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

Peter Gammie, Tony Hosking and Kai Engelhardt

October 11, 2017

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

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

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

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

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

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

2 The Schism garbage collector

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

We additionally need to model TSO memory, the handshakes and compare-and-swap (`CAS`). We closely model things where interference is possible and abstract everything else.

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

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

type-synonym *location* = *char list*

datatype *'mut process-name* = *mutator 'mut* | *gc* | *sys*

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

datatype *gc-phase*

= *ph-Idle*
| *ph-Init*
| *ph-Mark*
| *ph-Sweep*

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

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*

| *mw-fA gc-mark*
| *mw-fM gc-mark*
| *mw-Phase gc-phase*

The following record is the type of all processes's local states. For the mutators and the garbage collector, consider these to be local variables or registers.

The system's *fA*, *fM*, *phase* and *heap* variables are subject to the TSO memory model, as are all heap operations.

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

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

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

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

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

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

= *mr-Ref* *'ref* *'field*
| *mr-Mark* *'ref*
| *mr-Phase*
| *mr-fM*
| *mr-fA*

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

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

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

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

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

We instantiate CIMP's types as follows:

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

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

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

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

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

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

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

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

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

2.1 Object marking

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

context
fixes *p* :: *'mut process-name*

begin

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

abbreviation *unlock* :: *location* ⇒ (*'field*, *'mut*, *'ref*) *gc-com* **where**
unlock l ≡ ⦇*l*⦈ *Request* (λ*s*. (*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* (λ*s*. (*p*, *ReadMark* (*r s*))) (λ*mv s*. { *upd* ⟨*m*⟩ *s* |*m*. *mv* = *mv-Mark m* })
notation *read-mark* (⦇-⦈ *read'-mark*)

abbreviation *read-fM* :: *location* ⇒ (*'field*, *'mut*, *'ref*) *gc-com* **where**
read-fM l ≡ ⦇*l*⦈ *Request* (λ*s*. (*p*, *ro-Read mr-fM*)) (λ*mv 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* (λ*s*. (*p*, *ReadPhase*)) (λ*mv s*. { *s*(|*phase* := *ph*|) |*ph*. *mv* = *mv-Phase ph* })
notation *read-phase* (⦇-⦈ *read'-phase*)

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

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

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

TSO means we cannot avoid having the mark write pending in a store buffer; in other words, we cannot have objects atomically transition from white to grey. The following scheme black-

ens a white object, and then reverts it to grey. The *ghost-honorary-grey* variable is used to track objects undergoing this transition.

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

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

end

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

2.2 Handshakes

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

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

The system and mutators track handshake phases using ghost state.

abbreviation *hp-step* :: *handshake-type* ⇒ *handshake-phase* ⇒ *handshake-phase* **where**
hp-step ht ≡
  case ht of

```
    ht-NOOP ⇒ handshake-step
  | ht-GetRoots ⇒ handshake-step
  | ht-GetWork ⇒ id

context sys
begin

definition handshake :: ('field, 'mut, 'ref) gc-com where
  handshake ≡
    {|''sys-hs-gc-set-type''|} Response
      (λreq s. { (s(| 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(| handshake-pending := (handshake-pending s)(m := True) |), mv-Void)
               |m. req = (gc, ro-hs-gc-set-pending m) })
  ⊔ {|''sys-hs-gc-done''|} Response
      (λreq s. { (s, mv-Bool (¬handshake-pending s m))
               |m. req = (gc, ro-hs-gc-read-pending m) })
  ⊔ {|''sys-hs-gc-load-W''|} Response
      (λreq s. { (s(| W := {} |), mv-Refs (W s))
               |-::unit. req = (gc, ro-hs-gc-load-W) })
  ⊔ {|''sys-hs-mut''|} Response
      (λreq s. { (s, mv-Void)
               |m. req = (mutator m, ro-hs-mut-read-type (handshake-type s))
                  ∧ handshake-pending s m })
  ⊔ {|''sys-hs-mut-done''|} Response
      (λreq s. { (s(| handshake-pending := (handshake-pending s)(m := False),
                      W := W s ∪ W',
                      ghost-handshake-in-sync := (ghost-handshake-in-sync s)(m := True) |),
                  mv-Void)
               |m W'. req = (mutator m, ro-hs-mut-done W') })

end
```

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

```
context mut-m
begin

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

abbreviation mfence :: location ⇒ ('field, 'mut, 'ref) gc-com ({|-|} MFENCE) where
  {|l|} MFENCE ≡ {|l|} Request (λs. (mutator m, ro-MFENCE)) (λ- s. {s})
```

abbreviation *hs-read-type* :: *location* ⇒ *handshake-type* ⇒ (*'field*, *'mut*, *'ref*) *gc-com* (⦇-⦈ *hs'-read'-type*) **where**
⦇*l*⦈ *hs-read-type ht* ≡ ⦇*l*⦈ *Request* (λ*s*. (*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* (λ*s*. (*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*) ⇒ (*'field*, *'mut*, *'ref*) *gc-com* (⦇-⦈ *hs'-get'-roots'-done*) **where**
⦇*l*⦈ *hs-get-roots-done wl* ≡ ⦇*l*⦈ *Request* (λ*s*. (*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*) ⇒ (*'field*, *'mut*, *'ref*) *gc-com* (⦇-⦈ *hs'-get'-work'-done*) **where**
⦇*l*⦈ *hs-get-work-done wl* ≡ ⦇*l*⦈ *Request* (λ*s*. (*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*
⊔
⦇*''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*
⊔
⦇*''hs-get-work''*⦈ *hs-read-type ht-GetWork* ;;
⦇*''hs-get-work-mfence''*⦈ *MFENCE* ;;
⦇*''hs-get-work-done''*⦈ *hs-get-work-done W*

end

The garbage collector's side of the interface.

context *gc*
begin

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

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

definition
handshake-init :: *location* ⇒ *handshake-type* ⇒ (*'field*, *'mut*, *'ref*) *gc-com* (⦇-⦈ *handshake'-init*)
where
⦇*l*⦈ *handshake-init req* ≡
⦇*l* @ *''-init-type''*⦈ *set-hs-type req* ;;
⦇*l* @ *''-init-muts''*⦈ ´*muts* := *UNIV* ;;
⦇*l* @ *''-init-loop''*⦈ *WHILE not empty muts DO*
⦇*l* @ *''-init-loop-choose-mut''*⦈ ´*mut* :∈ ´*muts* ;;
⦇*l* @ *''-init-loop-set-pending''*⦈ *set-hs-pending mut* ;;
⦇*l* @ *''-init-loop-done''*⦈ ´*muts* := (´*muts* − {´*mut*})
OD

definition
handshake-done :: *location* ⇒ (*'field*, *'mut*, *'ref*) *gc-com* (⦇-⦈ *handshake'-done*)
where
⦇*l*⦈ *handshake-done* ≡
⦇*l* @ *''-done-muts''*⦈ ´*muts* := *UNIV* ;;
⦇*l* @ *''-done-loop''*⦈ *WHILE not empty muts DO*
⦇*l* @ *''-done-loop-choose-mut''*⦈ ´*mut* :∈ ´*muts* ;;
⦇*l* @ *''-done-loop-rendezvous''*⦈ *Request*
(λ*s*. (*gc*, *ro-hs-gc-read-pending* (*mut s*)))
(λ*mv s*. { *s*(| *muts* := *muts s* − { *mut s* |*done*. *mv* = *mv-Bool done* ∧ *done* } |)})
OD

