Relaxing Safely: Verified On-the-Fly Garbage Collection for 
\textit{x86-TSO}

Peter Gammie, Tony Hosking and Kai Engelhardt

September 19, 2015

\textbf{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 \textit{x86-TSO}.

\textbf{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 ......................... 24
  3.6 Handshake phases .................................................... 25
  3.7 Object colours, reference validity, worklist validity .............. 31
  3.8 Mark Object .......................................................... 32
  3.9 The strong-tricolour invariant ..................................... 37
  3.10 Invariants ............................................................ 39
  3.11 Lonely mutator assertions .......................................... 40
  3.12 The infamous termination argument ................................ 40
  3.13 Sweep loop invariants .............................................. 41
1 Introduction

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

\texttt{type-synonym location = char list}

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

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

\texttt{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 \( fA \), \( fM \), \( phase \) and \( heap \) variables are subject to the TSO memory model, as are all heap operations.

\[
\text{record } (\text{\textquoteleft field\textquoteright, \textquoteleft mut\textquoteright, \textquoteleft ref\textquoteright}) \text{\ local-state } = \\
\quad \text{— System-specific fields} \\
\quad \text{heap :: \textquoteleft ref \textrighthook\ (\textquoteleft field\textquoteright, \textquoteleft ref\textquoteright) object option} \\
\quad \text{— TSO memory state} \\
\quad \text{mem-write-buffers :: \textquoteleft mut process-name \textrighthook\ (\textquoteleft field\textquoteright, \textquoteleft ref\textquoteright) mem-write-action list} \\
\quad \text{mem-lock :: \textquoteleft mut process-name option} \\
\quad \text{— The state of the handshakes} \\
\quad \text{handshake-type :: handshake-type} \\
\quad \text{handshake-pending :: \textquoteleft mut \textrighthook\ bool} \\
\quad \text{— Ghost state} \\
\quad \text{ghost-handshake-in-sync :: \textquoteleft mut \textrighthook\ bool} \\
\quad \text{ghost-handshake-phase :: handshake-phase} \\
\quad \text{— Mutator-specific temporaries} \\
\quad \text{new-ref :: \textquoteleft ref option} \\
\quad \text{roots :: \textquoteleft ref set} \\
\quad \text{ghost-honorary-root :: \textquoteleft ref set} \\
\quad \text{— Garbage collector-specific temporaries} \\
\quad \text{field-set :: \textquoteleft field set} \\
\quad \text{mut :: \textquoteleft mut} \\
\quad \text{muts :: \textquoteleft mut set} \\
\quad \text{— Local variables used by multiple processes} \\
\quad \text{fA :: gc-mark} \\
\quad \text{fM :: gc-mark} \\
\quad \text{cas-mark :: gc-mark option} \\
\quad \text{field :: \textquoteleft field} \\
\quad \text{mark :: gc-mark option} \\
\quad \text{phase :: gc-phase} \\
\quad \text{tmp-ref :: \textquoteleft ref} \\
\quad \text{ref :: \textquoteleft ref option} \\
\quad \text{refs :: \textquoteleft ref set} \\
\quad \text{W :: \textquoteleft ref set} \\
\quad \text{— Ghost state} \\
\quad \text{ghost-honorary-grey :: \textquoteleft ref set} \\
\]

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

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

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

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

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

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

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

We instantiate CIMP’s types as follows:


