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

Peter Gammie, Tony Hosking and Kai Engelhardt

September 19, 2015

Abstract

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

Contents

1 Introduction 2

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

3 Invariants and Proofs 19
  3.1 Constructors for sets of locations. 19
  3.2 Hoare triples 20
  3.3 Functions and predicates 20
  3.4 Garbage collector locations. 24
  3.5 Coarse TSO invariants 24
    3.5.1 Locations where the TSO lock is held 25
  3.6 Handshake phases 25
  3.7 Object colours, reference validity, worklist validity 31
  3.8 Mark Object 32
  3.9 The strong-tricolour invariant 38
  3.10 Invariants 39
  3.11 Lonely mutator assertions 40
  3.12 The infamous termination argument 40
  3.13 Sweep loop invariants 41
1 Introduction

We verify the memory safety of one of the Schism garbage collectors as developed by Pizlo (201x); Pizlo, Ziarek, Maj, Hosking, Blanton, and Vitek (2010) with respect to the x86-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.

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.

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

**datatype** handshake-type

= ht-NOOP
| ht-GetRoots
| ht-GetWork

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

**datatype** handshake-phase

= hp-Idle
| hp-IdleInit
| hp-InitMark
| hp-Mark
| hp-IdleMarkSweep

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

handshake-step ph ≡ case ph of

- hp-Idle ⇒ hp-IdleInit
- hp-IdleInit ⇒ hp-InitMark
- hp-InitMark ⇒ hp-Mark
- hp-Mark ⇒ hp-IdleMarkSweep
- hp-IdleMarkSweep ⇒ hp-Idle

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

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

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

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

**record** (field, ref) object =

- obj-mark :: gc-mark
- obj-fields :: field ⇒ ref option

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

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

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

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

**record**  $(\prime field, \prime mut, \prime ref)$ local-state =

— System-specific fields

heap :: $\prime ref => (\prime field, \prime ref)$ object option  

— TSO memory state

mem-write-buffers :: $\prime mut$ process-name => $(\prime field, \prime ref)$ mem-write-action list  

mem-lock :: $\prime mut$ process-name option 

— The state of the handshakes

handshake-type :: handshake-type  

handshake-pending :: $\prime mut$ => bool  

— Ghost state

ghost-handshake-in-sync :: $\prime mut$ => bool  

ghost-handshake-phase :: handshake-phase

— Mutator-specific temporaries

new-ref :: $\prime ref$ option  

roots :: $\prime ref$ set  

ghost-honorary-root :: $\prime ref$ set

— Garbage collector-specific temporaries

field-set :: $\prime field$ set  

mut :: $\prime mut$  

muts :: $\prime mut$ set

— Local variables used by multiple processes

$f_A$ :: gc-mark  

$f_M$ :: gc-mark  

cas-mark :: gc-mark option  

field :: $\prime field$  

mark :: gc-mark option  

phase :: gc-phase  

tmp-ref :: $\prime ref$  

ref :: $\prime ref$ option  

refs :: $\prime ref$ set  

$W$ :: $\prime ref$ set  

— Ghost state

ghost-honorary-grey :: $\prime ref$ set

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

**datatype**  $(\prime field, \prime ref)$ mem-read-action

4
= 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) pred

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

type-synonym (′field, 'mut, 'ref) gc-event
= (′field, 'mut, 'ref) request × (′field, 'ref) response

type-synonym (′field, 'mut, 'ref) gc-history
= (′field, 'mut, 'ref) gc-event list

type-synonym (′field, 'mut, 'ref) lst-pred
= (′field, 'mut, 'ref) local-state ⇒ bool

type-synonym (′field, 'mut, 'ref) lsts
= 'mut process-name ⇒ (′field, 'mut, 'ref) local-state

type-synonym (′field, 'mut, 'ref) lsts-pred
= (′field, 'mut, 'ref) lsts ⇒ bool

We use one locale per process to define a namespace for definitions local to these processes. Mutator definitions are parametrised by the mutator’s identifier $m$. We never interpret these locales; we use their contents typically by prefixing their names the locale name. This might be considered an abuse. The attributes depend on locale scoping somewhat, which is a mixed blessing.

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

locale mut-$m$ = fixes $m :: 'mut$
locale mut-$m'$ = mut-$m$ + fixes $m' :: 'mut$ assumes $mm'[if]: m \neq m'$
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.

context
  fixes $p :: 'mut$ process-name
begin

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

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

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

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

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

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

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

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

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

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

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

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

2.2 Handshakes

The garbage collector needs to synchronise with the mutators. In practice this is implemented with some thread synchronisation primitives that include memory fences. The scheme we adopt here has the GC busy waiting. It sets a \textit{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 \textit{get-roots} and \textit{get-work} handshakes. In practise this involves a \textit{CAS} operation. We carefully model the effect these handshakes have on the process’s TSO buffers.

The system and mutators track handshake phases using ghost state.

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

class sys
begin

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

class mut-m
begin

abbreviation mark-object :: location ⇒ ('field, 'mut, 'ref) gc-com (\[\_\_\] mark' object) where
\[\text{l} \quad \text{mark-object} \equiv \text{mark-object-fn (mutator m) l}\]

abbreviation mfence :: location ⇒ ('field, 'mut, 'ref) gc-com (\[\_\_\] MFENCE) where
\[\text{l} \quad \text{MFENCE} \equiv \text{Request (λs. (mutator m, ro-MFENCE)) (λ- s. \{s\})}\]
abbreviation hs-read-type :: location ⇒ handshake-type ⇒ (′field, ′mut, ′ref) gc-com (\[\rightarrow\]) hs′-read′-type) where
\[\hspace{1cm} \{l\} \hspace{0.1cm} \text{hs-read-type } ht \equiv \{l\} \hspace{0.1cm} \text{Request } (\lambda s. (\text{mutator } m, \text{ro-hs-mut-read-type } ht)) (\lambda s. \{s\})\]

abbreviation hs-noop-done :: location ⇒ (′field, ′mut, ′ref) gc-com (\[\rightarrow\]) hs′-noop′-done) where
\[\hspace{1cm} \{l\} \hspace{0.1cm} \text{hs-noop-done } \equiv \{l\} \hspace{0.1cm} \text{Request } (\lambda s. (\text{mutator } m, \text{ro-hs-mut-done } \{\})) (\lambda s. \{s\}) \hspace{0.1cm} \text{ghost-handshake-phase} \equiv \text{handshake-step } s \{\}
\]

abbreviation hs-get-roots-done :: location ⇒ ((′field, ′mut, ′ref) local-state ⇒ ′ref set) ⇒ ((′field, ′mut, ′ref) gc-com (\[\rightarrow\]) hs′-get′-roots-done) where
\[\hspace{1cm} \{l\} \hspace{0.1cm} \text{hs-get-roots-done } w \equiv \{l\} \hspace{0.1cm} \text{Request } (\lambda s. (\text{mutator } m, \text{ro-hs-mut-done } w (s))) (\lambda s. \{s\} W := \{\}, \text{ghost-handshake-phase} \equiv \text{handshake-step } s \{\})\]

abbreviation hs-get-work-done :: location ⇒ ((′field, ′mut, ′ref) local-state ⇒ ′ref set) ⇒ ((′field, ′mut, ′ref) gc-com (\[\rightarrow\]) hs′-get′-work-done) where
\[\hspace{1cm} \{l\} \hspace{0.1cm} \text{hs-get-work-done } w \equiv \{l\} \hspace{0.1cm} \text{Request } (\lambda s. (\text{mutator } m, \text{ro-hs-mut-done } w (s))) (\lambda s. \{s\} W := \{\})\]

