

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

Peter Gammie, Tony Hosking and Kai Engelhardt

August 16, 2018

Abstract

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

Contents

1	Introduction	2
2	The Schism garbage collector	2
2.1	Object marking	6
2.2	Handshakes	8
2.3	The system process	12
2.4	Mutators	15
2.5	Garbage collector	17
3	Invariants and Proofs	19
3.1	Constructors for sets of locations.	19
3.2	Hoare triples	20
3.3	Functions and predicates	20
3.4	Garbage collector locations.	24
3.5	Coarse TSO invariants	24
3.5.1	Locations where the TSO lock is held	25
3.6	Handshake phases	25
3.7	Object colours, reference validity, worklist validity	31
3.8	Mark Object	32
3.9	The strong-tricolour invariant	38
3.10	Invariants	39
3.11	Lonely mutator assertions	40
3.12	The infamous termination argument.	40
3.13	Sweep loop invariants	41

4	Top-level safety	41
4.1	Invariants	41
4.2	Initial conditions	42
4.3	A concrete system state	44
	References	46

1 Introduction

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

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

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

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

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

2 The Schism garbage collector

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

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

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

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

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* \Rightarrow (*'field*, *'ref*) *object option*
- TSO memory state
 - mem-write-buffers* :: *'mut process-name* \Rightarrow (*'field*, *'ref*) *mem-write-action list*
 - mem-lock* :: *'mut process-name option*
- The state of the handshakes
 - handshake-type* :: *handshake-type*
 - handshake-pending* :: *'mut* \Rightarrow *bool*
- Ghost state
 - ghost-handshake-in-sync* :: *'mut* \Rightarrow *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* \equiv *ro-Read mr-fM*
abbreviation *ReadMark r* \equiv *ro-Read (mr-Mark r)*
abbreviation *ReadPhase* \equiv *ro-Read mr-Phase*
abbreviation *ReadRef r f* \equiv *ro-Read (mr-Ref r f)*

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

type-synonym ('field, 'mut, 'ref) *request*
 = 'mut *process-name* \times ('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 $lock :: location \Rightarrow ('field, 'mut, 'ref) gc-com$ **where**

$lock\ l \equiv \{\!|l|\} Request (\lambda s. (p, ro-Lock)) (\lambda s. \{s\})$

notation $lock\ (\{\!|-|\} lock)$

abbreviation $unlock :: location \Rightarrow ('field, 'mut, 'ref) gc-com$ **where**

$unlock\ l \equiv \{\!|l|\} Request (\lambda s. (p, ro-Unlock)) (\lambda s. \{s\})$

notation $unlock\ (\{\!|-|\} unlock)$

abbreviation

$read-mark :: location \Rightarrow (('field, 'mut, 'ref) local-state \Rightarrow 'ref)$

$\Rightarrow ((gc-mark\ option \Rightarrow gc-mark\ option)$

$\Rightarrow ('field, 'mut, 'ref) local-state$

$\Rightarrow ('field, 'mut, 'ref) local-state) \Rightarrow ('field, 'mut, 'ref) gc-com$

where

$read-mark\ l\ r\ upd \equiv \{\!|l|\} Request (\lambda s. (p, ReadMark (r\ s))) (\lambda mv\ s. \{ upd\ \langle m \rangle\ s\ |m. mv = mv-Mark\ m\ \})$

notation $read-mark\ (\{\!|-|\} read'-mark)$

abbreviation $read-fM :: location \Rightarrow ('field, 'mut, 'ref) gc-com$ **where**

$read-fM\ l \equiv \{\!|l|\} Request (\lambda s. (p, ro-Read\ mr-fM)) (\lambda mv\ s. \{ s(\!fM := m) |m. mv = mv-Mark\ (Some\ m)\ \})$

notation $read-fM\ (\{\!|-|\} read'-fM)$

abbreviation

$read-phase :: location \Rightarrow ('field, 'mut, 'ref) gc-com$

where

$read-phase\ l \equiv \{\!|l|\} Request (\lambda s. (p, ReadPhase)) (\lambda mv\ s. \{ s(\!phase := ph) |ph. mv = mv-Phase\ ph\ \})$

notation $read-phase\ (\{\!|-|\} read'-phase)$

abbreviation $write-mark :: location \Rightarrow (('field, 'mut, 'ref) local-state \Rightarrow 'ref) \Rightarrow (('field, 'mut, 'ref) local-state \Rightarrow bool) \Rightarrow ('field, 'mut, 'ref) gc-com$ **where**

$write-mark\ l\ r\ fl \equiv \{\!|l|\} Request (\lambda s. (p, WriteMark (r\ s) (fl\ s))) (\lambda s. \{ s(\!ghost-honorary-grey := \{r\ s\} \}) \})$

notation $write-mark\ (\{\!|-|\} write'-mark)$

abbreviation $add-to-W :: location \Rightarrow (('field, 'mut, 'ref) local-state \Rightarrow 'ref) \Rightarrow ('field, 'mut, 'ref) gc-com$ **where**

$add-to-W\ l\ r \equiv \{\!|l|\} [\lambda s. s(\!W := W\ s \cup \{r\ s\}, ghost-honorary-grey := \{\}) \!]$

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* \Rightarrow (*'field*, *'mut*, *'ref*) *gc-com* **where**

```

mark-object-fn l  $\equiv$ 
  {l @ "-mo-null"} IF not null ref THEN
    {l @ "-mo-mark"} read-mark (the  $\circ$  ref) mark-update ;;
    {l @ "-mo-fM"} read-fM ;;
    {l @ "-mo-mtest"} IF mark neq Some  $\circ$  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  $\circ$  ref) cas-mark-update ;;
        {l @ "-mo-co-ctest"} IF cas-mark eq mark THEN
          {l @ "-mo-co-mark"} write-mark (the  $\circ$  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  $\circ$  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* \Rightarrow *handshake-phase* \Rightarrow *handshake-phase* **where**