abbreviation *load-W* :: *location* ⇒ (*'field*, *'mut*, *'ref*) *gc-com* (⦇-⦈ *load'-W*) **where**
⦇*l*⦈ *load-W* ≡ ⦇*l* @ *''-load-W''*⦈ *Request* (λ*s*. (*gc*, *ro-hs-gc-load-W*))
(λ*resp s*. {*s*(| *W* := *W'* |) |*W'*. *resp* = *mv-Refs W'*})

abbreviation *mfence* :: *location* ⇒ (*'field*, *'mut*, *'ref*) *gc-com* (⦇-⦈ *MFENCE*) **where**
⦇*l*⦈ *MFENCE* ≡ ⦇*l*⦈ *Request* (λ*s*. (*gc*, *ro-MFENCE*)) (λ- *s*. {*s*})

definition
handshake-noop :: *location* ⇒ (*'field*, *'mut*, *'ref*) *gc-com* (⦇-⦈ *handshake'-noop*)
where
⦇*l*⦈ *handshake-noop* ≡
⦇*l* @ *''-mfence''*⦈ *MFENCE* ;;
⦇*l*⦈ *handshake-init ht-NOOP* ;;

$\{\!|l|\!\}$ *handshake-done*

definition
handshake-get-roots :: *location* ⇒ (*'field*, *'mut*, *'ref*) *gc-com* ($\{\!|$-$|\!\}$ *handshake'-get'-roots*)
where
$\{\!|l|\!\}$ *handshake-get-roots* ≡
$\{\!|l|\!\}$ *handshake-init ht-GetRoots* ;;
$\{\!|l|\!\}$ *handshake-done* ;;
$\{\!|l|\!\}$ *load-W*

definition
handshake-get-work :: *location* ⇒ (*'field*, *'mut*, *'ref*) *gc-com* ($\{\!|$-$|\!\}$ *handshake'-get'-work*)
where
$\{\!|l|\!\}$ *handshake-get-work* ≡
$\{\!|l|\!\}$ *handshake-init ht-GetWork* ;;
$\{\!|l|\!\}$ *handshake-done* ;;
$\{\!|l|\!\}$ *load-W*

end

2.3 The system process

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

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

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

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

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

```
  | 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* (λ*w*. *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 ≡ λ*s*.
case ract of
mr-Ref r f ⇒ *mv-Ref* (*heap s r* ⤜ (λ*obj*. *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

```
mem-TSO ≡
    ⦃"sys-read"⦄ Response (λreq s. { (s, sys-read p mr s)
                                  |p mr. req = (p, ro-Read mr) ∧ not-blocked s p })
  ⊔ ⦃"sys-write"⦄ Response (λreq s. { (s⦇ mem-write-buffers := (mem-write-buffers s)(p
:= mem-write-buffers s p @ [w]) ⦈, mv-Void)
                                  |p w. req = (p, ro-Write w) })
  ⊔ ⦃"sys-mfence"⦄ Response (λreq s. { (s, mv-Void)
                                 |p. req = (p, ro-MFENCE) ∧ mem-write-buffers s p = [] })
  ⊔ ⦃"sys-lock"⦄ Response (λreq s. { (s⦇ mem-lock := Some p ⦈, mv-Void)
                                  |p. req = (p, ro-Lock) ∧ mem-lock s = None })
  ⊔ ⦃"sys-unlock"⦄ Response (λreq s. { (s⦇ mem-lock := None ⦈, mv-Void)
                                      |p. req = (p, ro-Unlock) ∧ mem-lock s = Some p ∧
mem-write-buffers s p = [] })
  ⊔ ⦃"sys-dequeue-write-buffer"⦄ LocalOp (λs. { (do-write-action w s)⦇ mem-write-buffers
:= (mem-write-buffers s)(p := ws) ⦈
                                      | p w ws. mem-write-buffers s p = w # ws ∧
not-blocked s p ∧ p ≠ sys })
```

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

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

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

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

```
definition
  alloc :: ('field, 'mut, 'ref) gc-com
where
  alloc ≡ ⦃"sys-alloc"⦄ Response (λreq s.
    { ( s⦇ heap := (heap s)(r := Some (⦇ obj-mark = fA s, obj-fields = ⟨None⟩ ⦈)) ⦈, mv-Ref
(Some r) )
    |r. r ∉ dom (heap s) ∧ snd req = ro-Alloc })
```

References are freed by removing them from *heap*.