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

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

```
locale mut-m = fixes m :: 'mut
locale mut-m' = mut-m + fixes m' :: 'mut assumes mm'[iff]: m ≠ 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. \{s\})
notation lock (\{\} lock)

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

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

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

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

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

abbreviation add-to-W :: location ⇒ ((field, 'mut, 'ref) local-state ⇒ 'ref) ⇒ (field, 'mut, 'ref) gc-com where
  add-to-W l r ≡ \{l\} \{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 *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 ⟨ph-Idle⟩ THEN
  (∗ CAS: claim object ∗)
{l @ "-mo-co-lock"} lock ;;
{l @ "-mo-co-cmark"} read-mark (the ◦ ref) cas-mark-update ;;
{l @ "-mo-co-ctest"} IF cas-mark eq mark THEN
{l @ "-mo-co-mark"} write-mark (the ◦ ref) fM
FI ;;
{l @ "-mo-co-unlock"} unlock ;;
{l @ "-mo-co-won"} IF cas-mark eq mark THEN
  {l @ "-mo-co-W"} add-to-W (the ◦ ref)
FI
FI
FI
FI

**end**

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

### 2.2 Handshakes

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

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

The system and mutators track handshake phases using ghost state.

**abbreviation** hp-step :: handshake-type ⇒ handshake-phase ⇒ handshake-phase where

hp-step ht ≡
case ht of
\[
ht-\text{NOOP} \Rightarrow \text{handshake-step} \\
| \ht-\text{GetRoots} \Rightarrow \text{handshake-step} \\
| \ht-\text{GetWork} \Rightarrow \text{id}
\]

context syst

begin

definition handshake :: ('field, 'mut, 'ref) gc-com where

handshake ≡

\[
\begin{align*}
\{"\text{sys-hs-set-type}"\} & \text{Response} \\
(\lambda s. \{ s[\text{handshake-type} := \text{ht}, \\
\text{ghost-handshake-in-sync} := \langle \text{False} \rangle, \\
\text{ghost-handshake-phase} := \text{hp-step ht (ghost-handshake-phase s)} \}, \\
\text{mv-Void} \}) \\
& \text{| } \text{ht. req = (gc, ro-hs-gc-set-type ht) } \}
\end{align*}
\]

\[
\begin{align*}
\{"\text{sys-hs-mut-reqs}"\} & \text{Response} \\
(\lambda s. \{ s[\text{handshake-pending} := \langle \text{handshake-pending s}(m := \text{True}) \}, \\
\text{mv-Void} \}) \\
& \text{| } \text{m. req = (gc, ro-hs-gc-set-pending m) } \}
\end{align*}
\]

\[
\begin{align*}
\{"\text{sys-hs-gc-done}"\} & \text{Response} \\
(\lambda s. \{ s, \text{mv-Bool} (\neg \text{handshake-pending s m}) \\
& \text{| } \text{m. req = (gc, ro-hs-gc-read-pending m) } \}
\end{align*}
\]

\[
\begin{align*}
\{"\text{sys-hs-gc-load-W}"\} & \text{Response} \\
(\lambda s. \{ s[\text{W} := \{\}, \text{mv-Refs W s}) \\
& \text{| } \text{-::unit. req = (gc, ro-hs-gc-load-W) } \}
\end{align*}
\]

\[
\begin{align*}
\{"\text{sys-hs-mut}"\} & \text{Response} \\
(\lambda s. \{ s[\text{W} := \{\}, \text{mv-Void} \}) \\
& \text{| } \text{m. req = (mutator m, ro-hs-mut-read-type (handshake-type s)) \\
\text{\textwedge handshake-pending s m } \}
\end{align*}
\]

\[
\begin{align*}
\{"\text{sys-hs-mut-done}"\} & \text{Response} \\
(\lambda s. \{ s[\text{handshake-pending} := \langle \text{handshake-pending s}(m := \text{False}), \\
\text{W} := \text{W s U W'}, \\
\text{ghost-handshake-in-sync} := \langle \text{ghost-handshake-in-sync s}(m := \text{True}) \}, \\
\text{mv-Void} \}) \\
& \text{| } \text{m W'. req = (mutator m, ro-hs-mut-done W') } \}
\end{align*}
\]

end

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

context mut-m

begin

abbreviation mark-object :: location ⇒ ('field, 'mut, 'ref) gc-com where

\[
\begin{align*}
\{l\} & \text{mark-object} \equiv \text{mark-object-fn (mutator m) l}
\end{align*}
\]

abbreviation mfence :: location ⇒ ('field, 'mut, 'ref) gc-com where

\[
\begin{align*}
\{l\} & \text{MFENCE} \equiv \{l\} \text{ Request (\lambda s. (mutator m, ro-MFENCE)) (\lambda s. \{s\})}
\end{align*}
\]
abbreviation hs-read-type :: location \Rightarrow \text{handshake-type} \Rightarrow (\text{field, mut, ref}) \text{ gc-com} (\text{hs'-read'-type}) \quad \text{where}
\begin{align*}
\{l\} \text{ hs-read-type} ht & \equiv \{l\} \text{ Request} (\lambda s. (\text{mutator} m, \text{ro-hs-mut-read-type} ht)) (\lambda- \ s. \ \{s\}) \\
\end{align*}

abbreviation hs-noop-done :: location \Rightarrow (\text{field, mut, ref}) \text{ gc-com} (\text{hs'-noop'-done}) \quad \text{where}
\begin{align*}
\{l\} \text{ hs-noop-done} & \equiv \{l\} \text{ Request} (\lambda s. (\text{mutator} m, \text{ro-hs-mut-done} \ \{})) \\
& (\lambda- \ s. \ \{s\} \text{ ghost-handshake-phase} := \text{handshake-step} (\text{ghost-handshake-phase} s) \ ))
\end{align*}

abbreviation hs-get-roots-done :: location \Rightarrow (\text{field, mut, ref}) \text{ local-state} \Rightarrow \text{ref set) \Rightarrow} (\text{field, mut, ref}) \text{ gc-com} (\text{hs'-get-roots'-done}) \quad \text{where}
\begin{align*}
\{l\} \text{ hs-get-roots-done} & \equiv \{l\} \text{ Request} (\lambda s. (\text{mutator} m, \text{ro-hs-mut-done} (\text{wl s}))) \\
& (\lambda- \ s. \ \{s\} W := \{\}, \text{ghost-handshake-phase} := \text{handshake-step} (\text{ghost-handshake-phase} s) \ ))
\end{align*}

abbreviation hs-get-work-done :: location \Rightarrow (\text{field, mut, ref}) \text{ local-state} \Rightarrow \text{ref set) \Rightarrow} (\text{field, mut, ref}) \text{ gc-com} (\text{hs'-get-work'-done}) \quad \text{where}
\begin{align*}
\{l\} \text{ hs-get-work-done} & \equiv \{l\} \text{ Request} (\lambda s. (\text{mutator} m, \text{ro-hs-mut-done} (\text{wl s}))) \\
& (\lambda- \ s. \ \{s\} W := \{\} ))
\end{align*}

definition handshake :: (\text{field, mut, ref}) \text{ gc-com} \quad \text{where}
\begin{align*}
\text{handshake} & \equiv \\
\{\text{hs-noop}\} & \text{ hs-read-type} \ \text{ht-NOOP} ;; \\
\{\text{hs-noop-mfence}\} & \text{ MFENCE} ;; \\
\{\text{hs-noop-done}\} & \text{ hs-noop-done} \\
\end{align*}
\begin{align*}
\{\text{hs-get-roots}\} & \text{ hs-read-type} \ \text{ht-GetRoots} ;; \\
\{\text{hs-get-roots-mfence}\} & \text{ MFENCE} ;; \\
\{\text{hs-get-roots-refs}\} & \text{ `ref := `roots} ;; \\
\{\text{hs-get-roots-loop}\} & \text{ WHILE not empty refs DO} \\
\{\text{hs-get-roots-loop-choose-ref}\} & \text{ `ref := \text{Some `refs} ;;} \\
\{\text{hs-get-roots-loop}\} & \text{ mark-object} ;; \\
\{\text{hs-get-roots-loop-done}\} & \text{ `refs := (`refs \ \{-\text{the `ref}\})} \quad \text{OD} ;; \\
\{\text{hs-get-roots-done}\} & \text{ hs-get-roots-done} W \\
\end{align*}
\begin{align*}
\{\text{hs-get-work}\} & \text{ hs-read-type} \ \text{ht-GetWork} ;; \\
\{\text{hs-get-work-mfence}\} & \text{ MFENCE} ;; \\
\{\text{hs-get-work-done}\} & \text{ hs-get-work-done} W \\
\end{align*}
\end{definition}

end

The garbage collector’s side of the interface.

class 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 (\mut s\)))
    (λmv s. \{ s\ \muts\ := \muts\ s - \{ \mut\ s | done. mv = mv-Bool done ∧ done \} \})
OD

abbreviation load-W :: location ⇒ ('field, 'mut, 'ref) gc-com (\l\) load'-W') where
{l} load-W ≡ \l\ 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 @ ".mfence"] MFENCE ::
l handshake-init ht-NOOP ::
\[ \text{handshake-done} \]

\textbf{definition}

\[ \text{handshake-get-roots} :: \text{location} \Rightarrow (\text{'field}, \text{'mut}, \text{'ref}) \text{gc-com} (\text{\{\}} \text{handshake}'-get'-roots) \]

\textbf{where}

\[ \text{\{\} handshake-get-roots} \equiv \]

\[ \text{\{\} handshake-init ht-GetRoots} ;; \]

\[ \text{\{\} handshake-done} ;; \]

\[ \text{\{\} load-W} \]

\textbf{definition}

\[ \text{handshake-get-work} :: \text{location} \Rightarrow (\text{'field}, \text{'mut}, \text{'ref}) \text{gc-com} (\text{\{\}} \text{handshake}'-get'-work) \]

\textbf{where}

\[ \text{\{\} handshake-get-work} \equiv \]

\[ \text{\{\} handshake-init ht-GetWork} ;; \]

\[ \text{\{\} handshake-done} ;; \]

\[ \text{\{\} load-W} \]

\textbf{end}

\[ \text{2.3 The system process} \]

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.

\textbf{definition}

\[ \text{not-blocked} :: (\text{'field}, \text{'mut}, \text{'ref}) \text{local-state} \Rightarrow \text{'mut process-name} \Rightarrow \text{bool} \]

\textbf{where}

\[ \text{not-blocked s p} \equiv \text{case mem-lock s of None } \Rightarrow \text{True} \mid \text{Some p'} \Rightarrow \text{p = p'} \]

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

\textbf{definition}

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

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

\text{case wact of}

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

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

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

\[ | \text{mw-\text{fA gc-mark } } \Rightarrow s([\text{fA := gc-mark}]) \]
definition
fold-writes :: ('field, 'ref) mem-write-action list ⇒ ('field, 'mut, 'ref) local-state ⇒ ('field, 'mut, 'ref) local-state
where
fold-writes ws ≡ fold (λw. op ◦ (do-write-action w)) ws id

abbreviation
processors-view-of-memory :: 'mut process-name ⇒ ('field, 'mut, 'ref) local-state ⇒ ('field, 'mut, 'ref) local-state
where
processors-view-of-memory p s ≡ fold-writes (mem-write-buffers s p) s

definition
do-read-action :: ('field, 'ref) mem-read-action ⇒ ('field, 'mut, 'ref) local-state ⇒ ('field, 'ref) response
where
do-read-action ract ≡ λs.
case ract of
mr-Ref r f ⇒ mv-Ref (heap s r >>= (λobj. obj-fields obj f))
| mr-Mark r ⇒ mv-Mark (map-option obj-mark (heap s r))
| mr-Phase ⇒ mv-Phase (phase s)
| mr-fM ⇒ mv-Mark (Some (fM s))
| mr-fA ⇒ mv-Mark (Some (fA s))

definition
sys-read :: 'mut process-name ⇒ ('field, 'ref) mem-read-action ⇒ ('field, 'mut, 'ref) local-state ⇒ ('field, 'ref) response
where
sys-read p ract ≡ do-read-action ract ◦ 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
mem-TSO :: ('field, 'mut, 'ref) gc-com
where
$\text{mem-TSO} \equiv$

\[
\{"\text{sys-read}\}\ \text{Response}\ (\lambda\text{req}\ s.\ \{\text{s, sys-read } p \text{ mr } s\}
\begin{array}{c}
p \text{ mr. req } = (p, \text{ ro-Read mr}) \land \text{ not-blocked } s\ p \\ p \text{ mr. req } = (p, \text{ ro-Write w})\end{array}\}, \text{mv-Void})
\]

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

\[\sqcup\{"\text{sys-mfence}\}\ \text{Response}\ (\lambda\text{req}\ s.\ \{\text{s, mem-write-buffers } := (\text{mem-write-buffers } s)p := \text{mem-write-buffers } s \ p \ @ \ [w])\}, \text{mv-Void})
\begin{array}{c}
p \text{ w. req } = (p, \text{ ro-MFENCE}) \land \text{ mem-write-buffers } s \ p = \emptyset\end{array}\]

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

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

\[\sqcup\{"\text{sys-dequeue-write-buffer}\}\ \text{LocalOp}\ (\lambda s.\ \{\text{do-write-action } w s\}(\text{mem-write-buffers } := (\text{mem-write-buffers } s)(p := \text{mem-write-buffers } s \ p \ @ \ [w])\})
\begin{array}{c}
p \text{ w ws. mem-write-buffers } s \ p = w \# ws \land \\
\text{ not-blocked } s \ p \land p \neq \text{sys}\}\}
\]

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

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

**definition**

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

**where**

\[
\text{alloc } \equiv \{"\text{sys-alloc}\}\ \text{Response}\ (\lambda\text{req}\ s.
\begin{array}{c}
\{\text{s, heap } := (\text{heap } s)(r := \text{Some } (\text{fA } s, \text{obj-fields } = (\text{None } [])) )\}, \text{mv-Ref}
\begin{array}{c}
\text{Some } r\end{array}\}
\begin{array}{c}
\text{r, r } \notin \text{ dom } (\text{heap } s) \land \text{snd req } = \text{ro-Alloc}\end{array}\}
\]

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

**definition**

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

**where**

\[
\text{free } \equiv \{"\text{sys-free}\}\ \text{Response}\ (\lambda\text{req}\ s.
\begin{array}{c}
\{\text{s,heap } := (\text{heap } s)(r := \text{None})\}, \text{mv-Void}\}
\begin{array}{c}
r\text{ snd req } = \text{ro-Free } r\end{array}\}
\]

The top-level system process.

**definition**

\[
\text{com }::= (\text{field}, \text{mut, ref})\ \text{gc-com}
\]
where
\[
\text{com} \equiv \\
\text{LOOP DO} \\
\text{mem-\TSO} \\
\sqcup \text{alloc} \\
\sqcup \text{free} \\
\sqcup \text{handshake} \\
\text{OD}
\]

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.

c\text{ontext mut-m}

begin

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

abbreviation (in −) mut-alloc :: 'mut \Rightarrow ('field, 'mut, 'ref) gc-com where

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

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

\[
\text{alloc} \equiv \ \text{mut-alloc} \ m
\]

The mutator can always discard any references it holds.

abbreviation discard :: ('field, 'mut, 'ref) gc-com where

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

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

Dereferencing a reference can increase the set of mutator roots.

abbreviation load :: ('field, 'mut, 'ref) gc-com where

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

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

**abbreviation**

