (λobj. Some (obj-mark obj) = p-mark s) (p-the-ref s) s)
  ∨ 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)

abbreviation (input) p-valid-fM ≡ p-fM = sys-fM

abbreviation (input)
  p-ghg-eq-ref ≡ p-ghost-honorary-grey = pred-singleton (the ∘ p-ref)
abbreviation (input)
  p-ghg-inv ≡ If p-cas-mark = p-mark Then p-ghg-eq-ref Else EMPTY p-ghost-honorary-grey

definition mark-object-invL :: ('field, 'mut, 'ref) gc-pred where
  mark-object-invL =
    mark-object-invL =
    (at-p "-.mo-null"  (True)
    ∧ at-p "-.mo-mark"  (p-valid-ref)
    ∧ at-p "-.mo-fM"   (p-valid-ref ∨ p-en-cond (p-mark-inv))
    ∧ at-p "-.mo-mtest" (p-valid-ref ∨ p-en-cond (p-mark-inv ∨ p-valid-fM))
    ∧ at-p "-.mo-phase" (p-valid-ref ∨ p-mark ≠ Some ∘ p-fM ∨ p-en-cond (p-mark-inv ∨ p-valid-fM))
    ∧ at-p "-.mo-ptest" (p-valid-ref ∨ p-mark ≠ Some ∘ p-fM ∨ p-en-cond (p-mark-inv ∨ p-valid-fM))
    ∧ at-p "-.mo-co-lock" (p-valid-ref ∨ p-mark-inv ∨ p-valid-fM ∨ p-mark ≠ Some ∘ p-fM ∨ p-tso-no-pending-mark)
    ∧ at-p "-.mo-co-cmark" (p-valid-ref ∨ p-mark-inv ∨ p-valid-fM ∨ p-mark ≠ Some ∘ p-fM ∨ p-tso-no-pending-mark)
    ∧ at-p "-.mo-co-ctest" (p-valid-ref ∨ p-mark-inv ∨ p-valid-fM ∨ p-mark ≠ Some ∘ p-fM ∨ p-cas-mark-inv ∧
    p-tso-no-pending-mark)
The uses of mark-object-fn in the GC and during the root marking are straightforward.

**interpretation** gc-mark: mark-object gc "mark-loop" ⟨True⟩
by standard (simp add: eq-imp-def)

**lemmas** gc-mark-mark-object-invL-def2[inv] = gc-mark.mark-object-invL-def[simplified]

**interpretation** mut-get-roots: mark-object mutator m "hs-get-roots-loop" ⟨True⟩ for m
by standard (simp add: eq-imp-def)

**lemmas** mut-get-roots-mark-object-invL-def2[inv] = mut-get-roots.mark-object-invL-def[simplified]

The most interesting cases are the two asynchronous uses of 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** mut-store-del: mark-object mutator m "store-del" mut-m.mut-ghost-handshake-phase m ≠ ⟨hp-Idle⟩ for m
by standard (simp add: eq-imp-def)

**lemmas** mut-store-del-mark-object-invL-def2[inv] = mut-store-del.mark-object-invL-def[simplified]

**interpretation** mut-store-ins: mark-object mutator m "store-ins" mut-m.mut-ghost-handshake-phase m ≠ ⟨hp-Idle⟩ for m
by standard (simp add: eq-imp-def)

**lemmas** mut-store-ins-mark-object-invL-def2[inv] = mut-store-ins.mark-object-invL-def[simplified]

Local invariant for the mutator’s uses of mark-object.

**locset-definition** mut-hs-get-roots-loop-locs = prefixed "hs-get-roots-loop"

**locset-definition** mut-hs-get-roots-loop-mo-locs = prefixed "hs-get-roots-loop-mo" ∪ {"hs-get-roots-loop-done"}

**abbreviation** mut-async-mark-object-prefixes ≡ { "store-del", "store-ins" }

**locset-definition** mut-hs-not-hp-Idle-locs =
(∪ pref∈mut-async-mark-object-prefixes.
     (∪ l∈{"mo-co-lock", "mo-co-cmark", "mo-co-ctest", "mo-co-mark", "mo-co-unlock", "mo-co-won", "mo-co-W"}.
      {pref @ "." @ l}))

**locset-definition** mut-async-mo-ptest-locs =
(∪ pref∈mut-async-mark-object-prefixes. {pref @ "-mo-ptest"})

**locset-definition** mut-mo-ptest-locs =
(∪ pref∈mut-async-mark-object-prefixes. {pref @ "-mo-ptest"})

**locset-definition** mut-mo-valid-ref-locs =
(prefixed "store-del" ∪ prefixed "store-ins" ∪ { "deref-del", "lop-store-ins" })

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 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 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: it is key that after the "init-noop" handshake there are no pending white insertions (mutations that insert unmarked references). This ensures the deletion barrier does its job.

**locset-definition**

ghost-honorary-grey-empty-locs ≡

- (∪ pref ∈ {"mark-loop", "hs-get-roots-loop", "store-del", "store-ins"}
  ∪ l ∈ {"mo-co-unlock", "mo-co-won", "mo-co-W"}. {pref @ "-" @ l})

**locset-definition**

ghost-honorary-root-empty-locs ≡

- (prefixed "store-del" ∪ {"lop-store-ins"} ∪ prefixed "store-ins")

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

mark-object-invL =

( atS-mut mut-hs-get-roots-loop-locs (mut-refs ⊆ mut-roots ∧ (∀ r. (r) ∈ mut-roots − mut-refs → marked r))

∧ atS-mut mut-hs-get-roots-loop-mo-locs (¬(NULL mut-ref) ∧ mut-the-ref ∈ mut-roots)

∧ at-mut "hs-get-roots-loop-done" (marked $ mut-the-ref)

∧ at-mut "hs-get-roots-loop-mo-ptest" (mut-phase ≠ ⟨ph-Idle⟩)

∧ at-mut "hs-ghost-handshake-phase" (∀ r. (r) ∈ mut-roots → marked r)

∧ atS-mut mut-mo-valid-ref-locs (¬(NULL mut-new-ref) → mut-the-ref ∈ mut-roots)

∧ (mut-refs ∈ mut-roots)

∧ at-mut "store-del-mo-null" (¬(NULL mut-ref) → mut-the-ref ∈ mut-ghost-honorary-root)

∧ atS-mut (prefixed "store-del") (mut-phase ∈ {"store-del-mo-null"}) (mut-the-ref ∈ mut-ghost-honorary-root)

∧ atS-mut (prefixed "store-ins") (mut-ref = mut-new-ref)

∧ atS-mut (suffixed "-mo-ptest") (mut-phase ≠ ⟨ph-Idle⟩ → mut-ghost-handshake-phase ≠ ⟨hp-Idle⟩)

∧ atS-mut mut-hs-not-hp-Idle-locs (mut-ghost-handshake-phase ≠ ⟨hp-Idle⟩)

