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

Peter Gammie, Tony Hosking and Kai Engelhardt

May 27, 2015

Abstract

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

Contents

1 Introduction 2

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

3 Invariants and Proofs 19
3.1 Constructors for sets of locations. ....................... 19
3.2 Hoare triples .............................................. 20
3.3 Functions and predicates ................................... 20
3.4 Garbage collector locations. ............................ 24
3.5 Coarse TSO invariants ..................................... 24
3.5.1 Locations where the TSO lock is held .................. 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.

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

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

\textbf{type-synonym} location = char list

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

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

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

datatype handshake-type
  = ht-NOOP
  | ht-GetRoots
  | ht-GetWork

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

datatype handshake-phase
  = hp-Idle
  | hp-IdleInit
  | hp-InitMark
  | hp-Mark
  | hp-IdleMarkSweep

definition handshake-step :: handshake-phase ⇒ handshake-phase where
  handshake-step ph ≡ case ph of
    hp-Idle       ⇒ hp-IdleInit
    hp-IdleInit   ⇒ hp-InitMark
    hp-InitMark   ⇒ hp-Mark
    hp-Mark       ⇒ hp-IdleMarkSweep
    hp-IdleMarkSweep ⇒ hp-Idle

An object consists of a garbage collection mark and a function that maps its fields to values.
A value is either a reference or NULL.
'field' is the abstract type of fields. 'ref' is the abstract type of object references. 'mut' is the
abstract type of the mutators’ names.
For simplicity we assume all objects define all fields and ignore all non-reference payload in
objects.

type-synonym gc-mark = bool

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

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

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

The system’s \textit{fA}, \textit{fM}, \textit{phase} and \textit{heap} variables are subject to the TSO memory model, as are all heap operations.

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

\begin{itemize}
  \item System-specific fields
  \item heap :: \text{'ref} ⇒ \((\text{'field}, \text{'ref})\) object option
  \item TSO memory state
  \item mem-write-buffers :: \text{'mut process-name} ⇒ \((\text{'field}, \text{'ref})\) mem-write-action list
  \item mem-lock :: \text{'mut process-name} option
  \item The state of the handshakes
  \item handshake-type :: handshake-type
  \item handshake-pending :: \text{'mut} ⇒ \text{bool}
  \item Ghost state
  \item ghost-handshake-in-sync :: \text{'mut} ⇒ \text{bool}
  \item ghost-handshake-phase :: handshake-phase

  \item Mutator-specific temporaries
  \item new-ref :: \text{'ref} option
  \item roots :: \text{'ref} set
  \item ghost-honorary-root :: \text{'ref} set

  \item Garbage collector-specific temporaries
  \item field-set :: \text{'field} set
  \item mut :: \text{'mut}
  \item muts :: \text{'mut} set

  \item Local variables used by multiple processes
  \item fA :: gc-mark
  \item fM :: gc-mark
  \item cas-mark :: gc-mark option
  \item field :: \text{'field}
  \item mark :: gc-mark option
  \item phase :: gc-phase
  \item tmp-ref :: \text{'ref}
  \item ref :: \text{'ref} option
  \item refs :: \text{'ref} set
  \item W :: \text{'ref} set
  \item Ghost state
  \item ghost-honorary-grey :: \text{'ref} set
\end{itemize}

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

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

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

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

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

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

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

We instantiate CIMP’s types as follows:

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

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

locale mut-m = fixes m :: \( \text{‘mut} \)
locale mut-m' = mut-m + fixes m' :: \( \text{‘mut}\) assumes \( \text{mn}[\text{iff}]: m \neq m'\)
locale gc
locale sys

2.1 Object marking

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

context
  fixes p :: \( \text{‘mut}\) 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 r fl 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 s ≡ {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 s ≡ {l} | s| W := W s ∪ {r s}, ghost-honorary-grey := { } | ]
notation add-to-W (\{-\}) add'-to-W

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

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

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

\textbf{definition} mark-object-fn :: location ⇒ (‘field, ‘mut, ‘ref) gc-com \textbf{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-cctest"\} 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 \textit{pending} flag for each mutator and then waits
for each to respond.

The system side of the interface collects the responses from the mutators into a single worklist,
which acts as a proxy for the garbage collector’s local worklist during \texttt{get-roots} and \texttt{get-work}
handshakes. In practise this involves a \texttt{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.

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

countext sys
begin

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

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

countext mut-m
begin

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

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

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

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

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

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

The garbage collector’s side of the interface.

context gc
begin
abbreviation set-hs-type :: location ⇒ handshake-type ⇒ (‘field, ‘mut, ‘ref) gc-com (\ell \rightarrow \text{set-hs-type}) where
\{ \ell \} \text{set-hs-type } ht \equiv [\ell \text{ Request } (\lambda s. (gc, ro-hs-gc-set-type ht)) (\lambda - s. \{ s \})]

abbreviation set-hs-pending :: location ⇒ ((‘field, ‘mut, ‘ref) local-state ⇒ ‘mut) ⇒ (‘field, ‘mut, ‘ref) gc-com (\ell \rightarrow \text{set-hs-pending}) where
\{ \ell \} \text{set-hs-pending } m \equiv [\ell \text{ Request } (\lambda s. (gc, ro-hs-gc-set-pending (m s))) (\lambda - s. \{ s \})]

definition handshake-init :: location ⇒ handshake-type ⇒ (‘field, ‘mut, ‘ref) gc-com (\ell \rightarrow \text{handshake\text{-init}})
where
\{ \ell \} \text{handshake-init } req \equiv
  [\ell \text{ @ } \text{"-init-type"}] \text{set-hs-type } req ::
  [\ell \text{ @ } \text{"-init-muts"}] \text{´muts} \equiv \text{UNIV} ::
  [\ell \text{ @ } \text{"-init-loop"}] \text{WHILE not empty muts DO}
    [\ell \text{ @ } \text{"-init-loop-choose-mut"}] \text{´mut} \equiv \text{´muts} ::
    [\ell \text{ @ } \text{"-init-loop-set-pending"}] \text{set-hs-pending mut} ::
    [\ell \text{ @ } \text{"-init-loop-done"}] \text{´muts} \equiv \text{´muts} - \{ \text{´mut} \}
  OD

definition handshake-done :: location ⇒ (‘field, ‘mut, ‘ref) gc-com (\ell \rightarrow \text{handshake\text{-done}})
where
\{ \ell \} \text{handshake-done} \equiv
  [\ell \text{ @ } \text{"-done-muts"}] \text{´muts} \equiv \text{UNIV} ::
  [\ell \text{ @ } \text{"-done-loop"}] \text{WHILE not empty muts DO}
    [\ell \text{ @ } \text{"-done-loop-choose-mut"}] \text{´mut} \equiv \text{´muts} ::
    [\ell \text{ @ } \text{"-done-loop-set-pending"}] \text{set-hs-pending mut} ::
    [\ell \text{ @ } \text{"-done-loop-done"}] \text{´muts} \equiv \text{´muts} - \{ \text{´mut} \}
  OD

abbreviation load-W :: location ⇒ (‘field, ‘mut, ‘ref) gc-com (\ell \rightarrow \text{load\text{-W}}) where
\{ \ell \} \text{load-W} \equiv [\ell \text{ @ } \text{"-load-W"}] \text{Request } (\lambda s. (gc, ro-hs-gc-load-W))
  (\lambda res \ s. \{ s\{ W := W \prime \} | W \prime. \ res = mv-Refs \ W \prime \})

abbreviation mfence :: location ⇒ (‘field, ‘mut, ‘ref) gc-com (\ell \rightarrow \text{MFENCE}) where
\{ \ell \} \text{MFENCE} \equiv [\ell \text{ Request } (\lambda s. (gc, ro-MFENCE)) (\lambda - s. \{ s \})]

definition handshake-noop :: location ⇒ (‘field, ‘mut, ‘ref) gc-com (\ell \rightarrow \text{handshake\text{-noop}})
where
\{ \ell \} \text{handshake-noop} \equiv
  [\ell \text{ @ } \text{"-m fence"}] \text{MFENCE} ::
  [\ell \} \text{handshake-init } ht-NOOP ::
2.3 The system process