```
definition
  free :: ('field, 'mut, 'ref) gc-com
where
  free ≡ ⦃"sys-free"⦄ Response (λreq s.
    { (s⦇heap := (heap s)(r := None)⦈, mv-Void) |r. snd req = ro-Free r })
```

The top-level system process.

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

where
com ≡
LOOP DO
mem-TSO
⊔ *alloc*
⊔ *free*
⊔ *handshake*
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.

context *mut-m*
begin

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

abbreviation (**in** −) *mut-alloc* :: *'mut* ⇒ (*'field*, *'mut*, *'ref*) *gc-com* **where**
mut-alloc m ≡
⦃*''alloc''*⦄ *Request* (λ*s*. (*mutator m*, *ro-Alloc*))
(λ*mv s*. { *s*(| *roots* := *roots s* ∪ {*r*} |) |*r*. *mv* = *mv-Ref* (*Some r*) })

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

The mutator can always discard any references it holds.

abbreviation *discard* :: (*'field*, *'mut*, *'ref*) *gc-com* **where**
discard ≡
⦃*''discard-refs''*⦄ *LocalOp* (λ*s*. { *s*(| *roots* := *roots'* |) |*roots'*. *roots'* ⊆ *roots s* })

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

Dereferencing a reference can increase the set of mutator roots.

abbreviation *load* :: (*'field*, *'mut*, *'ref*) *gc-com* **where**
load ≡
⦃*''load-choose''*⦄ *LocalOp* (λ*s*. { *s*(| *tmp-ref* := *r*, *field* := *f* |) |*r f*. *r* ∈ *roots s* }) ;;
⦃*''load''*⦄ *Request* (λ*s*. (*mutator m*, *ReadRef* (*tmp-ref s*) (*field s*)))
(λ*mv s*. { *s*(| *roots* := *roots s* ∪ *set-option r* |)
|*r*. *mv* = *mv-Ref r* })

Storing a reference involves marking both the old and new references, i.e., both *insertion* and *deletion* barriers are installed. The deletion barrier preserves the *weak tricolour invariant*, and the insertion barrier preserves the *strong tricolour invariant*; see §3.9 for further discussion.

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

abbreviation
mut-deref :: *location*
⇒ (('*field*, '*mut*, '*ref*) *local-state* ⇒ '*ref*)
⇒ (('*field*, '*mut*, '*ref*) *local-state* ⇒ '*field*)
⇒ (('*ref option* ⇒ '*ref option*) ⇒ ('*field*, '*mut*, '*ref*) *local-state* ⇒ ('*field*, '*mut*, '*ref*) *local-state*) ⇒ ('*field*, '*mut*, '*ref*) *gc-com* (⦇-⦈ *deref*)
where
⦇*l*⦈ *deref r f upd* ≡ ⦇*l*⦈ *Request* (λ*s*. (*mutator m*, *ReadRef* (*r s*) (*f s*)))
(λ*mv s*. { *upd* ⟨*opt-r'*⟩ (*s*(|*ghost-honorary-root* := *set-option opt-r'*|)) |*opt-r'*. *mv* = *mv-Ref opt-r'* })

abbreviation
write-ref :: *location*
⇒ (('*field*, '*mut*, '*ref*) *local-state* ⇒ '*ref*)
⇒ (('*field*, '*mut*, '*ref*) *local-state* ⇒ '*field*)
⇒ (('*field*, '*mut*, '*ref*) *local-state* ⇒ '*ref option*)
⇒ ('*field*, '*mut*, '*ref*) *gc-com* (⦇-⦈ *write'-ref*)
where
⦇*l*⦈ *write-ref r f r'* ≡ ⦇*l*⦈ *Request* (λ*s*. (*mutator m*, *WriteRef* (*r s*) (*f s*) (*r' s*))) (λ- *s*. {*s*(|*ghost-honorary-root* := {}|)})

definition
store :: ('*field*, '*mut*, '*ref*) *gc-com*
where
store ≡
(∗ *Choose vars for*: *ref*→*field* := *new-ref* ∗)
⦇''*store-choose*''⦈ *LocalOp* (λ*s*. { *s*(| *tmp-ref* := *r*, *field* := *f*, *new-ref* := *r'* |)
|*r f r'*. *r* ∈ *roots s* ∧ *r'* ∈ *Some* ' *roots s* ∪ {*None*} }) ;;
(∗ *Mark the reference we're about to overwrite. Does not update roots.* ∗)
⦇''*deref-del*''⦈ *deref tmp-ref field ref-update* ;;
⦇''*store-del*''⦈ *mark-object* ;;
(∗ *Mark the reference we're about to insert.* ∗)
⦇''*lop-store-ins*''⦈ ´*ref* := ´*new-ref* ;;
⦇''*store-ins*''⦈ *mark-object* ;;
⦇''*store-ins*''⦈ *write-ref tmp-ref field new-ref*

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

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

LOOP DO
{|''mut local computation''|} SKIP
⊔ *alloc*
⊔ *discard*
⊔ *load*
⊔ *store*
⊔ *{|''mut mfence''|} MFENCE*
⊔ *handshake*
OD

end

2.5 Garbage collector

We abstract the primitive actions of the garbage collector thread.

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

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

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

context *gc*
begin

abbreviation *write-fA* :: *location* ⇒ ((*'field*, *'mut*, *'ref*) *local-state* ⇒ *gc-mark*) ⇒ (*'field*, *'mut*, *'ref*) *gc-com* (⦃-⦄ *write'-fA*) **where**
⦃*l*⦄ *write-fA f* ≡ ⦃*l*⦄ *Request* (λ*s*. (*gc*, *WritefA* (*f s*))) (λ- *s*. {*s*})

abbreviation *read-fM* :: *location* ⇒ (*'field*, *'mut*, *'ref*) *gc-com* (⦃-⦄ *read'-fM*) **where**
⦃*l*⦄ *read-fM* ≡ ⦃*l*⦄ *Request* (λ*s*. (*gc*, *ReadfM*)) (λ*mv s*. { *s*(|*fM* := *m*|) |*m*. *mv* = *mv-Mark* (*Some m*) })

abbreviation *write-fM* :: *location* ⇒ (*'field*, *'mut*, *'ref*) *gc-com* (⦃-⦄ *write'-fM*) **where**
⦃*l*⦄ *write-fM* ≡ ⦃*l*⦄ *Request* (λ*s*. (*gc*, *WritefM* (*fM s*))) (λ- *s*. {*s*})

abbreviation *write-phase* :: *location* ⇒ *gc-phase* ⇒ (*'field*, *'mut*, *'ref*) *gc-com* (⦃-⦄ *write'-phase*) **where**
⦃*l*⦄ *write-phase ph* ≡ ⦃*l*⦄ *Request* (λ*s*. (*gc*, *WritePhase ph*)) (λ- *s*. {*s*(| *phase* := *ph* |)})

abbreviation *mark-object* :: *location* ⇒ (*'field*, *'mut*, *'ref*) *gc-com* (⦃-⦄ *mark'-object*) **where**
⦃*l*⦄ *mark-object* ≡ *mark-object-fn gc l*

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

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

definition (**in** *gc*)
com :: (*'field*, *'mut*, *'ref*) *gc-com*
where
com ≡
LOOP DO
⦃*''idle-noop''*⦄ *handshake-noop* ;; (∗ *hp-Idle* ∗)

⦃*''idle-read-fM''*⦄ *read-fM* ;;
⦃*''idle-invert-fM''*⦄ *´fM* := (¬ *´fM*) ;;
⦃*''idle-write-fM''*⦄ *write-fM* ;;

⦃*''idle-flip-noop''*⦄ *handshake-noop* ;; (∗ *hp-IdleInit* ∗)

⦃*''idle-phase-init''*⦄ *write-phase ph-Init* ;;

⦃*''init-noop''*⦄ *handshake-noop* ;; (∗ *hp-InitMark* ∗)

⦃*''init-phase-mark''*⦄ *write-phase ph-Mark* ;;
⦃*''mark-read-fM''*⦄ *read-fM* ;;
⦃*''mark-write-fA''*⦄ *write-fA fM* ;;

⦃*''mark-noop''*⦄ *handshake-noop* ;; (∗ *hp-Mark* ∗)

```
    {''mark-loop-get-roots''} handshake-get-roots ;; (* hp-IdleMarkSweep *)

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

    (* sweep *)

    {''mark-end''} write-phase ph-Sweep ;;
    {''sweep-read-fM''} read-fM ;;
    {''sweep-refs''} ´refs := UNIV ;;
    {''sweep-loop''} WHILE not empty refs DO
      {''sweep-loop-choose-ref''} ´tmp-ref :∈ ´refs ;;
      {''sweep-loop-read-mark''} ´mark := ´tmp-ref→flag ;;
      {''sweep-loop-check''} IF not null mark and the ∘ mark neq fM THEN
        {''sweep-loop-free''} free tmp-ref
      FI ;;
      {''sweep-loop-ref-done''} ´refs := (´refs − {´tmp-ref})
    OD ;;
    {''sweep-idle''} write-phase ph-Idle
  OD

end

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

3 Invariants and Proofs

3.1 Constructors for sets of locations.

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

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

3.2 Hoare triples

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

definition
valid-proc :: ('*field*, '*mut*, '*ref*) *gc-pred* ⇒ '*mut process-name* ⇒ ('*field*, '*mut*, '*ref*) *gc-pred* ⇒ *bool* (⦃-⦄ - ⦃-⦄)
where
⦃*P*⦄ *p* ⦃*Q*⦄ ≡ ∀(*c*, *afts*) ∈ *vcg-fragments* (*gc-pgms p*). (*gc-pgms*, *p*, *afts* ⊨ ⦃*P*⦄ *c* ⦃*Q*⦄)

abbreviation
valid-proc-inv-syn :: ('*field*, '*mut*, '*ref*) *gc-pred* ⇒ '*mut process-name* ⇒ *bool* (⦃-⦄ - [*100*,*0*] *100*)
where
⦃*P*⦄ *p* ≡ ⦃*P*⦄ *p* ⦃*P*⦄⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