∧ atS-mut mut-hs-get-roots-loop-mo-pptest-locs (mut-phase = ⟨ph-Idle⟩ → (mut-ghost-handshake-phase ∈ ⟨hp-Idle, hp-IdleInit⟩)

∧ (mut-ghost-handshake-phase = ⟨hp-IdleMarkSweep⟩ ∧ sys-phase = ⟨ph-Idle⟩))

∧ atS-mut ghost-honorary-grey-empty-locs (EMPTY mut-ghost-honorary-grey)

 insertion barrier

∧ at-mut "store-ins"

∨ (mut-ghost-handshake-phase ∈ ⟨hp-IdleMark, hp-Mark⟩)

∧ ¬(NULL mut-new-ref)

∨ (mut-ghost-handshake-phase = ⟨hp-IdleMarkSweep⟩ ∧ sys-phase ≠ ⟨ph-Idle⟩)

→ marked $ mut-the-new-ref

deletion barrier

∧ atS-mut (prefixed "store-del-mo" ∪ {"lop-store-ins"})

( mut-ghost-handshake-phase = ⟨hp-Mark⟩

∨ (mut-ghost-handshake-phase = ⟨hp-IdleMarkSweep⟩ ∧ sys-phase ≠ ⟨ph-Idle⟩))

∧ (∀ ls. ∀ opt-r'. ¬tso-pending-write (mutator m) (mw-Mutate (mut-refs r s) s)

(mut-field s) opt-r' ) s)

→ (λ ls. obj-at-field-on-heap (λ r. mut-ref s = Some r ∨ marked r s) (mut-refs r s)

s) (mut-field s) s))

∧ at-mut "lop-store-ins"

( mut-ghost-handshake-phase = ⟨hp-Mark⟩

∨ (mut-ghost-handshake-phase = ⟨hp-IdleMarkSweep⟩ ∧ sys-phase ≠ ⟨ph-Idle⟩))

∧ ¬(NULL mut-ref)

→ marked $ mut-the-ref

∧ atS-mut (prefixed "store-ins")

( mut-ghost-handshake-phase = ⟨hp-Mark⟩

∨ (mut-ghost-handshake-phase = ⟨hp-IdleMarkSweep⟩ ∧ sys-phase ≠ ⟨ph-Idle⟩))
We now show that 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 =
  (∀ f. (f) ∈ (- gc-field-set) → (λ r. marked r s) (gc-tmp-ref s) f s))

locset-definition
  obj-fields-marked-locs ≡
  ∪ prefixed "mark-loop-mo"

inv-definition (in gc) obj-fields-marked-invL :: (field, 'mut', 'ref') gc-pred where
  obj-fields-marked-invL ≡
  (atS-gc obj-fields-marked-locs
   (obj-fields-marked-inv ∧ gc-tmp-ref ∈ gc-W)
   ∧ atS-gc (prefixed "mark-loop-mo")
   (∀ r. (gc-the-ref r) reaches y s) → valid-ref y)
  ∧ at-ge "mark-loop-fields" (gc-tmp-ref ∈ gc-W)
  ∧ at-ge "mark-loop-mark-field-done" (∃ gc-ref s = Some r ∨ marked r s) (gc-tmp-ref s)
  ∧ at-ge "mark-loop-blacken" (EMPTY gc-field-set)
  ∧ atS-gc ghost-honorary-grey-empty-locs (EMPTY gc-ghost-honorary-grey)

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 ∧ white y

definition strong-tricolour-inv :: (field, 'mut', 'ref') lsts-pred where
  strong-tricolour-inv = (∀ b w. black b →¬(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 ≡ λ s. (λ x y. (x points-to-white y s)** x y)

definition grey-protecs-white :: 'ref ⇒ 'ref ⇒ (field, 'mut', 'ref') lsts-pred (infix grey'.-protecs'-white 51) where
  g grey-protecs-white w = (grey g ∧ g has-white-path-to w)

definition weak-tricolour-inv :: (field, 'mut', 'ref') lsts-pred where
  weak-tricolour-inv =
  (∀ b w. black b ∧ b points-to-white w → (∃ g. g grey-protecs-white w))

lemma strong-tricolour-inv s ⇒ weak-tricolour-inv s
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 \(\Rightarrow\) ('field, 'mut, 'ref) lsts-pred where
in-snapshot \(r = (\text{black } r \lor (\exists \ g. \ g \text{ grey-protects-white } r))\)

**Definition** (in mut-m) reachable-snapshot-inv :: ('field, 'mut, 'ref) lsts-pred where
reachable-snapshot-inv = (\(\forall r. \text{reachable } r \rightarrow \text{in-snapshot } r\))

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 \(\Rightarrow\) ('field, 'mut, 'ref) lsts-pred where
marked-insertion \(w \equiv \lambda s. \text{case } w \text{ of } \text{mw-Mutate } r f (\text{Some } r') \Rightarrow \text{marked } r'\ s \ | \ - \Rightarrow \text{True}\)

**Definition** (in mut-m) marked-insertions :: ('field, 'mut, 'ref) lsts-pred where
marked-insertions = (\(\forall w. \text{tso-pending-write } (\text{mutator } m) w \rightarrow \text{marked-insertion } w\))

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

**Definition** (in mut-m) marked-deletions :: ('field, 'mut, 'ref) lsts-pred where
marked-deletions = (\(\forall w. \text{tso-pending-write } (\text{mutator } m) w \rightarrow \text{marked-deletion } w\))

Finally, in some phases the heap is somewhat monochrome.

**Definition** black-heap :: ('field, 'mut, 'ref) lsts-pred where
black-heap = (\(\forall r. \text{valid-ref } r \rightarrow \text{black } r\))

**Definition** white-heap :: ('field, 'mut, 'ref) lsts-pred where
white-heap = (\(\forall r. \text{valid-ref } r \rightarrow \text{white } r\))

**Definition** no-black-refs :: ('field, 'mut, 'ref) lsts-pred where
no-black-refs = (\(\forall r. \neg(\text{black } r)\))

**Definition** no-grey-refs :: ('field, 'mut, 'ref) lsts-pred where
no-grey-refs = (\(\forall r. \neg(\text{grey } r)\))

### 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 mut-m) mutator-phase-inv-aux :: handshake-phase \(\Rightarrow\) ('field, 'mut, 'ref) lsts-pred where
mutator-phase-inv-aux \(hp\text{-Idle} = (\text{True})\)
\(| \text{mutator-phase-inv-aux } hp\text{-IdleInit} = \neg\text{black-refs}\)
\(| \text{mutator-phase-inv-aux } hp\text{-InitMark} = \text{marked-insertions}\)
\(| \text{mutator-phase-inv-aux } hp\text{-Mark} = (\text{marked-insertions} \land \text{marked-deletions})\)
\(| \text{mutator-phase-inv-aux } hp\text{-IdleMarkSweep} = (\text{marked-insertions} \land \text{marked-deletions} \land \text{reachable-snapshot-inv})\)