definition handshake :: (′field, mut, ref) gc-com where
\[\hspace{1cm} \text{handshake} \equiv \{\}\hspace{0.1cm} \text{hs-read-type } ht-\text{NOOP} ;; \]
\[\{\}\hspace{0.1cm} \text{hs-noop-mfence} \} \hspace{0.1cm} \text{MFENCE} ;; \]
\[\{\}\hspace{0.1cm} \text{hs-noop-done} \} \hspace{0.1cm} \text{hs-noop-done} \]
\[\hspace{1cm} \}
\[\{\}\hspace{0.1cm} \text{hs-get-roots} \} \hspace{0.1cm} \text{hs-read-type } ht-\text{GetRoots} ;; \]
\[\{\}\hspace{0.1cm} \text{hs-get-roots-mfence} \} \hspace{0.1cm} \text{MFENCE} ;; \]
\[\{\}\hspace{0.1cm} \text{hs-get-roots-refs} \} \hspace{0.1cm} \text{′refs} := \text{′roots} ;; \]
\[\{\}\hspace{0.1cm} \text{hs-get-roots-loop} \} \hspace{0.1cm} \text{WHILE not empty refs DO} \]
\[\{\}\hspace{0.1cm} \text{hs-get-roots-loop-choose-ref} \} \hspace{0.1cm} \text{′ref} \in \text{Some } \text{′refs} ;; \]
\[\{\}\hspace{0.1cm} \text{hs-get-roots-loop} \} \hspace{0.1cm} \text{mark-object} ;; \]
\[\{\}\hspace{0.1cm} \text{hs-get-roots-loop-done} \} \hspace{0.1cm} \text{′refs} := (\text{′refs} - \{\text{the } \text{′ref}\}) \hspace{0.1cm} \text{OD} ;; \]
\[\{\}\hspace{0.1cm} \text{hs-get-roots-done} \} \hspace{0.1cm} \text{hs-get-roots-done} \]
\[\hspace{1cm} \}
\[\{\}\hspace{0.1cm} \text{hs-get-work} \} \hspace{0.1cm} \text{hs-read-type } ht-\text{GetWork} ;; \]
\[\{\}\hspace{0.1cm} \text{hs-get-work-mfence} \} \hspace{0.1cm} \text{MFENCE} ;; \]
\[\{\}\hspace{0.1cm} \text{hs-get-work-done} \} \hspace{0.1cm} \text{hs-get-work-done} \]

end

The garbage collector’s side of the interface.

context gc
begin
abbreviation set-hs-type :: location ⇒ handshake-type ⇒ ('field, 'mut, 'ref) gc-com (\l\) 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 (\l\) 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 (\l\) handshake'-init)
where
\{l\} handshake-init req ≡
\{l @ ".-init-type"\} set-hs-type req ;;
\{l @ ".-init-muts"\} 'muts := UNIV ;;
\{l @ ".-init-loop"\} WHILE not empty muts DO
\{l @ ".-init-loop-choose-mut"\} 'mut := 'muts ;;
\{l @ ".-init-loop-set-pending"\} set-hs-pending mut ;;
\{l @ ".-init-loop-done"\} 'muts := ('muts - \{mut\})
\OD

definition handshake-done :: location ⇒ ('field, 'mut, 'ref) gc-com (\l\) handshake'-done)
where
\{l\} handshake-done ≡
\{l @ ".-done-muts"\} 'muts := UNIV ;;
\{l @ ".-done-loop"\} WHILE not empty muts DO
\{l @ ".-done-loop-choose-mut"\} 'mut := 'muts ;;
\{l @ ".-done-loop-rendezvous"\} Request
(λs. (gc, ro-hs-gc-read-pending (muts 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 (\l\) load'-W') where
\{l\} load-W ≡ \{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 (\l\) MFENCE) where
\{l\} MFENCE ≡ \{l\} Request (λs. (gc, ro-MFENCE)) (λ- s. \{s\})

definition handshake-noop :: location ⇒ ('field, 'mut, 'ref) gc-com (\l\) handshake'-noop)
where
\{l\} handshake-noop ≡
\{l @ ".-mfnce"\} MFENCE ;;
\{l\} handshake-init ht-NOOP ;;
{% hallmark handshake-done

definition
handshake-get-roots :: location ⇒ ('field, 'mut, 'ref) gc-com (l handshake'-get'-roots)
where
{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 (l handshake'-get'-work)
where
{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.

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

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

definition
do-write-action :: ('field, 'ref) mem-write-action ⇒ ('field, 'mut, 'ref) local-state ⇒ ('field, 'mut, 'ref) local-state where
do-write-action wact ≡ λs.
case wact of
  mw-Mark r gc-mark ⇒ s(heap := (heap s)(r := map-option (λobj. obj(obj-mark := gc-mark))(heap s r)))
  | mw-Mutate r f new-r ⇒ s(heap := (heap s)(r := map-option (λobj. obj(obj-fields := (obj-fields obj)(f := new-r))(heap s r)))
  | mw-fM gc-mark ⇒ s[fm := gc-mark]
  | mw-fA gc-mark ⇒ s[fa := gc-mark]

12
| mw-Phase gc-phase \Rightarrow s(\text{phase} := gc-phase) |

**definition**

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

**where**

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

**abbreviation**

\text{processors-view-of-memory} :\text{'mut process-name} \Rightarrow \text{(}'field, 'mut, 'ref\text{)} \text{local-state} \Rightarrow \text{(}'field, 'mut, 'ref\text{)} \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, 'ref\text{)} \text{mem-read-action} \Rightarrow \text{(}'field, 'mut, 'ref\text{)} \text{local-state} \Rightarrow \text{(}'field, 'ref\text{)} \text{response}

**where**

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

**definition**

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

**where**

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

**context** sys

**begin**

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

**definition**

\text{mem-TSO} :\text{(}'field, 'mut, 'ref\text{)} \text{gc-com}

**where**
mem-TSO ≡
\[\{"sys-read"\} \text{ Response } (\lambda req s. \{ (s, sys-read p mr s) \\
| p mr. req = (p, ro-Read mr) \land \text{not-blocked } s \\p \}) \]

∪ \[\{"sys-write"\} \text{ Response } (\lambda req s. \{ (s\}, \text{mem-write-buffers} := (\text{mem-write-buffers} s)(p := \text{mem-write-buffers} s p @ [w]), \text{mv-Void} \\
| p w. req = (p, ro-Write w) \}) \]

∪ \[\{"sys-mfence"\} \text{ Response } (\lambda req s. \{ (s\}, \text{mv-Void} \\
| p. req = (p, ro-MFENCE) \land \text{mem-write-buffers} s p = [] \}) \]

∪ \[\{"sys-lock"\} \text{ Response } (\lambda req s. \{ (s\}, \text{mem-lock} := \text{Some } p), \text{mv-Void} \\
| p. req = (p, ro-Lock) \land \text{mem-lock } s = \text{None } \}) \]

∪ \[\{"sys-unlock"\} \text{ Response } (\lambda req s. \{ (s\}, \text{mem-lock} := \text{None }), \text{mv-Void} \\
| p. req = (p, ro-Unlock) \land \text{mem-lock } s = \text{Some } p \land \\
\text{mem-write-buffers} s p = [] \}) \]