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

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

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

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

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

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-\text{TSO}} \equiv \\
\{\"sys-read\"\} \text{ Response } (\lambda \text{req } s. \{ (s, \text{sys-read } p \text{ mr } s) \\
| \text{p mr. req } = (p, \text{ro-Read } \text{ mr}) \land \text{not-blocked } \text{ p } s \}) \\
\square \{\"sys-write\"\} \text{ Response } (\lambda \text{req } s. \{ (s| \text{mem-write-buffers } := (\text{mem-write-buffers } s)(p := \text{mem-write-buffers } s \text{ p } @ [w])) \}, \text{mv-Void}) \\
| \text{p w. req } = (p, \text{ro-Write } \text{ w}) \}) \\
\square \{\"sys-mfence\"\} \text{ Response } (\lambda \text{req } s. \{ (s| \text{mem-write-buffers } := (\text{mem-write-buffers } s)(p := \text{mem-write-buffers } s \text{ p } @ [w])) \}, \text{mv-Void}) \\
| \text{p w. req } = (p, \text{ro-MFENCE}) \land \text{mem-write-buffers } s \text{ p } = [] \}) \\
\square \{\"sys-lock\"\} \text{ Response } (\lambda \text{req } s. \{ (s| \text{mem-lock } := \text{Some p } \}, \text{mv-Void}) \\
| \text{p req } = (p, \text{ro-Lock}) \land \text{mem-lock } s = \text{None } \}) \\
\square \{\"sys-unlock\"\} \text{ Response } (\lambda \text{req } s. \{ (s| \text{mem-lock } := \text{None } \}, \text{mv-Void}) \\
| \text{p req } = (p, \text{ro-Unlock}) \land \text{mem-lock } s = \text{Some p } \land \\
\text{mem-write-buffers } s \text{ p } = [] \}) \\
\square \{\"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 \text{ p } @ [w])) \}) \\
| \text{p w ws. mem-write-buffers } s \text{ p } = w \# w s \land \text{not-blocked } \text{ s } \text{ p } \land \text{p } \neq \text{sys } \}) \\

We track which references are allocated using the domain of heap.

For now we assume that the system process magically allocates and deallocates references. To model this more closely we would need to take care of the underlying machine addresses.

We should be able to separate out those issues from GC correctness: the latter should imply that only alloc and free can interfere with each other.

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

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

\textbf{definition}

\texttt{alloc} :: (\texttt{field}, \texttt{mut}, \texttt{ref}) gc-com

\textbf{where}

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

References are freed by removing them from heap.

\textbf{definition}

\texttt{free} :: (\texttt{field}, \texttt{mut}, \texttt{ref}) gc-com

\textbf{where}

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

The top-level system process.

\textbf{definition}

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

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 } \text{mut-m}
\begin{align*}
\text{begin} \\
\text{Allocation is defined in Pizlo (201x, Figure 2.9). See §2.3 for how we abstract it.}
\end{align*}

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

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

The mutator can always discard any references it holds.

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

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

Dereferencing a reference can increase the set of mutator roots.

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

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

**abbreviation**

\[
\text{mut-deref} :: \text{location} \\
\Rightarrow (('field', 'mut', 'ref) \text{ local-state} \Rightarrow 'ref) \\
\Rightarrow (('field', 'mut', 'ref) \text{ local-state} \Rightarrow 'field) \\
\Rightarrow (('ref \text{ option} \Rightarrow 'ref \text{ option}) \Rightarrow (('field', 'mut', 'ref) \text{ local-state} \Rightarrow (('field', 'mut', 'ref) \text{ local-state} \Rightarrow (('field', 'mut', 'ref) \text{ gc-com} (\{l\} \text{ deref})) \\
\text{where} \\
\{l\} \text{ deref } r \text{ field upd} \equiv \{l\} \text{ Request } (\lambda s. (\text{mutator } m, \text{ReadRef } (r s) (f s))) \\
\text{ (} \lambda m v s. \{ \text{ upd } (\text{opt-r}') (s(\text{ghost-honorary-root} := \text{set-option opt-r'}))(\text{opt-r'})) \text{ opt-r'}. m v = \text{mv-Ref opt-r'} \})
\]

**abbreviation**

\[
\text{write-ref} :: \text{location} \\
\Rightarrow (('field', 'mut', 'ref) \text{ local-state} \Rightarrow 'ref) \\
\Rightarrow (('field', 'mut', 'ref) \text{ local-state} \Rightarrow 'field) \\
\Rightarrow (('field', 'mut', 'ref) \text{ local-state} \Rightarrow 'ref \text{ option}) \\
\Rightarrow (('field', 'mut', 'ref) \text{ gc-com} (\{l\} \text{ write-ref})) \\
\text{where} \\
\{l\} \text{ write-ref } r \text{ field } r' \equiv \{l\} \text{ Request } (\lambda s. (\text{mutator } m, \text{WriteRef } (r s) (f s) (r' s))) (\lambda- s. \{s(\text{ghost-honorary-root} := \{\}))))
\]

**definition**

\[
\text{store} :: (('field', 'mut', 'ref) \text{ gc-com}) \\
\text{where} \\
\text{store} \equiv \{ \text{ Choose vars for: ref → field := new-ref * } \} \\
\{"store-choose"\} \text{ LocalOp } (\lambda s. \{ s\{ \text{ tmp-ref} := r, \text{ field} := f, \text{ new-ref} := r' \}) \\
\text{ | } r f r', r \in \text{ roots } s \land r' \in \text{ Some } \{ \text{ roots } s \cup \{\text{None} \} \} \}) ;; \\
\{ \text{ Mark the reference we're about to overwrite. Does not update roots. * } \} \\
\{"deref-del"\} \text{ deref tmp-ref field ref-update } ;; \\
\{"store-del"\} \text{ mark-object } ;; \\
\{ \text{ Mark the reference we're about to insert. * } \} \\
\{"lop-store-ins"\} \text{ 'ref := 'new-ref } ;; \\
\{"store-ins"\} \text{ mark-object } ;; \\
\{"store-ins"\} \text{ write-ref tmp-ref field new-ref}
\]

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

**definition**

\[
\text{com} :: (('field', 'mut', 'ref) \text{ gc-com}) \\
\text{where} \\
\text{com} \equiv \}
\]
LOOP DO
    "mut local computation" SKIP
    alloc
discard
load
store
"mut mfence" MFENCE
handshake
OD

2.5 Garbage collector

We abstract the primitive actions of the garbage collector thread.