```
mut-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 (• defer)
```

**where**

```
{l} defer r f upd ≡ {l} Request (λs. (mutator m, ReadRef (r s) (f s)))
(λmv s. { upd (opt-r′) (s(ghost-honorary-root := set-option opt-r')) | opt-r'. mv = mv-Ref opt-r' })
```

**abbreviation**

```
write-ref :: location
⇒ (('field, 'mut, 'ref) local-state ⇒ 'ref)
⇒ (('field, 'mut, 'ref) local-state ⇒ 'field)
⇒ (('field, 'mut, 'ref) local-state ⇒ 'ref option)
⇒ ('field, 'mut, 'ref) gc-com (• defer)
```

**where**

```
{l} write-ref r f r′ ≡ {l} Request (λs. (mutator m, WriteRef (r s) (f s) (r′ s))) (λs. s(ghost-honorary-root := {}))
```

**definition**

```
store :: ('field, 'mut, 'ref) gc-com
where
store ≡
(* Choose vars for: ref→field := new-ref *)
{l"store-choose"} LocalOp (λs. { s| tmp-ref := r, field := f, new-ref := r' |}
| r f r′. r ∈ roots s ∧ r′ ∈ Some ' roots s ∪ {None} }) ;;
(* Mark the reference we’re about to overwrite. Does not update roots. *)
{l"deref-del"} deref tmp-ref field ref-update ;;
{l"store-del"} mark-object ;;
(* Mark the reference we’re about to insert. *)
{l"lop-store-ins"} 'ref := 'new-ref ;;
{l"store-ins"} mark-object ;;
{l"write-ref"} 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 :: ('field, 'mut, 'ref) gc-com
where
com ≡
```
LOOP DO
   \{"mut local computation"\} SKIP
   alloc
   discard
   load
   store
   \{"mut mfence"\} MFENCE
end

2.5 Garbage collector

We abstract the primitive actions of the garbage collector thread.

abbreviation
gc-deref :: location
   ⇒ (('field, 'mut, 'ref) local-state ⇒ 'ref)
   ⇒ (('field, 'mut, 'ref) local-state ⇒ 'field)
   ⇒ (('ref option ⇒ 'ref option) ⇒ ('field, 'mut, 'ref) local-state ⇒ ('field, 'mut, 'ref) local-state) ⇒ ('field, 'mut, 'ref) gc-com
where
   gc-deref l r f upd ≡ \{l\} Request (\λ s. (gc, ReadRef (r s) (f s)))
      (\λ mv 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 (\λ s. (gc, ReadMark (r s)))
      (\λ mv 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 (\-update-name q)
   \{l\} 'm := 'r→flag => CONST gc-read-mark l r (\-update-name m)

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

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

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

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

(* sweep *)

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

end

primrec
gc-pgms :: "mut process-name \Rightarrow ("field, "mut, "ref) gc-com"
where
gc-pgms (mutator m) = mut-m.com m
\mid gc-pgms gc = gc.com
\mid 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 ⇒ ('field, 'mut, 'ref) gc-pred ⇒ bool (\{} P \} - \{} Q \})
where
\{P\} p \{Q\} ≡ ∀ (c, afts) ∈ vcg-fragments (gc-pgms p). (gc-pgms, p, afts ⊨ \{P\} c \{Q\})

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

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

3.3 Functions and predicates

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

**abbreviation** is-mw-Mark w ≡ ∃ r fl. w = mw-Mark r fl
**abbreviation** is-mw-Mutate w ≡ ∃ r f r'. w = mw-Mutate r f r'
**abbreviation** is-mw-fA w ≡ ∃ fl. w = mw-fA fl
**abbreviation** is-mw-fM w ≡ ∃ fl. w = mw-fM fl
**abbreviation** is-mw-Phase w ≡ ∃ ph. w = mw-Phase ph

**abbreviation** (input) pred-in-W :: 'ref ⇒ 'mut process-name ⇒ ('field, 'mut, 'ref) lsts-pred (infix in\textsuperscript{\{} 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\textsuperscript{'-ghost'-'honorary'-'grey} 50) where
\textit{r in-ghost-honorary-grey p} \equiv \lambda s. \textit{r} \in \textit{ghost-honorary-grey (s p)}

\textbf{context} \ gc
\textbf{begin}

\textbf{abbreviation}
\textit{valid-gc-syn} :: (\textit{field, mut, ref}) gc-loc-comp \Rightarrow (\textit{field, mut, ref}) gc-pred \Rightarrow (\textit{field, mut, ref}) gc-com \Rightarrow (\textit{field, mut, ref}) gc-pred \Rightarrow \textit{bool}
\hspace{1em}(- \models \{ \cdot \}/ \cdot/ \{ \cdot \})

\textbf{where}
\textit{afts} \models \{ \textit{P} \} c \{ \textit{Q} \} \equiv \textit{gc-pgms}, \textit{gc}, \textit{afts} \models \{ \textit{P} \} c \{ \textit{Q} \}

\textbf{abbreviation} \textit{valid-gc-inv-syn} :: (\textit{field, mut, ref}) gc-loc-comp \Rightarrow (\textit{field, mut, ref}) gc-pred \Rightarrow (\textit{field, mut, ref}) gc-com \Rightarrow (\textit{field, mut, ref}) gc-pred \Rightarrow (\textit{field, mut, ref}) gc-pred \Rightarrow \textit{bool}
\hspace{1em}(- \models \{ \cdot \}/ \cdot/ \{ \cdot \})

\textbf{where}
\textit{afts} \models \{ \textit{P} \} c \equiv \textit{afts} \models \{ \textit{P} \} c \{ \textit{P} \}

\textbf{end}

\textbf{abbreviation} \textit{gc-cas-mark s} \equiv \textit{cas-mark (s gc)}
\textbf{abbreviation} \textit{gc-fM s} \equiv \textit{fM (s gc)}
\textbf{abbreviation} \textit{gc-field s} \equiv \textit{field (s gc)}
\textbf{abbreviation} \textit{gc-mark s} \equiv \textit{mark (s gc)}
\textbf{abbreviation} \textit{gc-mut s} \equiv \textit{mut (s gc)}
\textbf{abbreviation} \textit{gc-muts s} \equiv \textit{muts (s gc)}
\textbf{abbreviation} \textit{gc-phase s} \equiv \textit{phase (s gc)}
\textbf{abbreviation} \textit{gc-tmp-ref s} \equiv \textit{tmp-ref (s gc)}
\textbf{abbreviation} \textit{gc-ghost-honorary-grey s} \equiv \textit{ghost-honorary-grey (s gc)}
\textbf{abbreviation} \textit{gc-ref s} \equiv \textit{ref (s gc)}
\textbf{abbreviation} \textit{gc-refs s} \equiv \textit{refs (s gc)}
\textbf{abbreviation} \textit{gc-the-ref} \equiv \textit{the \circ gc-ref}
\textbf{abbreviation} \textit{gc-W s} \equiv \textit{W (s gc)}

\textbf{abbreviation} \textit{at-gc} :: \textit{location} \Rightarrow (\textit{field, mut, ref}) \textit{lsts-pred} \Rightarrow (\textit{field, mut, ref}) gc-pred
\hspace{1em}where
\textit{at-gc l P} \equiv \textit{at gc l imp LSTP P}

\textbf{abbreviation} \textit{atS-gc} :: \textit{location set} \Rightarrow (\textit{field, mut, ref}) \textit{lsts-pred} \Rightarrow (\textit{field, mut, ref}) gc-pred
\hspace{1em}where
\textit{atS-gc ls P} \equiv \textit{atS gc ls imp LSTP P}

\textbf{context} \ mut-m
\textbf{begin}

\textbf{abbreviation}
\textit{valid-mut-syn} :: (\textit{field, mut, ref}) gc-loc-comp \Rightarrow (\textit{field, mut, ref}) gc-pred \Rightarrow (\textit{field, mut, ref}) gc-com \Rightarrow (\textit{field, mut, ref}) gc-pred \Rightarrow \textit{bool}
valid-mut-inv-syn :: ('field, 'mut, 'ref) gc-loc-comp ⇒ ('field, 'mut, 'ref) gc-pred
⇒ ('field, 'mut, 'ref) gc-com ⇒ bool (- ⊨ \{\} / -) where
  afts ⊨ \{P\} c \{Q\} ≡ gc-pgms, mutator m, afts ⊨ \{P\} c \{Q\}

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

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

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

end

context sys
begin

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

abbreviation valid-sys-inv-syn :: ('field, 'mut, 'ref) gc-loc-comp ⇒ ('field, 'mut, 'ref) gc-pred ⇒ ('field, 'mut, 'ref) gc-com ⇒ bool (- ⊨ \{\} / -) where
afts \models \{ P \} \; c \equiv afts \models \{ P \} \{ 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) lsts-pred ⇒ ('field, 'mut, 'ref) lsts-pred 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 w s ≡ w ∈ set (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 ≡ case r of None ⇒ (True) | Some r' ⇒ valid-ref r'

abbreviation pred-points-to :: 'ref ⇒ 'ref ⇒ ('field, 'mut, 'ref) lsts-pred (infix points-to 51) where