∪ \[\{"sys-dequeue-write-buffer"\} \text{ LocalOp } (\lambda s. \{ (\text{do-write-action } w s)\}, \text{mem-write-buffers} := (\text{mem-write-buffers} s)(p := ws) \}) \]

\[| p w ws. \text{mem-write-buffers} s p = w \# ws \land \\
\text{not-blocked } s p \land p \neq \text{sys } \}) \]

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
alloc ≡ \[\{"sys-alloc"\} \text{ Response } (\lambda req s. \\
| \{ (s\}, \text{heap} := \text{(heap } s)(r := \text{Some } (\text{obj-mark } = \text{fA } s, \text{obj-fields } = \langle \text{None } \rangle)), \text{mv-Ref} \\
| (\text{Some } r) \} \]

| r. r \notin \text{dom } (\text{heap } s) \land \text{snd req } = \text{ro-Alloc} \}) \]

References are freed by removing them from heap.

definition free :: (field, mut, ref) gc-com
where
free ≡ \[\{"sys-free"\} \text{ Response } (\lambda req s. \\
| \{ (s\}, \text{heap} := \text{(heap } s)(r := \text{None})), \text{mv-Void} \}

| r. \text{snd req } = \text{ro-Free } r \}) \]

The top-level system process.

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

where

\[
\begin{align*}
\text{com} & \equiv \\
& \text{LOOP DO} \\
& \quad \text{mem-TSO} \\
& \quad \sqcup \text{alloc} \\
& \quad \sqcup \text{free} \\
& \quad \sqcup \text{handshake} \\
& \text{OD}
\end{align*}
\]

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.

\text{context mut-m}

begin

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

\text{abbreviation (in –) mut-alloc :: } \langle \text{field, mut, ref} \rangle \text{ gc-com where}

\[
\begin{align*}
\text{mut-alloc} \ m & \equiv \\
& \{ |\text{alloc}| \} \text{Request} (\lambda s. (\text{mutator } m, \text{ro-Alloc})) \\
& (\lambda mv s. \{ s(\text{roots} := \text{roots } s \cup \{r\} \mid r. \text{mv} = \text{mv-Ref} (\text{Some } r) ) \})
\end{align*}
\]

\text{abbreviation alloc :: } \langle \text{field, mut, ref} \rangle \text{ gc-com where}

\[
\begin{align*}
\text{alloc} & \equiv \text{mut-alloc} \ m
\end{align*}
\]

The mutator can always discard any references it holds.

\text{abbreviation discard :: } \langle \text{field, mut, ref} \rangle \text{ gc-com where}

\[
\begin{align*}
\text{discard} & \equiv \\
& \{ |\text{discard-refs}| \} \text{LocalOp} (\lambda s. \{ s(\text{roots} := \text{roots}=\text{roots}. \text{roots} \subseteq \text{roots } s) \})
\end{align*}
\]

Load and store are defined in Pizlo (201x, Figure 2.9).

 Dereferencing a reference can increase the set of mutator roots.

\text{abbreviation load :: } \langle \text{field, mut, ref} \rangle \text{ gc-com where}

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

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

**abbreviation**

mut-deref :: location

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

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

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

\[\Rightarrow \langle \text{field, mut, ref} \rangle, \text{local-state} \Rightarrow \langle \text{field} \rangle\]

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

\[\Rightarrow \langle \text{field, mut, ref} \rangle, \text{local-state} \Rightarrow \langle \text{field} \rangle\]

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

\[\Rightarrow \langle \text{field, mut, ref} \rangle, \text{gc-com} (\langle \text{ref} \rangle \text{ deref})\]

where

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

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

**abbreviation**

write-ref :: location

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

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

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

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

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

\[\Rightarrow \langle \text{field, mut, ref} \rangle, \text{local-state} \Rightarrow \langle \text{field} \rangle\]

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

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

where

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

**definition**

store :: (\langle \text{field, mut, ref} \rangle, \text{gc-com})

where

store \equiv

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

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

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

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

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

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

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

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

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

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

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**

com :: (\langle \text{field, mut, ref} \rangle, \text{gc-com})

where

com \equiv
LOOP DO
   "mut local computation" SKIP
   alloc
   discard
   load
   store
   "mut mfence" MFENCE
OD

end

2.5 Garbage collector
We abstract the primitive actions of the garbage collector thread.

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

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

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

context gc
begin
abbreviation write-fM :: location ⇒ ("field", "mut", "ref") local-state ⇒ gc-mark) ⇒ ("field", "mut", "ref") gc-com (|{|} write-fM) where
|{|} write-fM f ≡ |{|} Request (λs. (gc, WritefM (f s))) (λ- s. {s})

abbreviation read-fM :: location ⇒ ("field", "mut", "ref") gc-com (|{|} read-fM) where
|{|} read-fM ≡ |{|} Request (λs. (gc, ReadfM)) (λm v s. {s mv := m} |m. m v = mv-Mark (Some m) )

abbreviation write-fM :: location ⇒ ("field", "mut", "ref") gc-com (|{|} write-fM) where
|{|} write-fM ≡ |{|} Request (λs. (gc, WritefM (fM s))) (λ- s. {s})

abbreviation write-phase :: location ⇒ gc-phase ⇒ ("field", "mut", "ref") gc-com (|{|} write-phase) where
|{|} write-phase ph ≡ |{|} Request (λs. (gc, WritePhase ph)) (λ- s. {s phase := ph })

abbreviation mark-object :: location ⇒ ("field", "mut", "ref") gc-com (|{|} mark-object) where
|{|} mark-object ≡ mark-object-fn gc l

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

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

definition (in gc)
com :: ("field", "mut", "ref") gc-com
where
com ≡

LOOP
  |{|} idle-noop
  |{|} idle-read-fM
  |{|} idle-invert-fM "m := (¬ "m)"
  |{|} idle-write-fM
  |{|} idle-flip-noop
  |{|} idle-phase-init
  |{|} init-noop
  |{|} init-phase-mark
  |{|} mark-read-fM
  |{|} mark-write-fA
  |{|} mark-noop

end
\[
\text{"mark-loop-get-roots"} \text{ handshake-get-roots ;; (* hp-IdleMarkSweep *)}
\]

\[
\text{"mark-loop"} \text{ WHILE not empty W DO}
\]

\[
\text{"mark-loop-inner"} \text{ WHILE not empty W DO}
\]

\[
\text{"mark-loop-choose-ref"} \text{ tmp-ref :\ W ;;}
\]

\[
\text{"mark-loop-fields"} \text{ field-set := UNIV ;;}
\]

\[
\text{"mark-loop-mark-object-loop"} \text{ WHILE not empty field-set DO}
\]

\[
\text{"mark-loop-mark-choose-field"} \text{ field \in field-set ;;}
\]

\[
\text{"mark-loop-mark-deref"} \text{ ref := tmp-ref \rightarrow field ;;}
\]

\[
\text{"mark-loop"} \text{ mark-object ;;}
\]

\[
\text{"mark-loop-mark-field-done"} \text{ field-set := (field-set - \{field\})}
\]

\[
\text{"mark-loop-blacken"} \text{ W := (W - \{tmp-ref\})}
\]

\[
\text{OD ;;}
\]

\[
\text{"mark-loop-get-work"} \text{ handshake-get-work}
\]

\[
\text{OD ;;}
\]

\[
\text{(* sweep *)}
\]