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

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

syntax
-gc-fassign :: location ⇒ idt ⇒ 'ref ⇒ 'field ⇒ ('field, 'mut, 'ref) gc-com (\l f \q \r f := \q \r \n - [0, 0, 70] 71)
-gc-massign :: location ⇒ idt ⇒ 'ref ⇒ ('field, 'mut, 'ref) gc-com (\l m \q \r m := \q \r \n flag [0, 0] 71)
translations
\l l f := \l f m := \l m flag => CONST gc-deref l r f (\update-name \q)
\l l m := \l m 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 :: (field, mut, ref) gc-com
where
  com ≡
  LOOP DO
  "idle-noop" handshakenoop ;; (* hp-Idle *)
  "idle-read-fM" read-fM ;;
  "idle-invert-fM" fM := (~ fM) ;;
  "idle-write-fM" write-fM ;;
  "idle-flip-noop" handshakenoop ;; (* hp-IdleInit *)
  "idle-phase-init" write-phase ph-Init ;;
  "init-noop" handshakenoop ;; (* hp-InitMark *)
  "init-phase-mark" write-phase ph-Mark ;;
  "mark-read-fM" read-fM ;;
  "mark-write-fA" write-fA fM ;;
  "mark-noop" handshakenoop ;; (* hp-Mark *)

...
```ml
\{"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"\} \texttt{tmp-ref} \in W ;
    \{"mark-loop-fields"\} \texttt{field-set} := UNIV ;
    \{"mark-loop-mark-object-loop"\} WHILE not empty \texttt{field-set} DO
      \{"mark-loop-mark-choose-field"\} \texttt{field} \in \texttt{field-set} ;
      \{"mark-loop-mark-deref"\} \texttt{ref} := \texttt{tmp-ref} \rightarrow \texttt{field} ;
      \{"mark-loop"\} \texttt{mark-object} ;
      \{"mark-loop-mark-field-done"\} \texttt{field-set} := (\texttt{field-set} - \{\texttt{field}\})
    OD ;
    \{"mark-loop-blacken"\} W := (W - \{\texttt{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"\} \texttt{refs} := UNIV ;
\{"sweep-loop"\} WHILE not empty \texttt{refs} DO
  \{"sweep-loop-choose-ref"\} \texttt{tmp-ref} \in \texttt{refs} ;
  \{"sweep-loop-read-mark"\} \texttt{mark} := \texttt{tmp-ref} \rightarrow \texttt{flag} ;
  \{"sweep-loop-check"\} IF not null \texttt{mark} and the \texttt{mark} neq fM THEN
    \{"sweep-loop-free"\} free \texttt{tmp-ref}
  FI ;
  \{"sweep-loop-ref-done"\} \texttt{refs} := (\texttt{refs} - \{\texttt{tmp-ref}\})
OD ;
\{"sweep-idle"\} write-phase ph-Idle
OD

end

primrec
  gc-pgms :: \texttt{mut process-name} ⇒ (\texttt{field, mut, ref}) gc-com
where
  gc-pgms (mutator m) = mut-m.com m
| gc-pgms gc = gc.com
| gc-pgms sys = sys.com
```
3 Invariants and Proofs

3.1 Constructors for sets of locations.

**abbreviation** prefixed :: location \(\Rightarrow\) location set where

\[ \text{prefixed } p \equiv \{ l \cdot \text{prefixeq } p \ l \} \]

**abbreviation** suffixed :: location \(\Rightarrow\) location set where

\[ \text{suffixed } p \equiv \{ l \cdot \text{suffixeq } p \ l \} \]

3.2 Hoare triples

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

**definition**

\[ \text{valid-proc} :: (\text{'field}, \text{'mut}, \text{'ref}) \text{gc-pred} \Rightarrow \text{'mut process-name} \Rightarrow (\text{'field}, \text{'mut}, \text{'ref}) \text{gc-pred} \Rightarrow \text{bool } ([\text{l}, \text{r}] - [\text{l}, \text{r}]) \]

where

\[ \{ P \} p \{ Q \} \equiv \forall (c, \text{aft}) \in \text{vcg-fragments} \text{(gc-pgms } p) \text{. (gc-pgms, } p, \text{aft) }\models\{ P \} c \{ Q \}\]

**abbreviation**

\[ \text{valid-proc-inv-syn} :: (\text{'field}, \text{'mut}, \text{'ref}) \text{gc-pred} \Rightarrow \text{'mut process-name} \Rightarrow \text{bool } ([\text{l}, \text{r}] - [100,0]) 100\]

where

\[ \{ P \} p \equiv \{ P \} p \{ P \}\]

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

3.3 Functions and predicates

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

**abbreviation** is-mw-Mark w \(\equiv\) \(\exists r \text{ fl. } w = \text{mw-Mark } r \text{ fl}\)

**abbreviation** is-mw-Mutate w \(\equiv\) \(\exists r f r'. w = \text{mw-Mutate } r f r'\)

**abbreviation** is-mw-fA w \(\equiv\) \(\exists \text{ fl. } w = \text{mw-fA } \text{ fl}\)

**abbreviation** is-mw-fM w \(\equiv\) \(\exists \text{ fl. } w = \text{mw-fM } \text{ fl}\)

**abbreviation** is-mw-Phase w \(\equiv\) \(\exists \text{ ph. } w = \text{mw-Phase } \text{ ph}\)

**abbreviation** (input) pred-in-W :: 'ref \(\Rightarrow\) 'mut process-name \(\Rightarrow\) ('field, 'mut, 'ref) lsts-pred (infix in\text{'-W} 50) where

\[ r \text{ in-W } p \equiv \lambda s. r \in W (s p) \]

**abbreviation** (input) pred-in-ghost-honorary-grey :: 'ref \(\Rightarrow\) 'mut process-name \(\Rightarrow\) ('field, 'mut, 'ref) lsts-pred (infix in\text{'-ghost'-honorary'-grey} 50) where
\[
\begin{align*}
\text{\textit{r} in-ghost-honorary-grey p} & \equiv \lambda s. \, \text{\textit{r} \in ghost-honorary-grey (s p)} \\
\text{context } gc \\
\text{begin} \\
\text{abbreviation} \\
\text{valid-gc-syn} :: (\text{'field}, \text{'mut}, \text{'ref}) gc-loc-comp \Rightarrow (\text{'field}, \text{'mut}, \text{'ref}) gc-pred \Rightarrow (\text{'field}, \text{'mut}, \text{'ref}) gc-com \Rightarrow (\text{'field}, \text{'mut}, \text{'ref}) gc-pred \Rightarrow bool \\
(\text{\textit{\$}} / \text{\textit{\$}}/ \text{\textit{\$}}) \\
\text{where} \\
afts \models \{P\} c \{Q\} \equiv gc-pgms, gc, afts \models \{P\} c \{Q\} \\
\text{abbreviation} valid-gc-inv-syn :: (\text{'field}, \text{'mut}, \text{'ref}) gc-loc-comp \Rightarrow (\text{'field}, \text{'mut}, \text{'ref}) gc-pred \Rightarrow (\text{'field}, \text{'mut}, \text{'ref}) gc-com \Rightarrow bool (- \models /- /-) \\
\text{where} \\
afts \models \{P\} c \equiv afts \models \{P\} c \{P\} \\
\text{end} \\
\text{abbreviation} gc-cas-mark s \equiv \text{cas-mark (s gc)} \\
\text{abbreviation} gc-fM s \equiv \text{fM (s gc)} \\
\text{abbreviation} gc-field s \equiv \text{field (s gc)} \\
\text{abbreviation} gc-field-set s \equiv \text{field-set (s gc)} \\
\text{abbreviation} gc-mark s \equiv \text{mark (s gc)} \\
\text{abbreviation} gc-mut s \equiv \text{mut (s gc)} \\
\text{abbreviation} gc-muts s \equiv \text{muts (s gc)} \\
\text{abbreviation} gc-phase s \equiv \text{phase (s gc)} \\
\text{abbreviation} gc-tmp-ref s \equiv \text{tmp-ref (s gc)} \\
\text{abbreviation} gc-ghost-honorary-grey s \equiv \text{ghost-honorary-grey (s gc)} \\
\text{abbreviation} gc-ref s \equiv \text{ref (s gc)} \\
\text{abbreviation} gc-refs s \equiv \text{refs (s gc)} \\
\text{abbreviation} gc-the-ref \equiv \text{the} \circ\ \text{gc-ref} \\
\text{abbreviation} gc-W s \equiv \text{W (s gc)} \\
\text{abbreviation} at-gc :: \text{location} \Rightarrow (\text{'field}, \text{'mut}, \text{'ref}) lsts-pred \Rightarrow (\text{'field}, \text{'mut}, \text{'ref}) gc-pred \\
\text{where} \\
at-gc l P \equiv at gc l \text{ imp LSTP P} \\
\text{abbreviation} atS-gc :: \text{location set} \Rightarrow (\text{'field}, \text{'mut}, \text{'ref}) lsts-pred \Rightarrow (\text{'field}, \text{'mut}, \text{'ref}) gc-pred \\
\text{where} \\
atS-gc ls P \equiv atS gc ls \text{ imp LSTP P} \\
\text{context} mut-m \\
\text{begin} \\
\text{abbreviation} \\
\text{valid-mut-syn} :: (\text{'field}, \text{'mut}, \text{'ref}) gc-loc-comp \Rightarrow (\text{'field}, \text{'mut}, \text{'ref}) gc-pred \Rightarrow (\text{'field}, \text{'mut}, \text{'ref}) gc-com \Rightarrow (\text{'field}, \text{'mut}, \text{'ref}) gc-pred \Rightarrow bool
\end{align*}
\]
\[(\cdot \models \{\cdot\} / \cdot \models \{\cdot\})\]

