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

Peter Gammie, Tony Hosking and Kai Engelhardt

March 17, 2025

Abstract

We model an instance of Schism, a state-of-the-art real-time garbage collection scheme for weak memory, and show that it is safe on x86-TSO.

Contents

1 Introduction

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

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

We present our model of the garbage collector in §2, the predicates we use in our assertions in §3, the detailed invariants in §4 and §5, and the high-level safety results in §17. A concrete system state that satisfies our invariants is exhibited in §18. The other sections contain the often gnarly proofs and lemmas starring in supporting roles.

The modelling language CIMP used in this development is described in the AFP entry ConcurrentIMP (Gammie 2015).

2 A model of a Schism garbage collector

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

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

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

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

type-synonym *location* = *string*

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 *hs-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 *hs-phase*
  = *hp-Idle* — done 1 noop
  | *hp-IdleInit*
  | *hp-InitMark*
  | *hp-Mark* — done 4 noops
  | *hp-IdleMarkSweep* — done get roots

definition
  hs-step :: *hs-phase* ⇒ *hs-phase*
where
  hs-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 two partial maps. Firstly the types:

- *′field* is the abstract type of fields.

- *'ref* is the abstract type of object references.
- *'mut* is the abstract type of the mutators' names.

  The maps:

- *obj-fields* maps *'field*s to object references (or *None* signifying `NULL` or type error).
- *obj-payload* maps a *'field* to non-reference data. For convenience we similarly allow that to be `NULL`.

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

record (*'field*, *'payload*, *'ref*) *object* =
  obj-mark :: *gc-mark*
  obj-fields :: *'field* ⇀ *'ref*
  obj-payload :: *'field* ⇀ *'payload*

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

datatype (*'field*, *'payload*, *'ref*) *mem-store-action*
  = *mw-Mark 'ref gc-mark*
  | *mw-Mutate 'ref 'field 'ref option*
  | *mw-Mutate-Payload 'ref 'field 'payload option*
  | *mw-fA gc-mark*
  | *mw-fM gc-mark*
  | *mw-Phase gc-phase*

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

datatype (*'field*, *'ref*) *mem-load-action*
  = *mr-Ref 'ref 'field*
  | *mr-Payload 'ref 'field*
  | *mr-Mark 'ref*
  | *mr-Phase*
  | *mr-fM*
  | *mr-fA*

datatype (*'field*, *'mut*, *'payload*, *'ref*) *request-op*
  = *ro-MFENCE*
  | *ro-Load* (*'field*, *'ref*) *mem-load-action*
  | *ro-Store* (*'field*, *'payload*, *'ref*) *mem-store-action*
  | *ro-Lock*
  | *ro-Unlock*
  | *ro-Alloc*
  | *ro-Free 'ref*
  | *ro-hs-gc-load-pending 'mut*
  | *ro-hs-gc-store-type hs-type*
  | *ro-hs-gc-store-pending 'mut*
  | *ro-hs-gc-load-W*
  | *ro-hs-mut-load-pending*
  | *ro-hs-mut-load-type*
  | *ro-hs-mut-done 'ref set*

abbreviation *LoadfM* ≡ *ro-Load mr-fM*
abbreviation *LoadMark r* ≡ *ro-Load* (*mr-Mark r*)
abbreviation *LoadPayload r f* ≡ *ro-Load* (*mr-Payload r f*)
abbreviation *LoadPhase* ≡ *ro-Load mr-Phase*
abbreviation *LoadRef r f* ≡ *ro-Load* (*mr-Ref r f*)

abbreviation *StorefA m* ≡ *ro-Store* (*mw-fA m*)
abbreviation *StorefM m* ≡ *ro-Store* (*mw-fM m*)

abbreviation *StoreMark r m* ≡ *ro-Store* (*mw-Mark r m*)
abbreviation *StorePayload r f pl* ≡ *ro-Store* (*mw-Mutate-Payload r f pl*)
abbreviation *StorePhase ph* ≡ *ro-Store* (*mw-Phase ph*)
abbreviation *StoreRef r f r′* ≡ *ro-Store* (*mw-Mutate r f r′*)

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

datatype (*′field*, *′payload*, *′ref*) *response*
  = *mv-Bool bool*
  | *mv-Mark gc-mark option*
  | *mv-Payload ′payload option* — the requested reference might be invalid
  | *mv-Phase gc-phase*
  | *mv-Ref ′ref option*
  | *mv-Refs ′ref set*
  | *mv-Void*
  | *mv-hs-type hs-type*

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*, *′payload*, *′ref*) *local-state* =
  — System-specific fields
  heap :: *′ref* ⇀ (*′field*, *′payload*, *′ref*) *object*
  — TSO memory state
  mem-store-buffers :: *′mut process-name* ⇒ (*′field*, *′payload*, *′ref*) *mem-store-action list*
  mem-lock :: *′mut process-name option*
  — Handshake state
  hs-pending :: *′mut* ⇒ *bool*
  — Ghost state
  ghost-hs-in-sync :: *′mut* ⇒ *bool*
  ghost-hs-phase :: *hs-phase*

  — Mutator-specific temporaries
  new-ref :: *′ref option*
  roots :: *′ref set*
  ghost-honorary-root :: *′ref set*
  payload-value :: *′payload option*
  mutator-data :: *′field* ⇀ *′payload*
  mutator-hs-pending :: *bool*

  — 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*
  — Handshake state

hs-type :: *hs-type*
— Ghost state
ghost-honorary-grey :: *'ref set*

We instantiate CIMP's types as follows:

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

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

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

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

type-synonym (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred*
= (*'field*, *'mut*, *'payload*, *'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 typically use their contents by prefixing identifiers with 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*]: $m \neq m'$
locale *gc*
locale *sys*

2.1 Object marking

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

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

abbreviation *lock-syn* :: *location* ⇒ (*'field*, *'mut*, *'payload*, *'ref*) *gc-com* **where**
lock-syn l ≡ ⦃*l*⦄ *Request* (λ*s*. (*p*, *ro-Lock*)) (λ- *s*. {*s*})
notation *lock-syn* (‹⦃-⦄ *lock*›)

abbreviation *unlock-syn* :: *location* ⇒ (*'field*, *'mut*, *'payload*, *'ref*) *gc-com* **where**

unlock-syn l ≡ ⦃*l*⦄ *Request* (*λs.* (*p*, *ro-Unlock*)) (*λ- s.* {*s*})
notation *unlock-syn* (‹⦃-⦄ *unlock*›)

abbreviation
load-mark-syn :: *location* ⇒ ((′*field*, ′*mut*, ′*payload*, ′*ref*) *local-state* ⇒ ′*ref*)
⇒ ((*gc-mark option* ⇒ *gc-mark option*)
⇒ (′*field*, ′*mut*, ′*payload*, ′*ref*) *local-state*
⇒ (′*field*, ′*mut*, ′*payload*, ′*ref*) *local-state*) ⇒ (′*field*, ′*mut*, ′*payload*, ′*ref*) *gc-com*
where
load-mark-syn l r upd ≡ ⦃*l*⦄ *Request* (*λs.* (*p*, *LoadMark* (*r s*))) (*λmv s.* { *upd* ⟨*m*⟩ *s* |*m. mv* = *mv-Mark m* })
notation *load-mark-syn* (‹⦃-⦄ *load*′*-mark*›)

abbreviation *load-fM-syn* :: *location* ⇒ (′*field*, ′*mut*, ′*payload*, ′*ref*) *gc-com* **where**
load-fM-syn l ≡ ⦃*l*⦄ *Request* (*λs.* (*p*, *ro-Load mr-fM*)) (*λmv s.* { *s*⦇*fM* := *m*⦈ |*m. mv* = *mv-Mark* (*Some m*) })
notation *load-fM-syn* (‹⦃-⦄ *load*′*-fM*›)

abbreviation
load-phase :: *location* ⇒ (′*field*, ′*mut*, ′*payload*, ′*ref*) *gc-com*
where
load-phase l ≡ ⦃*l*⦄ *Request* (*λs.* (*p*, *LoadPhase*)) (*λmv s.* { *s*⦇*phase* := *ph*⦈ |*ph. mv* = *mv-Phase ph* })
notation *load-phase* (‹⦃-⦄ *load*′*-phase*›)

abbreviation
store-mark-syn :: *location* ⇒ ((′*field*, ′*mut*, ′*payload*, ′*ref*) *local-state* ⇒ ′*ref*) ⇒ ((′*field*, ′*mut*, ′*payload*, ′*ref*) *local-state* ⇒ *bool*) ⇒ (′*field*, ′*mut*, ′*payload*, ′*ref*) *gc-com*
where
store-mark-syn l r fl ≡ ⦃*l*⦄ *Request* (*λs.* (*p*, *StoreMark* (*r s*) (*fl s*))) (*λ- s.* { *s*⦇ *ghost-honorary-grey* := {*r s*} ⦈ })
notation *store-mark-syn* (‹⦃-⦄ *store*′*-mark*›)

abbreviation
add-to-W-syn :: *location* ⇒ ((′*field*, ′*mut*, ′*payload*, ′*ref*) *local-state* ⇒ ′*ref*) ⇒ (′*field*, ′*mut*, ′*payload*, ′*ref*) *gc-com*
where
add-to-W-syn l r ≡ ⦃*l*⦄ ⌊*λs. s*⦇ *W* := *W s* ∪ {*r s*}, *ghost-honorary-grey* := {} ⦈⌋
notation *add-to-W-syn* (‹⦃-⦄ *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 store pending in a store buffer; in other words, we cannot have objects atomically transition from white to grey. The following scheme blackens a white object, and then reverts it to grey. The *ghost-honorary-grey* variable is used to track objects undergoing this transition.

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

definition
mark-object-fn :: *location* ⇒ (′*field*, ′*mut*, ′*payload*, ′*ref*) *gc-com*
where
mark-object-fn l =
⦃*l* @ ′′*-mo-null*′′⦄ *IF* ¬ (*NULL ref*) *THEN*
⦃*l* @ ′′*-mo-mark*′′⦄ *load-mark* (*the* ∘ *ref*) *mark-update* ;;
⦃*l* @ ′′*-mo-fM*′′⦄ *load-fM* ;;
⦃*l* @ ′′*-mo-mtest*′′⦄ *IF mark* ≠ *Some* ∘ *fM THEN*
⦃*l* @ ′′*-mo-phase*′′⦄ *load-phase* ;;
⦃*l* @ ′′*-mo-ptest*′′⦄ *IF phase* ≠ ⟨*ph-Idle*⟩ *THEN*
— CAS: claim object
⦃*l* @ ′′*-mo-co-lock*′′⦄ *lock* ;;
⦃*l* @ ′′*-mo-co-cmark*′′⦄ *load-mark* (*the* ∘ *ref*) *cas-mark-update* ;;
⦃*l* @ ′′*-mo-co-ctest*′′⦄ *IF cas-mark* = *mark THEN*
⦃*l* @ ′′*-mo-co-mark*′′⦄ *store-mark* (*the* ∘ *ref*) *fM*
FI ;;

$\{\!|l\ @\ ''\text{-}mo\text{-}co\text{-}unlock''|\!\}\ unlock\ ;;$
$\{\!|l\ @\ ''\text{-}mo\text{-}co\text{-}won''|\!\}\ IF\ cas\text{-}mark = mark\ THEN$
$\{\!|l\ @\ ''\text{-}mo\text{-}co\text{-}W''|\!\}\ add\text{-}to\text{-}W\ (the \circ ref)$
FI
FI
FI
FI

end

The worklists (field W) are not subject to TSO. As we later show (§4.4), 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. Here we do so by having the GC busy-wait: 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. We carefully model the effect these handshakes have on the processes' TSO buffers.

The system and mutators track handshake phases using ghost state; see §4.3.

The handshake type and handshake pending bit are not subject to TSO as we expect a realistic implementation of handshakes would involve synchronisation.