```

hp-step ht  $\equiv$ 
  case ht of

```


$ht\text{-}NOOP \Rightarrow handshake\text{-}step$
 $| ht\text{-}GetRoots \Rightarrow handshake\text{-}step$
 $| ht\text{-}GetWork \Rightarrow id$

context *sys*
begin

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

handshake \equiv
 $\{\{ "sys\text{-}hs\text{-}gc\text{-}set\text{-}type" \} \}$ Response
 $(\lambda req\ s. \{ (s \{ handshake\text{-}type := ht,$
 $\quad ghost\text{-}handshake\text{-}in\text{-}sync := \langle False \rangle,$
 $\quad ghost\text{-}handshake\text{-}phase := hp\text{-}step\ ht\ (ghost\text{-}handshake\text{-}phase\ s) \},$
 $\quad mv\text{-}Void)$
 $| ht. req = (gc, ro\text{-}hs\text{-}gc\text{-}set\text{-}type\ ht) \}$)
 $\sqcup \{\{ "sys\text{-}hs\text{-}gc\text{-}mut\text{-}reqs" \} \}$ Response
 $(\lambda req\ s. \{ (s \{ handshake\text{-}pending := (handshake\text{-}pending\ s)(m := True) \}, mv\text{-}Void)$
 $| m. req = (gc, ro\text{-}hs\text{-}gc\text{-}set\text{-}pending\ m) \}$)
 $\sqcup \{\{ "sys\text{-}hs\text{-}gc\text{-}done" \} \}$ Response
 $(\lambda req\ s. \{ (s, mv\text{-}Bool\ (\neg handshake\text{-}pending\ s\ m))$
 $| m. req = (gc, ro\text{-}hs\text{-}gc\text{-}read\text{-}pending\ m) \}$)
 $\sqcup \{\{ "sys\text{-}hs\text{-}gc\text{-}load\text{-}W" \} \}$ Response
 $(\lambda req\ s. \{ (s \{ W := \{ \} \}, mv\text{-}Refs\ (W\ s))$
 $| :::unit. req = (gc, ro\text{-}hs\text{-}gc\text{-}load\text{-}W) \}$)
 $\sqcup \{\{ "sys\text{-}hs\text{-}mut" \} \}$ Response
 $(\lambda req\ s. \{ (s, mv\text{-}Void)$
 $| m. req = (mutator\ m, ro\text{-}hs\text{-}mut\text{-}read\text{-}type\ (handshake\text{-}type\ s))$
 $\quad \wedge\ handshake\text{-}pending\ s\ m \}$)
 $\sqcup \{\{ "sys\text{-}hs\text{-}mut\text{-}done" \} \}$ Response
 $(\lambda req\ s. \{ (s \{ handshake\text{-}pending := (handshake\text{-}pending\ s)(m := False),$
 $\quad W := W\ s \cup W',$
 $\quad ghost\text{-}handshake\text{-}in\text{-}sync := (ghost\text{-}handshake\text{-}in\text{-}sync\ s)(m := True) \},$
 $\quad mv\text{-}Void)$
 $| m\ W'. req = (mutator\ m, ro\text{-}hs\text{-}mut\text{-}done\ W') \}$)

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 \Rightarrow ('field, 'mut, 'ref) gc-com ($\{\{-\}$ mark'-object) **where**
 $\{\{ l \} \}$ mark-object \equiv mark-object-fn (mutator m) l

abbreviation *mfence* :: location \Rightarrow ('field, 'mut, 'ref) gc-com ($\{\{-\}$ MFENCE) **where**
 $\{\{ l \} \}$ MFENCE \equiv $\{\{ l \} \}$ Request ($\lambda s. (mutator\ m, ro\text{-}MFENCE)$) ($\lambda s. \{ s \}$)

abbreviation *hs-read-type* :: *location* \Rightarrow *handshake-type* \Rightarrow ('field, 'mut, 'ref) *gc-com* ($\{\!-\!\}$ *hs'-read'-type*) **where**

$\{\!|l|\}$ *hs-read-type* *ht* \equiv $\{\!|l|\}$ *Request* ($\lambda s.$ (*mutator* *m*, *ro-hs-mut-read-type* *ht*)) ($\lambda-$ *s*. $\{s\}$)

abbreviation *hs-noop-done* :: *location* \Rightarrow ('field, 'mut, 'ref) *gc-com* ($\{\!-\!\}$ *hs'-noop'-done*) **where**

$\{\!|l|\}$ *hs-noop-done* \equiv $\{\!|l|\}$ *Request* ($\lambda s.$ (*mutator* *m*, *ro-hs-mut-done* $\{\}$))
 $(\lambda-$ *s*. $\{s(\text{ghost-handshake-phase} := \text{handshake-step}$
 $(\text{ghost-handshake-phase } s) \})\})$)

abbreviation *hs-get-roots-done* :: *location* \Rightarrow (('field, 'mut, 'ref) *local-state* \Rightarrow 'ref *set*) \Rightarrow ('field, 'mut, 'ref) *gc-com* ($\{\!-\!\}$ *hs'-get'-roots'-done*) **where**

$\{\!|l|\}$ *hs-get-roots-done* *wl* \equiv $\{\!|l|\}$ *Request* ($\lambda s.$ (*mutator* *m*, *ro-hs-mut-done* (*wl* *s*)))
 $(\lambda-$ *s*. $\{s(\text{ghost-handshake-phase} := \text{handshake-step}$
 $(\text{ghost-handshake-phase } s) \})\})$)

abbreviation *hs-get-work-done* :: *location* \Rightarrow (('field, 'mut, 'ref) *local-state* \Rightarrow 'ref *set*) \Rightarrow ('field, 'mut, 'ref) *gc-com* ($\{\!-\!\}$ *hs'-get'-work'-done*) **where**

$\{\!|l|\}$ *hs-get-work-done* *wl* \equiv $\{\!|l|\}$ *Request* ($\lambda s.$ (*mutator* *m*, *ro-hs-mut-done* (*wl* *s*)))
 $(\lambda-$ *s*. $\{s(\text{ghost-handshake-phase} := \text{handshake-step}$
 $(\text{ghost-handshake-phase } s) \})\})$)

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

handshake \equiv

$\{\!|hs-noop|\}$ *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* \Rightarrow *handshake-type* \Rightarrow (*'field*, *'mut*, *'ref*) *gc-com* ($\{\!-\!\}$ *set'-hs'-type*) **where**

$\{\!l\!\}$ *set-hs-type* *ht* \equiv $\{\!l\!\}$ *Request* ($\lambda s. (gc, ro-hs-gc-set-type\ ht)$) ($\lambda s. \{s\}$)

abbreviation *set-hs-pending* :: *location* \Rightarrow ((*'field*, *'mut*, *'ref*) *local-state* \Rightarrow *'mut*) \Rightarrow (*'field*, *'mut*, *'ref*) *gc-com* ($\{\!-\!\}$ *set'-hs'-pending*) **where**

$\{\!l\!\}$ *set-hs-pending* *m* \equiv $\{\!l\!\}$ *Request* ($\lambda s. (gc, ro-hs-gc-set-pending\ (m\ s))$) ($\lambda s. \{s\}$)

definition

handshake-init :: *location* \Rightarrow *handshake-type* \Rightarrow (*'field*, *'mut*, *'ref*) *gc-com* ($\{\!-\!\}$ *handshake'-init*)

where

$\{\!l\!\}$ *handshake-init* *req* \equiv
 $\{\!l\ @\ \text{"-init-type"}\!\}$ *set-hs-type* *req* ;;
 $\{\!l\ @\ \text{"-init-muts"}\!\}$ *'muts* := *UNIV* ;;
 $\{\!l\ @\ \text{"-init-loop"}\!\}$ *WHILE* *not empty muts DO*
 $\{\!l\ @\ \text{"-init-loop-choose-mut"}\!\}$ *'mut* \in *'muts* ;;
 $\{\!l\ @\ \text{"-init-loop-set-pending"}\!\}$ *set-hs-pending* *mut* ;;
 $\{\!l\ @\ \text{"-init-loop-done"}\!\}$ *'muts* := (*'muts* - {*'mut*})
OD

definition

handshake-done :: *location* \Rightarrow (*'field*, *'mut*, *'ref*) *gc-com* ($\{\!-\!\}$ *handshake'-done*)

where

$\{\!l\!\}$ *handshake-done* \equiv
 $\{\!l\ @\ \text{"-done-muts"}\!\}$ *'muts* := *UNIV* ;;
 $\{\!l\ @\ \text{"-done-loop"}\!\}$ *WHILE* *not empty muts DO*
 $\{\!l\ @\ \text{"-done-loop-choose-mut"}\!\}$ *'mut* \in *'muts* ;;
 $\{\!l\ @\ \text{"-done-loop-rendezvous"}\!\}$ *Request*
($\lambda s. (gc, ro-hs-gc-read-pending\ (mut\ s))$)
($\lambda mv\ s. \{s\mid muts := muts\ s - \{mut\ s \mid done.\ mv = mv-Bool\ done \wedge done\}\}$)
OD

abbreviation *load-W* :: *location* \Rightarrow (*'field*, *'mut*, *'ref*) *gc-com* ($\{\!-\!\}$ *load'-W*) **where**

$\{\!l\!\}$ *load-W* \equiv $\{\!l\ @\ \text{"-load-W"}\!\}$ *Request* ($\lambda s. (gc, ro-hs-gc-load-W)$)
($\lambda resp\ s. \{s\mid W := W'\} \mid W'.\ resp = mv-Refs\ W'$)

abbreviation *mfence* :: *location* \Rightarrow (*'field*, *'mut*, *'ref*) *gc-com* ($\{\!-\!\}$ *MFENCE*) **where**

$\{\!l\!\}$ *MFENCE* \equiv $\{\!l\!\}$ *Request* ($\lambda s. (gc, ro-MFENCE)$) ($\lambda s. \{s\}$)

definition

handshake-noop :: *location* \Rightarrow (*'field*, *'mut*, *'ref*) *gc-com* ($\{\!-\!\}$ *handshake'-noop*)

where

$\{\!l\!\}$ *handshake-noop* \equiv
 $\{\!l\ @\ \text{"-mfence"}\!\}$ *MFENCE* ;;
 $\{\!l\!\}$ *handshake-init* *ht-NOOP* ;;

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