\[
\text{"mark-end"} \text{ write-phase ph-Sweep ;;}
\]

\[
\text{"sweep-read-fM"} \text{ read-fM ;;}
\]

\[
\text{"sweep.refs"} \text{ refs := UNIV ;;}
\]

\[
\text{"sweep-loop"} \text{ WHILE not empty refs DO}
\]

\[
\text{"sweep-loop-choose-ref"} \text{ tmp-ref :\ refs ;;}
\]

\[
\text{"sweep-loop-read-mark"} \text{ mark := tmp-ref \rightarrow flag ;;}
\]

\[
\text{"sweep-loop-check"} \text{ IF not null mark and the \circ mark neq fM THEN}
\]

\[
\text{"sweep-loop-free"} \text{ free tmp-ref}
\]

\[
\text{FI ;;}
\]

\[
\text{"sweep-loop-ref-done"} \text{ refs := (refs - \{tmp-ref\})}
\]

\[
\text{OD ;;}
\]

\[
\text{"sweep-idle"} \text{ write-phase ph-Idle}
\]

\[
\text{OD}
\]

end

primrec
gc-pgms :: 'mut process-name \Rightarrow ('field, 'mut, 'ref) gc-com

where
gc-pgms (mutator m) = mut-m.com m
| gc-pgms gc = gc.com
| gc-pgms sys = sys.com
3 Invariants and Proofs

3.1 Constructors for sets of locations.

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

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

3.2 Hoare triples

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

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

abbreviation valid-proc-inv-syn :: (′field, ′mut, ′ref) gc-pred ⇒ ′mut process-name ⇒ bool (\{P\} - \{Q\})
where
  \{P\} p \{P\} ≡ \forall (c, afts) ∈ vcg-fragments (gc-pgms p). (gc-pgms, p, afts \models \{P\} c \{Q\})

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)).

abbreviation is-mw-Mark w ≡ \exists r fl. w = mw-Mark r fl
abbreviation is-mw-Mutate w ≡ \exists r f r’. w = mw-Mutate r f r’
abbreviation is-mw-fA w ≡ \exists fl. w = mw-fA fl
abbreviation is-mw-fM w ≡ \exists fl. w = mw-fM fl
abbreviation is-mw-Phase w ≡ \exists ph. w = mw-Phase ph

abbreviation (input) pred-in-W :: ′ref ⇒ ′mut process-name ⇒ (′field, ′mut, ′ref) lsts-pred
  (infix in-W 50) where
  r in-W p ≡ \lambda s. r ∈ W (s p)

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

**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

\[- \models \{} P\}/ \text{-}/ \{} Q\}

**where**

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

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

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

**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 \text{the} \circ 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**

\[ \text{at-gc} l P \equiv \text{at gc} l \text{ imp LSTP} P \]

**abbreviation** atS-gc :: location set \Rightarrow (field, 'mut, 'ref) lsts-pred \Rightarrow (field, 'mut, 'ref) gc-pred

**where**

\[ \text{atS-gc} ls P \equiv \text{atS gc} ls \text{ imp LSTP} P \]

**context mut-m**

**begin**

**abbreviation**

valid-mut-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
\[ (- | \cfrac{\cdot}{\cdot} / \cfrac{\cdot}{\cdot}) \]

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

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

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

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

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

end

context \ sys
begin

abbreviation
\[
\text{valid-sys-syn} :: ('field, 'mut, 'ref) gc-loc-comp ⇒ ('field, 'mut, 'ref) gc-pred ⇒ ('field, 'mut, 'ref) gc-com ⇒ bool \ (- | \cfrac{\cdot}{\cdot} / \cfrac{\cdot}{\cdot})
\]

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

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

end

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

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

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

Projections on TSO buffers.

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

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

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

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

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

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

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

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

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

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

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

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

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

3.4 Garbage collector locations.

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

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

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

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

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

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 ⇒ bool where 
\( \text{gc-writes } w \equiv \text{case } w \text{ of } \text{mw-Mark } - - \Rightarrow \text{True} \mid \text{mw-Phase } - \Rightarrow \text{True} \mid \text{mw-fM } - \Rightarrow \text{True} \mid \text{mw-fA } - \Rightarrow \text{True} \mid - \Rightarrow \text{False} \)

abbreviation mut-writes :: ('field, 'ref) mem-write-action ⇒ bool where 
\( \text{mut-writes } w \equiv \text{case } w \text{ of } \text{mw-Mutate } - - \Rightarrow \text{True} \mid \text{mw-Mark } - \Rightarrow \text{True} \mid - \Rightarrow \text{False} \)
**3.5.1 Locations where the TSO lock is held**

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

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gc-tso-lock-locs</code></td>
<td><code>location set</code></td>
</tr>
<tr>
<td><code>tso-lock-invL</code></td>
<td><code>gc-pred</code></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mut-tso-lock-locs</code></td>
<td><code>location set</code></td>
</tr>
<tr>
<td><code>tso-lock-invL</code></td>
<td><code>gc-pred</code></td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hp-step-rel</code></td>
<td><code>(bool × handshake-type × handshake-phase × handshake-phase) set</code></td>
</tr>
</tbody>
</table>

### ML Import

<table>
<thead>
<tr>
<th>Import</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ML</code></td>
<td>Import for ML definitions</td>
</tr>
</tbody>
</table>

---

25
definition handshake-phase-inv :: ('field', 'mut', 'ref) lsts-pred where
handshake-phase-inv ≡ ALLS m.
(sys-ghost-handshake-in-sync m ⊗ sys-handshake-type
⊗ sys-ghost-handshake-phase ⊗ mut-m.mut-ghost-handshake-phase m) in ⟨hp-step-rel⟩
and (sys-handshake-pending m imp not sys-ghost-handshake-in-sync m)(proof)(proof)(proof)(proof)(proof)(proof)

Connect sys-ghost-handshake-phase with locations in the GC.

definition hp-Idle-locs ≡
(prefixed "idle-noop" − { "idle-noop-mfence", "idle-noop-init-type" })
∪ { "idle-read-fM", "idle-invert-fM", "idle-write-fM", "idle-flip-noop-mfence", "idle-flip-noop-init-type"
}
⟨ML⟩

definition hp-IdleInit-locs ≡
(prefixed "idle-flip-noop" − { "idle-flip-noop-mfence", "idle-flip-noop-init-type" })
∪ { "idle-phase-init", "init-noop-mfence", "init-noop-init-type"
}
⟨ML⟩

definition hp-InitMark-locs ≡
(prefixed "init-noop" − { "init-noop-mfence", "init-noop-init-type" })
}
⟨ML⟩

definition hp-IdleMarkSweep-locs ≡
{ "idle-noop-mfence", "idle-noop-init-type", "mark-end" }
∪ sweep-locs
∪ (mark-loop-locs − { "mark-loop-get-roots-init-type" })
⟨ML⟩

definition hp-Mark-locs ≡
(prefixed "mark-noop" − { "mark-noop-mfence", "mark-noop-init-type" })
∪ { "mark-loop-get-roots-init-type"
}
⟨ML⟩

abbreviation
hs-noop-prefixes ≡ {"idle-noop", "idle-flip-noop", "init-noop", "mark-noop" }

definition hs-noop-locs ≡
∪ l ∈ hs-noop-prefixes. prefixed l − (suffixed "-noop-mfence" ⊗ suffixed "-noop-init-type")
⟨ML⟩

definition hs-get-roots-locs ≡
prefixed "mark-loop-get-roots" − {"mark-loop-get-roots-init-type"}
definition hs-get-work-locs ≡
    prefixed "mark-loop-get-work" − {"mark-loop-get-work-init-type"}