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

context *sys*
begin

definition
handshake :: ($'field$, $'mut$, $'payload$, $'ref$) *gc-com*
where
handshake =
$\{\!|''sys\text{-}hs\text{-}gc\text{-}set\text{-}type''|\!\}\ Response$
$(\lambda req\ s.\ \{\ (s(\!|\ hs\text{-}type := ht,$
$ghost\text{-}hs\text{-}in\text{-}sync := \langle False\rangle,$
$ghost\text{-}hs\text{-}phase := hp\text{-}step\ ht\ (ghost\text{-}hs\text{-}phase\ s)\ |\!),$
$mv\text{-}Void)$
$|ht.\ req = (gc,\ ro\text{-}hs\text{-}gc\text{-}store\text{-}type\ ht)\ \})$
$\oplus \{\!|''sys\text{-}hs\text{-}gc\text{-}mut\text{-}reqs''|\!\}\ Response$
$(\lambda req\ s.\ \{\ (s(\!|\ hs\text{-}pending := (hs\text{-}pending\ s)(m := True)\ |\!),\ mv\text{-}Void)$
$|m.\ req = (gc,\ ro\text{-}hs\text{-}gc\text{-}store\text{-}pending\ m)\ \})$
$\oplus \{\!|''sys\text{-}hs\text{-}gc\text{-}done''|\!\}\ Response$
$(\lambda req\ s.\ \{\ (s,\ mv\text{-}Bool\ (\neg hs\text{-}pending\ s\ m))$
$|m.\ req = (gc,\ ro\text{-}hs\text{-}gc\text{-}load\text{-}pending\ m)\ \})$
$\oplus \{\!|''sys\text{-}hs\text{-}gc\text{-}load\text{-}W''|\!\}\ Response$
$(\lambda req\ s.\ \{\ (s(\!|\ W := \{\}\ |\!),\ mv\text{-}Refs\ (W\ s))$
$|\text{-}::unit.\ req = (gc,\ ro\text{-}hs\text{-}gc\text{-}load\text{-}W)\ \})$
$\oplus \{\!|''sys\text{-}hs\text{-}mut\text{-}pending''|\!\}\ Response$
$(\lambda req\ s.\ \{\ (s,\ mv\text{-}Bool\ (hs\text{-}pending\ s\ m))$
$|m.\ req = (mutator\ m,\ ro\text{-}hs\text{-}mut\text{-}load\text{-}pending)\ \})$
$\oplus \{\!|''sys\text{-}hs\text{-}mut''|\!\}\ Response$

(λreq s. { (s, mv-hs-type (hs-type s))
|m. req = (mutator m, ro-hs-mut-load-type) })
⊕ ⦃''sys-hs-mut-done''⦄ Response
(λreq s. { (s⦇ hs-pending := (hs-pending s)(m := False),
W := W s ∪ W',
ghost-hs-in-sync := (ghost-hs-in-sync s)(m := True) ⦈,
mv-Void)
|m W'. req = (mutator m, ro-hs-mut-done W') })

end

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

Again we could make these subject to TSO, but that would be over specification.

context *mut-m*
begin

abbreviation *mark-object-syn* :: *location* ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *gc-com* (‹⦃-⦄ *mark'-object*› [0] 71) **where**
⦃l⦄ *mark-object* ≡ *mark-object-fn* (*mutator m*) *l*

abbreviation *mfence-syn* :: *location* ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *gc-com* (‹⦃-⦄ *MFENCE*› [0] 71) **where**
⦃l⦄ *MFENCE* ≡ ⦃l⦄ *Request* (λs. (*mutator m*, *ro-MFENCE*)) (λ- s. {s})

abbreviation *hs-load-pending-syn* :: *location* ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *gc-com* (‹⦃-⦄ *hs'-load'-pending'-*› [0] 71) **where**
⦃l⦄ *hs-load-pending-* ≡ ⦃l⦄ *Request* (λs. (*mutator m*, *ro-hs-mut-load-pending*)) (λmv s. { s⦇ *mutator-hs-pending* := b ⦈ |b. mv = *mv-Bool* b })

abbreviation *hs-load-type-syn* :: *location* ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *gc-com* (‹⦃-⦄ *hs'-load'-type*› [0] 71) **where**
⦃l⦄ *hs-load-type* ≡ ⦃l⦄ *Request* (λs. (*mutator m*, *ro-hs-mut-load-type*)) (λmv s. { s⦇ *hs-type* := ht ⦈ |ht. mv = *mv-hs-type* ht})

abbreviation *hs-noop-done-syn* :: *location* ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *gc-com* (‹⦃-⦄ *hs'-noop'-done'-*›) **where**
⦃l⦄ *hs-noop-done-* ≡ ⦃l⦄ *Request* (λs. (*mutator m*, *ro-hs-mut-done* {}))
(λ- s. {s⦇ *ghost-hs-phase* := *hs-step* (*ghost-hs-phase* s) ⦈})

abbreviation *hs-get-roots-done-syn* :: *location* ⇒ (('*field*, '*mut*, '*payload*, '*ref*) *local-state* ⇒ '*ref set*) ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *gc-com* (‹⦃-⦄ *hs'-get'-roots'-done'-*›) **where**
⦃l⦄ *hs-get-roots-done-* wl ≡ ⦃l⦄ *Request* (λs. (*mutator m*, *ro-hs-mut-done* (wl s)))
(λ- s. {s⦇ W := {}, *ghost-hs-phase* := *hs-step* (*ghost-hs-phase* s) ⦈})

abbreviation *hs-get-work-done-syn* :: *location* ⇒ (('*field*, '*mut*, '*payload*, '*ref*) *local-state* ⇒ '*ref set*) ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *gc-com* (‹⦃-⦄ *hs'-get'-work'-done*›) **where**
⦃l⦄ *hs-get-work-done* wl ≡ ⦃l⦄ *Request* (λs. (*mutator m*, *ro-hs-mut-done* (wl s)))
(λ- s. {s⦇ W := {} ⦈})

definition
handshake :: ('*field*, '*mut*, '*payload*, '*ref*) *gc-com*
where
handshake =
⦃''hs-load-pending''⦄ *hs-load-pending-* ;;
⦃''hs-pending''⦄ *IF mutator-hs-pending*
THEN
⦃''hs-mfence''⦄ *MFENCE* ;;
⦃''hs-load-ht''⦄ *hs-load-type* ;;
⦃''hs-noop''⦄ *IF hs-type* = ⟨*ht-NOOP*⟩

THEN
  ⦃''hs-noop-done''⦄ *hs-noop-done-*
ELSE ⦃''hs-get-roots''⦄ *IF hs-type* = ⟨*ht-GetRoots*⟩
THEN
  ⦃''hs-get-roots-refs''⦄ ´*refs* := ´*roots* ;;
  ⦃''hs-get-roots-loop''⦄ *WHILE* ¬*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*
ELSE ⦃''hs-get-work''⦄ *IF hs-type* = ⟨*ht-GetWork*⟩
THEN
  ⦃''hs-get-work-done''⦄ *hs-get-work-done W*
FI FI FI
FI

end

The garbage collector's side of the interface.

context *gc*
begin

abbreviation *set-hs-type* :: *location* ⇒ *hs-type* ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *gc-com* (‹⦃-⦄ *set*'-*hs*'-*type*›) **where**
⦃*l*⦄ *set-hs-type ht* ≡ ⦃*l*⦄ *Request* (λ*s*. (*gc*, *ro-hs-gc-store-type ht*)) (λ- *s*. {*s*})

abbreviation *set-hs-pending* :: *location* ⇒ (('*field*, '*mut*, '*payload*, '*ref*) *local-state* ⇒ '*mut*) ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *gc-com* (‹⦃-⦄ *set*'-*hs*'-*pending*›) **where**
⦃*l*⦄ *set-hs-pending m* ≡ ⦃*l*⦄ *Request* (λ*s*. (*gc*, *ro-hs-gc-store-pending* (*m s*))) (λ- *s*. {*s*})

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

abbreviation *mfence* :: *location* ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *gc-com* (‹⦃-⦄ *MFENCE*›) **where**
⦃*l*⦄ *MFENCE* ≡ ⦃*l*⦄ *Request* (λ*s*. (*gc*, *ro-MFENCE*)) (λ- *s*. {*s*})

definition
handshake-init :: *location* ⇒ *hs-type* ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *gc-com* (‹⦃-⦄ *handshake*'-*init*›)
where
⦃*l*⦄ *handshake-init req* =
  ⦃*l* @ ''-*init-type*''⦄ *set-hs-type req* ;;
  ⦃*l* @ ''-*init-muts*''⦄ ´*muts* := *UNIV* ;;
  ⦃*l* @ ''-*init-loop*''⦄ *WHILE* ¬ (*EMPTY muts*) *DO*
    ⦃*l* @ ''-*init-loop-choose-mut*''⦄ ´*mut* :∈ ´*muts* ;;
    ⦃*l* @ ''-*init-loop-set-pending*''⦄ *set-hs-pending mut* ;;
    ⦃*l* @ ''-*init-loop-done*''⦄ ´*muts* := (´*muts* − {´*mut*})
  OD

definition
handshake-done :: *location* ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *gc-com* (‹⦃-⦄ *handshake*'-*done*›)
where
⦃*l*⦄ *handshake-done* =
  ⦃*l* @ ''-*done-muts*''⦄ ´*muts* := *UNIV* ;;
  ⦃*l* @ ''-*done-loop*''⦄ *WHILE* ¬*EMPTY muts DO*
    ⦃*l* @ ''-*done-loop-choose-mut*''⦄ ´*mut* :∈ ´*muts* ;;
    ⦃*l* @ ''-*done-loop-rendezvous*''⦄ *Request*
      (λ*s*. (*gc*, *ro-hs-gc-load-pending* (*mut s*)))

$(\lambda mv\ s.\ \{\ s(\!|\ muts := muts\ s - \{\ mut\ s\ |done.\ mv = mv\text{-}Bool\ done \wedge done\ \}\ |\!)\})$
OD

definition
$handshake\text{-}noop :: location \Rightarrow ('field, 'mut, 'payload, 'ref)\ gc\text{-}com$ (‹{|-|} handshake′-noop›)
where
$\{|l|\}\ handshake\text{-}noop =$
$\{|l\ @\ ''\text{-}mfence''|\}\ MFENCE$;;
$\{|l|\}\ handshake\text{-}init\ ht\text{-}NOOP$;;
$\{|l|\}\ handshake\text{-}done$

definition
$handshake\text{-}get\text{-}roots :: location \Rightarrow ('field, 'mut, 'payload, 'ref)\ gc\text{-}com$ (‹{|-|} handshake′-get′-roots›)
where
$\{|l|\}\ handshake\text{-}get\text{-}roots =$
$\{|l|\}\ handshake\text{-}init\ ht\text{-}GetRoots$;;
$\{|l|\}\ handshake\text{-}done$;;
$\{|l|\}\ load\text{-}W$

definition
$handshake\text{-}get\text{-}work :: location \Rightarrow ('field, 'mut, 'payload, 'ref)\ gc\text{-}com$ (‹{|-|} handshake′-get′-work›)
where
$\{|l|\}\ handshake\text{-}get\text{-}work =$
$\{|l|\}\ handshake\text{-}init\ ht\text{-}GetWork$;;
$\{|l|\}\ handshake\text{-}done$;;
$\{|l|\}\ load\text{-}W$

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 store 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
$not\text{-}blocked :: ('field, 'mut, 'payload, 'ref)\ local\text{-}state \Rightarrow 'mut\ process\text{-}name \Rightarrow bool$
where
$not\text{-}blocked\ s\ p = (case\ mem\text{-}lock\ s\ of\ None \Rightarrow True\ |\ Some\ p' \Rightarrow p = p')$

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

definition
$do\text{-}store\text{-}action :: ('field, 'payload, 'ref)\ mem\text{-}store\text{-}action \Rightarrow ('field, 'mut, 'payload, 'ref)\ local\text{-}state \Rightarrow ('field, 'mut, 'payload, 'ref)\ local\text{-}state$
where
$do\text{-}store\text{-}action\ wact =$
$(\lambda s.\ case\ wact\ of$
$mw\text{-}Mark\ r\ gc\text{-}mark \Rightarrow s(\!|heap := (heap\ s)(r := map\text{-}option\ (\lambda obj.\ obj(\!|obj\text{-}mark := gc\text{-}mark|\!))\ (heap\ s\ r))|\!)$
$|\ mw\text{-}Mutate\ r\ f\ new\text{-}r \Rightarrow s(\!|heap := (heap\ s)(r := map\text{-}option\ (\lambda obj.\ obj(\!|obj\text{-}fields := (obj\text{-}fields\ obj)(f := new\text{-}r)\ |\!))\ (heap\ s\ r))|\!)$
$|\ mw\text{-}Mutate\text{-}Payload\ r\ f\ pl \Rightarrow s(\!|heap := (heap\ s)(r := map\text{-}option\ (\lambda obj.\ obj(\!|obj\text{-}payload := (obj\text{-}payload\ obj)(f := pl)\ |\!))\ (heap\ s\ r))|\!)$
$|\ mw\text{-}fM\ gc\text{-}mark \Rightarrow s(\!|fM := gc\text{-}mark|\!)$
$|\ mw\text{-}fA\ gc\text{-}mark \Rightarrow s(\!|fA := gc\text{-}mark|\!)$

| *mw-Phase gc-phase* ⇒ *s*⦇*phase* := *gc-phase*⦈)

definition
fold-stores :: ('*field*, '*payload*, '*ref*) *mem-store-action list* ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *local-state* ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *local-state*
where
fold-stores ws = *fold* (λ*w*. (∘) (*do-store-action w*)) *ws id*

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

definition
do-load-action :: ('*field*, '*ref*) *mem-load-action*
⇒ ('*field*, '*mut*, '*payload*, '*ref*) *local-state*
⇒ ('*field*, '*payload*, '*ref*) *response*
where
do-load-action ract =
(λ*s*. *case ract of*
mr-Ref r f ⇒ *mv-Ref* (*Option.bind* (*heap s r*) (λ*obj*. *obj-fields obj f*))
| *mr-Payload r f* ⇒ *mv-Payload* (*Option.bind* (*heap s r*) (λ*obj*. *obj-payload obj f*))
| *mr-Mark r* ⇒ *mv-Mark* (*map-option obj-mark* (*heap s r*))
| *mr-Phase* ⇒ *mv-Phase* (*phase s*)
| *mr-fM* ⇒ *mv-Mark* (*Some* (*fM s*))
| *mr-fA* ⇒ *mv-Mark* (*Some* (*fA s*)))

definition
sys-load :: '*mut process-name*
⇒ ('*field*, '*ref*) *mem-load-action*
⇒ ('*field*, '*mut*, '*payload*, '*ref*) *local-state*
⇒ ('*field*, '*payload*, '*ref*) *response*
where
sys-load p ract = *do-load-action ract* ∘ *processors-view-of-memory p*

context *sys*
begin

The semantics of TSO memory following Sewell et al. (2010, §3). This differs from the earlier Owens, Sarkar, and Sewell (2009) by allowing the TSO lock to be taken by a process with a non-empty store 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 store in the store buffer for processor p and commit it to memory, provided p either holds the lock or no processor does.

definition
mem-TSO :: ('*field*, '*mut*, '*payload*, '*ref*) *gc-com*
where
mem-TSO =
⦃''*tso-load*''⦄ *Response* (λ*req s*. { (*s*, *sys-load p mr s*)
|*p mr*. *req* = (*p*, *ro-Load mr*) ∧ *not-blocked s p* })
⊕ ⦃''*tso-store*''⦄ *Response* (λ*req s*. { (*s*⦇ *mem-store-buffers* := (*mem-store-buffers s*)(*p* := *mem-store-buffers s p* @ [*w*]) ⦈, *mv-Void*)
|*p w*. *req* = (*p*, *ro-Store w*) })
⊕ ⦃''*tso-mfence*''⦄ *Response* (λ*req s*. { (*s*, *mv-Void*)
|*p*. *req* = (*p*, *ro-MFENCE*) ∧ *mem-store-buffers s p* = [] })
⊕ ⦃''*tso-lock*''⦄ *Response* (λ*req s*. { (*s*⦇ *mem-lock* := *Some p* ⦈, *mv-Void*)
|*p*. *req* = (*p*, *ro-Lock*) ∧ *mem-lock s* = *None* })
⊕ ⦃''*tso-unlock*''⦄ *Response* (λ*req s*. { (*s*⦇ *mem-lock* := *None* ⦈, *mv-Void*)

$|p.\ req = (p,\ ro\text{-}Unlock) \wedge mem\text{-}lock\ s = Some\ p \wedge mem\text{-}store\text{-}buffers\ s\ p = []\ \})$

$\oplus$ ⦃''*tso-dequeue-store-buffer*''⦄ *LocalOp* (λ*s*. { (*do-store-action w s*)⦇ *mem-store-buffers* := (*mem-store-buffers s*)(*p* := *ws*) ⦈

| *p w ws*. *mem-store-buffers s p* = *w* # *ws* ∧ *not-blocked s p* ∧ *p* ≠ *sys* })

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

For now we assume that the system process magically allocates and deallocates references.