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

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

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

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

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

category sys

begin

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

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

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

end

abbreviation sys-heap :: (field, mut, ref) lsts \Rightarrow \{ field, ref \} object option where
sys-heap s \equiv heap (s sys)

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

abbreviation atS-sys :: location set \Rightarrow (field, mut, ref) object option where
atS-sys ls P \equiv atS sys ls \text{ imp } LSTP P

Projections on TSO buffers.

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

abbreviation (input) tso-pending p P s \equiv filter P (mem-write-buffers (s sys) p)
abbreviation (input) tso-pending-write p w s \equiv w \in set (mem-write-buffers (s sys) p)

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

abbreviation (input) tso-no-pending-marks \equiv 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 \Rightarrow bool \Rightarrow \{ field, ref \} object option where
obj-at P r \equiv \lambda s. case sys-heap s r of None \Rightarrow False | Some obj \Rightarrow P obj

abbreviation (input) valid-ref :: ref \Rightarrow (field, mut, ref) object option where
valid-ref r \equiv obj-at (True) r

definition valid-null-ref :: ref option \Rightarrow (field, mut, ref) object option where
valid-null-ref \( r \) \( \equiv \) case \( r \) of None \( \Rightarrow \) (True) | Some \( r^{'} \) \( \Rightarrow \) valid-ref \( r^{'} \)

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

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

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

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

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

3.4 Garbage collector locations.

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

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

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

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

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

3.5 Coarse TSO invariants

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

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

abbreviation mut-writes :: \( \text{('field, 'ref) mem-write-action} \Rightarrow \text{bool} \text{ where}
mut-writes w \equiv \text{case } w \text{ of mw-Mutate - - - } \Rightarrow \text{True} | \text{mw-Mark - - } \Rightarrow \text{True} | - \Rightarrow \text{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 ≡ ∪ l∈{ "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 ≡
    ∪ l∈{ "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-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) lsts-pred where
handshake-phase-inv ≡ ALLS m.
(sys-ghost-handshake-in-sync m ⊗ sys-handshake-type
⊗ sys-ghost-handshake-phase ⊗ mut-m.mut-ghost-handshake-phase m) in ⟨hp-step-rel⟩
and (sys-handshake-pending m imp not sys-ghost-handshake-in-sync m)

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

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

definition hp-IdleInit-locs ≡
(prefixed "init-noop" − { "init-noop-mfence", "init-noop-init-type" })
∪ { "init-phase-mark", "mark-read-fM", "mark-write-fA", "mark-noop-mfence", "mark-noop-init-type" }
local-setup ⟨⟨ Cimp.locset @{thm hp-IdleInit-locs-def} ⟩⟩

definition hp-InitMark-locs ≡
(prefixed "mark-noop" − { "mark-noop-mfence", "mark-noop-init-type" })
∪ { "mark-loop-get-roots-init-type" }
local-setup ⟨⟨ Cimp.locset @{thm hp-InitMark-locs-def} ⟩⟩

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

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

definition hs-noop-locs ≡
   ∪ l ∈ hs-noop-prefixes. prefixed l − (suffixed "-noop-mfence" ∪ suffixed "-noop-init-type")
local-setup ⟨⟨ Cimp.locset @{thm hs-noop-locs-def} ⟩⟩

definition hs-get-roots-locs ≡
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 \in hs-prefixes. prefixed (l @ "-init-loop")
local-setup ⟨⟨ Cimp.locset @{thm hs-init-loop-locs-def} ⟩⟩

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

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

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

definition hs-mut-in-muts-locs ≡ \bigcup l \in 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 \in 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 \in 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 [inv]: handshake-invL ≡
  atS-gc hs-noop-locs (sys-handshake-type eq \{ht-NOOP\})
  and atS-gc hs-get-roots-locs (sys-handshake-type eq \{ht-GetRoots\})
  and atS-gc hs-get-work-locs (sys-handshake-type eq \{ht-GetWork\})

27
and atS-gc hs-mut-in-muts-locs \((gc\text{-mut in gc-muts})\)

and atS-gc hs-init-loop-locs \((\text{ALLS m. not } \langle m \rangle \text{ in gc-muts imp sys-handshake-pending m})\)

or \(\text{sys-ghost-handshake-in-sync m}\)

and atS-gc hs-init-loop-not-done-locs \((\text{ALLS m. } \langle m \rangle \text{ in gc-muts imp not sys-handshake-pending m})\)

and not \(\text{sys-ghost-handshake-in-sync m}\)

and atS-gc hs-init-loop-done-locs \((\text{sys-handshake-pending } \triangleright gc\text{-mut}

\text{or sys-ghost-handshake-in-sync } \triangleright gc\text{-mut})

\text{and (ALLS m. } \langle m \rangle \text{ in gc-muts and } \langle m \rangle \text{ neq gc-mut}

\text{imp not sys-handshake-pending m}

\text{and not sys-ghost-handshake-in-sync m)})\)

and atS-gc hs-done-locs \((\text{ALLS m. sys-handshake-pending m or sys-ghost-handshake-in-sync m})\)

and atS-gc hs-done-loop-locs \((\text{ALLS m. not } \langle m \rangle \text{ in gc-muts imp not sys-handshake-pending m})\)

and atS-gc hs-none-pending-locs \((\text{ALLS m. not sys-handshake-pending m})\)

and atS-gc hs-in-sync-locs \((\text{ALLS m. sys-ghost-handshake-in-sync m})\)

and atS-gc hs-out-of-sync-locs \((\text{ALLS m. not sys-handshake-pending m}

\text{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 mut-no-pending-mutates-locs \equiv

\((\text{prefixed } "\text{hs-noop}" \text{ - } \{ "\text{hs-noop}"", "\text{hs-noop-mfence}" \})

\cup \text{ (prefixed } "\text{hs-get-roots}" \text{ - } \{ "\text{hs-get-roots}"", "\text{hs-get-roots-mfence}" \})

\cup \text{ (prefixed } "\text{hs-get-work}" \text{ - } \{ "\text{hs-get-work}"", "\text{hs-get-work-mfence}" \})\)

local-setup \(\ll Cimp.locset \oplus \{ \text{thm mut-no-pending-mutates-locs-def} \}\)