  x points-to y ≡ λs. obj-at (λobj. y ∈ ran (obj-fields 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 reaches y ≡ λs. (λx y. (x points-to y) s)** x y
The predicate 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
  obj-at-field-on-heap P r f ≡ case Option.map-option obj-fields (sys-heap s r) of
    None ⇒ False |
      Some fs ⇒ (case fs f of None ⇒ True |
        Some r' ⇒ P r')

3.4 Garbage collector locations.

definition idle-locs :: location set where
  idle-locs ≡ prefixed "idle"

definition init-locs :: location set where
  init-locs ≡ prefixed "init"

definition mark-locs :: location set where
  mark-locs ≡ prefixed "mark"

definition mark-loop-locs :: location set where
  mark-loop-locs ≡ prefixed "mark-loop"

definition sweep-locs :: location set where
  sweep-locs ≡ prefixed "sweep"

3.5 Coarse TSO invariants

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

abbreviation gc-writes :: ('field, 'ref) mem-write-action ⇒ bool where
  gc-writes w ≡ case w of mw-Mark - - ⇒ True | mw-Phase - ⇒ True | mw-fM - ⇒ True |
    mw-fA - ⇒ True | - ⇒ False

abbreviation mut-writes :: ('field, 'ref) mem-write-action ⇒ bool where
  mut-writes w ≡ case w of mw-Mutate - - ⇒ True | mw-Mark - ⇒ True | - ⇒ False
definition tso-writes-inv :: (field, mut, ref) lsts-pred where
tso-writes-inv ≡
  (ALLS w. tso-pending-write gc w imp (gc-writes w))
  and (ALLS m w. tso-pending-write (mutator m) w imp (mut-writes w))

3.5.1 Locations where the TSO lock is held

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

definition gc-tso-lock-locs :: location set where
gc-tso-lock-locs ≡ \{ mo-co-cmark, mo-co-ctest, mo-co-mark, mo-co-unlock \}.
suffixed l
local-setup ⟨⟨ Cimp.locset @\{ thm gc-tso-lock-locs-def \} ⟩⟩

definition (in gc) tso-lock-invL :: (field, mut, ref) gc-pred where
[inv]: tso-lock-invL ≡
  atS-gc gc-tso-lock-locs (tso-locked-by gc)
  and atS-gc (¬ gc-tso-lock-locs) (not tso-locked-by gc)

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

definition mut-tso-lock-locs ≡
  \{ mo-co-cmark, mo-co-ctest, mo-co-mark, mo-co-unlock \}.
suffixed l
local-setup ⟨⟨ Cimp.locset @\{ thm mut-tso-lock-locs-def \} ⟩⟩

definition (in mut-m) tso-lock-invL :: (field, mut, ref) gc-pred where
[inv]: tso-lock-invL ≡
  atS-mut mut-tso-lock-locs (tso-locked-by (mutator m))
  and atS-mut (¬ mut-tso-lock-locs) (not tso-locked-by (mutator m))

3.6 Handshake phases

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

definition hp-step-rel :: (bool × handshake-type × handshake-phase × handshake-phase) set
where
hp-step-rel ≡
  \{ True \} × \{ (ht-NOOP, hp, hp) | hp, hp ∈ \{ hp-Idle, hp-IdleInit, hp-InitMark, hp-Mark \} \}
  ∪ \{ (ht-GetRoots, hp-IdleMarkSweep, hp-IdleMarkSweep)
    , (ht-GetWork, hp-IdleMarkSweep, hp-IdleMarkSweep) \}
  ∪ \{ False \} × \{ (ht-NOOP, hp-Idle, hp-IdleMarkSweep)
    , (ht-NOOP, hp-IdleInit, hp-Idle) \}
  ∪ \{ (ht-NOOP, hp-InitMark, hp-IdleInit) \}
  ∪ \{ (ht-NOOP, hp-Mark, hp-InitMark) \}
definition handshake-phase-inv :: ('field, 'mut, 'ref) lists-pred where
handshake-phase-inv ≡ \( m \rightarrow (\text{sys-ghost-handshake-in-sync } m \otimes \text{sys-handshake-type} \otimes \text{sys-ghost-handshake-phase } m) \in \langle \text{hp-step-ref} \rangle \)
and (\( \text{sys-handshake-pending } m \implies \neg \text{sys-ghost-handshake-in-sync } m \))

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

definition hp-Idle-locs ≡
\( \langle \text{prefixed "idle-noop" - \{ "idle-noop-mfence", "idle-noop-init-type" \}} \rangle \)
∪ \( \{ "idle-read-fM", "idle-invert-fM", "idle-write-fM", "idle-flip-noop-mfence", "idle-flip-noop-init-type" \} \)
local-setup \( \langle\langle \text{Cimp.locset@\{thm hp-Idle-locs-def\}} \rangle\rangle \)

definition hp-IdleInit-locs ≡
\( \langle \text{prefixed "idle-flip-noop" - \{ "idle-flip-noop-mfence", "idle-flip-noop-init-type" \}} \rangle \)
∪ \( \{ "idle-phase-init", "init-noop-mfence", "init-noop-init-type" \} \)
local-setup \( \langle\langle \text{Cimp.locset@\{thm hp-IdleInit-locs-def\}} \rangle\rangle \)

definition hp-InitMark-locs ≡
\( \langle \text{prefixed "init-noop" - \{ "init-noop-mfence", "init-noop-init-type" \}} \rangle \)
∪ \( \{ "init-phase-mark", "mark-read-fM", "mark-write-fA", "mark-noop-mfence", "mark-noop-init-type" \} \)
local-setup \( \langle\langle \text{Cimp.locset@\{thm hp-InitMark-locs-def\}} \rangle\rangle \)

definition hp-Mark-locs ≡
\( \langle \text{prefixed "mark-noop" - \{ "mark-noop-mfence", "mark-noop-init-type" \}} \rangle \)
∪ \( \{ "mark-loop-get-roots-init-type" \} \)
local-setup \( \langle\langle \text{Cimp.locset@\{thm hp-Mark-locs-def\}} \rangle\rangle \)

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

definition hs-noop-locs ≡
\( \bigcup l \in \text{hs-noop-prefixes}. \text{prefixed } l - (\text{suffixed "-noop-mfence"} \cup \text{suffixed "-noop-init-type"}) \)
local-setup \( \langle\langle \text{Cimp.locset@\{thm hs-noop-locs-def\}} \rangle\rangle \)

definition hs-get-roots-locs ≡
\( \text{prefixed "mark-loop-get-roots"} - \{"mark-loop-get-roots-init-type"\} \)
local-setup \{ Cimp.locset @\{thm hs-get-roots-locs-def \} \}

definition hs-get-work-locs ≡
  prefixed "mark-loop-get-work" − \{"mark-loop-get-work-init-type"\}
local-setup \{ Cimp.locset @\{thm hs-get-work-locs-def \} \}

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

definition hs-init-loop-locs ≡ \bigcup l ∈ hs-prefixes. prefixed (l @ "-init-loop")
local-setup \{ Cimp.locset @\{thm hs-init-loop-locs-def \} \}

definition hs-done-loop-locs ≡ \bigcup l ∈ hs-prefixes. prefixed (l @ "-done-loop")
local-setup \{ Cimp.locset @\{thm hs-done-loop-locs-def \} \}

definition hs-done-locs ≡ \bigcup l ∈ hs-prefixes. prefixed (l @ "-done")
local-setup \{ Cimp.locset @\{thm hs-done-locs-def \} \}

definition hs-none-pending-locs ≡ − (hs-init-loop-locs ∪ hs-done-locs)
local-setup \{ Cimp.locset @\{thm hs-none-pending-locs-def \} \}

definition hs-in-sync-locs ≡
  (− ( \bigcup l ∈ hs-prefixes. prefixed (l @ "-init") ) ∪ hs-done-locs )
  ∪ ( \bigcup l ∈ hs-prefixes. \{l @ "-init-type"\} )
local-setup \{ Cimp.locset @\{thm hs-in-sync-locs-def \} \}

definition hs-out-of-sync-locs ≡ \bigcup l ∈ hs-prefixes. \{l @ "-init-muts"\}
local-setup \{ Cimp.locset @\{thm hs-out-of-sync-locs-def \} \}

definition hs-mut-in-muts-locs ≡ \bigcup l ∈ hs-prefixes. \{l @ "-init-loop-set-pending", l @ "-init-loop-done"\}
local-setup \{ Cimp.locset @\{thm hs-mut-in-muts-locs-def \} \}

definition hs-init-loop-done-locs ≡ \bigcup l ∈ hs-prefixes. \{l @ "-init-loop-done"\}
local-setup \{ Cimp.locset @\{thm hs-init-loop-done-locs-def \} \}

definition hs-init-loop-not-done-locs ≡
  hs-init-loop-locs − ( \bigcup l ∈ hs-prefixes. \{l @ "-init-loop-done"\} )
local-setup \{ Cimp.locset @\{thm hs-init-loop-not-done-locs-def \} \}

definition (in gc) handshake-invL :: \langle field, 'mut, 'ref \rangle gc-pred where
  invv: 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\})
and atS-gc hs-mut-in-muts-locs \((\text{gc-mut in gc-muts})\)
and atS-gc hs-init-loop-locs \((\text{ALLS m. not } \langle m \rangle \text{ in gc-muts imp sys-handshake-pending})\)
and atS-gc hs-init-loop-not-done-locs \((\text{ALLS m. } \langle m \rangle \text{ in gc-muts imp not sys-handshake-pending})\)
and atS-gc hs-init-loop-done-locs \((\text{(sys-handshake-pending } \triangledown \text{ gc-mut})\)
and not sys-ghost-handshake-in-sync m)
and atS-gc hs-done-locs \((\text{ALLS m. sys-handshake-pending m or sys-ghost-handshake-in-sync})\)