definition

handshake-get-roots :: *location* \Rightarrow (*'field*, *'mut*, *'ref*) *gc-com* ($\{\!|-|\!\}$ *handshake'-get'-roots*)

where

$\{\!|l|\!\}$ *handshake-get-roots* \equiv
 $\{\!|l|\!\}$ *handshake-init ht-GetRoots* ;;
 $\{\!|l|\!\}$ *handshake-done* ;;
 $\{\!|l|\!\}$ *load-W*

definition

handshake-get-work :: *location* \Rightarrow (*'field*, *'mut*, *'ref*) *gc-com* ($\{\!|-|\!\}$ *handshake'-get'-work*)

where

$\{\!|l|\!\}$ *handshake-get-work* \equiv
 $\{\!|l|\!\}$ *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* \Rightarrow *'mut process-name* \Rightarrow *bool*

where

not-blocked *s p* \equiv *case mem-lock s of None* \Rightarrow *True* | *Some p'* \Rightarrow *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* \Rightarrow (*'field*, *'mut*, *'ref*) *local-state* \Rightarrow (*'field*, *'mut*, *'ref*) *local-state* **where**

do-write-action *wact* \equiv $\lambda s.$

case wact of

mw-Mark *r gc-mark* \Rightarrow $s(\!|heap := (heap\ s)(r := map-option (\lambda obj. obj(\!|obj-mark := gc-mark)) (heap\ s\ r))\!|)$

| *mw-Mutate* *r f new-r* \Rightarrow $s(\!|heap := (heap\ s)(r := map-option (\lambda obj. obj(\!|obj-fields := (obj-fields\ obj)(f := new-r)\!|)) (heap\ s\ r))\!|)$

| *mw-fM* *gc-mark* \Rightarrow $s(\!|fM := gc-mark\!|)$

| *mw-fA* *gc-mark* \Rightarrow $s(\!|fA := gc-mark\!|)$

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

definition

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

where

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

abbreviation

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

where

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

definition

$$\begin{aligned} \text{do-read-action} &:: ('field, 'ref) \text{ mem-read-action} \\ &\Rightarrow ('field, 'mut, 'ref) \text{ local-state} \\ &\Rightarrow ('field, 'ref) \text{ response} \end{aligned}$$

where

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

case $ract$ of

$$\begin{aligned} | \text{mr-Ref } r \ f &\Rightarrow \text{mv-Ref } (\text{heap } s \ r \gg\equiv (\lambda obj. \text{obj-fields } obj \ f)) \\ | \text{mr-Mark } r &\Rightarrow \text{mv-Mark } (\text{map-option } \text{obj-mark } (\text{heap } s \ r)) \\ | \text{mr-Phase} &\Rightarrow \text{mv-Phase } (\text{phase } s) \\ | \text{mr-fM} &\Rightarrow \text{mv-Mark } (\text{Some } (fM \ s)) \\ | \text{mr-fA} &\Rightarrow \text{mv-Mark } (\text{Some } (fA \ s)) \end{aligned}$$

definition

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

where

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

context sys

begin

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

definition

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

where

$$\begin{aligned}
\text{mem-TSO} \equiv & \{ \text{"sys-read"} \} \text{Response } (\lambda \text{req } s. \{ (s, \text{sys-read } p \text{ mr } s) \\
& \quad | p \text{ mr. req} = (p, \text{ro-Read } \text{mr}) \wedge \text{not-blocked } s \text{ } p \}) \\
\sqcup & \{ \text{"sys-write"} \} \text{Response } (\lambda \text{req } s. \{ (s \mid \text{mem-write-buffers} := (\text{mem-write-buffers } s)(p \\
:= \text{mem-write-buffers } s \text{ } p \text{ } @ [w] \mid), \text{mv-void}) \\
& \quad | p \text{ w. req} = (p, \text{ro-Write } w) \}) \\
\sqcup & \{ \text{"sys-mfence"} \} \text{Response } (\lambda \text{req } s. \{ (s, \text{mv-void}) \\
& \quad | p. \text{req} = (p, \text{ro-MFENCE}) \wedge \text{mem-write-buffers } s \text{ } p = [] \}) \\
\sqcup & \{ \text{"sys-lock"} \} \text{Response } (\lambda \text{req } s. \{ (s \mid \text{mem-lock} := \text{Some } p \mid), \text{mv-void}) \\
& \quad | p. \text{req} = (p, \text{ro-Lock}) \wedge \text{mem-lock } s = \text{None} \}) \\
\sqcup & \{ \text{"sys-unlock"} \} \text{Response } (\lambda \text{req } s. \{ (s \mid \text{mem-lock} := \text{None} \mid), \text{mv-void}) \\
& \quad | p. \text{req} = (p, \text{ro-Unlock}) \wedge \text{mem-lock } s = \text{Some } p \wedge \\
& \text{mem-write-buffers } s \text{ } p = [] \}) \\
\sqcup & \{ \text{"sys-dequeue-write-buffer"} \} \text{LocalOp } (\lambda s. \{ (\text{do-write-action } w \text{ } s) \mid \text{mem-write-buffers} \\
:= (\text{mem-write-buffers } s)(p := w) \mid) \\
& \quad | p \text{ w } ws. \text{mem-write-buffers } s \text{ } p = w \# ws \wedge \\
& \text{not-blocked } s \text{ } p \wedge p \neq \text{sys} \})
\end{aligned}$$

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

$\text{alloc} :: ('field, 'mut, 'ref) \text{gc-com}$

where

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

References are freed by removing them from *heap*.

definition

$\text{free} :: ('field, 'mut, 'ref) \text{gc-com}$

where

$$\begin{aligned}
\text{free} \equiv & \{ \text{"sys-free"} \} \text{Response } (\lambda \text{req } s. \\
& \{ (s \mid \text{heap} := (\text{heap } s)(r := \text{None}) \mid), \text{mv-void}) \mid r. \text{snd req} = \text{ro-Free } r \})
\end{aligned}$$

The top-level system process.

definition

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

where

$com \equiv$
 $LOOP\ DO$
 $mem\text{-}TSO$
 $\sqcup\ alloc$
 $\sqcup\ free$
 $\sqcup\ handshake$
 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 $mut\text{-}m$

begin

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

abbreviation (**in** $-$) $mut\text{-}alloc :: 'mut \Rightarrow ('field, 'mut, 'ref)\ gc\text{-}com$ **where**

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

abbreviation $alloc :: ('field, 'mut, 'ref)\ gc\text{-}com$ **where**

$alloc \equiv mut\text{-}alloc\ m$

The mutator can always discard any references it holds.

abbreviation $discard :: ('field, 'mut, 'ref)\ gc\text{-}com$ **where**

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

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

Dereferencing a reference can increase the set of mutator roots.

abbreviation $load :: ('field, 'mut, 'ref)\ gc\text{-}com$ **where**

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

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

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

where

$\{\!|l|\!\} deref r f upd \equiv \{\!|l|\!\} Request (\lambda s. (mutator m, ReadRef (r s) (f s)))$
 $(\lambda mv s. \{ upd \langle opt-r \rangle (s(\ghost-honorary-root := set-option opt-r')) \mid opt-r'. mv = mv-Ref opt-r' \})$

abbreviation

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

where

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

definition

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

where

$store \equiv$
 — Choose vars for: $ref \rightarrow field := new-ref$
 $\{\!|store-choose|\!\} LocalOp (\lambda s. \{ s(\ tmp-ref := r, field := f, new-ref := r') \mid r f r'. r \in roots s \wedge r' \in Some ' roots s \cup \{None\} \}) ;;$
 — 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

$com \equiv$