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

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 ⇒ ('field', 'mut', 'ref') lsts-pred where
sys-phase-inv-aux hp-Idle = ( (If sys-fA = sys-fM Then black-heap Else white-heap) ∧ no-grey-refs )
sys-phase-inv-aux hp-IdleInit = no-black-refs
sys-phase-inv-aux hp-InitMark = (sys-fA ≠ sys-fM → no-black-refs)
sys-phase-inv-aux hp-Mark = ⟨True⟩
sys-phase-inv-aux hp-IdleMarkSweep = ( (sys-phase = ⟨ph-Idle⟩ ∨ tso-pending-write gc (mw-Phase ph-Idle)) → 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

3.11 Mutator proofs

lemma (in mut-m) reachable-snapshot-inv-hs-get-roots-done[simp]:
assumes sti: strong-tricolour-inv s
assumes m: ∀ r ∈ mut-roots s. marked r s
assumes ghr: mut-ghost-honorary-root s = {}
assumes t: tso-pending-mutate (mutator m) s = []
assumes vri: valid-refs-inv s
shows reachable-snapshot-inv
  (s(mutator m := s (mutator m)[W := {}, ghost-handshake-phase := ghp'],
   sys := s sys[handshake-pending := hp', W := sys-W s ∪ mut-W s, ghost-handshake-in-sync := in'])))
(is reachable-snapshot-inv ?s')

proof (rule, clarsimp)
fix r assume reachable r s
then show in-snapshot r ?s'
proof (induct rule: reachable-induct)
case (root x) with m show ?case
  apply (clarsimp simp: in-snapshot-def simp del: fun-upd-apply)
  apply (auto dest: marked-imp-black-or-grey)
done
next
case (ghost-honorary-root x) with ghr show ?case by simp
next
case (tso-root x) with t show ?case
  apply (clarsimp simp: filter-empty-conv tso-write-refs-def)
  apply (force split: mem-write-action.splits)
done
next
case (reaches x y)
from reaches vri have valid-ref x s valid-ref y s by auto
with reaches sti vri show ?case
  apply (clarsimp simp: in-snapshot-def simp del: fun-upd-apply)
  apply (elim disjE)
  apply (clarsimp simp: strong-tricolour-inv-def)
  apply (drule spec[where x=x])
  apply clarsimp
  apply (auto dest!: marked-imp-black-or-grey)[1]
  apply (cases white y s)
  apply (auto dest: grey-protects-whiteE dest!: marked-imp-black-or-grey)
done
qed

lemma (in mut-m) reachable-snapshot-inv-hs-get-work-done[simp]:
reachable-snapshot-inv s
⇒ reachable-snapshot-inv

(s(mutator m := s (mutator m))(W := {})),
sys := s sys(handshake-pending := (handshake-pending (s sys))(m := False), W := sys-W s ∪ mut-W s,
ghost-handshake-in-sync := (ghost-handshake-in-sync (s sys))(m := True)))

by (simp add: reachable-snapshot-inv-def in-snapshot-def grey-protects-white-def)

lemma (in mut-m) reachable-write-enqueue[simp]:
reachable r (s(sys := s sys(mem-write-buffers := (mem-write-buffers (s sys))(mutator m := sys-mem-write-buffers (mutator m) s @ [w])))\))
⇒ reachable r s ∨ (∃ x. x ∈ write-refs w ∧ (x reaches r) s)
by (auto simp: reachable-def iso-write-refs-def)

lemma no-black-refs-mo-co-mark[simp]:
[ ghost-honorary-grey (s p) = {} ]; white r s ]
⇒ no-black-refs (s(p := s p[ghost-honorary-grey := {r} ])) \no-black-refs s
by (auto simp: no-black-refs-def)

lemma (in mut-m) reachable-snapshot-inv-mo-co-mark[simp]:
[ ghost-honorary-grey (s p) = {} ]; reachable-snapshot-inv s ]
⇒ reachable-snapshot-inv (s(p := s p[ ghost-honorary-grey := {r} ])) \no-black-refs s
by (auto simp: in-snapshot-def reachable-snapshot-inv-def)

lemma (in mut-m) no-black-refs-alloc[simp]:
[ heap (s sys) r′ = None; no-black-refs s ]
⇒ no-black-refs (s(mutator m := s (mutator m)(roots := roots'))
sys := s sys(heap := sys-heap s(r′ → (obj-mark = fl, obj-fields = Map.empty)))
⇒ fl ≠ sys-M s ∨ grey r′ s
by (auto simp: no-black-refs-def)

lemma (in mut-m) reachable-snapshot-inv-load[simp]:
[ reachable-snapshot-inv s; sys-read (mutator m) (mr-Ref r f) (s sys) = mv-Ref r f; r ∈ mut-roots s ]
⇒ reachable-snapshot-inv (s(mutator m := s (mutator m)(roots := mut-roots s ∪ Option.set-option r′ )))
by (simp add: reachable-snapshot-inv-def in-snapshot-def grey-protects-white-def)

lemma (in mut-m) reachable-snapshot-inv-store-ins[simp]:
[ reachable-snapshot-inv s; r ∈ mut-roots s; (∃ r′. opt-r′ = Some r′ ) → the opt-r′ ∈ mut-roots s ]
⇒ reachable-snapshot-inv (s(mutator m := s (mutator m)(ghost-honorary-root := {})),
sys := s sys(mem-write-buffers := (mem-write-buffers (s sys))(mutator m := sys-mem-write-buffers (mutator m) s @ [mw-Mutate r f opt-r′ ])))
apply (clarsimp simp: reachable-snapshot-inv-def in-snapshot-def grey-protects-white-def)
apply (erule_tac x=x in spec)
apply (auto simp: reachable-def)
done

lemma (in mut-m) marked-insertions-store-ins[simp]:
[ marked-insertions s; (∃ r′. opt-r′ = Some r′ ) → marked (the opt-r′ ) s ]
⇒ marked-insertions
(s(mutator m := s (mutator m)(ghost-honorary-root := { })),
sys := s sys
(mem-write-buffers := (mem-write-buffers (s sys))(mutator m := sys-mem-write-buffers (mutator m) s @ [mw-Mutate r f opt-r′ ])))
by (auto simp: marked-insertions-def
split: mem-write-action.splits option.splits)

lemma (in mut-m) marked-deletions-store-ins[simp]:
[ marked-deletions s; obj-at-field-on-heap (λ r′. marked r′ s) r f s ]
⇒ marked-deletions
\[(s (\text{mutator } m := s (\text{mutator } m)) (\text{ghost-honorary-root} := \{\}))],
\[s \equiv s\]
\[(\text{mem-write-buffers} := (\text{mem-write-buffers} (s sys)) (\text{mutator } m := \text{sys-mem-write-buffers} (\text{mutator } m)) \ s @ [\text{mw-Mutate } r f \ \text{opt-r}'])]]\]

by \(\text{auto simp: marked-deletions-def split: mem-write-action.splits option.splits}\)