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. We instead signal the exhaustion of the heap explicitly, i.e., the *ro-Alloc* action cannot fail.

definition
  alloc :: (′*field*, ′*mut*, ′*payload*, ′*ref*) *gc-com*
where
  alloc = ⦃''*alloc*''⦄ *Response* (λ*req s*.
    if dom (*heap s*) = *UNIV*
    then {(*s*, *mv-Ref None*) |-::*unit*. *snd req* = *ro-Alloc* }
    else { (*s*⦇ *heap* := (*heap s*)(*r* := *Some* ⦇ *obj-mark* = *fA s*, *obj-fields* = *Map.empty*, *obj-payload* = *Map.empty* ⦈) ⦈, *mv-Ref* (*Some r*))
      |*r*. *r* ∉ *dom* (*heap s*) ∧ *snd req* = *ro-Alloc* })

References are freed by removing them from *heap*.

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

The top-level system process.

definition
  com :: (′*field*, ′*mut*, ′*payload*, ′*ref*) *gc-com*
where
  com =
   LOOP DO
     mem-TSO
    ⊕ *alloc*
    ⊕ *free*
    ⊕ *handshake*
   OD

end

2.4 Mutators

The mutators need to cooperate with the garbage collector. In particular, when the garbage collector is not idle the mutators use a *write barrier* (see §2.1).

The local state for each mutator tracks a working set of references, which abstracts from how the process's registers and stack are traversed to discover roots.

context *mut-m*
begin

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

abbreviation *alloc* :: (′*field*, ′*mut*, ′*payload*, ′*ref*) *gc-com* **where**
  alloc ≡
   ⦃''*alloc*''⦄ *Request* (λ*s*. (*mutator m*, *ro-Alloc*))
     (λ*mv s*. { *s*⦇ *roots* := *roots s* ∪ *set-option opt-r* ⦈ |*opt-r*. *mv* = *mv-Ref opt-r* })

The mutator can always discard any references it holds.

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

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

Dereferencing a reference can increase the set of mutator roots.

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

Storing a reference involves marking both the old and new references, i.e., both *insertion* and *deletion* barriers are installed. The deletion barrier preserves the *weak tricolour invariant*, and the insertion barrier preserves the *strong tricolour invariant*; see §4.2 for further discussion.

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

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

abbreviation
  store-ref :: *location*
        ⇒ (('*field*, '*mut*, '*payload*, '*ref*) *local-state* ⇒ '*ref*)
        ⇒ (('*field*, '*mut*, '*payload*, '*ref*) *local-state* ⇒ '*field*)
        ⇒ (('*field*, '*mut*, '*payload*, '*ref*) *local-state* ⇒ '*ref option*)
        ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *gc-com* (‹⦃-⦄ *store'-ref*›)
where
  ⦃*l*⦄ *store-ref r f r'* ≡ ⦃*l*⦄ *Request* (λ*s*. (*mutator m*, *StoreRef* (*r s*) (*f s*) (*r' s*))) (λ- *s*. {*s*⦇*ghost-honorary-root* := {}⦈})

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

Load and store payload data.

abbreviation *load-payload* :: ('*field*, '*mut*, '*payload*, '*ref*) *gc-com* **where**
load-payload ≡
⦃''*mut-load-payload-choose*''⦄ *LocalOp* (λ*s*. { *s*⦇ *tmp-ref* := *r*, *field* := *f* ⦈ |*r f*. *r* ∈ *roots s* }) ;;
⦃''*mut-load-payload*''⦄ *Request* (λ*s*. (*mutator m*, *LoadPayload* (*tmp-ref s*) (*field s*)))
(λ*mv s*. { *s*⦇ *mutator-data* := (*mutator-data s*)(*var* := *pl*) ⦈
|*var pl*. *mv* = *mv-Payload pl* })

abbreviation *store-payload* :: ('*field*, '*mut*, '*payload*, '*ref*) *gc-com* **where**
store-payload ≡
⦃''*mut-store-payload-choose*''⦄ *LocalOp* (λ*s*. { *s*⦇ *tmp-ref* := *r*, *field* := *f*, *payload-value* := *pl s* ⦈ |*r f pl*. *r* ∈ *roots s* }) ;;
⦃''*mut-store-payload*''⦄ *Request* (λ*s*. (*mutator m*, *StorePayload* (*tmp-ref s*) (*field s*) (*payload-value s*)))
(λ*mv s*. { *s*⦇ *mutator-data* := (*mutator-data s*)(*f* := *pl*) ⦈
|*f pl*. *mv* = *mv-Payload pl* })

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*, '*payload*, '*ref*) *gc-com*
where
com =
LOOP DO
⦃''*mut-local-computation*''⦄ *LocalOp* (λ*s*. {*s*⦇*mutator-data* := *f* (*mutator-data s*)⦈ |*f*. *True*})
⊕ *alloc*
⊕ *discard*
⊕ *load*
⊕ *store*
⊕ *load-payload*
⊕ *store-payload*
⊕ ⦃''*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*, '*payload*, '*ref*) *local-state* ⇒ '*ref*)
⇒ (('*field*, '*mut*, '*payload*, '*ref*) *local-state* ⇒ '*field*)
⇒ (('*ref option* ⇒ '*ref option*) ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *local-state* ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *local-state*) ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *gc-com*
where
gc-deref l r f upd ≡ ⦃*l*⦄ *Request* (λ*s*. (*gc*, *LoadRef* (*r s*) (*f s*)))
(λ*mv s*. { *upd* ⟨*r*'⟩ *s* |*r*'. *mv* = *mv-Ref r*' })

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

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

context gc
begin

abbreviation store-fA-syn :: location ⇒ ((′field, ′mut, ′payload, ′ref) local-state ⇒ gc-mark) ⇒ (′field, ′mut, ′payload, ′ref) gc-com (‹{|-|} store′-fA›) **where**
{|l|} store-fA f ≡ {|l|} Request (λs. (gc, StorefA (f s))) (λ- s. {s})

abbreviation load-fM-syn :: location ⇒ (′field, ′mut, ′payload, ′ref) gc-com (‹{|-|} load′-fM›) **where**
{|l|} load-fM ≡ {|l|} Request (λs. (gc, LoadfM)) (λmv s. { s(|fM := m|) |m. mv = mv-Mark (Some m) })

abbreviation store-fM-syn :: location ⇒ (′field, ′mut, ′payload, ′ref) gc-com (‹{|-|} store′-fM›) **where**
{|l|} store-fM ≡ {|l|} Request (λs. (gc, StorefM (fM s))) (λ- s. {s})

abbreviation store-phase-syn :: location ⇒ gc-phase ⇒ (′field, ′mut, ′payload, ′ref) gc-com (‹{|-|} store′-phase›) **where**
{|l|} store-phase ph ≡ {|l|} Request (λs. (gc, StorePhase ph)) (λ- s. {s(| phase := ph |)})

abbreviation mark-object-syn :: location ⇒ (′field, ′mut, ′payload, ′ref) gc-com (‹{|-|} mark′-object›) **where**
{|l|} mark-object ≡ mark-object-fn gc l

abbreviation free-syn :: location ⇒ ((′field, ′mut, ′payload, ′ref) local-state ⇒ ′ref) ⇒ (′field, ′mut, ′payload, ′ref) gc-com (‹{|-|} free›) **where**
{|l|} free r ≡ {|l|} Request (λs. (gc, ro-Free (r s))) (λ- s. {s})

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

definition (**in** gc)
com :: (′field, ′mut, ′payload, ′ref) gc-com
where
com =
LOOP DO
{|′′idle-noop′′|} handshake-noop ;; — hp-Idle

{|′′idle-load-fM′′|} load-fM ;;
{|′′idle-invert-fM′′|} ´fM := (¬ ´fM) ;;
{|′′idle-store-fM′′|} store-fM ;;

{|′′idle-flip-noop′′|} handshake-noop ;; — hp-IdleInit

{|′′idle-phase-init′′|} store-phase ph-Init ;;

{|′′init-noop′′|} handshake-noop ;; — hp-InitMark

{|′′init-phase-mark′′|} store-phase ph-Mark ;;
{|′′mark-load-fM′′|} load-fM ;;
{|′′mark-store-fA′′|} store-fA fM ;;

{|′′mark-noop′′|} handshake-noop ;; — hp-Mark

{|''mark-loop-get-roots''|} *handshake-get-roots* ;; — *hp-IdleMarkSweep*

{|''mark-loop''|} *WHILE* ¬*EMPTY W DO*
  {|''mark-loop-inner''|} *WHILE* ¬*EMPTY W DO*
    {|''mark-loop-choose-ref''|} ´*tmp-ref* :∈ ´*W* ;;
    {|''mark-loop-fields''|} ´*field-set* := *UNIV* ;;
    {|''mark-loop-mark-object-loop''|} *WHILE* ¬*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''|} *store-phase ph-Sweep* ;;
{|''sweep-load-fM''|} *load-fM* ;;
{|''sweep-refs''|} ´*refs* := *UNIV* ;;
{|''sweep-loop''|} *WHILE* ¬*EMPTY refs DO*
  {|''sweep-loop-choose-ref''|} ´*tmp-ref* :∈ ´*refs* ;;
  {|''sweep-loop-load-mark''|} ´*mark* := ´*tmp-ref*→*flag* ;;
  {|''sweep-loop-check''|} *IF* ¬*NULL mark* ∧ *the* ∘ *mark* ≠ *fM THEN*
    {|''sweep-loop-free''|} *free tmp-ref*
  FI ;;
  {|''sweep-loop-ref-done''|} ´*refs* := (´*refs* − {´*tmp-ref*})
OD ;;
{|''sweep-idle''|} *store-phase ph-Idle*
OD

end

primrec
  gc-coms :: ′*mut process-name* ⇒ (′*field*, ′*mut*, ′*payload*, ′*ref*) *gc-com*
where
  gc-coms (*mutator m*) = *mut-m.com m*
| *gc-coms gc* = *gc.com*
| *gc-coms sys* = *sys.com*

3 Proofs Basis

Extra HOL.

lemma *Set-bind-insert*[*simp*]:
  Set.bind (*insert a A*) *B* = *B a* ∪ (*Set.bind A B*)
⟨*proof*⟩

lemma *option-bind-invE*[*elim*]:
  ⟦ *Option.bind f g* = *None*; ⋀*a*. ⟦ *f* = *Some a*; *g a* = *None* ⟧ ⟹ *Q*; *f* = *None* ⟹ *Q* ⟧ ⟹ *Q*
  ⟦ *Option.bind f g* = *Some x*; ⋀*a*. ⟦ *f* = *Some a*; *g a* = *Some x* ⟧ ⟹ *Q* ⟧ ⟹ *Q*
⟨*proof*⟩

lemmas *conj-explode* = *conj-imp-eq-imp-imp*

Tweak the default simpset:

- "not in dom" as a premise negates the goal
- we always want to execute suffix
- we try to make simplification rules about *fun-upd* more stable

declare *dom-def*[*simp*]
declare *suffix-to-prefix*[*simp*]
declare *map-option.compositionality*[*simp*]
declare *o-def*[*simp*]
declare *Option.Option.option.set-map*[*simp*]
declare *bind-image*[*simp*]

declare *fun-upd-apply*[*simp del*]
declare *fun-upd-same*[*simp*]
declare *fun-upd-other*[*simp*]

declare *gc-phase.case-cong*[*cong*]
declare *mem-store-action.case-cong*[*cong*]
declare *process-name.case-cong*[*cong*]
declare *hs-phase.case-cong*[*cong*]
declare *hs-type.case-cong*[*cong*]

declare *if-split-asm*[*split*]

Collect the component definitions. Inline everything. This is what the proofs work on. Observe we lean heavily on locales.

context *gc*
begin

lemmas *all-com-defs* =
handshake-done-def handshake-init-def handshake-noop-def handshake-get-roots-def handshake-get-work-def
mark-object-fn-def

lemmas *com-def2* = *com-def*[*simplified all-com-defs append.simps if-True if-False*]

intern-com *com-def2*

end

context *mut-m*
begin

lemmas *all-com-defs* =
mut-m.handshake-def mut-m.store-def
mark-object-fn-def

lemmas *com-def2* = *mut-m.com-def*[*simplified all-com-defs append.simps if-True if-False*]

intern-com *com-def2*

end

context *sys*
begin

lemmas *all-com-defs* =
sys.alloc-def sys.free-def sys.mem-TSO-def sys.handshake-def

lemmas *com-def2* = *com-def*[*simplified all-com-defs append.simps if-True if-False*]

intern-com *com-def2*

end

lemmas *all-com-interned-defs* = *gc.com-interned mut-m.com-interned sys.com-interned*

named-theorems *inv Location−sensitive invariant definitions*
named-theorems *nie Non−interference elimination rules*

3.1 Model-specific 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 *prefixed* :: *location* ⇒ *location set* **where**
prefixed p ≡ { *l* . *prefix p l* }

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

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

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

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

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

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

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

context *mut-m*
begin

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

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

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

end

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

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

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

Projections on TSO buffers.