and atS-gc hs-done-loop-locs \((\text{ALLS m. not } \langle m \rangle \text{ in gc-muts imp not sys-handshake-pending})\)
and atS-gc hs-none-pending-locs \((\text{ALLS m. not sys-handshake-pending m})\)
and atS-gc hs-in-sync-locs \((\text{ALLS m. sys-ghost-handshake-in-sync})\)
and atS-gc hs-out-of-sync-locs \((\text{ALLS m. not sys-handshake-pending m and not sys-ghost-handshake-in-sync m})\)
and atS-gc hp-Idle-locs \((\text{sys-ghost-handshake-phase eq } \langle \text{hp-Idle} \rangle)\)
and atS-gc hp-IdleInit-locs \((\text{sys-ghost-handshake-phase eq } \langle \text{hp-IdleInit} \rangle)\)
and atS-gc hp-InitMark-locs \((\text{sys-ghost-handshake-phase eq } \langle \text{hp-InitMark} \rangle)\)
and atS-gc hp-IdleMarkSweep-locs \((\text{sys-ghost-handshake-phase eq } \langle \text{hp-IdleMarkSweep} \rangle)\)
and atS-gc hp-Mark-locs \((\text{sys-ghost-handshake-phase eq } \langle \text{hp-Mark} \rangle)\)

Local handshake phase invariant for the mutators.

**definition**

\[
\text{mut-no-pending-mutates-locs} \equiv \\
(\text{prefixed } "\text{hs-noop}" - \{ "\text{hs-noop}"", "\text{hs-noop-mfence}" \}) \\
\cup (\text{prefixed } "\text{hs-get-roots}" - \{ "\text{hs-get-roots}"", "\text{hs-get-roots-mfence}" \}) \\
\cup (\text{prefixed } "\text{hs-get-work}" - \{ "\text{hs-get-work}"", "\text{hs-get-work-mfence}" \})
\]

**local-setup** 
\[\langle Cimp.locset \, @\{ \text{thm mut-no-pending-mutates-locs-def} \} \rangle\]

**definition** \(\text{in (mut-m) handshake-invL :: } \langle \text{field, mut, ref} \rangle \text{ gc-pred where}
\]

[inv]: \(\text{handshake-invL} \equiv \)

\[
\text{ats-mut (prefixed } "\text{hs-noop-}" \text{) (sys-handshake-type eq } \langle \text{ht-NOOP} \rangle \text{ and sys-handshake-pending m)}\\
\text{ats-mut (prefixed } "\text{hs-get-roots-}" \text{) (sys-handshake-type eq } \langle \text{ht-GetRoots} \rangle \text{ and sys-handshake-pending m)}\\
\text{ats-mut (prefixed } "\text{hs-get-work-}" \text{) (sys-handshake-type eq } \langle \text{ht-GetWork} \rangle \text{ and sys-handshake-pending m)}\\
\text{ats-mut mut-no-pending-mutates-locs (list-null (iso-pending-mutate (mutator m)))}
\]

Relate \text{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
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, ′mut) mem-write-action list) set where
phase-rel ≡
{ (in-sync, hp, ph, [] | in-sync hp ph. handshake-phase-rel hp in-sync ph )
  ∪ ( {True} × { (hp-IdleInit, ph-Idle, [mw-Phase ph-Idle]),
       (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⟩

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" })

local-setup ⟨ ⟨ Cimp.locset @{thm no-pending-phase-locs-def} ⟩ ⟩

definition (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))

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.
\textbf{definition} \(fM\text{-rel}::(\text{bool} \times \text{handshake-phase} \times \text{gc-mark} \times \text{gc-mark} \times (\text{'field'}, \text{'ref'}) \text{ mem-write-action list} \times \text{bool}) \text{ set where} \\
\quad fM\text{-rel} = \\
\quad \{ \text{(in-sync, hp, } fM, [], l) | fM \text{ hp in-sync l, hp = hp-Idle } \rightarrow \neg \text{in-sync} \} \\
\quad \cup \{ \text{(in-sync, hp-Idle, } fM, fM', [], l) | fM fM' \text{ in-sync l, in-sync} \} \\
\quad \cup \{ \text{(in-sync, hp-Idle, } \neg fM, fM, [mw-FM (\neg fM)], False) | fM \text{ in-sync, in-sync} \}

\textbf{definition} \(fM\text{-rel-inv}::(\text{'field}, \text{'mut}, \text{'ref}) \text{lsts-pred where} \\
\quad fM\text{-rel-inv} \equiv (\text{ALLS} m. \text{sys-ghost-handshake-in-sync m}) \otimes \text{sys-ghost-handshake-phase} \otimes \text{gc-fM} \otimes \text{sys-fM} \otimes \text{ts-o-pending-fM gc} \otimes (\text{sys-mem-lock eq } \langle \text{Some gc} \rangle ) \text{ in } (fM\text{-rel})

\textbf{definition} \(fA\text{-rel}::(\text{bool} \times \text{handshake-phase} \times \text{gc-mark} \times \text{gc-mark} \times (\text{'field'}, \text{'ref'}) \text{ mem-write-action list} \text{ set where} \\
\quad fA\text{-rel} = \\
\quad \{ \text{(in-sync, hp-Idle, } fA, fM, []) | fA fM \text{ in-sync, } \neg \text{in-sync } \rightarrow fA = fM \} \\
\quad \cup \{ \text{(in-sync, hp-IdleInit, } fA, \neg fA, [], []) | fA \text{ in-sync, True} \} \\
\quad \cup \{ \text{(in-sync, hp-InitMark, } fA, \neg fA, [mw-fA (\neg fA)], fA \text{ in-sync, in-sync} \} \\
\quad \cup \{ \text{(in-sync, hp-InitMark, } fA, fM, []) | fA fM \text{ in-sync, } \neg \text{in-sync } \rightarrow fA \neq fM \} \\
\quad \cup \{ \text{(in-sync, hp-Mark, } fA, fA, []) | fA \text{ in-sync, True} \} \\
\quad \cup \{ \text{(in-sync, hp-IdleMarkSweep, } fA, fA, []) | fA \text{ in-sync, True} \}

\textbf{definition} \(fA\text{-rel-inv}::(\text{'field}, \text{'mut}, \text{'ref}) \text{lsts-pred where} \\
\quad fA\text{-rel-inv} \equiv (\text{ALLS} m. \text{sys-ghost-handshake-phase} \otimes \text{sys-fA} \otimes \text{gc-fM} \otimes \text{ts-o-pending-fA gc} \text{ in } (fA\text{-rel})

\textbf{definition} \(fM\text{-eq-locs}::\text{location set where} \\
\quad fM\text{-eq-locs} \equiv \langle \{ "idle-write-fM", "idle-flip-noop-mfence" \} \rangle

\textbf{local-setup} \langle \text{in } \rangle \langle \text{CLUimp.locset } @\{\text{thm fM-loc-set-def} \} \rangle

\textbf{definition} \(fM\text{-tso-empty-locs}::\text{location set where} \\
\quad fM\text{-tso-empty-locs} \equiv \langle \{ "idle-flip-noop-mfence" \} \rangle

\textbf{local-setup} \langle \text{in } \rangle \langle \text{CLUimp.locset } @\{\text{thm fM-tso-empty-locs-def} \} \rangle

\textbf{definition} \(fA\text{-tso-empty-locs}::\text{location set where} \\
\quad fA\text{-tso-empty-locs} \equiv \langle \{ "mark-noop-mfence" \} \rangle

\textbf{local-setup} \langle \text{in } \rangle \langle \text{CLUimp.locset } @\{\text{thm fA-tso-empty-locs-def} \} \rangle

\textbf{definition} \(fA\text{-eq-locs}::\text{location set where} \\
\quad fA\text{-eq-locs} \equiv \langle \{ "idle-read-fM", "idle-invert-fM" \} \cup \text{prefix } "idle-noop" \cup \{ \text{mark-locs } \rightarrow \{ "mark-read-fM", "mark-write-fA", "mark-noop-mfence" \} \cup \text{swEEP-locs} \rangle

\textbf{local-setup} \langle \text{CLUimp.locset } @\{\text{thm fA-eq-locs-def} \} \rangle

\textbf{definition} \(fA\text{-neq-locs}::\text{location set where} \\
\quad fA\text{-neq-locs} \equiv \langle \{ "idle-phase-init", "idle-write-fM", "mark-read-fM", "mark-write-fA" \} \cup \text{prefix } "idle-flip-noop" \rangle

30
\[ \bigcup \text{init-locs} \]

**local-setup** \( \langle \langle \text{Cimp}\cdot \text{locset} \oplus \{ \text{thm fA-neq-locs-def} \} \rangle \rangle \)

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

\[
\begin{align*}
\text{atS-gc } fM-f\text{eq-locs} & \equiv (\text{sysM eq gc-fM}) \\
\text{and at-gc } "\text{idle-write-fM}" & \equiv (\text{sysM neq gc-fM}) \\
\text{and at-gc } "\text{idle-flip-noop-mfence}" & \equiv (\text{sysM neq gc-fM imp (not list-null (tso-pending-fM gc))}) \\
\text{and atS-gc } fM-\text{tso-empty-locs} & \equiv (\text{list-null (tso-pending-fM gc)}) \\
\end{align*}
\]

\[
\begin{align*}
\text{and atS-gc } fA-\text{eq-locs} & \equiv (\text{sysA eq gc-fM}) \\
\text{and atS-gc } fA-\text{neq-locs} & \equiv (\text{sysA neq gc-fM}) \\
\text{and at-gc } "\text{mark-noop-mfence}" & \equiv (\text{sysA neq gc-fM imp (not list-null (tso-pending-fA gc))}) \\