**lemma in mut-m reachable-snapshot-inv-deref-del[simp]:**
\[[\text{reachable-snapshot-inv } s; \ \text{sys-read} (\text{mutator } m) \ (\text{mr-Ref } r f) \ (s sys) = \text{mv-Ref } \text{opt-r}'; \ r \in \text{mut-roots } s; \ \\
\text{mut-ghost-honorary-root } s = \{\} \ \rightarrow \ \text{reachable-snapshot-inv } (s (\text{mutator } m := s (\text{mutator } m)) (\text{ghost-honorary-root} := \text{Option.set-option } \text{opt-r}', \ ref := \text{opt-r}')]]\)

by \(\text{clarsimp simp: reachable-snapshot-inv-def in-snapshot-def grey-protects-white-def}\)

**lemma in mut-m mutator-phase-inv[intro]:**
\[[\text{handshake-invl} \ \\
\text{mark-object-invl} \ \\
\text{mut-get-roots.mark-object-invl } m \ \\
\text{mut-store-del.mark-object-invl } m \ \\
\text{mut-store-ins.mark-object-invl } m \ \\
\text{LSTP } (\text{fA-rel-inv } \text{fm-rel-inv } \text{handshake-phase-inv } \text{mutators-phase-inv } \text{phase-rel-inv } \text{strong-tricolour-inv} \ \\
\text{sys-phase-inv } \text{valid-refs-inv } \text{valid-W-inv}) \] \]

\[m \text{ mutator m} \]

by \(\text{clarsimp simp: handshake-phase-invD simp: fA-rel-inv-def fm-rel-inv-def; \ simp add: mutator-phase-inv-aux-case split: handshake-phase.splits if-splits}\)

**subgoal:**
apply \(\text{drule-tac } x=m \text{ in spec}\)
apply \(\text{clarsimp simp: fM-rel-def hp-step-rel-def}\)
apply \(\text{intro conjI impI; simp}\)
apply \(\text{elim disjE; force simp: fA-rel-def}\)
apply \(\text{rule reachable-snapshot-inv-alloc, simp-all}\)
apply \(\text{elim disjE; force simp: fA-rel-def}\)
done

**subgoal for s s'**
apply \(\text{drule-tac } x=m \text{ in spec}\)
apply \(\text{intro conjI impI}\)
apply \text{clarsimp}
apply \(\text{rule marked-deletions-store-ins, assumption}\)
apply \(\text{cases } (\forall \text{opt-r'I. mw-Mutate } (\text{mut-tmp-ref } s \downarrow) (\text{mut-field } s \downarrow) \text{opt-r'I} \notin \text{set (sys-mem-write-buffers} (\text{mutator } m) s \downarrow))\)
apply \text{force}
apply \(\text{force simp: marked-deletions-def}\)
apply \text{clarsimp}
apply \(\text{erule marked-insertions-store-ins}\)
apply \(\text{drule phase-rel-invD}\)
apply \(\text{clarsimp simp: phase-rel-def hp-step-rel-def; elim disjE; fastforce dest: reachable-blackD elim: blackD}\)
apply \text{clarsimp}
apply \(\text{rule marked-deletions-store-ins, assumption}\)
apply \text{clarsimp}
apply \(\text{erule disjE}\)
apply \(\text{drule phase-rel-invD}\)
apply \(\text{clarsimp simp: phase-rel-def}\)
apply \(\text{elim disjE, simp-all}[1]\)
apply \(\text{clarsimp simp: hp-step-rel-def}\)
apply \(\text{clarsimp simp: hp-step-rel-def}\)
apply (case-tac sys-ghost-handshake-phase $s \downarrow$, simp-all)[1]
apply (clarsimp simp: fA-rel-def fM-rel-def)
apply (elim disjE, simp-all)[1]
apply (clarsimp simp: obj-at-field-on-heap-def split: option.splits)
apply (rule conjI)
apply fast
apply clarsimp
apply (frule-tac r = x2a in blackD(1)[OF reachable-blackD], simp-all)[1]
apply (rule-tac x = mut-tmp-ref $s \downarrow$ in reachableE; auto simp: ran-def split: obj-at-splits; fail)
apply (clarsimp simp: obj-at-field-on-heap-def split: option.splits)
apply (rule conjI)
apply fast
apply clarsimp
apply clarsimp
apply clarsimp
apply (frule-tac r = x2a in blackD(1)[OF reachable-blackD], simp-all)[1]
apply (rule-tac x = mut-tmp-ref $s \downarrow$ in reachableE; auto simp: ran-def split: obj-at-splits; fail)
apply (force simp: marked-deletions-def)
done

subgoal for $s \leftrightarrow s'$
apply (drule-tac x = m in spec)
apply (simp add: fA-rel-def fM-rel-def hp-step-rel-def)
apply (cases mut-ghost-handshake-phase $s \downarrow$, simp-all)[1]
apply (erule marked-insertions-store-buffer-empty)
apply (erule marked-deletions-store-buffer-empty)
done

subgoal
apply (drule-tac x = m in spec)
apply (simp add: fA-rel-def fM-rel-def hp-step-rel-def)
done

subgoal
apply (drule-tac x = m in spec)
apply (simp add: fA-rel-def fM-rel-def hp-step-rel-def)
done

done

lemma (in mut-m') mutator-phase-inv[intro]:
notes mut-m.mark-object-invL-def[intv]
notes mut-m.handshake-invL-def[intv]
shows {\hspace{1cm}
\text{handshake-invL} \land \text{mut-m.handshake-invL $m'$} \\
\land \text{mut-m.mark-object-invL $m'$} \\
\land \text{mut-get-roots.mark-object-invL $m'$} \\
\land \text{mut-store-del.mark-object-invL $m'$} \\
\land \text{mut-store-ins.mark-object-invL $m'$} \\
\land \text{LSTP (fA-rel-inv \land fM-rel-inv \land handshake-phase-inv \land mutators-phase-inv \land valid-refs-inv)}}
mutator m'
\}
\text{LSTP mutator-phase-inv }
apply (vcg-jackhammer simp: fA-rel-inv-def fM-rel-inv-def dest!: handshake-phase-invD)
apply (simp-all add: mutator-phase-inv-aux-case split: handshake-phase.splits)

apply (drule spec[where $x=m$])
apply (intro conjI impI)
apply (clarsimp simp: fA-rel-def fM-rel-def hp-step-rel-def)
apply (elim disjE, auto)[1]

apply (rule reachable-snapshot-inv-alloc, simp-all)[1]
apply (clarsimp simp: fA-rel-def fM-rel-def hp-step-rel-def)
apply (elim disjE, auto)[1]

apply (drule spec[where x=m])
apply (clarsimp simp: no-black-refs-def)
apply (clarsimp simp: reachable-snapshot-inv-def in-snapshot-def grey-protects-white-def)