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

\text{inv]: handshake-invL } \equiv

atS-mut (\text{prefixed } "\text{hs-noop-1}" \text{ ) (sys-handshake-type eq } \langle \text{ht-NOOP} \rangle \text{ and sys-handshake-pending m})

and atS-mut (\text{prefixed } "\text{hs-get-roots-1}" \text{ ) (sys-handshake-type eq } \langle \text{ht-GetRoots} \rangle \text{ and sys-handshake-pending m})

and atS-mut (\text{prefixed } "\text{hs-get-work-1}" \text{ ) (sys-handshake-type eq } \langle \text{ht-GetWork} \rangle \text{ and sys-handshake-pending m})

and atS-mut mut-no-pending-mutates-locs \text{ (list-null (iso-pending-mutate (mutator m)))}\)

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

**Definition** `handshake-phase-rel` :: `handshake-phase ⇒ bool ⇒ gc-phase ⇒ bool` where
\[
\text{handshake-phase-rel \ hp \ in-sync \ ph} \equiv \\
\begin{cases}
\text{hp-Idle} & \Rightarrow ph = \text{ph-Idle} \\
\text{hp-IdleInit} & \Rightarrow ph = \text{ph-Idle} \lor (\text{in-sync} \land ph = \text{ph-Init}) \\
\text{hp-InitMark} & \Rightarrow ph = \text{ph-Init} \lor (\text{in-sync} \land ph = \text{ph-Mark}) \\
\text{hp-Mark} & \Rightarrow ph = \text{ph-Mark} \\
\text{hp-IdleMarkSweep} & \Rightarrow ph = \text{ph-Mark} \lor (\text{in-sync} \land ph \in \{ \text{ph-Idle}, \text{ph-Sweep} \})
\end{cases}
\]

**Definition** `phase-rel` :: `(bool × `handshake-phase` × `gc-phase` × `gc-phase` × (`field`, 'ref') mem-write-action list)` set where
\[
\text{phase-rel} \equiv \\
\{ (\text{in-sync}, \text{hp}, \text{ph}, [], []) \mid \text{in-sync hp ph. handshake-phase-rel hp in-sync ph} \} \\
\cup \{ \text{True} \} \times \{ (\text{hp-IdleInit}, \text{ph-Init}, \text{ph-Idle}, [\text{mw-Phase ph-Init}]), (\text{hp-InitMark}, \text{ph-Mark}, \text{ph-Init}, [\text{mw-Phase ph-Mark}]), (\text{hp-IdleMarkSweep}, \text{ph-Sweep}, \text{ph-Mark}, [\text{mw-Phase ph-Sweep}]), (\text{hp-IdleMarkSweep}, \text{ph-Idle}, \text{ph-Mark}, [\text{mw-Phase ph-Sweep, mw-Phase ph-Idle}]), (\text{hp-IdleMarkSweep}, \text{ph-Idle}, \text{ph-Sweep}, [\text{mw-Phase ph-Idle}]) \}
\]

**Definition** `phase-rel-inv` :: (`field`, 'mut', 'ref') lsts-pred where
\[
\text{phase-rel-inv} \equiv (\text{ALLS m. sys-ghost-handshake-in-sync m}) \otimes \text{sys-ghost-handshake-phase} \otimes \text{gc-phase} \otimes \text{sys-phase} \otimes \text{tso-pending-phase gc} \text{ in } (\text{phase-rel})
\]

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

**Definition** `no-pending-phase-locs` :: `location set` where
\[
\text{no-pending-phase-locs} \equiv \\
\{ \text{idle-locs} - \{ "\text{idle-noop-mfence}" \} \} \\
\cup \{ \text{init-locs} - \{ "\text{init-noop-mfence}" \} \} \\
\cup \{ \text{mark-locs} - \{ "\text{mark-read-fM", "mark-write-fA", "mark-noop-mfence"} \} \}
\]

**Local Setup** \(\text{Cimp.locset} @@\{\text{thm no-pending-phase-locs-def} \} \)

**Definition** `(in gc) phase-invL` :: (`field`, 'mut', 'ref') `gc-pred` where
\[
\text{[inv]: phase-invL} \equiv \\
\text{atS-gc idle-locs} \quad (\text{gc-phase eq } \langle \text{ph-Idle} \rangle) \\
\text{and atS-gc init-locs} \quad (\text{gc-phase eq } \langle \text{ph-Init} \rangle) \\
\text{and atS-gc mark-locs} \quad (\text{gc-phase eq } \langle \text{ph-Mark} \rangle) \\
\text{and atS-gc sweep-locs} \quad (\text{gc-phase eq } \langle \text{ph-Sweep} \rangle) \\
\text{and atS-gc no-pending-phase-locs} \quad (\text{list-null (tso-pending-phase gc)})
\]

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

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

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

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

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

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

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

\[
\begin{align*}
fA\text{-rel-inv} &= (\text{ALLS m. sys-ghost-handshake-phase} \times \text{sys-fA} \otimes \text{gc-fM} \otimes \text{tso-pending-fA gc} \text{ in (fA-rel)}
\end{align*}
\]

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

\[
\begin{align*}
fM\text{-eq-locs} &= -(\{ "\text{idle-write-fM}, \text{"idle-flip-noop-mfence}" \})
\end{align*}
\]

definition \( \text{local-setup (in } - \rangle \rangle \langle \text{Cimp.locset @\{thm fM-eq-locs-def} \rangle \) \)

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

\[
\begin{align*}
fM\text{-tso-empty-locs} &= -(\{ "\text{idle-flip-noop-mfence}" \})
\end{align*}
\]

definition \( \text{local-setup (in } - \rangle \langle \text{Cimp.locset @\{thm fM-tso-empty-locs-def} \rangle \) \)

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

\[
\begin{align*}
fA\text{-tso-empty-locs} &= -(\{ "\text{mark-noop-mfence}" \})
\end{align*}
\]

definition \( \text{local-setup (in } - \rangle \langle \text{Cimp.locset @\{thm fA-tso-empty-locs-def} \rangle \) \)

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

\[
\begin{align*}
fA\text{-eq-locs} &= \{ "\text{idle-read-fM}, \text{"idle-invert-fM}" \} \\
&\cup \text{prefix "\text{idle-noop}"} \\
&\cup \{ \text{mark-locs} - \{ "\text{mark-read-fM}, "\text{mark-write-fA}, "\text{mark-noop-mfence}" \} \\
&\cup \text{sweep-locs}
\end{align*}
\]

definition \( \text{local-setup (in } - \rangle \langle \text{Cimp.locset @\{thm fA-eq-locs-def} \rangle \) \)

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

\[
\begin{align*}
fA\text{-neq-locs} &= \{ "\text{idle-phase-init}, "\text{idle-write-fM}, "\text{mark-read-fM}, "\text{mark-write-fA}" \} \\
&\cup \text{prefix "\text{idle-flip-noop}"}
\end{align*}
\]

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

\textbf{local-setup} \{ \text{Cimp.locset} \ @\{ \text{thm fA-neq-locs-def} \} \}

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

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

\textbf{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:

\textbf{White} potential garbage, not yet reached

\textbf{Grey} reached, presumed live, a source of possible new references (work)

\textbf{Black} reached, presumed live, not a source of new references

In this particular setting we use the following interpretation:

\textbf{White}: not marked

\textbf{Grey}: on a worklist

\textbf{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 \textit{ghost-honorary-grey} for some process, i.e. grey.