```

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 (λs. (gc, ReadRef (r s) (f s)))
  (λmv s. { upd ⟨r'⟩ s | r'. mv = mv-Ref r' })

```

abbreviation

```

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

```

where

```

gc-read-mark l r upd ≡ {l} Request (λs. (gc, ReadMark (r s))) (λmv s. { upd ⟨m⟩ s | m.
mv = mv-Mark m })

```

syntax

```

-gc-fassign :: location ⇒ idt ⇒ 'ref ⇒ 'field ⇒ ('field, 'mut, 'ref) gc-com ({-} ' - := ' - →
- [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 $write-fA :: location \Rightarrow (('field, 'mut, 'ref) local-state \Rightarrow gc-mark) \Rightarrow ('field, 'mut, 'ref) gc-com \{\{-\}\} write'-fA$ **where**

$\{\{l\}\} write-fA f \equiv \{\{l\}\} Request (\lambda s. (gc, WritefA (f s))) (\lambda- s. \{s\})$

abbreviation $read-fM :: location \Rightarrow ('field, 'mut, 'ref) gc-com \{\{-\}\} read'-fM$ **where**

$\{\{l\}\} read-fM \equiv \{\{l\}\} Request (\lambda s. (gc, ReadfM)) (\lambda mv s. \{ s(\{fM := m\} | m. mv = mv-Mark (Some m)) \})$

abbreviation $write-fM :: location \Rightarrow ('field, 'mut, 'ref) gc-com \{\{-\}\} write'-fM$ **where**

$\{\{l\}\} write-fM \equiv \{\{l\}\} Request (\lambda s. (gc, WritefM (fM s))) (\lambda- s. \{s\})$

abbreviation $write-phase :: location \Rightarrow gc-phase \Rightarrow ('field, 'mut, 'ref) gc-com \{\{-\}\} write'-phase$ **where**

$\{\{l\}\} write-phase ph \equiv \{\{l\}\} Request (\lambda s. (gc, WritePhase ph)) (\lambda- s. \{s(\{ phase := ph \})\})$

abbreviation $mark-object :: location \Rightarrow ('field, 'mut, 'ref) gc-com \{\{-\}\} mark'-object$ **where**

$\{\{l\}\} mark-object \equiv mark-object-fn gc l$

abbreviation $free :: location \Rightarrow (('field, 'mut, 'ref) local-state \Rightarrow 'ref) \Rightarrow ('field, 'mut, 'ref) gc-com \{\{-\}\} free$ **where**

$\{\{l\}\} free r \equiv \{\{l\}\} Request (\lambda s. (gc, ro-Free (r s))) (\lambda- s. \{s\})$