apply (drule spec[where x=m])
apply (clarsimp simp: no-black-refs-def reachable-snapshot-inv-def in-snapshot-def grey-protects-white-def)

done

lemma (in sys) grey-protects-white-dequeue-mark[simp]:
  assumes fl: f️l = sys-fM s
  assumes r ∈ ghost-honorary-grey (s p)
  shows (∃ g. (g grey-protects-white w (s sys:= s sys(heap := (sys-heap s)(r := Option.map-option (obj-mark-update (λ-. f️l)) (sys-heap s r)))) mem-write-buffers := (mem-write-buffers (s sys))(p := ws))))
  ↔ (∃ g. (g grey-protects-white w)) s (is (∃ g. (g grey-protects-white w)) ?s' ↔ ?rhs)

proof
  assume ∃ g. (g grey-protects-white w) ?s'
  then obtain g where (g grey-protects-white w) ?s' by blast
  with assms show ?rhs
    apply (clarsimp simp: grey-protects-white-def)
    apply (rotate-tac −1)
    apply (induct rule: rtranclp.induct)
    apply fastforce
    apply clarsimp
    apply (rename-tac a b c g)
    apply (case-tac white c s)
    apply (rule-tac x=g in exI)
    apply clarsimp
    apply (erule rtranclp.intros)
    apply clarsimp
    apply (auto split: obj-at-splits if-splits)
    done
  next
  assume ?rhs
  then obtain g' where (g' grey-protects-white w) s by blast
  with assms show ∃ g. (g grey-protects-white w) ?s'
    apply (clarsimp simp: grey-protects-white-def)
    apply (rotate-tac −1)
    apply (induct rule: rtranclp.induct)
    apply (fastforce simp: grey-def)
    apply clarsimp
    apply (rename-tac a b c g)
    apply (case-tac c = r)
    apply (clarsimp simp: grey-def)
    apply blast
    apply (rule-tac x=g in exI)
    apply clarsimp
    apply (erule rtranclp.intros)
    apply clarsimp
    done
qed
lemma (in sys) reachable-snapshot-inv-dequeue-mark[simp]:

\[ \text{sys-mem-write-buffers } p \ s = \text{mw-Mark } r \ f \ l \ # \ ws; \ \text{mut-m.reachable-snapshot-inv } m \ s; \ \text{valid-W-inv } s \]

\[ \Rightarrow \ \text{mut-m.reachable-snapshot-inv } m \ (s\{\text{sys := s sys}}\{\text{heap := (sys-heap s)}(r := \text{Option.map-option (obj-mark-update (\lambda \cdot \text{fl}) (sys-heap s r))}, \ \text{mem-write-buffers} := (\text{mem-write-buffers} (s \text{sys}))(p := ws))\}) \]

apply (clarsimp simp; mut-m.reachable-snapshot-inv-def in-snapshot-def)
apply (rename-tac x)
apply (drule-tac x=x in spec)
apply clarsimp
apply (rename-tac x)
apply (drule-tac x=x in spec)
apply (auto split: mem-write-action.splits option.splits obj-at-splits)[1]
apply clarsimp
apply (rename-tac x)
apply (drule-tac x=x in spec)
apply (auto split: mem-write-action.splits option.splits obj-at-splits)[1]
done

lemma (in sys) marked-insertions-dequeue-mark[simp]:

\[ \text{sys-mem-write-buffers } p \ s = \text{mw-Mark } r \ f \ l \ # \ ws; \ \text{mut-m.marked-insertions } m \ s; \ \text{tso-writes-inv } s; \ \text{valid-W-inv } s \]

\[ \Rightarrow \ \text{mut-m.marked-insertions } m \ (s\{\text{sys := s sys}}\{\text{heap := (sys-heap s)}(r := \text{Option.map-option (obj-mark-update (\lambda \cdot \text{fl}) (sys-heap s r))}, \ \text{mem-write-buffers} := (\text{mem-write-buffers} (s \text{sys}))(p := ws))\}) \]

apply (clarsimp simp; mut-m.marked-insertions-def)
apply (cases mutator m = p)
apply clarsimp
apply (rename-tac x)
apply (drule-tac x=x in spec)
apply (auto split: mem-write-action.splits option.splits obj-at-splits)[1]
apply clarsimp
apply (rename-tac x)
apply (drule-tac x=x in spec)
apply (auto split: mem-write-action.splits option.splits obj-at-splits)[1]
done

lemma (in sys) marked-insertions-dequeue-ref[simp]:

\[ \text{sys-mem-write-buffers } p \ s = \text{mw-Mutate } r \ f \ r’ \ # \ ws; \ \text{mut-m.marked-insertions } m \ s \]

\[ \Rightarrow \ \text{mut-m.marked-insertions } m \ (s\{\text{sys := s sys}}\{\text{heap := (sys-heap s)}(r := \text{Option.map-option (\lambda \text{obj. obj}}\{\text{obj-fields := (obj-fields obj}}(f := \text{opt-r’})) (sys-heap s r)), \ \text{mem-write-buffers} := (\text{mem-write-buffers} (s \text{sys}))(p := ws))\}) \]

apply (clarsimp simp; mut-m.marked-insertions-def)
apply (cases mutator m = p)
apply clarsimp
apply (rename-tac x)
apply (drule-tac x=x in spec)
apply (auto split: mem-write-action.splits option.splits obj-at-splits)[1]
apply clarsimp
apply (rename-tac x)
apply (drule-tac x=x in spec)
apply (auto split: mem-write-action.splits option.splits obj-at-splits)[1]
done

lemma points-to-mw-Mutate:

\[(x \points-to \ y) \]

\[ (s\{\text{sys := s sys}}\{\text{heap := (sys-heap s)}(r := \text{Option.map-option (\lambda \text{obj. obj}}\{\text{obj-fields := (obj-fields obj}}(f := \text{opt-r’})) (sys-heap s r)), \ \text{mem-write-buffers} := (\text{mem-write-buffers} (s \text{sys}))(p := ws))\}) \]

\[ \iff \ (r \neq x \land (x \points-to \ y) \ s) \lor (r = x \land \text{valid-ref } r \ s \land (\text{opt-r’} = \text{Some } y \lor (x \points-to \ y) \ s \land \text{obj-at} \]
\(\lambda obj. \exists f'. \text{obj-fields } obj f' = \text{Some } y \land f \neq f'\) \(r\) \(s\))

by (auto simp: ran-def split: obj-at-splits)