\text{and atS-gc } fA-\text{tso-empty-locs} & \equiv (\text{list-null (tso-pending-fA gc)}) \\
\end{align*}
\]

### 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) \text{lsts-pred} \) where

\[
\text{marked } r s \equiv \text{obj-at} (\lambda \text{obj. obj-mark } \text{obj} = \text{sysM s}) \text{ r s}
\]

**abbreviation** white :: ‘ref \(\Rightarrow\) (‘field, ‘mut, ‘ref) \text{lsts-pred} \) where

\[
\text{white } r s \equiv \text{obj-at} (\lambda \text{obj. obj-mark } \text{obj} = (\neg \text{sysM s})) \text{ r s}
\]

**definition** \( WL :: \text{‘mut process-name} \Rightarrow (\text{‘field, ‘mut, ‘ref}) \text{lsts} \Rightarrow \text{‘ref set} \) where

\[
\text{WL } p \equiv \lambda s. \ W (s \ p) \cup \text{ghost-honorary-grey} (s \ p)
\]

**definition** grey :: ‘ref \(\Rightarrow\) (‘field, ‘mut, ‘ref) \text{lsts-pred} \) where

\[
\text{grey } r \equiv \text{EXS} p. (r) \text{ in WL } p
\]
definition black :: 'ref ⇒ ('field, 'mut, 'ref) lsts-pred where
black r ≡ marked r and not grey r

We show that if a mutator can load a reference into its roots (its working set of references),
then there is an object in the heap at that reference.

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

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

definition (in mut-m) tso-write-refs :: ('field, 'mut, 'ref) lsts ⇒ 'ref set where
tso-write-refs ≡ λs. \bigcup w ∈ set (sys-mem-write-buffers (mutator m) s). write-refs w

definition (in mut-m) reachable :: 'ref ⇒ ('field, 'mut, 'ref) lsts-pred where
reachable y ≡ EXS x. \langle x \rangle in (mut-roots union mut-ghost-honorary-root union tso-write-refs)
and x reaches y

definition grey-reachable :: 'ref ⇒ ('field, 'mut, 'ref) lsts-pred where
grey-reachable y ≡ EXS g. grey g and g reaches y

definition valid-refs-inv :: ('field, 'mut, 'ref) lsts-pred where
valid-refs-inv ≡ ALLS x. ((EXS m. mut-m. reachable m x) or grey-reachable x) imp valid-ref x

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

definition valid-W-inv :: ('field, 'mut, 'ref) lsts-pred where
valid-W-inv ≡ ALLS p q r.
(r in-W p or (sys-mem-lock neq (Some p) and r in-ghost-honorary-grey p) imp marked r)
and ((p ≠ q) imp not ((r) in WL p and (r) in WL q))
and (not (r in-ghost-honorary-grey p and r in-W q))
and (empty sys-ghost-honorary-grey)
and (ALLS fl. tso-pending-write p (mw-Mark r fl)
imp ( ⟨fl⟩ eq sys-fM
and r in-ghost-honorary-grey p
and tso-locked-by p
and white r
and tso-pending-mark p eq ⟨[mw-Mark r fl]⟩ ) )

32
3.8 Mark Object

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

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

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

begin

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

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

abbreviation (input) p-en-cond P ≡ p-ph-enabled imp P

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

abbreviation (input)
  p-valid-W-inv ≡ ((p-cas-mark neq p-mark or p-tso-no-pending-mark) imp marked ⊢ p-the-ref)
  and (tso-pending-mark p in (\l:: [[] , [mw-Mark (p-the-ref s) (p-fM s)]}))

abbreviation (input)
  p-mark-inv ≡ not null p-mark
  and ((\l:: obj-at (\obj. Some (obj-mark obj) = p-mark s) (p-the-ref s) s)
  or marked ⊢ p-the-ref)

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

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

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

definition mark-object-invL :: (field, mut, ref) gc-pred where
mark-object-invL \equiv
\begin{align*}
\text{at-p } & \text{"-mo-null" } \quad \langle \text{True} \rangle \\
\text{and at-p } & \text{"-mo-mark" } \quad \langle \text{p-valid-ref} \rangle \\
\text{and at-p } & \text{"-mo-fM" } \quad \langle \text{p-valid-ref and p-en-cond (p-mark-inv)} \rangle \\
\text{and at-p } & \text{"-mo-mtest" } \quad \langle \text{p-valid-ref and p-en-cond (p-mark-inv and p-valid-fM)} \rangle \\
\text{and at-p } & \text{"-mo-phase" } \quad \langle \text{p-valid-ref and p-mark neq Some } \odot \text{ p-fM and p-en-cond (p-mark-inv and p-valid-fM)} \rangle \\
\text{and at-p } & \text{"-mo-ptest" } \quad \langle \text{p-valid-ref and p-mark neq Some } \odot \text{ p-fM and p-en-cond (p-mark-inv and p-valid-fM)} \rangle \\
\text{and at-p } & \text{"-mo-co-lock" } \quad \langle \text{p-valid-ref and p-mark neq Some } \odot \text{ p-fM and p-tso-no-pending-mark} \rangle \\
\text{and at-p } & \text{"-mo-co-cmark" } \quad \langle \text{p-valid-ref and p-mark neq Some } \odot \text{ p-fM and p-tso-no-pending-mark} \rangle \\
\text{and at-p } & \text{"-mo-co-ctest" } \quad \langle \text{p-valid-ref and p-mark neq Some } \odot \text{ p-fM and p-mark-neq-mark} \rangle \\
\text{and at-p } & \text{"-mo-co-mark" } \quad \langle \text{p-cas-mark eq p-mark and p-valid-ref and p-valid-fM and white} \rangle \\
\text{and at-p } & \text{"-mo-co-unlock" } \quad \langle \text{p-ghg-inv and p-valid-ref and p-valid-fM and p-valid-W-inv} \rangle \\
\text{and at-p } & \text{"-mo-co-won" } \quad \langle \text{p-ghg-inv and p-valid-ref and p-valid-fM and marked } \uparrow \text{ p-the-ref and p-tso-no-pending-mutate} \rangle \\
\text{and at-p } & \text{"-mo-co-W" } \quad \langle \text{p-ghg-eq-ref and p-valid-ref and p-valid-fM and marked } \uparrow \text{ p-the-ref and p-tso-no-pending-mutate} \rangle \\
\end{align*}

end

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

interpretation gc-mark!: mark-object gc "mark-loop" \( \langle \text{True} \rangle \)
by standard (simp add: eq-imp-def)

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

interpretation mut-get-roots!: mark-object mutator m "hs-get-roots-loop" \( \langle \text{True} \rangle \) for m
by standard (simp add: eq-imp-def)

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

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

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

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

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

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

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

**definition** mut-hs-get-roots-loop-locs ≡
  prefixed "hs-get-roots-loop"
**local-setup** ≪ Cimp.locset @{thm mut-hs-get-roots-loop-locs-def} ≫

**definition** mut-hs-get-roots-loop-mo-locs ≡
  prefixed "hs-get-roots-loop-mo" ∪ {"hs-get-roots-loop-done"}
**local-setup** ≪ Cimp.locset @{thm mut-hs-get-roots-loop-mo-locs-def} ≫

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

**definition** mut-hs-not-hp-Idle-locs ≡
  ∪ pref ∈ mut-async-mark-object-prefixes.
  ∪ l∈{"mo-co-lock", "mo-co-cmark", "mo-co-ctest", "mo-co-mark", "mo-co-unlock", "mo-co-won", "mo-co-W"}. {pref @ "." @ l}
**local-setup** ≪ Cimp.locset @{thm mut-hs-not-hp-Idle-locs-def} ≫

**definition** mut-async-mo-ptest-locs ≡
  ∪ pref ∈ mut-async-mark-object-prefixes. {pref @ "-mo-ptest"}
**local-setup** ≪ Cimp.locset @{thm mut-async-mo-ptest-locs-def} ≫

**definition** mut-mo-ptest-locs ≡
  ∪ pref ∈ mut-async-mark-object-prefixes. {pref @ "-mo-ptest"}
**local-setup** ≪ Cimp.locset @{thm mut-mo-ptest-locs-def} ≫

**definition** mut-mo-valid-ref-locs ≡
  prefixed "store-del" ∪ prefixed "store-ins" ∪ {"deref-del", "lop-store-ins"}
**local-setup** ≪ Cimp.locset @{thm mut-mo-valid-ref-locs-def} ≫

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