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

$com \equiv$

$LOOP DO$

$\{\{ "idle-noop" \}\} handshake-noop ;; \text{--- } hp-Idle$

$\{\{ "idle-read-fM" \}\} read-fM ;;$

$\{\{ "idle-invert-fM" \}\} 'fM := (\neg 'fM) ;;$

$\{\{ "idle-write-fM" \}\} write-fM ;;$

$\{\{ "idle-flip-noop" \}\} handshake-noop ;; \text{--- } hp-IdleInit$

$\{\{ "idle-phase-init" \}\} write-phase ph-Init ;;$

$\{\{ "init-noop" \}\} handshake-noop ;; \text{--- } hp-InitMark$

$\{\{ "init-phase-mark" \}\} write-phase ph-Mark ;;$

$\{\{ "mark-read-fM" \}\} read-fM ;;$

$\{\{ "mark-write-fA" \}\} write-fA fM ;;$

$\{\{ "mark-noop" \}\} handshake-noop ;; \text{--- } hp-Mark$

```

    {"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* \Rightarrow *location set* **where**
prefixed *p* \equiv { *l* . *prefix p l* }

abbreviation *suffixed* :: *location* \Rightarrow *location set* **where**
suffixed *p* \equiv { *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* \Rightarrow '*mut process-name* \Rightarrow ('*field*, '*mut*, '*ref*) *gc-pred*
 \Rightarrow *bool* ($\{\!\{-\}\!\}$ - $\{\!\{-\}\!\}$)

where

$\{\!\{P\}\!\}$ *p* $\{\!\{Q\}\!\}$ \equiv $\forall (c, \text{afts}) \in \text{vcg-fragments } (gc\text{-pgms } p). (gc\text{-pgms}, p, \text{afts} \models \{\!\{P\}\!\} c \{\!\{Q\}\!\})$

abbreviation

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

where

$\{\!\{P\}\!\}$ *p* \equiv $\{\!\{P\}\!\}$ *p* $\{\!\{P\}\!\}\langle\text{proof}\rangle\langle\text{proof}\rangle\langle\text{proof}\rangle$

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.

$\langle ML \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle ML \rangle$

3.3 Functions and predicates

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

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

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

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

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

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

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

r in-W p \equiv $\lambda s. r \in W (s p)$

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

$r \text{ in-ghost-honorary-grey } p \equiv \lambda s. r \in \text{ghost-honorary-grey } (s \ p)$

context *gc*

begin

abbreviation

$\text{valid-gc-syn} :: ('field, 'mut, 'ref) \text{gc-loc-comp} \Rightarrow ('field, 'mut, 'ref) \text{gc-pred} \Rightarrow ('field, 'mut, 'ref) \text{gc-com} \Rightarrow ('field, 'mut, 'ref) \text{gc-pred} \Rightarrow \text{bool}$
 $(- \models \{\!\{-}\!\} / - / \{\!\{-}\!\})$

where

$\text{afts} \models \{\!\{P\}\!\} \ c \ \{\!\{Q\}\!\} \equiv \text{gc-pgms}, \text{gc}, \text{afts} \models \{\!\{P\}\!\} \ c \ \{\!\{Q\}\!\}$

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

$\text{afts} \models \{\!\{P\}\!\} \ c \equiv \text{afts} \models \{\!\{P\}\!\} \ c \ \{\!\{P\}\!\}$

end

abbreviation $\text{gc-cas-mark } s \equiv \text{cas-mark } (s \ \text{gc})$

abbreviation $\text{gc-fM } s \equiv \text{fM } (s \ \text{gc})$

abbreviation $\text{gc-field } s \equiv \text{field } (s \ \text{gc})$

abbreviation $\text{gc-field-set } s \equiv \text{field-set } (s \ \text{gc})$

abbreviation $\text{gc-mark } s \equiv \text{mark } (s \ \text{gc})$

abbreviation $\text{gc-mut } s \equiv \text{mut } (s \ \text{gc})$

abbreviation $\text{gc-muts } s \equiv \text{muts } (s \ \text{gc})$

abbreviation $\text{gc-phase } s \equiv \text{phase } (s \ \text{gc})$

abbreviation $\text{gc-tmp-ref } s \equiv \text{tmp-ref } (s \ \text{gc})$

abbreviation $\text{gc-ghost-honorary-grey } s \equiv \text{ghost-honorary-grey } (s \ \text{gc})$

abbreviation $\text{gc-ref } s \equiv \text{ref } (s \ \text{gc})$

abbreviation $\text{gc-refs } s \equiv \text{refs } (s \ \text{gc})$

abbreviation $\text{gc-the-ref} \equiv \text{the} \circ \text{gc-ref}$

abbreviation $\text{gc-W } s \equiv \text{W } (s \ \text{gc})$

abbreviation $\text{at-gc} :: \text{location} \Rightarrow ('field, 'mut, 'ref) \text{lsts-pred} \Rightarrow ('field, 'mut, 'ref) \text{gc-pred}$

where

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

abbreviation $\text{atS-gc} :: \text{location set} \Rightarrow ('field, 'mut, 'ref) \text{lsts-pred} \Rightarrow ('field, 'mut, 'ref) \text{gc-pred}$ **where**

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

context *mut-m*

begin

abbreviation

$\text{valid-mut-syn} :: ('field, 'mut, 'ref) \text{gc-loc-comp} \Rightarrow ('field, 'mut, 'ref) \text{gc-pred} \Rightarrow ('field, 'mut, 'ref) \text{gc-com} \Rightarrow ('field, 'mut, 'ref) \text{gc-pred} \Rightarrow \text{bool}$

$(- \models \{\!\{-\}\! / - / \{\!\{-\}\!)$

where

$afts \models \{\!\{P\}\! c \{\!\{Q\}\! \equiv gc\text{-pgms}, \text{mutator } m, afts \models \{\!\{P\}\! c \{\!\{Q\}\!$

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

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

abbreviation $at\text{-mut} :: location \Rightarrow ('field, 'mut, 'ref) lsts\text{-pred} \Rightarrow ('field, 'mut, 'ref) gc\text{-pred}$
where

$at\text{-mut } l P \equiv at (\text{mutator } m) l \text{ imp } LSTP P$

abbreviation $atS\text{-mut} :: location \text{ set} \Rightarrow ('field, 'mut, 'ref) lsts\text{-pred} \Rightarrow ('field, 'mut, 'ref) gc\text{-pred}$ **where**

$atS\text{-mut } ls P \equiv atS (\text{mutator } m) ls \text{ imp } LSTP P$

abbreviation $mut\text{-cas}\text{-mark } s \equiv cas\text{-mark } (s (\text{mutator } m))$

abbreviation $mut\text{-field } s \equiv field (s (\text{mutator } m))$

abbreviation $mut\text{-fM } s \equiv fM (s (\text{mutator } m))$

abbreviation $mut\text{-ghost}\text{-honorary}\text{-grey } s \equiv ghost\text{-honorary}\text{-grey } (s (\text{mutator } m))$

abbreviation $mut\text{-ghost}\text{-handshake}\text{-phase } s \equiv ghost\text{-handshake}\text{-phase } (s (\text{mutator } m))$

abbreviation $mut\text{-ghost}\text{-honorary}\text{-root } s \equiv ghost\text{-honorary}\text{-root } (s (\text{mutator } m))$

abbreviation $mut\text{-mark } s \equiv mark (s (\text{mutator } m))$

abbreviation $mut\text{-new}\text{-ref } s \equiv new\text{-ref } (s (\text{mutator } m))$

abbreviation $mut\text{-phase } s \equiv phase (s (\text{mutator } m))$

abbreviation $mut\text{-ref } s \equiv ref (s (\text{mutator } m))$

abbreviation $mut\text{-tmp}\text{-ref } s \equiv tmp\text{-ref } (s (\text{mutator } m))$

abbreviation $mut\text{-the}\text{-new}\text{-ref} \equiv the \circ mut\text{-new}\text{-ref}$

abbreviation $mut\text{-the}\text{-ref} \equiv the \circ mut\text{-ref}$

abbreviation $mut\text{-refs } s \equiv refs (s (\text{mutator } m))$

abbreviation $mut\text{-roots } s \equiv roots (s (\text{mutator } m))$

abbreviation $mut\text{-W } s \equiv W (s (\text{mutator } m))$

end

context sys

begin

abbreviation

$valid\text{-sys}\text{-syn} :: ('field, 'mut, 'ref) gc\text{-loc}\text{-comp} \Rightarrow ('field, 'mut, 'ref) gc\text{-pred} \Rightarrow ('field, 'mut, 'ref) gc\text{-com} \Rightarrow ('field, 'mut, 'ref) gc\text{-pred} \Rightarrow bool$

$(- \models \{\!\{-\}\! / - / \{\!\{-\}\!)$

where

$afts \models \{\!\{P\}\! c \{\!\{Q\}\! \equiv gc\text{-pgms}, sys, afts \models \{\!\{P\}\! c \{\!\{Q\}\!$

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

$afts \models \{P\} c \equiv afts \models \{P\} c \{P\}$

end

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

abbreviation $sys\text{-}fA s \equiv fA (s sys)$

abbreviation $sys\text{-}fM s \equiv fM (s sys)$

abbreviation $sys\text{-}ghost\text{-}honorary\text{-}grey s \equiv ghost\text{-}honorary\text{-}grey (s sys)$

abbreviation $sys\text{-}ghost\text{-}handshake\text{-}in\text{-}sync m s \equiv ghost\text{-}handshake\text{-}in\text{-}sync (s sys) m$

abbreviation $sys\text{-}ghost\text{-}handshake\text{-}phase s \equiv ghost\text{-}handshake\text{-}phase (s sys)$

abbreviation $sys\text{-}handshake\text{-}pending m s \equiv handshake\text{-}pending (s sys) m$

abbreviation $sys\text{-}handshake\text{-}type s \equiv handshake\text{-}type (s sys)$

abbreviation $sys\text{-}mem\text{-}write\text{-}buffers p s \equiv mem\text{-}write\text{-}buffers (s sys) p$

abbreviation $sys\text{-}mem\text{-}lock s \equiv mem\text{-}lock (s sys)$

abbreviation $sys\text{-}phase s \equiv phase (s sys)$

abbreviation $sys\text{-}W s \equiv W (s sys)$

abbreviation $atS\text{-}sys :: location\ set \Rightarrow ('field, 'mut, 'ref) lsts\text{-}pred \Rightarrow ('field, 'mut, 'ref) gc\text{-}pred$ **where**

$atS\text{-}sys ls P \equiv atS\ sys\ ls\ imp\ LSTP\ P$

Projections on TSO buffers.

abbreviation $(input) tso\text{-}unlocked s \equiv mem\text{-}lock (s sys) = None$

abbreviation $(input) tso\text{-}locked\text{-}by p s \equiv mem\text{-}lock (s sys) = Some\ p$

abbreviation $(input) tso\text{-}pending\ p\ P\ s \equiv filter\ P\ (mem\text{-}write\text{-}buffers\ (s\ sys)\ p)$

abbreviation $(input) tso\text{-}pending\text{-}write\ p\ w\ s \equiv w \in set\ (mem\text{-}write\text{-}buffers\ (s\ sys)\ p)$