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

definition hs-init-loop-locs ≡ ∪ l ∈ hs-prefixes. prefixed (l @ "-init-loop")

definition hs-done-loop-locs ≡ ∪ l ∈ hs-prefixes. prefixed (l @ "-done-loop")

definition hs-done-locs ≡ ∪ l ∈ hs-prefixes. prefixed (l @ "-done")

definition hs-none-pending-locs ≡ − (hs-init-loop-locs ∪ hs-done-locs)

definition hs-in-sync-locs ≡
    (− ( ( ∪ l ∈ hs-prefixes. prefixed (l @ "-init") ) ∪ hs-done-locs ))
    ∪ ( ∪ l ∈ hs-prefixes. {l @ "-init-type"})

definition hs-out-of-sync-locs ≡
    ∪ l ∈ hs-prefixes. {l @ "-init-muts"}

definition hs-mut-in-muts-locs ≡
    ∪ l ∈ hs-prefixes. {l @ "-init-loop-set-pending", l @ "-init-loop-done"}

definition hs-init-loop-done-locs ≡
    ∪ l ∈ hs-prefixes. {l @ "-init-loop-done"}

definition hs-init-loop-not-done-locs ≡
    hs-init-loop-locs − ( ∪ l ∈ hs-prefixes. {l @ "-init-loop-done"})

definition (in gc) handshake-invL :: (field, mut, ref) gc-pred where
    [inv]: handshake-invL ≡
        atS-gc hs-noop-locs (sys-handshake-type eq ⟨ht-NOOP⟩)
        and atS-gc hs-get-roots-locs (sys-handshake-type eq ⟨ht-GetRoots⟩)
        and atS-gc hs-get-work-locs (sys-handshake-type eq ⟨ht-GetWork⟩)

27
and atS-gc hs-mut-in-muts-locs (gc-mut in gc-muts)
and atS-gc hs-init-loop-locs (ALLS m. not ⟨m⟩ in gc-muts imp sys-handshake-pending m
or sys-ghost-handshake-in-sync m)
and atS-gc hs-init-loop-not-done-locs (ALLS m. ⟨m⟩ in gc-muts imp not sys-handshake-pending m
and not sys-ghost-handshake-in-sync m)
and atS-gc hs-init-loop-done-locs ((sys-handshake-pending ⊃ gc-mut
or sys-ghost-handshake-in-sync ⊃ gc-mut)
and (ALLS m. ⟨m⟩ in gc-muts and ⟨m⟩ neq gc-mut
imp not sys-handshake-pending m
and not sys-ghost-handshake-in-sync m))
and atS-gc hs-done-locs (ALLS m. sys-handshake-pending m or sys-ghost-handshake-in-sync m)
and atS-gc hs-done-loop-locs (ALLS m. not ⟨m⟩ in gc-muts imp not sys-handshake-pending m)
and atS-gc hs-none-pending-locs (ALLS m. not sys-handshake-pending m)
and atS-gc hs-in-sync-locs (ALLS m. sys-ghost-handshake-in-sync m)
and atS-gc hs-out-of-sync-locs (ALLS m. not sys-handshake-pending m
and not sys-ghost-handshake-in-sync m)

and atS-gc hp-Idle-locs (sys-ghost-handshake-phase eq ⟨hp-Idle⟩)
and atS-gc hp-IdleInit-locs (sys-ghost-handshake-phase eq ⟨hp-IdleInit⟩)
and atS-gc hp-IdleMark-locs (sys-ghost-handshake-phase eq ⟨hp-IdleMark⟩)
and atS-gc hp-Mark-locs (sys-ghost-handshake-phase eq ⟨hp-Mark⟩)

Local handshake phase invariant for the mutators.

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

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

Relate sys-ghost-handshake-phase, gc-phase, sys-phase and writes to the phase in the GC’s
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, ph, []) | in-sync hp ph. handshake-phase-rel hp in-sync ph }
∪ ({True} × { (hp-IdleInit, ph-Init, ph-Idle, [mw-Phase ph-Init]),
  (hp-InitMark, ph-Mark, ph-Init, [mw-Phase ph-Mark]),
  (hp-IdleMarkSweep, ph-Sweep, ph-Mark, [mw-Phase ph-Sweep]),
  (hp-IdleMarkSweep, ph-Idle, ph-Mark, [mw-Phase ph-Sweep, mw-Phase ph-Idle]),
  (hp-IdleMarkSweep, ph-Idle, ph-Sweep, [mw-Phase ph-Idle]) })

definition phase-rel-inv :: (field, 'mut, 'ref) lsts-pred where
phase-rel-inv ≡ (ALLS m. sys-ghost-handshake-in-sync m) ⊗ sys-ghost-handshake-phase ⊗ gc-phase ⊗ sys-phase ⊗ tso-pending-phase gc in ⟨phase-rel⟩⟨proof⟩⟨proof⟩

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

definition no-pending-phase-locs :: location set where
no-pending-phase-locs ≡
  (idle-locs − { "idle-noop-mfence" })
∪ (init-locs − { "init-noop-mfence" })
∪ (mark-locs − { "mark-read-fM", "mark-write-fA", "mark-noop-mfence" })

(in gc) phase-invL :: (field, 'mut, 'ref) gc-pred where
[inv]: phase-invL ≡
  atS-gc idle-locs (gc-phase eq ⟨ph-Idle⟩)
and atS-gc init-locs (gc-phase eq ⟨ph-Init⟩)
and atS-gc mark-locs (gc-phase eq ⟨ph-Mark⟩)
and atS-gc sweep-locs (gc-phase eq ⟨ph-Sweep⟩)
and atS-gc no-pending-phase-locs (list-null (tso-pending-phase gc))⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

30
∪ init-locs

\langle ML \rangle

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

3.7 Object colours, reference validity, worklist validity

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

White potential garbage, not yet reached

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

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

In this particular setting we use the following interpretation:

White: not marked

Grey: on a worklist

Black: marked and not on a worklist

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

abbreviation marked :: 'ref \Rightarrow ('field, 'mut, 'ref) lsts-pred where
  marked r s \equiv \lambda obj. \lambda obj-mail obj = sysM s \ r s

abbreviation white :: 'ref \Rightarrow ('field, 'mut, 'ref) lsts-pred where
  white r s \equiv \lambda obj. \lambda obj-mail obj = (\neg sysM s) \ r s

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

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

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

The worklists track the grey objects. The following invariant asserts that grey objects are marked on the heap except for a few steps near the end of mark-object-fn, the processes’ worklists and ghost-honorary-greys are disjoint, and that pending marks are sensible.