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

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

abbreviation (*input*) *tso-pending-fA p* ≡ *tso-pending p is-mw-fA*
abbreviation (*input*) *tso-pending-fM p* ≡ *tso-pending p is-mw-fM*
abbreviation (*input*) *tso-pending-mark p* ≡ *tso-pending p is-mw-Mark*
abbreviation (*input*) *tso-pending-mw-mutate p* ≡ *tso-pending p is-mw-Mutate*
abbreviation (*input*) *tso-pending-mutate p* ≡ *tso-pending p* (*is-mw-Mutate* ∨ *is-mw-Mutate-Payload*) — TSO makes it (mostly) not worth distinguishing these.
abbreviation (*input*) *tso-pending-phase p* ≡ *tso-pending p is-mw-Phase*

abbreviation (*input*) *tso-no-pending-marks* ≡ ∀ *p*. *LIST-NULL* (*tso-pending-mark p*)

A somewhat-useful abstraction of the heap, following l4.verified, which asserts that there is an object at the given reference with the given property. In some sense this encodes a three-valued logic.

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

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

definition *valid-null-ref* :: *'ref option* ⇒ (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**
  valid-null-ref r ≡ *case r of None* ⇒ ⟨*True*⟩ | *Some r'* ⇒ *valid-ref r'*

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

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

definition *reaches* :: *'ref* ⇒ *'ref* ⇒ (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* (**infix** ‹*reaches*› *51*) **where**
  x reaches y = (λ*s*. (λ*x y*. (*x points-to y*) *s*)** *x y*)

The predicate *obj-at-field-on-heap* asserts that *obj-at* (λ*s*. *True*) *r* and if *f* is a field of the object referred to by *r* then it it satisfies *P*.

definition *obj-at-field-on-heap* :: (*'ref* ⇒ *bool*) ⇒ *'ref* ⇒ *'field* ⇒ (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**

```
  obj-at-field-on-heap P r f ≡ λs.
    case map-option obj-fields (sys-heap s r) of
        None ⇒ False
      | Some fs ⇒ (case fs f of None ⇒ True
                              | Some r' ⇒ P r')
```

3.2 Object colours

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 or *ghost-honorary-grey*

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* ⇒ (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**
  marked r s ≡ *obj-at* (λ*obj*. *obj-mark obj* = *sys-fM s*) *r s*

definition *white* :: *'ref* ⇒ (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**
  white r s ≡ *obj-at* (λ*obj*. *obj-mark obj* ≠ *sys-fM s*) *r s*

definition *WL* :: *'mut process-name* ⇒ (*'field*, *'mut*, *'payload*, *'ref*) *lsts* ⇒ *'ref set* **where**
  WL p = (λ*s*. *W* (*s p*) ∪ *ghost-honorary-grey* (*s p*))

definition *grey* :: *'ref* ⇒ (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**
  grey r = (∃ *p*. ⟨*r*⟩ ∈ *WL p*)

definition *black* :: *'ref* ⇒ (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**
  black r ≡ *marked r* ∧ ¬*grey r*

These demonstrate the overlap in colours.

lemma *colours-distinct*[*dest*]:
  black r s ⟹ ¬*grey r s*
  black r s ⟹ ¬*white r s*
  grey r s ⟹ ¬*black r s*
  white r s ⟹ ¬*black r s*
⟨*proof*⟩

lemma *marked-imp-black-or-grey*:
  marked r s ⟹ *black r s* ∨ *grey r s*
  ¬ *white r s* ⟹ ¬ *valid-ref r s* ∨ *black r s* ∨ *grey r s*
⟨*proof*⟩

In some phases the heap is monochrome.

definition *black-heap* :: (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**
  black-heap = (∀ *r*. *valid-ref r* ⟶ *black r*)

definition *white-heap* :: (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**
  white-heap = (∀ *r*. *valid-ref r* ⟶ *white r*)

definition *no-black-refs* :: (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**
  no-black-refs = (∀ *r*. ¬*black r*)

definition *no-grey-refs* :: (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**
  no-grey-refs = (∀ *r*. ¬*grey r*)

3.3 Reachability

We treat pending TSO heap mutations as extra mutator roots.

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

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

abbreviation (**in** *mut-m*) *root* :: *'ref* ⇒ (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**
  root x ≡ ⟨*x*⟩ ∈ *mut-roots* ∪ *mut-ghost-honorary-root* ∪ *tso-store-refs*

definition (**in** *mut-m*) *reachable* :: *'ref* ⇒ (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**

reachable y = ($\exists$ *x. root x* $\wedge$ *x reaches y*)

definition *grey-reachable* :: *'ref* $\Rightarrow$ (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**
grey-reachable y = ($\exists$ *g. grey g* $\wedge$ *g reaches y*)

3.4 Sundry detritus

lemmas *eq-imp-simps* = — equations for deriving useful things from *eq-imp* facts
eq-imp-def
all-conj-distrib
split-paired-All split-def fst-conv snd-conv prod-eq-iff
conj-explode
simp-thms

lemma *p-not-sys*:
$p \neq sys \longleftrightarrow p = gc \vee (\exists m.\ p = mutator\ m)$
⟨*proof*⟩

lemma (**in** *mut-m'*) *m'm*[*iff*]: $m' \neq m$
⟨*proof*⟩

obj at

lemma *obj-at-cong*[*cong*]:
$[\![\bigwedge obj.\ sys\text{-}heap\ s\ r = Some\ obj \Longrightarrow P\ obj = P'\ obj;\ r = r';\ s = s']\!]$
$\Longrightarrow obj\text{-}at\ P\ r\ s \longleftrightarrow obj\text{-}at\ P'\ r'\ s'$
⟨*proof*⟩

lemma *obj-at-split*:
$Q\ (obj\text{-}at\ P\ r\ s) = ((sys\text{-}heap\ s\ r = None \longrightarrow Q\ False) \wedge (\forall obj.\ sys\text{-}heap\ s\ r = Some\ obj \longrightarrow Q\ (P\ obj)))$
⟨*proof*⟩

lemma *obj-at-split-asm*:
$Q\ (obj\text{-}at\ P\ r\ s) = (\neg\ ((sys\text{-}heap\ s\ r = None \wedge \neg Q\ False) \vee (\exists obj.\ sys\text{-}heap\ s\ r = Some\ obj \wedge \neg\ Q\ (P\ obj))))$
⟨*proof*⟩

lemmas *obj-at-splits* = *obj-at-split obj-at-split-asm*

lemma *obj-at-eq-imp*:
eq-imp (λ(-::*unit*) *s. map-option P* (*sys-heap s r*))
(*obj-at P r*)
⟨*proof*⟩

lemmas *obj-at-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF obj-at-eq-imp*, *simplified eq-imp-simps*]

lemma *obj-at-simps*:
$obj\text{-}at\ (\lambda obj.\ P\ obj \wedge Q\ obj)\ r\ s \longleftrightarrow obj\text{-}at\ P\ r\ s \wedge obj\text{-}at\ Q\ r\ s$

⟨*proof*⟩

obj at field on heap

lemma *obj-at-field-on-heap-cong*[*cong*]:
$[\![\bigwedge r'\ obj.\ [\![sys\text{-}heap\ s\ r = Some\ obj;\ obj\text{-}fields\ obj\ f = Some\ r']\!] \Longrightarrow P\ r' = P'\ r';\ r = r';\ f = f';\ s = s']\!]$
$\Longrightarrow obj\text{-}at\text{-}field\text{-}on\text{-}heap\ P\ r\ f\ s \longleftrightarrow obj\text{-}at\text{-}field\text{-}on\text{-}heap\ P'\ r'\ f'\ s'$
⟨*proof*⟩

lemma *obj-at-field-on-heap-split*:
$Q\ (obj\text{-}at\text{-}field\text{-}on\text{-}heap\ P\ r\ f\ s) \longleftrightarrow ((sys\text{-}heap\ s\ r = None \longrightarrow Q\ False)$
$\wedge (\forall obj.\ sys\text{-}heap\ s\ r = Some\ obj \wedge obj\text{-}fields\ obj\ f = None \longrightarrow Q\ True)$

∧ (∀ r′ obj. sys-heap s r = Some obj ∧ obj-fields obj f = Some r′ ⟶ Q (P r′)))
⟨proof⟩

lemma *obj-at-field-on-heap-split-asm*:
Q (obj-at-field-on-heap P r f s) ⟷ (¬ ((sys-heap s r = None ∧ ¬Q False)
∨ (∃ obj. sys-heap s r = Some obj ∧ obj-fields obj f = None ∧ ¬ Q True)
∨ (∃ r′ obj. sys-heap s r = Some obj ∧ obj-fields obj f = Some r′ ∧ ¬ Q (P r′))))
⟨proof⟩

lemmas *obj-at-field-on-heap-splits* = *obj-at-field-on-heap-split obj-at-field-on-heap-split-asm*

lemma *obj-at-field-on-heap-eq-imp*:
eq-imp (λ(-::unit) s. sys-heap s r)
(obj-at-field-on-heap P r f)
⟨proof⟩

lemmas *obj-at-field-on-heap-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF obj-at-field-on-heap-eq-imp*, *simplified eq-imp-simps*]

lemma *obj-at-field-on-heap-imp-valid-ref*[*elim*]:
obj-at-field-on-heap P r f s ⟹ valid-ref r s
obj-at-field-on-heap P r f s ⟹ valid-null-ref (Some r) s
⟨proof⟩

lemma *obj-at-field-on-heapE*[*elim*]:
⟦ obj-at-field-on-heap P r f s; sys-heap s′ r = sys-heap s r; ⋀r′. P r′ ⟹ P′ r′ ⟧
⟹ obj-at-field-on-heap P′ r f s′
⟨proof⟩

lemma *valid-null-ref-eq-imp*:
eq-imp (λ(-::unit) s. Option.bind r (map-option ⟨True⟩ ∘ sys-heap s))
(valid-null-ref r)
⟨proof⟩

lemmas *valid-null-ref-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF valid-null-ref-eq-imp*, *simplified*]

lemma *valid-null-ref-simps*[*simp*]:
valid-null-ref None s
valid-null-ref (Some r) s ⟷ valid-ref r s
⟨proof⟩

Derive simplification rules from *case* expressions

simps-of-case *hs-step-simps*[*simp*]: *hs-step-def* (*splits*: *hs-phase.split*)
simps-of-case *do-load-action-simps*[*simp*]: *fun-cong*[*OF do-load-action-def*[*simplified atomize-eq*]] (*splits*: *mem-load-act*
simps-of-case *do-store-action-simps*[*simp*]: *fun-cong*[*OF do-store-action-def*[*simplified atomize-eq*]] (*splits*: *mem-store-a*

lemma *do-store-action-prj-simps*[*simp*]:
fM (do-store-action w s) = fl ⟷ (fM s = fl ∧ w ≠ mw-fM (¬fM s)) ∨ w = mw-fM fl
fl = fM (do-store-action w s) ⟷ (fl = fM s ∧ w ≠ mw-fM (¬fM s)) ∨ w = mw-fM fl
fA (do-store-action w s) = fl ⟷ (fA s = fl ∧ w ≠ mw-fA (¬fA s)) ∨ w = mw-fA fl
fl = fA (do-store-action w s) ⟷ (fl = fA s ∧ w ≠ mw-fA (¬fA s)) ∨ w = mw-fA fl
ghost-hs-in-sync (do-store-action w s) = ghost-hs-in-sync s
ghost-hs-phase (do-store-action w s) = ghost-hs-phase s
ghost-honorary-grey (do-store-action w s) = ghost-honorary-grey s
hs-pending (do-store-action w s) = hs-pending s
hs-type (do-store-action w s) = hs-type s
heap (do-store-action w s) r = None ⟷ heap s r = None
mem-lock (do-store-action w s) = mem-lock s