abbreviation $(input) tso\text{-}pending\text{-}fA\ p \equiv tso\text{-}pending\ p\ is\text{-}mw\text{-}fA$

abbreviation $(input) tso\text{-}pending\text{-}fM\ p \equiv tso\text{-}pending\ p\ is\text{-}mw\text{-}fM$

abbreviation $(input) tso\text{-}pending\text{-}mark\ p \equiv tso\text{-}pending\ p\ is\text{-}mw\text{-}Mark$

abbreviation $(input) tso\text{-}pending\text{-}mutate\ p \equiv tso\text{-}pending\ p\ is\text{-}mw\text{-}Mutate$

abbreviation $(input) tso\text{-}pending\text{-}phase\ p \equiv tso\text{-}pending\ p\ is\text{-}mw\text{-}Phase$

abbreviation $(input) tso\text{-}no\text{-}pending\text{-}marks \equiv ALLS\ p.\ list\text{-}null\ (tso\text{-}pending\text{-}mark\ p)$

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 $obj\text{-}at :: (('field, 'ref) \text{ object} \Rightarrow bool) \Rightarrow 'ref \Rightarrow ('field, 'mut, 'ref) lsts\text{-}pred$ **where**
 $obj\text{-}at\ P\ r \equiv \lambda s.\ case\ sys\text{-}heap\ s\ r\ of\ None \Rightarrow False \mid Some\ obj \Rightarrow P\ obj$

abbreviation $(input) valid\text{-}ref :: 'ref \Rightarrow ('field, 'mut, 'ref) lsts\text{-}pred$ **where**
 $valid\text{-}ref\ r \equiv obj\text{-}at\ \langle True \rangle\ r$

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

$valid\text{-}null\text{-}ref\ r \equiv case\ r\ of\ None \Rightarrow \langle True \rangle \mid Some\ r' \Rightarrow valid\text{-}ref\ r'$

abbreviation $pred\text{-}points\text{-}to :: 'ref \Rightarrow 'ref \Rightarrow ('field, 'mut, 'ref)\ lsts\text{-}pred\ (\mathbf{infix}\ points'\text{-}to\ 51)\ \mathbf{where}$

$x\ points\text{-}to\ y \equiv \lambda s. obj\text{-}at\ (\lambda obj. y \in ran\ (obj\text{-}fields\ obj))\ x\ s$

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

abbreviation $pred\text{-}reaches :: 'ref \Rightarrow 'ref \Rightarrow ('field, 'mut, 'ref)\ lsts\text{-}pred\ (\mathbf{infix}\ reaches\ 51)\ \mathbf{where}$

$x\ reaches\ y \equiv \lambda s. (\lambda x\ y. (x\ points\text{-}to\ y)\ s)^{**}\ x\ y$

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

definition $obj\text{-}at\text{-}field\text{-}on\text{-}heap :: ('ref \Rightarrow bool) \Rightarrow 'ref \Rightarrow 'field \Rightarrow ('field, 'mut, 'ref)\ lsts\text{-}pred\ \mathbf{where}$

$obj\text{-}at\text{-}field\text{-}on\text{-}heap\ P\ r\ f \equiv \lambda s.$
 $case\ Option.\ map\text{-}option\ obj\text{-}fields\ (sys\text{-}heap\ s\ r)\ of$
 $None \Rightarrow False$
 $\mid Some\ fs \Rightarrow (case\ fs\ f\ of\ None \Rightarrow True$
 $\mid Some\ r' \Rightarrow P\ r')$

3.4 Garbage collector locations.

definition $idle\text{-}locs :: location\ set\ \mathbf{where}$

$idle\text{-}locs \equiv prefixed\ "idle"$

definition $init\text{-}locs :: location\ set\ \mathbf{where}$

$init\text{-}locs \equiv prefixed\ "init"$

definition $mark\text{-}locs :: location\ set\ \mathbf{where}$

$mark\text{-}locs \equiv prefixed\ "mark"$

definition $mark\text{-}loop\text{-}locs :: location\ set\ \mathbf{where}$

$mark\text{-}loop\text{-}locs \equiv prefixed\ "mark\text{-}loop"$

definition $sweep\text{-}locs :: location\ set\ \mathbf{where}$

$sweep\text{-}locs \equiv prefixed\ "sweep"\langle ML \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle$

3.5 Coarse TSO invariants

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

abbreviation $gc\text{-}writes :: ('field, 'ref)\ mem\text{-}write\text{-}action \Rightarrow bool\ \mathbf{where}$

$gc\text{-}writes\ w \equiv case\ w\ of\ mw\text{-}Mark\ - \Rightarrow True \mid mw\text{-}Phase\ - \Rightarrow True \mid mw\text{-}fM\ - \Rightarrow True \mid$
 $mw\text{-}fA\ - \Rightarrow True \mid - \Rightarrow False$

abbreviation $mut\text{-}writes :: ('field, 'ref)\ mem\text{-}write\text{-}action \Rightarrow bool\ \mathbf{where}$

$mut\text{-}writes\ w \equiv case\ w\ of\ mw\text{-}Mutate\ - - - \Rightarrow True \mid mw\text{-}Mark\ - - \Rightarrow True \mid - \Rightarrow False$

definition *tso-writes-inv* :: ('field, 'mut, 'ref) *lsts-pred* **where**

tso-writes-inv \equiv

(*ALLS* *w*. *tso-pending-write* *gc* *w* *imp* \langle *gc-writes* *w* \rangle)

and (*ALLS* *m w*. *tso-pending-write* (*mutator* *m*) *w* *imp* \langle *mut-writes* *w* \rangle) \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle

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 $\equiv \bigcup l \in \{ "mo-co-cmark", "mo-co-ctest", "mo-co-mark", "mo-co-unlock" \}$.
suffixed *l*
 $\langle ML \rangle$

definition (*in* *gc*) *tso-lock-invL* :: ('field, 'mut, 'ref) *gc-pred* **where**

[*inv*]: *tso-lock-invL* \equiv

atS-gc *gc-tso-lock-locs* (*tso-locked-by* *gc*)

and *atS-gc* ($-$ *gc-tso-lock-locs*) (*not tso-locked-by* *gc*) \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle

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

definition *mut-tso-lock-locs* \equiv

$\bigcup l \in \{ "mo-co-cmark", "mo-co-ctest", "mo-co-mark", "mo-co-unlock" \}$. *suffixed* *l*
 $\langle ML \rangle$

definition (*in* *mut-m*) *tso-lock-invL* :: ('field, 'mut, 'ref) *gc-pred* **where**

[*inv*]: *tso-lock-invL* \equiv

atS-mut *mut-tso-lock-locs* (*tso-locked-by* (*mutator* *m*))

and *atS-mut* ($-$ *mut-tso-lock-locs*) (*not tso-locked-by* (*mutator* *m*)) \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle

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 complicated by needing to consider the handshake type due to *get-work*. This relation is very precise.

definition *hp-step-rel* :: (*bool* \times *handshake-type* \times *handshake-phase* \times *handshake-phase*) *set* **where**

hp-step-rel \equiv
 $\{ True \} \times (\{ (ht-NOOP, hp, hp) \mid hp. hp \in \{ hp-Idle, hp-IdleInit, hp-InitMark, hp-Mark \} \}$
 $\}$

$\cup \{ (ht-GetRoots, hp-IdleMarkSweep, hp-IdleMarkSweep)$
 $, (ht-GetWork, hp-IdleMarkSweep, hp-IdleMarkSweep) \}$
 $\cup \{ False \} \times \{ (ht-NOOP, hp-Idle, hp-IdleMarkSweep)$
 $, (ht-NOOP, hp-IdleInit, hp-Idle)$
 $, (ht-NOOP, hp-InitMark, hp-IdleInit)$
 $, (ht-NOOP, hp-Mark, hp-InitMark)$

, (*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 \equiv *ALLS m*.

(*sys-ghost-handshake-in-sync m* \otimes *sys-handshake-type*
 \otimes *sys-ghost-handshake-phase* \otimes *mut-m.mut-ghost-handshake-phase m*) *in* (*hp-step-rel*)
and (*sys-handshake-pending m imp not sys-ghost-handshake-in-sync m*) \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle

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

definition *hp-Idle-locs* \equiv

(*prefixed "idle-noop"* - { *"idle-noop-mfence"*, *"idle-noop-init-type"* })
 \cup { *"idle-read-fM"*, *"idle-invert-fM"*, *"idle-write-fM"*, *"idle-flip-noop-mfence"*, *"idle-flip-noop-init-type"* }
 \langle *ML* \rangle

definition *hp-IdleInit-locs* \equiv

(*prefixed "idle-flip-noop"* - { *"idle-flip-noop-mfence"*, *"idle-flip-noop-init-type"* })
 \cup { *"idle-phase-init"*, *"init-noop-mfence"*, *"init-noop-init-type"* }
 \langle *ML* \rangle

definition *hp-InitMark-locs* \equiv

(*prefixed "init-noop"* - { *"init-noop-mfence"*, *"init-noop-init-type"* })
 \cup { *"init-phase-mark"*, *"mark-read-fM"*, *"mark-write-fA"*, *"mark-noop-mfence"*, *"mark-noop-init-type"* }
 \langle *ML* \rangle

definition *hp-IdleMarkSweep-locs* \equiv

{ *"idle-noop-mfence"*, *"idle-noop-init-type"*, *"mark-end"* }
 \cup *sweep-locs*
 \cup (*mark-loop-locs* - { *"mark-loop-get-roots-init-type"* })
 \langle *ML* \rangle

definition *hp-Mark-locs* \equiv

(*prefixed "mark-noop"* - { *"mark-noop-mfence"*, *"mark-noop-init-type"* })
 \cup { *"mark-loop-get-roots-init-type"* }
 \langle *ML* \rangle

abbreviation

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

definition *hs-noop-locs* \equiv

\bigcup *l* \in *hs-noop-prefixes*. *prefixed l* - (*suffixed "-noop-mfence"* \cup *suffixed "-noop-init-type"*)
 \langle *ML* \rangle

definition *hs-get-roots-locs* \equiv

prefixed "mark-loop-get-roots" - { *"mark-loop-get-roots-init-type"* }

$\langle ML \rangle$

definition $hs\text{-}get\text{-}work\text{-}locs \equiv$

$prefixed\ "mark\text{-}loop\text{-}get\text{-}work" - \{ "mark\text{-}loop\text{-}get\text{-}work\text{-}init\text{-}type" \}$

$\langle ML \rangle$

abbreviation $hs\text{-}prefixes \equiv$

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

definition $hs\text{-}init\text{-}loop\text{-}locs \equiv \bigcup l \in hs\text{-}prefixes. prefixed\ (l\ @\ "-init-loop")$

$\langle ML \rangle$

definition $hs\text{-}done\text{-}loop\text{-}locs \equiv \bigcup l \in hs\text{-}prefixes. prefixed\ (l\ @\ "-done-loop")$

$\langle ML \rangle$

definition $hs\text{-}done\text{-}locs \equiv \bigcup l \in hs\text{-}prefixes. prefixed\ (l\ @\ "-done")$

$\langle ML \rangle$

definition $hs\text{-}none\text{-}pending\text{-}locs \equiv - (hs\text{-}init\text{-}loop\text{-}locs \cup hs\text{-}done\text{-}locs)$

$\langle ML \rangle$

definition $hs\text{-}in\text{-}sync\text{-}locs \equiv$

$(- (\bigcup l \in hs\text{-}prefixes. prefixed\ (l\ @\ "-init") \cup hs\text{-}done\text{-}locs))$

$\cup (\bigcup l \in hs\text{-}prefixes. \{ l\ @\ "-init\text{-}type" \})$