The safety of the collector does not depend on disjointness; we include it as proof that the single-threading of grey objects in the implementation is sound.
3.8 Mark Object

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

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

\textbf{locale mark-object =}

\textbf{fixes} \( p :: \text{mut process-name} \)

\textbf{fixes} \( l :: \text{location} \)

\textbf{fixes} \( \text{p-ph-enabled} :: (\text{field, mut, ref}) \text{lsts-pred} \)

\textbf{assumes} \( \text{p-ph-enabled-eq-imp}: \text{eq-imp} (\lambda(\text{unit}) s \ s p) \text{ p-ph-enabled} \)

\textbf{begin}

\textbf{abbreviation (input)} \( p\text{-cas-mark} s \equiv \text{cas-mark} (s \ p) \)

\textbf{abbreviation (input)} \( p\text{-mark} s \equiv \text{mark} (s \ p) \)

\textbf{abbreviation (input)} \( p\text{-fM} s \equiv \text{fM} (s \ p) \)

\textbf{abbreviation (input)} \( p\text{-ghost-handshake-phase} s \equiv \text{ghost-handshake-phase} (s \ p) \)

\textbf{abbreviation (input)} \( p\text{-ghost-honorary-grey} s \equiv \text{ghost-honorary-grey} (s \ p) \)

\textbf{abbreviation (input)} \( p\text{-phase} s \equiv \text{phase} (s \ p) \)

\textbf{abbreviation (input)} \( p\text{-ref} s \equiv \text{ref} (s \ p) \)

\textbf{abbreviation (input)} \( p\text{-the-ref} \equiv \text{the} \circ p\text{-ref} \)

\textbf{abbreviation (input)} \( p\text{-W} s \equiv \text{W} (s \ p) \)

\textbf{abbreviation at-p :: location ⇒ (field, mut, ref) lsts-pred ⇒ (field, mut, ref) gc-pred where}

\( \text{at-p } l' \ P \equiv \text{at } p (l @ l') \text{ imp LSTP } P \)

\textbf{abbreviation (input)} \( p\text{-en-cond} P \equiv p\text{-ph-enabled imp } P \)

\textbf{abbreviation (input)} \( p\text{-valid-ref} \equiv \text{not null } p\text{-ref} \text{ and valid-ref } p\text{-the-ref} \)

\textbf{abbreviation (input)} \( p\text{-tso-no-pending-mark} \equiv \text{list-null } (\text{tso-pending-mark } p) \)

\textbf{abbreviation (input)} \( p\text{-tso-no-pending-mutate} \equiv \text{list-null } (\text{tso-pending-mutate } p) \)

\textbf{abbreviation (input)}

\( p\text{-valid-w-inv} \equiv ((p\text{-cas-mark neq p-mark} \text{ or p-tso-no-pending-mark} \text{ imp marked } p\text{-the-ref}) \text{ and } (\text{tso-pending-mark } p \text{ in } (\lambda s. \{[], [\text{mw-Mark} (p\text{-the-ref } s) (p\text{-fM } s)]\} ) ) \)

\textbf{abbreviation (input)}

\( p\text{-mark-inv} \equiv \text{not null } p\text{-mark} \text{ and } ((\lambda s. \text{obj-at } (\lambda obj. \text{Some } (obj\text{-mark } obj) = p\text{-mark } s) (p\text{-the-ref } s) s) \text{ or marked } p\text{-the-ref}) \)

\textbf{abbreviation (input)}

\( p\text{-cas-mark-inv} \equiv (\lambda s. \text{obj-at } (\lambda obj. \text{Some } (obj\text{-mark } obj) = p\text{-cas-mark } s) (p\text{-the-ref } s) s) \)
abbreviation (input) \( p\text{-valid-fM} \equiv p\text{-fM eq sysM} \)

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

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

definition mark-object-invL :: (field, mut, ref) gc-pred where
  mark-object-invL \equiv
  at-p "-mo-null" (True)
  and at-p "-mo-mark" (p-valid-ref)
  and at-p "-mo-fM" (p-valid-ref and p-en-cond (p-mark-inv))
  and at-p "-mo-mtest" (p-valid-ref and p-en-cond (p-mark-inv and p-valid-fM))
  and at-p "-mo-phase" (p-valid-ref and p-mark neq Some o p-fM and p-en-cond (p-mark-inv and p-valid-fM))
  and at-p "-mo-ptest" (p-valid-ref and p-mark neq Some o p-fM and p-en-cond (p-mark-inv and p-valid-fM))
  and at-p "-mo-co-lock" (p-valid-ref and p-mark-inv and p-valid-fM and p-mark neq Some o p-fM and p-tso-no-pending-mark)
  and at-p "-mo-co-cmark" (p-valid-ref and p-mark-inv and p-valid-fM and p-mark neq Some o p-fM and p-tso-no-pending-mark)
  and at-p "-mo-co-ctest" (p-valid-ref and p-mark-inv and p-valid-fM and p-mark neq Some o p-fM and p-cas-mark-inv and p-valid-fM and white
  mark-object-fn in the GC and during the root marking are straightforward.

interpretation gc-mark!: mark-object gc "mark-loop" (True)

(\ proof \)

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

(\ proof \)

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 \( m \ neq \langle \text{hp-Idle} \rangle \) for \( m \)

\( \langle \text{proof} \rangle \)

**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 \( m \ neq \langle \text{hp-Idle} \rangle \) for \( m \)

\( \langle \text{proof} \rangle \)

**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.

**definition** mut-hs-get-roots-loop-locs \( \equiv \)

\( \text{prefixed} \ "\text{hs-get-roots-loop}" \)

\( \langle \text{ML} \rangle \)

**definition** mut-hs-get-roots-loop-mo-locs \( \equiv \)

\( \text{prefixed} \ "\text{hs-get-roots-loop-mo}" \cup \{ "\text{hs-get-roots-loop-done}" \} \)

\( \langle \text{ML} \rangle \)

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

**definition** mut-hs-not-hp-Idle-locs \( \equiv \)

\( \bigcup \text{pref}\,\in\text{mut-async-mark-object-prefixes}.\ \bigcup \{ "\text{mo-co-lock}"", "\text{mo-co-cmark}"", "\text{mo-co-ctest}"", "\text{mo-co-mark}"", "\text{mo-co-unlock}"", "\text{mo-co-won}"", "\text{mo-co-W}" \}. \{ \text{pref} \@ "." \@ l \} \)

\( \langle \text{ML} \rangle \)

**definition** mut-async-mo-ptest-locs \( \equiv \)

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

\( \langle \text{ML} \rangle \)

**definition** mut-mo-ptest-locs \( \equiv \)

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

\( \langle \text{ML} \rangle \)

**definition** mut-mo-valid-ref-locs \( \equiv \)

\( \text{prefixed} \ "\text{store-del}" \cup \text{prefixed} \ "\text{store-ins}" \cup \{ "\text{deref-del}"", "\text{lop-store-ins}" \} \)

\( \langle \text{ML} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \)

This local invariant for the mutators illustrates the handshake structure: we can rely on the insertion barrier earlier than on the deletion barrier. Both need to be installed before 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.