$phase\ (do\text{-}store\text{-}action\ w\ s) = ph \longleftrightarrow (phase\ s = ph \wedge (\forall\, ph'.\ w \neq mw\text{-}Phase\ ph') \vee w = mw\text{-}Phase\ ph)$
$ph = phase\ (do\text{-}store\text{-}action\ w\ s) \longleftrightarrow (ph = phase\ s \wedge (\forall\, ph'.\ w \neq mw\text{-}Phase\ ph') \vee w = mw\text{-}Phase\ ph)$
$W\ (do\text{-}store\text{-}action\ w\ s) = W\ s$
⟨*proof*⟩

reaches

lemma *reaches-refl*[*iff*]:
$(r\ reaches\ r)\ s$
⟨*proof*⟩

lemma *reaches-step*[*intro*]:
$[\![(x\ reaches\ y)\ s;\ (y\ points\text{-}to\ z)\ s]\!] \Longrightarrow (x\ reaches\ z)\ s$
$[\![(y\ reaches\ z)\ s;\ (x\ points\text{-}to\ y)\ s]\!] \Longrightarrow (x\ reaches\ z)\ s$
⟨*proof*⟩

lemma *reaches-induct*[*consumes 1*, *case-names refl step*, *induct set*: *reaches*]:
assumes $(x\ reaches\ y)\ s$
assumes $\bigwedge x.\ P\ x\ x$
assumes $\bigwedge x\ y\ z.\ [\![(x\ reaches\ y)\ s;\ P\ x\ y;\ (y\ points\text{-}to\ z)\ s]\!] \Longrightarrow P\ x\ z$
shows $P\ x\ y$
⟨*proof*⟩

lemma *converse-reachesE*[*consumes 1*, *case-names base step*]:
assumes $(x\ reaches\ z)\ s$
assumes $x = z \Longrightarrow P$
assumes $\bigwedge y.\ [\![(x\ points\text{-}to\ y)\ s;\ (y\ reaches\ z)\ s]\!] \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *reaches-fields*: — Complicated condition takes care of *alloc*: collapses no object and object with no fields
assumes $(x\ reaches\ y)\ s'$
assumes $\forall\, r'.\ \bigcup(ran\ `\ obj\text{-}fields\ `\ set\text{-}option\ (sys\text{-}heap\ s'\ r')) = \bigcup(ran\ `\ obj\text{-}fields\ `\ set\text{-}option\ (sys\text{-}heap\ s\ r'))$
shows $(x\ reaches\ y)\ s$
⟨*proof*⟩

lemma *reaches-eq-imp*:
$eq\text{-}imp\ (\lambda r'\ s.\ \bigcup(ran\ `\ obj\text{-}fields\ `\ set\text{-}option\ (sys\text{-}heap\ s\ r')))$
$(x\ reaches\ y)$
⟨*proof*⟩

lemmas *reaches-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF reaches-eq-imp*, *simplified eq-imp-simps*, *rule-format*]

Location-specific facts.

lemma *obj-at-mark-dequeue*[*simp*]:
$obj\text{-}at\ P\ r\ (s(sys := s\ sys(\!|\ heap := (sys\text{-}heap\ s)(r' := map\text{-}option\ (obj\text{-}mark\text{-}update\ (\lambda\text{-}.\ fl))\ (sys\text{-}heap\ s\ r')),$ $mem\text{-}store\text{-}buffers := wb'\ |\!)))$
$\longleftrightarrow obj\text{-}at\ (\lambda obj.\ (P\ (if\ r = r'\ then\ obj(\!|\ obj\text{-}mark := fl\ |\!)\ else\ obj)))\ r\ s$
⟨*proof*⟩

lemma *obj-at-field-on-heap-mw-simps*[*simp*]:
$obj\text{-}at\text{-}field\text{-}on\text{-}heap\ P\ r0\ f0$
$(s(sys := (s\ sys)(\!|\ heap := (sys\text{-}heap\ s)(r := map\text{-}option\ (\lambda obj :: ('field, 'payload, 'ref)\ object.\ obj(\!|obj\text{-}fields := (obj\text{-}fields\ obj)(f := opt\text{-}r')|\!))\ (sys\text{-}heap\ s\ r)),$
$mem\text{-}store\text{-}buffers := (mem\text{-}store\text{-}buffers\ (s\ Sys))(p := ws)\ |\!)))$
$\longleftrightarrow (\ (r \neq r0 \vee f \neq f0) \wedge obj\text{-}at\text{-}field\text{-}on\text{-}heap\ P\ r0\ f0\ s\ )$
$\vee (r = r0 \wedge f = f0 \wedge valid\text{-}ref\ r\ s \wedge (case\ opt\text{-}r'\ of\ Some\ r'' \Rightarrow P\ r''\ |\ \text{-} \Rightarrow True))$
$obj\text{-}at\text{-}field\text{-}on\text{-}heap\ P\ r\ f\ (s(sys := s\ sys(\!|heap := (sys\text{-}heap\ s)(r' := map\text{-}option\ (obj\text{-}mark\text{-}update\ (\lambda\text{-}.\ fl))$ $(sys\text{-}heap\ s\ r')),\ mem\text{-}store\text{-}buffers := sb'|\!)))$

$\longleftrightarrow$ *obj-at-field-on-heap P r f s*
⟨*proof*⟩

lemma *obj-at-field-on-heap-no-pending-stores*:
⟦ *sys-load* (*mutator m*) (*mr-Ref r f*) (*s sys*) = *mv-Ref opt-r′*; ∀ *opt-r′*. *mw-Mutate r f opt-r′* ∉ *set* (*sys-mem-store-buffers* (*mutator m*) *s*); *valid-ref r s* ⟧
$\Longrightarrow$ *obj-at-field-on-heap* (λ*r*. *opt-r′* = *Some r*) *r f s*
⟨*proof*⟩

4 Global Invariants

4.1 The valid references invariant

The key safety property of a GC is that it does not free objects that are reachable from mutator roots. The GC also requires that there are objects for all references reachable from grey objects.

definition *valid-refs-inv* :: (*′field*, *′mut*, *′payload*, *′ref*) *lsts-pred* **where**
valid-refs-inv = (∀ *m x*. *mut-m.reachable m x* ∨ *grey-reachable x* ⟶ *valid-ref x*)

The remainder of the invariants support the inductive argument that this one holds.

4.2 The strong-tricolour invariant

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

abbreviation *points-to-white* :: *′ref* ⇒ *′ref* ⇒ (*′field*, *′mut*, *′payload*, *′ref*) *lsts-pred* (**infix** ‹*points′-to′-white*› *51*) **where**
x points-to-white y ≡ *x points-to y* ∧ *white y*

definition *strong-tricolour-inv* :: (*′field*, *′mut*, *′payload*, *′ref*) *lsts-pred* **where**
strong-tricolour-inv = (∀ *b w*. *black b* ⟶ ¬*b points-to-white w*)

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

definition *has-white-path-to* :: *′ref* ⇒ *′ref* ⇒ (*′field*, *′mut*, *′payload*, *′ref*) *lsts-pred* (**infix** ‹*has′-white′-path′-to*› *51*) **where**
x has-white-path-to y = (λ*s*. (λ*x y*. (*x points-to-white y*) *s*)** *x y*)

definition *grey-protects-white* :: *′ref* ⇒ *′ref* ⇒ (*′field*, *′mut*, *′payload*, *′ref*) *lsts-pred* (**infix** ‹*grey′-protects′-white*› *51*) **where**
g grey-protects-white w = (*grey g* ∧ *g has-white-path-to w*)

definition *weak-tricolour-inv* :: (*′field*, *′mut*, *′payload*, *′ref*) *lsts-pred* **where**
weak-tricolour-inv =
(∀ *b w*. *black b* ∧ *b points-to-white w* ⟶ (∃ *g*. *g grey-protects-white w*))

lemma *strong-tricolour-inv s* $\Longrightarrow$ *weak-tricolour-inv s*
⟨*proof*⟩

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

definition *in-snapshot* :: *′ref* ⇒ (*′field*, *′mut*, *′payload*, *′ref*) *lsts-pred* **where**
in-snapshot r = (*black r* ∨ (∃ *g*. *g grey-protects-white r*))

definition (**in** *mut-m*) *reachable-snapshot-inv* :: (*′field*, *′mut*, *′payload*, *′ref*) *lsts-pred* **where**
reachable-snapshot-inv = (∀ *r*. *reachable r* ⟶ *in-snapshot r*)

4.3 Phase invariants

The phase structure of this GC algorithm greatly complicates this safety proof. The following assertions capture this structure in several relations.

We begin by relating the mutators' *mut-ghost-hs-phase* to *sys-ghost-hs-phase*, which tracks the GC's. Each mutator can be at most one handshake step behind the GC. If any mutator is behind then the GC is stalled on a pending handshake. We include the handshake type as *get-work* can occur any number of times.

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

definition *handshake-phase-inv* :: ('*field*, '*mut*, '*payload*, '*ref*) *lsts-pred* **where**
  handshake-phase-inv = (∀ *m*.
    sys-ghost-hs-in-sync m ⊗ *sys-hs-type* ⊗ *sys-ghost-hs-phase* ⊗ *mut-m.mut-ghost-hs-phase m* ∈ ⟨*hp-step-rel*⟩
  ∧ (*sys-hs-pending m* ⟶ ¬*sys-ghost-hs-in-sync m*))

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 the process doing the marking holds the TSO lock until the mark is committed to the shared memory (see §4.4).

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.

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

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

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

context *mut-m*
begin

definition *marked-insertions* :: ('*field*, '*mut*, '*payload*, '*ref*) *lsts-pred* **where**
  marked-insertions = (∀ *w*. *tso-pending-store* (*mutator m*) *w* ⟶ *marked-insertion w*)

definition *marked-deletions* :: ('*field*, '*mut*, '*payload*, '*ref*) *lsts-pred* **where**
  marked-deletions = (∀ *w*. *tso-pending-store* (*mutator m*) *w* ⟶ *marked-deletion w*)

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

abbreviation *mutator-phase-inv* :: ('*field*, '*mut*, '*payload*, '*ref*) *lsts-pred* **where**

mutator-phase-inv ≡ *mutator-phase-inv-aux* **$** *mut-ghost-hs-phase*

end

abbreviation *mutators-phase-inv* :: ('*field*, '*mut*, '*payload*, '*ref*) *lsts-pred* **where**
mutators-phase-inv ≡ (∀ *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* :: *hs-phase* ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *lsts-pred* **where**
sys-phase-inv-aux hp-Idle = ((*If sys-fA* = *sys-fM Then black-heap Else white-heap*) ∧ *no-grey-refs*)
| *sys-phase-inv-aux hp-IdleInit* = *no-black-refs*
| *sys-phase-inv-aux hp-InitMark* = (*sys-fA* ≠ *sys-fM* ⟶ *no-black-refs*)
| *sys-phase-inv-aux hp-Mark* = ⟨*True*⟩
| *sys-phase-inv-aux hp-IdleMarkSweep* = ((*sys-phase* = ⟨*ph-Idle*⟩ ∨ *tso-pending-store gc* (*mw-Phase ph-Idle*)) ⟶ *no-grey-refs*)

abbreviation *sys-phase-inv* :: ('*field*, '*mut*, '*payload*, '*ref*) *lsts-pred* **where**
sys-phase-inv ≡ *sys-phase-inv-aux* **$** *sys-ghost-hs-phase*

4.3.1 Writes to shared GC variables

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

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

definition *phase-rel-inv* :: ('*field*, '*mut*, '*payload*, '*ref*) *lsts-pred* **where**
phase-rel-inv = ((∀ *m*. *sys-ghost-hs-in-sync m*) ⊗ *sys-ghost-hs-phase* ⊗ *gc-phase* ⊗ *sys-phase* ⊗ *tso-pending-phase gc* ∈ ⟨*phase-rel*⟩)

Similarly we track the validity of *sys-fM* (respectively, *sys-fA*) wrt *gc-fM* (*sys-fA*) and the handshake phase. We also include the TSO lock to rule out the GC having any pending marks during the *hp-Idle* handshake phase.

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

definition *fM-rel-inv* :: (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**
fM-rel-inv = ((∀ *m*. *sys-ghost-hs-in-sync m*) ⊗ *sys-ghost-hs-phase* ⊗ *gc-fM* ⊗ *sys-fM* ⊗ *tso-pending-fM gc* ⊗ (*sys-mem-lock* = ⟨*Some gc*⟩) ∈ ⟨*fM-rel*⟩)

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

definition *fA-rel-inv* :: (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**
fA-rel-inv = ((∀ *m*. *sys-ghost-hs-in-sync m*) ⊗ *sys-ghost-hs-phase* ⊗ *sys-fA* ⊗ *gc-fM* ⊗ *tso-pending-fA gc* ∈ ⟨*fA-rel*⟩)

4.4 Worklist invariants

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

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

Note that the phase invariants of §4.3 limit the scope of this invariant.

definition *valid-W-inv* :: (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**
valid-W-inv =
((∀ *p r*. *r in-W p* ∨ (*sys-mem-lock* ≠ ⟨*Some p*⟩ ∧ *r in-ghost-honorary-grey p*) ⟶ *marked r*)
∧ (∀ *p q*. ⟨*p* ≠ *q*⟩ ⟶ *WL p* ∩ *WL q* = ⟨{}⟩)
∧ (∀ *p q r*. ¬(*r in-ghost-honorary-grey p* ∧ *r in-W q*))
∧ (*EMPTY sys-ghost-honorary-grey*)
∧ (∀ *p r fl*. *tso-pending-store p* (*mw-Mark r fl*)
⟶ ⟨*fl*⟩ = *sys-fM*
∧ *r in-ghost-honorary-grey p*
∧ *tso-locked-by p*
∧ *white r*
∧ *tso-pending-mark p* = ⟨[*mw-Mark r fl*]⟩))

4.5 Coarse invariants about the stores a process can issue

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

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

definition *tso-store-inv* :: (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**
tso-store-inv =
((∀ *w*. *tso-pending-store gc* *w* ⟶ ⟨*gc-writes w*⟩)
∧ (∀ *m w*. *tso-pending-store* (*mutator m*) *w* ⟶ ⟨*mut-writes w*⟩))

4.6 The global invariants collected

definition *invs* :: (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**
invs =
(*handshake-phase-inv*

$\wedge$ *phase-rel-inv*
$\wedge$ *strong-tricolour-inv*
$\wedge$ *sys-phase-inv*
$\wedge$ *tso-store-inv*
$\wedge$ *valid-refs-inv*
$\wedge$ *valid-W-inv*
$\wedge$ *mutators-phase-inv*
$\wedge$ *fA-rel-inv* $\wedge$ *fM-rel-inv*)

4.7 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*, '*payload*, '*ref*) *lst-pred* **where**
gc-initial-state s =
(*fM s* = *initial-mark*
$\wedge$ *phase s* = *ph-Idle*
$\wedge$ *ghost-honorary-grey s* = {}
$\wedge$ *W s* = {})