**lemma** \((in\ sys)\) grey-protects-white-dequeue-ref[simp]:

assumes \(sb: \text{sys-mem-write-buffers} (\text{mutator } m) s = \text{mw-Mutate } r f \text{ opt-r' } \# \ ws\)

assumes \(mi: \text{mut-m.marked-insertions } m s\)

assumes \(md: \text{mut-m.marked-deletions } m s\)

notes map-option.compositionality[simp] o-def[simp]

shows \((\exists g. (g \text{ grey-protects-white } w) (s(sys := s)\text{sys} \text{heap := (sys-heap s)}(r := \text{Option.map-option}(\lambda obj. \text{obj-fields := (obj-fields obj)}(f := \text{opt-r'})))(\text{sys-heap s r})), \text{mem-write-buffers := (mem-write-buffers (s sys)}(\text{mutator } m := ws))))\)

\[\leftrightarrow (\exists g. (g \text{ grey-protects-white } w) (\text{sys} s \text{grey-protects-white-dequeue-ref}))\]

proof

assume \((\exists g. (g \text{ grey-protects-white } w) ?s')\)

then obtain \(g\) where \((g \text{ grey-protects-white } w) ?s'\) by blast

with \(mi sb\) show \(?rhs\)

apply (clarsimp simp: grey-protects-white-def has-white-path-to-def)

apply (rotate-tac \(-1\))

apply fastforce

apply (auto simp: points-to-mw-Mutate elim: rtranclp.intros(2))

apply (rename-tac a c g)

apply (clarsimp simp: mut-m.marked-insertions-def)

apply (drule-tac \(x=\text{mw-Mutate } r f \text{ Some c}\) in spec)

apply (simp split: obj-at-splits)

don

next

assume \(?rhs\) then show \((\exists g. (g \text{ grey-protects-white } w) ?s')\)

proof(clarsimp)

fix \(g\) assume \((g \text{ grey-protects-white } w) s\)

with \(md sb\) show \(?thesis\)

apply (clarsimp simp: grey-protects-white-def has-white-path-to-def)

apply (rotate-tac \(-1\))

apply fastforce

apply (rename-tac a c g)

apply clarsimp

apply (rename-tac a b c g)

apply case-tac b = \(r\)

defer

apply (auto simp: points-to-mw-Mutate elim: rtranclp.intros(2))[1]

apply clarsimp

apply (subst (asm) \(\exists g\) obj-at-def)

apply (clarsimp simp: ran-def split: option.splits)

apply (rename-tac a c g x2 aa)

apply (case-tac aa = \(f\))

defer

apply (rule-tac \(x=g\) in \(exI\))

apply clarsimp

apply (erule rtranclp.intros)

apply (clarsimp simp: obj-at-splits)[1]

apply clarsimp

apply (clarsimp simp: mut-m.marked-deletions-def)

apply (erule spec)[where \(x=\text{mw-Mutate } r f \text{ opt-r'}\)]

apply (clarsimp simp: obj-at-field-on-heap-def)
apply (simp split: obj-at-splits)
done

qed

lemma (in sys) reachable-snapshot-inv-dequeue-ref [simp]:
fixes s :: ('field, 'mut, 'ref) lsts
assumes sb: sys-mem-write-buffers (mutator m') s = mw-Mutate r f opt-r' # ws
assumes mi: mut-m.marked-insertions m' s
assumes md: mut-m.marked-deletions m' s
assumes rsi: mut-m.reachable-snapshot-inv m s
assumes sti: strong-tricolour-inv s
assumes vri: valid-refs-inv s
notes map-option.compositionality [simp] o-def [simp]
shows mut-m.reachable-snapshot-inv m (s(sys := s sys(heap := (sys.heap s))(r := Option.map-option (λobj. obj(obj-fields := (obj-fields obj))(f := opt-r')))) (sys.heap s r)),
mem-write-buffers := (mem-write-buffers (s sys))(mutator m' := ws))) (is mut-m.reachable-snapshot-inv m ?s')

proof (rule mut-m.reachable-snapshot-invI)
fix y assume y: mut-m.reachable m y ?s'
then have (mut-m.reachable m y s ∨ mut-m.reachable m' y s)
  proof (induct rule: reachable-induct)
  case (root x) with mi md rsi sb show ?case
    apply (clarsimp simp: mut-m.reachable-snapshot-inv-def in-snapshot-def
      simp del: fun-upd-apply)
    apply auto
done
next
  case (ghost-honorary-root x) with mi md rsi sb show ?case
    apply (clarsimp simp: mut-m.reachable-snapshot-inv-def in-snapshot-def
      simp del: fun-upd-apply)
    apply auto
done
next
  case (tso-root x) with mi md rsi sb show ?case
    apply (clarsimp simp: mut-m.reachable-snapshot-inv-def in-snapshot-def
      simp del: fun-upd-apply)
    apply (clarsimp split: if-splits)
    apply (rename-tac xa)
    apply (case-tac xa simp-all)[1]
    apply (rename-tac ref field option)
    apply (clarsimp simp: mut-m.marked-deletions-def mut-m.marked-insertions-def)
    apply (drule-tac x=mw-Mutate ref field option in spec)
    apply (drule-tac x=mw-Mutate ref field option in spec)
    apply clarsimp
    apply (frule spec[where x=x])
    apply (subgoal-tac mut-m.reachable m x s)
    apply force
    apply (rule reachableI(2))
    apply (force simp: mut-m.tso-write-refs-def)
    apply auto
done
next
  case (reaches x y)
  from reaches sb have y: mut-m.reachable m y s ∨ mut-m.reachable m' y s
    apply (clarsimp simp: points-to-mw-Mutate mut-m.reachable-snapshot-inv-def in-snapshot-def)
    apply (elim disjE, (force dest!: reachableE mutator-reachable-tso)+)[1]
moreover

from \texttt{y vri have valid-ref y s by auto}

with \texttt{reaches mi md rsi sb sti y have (black y s \lor (\exists x. (x grey-protects-white y s))}

apply (clarsimp simp: mut-m.reachable-snapshot-inv-def in-snapshot-def
simp del: fun-upd-apply)
apply clarsimp
apply (drule spec[where \texttt{x=y[}])
apply (clarsimp simp: points-to-mw-Mutate mut-m.marked-insertions-def mut-m.marked-deletions-def)
apply (drule spec[where \texttt{x=mw-Mutate r f opt-r}])+
apply clarsimp
apply (elim disjE, simp-all)
apply (force dest!: reachableE)
apply (force dest!: reachableE)

apply clarsimp
apply (drule (3) strong-tricolour-invD)
apply (metis (no-types) grey-protects-whiteI marked-imp-black-or-grey(1))

apply clarsimp
apply (cases white y s)
apply (drule (2) grey-protects-whiteE)
apply blast
apply (force simp: black-def split: obj-at-splits)
apply clarsimp
apply (elim disjE, simp-all)
apply (force simp: black-def)
apply clarsimp
apply (drule (3) strong-tricolour-invD)
apply (force simp: black-def)

apply clarsimp
apply (elim disjE, simp-all)
apply (force simp: black-def)
apply clarsimp
apply (cases white y s)
apply (drule (2) grey-protects-whiteE)
apply blast
apply (force simp: black-def split: obj-at-splits)
apply clarsimp
apply (elim disjE, simp-all)
apply (force simp: black-def)
apply clarsimp
apply (drule (3) strong-tricolour-invD)
apply (force simp: black-def)