**definition** ghost-honorary-grey-empty-locs :: location set where

\[
\text{ghost-honorary-grey-empty-locs} \equiv \\
\left( \bigcup \text{pref} \in \{ \text{"mark-loop"}, \text{"hs-get-roots-loop"}, \text{"store-del"}, \text{"store-ins"} \}. \{ \text{pref @"." @ l} \} \right) \\
\left( \bigcup \text{l} \in \{ \text{"mo-co-unlock"}, \text{"mo-co-won"}, \text{"mo-co-W"} \}. \{ \text{pref @"." @ l} \} \right)
\]

(*insertion barrier*)

and at-mut "store-ins" (mut-ghost-handshake-phase eq (hp-Idle))

and sys-phase neq (hp-Idle))

and not null mut-new-ref

imp marked \(\triangleright\) mut-the-new-ref

(*deletion barrier*)

\[ \text{(ML)} \]
and atS-mut (prefixed "store-del-mo") ∪ \{"lop-store-ins"\}

( (mut-ghost-handshake-phase eq (hp-Mark)
  or (mut-ghost-handshake-phase eq (hp-IdleMarkSweep)

and sys-phase neq (ph-Idle)))

and (λs. ∀ opt-r'. ¬tso-pending-write (mutator m) (mw-Mutate
(mut-temp-ref s) (mut-field s) opt-r') s)

imp (λs. obj-at-field-on-heap (λr. mut-ref s = Some r ∨ marked r s) (mut-temp-ref s) (mut-field s) s))

and at-mut "lop-store-ins" ( (mut-ghost-handshake-phase eq (hp-Mark)
  or (mut-ghost-handshake-phase eq (hp-IdleMarkSweep)

and sys-phase neq (ph-Idle)))

and not null mut-ref
imp marked ▷ mut-the-ref )

and atS-mut (prefixed "store-ins")

( (mut-ghost-handshake-phase eq (hp-Mark)
  or (mut-ghost-handshake-phase eq (hp-IdleMarkSweep)

and sys-phase neq (ph-Idle)))

and (λs. ∀ opt-r'. ¬tso-pending-write (mutator m) (mw-Mutate
(mut-temp-ref s) (mut-field s) opt-r') s)

imp (λs. obj-at-field-on-heap (λr'. marked r' s) (mut-temp-ref s)
(mut-field s) s) (proof) (proof) (proof) (proof) (proof) (proof) (proof) (proof) (proof) (proof) (proof) (proof)

The GC’s use of mark-object-fn is correct.

When we take grey tmp-ref to black, all of the objects it points to are marked, ergo the new black does not point to white, and so we preserve the strong tricolour invariant.

definition (in gc) obj-fields-marked-inv :: (field, 'mut, 'ref) lsts-pred

where

obj-fields-marked-inv ≡

ALLS f. \{ f \} in (¬ gc-field-set) imp (λs. obj-at-field-on-heap (λr. marked r s) (gc-temp-ref s) f s) (proof) (proof) (proof)

definition obj-fields-marked-locs :: location set

where

obj-fields-marked-locs ≡

\{ "mark-loop-mark-object-loop", "mark-loop-mark-choose-field", "mark-loop-mark-deref",
"mark-loop-mark-field-done", "mark-loop-blacken" \}

∪ prefixed "mark-loop-mo"

ML

definition (in gc) obj-fields-marked-invL :: (field, 'mut, 'ref) gc-pred

where

[inv]: obj-fields-marked-invL ≡

atS-gc obj-fields-marked-locs (obj-fields-marked-inv and gc-temp-ref in gc-W)

and atS-gc (prefixed "mark-loop-mo" ∪ \{ "mark-loop-mark-field-done" \})

(λs. obj-at-field-on-heap (λr. gc-ref s = Some r ∨ marked r s) (gc-temp-ref s) (gc-field s) s)

and atS-gc (prefixed "mark-loop-mo") (ALLS y. not null gc-ref and (λs. ((gc-the-ref s) reaches y) s) imp valid-ref y)

and at-gc "mark-loop-fields" (gc-temp-ref in gc-W)
and at-gc "mark-loop-mark-field-done" (not null gc-ref imp marked ⊢ gc-the-ref)
and at-gc "mark-loop-blacken" (empty gc-field-set)
and atS-gc ghost-honorary-grey-empty-locs (empty gc-ghost-honorary-grey)

3.9 The strong-tricolour invariant

As the GC algorithm uses both insertion and deletion barriers, it preserves the strong tricolour-invariant:

abbreviation points-to-white :: 'ref ⇒ 'ref ⇒ ('field, 'mut, 'ref) lsts-pred (infix points'-to'-white 51) where
  x points-to-white y ≡ x points-to y and white y

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

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

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

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

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

lemma strong-tricolour-inv s ⇒ weak-tricolour-inv s(proof)

The key invariant that the mutators establish as they perform get-roots: they protect their white-reachable references with grey objects.

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

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

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

In some phases we need to know that the insertion and deletion barriers are installed, in order to preserve the snapshot. These can ignore TSO effects as marks hit system memory in a timely way.
abbreviation marked-insertion :: ('field, 'ref) mem-write-action ⇒ ('field, 'mut, 'ref) lsts-pred
where
marked-insertion w ≡ λs. case w of mw-Mutate r f (Some r') ⇒ marked r' s | - ⇒ True

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

abbreviation marked-deletion :: ('field, 'ref) mem-write-action ⇒ ('field, 'mut, 'ref) lsts-pred
where
marked-deletion w ≡ λs. case w of mw-Mutate r f opt-r' ⇒ obj-at-field-on-heap (λr'. marked r' s) r f s | - ⇒ True

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

Finally, in some phases the heap is somewhat monochrome.

definition black-heap :: ('field, 'mut, 'ref) lsts-pred where
black-heap ≡ ALLS r. black r or not valid-ref r

definition white-heap :: ('field, 'mut, 'ref) lsts-pred where
white-heap ≡ ALLS r. white r or not valid-ref r

definition no-black-ref :: ('field, 'mut, 'ref) lsts-pred where
no-black-ref ≡ ALLS r. not black r

definition no-grey-ref :: ('field, 'mut, 'ref) lsts-pred where
no-grey-ref ≡ ALLS r. not grey r (proof) (proof) (proof) (proof) (proof) (proof) (proof) (proof) (proof) (proof) (proof) (proof) (proof) (proof)

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 ⇒ ('field, 'mut, 'ref) lsts-pred
where
mutator-phase-inv-aux hp-Idle = (True)
| mutator-phase-inv-aux hp-IdleInit = no-black-ref
| mutator-phase-inv-aux hp-InitMark = marked-insertions
| mutator-phase-inv-aux hp-Mark = (marked-insertions and marked-deletions)
| mutator-phase-inv-aux hp-IdleMarkSweep = (marked-insertions and marked-deletions and
reachable-snapshot-inv)

abbreviation (in mut-m) mutator-phase-inv :: ('field, 'mut, 'ref) lsts-pred where
mutator-phase-inv s ≡ mutator-phase-inv-aux (mut-ghost-handshake-phase s) s
abbreviation mutators-phase-inv :: ('field, 'mut, 'ref) lsts-pred where
mutators-phase-inv ≡ ALLS m. 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 eq sys-fM Then black-heap Else white-heap)
and no-grey-refs )
sys-phase-inv-aux hp-IdleInit = no-black-refs
sys-phase-inv-aux hp-InitMark = (sys-fA neq sys-fM imp no-black-refs)
sys-phase-inv-aux hp-Mark = ⟨True⟩
sys-phase-inv-aux hp-IdleMarkSweep = ( (sys-phase eq ⟨ph-Idle⟩) or tso-pending-write gc
(mw-Phase ph-Idle)) imp no-grey-refs )

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

3.11 Lonely mutator assertions

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

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

(ML)

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

(ML)

3.12 The infamous termination argument.

We need to know that if the GC does not receive any further work to do at get-roots and
get-work, then there are no grey objects left. Essentially this encodes the stability property
that grey objects must exist for mutators to create grey objects.
Note that this is not invariant across the scan: it is possible for the GC to hold all the grey
references. The two handshakes transform the GC’s local knowledge that it has no more work
to do into a global property, or gives it more work.