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

⟨*ML*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*ML*⟩

3.3 Functions and predicates

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

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

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

abbreviation (*input*) *pred-in-ghost-honorary-grey* :: '*ref* ⇒ '*mut process-name* ⇒ ('*field*, '*mut*, '*ref*) *lsts-pred* (**infix** *in'-ghost'-honorary'-grey 50*) **where**

r in-ghost-honorary-grey p ≡ λ*s*. *r* ∈ *ghost-honorary-grey* (*s p*)

context *gc*
begin

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

abbreviation *valid-gc-inv-syn* :: ('*field*, '*mut*, '*ref*) *gc-loc-comp* ⇒ ('*field*, '*mut*, '*ref*) *gc-pred* ⇒ ('*field*, '*mut*, '*ref*) *gc-com* ⇒ *bool* (- ⊨ ⦃-⦄/ -) **where**
afts ⊨ ⦃*P*⦄ *c* ≡ *afts* ⊨ ⦃*P*⦄ *c* ⦃*P*⦄

end

abbreviation *gc-cas-mark s* ≡ *cas-mark* (*s gc*)
abbreviation *gc-fM s* ≡ *fM* (*s gc*)
abbreviation *gc-field s* ≡ *field* (*s gc*)
abbreviation *gc-field-set s* ≡ *field-set* (*s gc*)
abbreviation *gc-mark s* ≡ *mark* (*s gc*)
abbreviation *gc-mut s* ≡ *mut* (*s gc*)
abbreviation *gc-muts s* ≡ *muts* (*s gc*)
abbreviation *gc-phase s* ≡ *phase* (*s gc*)
abbreviation *gc-tmp-ref s* ≡ *tmp-ref* (*s gc*)
abbreviation *gc-ghost-honorary-grey s* ≡ *ghost-honorary-grey* (*s gc*)
abbreviation *gc-ref s* ≡ *ref* (*s gc*)
abbreviation *gc-refs s* ≡ *refs* (*s gc*)
abbreviation *gc-the-ref* ≡ *the* ∘ *gc-ref*
abbreviation *gc-W s* ≡ *W* (*s gc*)

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

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

context *mut-m*
begin

abbreviation
valid-mut-syn :: ('*field*, '*mut*, '*ref*) *gc-loc-comp* ⇒ ('*field*, '*mut*, '*ref*) *gc-pred* ⇒ ('*field*, '*mut*, '*ref*) *gc-com* ⇒ ('*field*, '*mut*, '*ref*) *gc-pred* ⇒ *bool*

(- ⊨ ⦃-⦄/ -/ ⦃-⦄)
where
afts ⊨ ⦃*P*⦄ *c* ⦃*Q*⦄ ≡ *gc-pgms*, *mutator m*, *afts* ⊨ ⦃*P*⦄ *c* ⦃*Q*⦄

abbreviation *valid-mut-inv-syn* :: ('*field*, '*mut*, '*ref*) *gc-loc-comp* ⇒ ('*field*, '*mut*, '*ref*) *gc-pred* ⇒ ('*field*, '*mut*, '*ref*) *gc-com* ⇒ *bool* (- ⊨ ⦃-⦄/ -) **where**
afts ⊨ ⦃*P*⦄ *c* ≡ *afts* ⊨ ⦃*P*⦄ *c* ⦃*P*⦄

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

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

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

end

context *sys*
begin

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

abbreviation *valid-sys-inv-syn* :: ('*field*, '*mut*, '*ref*) *gc-loc-comp* ⇒ ('*field*, '*mut*, '*ref*) *gc-pred* ⇒ ('*field*, '*mut*, '*ref*) *gc-com* ⇒ *bool* (- ⊨ ⦃-⦄/ -) **where**

afts $\models \{\!|P|\!\}\ c \equiv$ *afts* $\models \{\!|P|\!\}\ c\ \{\!|P|\!\}$

end

abbreviation *sys-heap* :: (*'field*, *'mut*, *'ref*) *lsts* $\Rightarrow$ *'ref* $\Rightarrow$ (*'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-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*) *lsts-pred* $\Rightarrow$ (*'field*, *'mut*, *'ref*) *gc-pred* **where**
atS-sys ls P $\equiv$ *atS sys ls 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*) $\Rightarrow$ *'ref* $\Rightarrow$ (*'field*, *'mut*, *'ref*) *lsts-pred* **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*) *lsts-pred* **where**
valid-ref r $\equiv$ *obj-at* $\langle$*True*$\rangle$ *r*

definition *valid-null-ref* :: *'ref option* $\Rightarrow$ (*'field*, *'mut*, *'ref*) *lsts-pred* **where**

valid-null-ref r ≡ *case r of None* ⇒ ⟨*True*⟩ | *Some r′* ⇒ *valid-ref r′*

abbreviation *pred-points-to* :: *'ref* ⇒ *'ref* ⇒ (*'field*, *'mut*, *'ref*) *lsts-pred* (**infix** *points'-to* *51*) **where**
x points-to y ≡ λ*s*. *obj-at* (λ*obj*. *y* ∈ *ran* (*obj-fields obj*)) *x s*

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

abbreviation *pred-reaches* :: *'ref* ⇒ *'ref* ⇒ (*'field*, *'mut*, *'ref*) *lsts-pred* (**infix** *reaches* *51*) **where**
x reaches y ≡ λ*s*. (λ*x y*. (*x points-to y*) *s*)** *x y*

The predicate *obj-at-field-on-heap* asserts that if there is an object at $r.f$ on the heap, then it satisfies P.

definition *obj-at-field-on-heap* :: (*'ref* ⇒ *bool*) ⇒ *'ref* ⇒ *'field* ⇒ (*'field*, *'mut*, *'ref*) *lsts-pred* **where**
obj-at-field-on-heap P r f ≡ λ*s*.
case Option.map-option obj-fields (*sys-heap s r*) *of*
None ⇒ *False*
| *Some fs* ⇒ (*case fs f of None* ⇒ *True*
| *Some r′* ⇒ *P r′*)

3.4 Garbage collector locations.

definition *idle-locs* :: *location set* **where**
idle-locs ≡ *prefixed ″idle″*

definition *init-locs* :: *location set* **where**
init-locs ≡ *prefixed ″init″*

definition *mark-locs* :: *location set* **where**
mark-locs ≡ *prefixed ″mark″*

definition *mark-loop-locs* :: *location set* **where**
mark-loop-locs ≡ *prefixed ″mark-loop″*

definition *sweep-locs* :: *location set* **where**
sweep-locs ≡ *prefixed ″sweep″*⟨*ML*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

3.5 Coarse TSO invariants

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

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

abbreviation *mut-writes* :: (*'field*, *'ref*) *mem-write-action* ⇒ *bool* **where**
mut-writes w ≡ *case w of mw-Mutate - - - ⇒ True* | *mw-Mark - - ⇒ True* | *- ⇒ False*

definition *tso-writes-inv* :: (*'field*, *'mut*, *'ref*) *lsts-pred* **where**
tso-writes-inv ≡
(*ALLS w*. *tso-pending-write gc* *w imp* ⟨*gc-writes w*⟩)
and (*ALLS m w*. *tso-pending-write* (*mutator m*) *w imp* ⟨*mut-writes w*⟩)⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨

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*
⟨*ML*⟩

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*)⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

A mutator holds the TSO lock only during the `CAS`s in *mark-object*.

definition *mut-tso-lock-locs* ≡
⋃ *l*∈{ *''mo-co-cmark''*, *''mo-co-ctest''*, *''mo-co-mark''*, *''mo-co-unlock''* }. *suffixed l*
⟨*ML*⟩

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*))⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