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

definition *sys-initial-state* :: ('*field*, '*mut*, '*payload*, '*ref*) *lst-pred* **where**
sys-initial-state s =
(($\forall$ *m*. $\neg$*hs-pending s m* $\wedge$ *ghost-hs-in-sync s m*)
$\wedge$ *ghost-hs-phase s* = *hp-IdleMarkSweep* $\wedge$ *hs-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-store-buffers s p* = [])
$\wedge$ *mem-lock s* = *None*)

abbreviation
root-reachable y $\equiv$ $\exists$ *m x*. $\langle x \rangle$ $\in$ *mut-m.mut-roots m* $\wedge$ *x reaches y*

definition *valid-refs* :: ('*field*, '*mut*, '*payload*, '*ref*) *lsts-pred* **where**
valid-refs = ($\forall$ *y*. *root-reachable y* $\longrightarrow$ *valid-ref y*)

definition *gc-system-init* :: (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**
gc-system-init =
((λ*s*. *gc-initial-state* (*s gc*))
∧ (λ*s*. ∀ *m*. *mut-initial-state* (*s* (*mutator m*)))
∧ (λ*s*. *sys-initial-state* (*s sys*))
∧ *valid-refs*)

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

abbreviation *gc-system* :: (*'field*, *'mut*, *'payload*, *'ref*) *gc-system* **where**
gc-system ≡ (|*PGMs* = *gc-coms*, *INIT* = *gc-system-init*, *FAIR* = ⟨*True*⟩|)

end

5 Local invariants

5.1 TSO invariants

context *gc*
begin

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

locset-definition *tso-lock-locs* :: *location set* **where**
tso-lock-locs = (⋃ *l*∈{ *''mo-co-cmark''*, *''mo-co-ctest''*, *''mo-co-mark''*, *''mo-co-unlock''* }. *suffixed l*)

definition *tso-lock-invL* :: (*'field*, *'mut*, *'payload*, *'ref*) *gc-pred* **where**
[*inv*]: *tso-lock-invL* =
(*atS-gc tso-lock-locs* (*tso-locked-by gc*)
∧ *atS-gc* (− *tso-lock-locs*) (¬ *tso-locked-by gc*))

end

context *mut-m*
begin

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

locset-definition *tso-lock-locs* =
(⋃ *l*∈{ *''mo-co-cmark''*, *''mo-co-ctest''*, *''mo-co-mark''*, *''mo-co-unlock''* }. *suffixed l*)

definition *tso-lock-invL* :: (*'field*, *'mut*, *'payload*, *'ref*) *gc-pred* **where**
[*inv*]: *tso-lock-invL* =
(*atS-mut tso-lock-locs* (*tso-locked-by* (*mutator m*))
∧ *atS-mut* (−*tso-lock-locs*) (¬*tso-locked-by* (*mutator m*)))

end

5.2 Handshake phases

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

context *gc*
begin

locset-definition *idle-locs* = *prefixed ''idle''*
locset-definition *init-locs* = *prefixed ''init''*
locset-definition *mark-locs* = *prefixed ''mark''*
locset-definition *sweep-locs* = *prefixed ''sweep''*
locset-definition *mark-loop-locs* = *prefixed ''mark-loop''*

locset-definition *hp-Idle-locs* =
  (*prefixed* ''*idle-noop*'' − { *idle-noop-mfence*, *idle-noop-init-type* })
 ∪ { *idle-load-fM*, *idle-invert-fM*, *idle-store-fM*, *idle-flip-noop-mfence*, *idle-flip-noop-init-type* }

locset-definition *hp-IdleInit-locs* =
  (*prefixed* ''*idle-flip-noop*'' − { *idle-flip-noop-mfence*, *idle-flip-noop-init-type* })
 ∪ { *idle-phase-init*, *init-noop-mfence*, *init-noop-init-type* }

locset-definition *hp-InitMark-locs* =
  (*prefixed* ''*init-noop*'' − { *init-noop-mfence*, *init-noop-init-type* })
 ∪ { *init-phase-mark*, *mark-load-fM*, *mark-store-fA*, *mark-noop-mfence*, *mark-noop-init-type* }

locset-definition *hp-IdleMarkSweep-locs* =
  { *idle-noop-mfence*, *idle-noop-init-type*, *mark-end* }
 ∪ *sweep-locs*
 ∪ (*mark-loop-locs* − { *mark-loop-get-roots-init-type* })

locset-definition *hp-Mark-locs* =
  (*prefixed* ''*mark-noop*'' − { *mark-noop-mfence*, *mark-noop-init-type* })
 ∪ { *mark-loop-get-roots-init-type* }

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

locset-definition *hs-noop-locs* =
 (⋃ *l* ∈ *hs-noop-prefixes*. *prefixed l* − (*suffixed* ''*-noop-mfence*'' ∪ *suffixed* ''*-noop-init-type*''))

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

locset-definition *hs-get-work-locs* =
 prefixed ''*mark-loop-get-work*'' − {*mark-loop-get-work-init-type*}

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

locset-definition *hs-init-loop-locs* = (⋃ *l* ∈ *hs-prefixes*. *prefixed* (*l* @ ''*-init-loop*''))
locset-definition *hs-done-loop-locs* = (⋃ *l* ∈ *hs-prefixes*. *prefixed* (*l* @ ''*-done-loop*''))
locset-definition *hs-done-locs* = (⋃ *l* ∈ *hs-prefixes*. *prefixed* (*l* @ ''*-done*''))
locset-definition *hs-none-pending-locs* = − (*hs-init-loop-locs* ∪ *hs-done-locs*)
locset-definition *hs-in-sync-locs* =
 (− ((⋃ *l* ∈ *hs-prefixes*. *prefixed* (*l* @ ''*-init*'')) ∪ *hs-done-locs*))
 ∪ (⋃ *l* ∈ *hs-prefixes*. {*l* @ ''*-init-type*''})

locset-definition *hs-out-of-sync-locs* =
 (⋃ *l* ∈ *hs-prefixes*. {*l* @ ''*-init-muts*''})

locset-definition *hs-mut-in-muts-locs* =
 (⋃ *l* ∈ *hs-prefixes*. {*l* @ ''*-init-loop-set-pending*'', *l* @ ''*-init-loop-done*''})

locset-definition *hs-init-loop-done-locs* =
 (⋃ *l* ∈ *hs-prefixes*. {*l* @ ''*-init-loop-done*''})

locset-definition *hs-init-loop-not-done-locs* =
 (*hs-init-loop-locs* − (⋃ *l* ∈ *hs-prefixes*. {*l* @ ''*-init-loop-done*''}))

definition *handshake-invL* :: ('*field*, '*mut*, '*payload*, '*ref*) *gc-pred* **where**

[*inv*]: *handshake-invL* =
  (*atS-gc hs-noop-locs* (*sys-hs-type* = ⟨*ht-NOOP*⟩)
  ∧ *atS-gc hs-get-roots-locs* (*sys-hs-type* = ⟨*ht-GetRoots*⟩)
  ∧ *atS-gc hs-get-work-locs* (*sys-hs-type* = ⟨*ht-GetWork*⟩)
  ∧ *atS-gc hs-mut-in-muts-locs* (*gc-mut* ∈ *gc-muts*)
  ∧ *atS-gc hs-init-loop-locs* (∀ *m*. ¬⟨*m*⟩ ∈ *gc-muts* ⟶ *sys-hs-pending m*
      ∨ *sys-ghost-hs-in-sync m*)
  ∧ *atS-gc hs-init-loop-not-done-locs* (∀ *m*. ⟨*m*⟩ ∈ *gc-muts* ⟶ ¬*sys-hs-pending m*
      ∧ ¬*sys-ghost-hs-in-sync m*)
  ∧ *atS-gc hs-init-loop-done-locs* ((*sys-hs-pending* **$** *gc-mut*
      ∨ *sys-ghost-hs-in-sync* **$** *gc-mut*)
      ∧ (∀ *m*. ⟨*m*⟩ ∈ *gc-muts* ∧ ⟨*m*⟩ ≠ *gc-mut*
      ⟶ ¬*sys-hs-pending m*
      ∧ ¬*sys-ghost-hs-in-sync m*))
  ∧ *atS-gc hs-done-locs* (∀ *m*. *sys-hs-pending m* ∨ *sys-ghost-hs-in-sync m*)
  ∧ *atS-gc hs-done-loop-locs* (∀ *m*. ¬⟨*m*⟩ ∈ *gc-muts* ⟶ ¬*sys-hs-pending m*)
  ∧ *atS-gc hs-none-pending-locs* (∀ *m*. ¬*sys-hs-pending m*)
  ∧ *atS-gc hs-in-sync-locs* (∀ *m*. *sys-ghost-hs-in-sync m*)
  ∧ *atS-gc hs-out-of-sync-locs* (∀ *m*. ¬*sys-hs-pending m*
      ∧ ¬*sys-ghost-hs-in-sync m*)

  ∧ *atS-gc hp-Idle-locs* (*sys-ghost-hs-phase* = ⟨*hp-Idle*⟩)
  ∧ *atS-gc hp-IdleInit-locs* (*sys-ghost-hs-phase* = ⟨*hp-IdleInit*⟩)
  ∧ *atS-gc hp-InitMark-locs* (*sys-ghost-hs-phase* = ⟨*hp-InitMark*⟩)
  ∧ *atS-gc hp-IdleMarkSweep-locs* (*sys-ghost-hs-phase* = ⟨*hp-IdleMarkSweep*⟩)
  ∧ *atS-gc hp-Mark-locs* (*sys-ghost-hs-phase* = ⟨*hp-Mark*⟩))

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

locset-definition *no-pending-phase-locs* :: *location set* **where**
  no-pending-phase-locs =
    (*idle-locs* − { *idle-noop-mfence* })
  ∪ (*init-locs* − { *init-noop-mfence* })
  ∪ (*mark-locs* − { *mark-load-fM*, *mark-store-fA*, *mark-noop-mfence* })

definition *phase-invL* :: (*'field*, *'mut*, *'payload*, *'ref*) *gc-pred* **where**
[*inv*]: *phase-invL* =
  (*atS-gc idle-locs* (*gc-phase* = ⟨*ph-Idle*⟩)
  ∧ *atS-gc init-locs* (*gc-phase* = ⟨*ph-Init*⟩)
  ∧ *atS-gc mark-locs* (*gc-phase* = ⟨*ph-Mark*⟩)
  ∧ *atS-gc sweep-locs* (*gc-phase* = ⟨*ph-Sweep*⟩)
  ∧ *atS-gc no-pending-phase-locs* (*LIST-NULL* (*tso-pending-phase gc*)))

end

Local handshake phase invariant for the mutators.

context *mut-m*
begin

locset-definition *hs-noop-locs* = *prefixed* ''*hs-noop-*''
locset-definition *hs-get-roots-locs* = *prefixed* ''*hs-get-roots-*''
locset-definition *hs-get-work-locs* = *prefixed* ''*hs-get-work-*''
locset-definition *no-pending-mutations-locs* =
    { *hs-load-ht* }
  ∪ (*prefixed* ''*hs-noop*'')
  ∪ (*prefixed* ''*hs-get-roots*'')
  ∪ (*prefixed* ''*hs-get-work*'')
locset-definition *hs-pending-loaded-locs* = (*prefixed* ''*hs-*'' − { *hs-load-pending* })
locset-definition *hs-pending-locs* = (*prefixed* ''*hs-*'' − { *hs-load-pending*, *hs-pending* })

locset-definition *ht-loaded-locs* = (*prefixed* ''*hs-*'' − { *hs-load-pending*, *hs-pending*, *hs-mfence*, *hs-load-ht* })

definition *handshake-invL* :: ('*field*, '*mut*, '*payload*, '*ref*) *gc-pred* **where**
[*inv*]: *handshake-invL* =
(*atS-mut hs-noop-locs* (*sys-hs-type* = ⟨*ht-NOOP*⟩)
∧ *atS-mut hs-get-roots-locs* (*sys-hs-type* = ⟨*ht-GetRoots*⟩)
∧ *atS-mut hs-get-work-locs* (*sys-hs-type* = ⟨*ht-GetWork*⟩)
∧ *atS-mut ht-loaded-locs* (*mut-hs-pending* ⟶ *mut-hs-type* = *sys-hs-type*)
∧ *atS-mut hs-pending-loaded-locs* (*mut-hs-pending* ⟶ *sys-hs-pending m*)
∧ *atS-mut hs-pending-locs* (*mut-hs-pending*)
∧ *atS-mut no-pending-mutations-locs* (*LIST-NULL* (*tso-pending-mutate* (*mutator m*))))

end

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.

context *gc*
begin

locset-definition *fM-eq-locs* = (− { *idle-store-fM*, *idle-flip-noop-mfence* })
locset-definition *fM-tso-empty-locs* = (− { *idle-flip-noop-mfence* })
locset-definition *fA-tso-empty-locs* = (− { *mark-noop-mfence* })

locset-definition
fA-eq-locs = { *idle-load-fM*, *idle-invert-fM* }
∪ *prefixed* ''*idle-noop*''
∪ (*mark-locs* − { *mark-load-fM*, *mark-store-fA*, *mark-noop-mfence* })
∪ *sweep-locs*

locset-definition
fA-neq-locs = { *idle-phase-init*, *idle-store-fM*, *mark-load-fM*, *mark-store-fA* }
∪ *prefixed* ''*idle-flip-noop*''
∪ *init-locs*

definition *fM-fA-invL* :: ('*field*, '*mut*, '*payload*, '*ref*) *gc-pred* **where**
[*inv*]: *fM-fA-invL* =
(*atS-gc fM-eq-locs* (*gc-fM* = *sys-fM*)
∧ *at-gc idle-store-fM* (*gc-fM* ≠ *sys-fM*)
∧ *at-gc idle-flip-noop-mfence* (*sys-fM* ≠ *gc-fM* ⟶ ¬*LIST-NULL* (*tso-pending-fM gc*))
∧ *atS-gc fM-tso-empty-locs* (*LIST-NULL* (*tso-pending-fM gc*))