**local-setup** { Cimp.locset \&\{\text{thm ghost-honorary-grey-empty-locs-def} \} }

**definition** (in mut-m) mark-object-invL :: ('field, 'mut, 'ref) gc-pred where
\[
\text{inv} : \text{mark-object-invL} \equiv \begin{cases}
\text{atS-mut mut-hs-get-roots-loop-locs} & (\text{mut-refs subseteq mut-roots and (ALLS r. (r) in mut-roots diff mut-refs imp marked r)})
\text{and atS-mut mut-hs-get-roots-loop-mo-locs} & (\text{not null mut-ref and mut-the-ref in mut-roots})
\text{and at-mut "hs-get-roots-loop-done"} & (\text{marked \& mut-the-ref})
\text{and at-mut "hs-get-roots-loop-mo-ptest"} & (\text{mut-phase neq (ph-Idle)})
\text{and at-mut "hs-get-roots-done"} & (\text{ALLS r. (r) in mut-roots imp marked r})
\text{and atS-mut mut-mo-valid-ref-locs} & (\text{not null mut-new-ref imp mut-the-new-ref in mut-roots})
\text{and (mut-temp-ref in mut-roots)}
\text{and at-mut "store-del-mo-null"} & (\text{not null mut-ref imp mut-the-ref in mut-ghost-honorary-root})
\text{and atS-mut (prefixed "store-del" \& \"store-del-mo-null\")} & (\text{mut-the-ref in mut-ghost-honorary-root})
\text{and atS-mut \"store-ins\"} & (\text{mut-ref eq mut-new-ref})
\text{and atS-mut (suffixed \"-mo-ptest\")} & (\text{mut-phase neq (ph-Idle) imp mut-ghost-handshake-phase neq (hp-Idle)})
\text{and atS-mut mut-hs-not-hp-Idle-locs} & (\text{mut-ghost-handshake-phase neq (hp-Idle)})
\text{and atS-mut mut-mo-ptest-locs} & (\text{mut-phase eq (ph-Idle) imp (mut-ghost-handshake-phase eq}}
\text{\{hp-IdleMarkSweep\}) or (\text{mut-ghost-handshake-phase eq}}
\text{and sys-phase eq (ph-Idle)\}))
\text{and atS-mut ghost-honorary-grey-empty-locs} & (\text{empty mut-ghost-honorary-grey})
\end{cases}
\]

(* insertion barrier *)
\[
\text{and at-mut "store-ins"} & (\text{mut-ghost-handshake-phase in \{hp-InitMark, hp-Mark\}})
\text{or (mut-ghost-handshake-phase eq (hp-IdleMarkSweep) and not null mut-new-ref imp marked \& mut-the-new-ref})
\]

(* deletion barrier *)
and atS-mut (prefixed "store-del-mo" \(\cup\) \{"lop-store-ins"\})
\[
(\text{(mut-ghost-handshake-phase eq } \langle\text{hp-Mark}\rangle \\
\text{or (mut-ghost-handshake-phase eq } \langle\text{hp-IdleMarkSweep}\rangle)
\]
and sys-phase neq \langle ph-Idle \rangle))
\[
(\lambda s. \forall \text{opt-r'. tso-pending-write (mutator m) (mw-Mutate (mut-tmp-ref s) (mut-field s) opt-r') s} \\
\text{imp (\lambda s. obj-at-field-on-heap (\lambda r. mut-ref s = Some r \lor marked r s) (mut-tmp-ref s) (mut-field s) s))}
\]
and at-mut "lop-store-ins" (\text{(mut-ghost-handshake-phase eq } \langle\text{hp-Mark}\rangle \\
\text{or (mut-ghost-handshake-phase eq } \langle\text{hp-IdleMarkSweep}\rangle)
\]
and sys-phase neq \langle ph-Idle \rangle))
\[
\text{and not null mut-ref} \\
\text{imp marked }\triangledown\text{ mut-the-ref })
\]
and atS-mut (prefixed "store-ins")
\[
(\text{(mut-ghost-handshake-phase eq } \langle\text{hp-Mark}\rangle \\
\text{or (mut-ghost-handshake-phase eq } \langle\text{hp-IdleMarkSweep}\rangle)
\]
and sys-phase neq \langle ph-Idle \rangle))
\[
(\lambda s. \forall \text{opt-r'. tso-pending-write (mutator m) (mw-Mutate (mut-tmp-ref s) (mut-field s) opt-r') s} \\
\text{imp (\lambda s. obj-at-field-on-heap (\lambda r'. marked r' s) (mut-tmp-ref s) (mut-field s) s))}
\]

The GC's use of \text{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.

\textbf{definition (in gc) obj-fields-marked-inv} :: (field, mut, ref) \text{lst-pred} where
\[\text{obj-fields-marked-inv} \equiv \text{ALLS f. } f \text{ in } (\neg \text{gc-field-set}) \text{ imp } (\lambda s. \text{obj-at-field-on-heap } (\lambda r. \text{marked } r s) (\text{gc-tmp-ref s} f s))\]

\textbf{definition obj-fields-marked-locs :: location set} where
\[\text{obj-fields-marked-locs} \equiv \{"mark-loop-mark-object-loop", "mark-loop-mark-choose-field", "mark-loop-mark-derref", "mark-loop-mark-field-done", "mark-loop-blacken"\} \\
\text{\cup prefixed "mark-loop-mo"}
\]
\textbf{local-setup} \(\langle\text{Cimp.locset }\oplus\{\text{thm obj-fields-marked-locs-def}\} \rangle\)

\textbf{definition (in gc) obj-fields-marked-invL} :: (field, mut, ref) gc-pred where
\[\text{[inv]: } \text{obj-fields-marked-invL} \equiv \text{atS-gc obj-fields-marked-locs } (\text{obj-fields-marked-inv and gc-tmp-ref in gc-W}) \]
\[\text{and atS-gc (prefixed "mark-loop-mo" } \cup\{"mark-loop-mark-field-done"\})\]
\[\text{\(\lambda\text{s. obj-at-field-on-heap } (\lambda r. \text{gc-ref s = Some r } \lor \text{marked r s) (gc-tmp-ref s) (gc-field s) s})\text{ imp valid-ref y})\]
\[\text{and at-gc "mark-loop-fields" } (\text{gc-tmp-ref in gc-W})\]

37
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

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

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

3.10 Invariants

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

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

primrec (in mut-m) mutator-phase-inv-aux :: handshake-phase ⇒ ('field, 'mut, 'ref) lsts-pred
where
mutator-phase-inv-aux hp-Idle = (True)
| mutator-phase-inv-aux hp-IdleInit = no-black-refs
| 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) 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")
local-setup ⟨⟨ Cimp.locset @{thm ghost-honorary-root-empty-locs-def} ⟩⟩

definition (in mut-m) load-invL :: (field, mut, ref) gc-pred where
  load-invL ≡:
  [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)

3.12 The infamous termination argument.

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

definition (in mut-m) gc-W-empty-mut-inv :: (field, mut, ref) lsts-pred where
  gc-W-empty-mut-inv ≡
  ( empty sys-W and sys-ghost-handshake-in-sync m and not empty (WL (mutator m)) )
  imp ( EXS m’. not (sys-ghost-handshake-in-sync m’) and not empty (WL (mutator m’)) )
definition (in −) gc-W-empty-locs :: location set where
gc-W-empty-locs ≡
  idle-locs ∪ init-locs ∪ sweep-locs ∪ { "mark-read-fM", "mark-write-fA", "mark-end" }
  ∪ prefixed "mark-noop"
  ∪ prefixed "mark-loop-get-roots"
  ∪ prefixed "mark-loop-get-work"
local-setup (in −) ⟨⟨ Cimp.locset @{thm gc-W-empty-locs-def} ⟩⟩

definition black-heap-locs ≡ { "sweep-idle", "idle-noop-mfence", "idle-noop-init-type" }
local-setup ⟨⟨ Cimp.locset @{thm black-heap-locs-def} ⟩⟩

definition no-grey-refs-locs ≡ black-heap-locs ∪ sweep-locs ∪ { "mark-end" }
local-setup ⟨⟨ Cimp.locset @{thm no-grey-refs-locs-def} ⟩⟩

3.13 Sweep loop invariants

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 no-grey-refs-locs no-grey-refs
  and atS-gc gc-W-empty-locs (empty gc-W)

local-setup ⟨⟨ Cimp.locset @{thm gc-W-empty-invL} ⟩⟩
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

4.2 Initial conditions

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

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

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

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

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

definition gc-initial-state :: ('field, 'mut, 'ref) lsts-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) lsts-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) lsts-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 ≡
```
\[(\lambda s. gc-initial-state (s gc))
\quad \text{and} \quad (\lambda s. \forall m. mut-initial-state (s (mutator m)))
\quad \text{and} \quad (\lambda s. sys-initial-state (s sys))
\quad \text{and} \quad \text{valid-refs}
\]