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

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

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

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

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

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

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

\[ \text{grey r} \equiv \text{EXS p. (r) in WL p} \]
definition \texttt{black} :: 'ref \Rightarrow ('field, 'mut, 'ref) lsts-pred \textbf{where}
black r \equiv \text{marked } r \text{ 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 \texttt{write-refs} :: ('field, 'ref) \textbf{mem-write-action} \Rightarrow 'ref set \textbf{where}
write-refs w \equiv \text{case } w \text{ of } mw-Mutate r f r' \Rightarrow \{ r \} \cup \text{Option.set-option } r'

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

definition \textbf{(in} \texttt{mut-m}) \texttt{reachable} :: 'ref \Rightarrow ('field, 'mut, 'ref) lsts-pred \textbf{where}
reachable y \equiv \text{EXS } x. (x) \text{ in } (\text{mut-roots union mut-ghost-honorary-root union tso-write-refs})
and x reaches y

definition \texttt{grey-reachable} :: 'ref \Rightarrow ('field, 'mut, 'ref) lsts-pred \textbf{where}
grey-reachable y \equiv \text{EXS } g. \text{grey } g \text{ and } g \text{ reaches } y

definition \texttt{valid-refs-inv} :: ('field, 'mut, 'ref) lsts-pred \textbf{where}
valid-refs-inv \equiv \text{ALLS } x. ((\text{EXS } m. \text{mut-m}.reachable m x) \text{ or } \text{grey-reachable } x) \text{ imp } \text{valid-ref } x

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

definition \texttt{valid-W-inv} :: ('field, 'mut, 'ref) lsts-pred \textbf{where}
valid-W-inv \equiv \text{ALLS } p q r.
(r \text{ in-W } p \text{ or } (\text{sys-mem-lock neq } \langle \text{Some } p \rangle \text{ and } r \text{ in-ghost-honorary-grey } p) \text{ imp } \text{marked } r)
and ((p \neq q) \text{ imp } \text{not } ((r) \text{ in WL } p \text{ and } (r) \text{ in WL } q))
and (\text{not } (r \text{ in-ghost-honorary-grey } p \text{ and } r \text{ in-W } q))
and (\text{empty } \text{sys-ghost-honorary-grey})
and (\text{ALLS } fl. \text{tso-pending-write } p (\text{mw-Mark } r fl))
\text{ imp } ( (\langle fl \rangle) \text{ eq sys-fM })
and r \text{ in-ghost-honorary-grey } p
and \text{tso-locked-by } p
and \text{white } r
and \text{tso-pending-mark } p \text{ eq } (\langle \text{mw-Mark } r fl \rangle ) )
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 (\(\lambda\).: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 (\(\lambda\)s. {[], [mw-Mark (p-the-ref s) (p-fM s)]}))

abbreviation (input)
  p-mark-inv ≡ not null p-mark
  and ((\(\lambda\)s. 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 ≡ (\(\lambda\)s. obj-at (\obj. Some (obj-mark \obj) = p-cas-mark s) (p-the-ref s) s)
abbreviation (input) p-valid-fM ≡ p-fM eq sys-fM

abbreviation (input)
  p-ghost-eq-ref ≡ p-ghost-honorary-grey eq pred-singleton (the ◦ p-ref)

abbreviation (input)
  p-ghost-inv ≡ If p-cas-mark eq p-mark Then p-ghost-eq-ref Else empty p-ghost-honorary-grey

definition mark-object-invL :: ('field, 'mut, 'ref) gc-pred where
  mark-object-invL ≡
  at-p "'-mo-null" (True)
  and at-p "'-mo-mark" (p-valid-ref)
  and at-p "'-mo-fM" (p-valid-ref and p-en-cond (p-mark-inv))
  and at-p "'-mo-mtest" (p-valid-ref and p-en-cond (p-mark-inv and p-valid-fM))
  and at-p "'-mo-phase" (p-valid-ref and p-mark neq Some ◦ p-fM and p-en-cond (p-mark-inv and p-valid-fM))
  and at-p "'-mo-ptest" (p-valid-ref and p-mark neq Some ◦ p-fM and p-en-cond (p-mark-inv and p-valid-fM))
  and at-p "'-mo-co-lock" (p-valid-ref and p-mark-inv and p-valid-fM and p-mark neq Some ◦ p-fM and p-tso-no-pending-mark)
  and at-p "'-mo-co-cmark" (p-valid-ref and p-mark-inv and p-valid-fM and p-mark neq Some ◦ p-fM and p-tso-no-pending-mark)
  and at-p "'-mo-co-ctest" (p-valid-ref and p-mark-inv and p-valid-fM and p-mark neq Some ◦ p-fM and p-cas-mark-inv and p-valid-fM and white ◦ p-the-ref and p-tso-no-pending-mark)
  and at-p "'-mo-co-mark" (p-cas-mark eq p-mark and p-valid-ref and p-valid-fM and p-mark neq Some ◦ p-fM and p-tso-no-pending-mark)
  and at-p "'-mo-co-unlock" (p-ggh-inv and p-valid-ref and p-valid-fM and p-valid-W-inv)
  and at-p "'-mo-co-won" (p-ggh-inv and p-valid-ref and p-valid-fM and marked ◦ p-the-ref and p-tso-no-pending-mutate)
  end

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

interpretation gc-mark!: mark-object gc "mark-loop" (True) by default (simp add: eq-imp-def)
lemmas gc-mark-mark-object-invL-def2 [inv] = gc-mark.mark-object-invL-def [simplified]

interpretation mut-get-roots!: mark-object mutator m "hs-get-roots-loop" (True) for m by default (simp add: eq-imp-def)
lemmas mut-get-roots-mark-object-invL-def2 [inv] = mut-get-roots.mark-object-invL-def [simplified]

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

interpretation mut-store-del!: mark-object mutator m "store-del" mut.m.mut-ghost-handshake-phase m neq (hp-Idle) for m by default (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 default (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
  ghost-honorary-grey-empty-locs ≡
  − (⋃ pref ∈ {"mark-loop", "hs-get-roots-loop", "store-del", "store-ins"}.}
\[ \bigcup l \in \{ "mo-co-unlock", "mo-co-won", "mo-co-W" \}. \{ \text{pref @ "." @ l} \} \]

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

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

\[ \text{inv} : \text{mark-object-invL} \equiv \]

\[ \text{atS-mut mut-hs-get-roots-loop-locs} \quad \text{(mut-refs subseteq mut-roots and (ALLS r. (r) in mut-roots diff mut-refs imp marked r))} \]

\[ \text{and atS-mut mut-hs-get-roots-loop-mo-locs} \quad \text{(not null mut-ref and mut-the-ref in mut-roots)} \]

\[ \text{and at-mut "hs-get-roots-loop-done"} \quad \text{(marked > mut-the-ref)} \]

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

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

\[ \text{and at-S-mut mut-mo-valid-ref-locs} \quad \text{(not null mut-new-ref imp mut-the-new-ref in mut-roots)} \]

\[ \text{and (mut-tmp-ref in mut-roots)} \]

\[ \text{and at-mut "store-del-mo-null"} \quad \text{(not null mut-ref imp mut-the-ref in mut-ghost-honorary-root)} \]

\[ \text{and atS-mut (prefixed "store-del" \( \{ "store-del-mo-null" \})} \quad \text{(mut-the-ref in mut-ghost-honorary-root)} \]

\[ \text{and atS-mut (prefixed "store-ins")} \quad \text{(mut-ref eq mut-new-ref)} \]

\[ \text{and atS-mut mut-mo-valid-ref-locs} \quad \text{(not null mut-new-ref imp marked \( \diamond \) mut-the-new-ref in mut-ghost-honorary-root)} \]

\[ \text{and atS-mut (suffixed "~mo-ptest")} \quad \text{(mut-phase neq \( \langle \text{ph-Idle} \rangle \))} \]

\[ \text{and atS-mut mut-hs-get-roots-loop-mo-locs} \quad \text{(mut-phase in \( \langle \{ \text{hp-InitMark, hp-Mark} \} \})} \]

\[ \text{or (mut-ghost-handshake-phase eq \( \langle \text{ph-IdleMarkSweep} \rangle \))} \]

\[ \text{and atS-mut ghost-honorary-grey-empty-locs} \quad \text{(empty mut-ghost-honorary-grey)} \]

\[ \text{(* insertion barrier *)} \]