3.6 Handshake phases

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

definition *hp-step-rel* :: (*bool* × *handshake-type* × *handshake-phase* × *handshake-phase*) *set* **where**
hp-step-rel ≡
{ *True* } × ({ (*ht-NOOP*, *hp*, *hp*) |*hp*. *hp* ∈ {*hp-Idle*, *hp-IdleInit*, *hp-InitMark*, *hp-Mark*} }
∪ { (*ht-GetRoots*, *hp-IdleMarkSweep*, *hp-IdleMarkSweep*)
, (*ht-GetWork*, *hp-IdleMarkSweep*, *hp-IdleMarkSweep*) })
∪ { *False* } × { (*ht-NOOP*, *hp-Idle*, *hp-IdleMarkSweep*)
, (*ht-NOOP*, *hp-IdleInit*, *hp-Idle*)
, (*ht-NOOP*, *hp-InitMark*, *hp-IdleInit*)
, (*ht-NOOP*, *hp-Mark*, *hp-InitMark*)

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

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

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

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

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

definition *hp-InitMark-locs* ≡
(*prefixed ''init-noop''* − { *''init-noop-mfence''*, *''init-noop-init-type''* })
∪ { *''init-phase-mark''*, *''mark-read-fM''*, *''mark-write-fA''*, *''mark-noop-mfence''*, *''mark-noop-init-type''*
}
⟨*ML*⟩

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

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

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

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

definition *hs-get-roots-locs* ≡
prefixed ''mark-loop-get-roots'' − {*''mark-loop-get-roots-init-type''*}

⟨*ML*⟩

definition *hs-get-work-locs* ≡
  prefixed ″*mark-loop-get-work*″ − {″*mark-loop-get-work-init-type*″}
⟨*ML*⟩

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

definition *hs-init-loop-locs* ≡ ⋃ *l* ∈ *hs-prefixes*. *prefixed* (*l* @ ″*-init-loop*″)
⟨*ML*⟩

definition *hs-done-loop-locs* ≡ ⋃ *l* ∈ *hs-prefixes*. *prefixed* (*l* @ ″*-done-loop*″)
⟨*ML*⟩

definition *hs-done-locs* ≡ ⋃ *l* ∈ *hs-prefixes*. *prefixed* (*l* @ ″*-done*″)
⟨*ML*⟩

definition *hs-none-pending-locs* ≡ − (*hs-init-loop-locs* ∪ *hs-done-locs*)
⟨*ML*⟩

definition *hs-in-sync-locs* ≡
  (− ((⋃ *l* ∈ *hs-prefixes*. *prefixed* (*l* @ ″*-init*″)) ∪ *hs-done-locs*))
  ∪ (⋃ *l* ∈ *hs-prefixes*. {*l* @ ″*-init-type*″})
⟨*ML*⟩

definition *hs-out-of-sync-locs* ≡
  ⋃ *l* ∈ *hs-prefixes*. {*l* @ ″*-init-muts*″}
⟨*ML*⟩

definition *hs-mut-in-muts-locs* ≡
  ⋃ *l* ∈ *hs-prefixes*. {*l* @ ″*-init-loop-set-pending*″, *l* @ ″*-init-loop-done*″}
⟨*ML*⟩

definition *hs-init-loop-done-locs* ≡
  ⋃ *l* ∈ *hs-prefixes*. {*l* @ ″*-init-loop-done*″}
⟨*ML*⟩

definition *hs-init-loop-not-done-locs* ≡
  hs-init-loop-locs − (⋃ *l* ∈ *hs-prefixes*. {*l* @ ″*-init-loop-done*″})
⟨*ML*⟩

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

and atS-gc hs-mut-in-muts-locs (*gc-mut in gc-muts*)
and atS-gc hs-init-loop-locs (*ALLS m. not* ⟨*m*⟩ *in gc-muts imp sys-handshake-pending m*
or sys-ghost-handshake-in-sync m)
and atS-gc hs-init-loop-not-done-locs (*ALLS m.* ⟨*m*⟩ *in gc-muts imp not sys-handshake-pending m*
and not sys-ghost-handshake-in-sync m)
and atS-gc hs-init-loop-done-locs ((*sys-handshake-pending* ▷ *gc-mut*
or sys-ghost-handshake-in-sync ▷ *gc-mut*)
and (*ALLS m.* ⟨*m*⟩ *in gc-muts and* ⟨*m*⟩ *neq gc-mut*
imp not sys-handshake-pending m
and not sys-ghost-handshake-in-sync m))

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

and atS-gc hp-Idle-locs (*sys-ghost-handshake-phase eq* ⟨*hp-Idle*⟩)
and atS-gc hp-IdleInit-locs (*sys-ghost-handshake-phase eq* ⟨*hp-IdleInit*⟩)
and atS-gc hp-InitMark-locs (*sys-ghost-handshake-phase eq* ⟨*hp-InitMark*⟩)
and atS-gc hp-IdleMarkSweep-locs (*sys-ghost-handshake-phase eq* ⟨*hp-IdleMarkSweep*⟩)
and atS-gc hp-Mark-locs (*sys-ghost-handshake-phase eq* ⟨*hp-Mark*⟩)⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*

Local handshake phase invariant for the mutators.

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

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

Relate *sys-ghost-handshake-phase*, *gc-phase*, *sys-phase* and writes to the phase in the GC's

TSO buffer.

The first relation treats the case when the GC's TSO buffer does not contain any writes to the phase.

The second relation exhibits the data race on the phase variable: we need to precisely track the possible states of the GC's TSO buffer.

definition *handshake-phase-rel* :: *handshake-phase* ⇒ *bool* ⇒ *gc-phase* ⇒ *bool* **where**
handshake-phase-rel hp in-sync ph ≡
case hp of
hp-Idle ⇒ *ph* = *ph-Idle*
| *hp-IdleInit* ⇒ *ph* = *ph-Idle* ∨ (*in-sync* ∧ *ph* = *ph-Init*)
| *hp-InitMark* ⇒ *ph* = *ph-Init* ∨ (*in-sync* ∧ *ph* = *ph-Mark*)
| *hp-Mark* ⇒ *ph* = *ph-Mark*
| *hp-IdleMarkSweep* ⇒ *ph* = *ph-Mark* ∨ (*in-sync* ∧ *ph* ∈ { *ph-Idle*, *ph-Sweep* })

definition *phase-rel* :: (*bool* × *handshake-phase* × *gc-phase* × *gc-phase* × ('*field*, '*ref*) *mem-write-action list*) *set* **where**
phase-rel ≡
{ (*in-sync*, *hp*, *ph*, *ph*, []) |*in-sync hp ph*. *handshake-phase-rel hp in-sync ph* }
∪ ({*True*} × { (*hp-IdleInit*, *ph-Init*, *ph-Idle*, [*mw-Phase ph-Init*]),
(*hp-InitMark*, *ph-Mark*, *ph-Init*, [*mw-Phase ph-Mark*]),
(*hp-IdleMarkSweep*, *ph-Sweep*, *ph-Mark*, [*mw-Phase ph-Sweep*]),
(*hp-IdleMarkSweep*, *ph-Idle*, *ph-Mark*, [*mw-Phase ph-Sweep*, *mw-Phase ph-Idle*]),
(*hp-IdleMarkSweep*, *ph-Idle*, *ph-Sweep*, [*mw-Phase ph-Idle*]) })

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

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

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

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

Validity of *sys-fM* wrt *gc-fM* and the handshake phase. Effectively we use *gc-fM* as ghost state. We also include the TSO lock to rule out the GC having any pending marks during the *hp-Idle* handshake phase.