∧ *atS-gc fA-eq-locs* (*gc-fM* = *sys-fA*)
∧ *atS-gc fA-neq-locs* (*gc-fM* ≠ *sys-fA*)
∧ *at-gc mark-noop-mfence* (*gc-fM* ≠ *sys-fA* ⟶ ¬*LIST-NULL* (*tso-pending-fA gc*))
∧ *atS-gc fA-tso-empty-locs* (*LIST-NULL* (*tso-pending-fA gc*)))

end

5.3 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*, *'payload*, *'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-hs-phase s* ≡ *ghost-hs-phase* (*s p*)
abbreviation (*input*) *p-ghost-honorary-grey s* ≡ *ghost-honorary-grey* (*s p*)
abbreviation (*input*) *p-ghost-hs-in-sync s* ≡ *ghost-hs-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*, *'payload*, *'ref*) *lsts-pred* ⇒ (*'field*, *'mut*, *'payload*, *'ref*) *gc-pred*
where
at-p l' P ≡ *at p* (*l* @ *l'*) ⟶ *LSTP P*

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

abbreviation (*input*) *p-valid-ref* ≡ ¬*NULL p-ref* ∧ *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* ≠ *p-mark* ∨ *p-tso-no-pending-mark*) ⟶ *marked* **$** *p-the-ref*)
∧ (*tso-pending-mark p* ∈ (λ*s*. {[], [*mw-Mark* (*p-the-ref s*) (*p-fM s*)]}))

abbreviation (*input*)
p-mark-inv ≡ ¬*NULL p-mark*
∧ ((λ*s*. *obj-at* (λ*obj*. *Some* (*obj-mark obj*) = *p-mark s*) (*p-the-ref s*) *s*)
∨ *marked* **$** *p-the-ref*)

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

abbreviation (*input*) *p-valid-fM* ≡ *p-fM* = *sys-fM*

abbreviation (*input*)
p-ghg-eq-ref ≡ *p-ghost-honorary-grey* = *pred-singleton* (*the* ∘ *p-ref*)
abbreviation (*input*)
p-ghg-inv ≡ *If p-cas-mark* = *p-mark Then p-ghg-eq-ref Else EMPTY p-ghost-honorary-grey*

definition *mark-object-invL* :: (*'field*, *'mut*, *'payload*, *'ref*) *gc-pred* **where**
mark-object-invL =
(*at-p ''-mo-null''* ⟨*True*⟩
∧ *at-p ''-mo-mark''* (*p-valid-ref*)
∧ *at-p ''-mo-fM''* (*p-valid-ref* ∧ *p-en-cond* (*p-mark-inv*))
∧ *at-p ''-mo-mtest''* (*p-valid-ref* ∧ *p-en-cond* (*p-mark-inv* ∧ *p-valid-fM*))
∧ *at-p ''-mo-phase''* (*p-valid-ref* ∧ *p-mark* ≠ *Some* ∘ *p-fM* ∧ *p-en-cond* (*p-mark-inv* ∧ *p-valid-fM*))
∧ *at-p ''-mo-ptest''* (*p-valid-ref* ∧ *p-mark* ≠ *Some* ∘ *p-fM* ∧ *p-en-cond* (*p-mark-inv* ∧ *p-valid-fM*))
∧ *at-p ''-mo-co-lock''* (*p-valid-ref* ∧ *p-mark-inv* ∧ *p-valid-fM* ∧ *p-mark* ≠ *Some* ∘ *p-fM* ∧ *p-tso-no-pending-mark*)
∧ *at-p ''-mo-co-cmark''* (*p-valid-ref* ∧ *p-mark-inv* ∧ *p-valid-fM* ∧ *p-mark* ≠ *Some* ∘ *p-fM* ∧ *p-tso-no-pending-mark*)
∧ *at-p ''-mo-co-ctest''* (*p-valid-ref* ∧ *p-mark-inv* ∧ *p-valid-fM* ∧ *p-mark* ≠ *Some* ∘ *p-fM* ∧ *p-cas-mark-inv* ∧ *p-tso-no-pending-mark*)
∧ *at-p ''-mo-co-mark''* (*p-cas-mark* = *p-mark* ∧ *p-valid-ref* ∧ *p-valid-fM* ∧ *white* **$** *p-the-ref* ∧ *p-tso-no-pending-mark*

∧ *at-p ''-mo-co-unlock''* (*p-ghg-inv* ∧ *p-valid-ref* ∧ *p-valid-fM* ∧ *p-valid-W-inv*)
∧ *at-p ''-mo-co-won''* (*p-ghg-inv* ∧ *p-valid-ref* ∧ *p-valid-fM* ∧ *marked* **$** *p-the-ref* ∧ *p-tso-no-pending-mutate*)
∧ *at-p ''-mo-co-W''* (*p-ghg-eq-ref* ∧ *p-valid-ref* ∧ *p-valid-fM* ∧ *marked* **$** *p-the-ref* ∧ *p-tso-no-pending-mutate*))

end

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

interpretation *gc-mark*: *mark-object gc gc.mark-loop* ⟨*True*⟩
⟨*proof*⟩

lemmas (**in** *gc*) *gc-mark-mark-object-invL-def2* [*inv*] = *gc-mark.mark-object-invL-def* [*unfolded loc-defs*, *simplified*, *folded loc-defs*]

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

lemmas (**in** *mut-m*) *mut-get-roots-mark-object-invL-def2* [*inv*] = *mut-get-roots.mark-object-invL-def* [*unfolded loc-defs*, *simplified*, *folded loc-defs*]

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-hs-phase m* ≠ ⟨*hp-Idle*⟩ **for** *m*
⟨*proof*⟩

lemmas (**in** *mut-m*) *mut-store-del-mark-object-invL-def2* [*inv*] = *mut-store-del.mark-object-invL-def* [*simplified*, *folded loc-defs*]

interpretation *mut-store-ins*: *mark-object mutator m mut-m.store-ins mut-m.mut-ghost-hs-phase m* ≠ ⟨*hp-Idle*⟩ **for** *m*
⟨*proof*⟩

lemmas (**in** *mut-m*) *mut-store-ins-mark-object-invL-def2* [*inv*] = *mut-store-ins.mark-object-invL-def* [*unfolded loc-defs*, *simplified*, *folded loc-defs*]

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

context *mut-m*
begin

locset-definition *hs-get-roots-loop-locs* = *prefixed ''hs-get-roots-loop''*
locset-definition *hs-get-roots-loop-mo-locs* =
prefixed ''hs-get-roots-loop-mo'' ∪ {*hs-get-roots-loop-done*}

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

locset-definition *hs-not-hp-Idle-locs* =
(⋃*pref*∈*mut-async-mark-object-prefixes*.
⋃*l*∈{*''mo-co-lock''*, *''mo-co-cmark''*, *''mo-co-ctest''*, *''mo-co-mark''*, *''mo-co-unlock''*, *''mo-co-won''*, *''mo-co-W''*}.
{*pref* @ *''-''* @ *l*})

locset-definition *async-mo-ptest-locs* =
(⋃*pref*∈*mut-async-mark-object-prefixes*. {*pref* @ *''-mo-ptest''*})

locset-definition *mo-ptest-locs* =
(⋃*pref*∈*mut-async-mark-object-prefixes*. {*pref* @ *''-mo-ptest''*})

locset-definition *mo-valid-ref-locs* =
(*prefixed ''store-del''* ∪ *prefixed ''store-ins''* ∪ {*deref-del*, *lop-store-ins*})

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 stores pending in TSO buffers: it is key that after the *''init-noop''* handshake there are no pending white insertions (mutations that insert unmarked references). This ensures the deletion barrier does its job.

locset-definition
ghost-honorary-grey-empty-locs =
(− (⋃ *pref*∈{ *''hs-get-roots-loop''*, *''store-del''*, *''store-ins''* }.
⋃ *l*∈{ *''mo-co-unlock''*, *''mo-co-won''*, *''mo-co-W''* }. {*pref* @ *''-''* @ *l*}))

locset-definition
ghost-honorary-root-empty-locs =
(− (*prefixed ''store-del''* ∪ {*lop-store-ins*} ∪ *prefixed ''store-ins''*))

locset-definition *ghost-honorary-root-nonempty-locs* = *prefixed ''store-del''* − {*store-del-mo-null*}
locset-definition *not-idle-locs* = *suffixed ''-mo-ptest''*
locset-definition *ins-barrier-locs* = *prefixed ''store-ins''*
locset-definition *del-barrier1-locs* = *prefixed ''store-del-mo''* ∪ {*lop-store-ins*}

definition *mark-object-invL* :: (*'field*, *'mut*, *'payload*, *'ref*) *gc-pred* **where**
[*inv*]: *mark-object-invL* =
(*atS-mut hs-get-roots-loop-locs* (*mut-refs* ⊆ *mut-roots* ∧ (∀ *r*. ⟨*r*⟩ ∈ *mut-roots* − *mut-refs* ⟶ *marked r*))
∧ *atS-mut hs-get-roots-loop-mo-locs* (¬*NULL mut-ref* ∧ *mut-the-ref* ∈ *mut-roots*)
∧ *at-mut hs-get-roots-loop-done* (*marked* **$** *mut-the-ref*)
∧ *at-mut hs-get-roots-loop-mo-ptest* (*mut-phase* ≠ ⟨*ph-Idle*⟩)
∧ *at-mut hs-get-roots-done* (∀ *r*. ⟨*r*⟩ ∈ *mut-roots* ⟶ *marked r*)

∧ *atS-mut mo-valid-ref-locs* ((¬*NULL mut-new-ref* ⟶ *mut-the-new-ref* ∈ *mut-roots*)
∧ (*mut-tmp-ref* ∈ *mut-roots*))
∧ *at-mut store-del-mo-null* (¬*NULL mut-ref* ⟶ *mut-the-ref* ∈ *mut-ghost-honorary-root*)
∧ *atS-mut ghost-honorary-root-nonempty-locs* (*mut-the-ref* ∈ *mut-ghost-honorary-root*)

∧ *atS-mut not-idle-locs* (*mut-phase* ≠ ⟨*ph-Idle*⟩ ⟶ *mut-ghost-hs-phase* ≠ ⟨*hp-Idle*⟩)
∧ *atS-mut hs-not-hp-Idle-locs* (*mut-ghost-hs-phase* ≠ ⟨*hp-Idle*⟩)

∧ *atS-mut mo-ptest-locs* (*mut-phase* = ⟨*ph-Idle*⟩ ⟶ (*mut-ghost-hs-phase* ∈ ⟨{*hp-Idle*, *hp-IdleInit*}⟩
∨ (*mut-ghost-hs-phase* = ⟨*hp-IdleMarkSweep*⟩
∧ *sys-phase* = ⟨*ph-Idle*⟩)))
∧ *atS-mut ghost-honorary-grey-empty-locs* (*EMPTY mut-ghost-honorary-grey*)
— insertion barrier
∧ *at-mut store-ins* ((*mut-ghost-hs-phase* ∈ ⟨{*hp-InitMark*, *hp-Mark*}⟩
∨ (*mut-ghost-hs-phase* = ⟨*hp-IdleMarkSweep*⟩ ∧ *sys-phase* ≠ ⟨*ph-Idle*⟩))
∧ ¬*NULL mut-new-ref*
⟶ *marked* **$** *mut-the-new-ref*)
∧ *atS-mut ins-barrier-locs* (((*mut-ghost-hs-phase* = ⟨*hp-Mark*⟩
∨ (*mut-ghost-hs-phase* = ⟨*hp-IdleMarkSweep*⟩ ∧ *sys-phase* ≠ ⟨*ph-Idle*⟩))
∧ (λ*s*. ∀ *opt-r'*. ¬*tso-pending-store* (*mutator m*) (*mw-Mutate* (*mut-tmp-ref s*) (*mut-field s*) *opt-r'*) *s*)
⟶ (λ*s*. *obj-at-field-on-heap* (λ*r'*. *marked r' s*) (*mut-tmp-ref s*) (*mut-field s*) *s*))
∧ (*mut-ref* = *mut-new-ref*))