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

\[ \text{or (mut-ghost-handshake-phase eq \( \langle \text{hp-IdleMarkSweep} \rangle \))} \]

\[ \text{and sys-phase neq \( \langle \text{ph-Idle} \rangle \}))} \]

\[ \text{and not null mut-new-ref} \]

\[ \text{imp marked > mut-the-new-ref) } \]

\[ \text{(* deletion barrier *)} \]

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

\[ \text{(mut-ghost-handshake-phase eq \( \langle \text{hp-Mark} \rangle \))} \]

\[ \text{or (mut-ghost-handshake-phase eq \( \langle \text{hp-IdleMarkSweep} \rangle \))} \]

\[ \text{and sys-phase neq \( \langle \text{ph-Idle} \rangle \))} \]

\[ \text{and (\( \lambda s. \forall \text{opt-r'. tso-pending-write} \) (mutator m) (mw-Mutate (mut-tmp-ref s) (mut-field s) opt-r')} s \]

\[ \text{imp (\( \lambda s. \text{obj-at-field-on-heap} \) (\( \lambda r. \text{mut-ref s = Some r \lor marked r s} \)) (mut-tmp-ref s) (mut-field s) s))} \]

36
and at-mut "lop-store-""lop-store-ins"
and sys-phase neq ⟨ph-Idle⟩)
and not null mut-ref
imp marked ▷ mut-the-ref )
and atS-mut (prefixed "store-"
and sys-phase neq ⟨ph-Idle⟩)
and not null mut-ref
imp marked ▷ mut-the-ref )
and sys-phase neq ⟨ph-Idle⟩)

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

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

definition obj-fields-marked-locs :: location set where
obj-fields-marked-locs ≡
{ "mark-loop-mark-object-loop", "mark-loop-mark-choose-field", "mark-loop-mark-deref", "mark-loop-mark-field-done", "mark-loop-blacken" } 
∪ prefixed "mark-loop-mo"
local-setup ⟨⟨ Cimp.locset @{thm obj-fields-marked-locs-def} ⟩⟩

definition (in gc) obj-fields-marked-invL :: ('field, 'mut, 'ref) gc-pred where
{inv}: obj-fields-marked-invL ≡
atS-gc obj-fields-marked-locs (obj-fields-marked-inv and gc-tmp-ref in gc-W)
and atS-gc (prefixed "mark-loop-mo") (ALLS y. not null gc-ref and (λs. ((gc-the-ref s) reaches y) s) imp valid-ref y)
and at-gc "mark-loop-fields" (gc-tmp-ref in gc-W)
and at-gc "mark-loop-mark-field-done" (not null gc-ref imp marked ▷ gc-the-ref)
and at-gc "mark-loop-blacken" (empty gc-field-set)
and atS-gc ghost-honorary-grey-empty-locs (empty gc-ghost-honorary-grey)

3.9 The strong-tricolour invariant

As the GC algorithm uses both insertion and deletion barriers, it preserves the strong tricolour-invariant:
abbreviation \text{points-to-white} :: \text{ref} \Rightarrow \text{ref} \Rightarrow (\text{field}, \text{mut}, \text{ref}) \text{lsts-pred} \text{ (infix points'-'to'-white 51)} \text{ where}
\begin{align*}
x \text{ points-to-white } y &\equiv x \text{ points-to } y \text{ and white } y
\end{align*}

definition \text{strong-tricolour-inv} :: (\text{field}, \text{mut}, \text{ref}) \text{lsts-pred} \text{ where}
\begin{align*}
\text{strong-tricolour-inv} &\equiv \text{ALLS } b \text{ w. black } b \text{ imp not } b \text{ points-to-white } w
\end{align*}

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 \text{has-white-path-to} :: \text{ref} \Rightarrow \text{ref} \Rightarrow (\text{field}, \text{mut}, \text{ref}) \text{lsts-pred} \text{ (infix has'-'white'-'path'-to 51)} \text{ where}
\begin{align*}
x \text{ has-white-path-to } y &\equiv \lambda s. (\lambda x y. (x \text{ points-to-white } y) s) \ast x y
\end{align*}

definition \text{grey-protects-white} :: \text{ref} \Rightarrow \text{ref} \Rightarrow (\text{field}, \text{mut}, \text{ref}) \text{lsts-pred} \text{ (infix grey'-'protects'-'white 51)} \text{ where}
\begin{align*}
g \text{ grey-protects-white } w &\equiv \text{grey } g \text{ and } g \text{ has-white-path-to } w
\end{align*}

definition \text{weak-tricolour-inv} :: (\text{field}, \text{mut}, \text{ref}) \text{lsts-pred} \text{ where}
\begin{align*}
\text{weak-tricolour-inv} &\equiv \text{ALLS } b \text{ w. black } b \text{ and } b \text{ points-to-white } w \text{ imp } (\text{EXS } g. \text{ g grey-protects-white } w)
\end{align*}

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

definition \text{in-snapshot} :: \text{ref} \Rightarrow (\text{field}, \text{mut}, \text{ref}) \text{lsts-pred} \text{ where}
\begin{align*}
\text{in-snapshot } r &\equiv \text{black } r \text{ or } (\text{EXS } g. \text{ g grey-protects-white } r)
\end{align*}

definition \text{(in mut-m) reachable-snapshot-inv} :: (\text{field}, \text{mut}, \text{ref}) \text{lsts-pred} \text{ where}
\begin{align*}
\text{reachable-snapshot-inv} &\equiv \text{ALLS } r. \text{ reachable } r \text{ imp in-snapshot } r
\end{align*}

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 \text{marked-insertion} :: (\text{field}, \text{ref}) \text{mem-write-action} \Rightarrow (\text{field}, \text{mut}, \text{ref}) \text{lsts-pred} \text{ where}
\begin{align*}
\text{marked-insertion } w &\equiv \lambda s. \text{case } w \text{ of } \text{mw-Mutate } r f (\text{Some } r') \Rightarrow \text{marked } r' s \mid - \Rightarrow \text{True}
\end{align*}

definition \text{(in mut-m) marked-insertions} :: (\text{field}, \text{mut}, \text{ref}) \text{lsts-pred} \text{ where}
\begin{align*}
\text{marked-insertions} &\equiv \text{ALLS } w. \text{ tso-pending-write } (\text{mutator } m) w \text{ imp marked-insertion } w
\end{align*}

abbreviation \text{marked-deletion} :: (\text{field}, \text{ref}) \text{mem-write-action} \Rightarrow (\text{field}, \text{mut}, \text{ref}) \text{lsts-pred} \text{ 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 )
abbreviation \( \text{sys-phase-inv} :: (\text{field}, \text{mut}, \text{ref}) \text{lsts-pred} \) where
\[
\text{sys-phase-inv} \ s \equiv \text{sys-phase-inv-aux} \ (\text{sys-ghost-handshake-phase} \ s) \ s
\]