definition *fM-rel* :: (*bool* × *handshake-phase* × *gc-mark* × *gc-mark* × (*'field*, *'ref*) *mem-write-action list* × *bool*) *set* **where**
fM-rel =
{ (*in-sync*, *hp*, *fM*, *fM*, [], *l*) |*fM hp in-sync l*. *hp* = *hp-Idle* ⟶ ¬*in-sync* }
∪ { (*in-sync*, *hp-Idle*, *fM*, *fM'*, [], *l*) |*fM fM' in-sync l*. *in-sync* }
∪ { (*in-sync*, *hp-Idle*, ¬*fM*, *fM*, [*mw-fM* (¬*fM*)], *False*) |*fM in-sync*. *in-sync* }

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

definition *fA-rel* :: (*bool* × *handshake-phase* × *gc-mark* × *gc-mark* × (*'field*, *'ref*) *mem-write-action list*) *set* **where**
fA-rel =
{ (*in-sync*, *hp-Idle*, *fA*, *fM*, []) |*fA fM in-sync*. ¬*in-sync* ⟶ *fA* = *fM* }
∪ { (*in-sync*, *hp-IdleInit*, *fA*, ¬*fA*, []) |*fA in-sync*. *True* }
∪ { (*in-sync*, *hp-InitMark*, *fA*, ¬*fA*, [*mw-fA* (¬*fA*)]) |*fA in-sync*. *in-sync* }
∪ { (*in-sync*, *hp-InitMark*, *fA*, *fM*, []) |*fA fM in-sync*. ¬*in-sync* ⟶ *fA* ≠ *fM* }
∪ { (*in-sync*, *hp-Mark*, *fA*, *fA*, []) |*fA in-sync*. *True* }
∪ { (*in-sync*, *hp-IdleMarkSweep*, *fA*, *fA*, []) |*fA in-sync*. *True* }

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

definition *fM-eq-locs* :: *location set* **where**
fM-eq-locs ≡ (− { *''idle-write-fM''*, *''idle-flip-noop-mfence''* })
⟨*ML*⟩

definition *fM-tso-empty-locs* :: *location set* **where**
fM-tso-empty-locs ≡ (− { *''idle-flip-noop-mfence''* })
⟨*ML*⟩

definition *fA-tso-empty-locs* :: *location set* **where**
fA-tso-empty-locs ≡ (− { *''mark-noop-mfence''* })
⟨*ML*⟩

definition *fA-eq-locs* :: *location set* **where**
fA-eq-locs ≡ { *''idle-read-fM''*, *''idle-invert-fM''* }
∪ *prefixed ''idle-noop''*
∪ (*mark-locs* − { *''mark-read-fM''*, *''mark-write-fA''*, *''mark-noop-mfence''* })
∪ *sweep-locs*
⟨*ML*⟩

definition *fA-neq-locs* :: *location set* **where**
fA-neq-locs ≡ { *''idle-phase-init''*, *''idle-write-fM''*, *''mark-read-fM''*, *''mark-write-fA''* }
∪ *prefixed ''idle-flip-noop''*

$\cup$ *init-locs*
$\langle ML \rangle$

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

 and atS-gc fA-eq-locs (*sys-fA eq gc-fM*)
 and atS-gc fA-neq-locs (*sys-fA neq gc-fM*)
 and at-gc ''mark-noop-mfence'' (*sys-fA neq gc-fM imp* (*not list-null* (*tso-pending-fA gc*)))
 and atS-gc fA-tso-empty-locs (*list-null* (*tso-pending-fA gc*))$\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proo$

3.7 Object colours, reference validity, worklist validity

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

White potential garbage, not yet reached

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

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

In this particular setting we use the following interpretation:

White: not marked

Grey: on a worklist

Black: marked and not on a worklist

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

abbreviation *marked* :: *'ref* $\Rightarrow$ (*'field*, *'mut*, *'ref*) *lsts-pred* **where**
 marked r s $\equiv$ *obj-at* (λ*obj*. *obj-mark obj* = *sys-fM s*) *r s*

abbreviation *white* :: *'ref* $\Rightarrow$ (*'field*, *'mut*, *'ref*) *lsts-pred* **where**
 white r s $\equiv$ *obj-at* (λ*obj*. *obj-mark obj* = ($\neg$*sys-fM s*)) *r s*

definition *WL* :: *'mut process-name* $\Rightarrow$ (*'field*, *'mut*, *'ref*) *lsts* $\Rightarrow$ *'ref set* **where**
 WL p $\equiv$ λ*s*. *W* (*s p*) $\cup$ *ghost-honorary-grey* (*s p*)

definition *grey* :: *'ref* $\Rightarrow$ (*'field*, *'mut*, *'ref*) *lsts-pred* **where**
 grey r $\equiv$ *EXS p*. $\langle r \rangle$ *in WL p*

definition *black* :: *'ref* ⇒ (*'field*, *'mut*, *'ref*) *lsts-pred* **where**
black r ≡ *marked r and not grey r*

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

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

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

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

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

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

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

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

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

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

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* (λ(-::*unit*) *s*. *s p*) *p-ph-enabled*
begin

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

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

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

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

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

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

abbreviation (*input*)
  p-cas-mark-inv ≡ (λ*s*. *obj-at* (λ*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-ghg-eq-ref ≡ *p-ghost-honorary-grey eq pred-singleton* (*the* ∘ *p-ref*)
abbreviation (*input*)
p-ghg-inv ≡ *If p-cas-mark eq p-mark Then p-ghg-eq-ref Else empty p-ghost-honorary-grey*

definition *mark-object-invL* :: (*'field*, *'mut*, *'ref*) *gc-pred* **where**
mark-object-invL ≡
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-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 white* ▷ *p-the-ref and p-tso-no-pending-mark*)
and at-p ''-mo-co-unlock'' (*p-ghg-inv and p-valid-ref and p-valid-fM and p-valid-W-inv*)
and at-p ''-mo-co-won'' (*p-ghg-inv and p-valid-ref and p-valid-fM and marked* ▷ *p-the-ref and p-tso-no-pending-mutate*)
and at-p ''-mo-co-W'' (*p-ghg-eq-ref and p-valid-ref and p-valid-fM and marked* ▷ *p-the-ref and p-tso-no-pending-mutate*)⟨*proof*⟩
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*⟩
⟨*proof*⟩

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

interpretation *mut-get-roots*: *mark-object mutator m ''hs-get-roots-loop''* ⟨*True*⟩ **for** *m*
⟨*proof*⟩

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

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

interference by an *fM* flip.

interpretation *mut-store-del*: *mark-object mutator m ''store-del'' mut-m.mut-ghost-handshake-phase m neq ⟨hp-Idle⟩* **for** *m*
⟨*proof*⟩

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*
⟨*proof*⟩

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''
⟨*ML*⟩

definition *mut-hs-get-roots-loop-mo-locs* ≡
prefixed ''hs-get-roots-loop-mo'' ∪ {*''hs-get-roots-loop-done''*}
⟨*ML*⟩

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*}
⟨*ML*⟩

definition *mut-async-mo-ptest-locs* ≡
⋃*pref*∈*mut-async-mark-object-prefixes*. {*pref* @ *''-mo-ptest''*}
⟨*ML*⟩

definition *mut-mo-ptest-locs* ≡
⋃*pref*∈*mut-async-mark-object-prefixes*. {*pref* @ *''-mo-ptest''*}
⟨*ML*⟩

definition *mut-mo-valid-ref-locs* ≡
prefixed ''store-del'' ∪ *prefixed ''store-ins''* ∪ { *''deref-del''*, *''lop-store-ins''*}
⟨*ML*⟩⟨*proof*⟩⟨*proof*⟩