$\langle ML \rangle$

definition $hs\text{-}out\text{-}of\text{-}sync\text{-}locs \equiv$

$\bigcup l \in hs\text{-}prefixes. \{ l\ @\ "-init\text{-}muts" \}$

$\langle ML \rangle$

definition $hs\text{-}mut\text{-}in\text{-}muts\text{-}locs \equiv$

$\bigcup l \in hs\text{-}prefixes. \{ l\ @\ "-init\text{-}loop\text{-}set\text{-}pending", l\ @\ "-init\text{-}loop\text{-}done" \}$

$\langle ML \rangle$

definition $hs\text{-}init\text{-}loop\text{-}done\text{-}locs \equiv$

$\bigcup l \in hs\text{-}prefixes. \{ l\ @\ "-init\text{-}loop\text{-}done" \}$

$\langle ML \rangle$

definition $hs\text{-}init\text{-}loop\text{-}not\text{-}done\text{-}locs \equiv$

$hs\text{-}init\text{-}loop\text{-}locs - (\bigcup l \in hs\text{-}prefixes. \{ l\ @\ "-init\text{-}loop\text{-}done" \})$

$\langle ML \rangle$

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

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

$atS\text{-}gc\ hs\text{-}noop\text{-}locs \quad (sys\text{-}handshake\text{-}type\ eq\ \langle ht\text{-}NOOP \rangle)$

$and\ atS\text{-}gc\ hs\text{-}get\text{-}roots\text{-}locs \quad (sys\text{-}handshake\text{-}type\ eq\ \langle ht\text{-}GetRoots \rangle)$

$and\ atS\text{-}gc\ hs\text{-}get\text{-}work\text{-}locs \quad (sys\text{-}handshake\text{-}type\ eq\ \langle ht\text{-}GetWork \rangle)$

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* \Rightarrow *bool* \Rightarrow *gc-phase* \Rightarrow *bool* **where**

handshake-phase-rel *hp in-sync ph* \equiv
case hp of
hp-Idle \Rightarrow *ph = ph-Idle*
| *hp-IdleInit* \Rightarrow *ph = ph-Idle* \vee (*in-sync* \wedge *ph = ph-Init*)
| *hp-InitMark* \Rightarrow *ph = ph-Init* \vee (*in-sync* \wedge *ph = ph-Mark*)
| *hp-Mark* \Rightarrow *ph = ph-Mark*
| *hp-IdleMarkSweep* \Rightarrow *ph = ph-Mark* \vee (*in-sync* \wedge *ph* \in { *ph-Idle*, *ph-Sweep* })

definition *phase-rel* :: (*bool* \times *handshake-phase* \times *gc-phase* \times *gc-phase* \times (*'field*, *'ref*) *mem-write-action list*) *set* **where**

phase-rel \equiv
{ (*in-sync*, *hp*, *ph*, *ph*, []) | *in-sync hp ph. handshake-phase-rel hp in-sync ph* }
 \cup ({ *True* } \times { (*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 \equiv (*ALLS m. sys-ghost-handshake-in-sync m*) \otimes *sys-ghost-handshake-phase* \otimes *gc-phase* \otimes *sys-phase* \otimes *tso-pending-phase gc in* \langle *phase-rel* \rangle \langle *proof* \rangle \langle *proof* \rangle

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 \equiv
(*idle-locs* - { "*idle-noop-mfence*" })
 \cup (*init-locs* - { "*init-noop-mfence*" })
 \cup (*mark-locs* - { "*mark-read-fM*", "*mark-write-fA*", "*mark-noop-mfence*" })
 \langle *ML* \rangle

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

[*inv*]: *phase-invL* \equiv
atS-gc idle-locs (*gc-phase eq* \langle *ph-Idle* \rangle)
and atS-gc init-locs (*gc-phase eq* \langle *ph-Init* \rangle)
and atS-gc mark-locs (*gc-phase eq* \langle *ph-Mark* \rangle)
and atS-gc sweep-locs (*gc-phase eq* \langle *ph-Sweep* \rangle)
and atS-gc no-pending-phase-locs (*list-null (tso-pending-phase gc)*) \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle

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

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

$$\begin{aligned} fM\text{-rel} = & \\ & \{ (in\text{-sync}, hp, fM, fM, [], l) \mid fM \text{ hp } in\text{-sync } l. hp = hp\text{-Idle} \longrightarrow \neg in\text{-sync} \} \\ & \cup \{ (in\text{-sync}, hp\text{-Idle}, fM, fM', [], l) \mid fM fM' in\text{-sync } l. in\text{-sync} \} \\ & \cup \{ (in\text{-sync}, hp\text{-Idle}, \neg fM, fM, [mw\text{-}fM (\neg fM)], False) \mid fM in\text{-sync}. in\text{-sync} \} \end{aligned}$$

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

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

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

$$\begin{aligned} fA\text{-rel} = & \\ & \{ (in\text{-sync}, hp\text{-Idle}, fA, fM, []) \mid fA fM in\text{-sync}. \neg in\text{-sync} \longrightarrow fA = fM \} \\ & \cup \{ (in\text{-sync}, hp\text{-IdleInit}, fA, \neg fA, []) \mid fA in\text{-sync}. True \} \\ & \cup \{ (in\text{-sync}, hp\text{-InitMark}, fA, \neg fA, [mw\text{-}fA (\neg fA)]) \mid fA in\text{-sync}. in\text{-sync} \} \\ & \cup \{ (in\text{-sync}, hp\text{-InitMark}, fA, fM, []) \mid fA fM in\text{-sync}. \neg in\text{-sync} \longrightarrow fA \neq fM \} \\ & \cup \{ (in\text{-sync}, hp\text{-Mark}, fA, fA, []) \mid fA in\text{-sync}. True \} \\ & \cup \{ (in\text{-sync}, hp\text{-IdleMarkSweep}, fA, fA, []) \mid fA in\text{-sync}. True \} \end{aligned}$$

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

$$fA\text{-rel-inv} \equiv (ALLS \ m. \text{sys-ghost-handshake-in-sync } m) \otimes \text{sys-ghost-handshake-phase} \otimes \text{sys-fA} \otimes \text{gc-fM} \otimes \text{tso-pending-fA } gc \text{ in } \langle fA\text{-rel} \rangle$$

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

$$fM\text{-eq-locs} \equiv (- \{ "idle\text{-write-fM}", "idle\text{-flip-noop-mfence" \}) \langle ML \rangle$$

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

$$fM\text{-tso-empty-locs} \equiv (- \{ "idle\text{-flip-noop-mfence" \}) \langle ML \rangle$$

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

$$fA\text{-tso-empty-locs} \equiv (- \{ "mark\text{-noop-mfence" \}) \langle ML \rangle$$

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

$$\begin{aligned} fA\text{-eq-locs} \equiv & \{ "idle\text{-read-fM}", "idle\text{-invert-fM" \} \\ & \cup \text{prefixed } "idle\text{-noop" \\ & \cup (\text{mark-locs} - \{ "mark\text{-read-fM}", "mark\text{-write-fA}", "mark\text{-noop-mfence" \}) \\ & \cup \text{sweep-locs} \end{aligned} \langle ML \rangle$$

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

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

$\cup \text{init-locs}$
 $\langle ML \rangle$

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

$[inv]: fM\text{-}fA\text{-}invL \equiv$

$atS\text{-}gc\ fM\text{-}eq\text{-}locs \quad (sys\text{-}fM\ eq\ gc\text{-}fM)$
 $and\ at\text{-}gc\ "idle\text{-}write\text{-}fM" \quad (sys\text{-}fM\ neq\ gc\text{-}fM)$
 $and\ at\text{-}gc\ "idle\text{-}flip\text{-}noop\text{-}mfence" \quad (sys\text{-}fM\ neq\ gc\text{-}fM\ imp\ (not\ list\text{-}null\ (tso\text{-}pending\text{-}fM\ gc)))$
 $and\ atS\text{-}gc\ fM\text{-}tso\text{-}empty\text{-}locs \quad (list\text{-}null\ (tso\text{-}pending\text{-}fM\ gc))$

$and\ atS\text{-}gc\ fA\text{-}eq\text{-}locs \quad (sys\text{-}fA\ eq\ gc\text{-}fM)$
 $and\ atS\text{-}gc\ fA\text{-}neq\text{-}locs \quad (sys\text{-}fA\ neq\ gc\text{-}fM)$
 $and\ at\text{-}gc\ "mark\text{-}noop\text{-}mfence" \quad (sys\text{-}fA\ neq\ gc\text{-}fM\ imp\ (not\ list\text{-}null\ (tso\text{-}pending\text{-}fA\ gc)))$
 $and\ atS\text{-}gc\ fA\text{-}tso\text{-}empty\text{-}locs \quad (list\text{-}null\ (tso\text{-}pending\text{-}fA\ gc))$

$\langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle$

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

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

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

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

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

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

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

$grey\ r \equiv EXS\ p. \langle r \rangle\ in\ WL\ p$

definition *black* :: 'ref \Rightarrow ('field, 'mut, 'ref) lsts-pred **where**
black *r* \equiv 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 \Rightarrow 'ref set **where**
write-refs *w* \equiv case *w* of *mw-Mutate* *r f r'* \Rightarrow {*r*} \cup *Option.set-option* *r'* | - \Rightarrow {}

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

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

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

definition *valid-refs-inv* :: ('field, 'mut, 'ref) lsts-pred **where**
valid-refs-inv \equiv *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-greys* are disjoint, and that pending marks are sensible.

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

definition *valid-W-inv* :: ('field, 'mut, 'ref) lsts-pred **where**
valid-W-inv \equiv *ALLS* *p q r.*
(*r in-W* *p* or (*sys-mem-lock neq* \langle *Some* *p* \rangle and *r in-ghost-honorary-grey* *p*) imp marked *r*)
and (\langle *p* \neq *q* \rangle imp not (\langle *r* \rangle in *WL* *p* and \langle *r* \rangle 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 (\langle *fl* \rangle eq *sys-fM*
and *r in-ghost-honorary-grey* *p*
and *tso-locked-by* *p*
and white *r*
and *tso-pending-mark* *p* eq \langle \llbracket *mw-Mark* *r fl* \rrbracket \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 \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle

3.8 Mark Object

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

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