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 \( \text{ghost-honorary-root-empty-locs} :: \text{location set} \) where
\[
\text{ghost-honorary-root-empty-locs} \equiv
\begin{array}{l}
- \ (\text{prefixed} \ "\text{store-del}\" \cup \{"\text{lop-store-ins}\"\} \cup \text{prefixed} \ "\text{store-ins}\")
\end{array}
\]

local-setup \( \langle\langle \text{Cimp.locset} \ @\{\text{thm ghost-honorary-root-empty-locs-def}\} \rangle\rangle \)

definition \( \text{load-invL} :: (\text{field}, \text{mut}, \text{ref}) \text{gc-pred} \) where
\[
\text{[inv]}: \text{load-invL} \equiv
\begin{array}{l}
\begin{array}{l}
\text{at-mut "load" (mut-refs in mut-roots)}
\end{array}
\end{array}
\begin{array}{l}
\begin{array}{l}
\text{and at-mut "hs-noop-done" (list-null (tso-pending-mutate (mutator m)))}
\end{array}
\end{array}
\begin{array}{l}
\begin{array}{l}
\text{and atS-mut \text{ghost-honorary-root-empty-locs} (empty mut-ghost-honorary-root)}
\end{array}
\end{array}
\]

3.12 The infamous termination argument.

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

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

definition \( \text{gc-W-empty-mut-inv} :: (\text{field}, \text{mut}, \text{ref}) \text{lsts-pred} \) where
\[
\text{gc-W-empty-mut-inv} \equiv
\begin{array}{l}
\text{(empty sys-W and sys-ghost-handshake-in-sync m and not empty (WL (mutator m)))}
\end{array}
\begin{array}{l}
\begin{array}{l}
\text{imp (EXS m'. not (sys-ghost-handshake-in-sync m') and not empty (WL (mutator m')))}
\end{array}
\end{array}
\]

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

local-setup \( \langle\langle \text{Cimp.locset} \ @\{\text{thm gc-W-empty-locs-def}\} \rangle\rangle \)
**definition** black-heap-locs ≡ \{ "sweep-idle", "idle-noop-mfence", "idle-noop-init-type" \}

**local-setup** \&\langle\text{Cimp.locset @\{thm black-heap-locs-def\}}\rangle

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

**local-setup** \&\langle\text{Cimp.locset @\{thm no-grey-refs-locs-def\}}\rangle

**definition** (in gc) gc-W-empty-invL :: (f\text{ield}, m\text{ut}, r\text{ef}) gc-pred where

\[ \text{inv}: \quad \text{gc-W-empty-invL} \equiv \]
\[
\begin{align*}
\text{ats-gc (hs-get-roots-locs } \cup \text{ hs-get-work-locs) (ALLS m. mut-m.gc-W-empty-mut-inv m)} \\
\text{and at-gc "mark-loop-get-roots-load-W" (empty sys-W imp no-grey-refs)} \\
\text{and at-gc "mark-loop-get-work-load-W" (empty sys-W imp no-grey-refs)} \\
\text{and atS-gc no-grey-refs-locs no-grey-refs} \\
\text{and atS-gc gc-W-empty-locs (empty gc-W)}
\end{align*}
\]

3.13 Sweep loop invariants

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

**local-setup** \&\langle\text{Cimp.locset @\{thm sweep-loop-locs-def\}}\rangle

**definition** (in gc) sweep-loop-invL :: (f\text{ield}, m\text{ut}, r\text{ef}) gc-pred where

\[ \text{inv}: \quad \text{sweep-loop-invL} \equiv \]
\[
\begin{align*}
\text{at-gc "sweep-loop-check" (not null gc-mark imp (λs. obj-at (λobj. \text{Some (obj-mark obj) = gc-mark s (gc-tmp-ref s s)}) \\
\text{and (null gc-mark imp (marked } \triangleright \text{ gc-tmp-ref or not valid-ref}} \\
\text{ gc-tmp-ref)}) } ) \\
\text{and at-gc "sweep-loop-free" (not null gc-mark and the } \circ \text{ gc-mark neq gc-fM and (λs. obj-at (λobj. \text{Some (obj-mark obj) = gc-mark s (gc-tmp-ref s s)}) \\
\text{and at-gc "sweep-loop-ref-done" (marked } \triangleright \text{ gc-tmp-ref or not valid-ref } \triangleright \text{ gc-tmp-ref) } ) \\
\text{and atS-gc sweep-loop-locs (ALLS r. not } \langle r \rangle \text{ in gc-refs imp (marked r or not valid-ref r)}) \\
\text{and atS-gc black-heap-locs (ALLS r. marked r or not valid-ref r)} \\
\text{and atS-gc (prefixed "sweep-loop" – \{"sweep-loop-choose-ref"\}) (gc-tmp-ref in gc-refs)}
\end{align*}
\]

4 Top-level safety

4.1 Invariants

**definition** (in gc) invsL :: (f\text{ield}, m\text{ut}, r\text{ef}) gc-pred where

\[ \text{invsL} \equiv \]
\[
\begin{align*}
\text{fM-fA-invL} \\
\text{and gc-mark.mark-object-invL} \\
\text{and gc-W-empty-invL} \\
\text{and handshake-invL} \\
\text{and obj-fields-marked-invL} \\
\text{and phase-invL} \\
\text{and sweep-loop-invL}
\end{align*}
\]
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.

**locale** gc-system =
  fixes initial-mark :: gc-mark

**begin**
definition gc-initial-state :: (field, mut, ref) lst-pred where
gc-initial-state s ≡
  fM s = initial-mark
∧ phase s = ph-Idle
∧ ghost-honorary-grey s = {}
∧ W s = {}
definition mut-initial-state :: (field, mut, ref) lst-pred where
mut-initial-state s ≡
  ghost-handshake-phase s = hp-IdleMarkSweep
∧ ghost-honorary-grey s = {}
∧ ghost-honorary-root s = {}
∧ W s = {}
definition sys-initial-state :: (field, mut, ref) lst-pred where
sys-initial-state s ≡
  (∀ m. ¬handshake-pending s m ∧ ghost-handshake-in-sync s m)
∧ ghost-handshake-phase s = hp-IdleMarkSweep ∧ handshake-type s = ht-GetRoots
∧ obj-mark ' ran (heap s) ⊆ {initial-mark}
∧ fA s = initial-mark
∧ fM s = initial-mark
∧ phase s = ph-Idle
∧ ghost-honorary-grey s = {}
∧ W s = {}
∧ (∀ p. mem-write-buffers s p = [])
∧ mem-lock s = None
abbreviation
root-reachable y ≡ EXS m x. ⟨x⟩ in mut-m.mut-roots m and x reaches y
definition valid-refs :: (field, mut, ref) lsts-pred where
valid-refs ≡ ALLS y. root-reachable y imp valid-ref y
definition gc-system-init :: (field, mut, ref) lsts-pred where
gc-system-init ≡
  (λs. gc-initial-state (s gc))
  and (λs. ∀ m. mut-initial-state (s (mutator m)))
  and (λs. sys-initial-state (s sys))
  and valid.refs

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

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

theorem inv: s ∈ reachable-states gc-system ⇒ 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 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 where

context gc-system begin

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

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

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

end

class gc-system
begin

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

, ghost-handshake-in-sync := (True)
, ghost-handshake-phase := hp-IdleMarkSweep )

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


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

abbreviation muts-init-states :: (mut × concrete-local-state) list where
muts-init-states ≡ [ (0, mut-init-state0), (1, mut-init-state1), (2, mut-init-state2) ]
\textbf{abbreviation} \textit{init-state :: clsts where}\\
init-state \equiv \lambda p. \text{case } p \text{ of}\\
gc \Rightarrow gc\text{-init-state}\\
| \text{sys} \Rightarrow sys\text{-init-state}\\
| \text{mutator } m \Rightarrow \text{lookup } \textit{muts-init-states } \text{mut-common-init-state } m\\
\\
\textbf{lemma}\\
\textit{gc-system-init } \textit{init-state}\\
\\
\textbf{end}\\

\textbf{References}\\


Peter Gammie. Concurrent IMP. \textit{Archive of Formal Proofs}, April 2015. ISSN 2150-914x. \url{http://afp.sf.net/entries/ConcurrentIMP.shtml}, Formal proof development.\\


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


46