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*″ }.
⋃ *l*∈{ ″*mo-co-unlock*″, ″*mo-co-won*″, ″*mo-co-W*″ }. {*pref* @ ″-″ @ *l*})
⟨*ML*⟩

definition (**in** *mut-m*) *mark-object-invL* :: (′*field*, ′*mut*, ′*ref*) *gc-pred* **where**
[*inv*]: *mark-object-invL* ≡
atS-mut mut-hs-get-roots-loop-locs (*mut-refs subseteq mut-roots and* (*ALLS r*. ⟨*r*⟩ *in mut-roots diff mut-refs imp marked r*))
and atS-mut mut-hs-get-roots-loop-mo-locs (*not null mut-ref and mut-the-ref in mut-roots*)
and at-mut ″*hs-get-roots-loop-done*″ (*marked* ▷ *mut-the-ref*)
and at-mut ″*hs-get-roots-loop-mo-ptest*″ (*mut-phase neq* ⟨*ph-Idle*⟩)
and at-mut ″*hs-get-roots-done*″ (*ALLS r*. ⟨*r*⟩ *in mut-roots imp marked r*)

and atS-mut mut-mo-valid-ref-locs ((*not null mut-new-ref imp mut-the-new-ref in mut-roots*)
and (*mut-tmp-ref in mut-roots*))
and at-mut ″*store-del-mo-null*″ (*not null mut-ref imp mut-the-ref in mut-ghost-honorary-root*)
and atS-mut (*prefixed* ″*store-del*″ − {″*store-del-mo-null*″}) (*mut-the-ref in mut-ghost-honorary-root*)
and atS-mut (*prefixed* ″*store-ins*″) (*mut-ref eq mut-new-ref*)

and atS-mut (*suffixed* ″*-mo-ptest*″) (*mut-phase neq* ⟨*ph-Idle*⟩ *imp mut-ghost-handshake-phase neq* ⟨*hp-Idle*⟩)
and atS-mut mut-hs-not-hp-Idle-locs (*mut-ghost-handshake-phase neq* ⟨*hp-Idle*⟩)

and atS-mut mut-mo-ptest-locs (*mut-phase eq* ⟨*ph-Idle*⟩ *imp* (*mut-ghost-handshake-phase in* ⟨{*hp-Idle*, *hp-IdleInit*}⟩
or (*mut-ghost-handshake-phase eq* ⟨*hp-IdleMarkSweep*⟩
and sys-phase eq ⟨*ph-Idle*⟩))))

and atS-mut ghost-honorary-grey-empty-locs (*empty mut-ghost-honorary-grey*)

(∗ *insertion barrier* ∗)
and at-mut ″*store-ins*″ ((*mut-ghost-handshake-phase in* ⟨{*hp-InitMark*, *hp-Mark*}⟩
or (*mut-ghost-handshake-phase eq* ⟨*hp-IdleMarkSweep*⟩ *and sys-phase neq* ⟨*ph-Idle*⟩))
and not null mut-new-ref
imp marked ▷ *mut-the-new-ref*)
(∗ *deletion barrier* ∗)

and atS-mut (*prefixed* ″*store-del-mo*″ ∪ {″*lop-store-ins*″})
((*mut-ghost-handshake-phase eq* ⟨*hp-Mark*⟩
or (*mut-ghost-handshake-phase eq* ⟨*hp-IdleMarkSweep*⟩
and sys-phase neq ⟨*ph-Idle*⟩))
and (λ*s*. ∀ *opt-r*′. ¬*tso-pending-write* (*mutator m*) (*mw-Mutate* (*mut-tmp-ref s*) (*mut-field s*) *opt-r*′) *s*)
imp (λ*s*. *obj-at-field-on-heap* (λ*r*. *mut-ref s* = *Some r* ∨ *marked r s*) (*mut-tmp-ref s*) (*mut-field s*) *s*))

and at-mut ″*lop-store-ins*″ ((*mut-ghost-handshake-phase eq* ⟨*hp-Mark*⟩
or (*mut-ghost-handshake-phase eq* ⟨*hp-IdleMarkSweep*⟩
and sys-phase neq ⟨*ph-Idle*⟩))
and not null mut-ref
imp marked ▷ *mut-the-ref*)
and atS-mut (*prefixed* ″*store-ins*″)
((*mut-ghost-handshake-phase eq* ⟨*hp-Mark*⟩
or (*mut-ghost-handshake-phase eq* ⟨*hp-IdleMarkSweep*⟩
and sys-phase neq ⟨*ph-Idle*⟩))
and (λ*s*. ∀ *opt-r*′. ¬*tso-pending-write* (*mutator m*) (*mw-Mutate* (*mut-tmp-ref s*) (*mut-field s*) *opt-r*′) *s*)
imp (λ*s*. *obj-at-field-on-heap* (λ*r*′. *marked r*′ *s*) (*mut-tmp-ref s*) (*mut-field s*) *s*))⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proo*

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* (λ*s*. *obj-at-field-on-heap* (λ*r*. *marked r s*) (*gc-tmp-ref s*) *f s*)⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

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

definition (**in** *gc*) *obj-fields-marked-invL* :: (′*field*, ′*mut*, ′*ref*) *gc-pred* **where**
[*inv*]: *obj-fields-marked-invL* ≡
atS-gc obj-fields-marked-locs (*obj-fields-marked-inv and gc-tmp-ref in gc-W*)
and atS-gc (*prefixed* ″*mark-loop-mo*″ ∪ { ″*mark-loop-mark-field-done*″ }) (λ*s*. *obj-at-field-on-heap* (λ*r*. *gc-ref s* = *Some r* ∨ *marked r s*) (*gc-tmp-ref s*) (*gc-field s*) *s*)
and atS-gc (*prefixed* ″*mark-loop-mo*″) (*ALLS y*. *not null gc-ref and* (λ*s*. ((*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)⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨pro

3.9 The strong-tricolour invariant

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

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

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

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

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

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

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

lemma *strong-tricolour-inv s* ⟹ *weak-tricolour-inv s⟨proof⟩*

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

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

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

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

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

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

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

abbreviation *marked-deletion* :: (*'field*, *'ref*) *mem-write-action* ⇒ (*'field*, *'mut*, *'ref*) *lsts-pred* **where**
marked-deletion w ≡ λ*s*. *case w of mw-Mutate r f opt-r'* ⇒ *obj-at-field-on-heap* (λ*r'*. *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*⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨

3.10 Invariants

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

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

primrec (**in** *mut-m*) *mutator-phase-inv-aux* :: *handshake-phase* ⇒ (*'field*, *'mut*, *'ref*) *lsts-pred* **where**
mutator-phase-inv-aux hp-Idle = ⟨*True*⟩
| *mutator-phase-inv-aux hp-IdleInit* = *no-black-refs*
| *mutator-phase-inv-aux hp-InitMark* = *marked-insertions*
| *mutator-phase-inv-aux hp-Mark* = (*marked-insertions and marked-deletions*)
| *mutator-phase-inv-aux hp-IdleMarkSweep* = (*marked-insertions and marked-deletions and reachable-snapshot-inv*)

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

abbreviation *mutators-phase-inv* :: (*'field*, *'mut*, *'ref*) *lsts-pred* **where**
mutators-phase-inv ≡ *ALLS m*. *mut-m.mutator-phase-inv m*