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

40
definition (in ⊣) gc-W-empty-locs :: location set where

gc-W-empty-locs ≡
  idle-locs ∪ init-locs ∪ sweep-locs ∪ { "mark-read-fM", "mark-write-fA", "mark-end" }
  ∪ prefixed "mark-noop"
  ∪ prefixed "mark-loop-get-roots"
  ∪ prefixed "mark-loop-get-work"

(ML)

definition black-heap-locs ≡ { "sweep-idle", "idle-noop-mfence", "idle-noop-init-type" }

(ML)

definition no-grey-refs-locs ≡ black-heap-locs ∪ sweep-locs ∪ { "mark-end"

(ML)

definition (in gc) gc-W-empty-invL :: (field, 'mut, 'ref) gc-pred where
[inv]: gc-W-empty-invL ≡
  atS-gc (hs-get-roots-locs ∪ hs-get-work-locs) (ALLS m. mut-m.gc-W-empty-mut-inv m)
  and at-gc "mark-loop-get-roots-load-W" (empty sys-W imp no-grey-refs)  
  and at-gc "mark-loop-get-work-load-W" (empty sys-W imp no-grey-refs)
  and at-gc "mark-loop" (empty gc-W imp no-grey-refs)
  and atS-gc no-grey-refs-locs no-grey-refs
  and atS-gc gc-W-empty-locs (empty gc-W)⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩

(ML)

3.13 Sweep loop invariants

definition sweep-loop-locs ≡ prefixed "sweep-loop"

(ML)

definition (in gc) sweep-loop-invL :: (field, 'mut, 'ref) gc-pred where
[inv]: sweep-loop-invL ≡
  at-gc "sweep-loop-check" (not null gc-mark imp (λs. obj-at (λobj. Some (obj-mark obj) = gc-mark s (gc-tmp-ref s) s))
    and ( null gc-mark imp (marked ⊢ gc-tmp-ref or not valid-ref ≠ gc-tmp-ref) ) )
  and at-gc "sweep-loop-free" (not null gc-mark and the o gc-mark neg gc-fM and (λs. obj-at (λobj. Some (obj-mark obj) = gc-mark s (gc-tmp-ref s) s) )
    and at-gc "sweep-loop-ref-done" (marked ⊢ gc-tmp-ref or not valid-ref ⊢ gc-tmp-ref )
    and atS-gc sweep-loop-locs (ALLS r. not ⟨r⟩ in gc-refs imp (marked r or not valid-ref r))
  and atS-gc black-heap-locs (ALLS r. marked r or not valid-ref r)
  and atS-gc (prefixed "sweep-loop:" − { "sweep-loop-choose-ref"}) (gc-tmp-ref in gc-refs)⟨proof⟩⟨proof⟩⟨proof⟩

41
4 Top-level safety

4.1 Invariants

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

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

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

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

4.2 Initial conditions

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

- All objects on the heap are marked (have their flags equal to sys-fM, and there are no
grey references, i.e. the heap is uniformly black.

- The GC and system have the same values for $f_A$, $f_M$, etc. and the phase is $Idle$.
- No process holds the TSO lock and all write buffers are empty.
- All root-reachable references are backed by objects.

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

```
locale gc-system =  
  fixes initial-mark :: gc-mark
begin

definition gc-initial-state :: ('field, 'mut, 'ref) lst-pred where
  gc-initial-state s ≡
  fM s = initial-mark
  ∧ phase s = ph-Idle
  ∧ ghost-honorary-grey s = {}
  ∧ W s = {}

definition mut-initial-state :: ('field, 'mut, 'ref) lst-pred where
  mut-initial-state s ≡
  ghost-handshake-phase s = hp-IdleMarkSweep
  ∧ ghost-honorary-grey s = {}
  ∧ ghost-honorary-root s = {}
  ∧ W s = {}

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

abbreviation
  root-reachable y ≡ EXS m x. ⟨x⟩ in mut-m.mut-roots m and x reaches y

definition valid-refs :: ('field, 'mut, 'ref) lsts-pred where
  valid-refs ≡ ALLS y. root-reachable y imp valid-ref y

definition gc-system-init :: ('field, 'mut, 'ref) lsts-pred where
  gc-system-init ≡
```
The system consists of the programs and these constraints on the initial state.

**Abbreviation**  
\( \text{gc-system} :: ('\text{field}, '\text{mut}, '\text{ref}) \text{gc-system} \)  
\( \text{gc-system} \equiv (\text{gc-pgms}, \text{gc-system-init})(\text{proof})(\text{proof})(\text{proof})(\text{proof})(\text{proof})(\text{proof})(\text{proof})(\text{proof})(\text{proof})(\text{proof}) \)

**Theorem**  
\( \text{inv: } s \in \text{reachable-states gc-system} \implies I (\text{mkP } s) \)

Our headline safety result follows directly.

**Corollary**  
\( s \in \text{reachable-states gc-system} \implies \text{valid-refs } (\text{mkP } s) \)

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**  
\textit{Concrete-heap}

**Imports**  
\textit{src/HOL/Library/Saturated}

**Begin**

**Type-Synonyms**  
\( \text{field} = 3 \)  
\( \text{mut} = 2 \)  
\( \text{ref} = 5 \)

\( \text{Concrete-local-state} = (\text{field}, \text{mut}, \text{ref}) \text{ local-state} \)

\( \text{Clsts} = (\text{field}, \text{mut}, \text{ref}) \text{ lsts} \)

**Abbreviation**  
\( \text{Mut-common-init-state :: Concrete-local-state where} \)
\( \text{Mut-common-init-state} \equiv \text{undefined} \)  
\( \text{ghost-handshake-phase} := \text{hp-IdleMarkSweep}, \text{ghost-honorary-grey} := \{\}, \text{ghost-honorary-root} := \{\}, \text{roots} := \{\}, W := \{\} \)

**Context**  
\textit{gc-system}

**Begin**

**Abbreviation**  
\( \text{Sys-init-heap :: ref} \Rightarrow (\text{field}, \text{ref}) \text{ object option where} \)
\( \text{Sys-init-heap} \equiv \)  
\[ \theta \mapsto \emptyset \text{ obj-mark} = \text{initial-mark}, \]
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

45
\[ fA := \text{initial-mark} \]
\[ \text{phase} := \text{ph-Idle} \]
\[ W := \{ \} \]
\[ \text{ghost-honorary-grey} := \{ \} \]

primrec lookup :: \('k \times 'v\) list \Rightarrow 'v \Rightarrow 'k \Rightarrow 'v where
\[ \text{lookup} \; [] \; v0 \; k = v0 \]
\[ \text{lookup} \; (kv \# kvs) \; v0 \; k = (\text{if} \; \text{fst} \; kv = k \; \text{then} \; \text{snd} \; kv \; \text{else} \; \text{lookup} \; kvs \; v0 \; k) \]

abbreviation muts-init-states :: (mut \times \text{concrete-local-state}) list where
muts-init-states \equiv \[ (0, \text{mut-init-state0}), (1, \text{mut-init-state1}), (2, \text{mut-init-state2}) \]

abbreviation init-state :: clsts where
init-state \equiv \lambda p. \text{case} \; p \; \text{of}
\[ \text{gc} \Rightarrow \text{gc-init-state} \]
\[ \text{sys} \Rightarrow \text{sys-init-state} \]
\[ \text{mutator} \; m \Rightarrow \text{lookup} \; \text{muts-init-states} \; \text{mut-common-init-state} \; m \]

lemma
gc-system-init init-state \langle proof \rangle

end

References


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