```

locale mark-object =
  fixes p :: 'mut process-name
  fixes l :: location
  fixes p-ph-enabled :: ('field, 'mut, 'ref) lsts-pred
  assumes p-ph-enabled-eq-imp: eq-imp (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 (λs. {[], [mw-Mark (p-the-ref s) (p-fM s)]}))

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

abbreviation (input)
  p-cas-mark-inv ≡ (λs. obj-at (λobj. Some (obj-mark obj) = p-cas-mark s) (p-the-ref s) s)

```

abbreviation (*input*) $p\text{-valid-fM} \equiv p\text{-fM eq sys-fM}$

abbreviation (*input*)

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

abbreviation (*input*)

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

definition $\text{mark-object-invL} :: ('field, 'mut, 'ref) \text{ gc-pred where}$

$\text{mark-object-invL} \equiv$

$\text{at-p } \text{"-mo-null"} \quad \langle \text{True} \rangle$

$\text{and at-p } \text{"-mo-mark"} \quad (p\text{-valid-ref})$

$\text{and at-p } \text{"-mo-fM"} \quad (p\text{-valid-ref and } p\text{-en-cond (} p\text{-mark-inv)})$

$\text{and at-p } \text{"-mo-mtest"} \quad (p\text{-valid-ref and } p\text{-en-cond (} p\text{-mark-inv and } p\text{-valid-fM)})$

$\text{and at-p } \text{"-mo-phase"} \quad (p\text{-valid-ref and } p\text{-mark neq Some } \circ p\text{-fM and } p\text{-en-cond (} p\text{-mark-inv and } p\text{-valid-fM)})$

$\text{and at-p } \text{"-mo-ptest"} \quad (p\text{-valid-ref and } p\text{-mark neq Some } \circ p\text{-fM and } p\text{-en-cond (} p\text{-mark-inv and } p\text{-valid-fM)})$

$\text{and at-p } \text{"-mo-co-lock"} \quad (p\text{-valid-ref and } p\text{-mark-inv and } p\text{-valid-fM and } p\text{-mark neq Some } \circ p\text{-fM and } p\text{-tso-no-pending-mark})$

$\text{and at-p } \text{"-mo-co-cmark"} \quad (p\text{-valid-ref and } p\text{-mark-inv and } p\text{-valid-fM and } p\text{-mark neq Some } \circ p\text{-fM and } p\text{-tso-no-pending-mark})$

$\text{and at-p } \text{"-mo-co-ctest"} \quad (p\text{-valid-ref and } p\text{-mark-inv and } p\text{-valid-fM and } p\text{-mark neq Some } \circ p\text{-fM and } p\text{-cas-mark-inv and } p\text{-tso-no-pending-mark})$

$\text{and at-p } \text{"-mo-co-mark"} \quad (p\text{-cas-mark eq } p\text{-mark and } p\text{-valid-ref and } p\text{-valid-fM and } \text{white } \triangleright p\text{-the-ref and } p\text{-tso-no-pending-mark})$

$\text{and at-p } \text{"-mo-co-unlock"} \quad (p\text{-ghg-inv and } p\text{-valid-ref and } p\text{-valid-fM and } p\text{-valid-W-inv})$

$\text{and at-p } \text{"-mo-co-won"} \quad (p\text{-ghg-inv and } p\text{-valid-ref and } p\text{-valid-fM and } \text{marked } \triangleright p\text{-the-ref and } p\text{-tso-no-pending-mutate})$

$\text{and at-p } \text{"-mo-co-W"} \quad (p\text{-ghg-eq-ref and } p\text{-valid-ref and } p\text{-valid-fM and } \text{marked } \triangleright p\text{-the-ref and } p\text{-tso-no-pending-mutate}) \langle \text{proof} \rangle$

end

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

interpretation $\text{gc-mark: mark-object gc } \text{"mark-loop"} \langle \text{True} \rangle$

$\langle \text{proof} \rangle$

lemmas $\text{gc-mark-mark-object-invL-def2[inv]} = \text{gc-mark.mark-object-invL-def[simplified]}$

interpretation $\text{mut-get-roots: mark-object mutator } m \text{"hs-get-roots-loop"} \langle \text{True} \rangle \text{ for } m$

$\langle \text{proof} \rangle$

lemmas $\text{mut-get-roots-mark-object-invL-def2[inv]} = \text{mut-get-roots.mark-object-invL-def[simplified]}$

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*
 $m \text{ neq } \langle hp\text{-Idle} \rangle$ **for** m
 $\langle \text{proof} \rangle$

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

interpretation *mut-store-ins*: mark-object mutator m "store-ins" *mut-m.mut-ghost-handshake-phase*
 $m \text{ neq } \langle hp\text{-Idle} \rangle$ **for** m
 $\langle \text{proof} \rangle$

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

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

definition *mut-hs-get-roots-loop-locs* \equiv
 $\text{prefixed } "hs\text{-get-roots-loop}"$
 $\langle ML \rangle$

definition *mut-hs-get-roots-loop-mo-locs* \equiv
 $\text{prefixed } "hs\text{-get-roots-loop-mo}" \cup \{ "hs\text{-get-roots-loop-done}" \}$
 $\langle ML \rangle$

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

definition *mut-hs-not-hp-Idle-locs* \equiv
 $\bigcup \text{pref} \in \text{mut-async-mark-object-prefixes}.$
 $\bigcup l \in \{ "mo\text{-co-lock}", "mo\text{-co-cmark}", "mo\text{-co-ctest}", "mo\text{-co-mark}", "mo\text{-co-unlock}", "mo\text{-co-won}",$
 $"mo\text{-co-W}" \}. \{ \text{pref } @ \text{"-"} @ l \}$
 $\langle ML \rangle$

definition *mut-async-mo-ptest-locs* \equiv
 $\bigcup \text{pref} \in \text{mut-async-mark-object-prefixes}. \{ \text{pref } @ \text{"-mo-ptest"} \}$
 $\langle ML \rangle$

definition *mut-mo-ptest-locs* \equiv
 $\bigcup \text{pref} \in \text{mut-async-mark-object-prefixes}. \{ \text{pref } @ \text{"-mo-ptest"} \}$
 $\langle ML \rangle$

definition *mut-mo-valid-ref-locs* \equiv
 $\text{prefixed } "store-del" \cup \text{prefixed } "store-ins" \cup \{ "deref-del", "lop-store-ins" \}$
 $\langle ML \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle$

This local invariant for the mutators illustrates the handshake structure: we can rely on the insertion barrier earlier than on the deletion barrier. Both need to be installed before *get-roots* to ensure we preserve the strong tricolour invariant. All black objects at that point are allocated: we need to know that the insertion barrier is installed to preserve it. This limits when fA can be set.

It is interesting to contrast the two barriers. Intuitively a mutator can locally guarantee that it, in the relevant phases, will insert only marked references. Less often can it be sure that the reference it is overwriting is marked. We also need to consider writes pending in TSO buffers.

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

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

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

[*inv*]: *mark-object-invL* \equiv

atS-mut mut-hs-get-roots-loop-locs (*mut-refs* *subsetq* *mut-roots* and (*ALLS* *r*. $\langle r \rangle$ 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* \triangleright *mut-the-ref*)

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

and *at-mut "hs-get-roots-done"* (*ALLS* *r*. $\langle r \rangle$ 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* $\langle \text{ph-Idle} \rangle$ *imp* *mut-ghost-handshake-phase* *neq* $\langle \text{hp-Idle} \rangle$)

and *atS-mut mut-hs-not-hp-Idle-locs* (*mut-ghost-handshake-phase* *neq* $\langle \text{hp-Idle} \rangle$)

and *atS-mut mut-mo-ptest-locs* (*mut-phase* *eq* $\langle \text{ph-Idle} \rangle$ *imp* (*mut-ghost-handshake-phase* in $\{ \langle \text{hp-Idle} \rangle, \langle \text{hp-IdleInit} \rangle \}$)

$\langle \text{hp-IdleMarkSweep} \rangle$

or (*mut-ghost-handshake-phase* *eq*

and *sys-phase* *eq* $\langle \text{ph-Idle} \rangle$))

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 $\{ \langle \text{hp-InitMark} \rangle, \langle \text{hp-Mark} \rangle \}$)

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

and *sys-phase* *neq* $\langle \text{ph-Idle} \rangle$))

and *not null* *mut-new-ref*

imp *marked* \triangleright *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 (λs. ∀ opt-r'. ¬tso-pending-write (mutator m) (mw-Mutate
(mut-tmp-ref s) (mut-field s) opt-r') s)
imp (λs. obj-at-field-on-heap (λr. mut-ref s = Some r ∨
marked r s) (mut-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 (λs. ∀ opt-r'. ¬tso-pending-write (mutator m) (mw-Mutate
(mut-tmp-ref s) (mut-field s) opt-r') s)
imp (λs. obj-at-field-on-heap (λr'. marked r' s) (mut-tmp-ref s)
(mut-field s) s))⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩

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

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

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

obj-fields-marked-inv ≡

ALLS f. ⟨f⟩ in (− gc-field-set) imp (λs. obj-at-field-on-heap (λr. marked r s) (gc-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" })

(λs. obj-at-field-on-heap (λr. 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 (λs. ((gc-the-ref s) reaches y) s) imp valid-ref y)

and at-gc "mark-loop-fields" (gc-tmp-ref in gc-W)

and at-gc "mark-loop-mark-field-done" (not null gc-ref imp marked \triangleright 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⟩⟨proof⟩

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 \Rightarrow 'ref \Rightarrow ('field, 'mut, 'ref) lsts-pred (**infix** *points'-to'-white* 51) **where**

x points-to-white y \equiv *x points-to y* and *white y*

definition *strong-tricolour-inv* :: ('field, 'mut, 'ref) lsts-pred **where**

strong-tricolour-inv \equiv 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 \Rightarrow 'ref \Rightarrow ('field, 'mut, 'ref) lsts-pred (**infix** *has'-white'-path'-to* 51) **where**

x has-white-path-to y \equiv $\lambda s. (\lambda x y. (x \text{ points-to-white } y) s)^{**} x y$

definition *grey-protects-white* :: 'ref \Rightarrow 'ref \Rightarrow ('field, 'mut, 'ref) lsts-pred (**infix** *grey'-protects'-white* 51) **where**

g grey-protects-white w \equiv *grey g* and *g has-white-path-to w*

definition *weak-tricolour-inv* :: ('field, 'mut, 'ref) lsts-pred **where**

weak-tricolour-inv \equiv

ALLS *b w. black b* and *b points-to-white w imp (EXS g. g grey-protects-white w)*

lemma *strong-tricolour-inv s* \implies *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 \Rightarrow ('field, 'mut, 'ref) lsts-pred **where**

in-snapshot r \equiv *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 \equiv 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 *mutators-phase-inv* :: ('field, 'mut, 'ref) *lst-pred* **where**
mutators-phase-inv \equiv *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* \Rightarrow ('field, 'mut, 'ref) *lst-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* = \langle *True* \rangle
| *sys-phase-inv-aux hp-IdleMarkSweep* = ((*sys-phase* eq \langle *ph-Idle* \rangle) or *tso-pending-write gc*
(*mw-Phase ph-Idle*) imp *no-grey-refs*)

abbreviation *sys-phase-inv* :: ('field, 'mut, 'ref) *lst-pred* **where**
sys-phase-inv s \equiv *sys-phase-inv-aux (sys-ghost-handshake-phase s) s* \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle

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 \equiv
– (*prefixed "store-del"* \cup {*"lop-store-ins"*} \cup *prefixed "store-ins"*)
 \langle *ML* \rangle

definition (in *mut-m*) *load-invL* :: ('field, 'mut, 'ref) *gc-pred* **where**
 \langle *inv* \rangle : *load-invL* \equiv
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)* \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle \langle *proof* \rangle

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) *lst-pred* **where**
gc-W-empty-mut-inv \equiv
(*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 \equiv
 $idle-locs \cup init-locs \cup sweep-locs \cup \{ "mark-read-fM", "mark-write-fA", "mark-end" \}$
 $\cup prefixed "mark-noop"$
 $\cup prefixed "mark-loop-get-roots"$
 $\cup prefixed "mark-loop-get-work"$
 $\langle ML \rangle$

definition *black-heap-locs* $\equiv \{ "sweep-idle", "idle-noop-mfence", "idle-noop-init-type" \}$
 $\langle ML \rangle$

definition *no-grey-refs-locs* $\equiv black-heap-locs \cup sweep-locs \cup \{ "mark-end" \}$
 $\langle ML \rangle$

definition (in *gc*) *gc-W-empty-invL* :: ('field, 'mut, 'ref) *gc-pred* **where**

[*inv*]: *gc-W-empty-invL* \equiv
 $atS-gc (hs-get-roots-locs \cup hs-get-work-locs) (ALLS m. mut-m.gc-W-empty-mut-inv m)$
 $and at-gc "mark-loop-get-roots-load-W" (empty sys-W imp no-grey-refs)$
 $and at-gc "mark-loop-get-work-load-W" (empty sys-W imp no-grey-refs)$
 $and at-gc "mark-loop" (empty gc-W imp no-grey-refs)$
 $and atS-gc no-grey-refs-locs no-grey-refs$
 $and atS-gc gc-W-empty-locs (empty gc-W) \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle$

3.13 Sweep loop invariants

definition *sweep-loop-locs* $\equiv prefixed "sweep-loop"$
 $\langle ML \rangle$

definition (in *gc*) *sweep-loop-invL* :: ('field, 'mut, 'ref) *gc-pred* **where**

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

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 $s \equiv$
 $fM\ s = initial-mark$
 $\wedge\ phase\ s = ph-Idle$
 $\wedge\ ghost-honorary-grey\ s = \{\}$
 $\wedge\ W\ s = \{\}$

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

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

abbreviation

root-reachable $y \equiv EXS\ m\ x. \langle x \rangle\ in\ mut-m.mut-roots\ m\ and\ x\ reaches\ y$

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

definition *gc-system-init* :: (*'field*, *'mut*, *'ref*) *lsts-pred* **where**
gc-system-init \equiv

$(\lambda s. gc\text{-initial-state } (s \text{ gc}))$
and $(\lambda s. \forall m. mut\text{-initial-state } (s \text{ (mutator } m)))$
and $(\lambda s. sys\text{-initial-state } (s \text{ sys}))$
and *valid-refs*

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

abbreviation *gc-system* :: (*'field*, *'mut*, *'ref*) *gc-system* **where**

gc-system $\equiv (gc\text{-pgms}, gc\text{-system-init})\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle$

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

Our headline safety result follows directly.

corollary *safety*:

$s \in reachable\text{-states } gc\text{-system} \implies valid\text{-refs } (mkP \ s)\downarrow\langle proof \rangle$

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-common-init-state $\equiv undefined(\ | \ ghost\text{-handshake-phase} := hp\text{-IdleMarkSweep}, \ ghost\text{-honorary-grey} := \{\}, \ ghost\text{-honorary-root} := \{\}, \ roots := \{\}, \ W := \{\} \ |)$

context *gc-system*

begin

abbreviation *sys-init-heap* :: *ref* \Rightarrow (*field*, *ref*) *object option* **where**

sys-init-heap \equiv

$[\ 0 \mapsto (\ | \ obj\text{-mark} = initial\text{-mark},$

```

    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*

, $fA := \text{initial-mark}$
 , $\text{phase} := \text{ph-Idle}$
 , $W := \{\}$
 , $\text{ghost-honorary-grey} := \{\} \mid$

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

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

abbreviation $\text{init-state} :: \text{clsts}$ **where**
 $\text{init-state} \equiv \lambda p. \text{case } p \text{ of}$
 $\quad gc \Rightarrow gc\text{-init-state}$
 $\quad \mid \text{sys} \Rightarrow \text{sys-init-state}$
 $\quad \mid \text{mutator } m \Rightarrow \text{lookup } \text{mut-init-states } \text{mut-common-init-state } m$

lemma
 $gc\text{-system-init } \text{init-state} \langle \text{proof} \rangle$

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.