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

primrec *sys-phase-inv-aux* :: *handshake-phase* ⇒ (*'field*, *'mut*, *'ref*) *lsts-pred* **where**
sys-phase-inv-aux hp-Idle = ((*If sys-fA eq sys-fM Then black-heap Else white-heap*) *and no-grey-refs*)
| *sys-phase-inv-aux hp-IdleInit* = *no-black-refs*
| *sys-phase-inv-aux hp-InitMark* = (*sys-fA neq sys-fM imp no-black-refs*)
| *sys-phase-inv-aux hp-Mark* = ⟨*True*⟩
| *sys-phase-inv-aux hp-IdleMarkSweep* = ((*sys-phase eq* ⟨*ph-Idle*⟩ *or tso-pending-write gc* (*mw-Phase ph-Idle*)) *imp no-grey-refs*)

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

3.11 Lonely mutator assertions

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

definition *ghost-honorary-root-empty-locs* :: *location set* **where**
ghost-honorary-root-empty-locs ≡
− (*prefixed ''store-del''* ∪ {*''lop-store-ins''*} ∪ *prefixed ''store-ins''*)
⟨*ML*⟩

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

3.12 The infamous termination argument.

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

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

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

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

definition *black-heap-locs* ≡ { ″*sweep-idle*″, ″*idle-noop-mfence*″, ″*idle-noop-init-type*″ }
⟨*ML*⟩

definition *no-grey-refs-locs* ≡ *black-heap-locs* ∪ *sweep-locs* ∪ {″*mark-end*″}
⟨*ML*⟩

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

3.13 Sweep loop invariants

definition *sweep-loop-locs* ≡ *prefixed* ″*sweep-loop*″
⟨*ML*⟩

definition (**in** *gc*) *sweep-loop-invL* :: (′*field*, ′*mut*, ′*ref*) *gc-pred* **where**
[*inv*]: *sweep-loop-invL* ≡
at-gc ″*sweep-loop-check*″ ((*not null gc-mark imp* (λ*s. obj-at* (λ*obj. Some* (*obj-mark obj*) = *gc-mark s*) (*gc-tmp-ref s*) *s*))
and (*null gc-mark imp* (*marked* ▷ *gc-tmp-ref or not valid-ref* ▷ *gc-tmp-ref*)))
and at-gc ″*sweep-loop-free*″ (*not null gc-mark and the* ∘ *gc-mark neq gc-fM and* (λ*s. obj-at* (λ*obj. Some* (*obj-mark obj*) = *gc-mark s*) (*gc-tmp-ref s*) *s*))
and at-gc ″*sweep-loop-ref-done*″ (*marked* ▷ *gc-tmp-ref or not valid-ref* ▷ *gc-tmp-ref*)
and atS-gc sweep-loop-locs (*ALLS r. not* ⟨*r*⟩ *in gc-refs imp* (*marked r or not valid-ref r*))
and atS-gc black-heap-locs (*ALLS r. marked r or not valid-ref r*)
and atS-gc (*prefixed* ″*sweep-loop-*″ − { ″*sweep-loop-choose-ref*″ }) (*gc-tmp-ref in gc-refs*)⟨*proof*⟩⟨*proof*⟩⟨*proo*

4 Top-level safety

4.1 Invariants

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

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

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

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

4.2 Initial conditions

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

- All objects on the heap are marked (have their flags equal to *sys-fM*, and there are no

grey references, i.e. the heap is uniformly black.

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

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

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

definition *gc-initial-state* :: (*'field*, *'mut*, *'ref*) *lst-pred* **where**
  gc-initial-state s ≡
    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 ≡

$(\lambda s.\ gc\text{-}initial\text{-}state\ (s\ gc))$
and $(\lambda s.\ \forall m.\ mut\text{-}initial\text{-}state\ (s\ (mutator\ m)))$
and $(\lambda s.\ sys\text{-}initial\text{-}state\ (s\ sys))$
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**
$gc\text{-}system \equiv (gc\text{-}pgms, gc\text{-}system\text{-}init)\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle$

theorem *inv*: $s \in reachable\text{-}states\ gc\text{-}system \Longrightarrow I\ (mkP\ s)\langle proof\rangle$

Our headline safety result follows directly.

corollary *safety*:
$s \in reachable\text{-}states\ gc\text{-}system \Longrightarrow valid\text{-}refs\ (mkP\ s)\downarrow\langle proof\rangle$

end

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

interpretation *gc-system-interpretation*: *gc-system undefined* $\langle proof\rangle$

4.3 A concrete system state

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

theory *Concrete-heap*
imports
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**
$mut\text{-}common\text{-}init\text{-}state \equiv undefined(\!|\ ghost\text{-}handshake\text{-}phase := hp\text{-}IdleMarkSweep, ghost\text{-}honorary\text{-}grey := \{\}, ghost\text{-}honorary\text{-}root := \{\}, roots := \{\}, W := \{\}\ |\!)$

context *gc-system*
begin

abbreviation *sys-init-heap* :: *ref* ⇒ (*field*, *ref*) *object option* **where**
sys-init-heap ≡
[*0* ↦ (| *obj-mark* = *initial-mark*,

```
        obj-fields = [ 0 ↦ 5 ] |),
    1 ↦ (| obj-mark = initial-mark,
        obj-fields = Map.empty |),
    2 ↦ (| obj-mark = initial-mark,
        obj-fields = Map.empty |),
    3 ↦ (| obj-mark = initial-mark,
        obj-fields = [ 0 ↦ 1 , 1 ↦ 2 ] |),
    4 ↦ (| obj-mark = initial-mark,
        obj-fields = [ 1 ↦ 0 ] |),
    5 ↦ (| obj-mark = initial-mark,
        obj-fields = Map.empty |)
 ]

abbreviation mut-init-state0 :: concrete-local-state where
  mut-init-state0 ≡ mut-common-init-state (| roots := {1, 2, 3} |)

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

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

end

end

context gc-system
begin

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

abbreviation gc-init-state :: concrete-local-state where
  gc-init-state ≡
    undefined(| fM := initial-mark
```

```
        , fA := initial-mark
        , phase := ph-Idle
        , W := {}
        , ghost-honorary-grey := {} ⦈

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

abbreviation muts-init-states :: (mut × concrete-local-state) list where
  muts-init-states ≡ [ (0, mut-init-state0), (1, mut-init-state1), (2, mut-init-state2) ]

abbreviation init-state :: clsts where
  init-state ≡ λp. case p of
          gc ⇒ gc-init-state
        | sys ⇒ sys-init-state
        | mutator m ⇒ lookup muts-init-states mut-common-init-state m

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

end
```

References

E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11): 966–975, 1978.

D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In H.-J. Boehm, B. Lang, and D. M. Yellin, editors, *POPL*, pages 70–83. ACM Press, 1994.

Peter Gammie. Concurrent IMP. *Archive of Formal Proofs*, April 2015. ISSN 2150-914x. http://isa-afp.org/entries/ConcurrentIMP.shtml, Formal proof development.

S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-tso. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer, 2009. doi: 10.1007/978-3-642-03359-9_27.

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

F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–159, Toronto, Canada, June 2010. doi: 10.1145/1806596.1806615.

N. Schirmer and M. Wenzel. State spaces - the locale way. *Electr. Notes Theor. Comput. Sci.*, 254:161–179, 2009.

P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53 (7):89–97, 2010.