apply clarsimp
apply (elim disjE, simp-all)
apply (force simp: black-def)
apply clarsimp
apply (cases white y s)
apply (drule (2) grey-protects-whiteE)
apply blast
apply (force simp: black-def split: obj-at-splits)
done

moreover note \texttt{mi md rsi sb}
ultimately show ?case
  apply (clarsimp simp: mut-m.reachable-snapshot-inv-def in-snapshot-def
         simp del: fun-upd-apply)
  apply clarsimp
  done
qed
then show in-snapshot y ?s' by blast
qed

lemma valid-refs-invI:
[ \[ \forall x y. (x reaches y) s; x \in mut-m.mut-roots m s \land x \in mut-m.mut-ghost-honorary-root m s \land x \in mut-m.tso-write-refs m s \land grey x s \] \implies valid-ref y s
 ] \implies valid-refs-inv s
by (auto simp: valid-refs-inv-def mut-m.reachable-def grey-reachable-def)

lemma black-heap-reachable:
  assumes mut-m.reachable m y s
  assumes bh: black-heap s
  assumes vri: valid-refs-inv s
  shows black y s
using assms
apply (induct rule: reachable-induct)
apply (simp-all add: black-heap-def valid-refs-invD)
apply (metis obj-at-weakenE reachableE valid-refs-inv-def)
done

lemma valid-refs-inv-dequeue-ref
notes map-option.compositionality[simp] o-def[simp]
fixes s :: ('field, 'mut, 'ref) lsts
assumes vri: valid-refs-inv s
assumes sb: sys-mem-write-buffers (mutator m') s = mw-Mutate r f opt-r # ws
shows valid-refs-inv (s(sys := s sys\heap := (sys-heap s)(r := Option.map-option (\obj. \obj\(\obj\fields := (\obj-fields obj)(f := opt-r'))))(sys-heap s r)),
      mem-write-buffers := (mem-write-buffers (s sys))(mutator m' := ws))) (is valid-refs-inv ?s')
proof (rule valid-refs-invI)
fix m
let \root = \lambda m x (s:(\field, \mut, \ref) lsts), x \in mut-m.mut-roots m s \land x \in mut-m.mut-ghost-honorary-root m s \land x \in mut-m.tso-write-refs m s \land grey x s
fix x y assume xy: (x reaches y) ?s' and x: ?root m x ?s'
from xy have (\exists m x. ?root m x s \land (x reaches y) s) \land valid-ref y ?s'
proof induct
  case base with x sb vri show ?case
    apply -
    apply (subst obj-at-fun-upd)
    apply (auto simp: mut-m.tso-write-refs-def split: if-splits intro: valid-refs-invD(5)[where m=m])
    apply (metis list.set-intros(2) rtranclp.rtrancl-refl)
    done
next
  case (step y z)
  with sb vri show ?case
    apply -
    apply (subst obj-at-fun-upd, clarsimp)
    apply (subst (asm) obj-at-fun-upd, fastforce)
    apply (clarsimp simp: points-to-mw-Mutate simp del: fun-upd-apply)
    apply (fastforce elim: rtranclp.intros(2) simp: mut-m.tso-write-refs-def intro: exI[where x=m] valid-refs-invD(5)[where m=m'])
    done

39
qed
then show valid-ref y ?s' by blast
qed

declare map-option.compositionality[simp] o-def [simp]

lemma (in sys) reachable-snapshot-inv-black-heap-no-grey-refs-dequeue-ref[simp]:
  assumes sb: sys-mem-write-buffers (mutator m') s = mw-Mutate r f opt-r' # ws
  assumes bh: black-heap s
  assumes ngr: no-grey-refs s
  assumes vri: valid-refs-inv s
  shows mut-mreachable-snapshot-inv m (s(sys := s sys(\heap := (sys-heap s)(r := Option.map-option (\obj. obj(obj-fields := (obj-fields obj)(f := opt-r'))))) (sys-heap s r)),
    mem-write-buffers := (mem-write-buffers (s sys))(mutator m' := ws))
apply (rule mut-mreachable-snapshot-invI)
apply (rule in-snapshotI)
apply (erule black-heap-reachable)
using bh vri
apply (simp add: black-heap-def)
using bh ngr sb vri
apply (subst valid-refs-inv-def)
apply clarsimp
apply (simp add: no-grey-refs-def grey-reachable-def)
apply clarsimp
apply (drule black-heap-reachable)
apply (simp add: black-heap-def)
apply clarsimp
apply auto
done

lemma (in sys) marked-deletions-dequeue-mark[simp]:
  [ sys-mem-write-buffers p s = mw-Mark r fl # ws; mut-mmarked-deletions m s; tso-writes-inv s; valid-W-inv s ]
  \implies mut-m-marked-deletions m (s(sys := s sys(\heap := (sys-heap s)(r := Option.map-option (\obj. obj-mark-update (\obj. fl)) (sys-heap s r))), mem-write-buffers := (mem-write-buffers (s sys))(p := ws))]
apply (clarsimp simp: mut-m-marked-deletions-def)
apply (cases mutator m = p)
apply clarsimp
apply (rename-tac x)
apply (drule-tac x=x in spec)
apply (clarsimp split: mem-write-action.splits)
apply (simp add: obj-at-field-on-heap-def cong: option.case-cong)
apply (auto split: option.splits)[1]
apply clarsimp
apply (rename-tac x)
apply (drule-tac x=x in spec)
apply (clarsimp split: mem-write-action.splits)
apply (simp add: obj-at-field-on-heap-def cong: option.case-cong)
apply (auto split: option.splits)[1]
done

lemma (in sys) marked-deletions-dequeue-ref[simp]:
  [ sys-mem-write-buffers (mutator m') s = mw-Mutate r f opt-r' # ws; mut-m-marked-deletions m s; mut-m-marked-insert m' s ]
  \implies mut-m-marked-deletions m (s(sys := s sys(\heap := (sys-heap s)(r := Option.map-option (\obj. obj(obj-fields := (obj-fields obj)(f := opt-r'))) (sys-heap s r))),
    mem-write-buffers := (mem-write-buffers (s sys))(mutator m' := ws))))
apply (clarsimp simp: mut-m.marked-deletions-def)
apply (cases m = m')
apply clarsimp
apply (rename-tac x)
apply (drule-tac x=x in spec)
apply (fastforce simp: obj-at-field-on-heap-def mut-m.marked-insertions-def split: mem-write-action.splits obj-at-splits option.splits)
apply clarsimp
apply (rename-tac x)
apply (drule-tac x=x in spec)
apply (fastforce simp: obj-at-field-on-heap-def mut-m.marked-insertions-def split: mem-write-action.splits obj-at-splits option.splits)
done