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

**abbreviation** \text{gc-system} :: (\text{field}, \text{mut}, \text{ref}) gc-system \textbf{where}
\text{gc-system} \equiv (\text{gc-pgms}, \text{gc-system-init})

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

Our headline safety result follows directly.

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

**end**

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

**interpretation** gc-system-interpretation!: gc-system \textbf{undefined} .

### 4.3 A concrete system state

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

**theory** Concrete-heap

**imports**
~~/src/HOL/Library/Saturated
../Proofs

**begin**

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

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

**abbreviation** mut-common-init-state :: concrete-local-state \textbf{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** gc-system

**begin**

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

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 :: \((\text{'k} \times \text{'v})\) list \(\Rightarrow\) \text{'v} \(\Rightarrow\) \text{'k} \(\Rightarrow\) \text{'v}\) where

\[
\text{lookup} \ [\] \text{v0} \text{ k} = \text{v0} \\
| \text{lookup} \ (\text{kv} \# \text{kvs}) \text{ v0} \text{ k} = (\text{if} \ \text{fst} \ \text{kv} = \text{k} \ \text{then} \ \text{snd} \ \text{kv} \ \text{else} \ \text{lookup} \ \text{kvs} \ \text{v0} \ \text{k})
\]

**abbreviation** muts-init-states :: (mut \(\times\) 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 \(\equiv\) \(\lambda\) \text{p}. case \text{p} of

gc \(\Rightarrow\) gc-init-state \\
| sys \(\Rightarrow\) sys-init-state \\
| mutator \text{m} \(\Rightarrow\) lookup muts-init-states mut-common-init-state \text{m}

**lemma**

gc-system-init init-state

**end**

**References**


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