lemma (in sys) black-heap-marked-insertions-dequeue[simp]:
\[
\begin{array}{l}
\text{black-heap } s; \text{valid-refs-inv } s \implies \text{mut-m.marked-insertions } m \ s \\
\end{array}
\]
by (auto simp: mut-m.marked-insertions-def black-heap-def black-def
     split: mem-write-action.splits option.splits
     dest: valid-refs-invD)

lemma (in sys) mutator-phase-inv[intro]:
\[
\begin{array}{l}
\text{LSTP } (fA-rel-inv \land fM-rel-inv \land \text{handshake-phase-inv} \land \text{mutators-phase-inv} \land \text{strong-tricolour-inv} \land \text{sys-phase-inv} \\
\quad \land \text{tso-writes-inv} \land \text{valid-refs-inv} \land \text{valid-W-inv}) \\
\end{array}
\]
apply vcg-nihe
apply vcg-ni

     split: handshake-phase.splits mem-write-action.splits)

prefer 2
apply (drule mut-m.handshake-phase-invD[where m=m])
apply (erule disjE)
apply (clarsimp simp: fM-rel-def hp-step-rel-def)
apply clarsimp

     split: handshake-phase.splits mem-write-action.splits)

     split: handshake-phase.splits mem-write-action.splits)

apply clarsimp
apply \((\text{erule } \text{disjE}, \text{simp})\)
apply \((\text{clarsimp simp: hp-step-rel-def})\)
apply \((\text{frule-tac } m = \text{ma in mut-m.handshake-phase-invD}, \text{clarsimp simp: hp-step-rel-def})\)
apply \((\text{elim disjE, simp-all split: if-splits})[1]\)
apply \((\text{clarsimp simp: fA-rel-def fM-rel-def, blast})\)
done

3.12 The infamous termination argument.

We need to know that if the GC does not receive any further work to do at \text{get-roots} and \text{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 \(\text{in } \text{mut-m}\) \(\text{gc-W-empty-mut-inv} :: (\text{field, } \text{mut, } \text{ref}) \text{lsts-pred where}\)
\[\text{gc-W-empty-mut-inv} =\]
\[\{(\text{EMPTY } \text{sys-W} \land \text{sys-ghost-handshake-in-sync } m \land \neg(\text{EMPTY } (\text{WL } (\text{mutator } m))))\} \rightarrow (\exists m', \neg(\text{sys-ghost-handshake-in-sync } m') \land \neg(\text{EMPTY } (\text{WL } (\text{mutator } m'))))\]

locset-definition \(\text{in } \) \(\text{gc-W-empty-locs} :: \text{location set where}\)
\[\text{gc-W-empty-locs} \equiv\]
\[\text{idle-locs} \cup \text{init-locs} \cup \text{sweep-locs} \cup \{ "\text{mark-read-fM}, "\text{mark-write-fA}, "\text{mark-end}" \}
\cup \text{prefixed "mark-noop"}
\cup \text{prefixed "mark-loop-get-roots"}
\cup \text{prefixed "mark-loop-get-work"}

locset-definition \(\text{black-heap-locs} = \{ "\text{sweep-idle}, "\text{idle-noop-mfence}, "\text{idle-noop-init-type"}\}
locset-definition \(\text{no-grey-refs-locs} = \text{black-heap-locs} \cup \text{sweep-locs} \cup \{"\text{mark-end}"\}

inv-definition \(\text{in } \text{gc}\) \(\text{gc-W-empty-invL} :: (\text{field, } \text{mut, } \text{ref}) \text{gc-pred where}\)
\[\text{gc-W-empty-invL} =\]
\[(\text{atS-gc } (\text{hs-get-roots-locs} \cup \text{hs-get-work-locs}) \land (\forall m. \text{mut-m.gc-W-empty-mut-inv } m)) \land \text{at-gc "mark-loop-get-roots-load-W"} (\text{EMPTY } \text{sys-W} \rightarrow \text{no-grey-refs}) \land \text{at-gc "mark-loop-get-work-load-W"} (\text{EMPTY } \text{sys-W} \rightarrow \text{no-grey-refs}) \land \text{at-gc "mark-loop"} (\text{EMPTY } \text{gc-W} \rightarrow \text{no-grey-refs}) \land \text{atS-gc no-grey-refs-loc} \land \text{sweep-loop-choose-ref} \land \text{sweep-loop-idle} \land (\text{EMPTY } \text{gc-W})\]

3.13 Sweep loop invariants

locset-definition \(\text{sweep-loop-locs} = \text{prefixed "sweep-loop"}\)

inv-definition \(\text{in } \text{gc}\) \(\text{sweep-loop-invL} :: (\text{field, } \text{mut, } \text{ref}) \text{gc-pred where}\)
\[\text{sweep-loop-invL} =\]
\[(\text{at-gc "sweep-loop-check"} \land (\neg(\text{NULL gc-mark}) \rightarrow (\lambda s. \text{obj-at } (\lambda obj. \text{Some } (\text{obj-mark obj } = \text{gc-mark s}) (\text{gc-tmp-ref s}) s)) \land (\neg(\text{NULL gc-mark}) \rightarrow \text{valid-ref } \text{gc-tmp-ref} \rightarrow \text{marked } \text{gc-tmp-ref} )) \land \text{at-gc "sweep-loop-free"} (\land (\neg(\text{NULL gc-mark}) \land \text{the } \circ \text{gc-mark } \neq \text{gc-fM} \land (\lambda s. \text{obj-at } (\lambda obj. \text{Some } (\text{obj-mark obj } = \text{gc-mark s}) (\text{gc-tmp-ref s}) s)) \land (\text{at-gc "sweep-loop-done"} (\text{valid-ref } \text{gc-tmp-ref} \rightarrow \text{marked } \text{gc-tmp-ref}) \land \text{atS-gc sweep-loop-loc (}\forall r. \neg ((r) \in \text{gc-refs}) \rightarrow \text{valid-ref } r \rightarrow \text{marked } r) \land \text{atS-gc black-heap-loc} (\forall r. \text{valid-ref } r \rightarrow \text{marked } r) \land \text{atS-gc (prefixed "sweep-loop" - \{ "sweep-loop-choose-ref" \} ) (\text{gc-tmp-ref } \in \text{gc-refs}))\)\]
4 Top-level safety

4.1 Invariants

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

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

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

definition I :: ('field, 'mut, 'ref) gc-pred where
  I ≡
  gc.invsL
  ∧ (\(\forall m.\) mut-m.invsL m)
  ∧ LSTP invs

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

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

The GC is correct for the remaining fixed-but-arbitrary initial conditions.

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

import
  HOL-Library.Saturated

begin

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

theory concrete-local-state = (field, mut, ref) local-state
theory 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 ↦ 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 ↦ 1 , 1 ↦ 2 ] ),
    4 ↦ ( obj-mark = initial-mark, 
      obj-fields = [ 1 ↦ 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

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

References


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