— deletion barrier
∧ *atS-mut del-barrier1-locs* ((*mut-ghost-hs-phase* = ⟨*hp-Mark*⟩
∨ (*mut-ghost-hs-phase* = ⟨*hp-IdleMarkSweep*⟩ ∧ *sys-phase* ≠ ⟨*ph-Idle*⟩))
∧ (λ*s*. ∀ *opt-r'*. ¬*tso-pending-store* (*mutator m*) (*mw-Mutate* (*mut-tmp-ref s*) (*mut-field s*) *opt-r'*) *s*)
⟶ (λ*s*. *obj-at-field-on-heap* (λ*r*. *mut-ref s* = *Some r* ∨ *marked r s*) (*mut-tmp-ref s*) (*mut-field s*) *s*))
∧ *at-mut lop-store-ins* ((*mut-ghost-hs-phase* = ⟨*hp-Mark*⟩
∨ (*mut-ghost-hs-phase* = ⟨*hp-IdleMarkSweep*⟩ ∧ *sys-phase* ≠ ⟨*ph-Idle*⟩))
∧ ¬*NULL mut-ref*
⟶ *marked* $ *mut-the-ref*)
— after *init-noop*. key: no pending white insertions *at-mut hs-noop-done* which we get from *handshake-invL*.
∧ *at-mut mut-load* (*mut-tmp-ref* ∈ *mut-roots*)
∧ *atS-mut ghost-honorary-root-empty-locs* (*EMPTY mut-ghost-honorary-root*))

end

5.4 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*, *'payload*, *'ref*) *lsts-pred* **where**
gc-W-empty-mut-inv =
((*EMPTY sys-W* ∧ *sys-ghost-hs-in-sync m* ∧ ¬*EMPTY* (*WL* (*mutator m*)))
⟶ (∃ *m'*. ¬*sys-ghost-hs-in-sync m'* ∧ ¬*EMPTY* (*WL* (*mutator m'*))))

context *gc*
begin

locset-definition *gc-W-empty-locs* :: *location set* **where**
gc-W-empty-locs =
idle-locs ∪ *init-locs* ∪ *sweep-locs* ∪ {*mark-load-fM*, *mark-store-fA*, *mark-end*}
∪ *prefixed ''mark-noop''*
∪ *prefixed ''mark-loop-get-roots''*
∪ *prefixed ''mark-loop-get-work''*

locset-definition *get-roots-UN-get-work-locs* = *hs-get-roots-locs* ∪ *hs-get-work-locs*
locset-definition *black-heap-locs* = {*sweep-idle*, *idle-noop-mfence*, *idle-noop-init-type*}
locset-definition *no-grey-refs-locs* = *black-heap-locs* ∪ *sweep-locs* ∪ {*mark-end*}

definition *gc-W-empty-invL* :: (*'field*, *'mut*, *'payload*, *'ref*) *gc-pred* **where**
[*inv*]: *gc-W-empty-invL* =
(*atS-gc get-roots-UN-get-work-locs* (∀ *m*. *mut-m.gc-W-empty-mut-inv m*)
∧ *at-gc mark-loop-get-roots-load-W* (*EMPTY sys-W* ⟶ *no-grey-refs*)
∧ *at-gc mark-loop-get-work-load-W* (*EMPTY sys-W* ⟶ *no-grey-refs*)
∧ *at-gc mark-loop* (*EMPTY gc-W* ⟶ *no-grey-refs*)
∧ *atS-gc no-grey-refs-locs* *no-grey-refs*
∧ *atS-gc gc-W-empty-locs* (*EMPTY gc-W*))

end

5.5 Sweep loop invariants

context *gc*

begin

locset-definition *sweep-loop-locs* = *prefixed* ''*sweep-loop*''
locset-definition *sweep-loop-not-choose-ref-locs* = (*prefixed* ''*sweep-loop-*'' − {*sweep-loop-choose-ref*})

definition *sweep-loop-invL* :: ('*field*, '*mut*, '*payload*, '*ref*) *gc-pred* **where**
[*inv*]: *sweep-loop-invL* =
(*at-gc sweep-loop-check* ((¬*NULL gc-mark* ⟶ (λ*s*. *obj-at* (λ*obj*. *Some* (*obj-mark obj*) = *gc-mark s*) (*gc-tmp-ref s*) *s*))
∧ (*NULL gc-mark* ∧ *valid-ref* **$** *gc-tmp-ref* ⟶ *marked* **$** *gc-tmp-ref*))
∧ *at-gc sweep-loop-free* (¬*NULL gc-mark* ∧ *the* ∘ *gc-mark* ≠ *gc-fM* ∧ (λ*s*. *obj-at* (λ*obj*. *Some* (*obj-mark obj*) = *gc-mark s*) (*gc-tmp-ref s*) *s*))
∧ *at-gc sweep-loop-ref-done* (*valid-ref* **$** *gc-tmp-ref* ⟶ *marked* **$** *gc-tmp-ref*)
∧ *atS-gc sweep-loop-locs* (∀ *r*. ¬⟨*r*⟩ ∈ *gc-refs* ∧ *valid-ref r* ⟶ *marked r*)
∧ *atS-gc black-heap-locs* (∀ *r*. *valid-ref r* ⟶ *marked r*)
∧ *atS-gc sweep-loop-not-choose-ref-locs* (*gc-tmp-ref* ∈ *gc-refs*))

For showing that 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 *obj-fields-marked* :: ('*field*, '*mut*, '*payload*, '*ref*) *lsts-pred* **where**
obj-fields-marked =
(∀ *f*. ⟨*f*⟩ ∈ (− *gc-field-set*) ⟶ (λ*s*. *obj-at-field-on-heap* (λ*r*. *marked r s*) (*gc-tmp-ref s*) *f s*))

locset-definition *mark-loop-mo-locs* = *prefixed* ''*mark-loop-mo*''
locset-definition *obj-fields-marked-good-ref-locs* = *mark-loop-mo-locs* ∪ {*mark-loop-mark-field-done*}

locset-definition
ghost-honorary-grey-empty-locs =
(− { *mark-loop-mo-co-unlock*, *mark-loop-mo-co-won*, *mark-loop-mo-co-W* })

locset-definition
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*}
∪ *mark-loop-mo-locs*

definition *obj-fields-marked-invL* :: ('*field*, '*mut*, '*payload*, '*ref*) *gc-pred* **where**
[*inv*]: *obj-fields-marked-invL* =
(*atS-gc obj-fields-marked-locs* (*obj-fields-marked* ∧ *gc-tmp-ref* ∈ *gc-W*)
∧ *atS-gc obj-fields-marked-good-ref-locs* (λ*s*. *obj-at-field-on-heap* (λ*r*. *gc-ref s* = *Some r* ∨ *marked r s*) (*gc-tmp-ref s*) (*gc-field s*) *s*)
∧ *atS-gc mark-loop-mo-locs* (∀ *y*. ¬*NULL gc-ref* ∧ (λ*s*. ((*gc-the-ref s*) *reaches y*) *s*) ⟶ *valid-ref y*)
∧ *at-gc mark-loop-fields* (*gc-tmp-ref* ∈ *gc-W*)
∧ *at-gc mark-loop-mark-field-done* (¬*NULL gc-ref* ⟶ *marked* **$** *gc-the-ref*)
∧ *at-gc mark-loop-blacken* (*EMPTY gc-field-set*)
∧ *atS-gc ghost-honorary-grey-empty-locs* (*EMPTY gc-ghost-honorary-grey*))

end

5.6 The local innvariants collected

definition (**in** *gc*) *invsL* :: ('*field*, '*mut*, '*payload*, '*ref*) *gc-pred* **where**
invsL =
(*fM-fA-invL*
∧ *gc-mark.mark-object-invL*
∧ *gc-W-empty-invL*
∧ *handshake-invL*

∧ *obj-fields-marked-invL*
∧ *phase-invL*
∧ *sweep-loop-invL*
∧ *tso-lock-invL*)

definition (**in** *mut-m*) *invsL* :: ('*field*, '*mut*, '*payload*, '*ref*) *gc-pred* **where**
invsL =
(*mark-object-invL*
∧ *mut-get-roots.mark-object-invL m*
∧ *mut-store-ins.mark-object-invL m*
∧ *mut-store-del.mark-object-invL m*
∧ *handshake-invL*
∧ *tso-lock-invL*)

definition *invsL* :: ('*field*, '*mut*, '*payload*, '*ref*) *gc-pred* **where**
invsL = (*gc.invsL* ∧ (∀ *m*. *mut-m.invsL m*))

6 CIMP specialisation

6.1 Hoare triples

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

definition
valid-proc :: ('*field*, '*mut*, '*payload*, '*ref*) *gc-pred* ⇒ '*mut process-name* ⇒ ('*field*, '*mut*, '*payload*, '*ref*) *gc-pred* ⇒ *bool* (‹⦃-⦄ - ⦃-⦄›)
where
⦃*P*⦄ *p* ⦃*Q*⦄ = (∀ (*c*, *afts*) ∈ *vcg-fragments* (*gc-coms p*). *gc-coms*, *p*, *afts* ⊢ ⦃*P*⦄ *c* ⦃*Q*⦄)

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

lemma *valid-pre*:
assumes ⦃*Q*⦄ *p* ⦃*R*⦄
assumes ⋀*s*. *P s* ⟹ *Q s*
shows ⦃*P*⦄ *p* ⦃*R*⦄
⟨*proof*⟩

lemma *valid-conj-lift*:
assumes *x*: ⦃*P*⦄ *p* ⦃*Q*⦄
assumes *y*: ⦃*P*'⦄ *p* ⦃*Q*'⦄
shows ⦃*P* ∧ *P*'⦄ *p* ⦃*Q* ∧ *Q*'⦄
⟨*proof*⟩

lemma *valid-all-lift*:
assumes ⋀*x*. ⦃*P x*⦄ *p* ⦃*Q x*⦄
shows ⦃λ*s*. ∀ *x*. *P x s*⦄ *p* ⦃λ*s*. ∀ *x*. *Q x s*⦄
⟨*proof*⟩

6.2 Tactics

6.2.1 Model-specific

The following is unfortunately overspecialised to the GC. One might hope for general tactics that work on all CIMP programs.

The system responds to all requests. The schematic variable is instantiated with the semantics of the responses. Thanks to Thomas Sewell for the hackery.

schematic-goal *system-responds-actionE*:
$\llbracket$ ($\{\!|l|\!\}$ *Response action*, *afts*) $\in$ *fragments* (*gc-coms p*) {}; $v \in$ *action x s*;
$\llbracket$ *p* = *sys*; *?P* $\rrbracket \Longrightarrow Q \rrbracket \Longrightarrow Q$
⟨*proof*⟩

schematic-goal *system-responds-action-caseE*:
$\llbracket$ ($\{\!|l|\!\}$ *Response action*, *afts*) $\in$ *fragments* (*gc-coms p*) {}; $v \in$ *action* (*pname*, *req*) *s*;
$\llbracket$ *p* = *sys*; *case-request-op ?P1 ?P2 ?P3 ?P4 ?P5 ?P6 ?P7 ?P8 ?P9 ?P10 ?P11 ?P12 ?P13 ?P14 req* $\rrbracket \Longrightarrow$ $Q \rrbracket \Longrightarrow Q$
⟨*proof*⟩

schematic-goal *system-responds-action-specE*:
$\llbracket$ ($\{\!|l|\!\}$ *Response action*, *afts*) $\in$ *fragments* (*gc-coms p*) {}; $v \in$ *action x s*;
$\llbracket$ *p* = *sys*; *case-request-op ?P1 ?P2 ?P3 ?P4 ?P5 ?P6 ?P7 ?P8 ?P9 ?P10 ?P11 ?P12 ?P13 ?P14* (*snd x*) $\rrbracket$ $\Longrightarrow Q \rrbracket \Longrightarrow Q$
⟨*proof*⟩

6.2.2 Locations

lemma *atS-dests*:
$\llbracket$ *atS p ls s*; *atS p ls' s* $\rrbracket \Longrightarrow$ *atS p* (*ls* $\cup$ *ls'*) *s*
$\llbracket$ $\neg$*atS p ls s*; $\neg$*atS p ls' s* $\rrbracket \Longrightarrow \neg$*atS p* (*ls* $\cup$ *ls'*) *s*
$\llbracket$ $\neg$*atS p ls s*; *atS p ls' s* $\rrbracket \Longrightarrow$ *atS p* (*ls'* $-$ *ls*) *s*
$\llbracket$ $\neg$*atS p ls s*; *at p l s* $\rrbracket \Longrightarrow$ *atS p* ({*l*} $-$ *ls*) *s*
⟨*proof*⟩

lemma *schematic-prem*: $\llbracket Q \Longrightarrow P;\ Q\rrbracket \Longrightarrow P$
⟨*proof*⟩

lemma *TrueE*: $\llbracket$*True*; $P\rrbracket \Longrightarrow P$
⟨*proof*⟩

lemma *thin-locs-pre-discardE*:
$\llbracket$*at p l' s* $\longrightarrow P$; *at p l s*; $l' \neq l$; $Q\rrbracket \Longrightarrow Q$
$\llbracket$*atS p ls s* $\longrightarrow P$; *at p l s*; $l \notin ls$; $Q\rrbracket \Longrightarrow Q$
⟨*proof*⟩

lemma *thin-locs-pre-keep-atE*:
$\llbracket$*at p l s* $\longrightarrow P$; *at p l s*; $P \Longrightarrow Q\rrbracket \Longrightarrow Q$
⟨*proof*⟩

lemma *thin-locs-pre-keep-atSE*:
$\llbracket$*atS p ls s* $\longrightarrow P$; *at p l s*; $l \in ls$; $P \Longrightarrow Q\rrbracket \Longrightarrow Q$
⟨*proof*⟩

lemma *thin-locs-post-discardE*:
$\llbracket$*AT s'* = (*AT s*)(*p* := *lfn*, *q* := *lfn'*); $l' \notin lfn$; $p \neq q\rrbracket \Longrightarrow$ *at p l' s'* $\longrightarrow P$
$\llbracket$*AT s'* = (*AT s*)(*p* := *lfn*); $l' \notin lfn\rrbracket \Longrightarrow$ *at p l' s'* $\longrightarrow P$
$\llbracket$*AT s'* = (*AT s*)(*p* := *lfn*, *q* := *lfn'*); $\bigwedge l.\ l \in lfn \Longrightarrow l \notin ls$; $p \neq q\rrbracket \Longrightarrow$ *atS p ls s'* $\longrightarrow P$
$\llbracket$*AT s'* = (*AT s*)(*p* := *lfn*); $\bigwedge l.\ l \in lfn \Longrightarrow l \notin ls\rrbracket \Longrightarrow$ *atS p ls s'* $\longrightarrow P$
⟨*proof*⟩

lemmas *thin-locs-post-discard-conjE* =
conjI[*OF thin-locs-post-discardE*(*1*)]
conjI[*OF thin-locs-post-discardE*(*2*)]
conjI[*OF thin-locs-post-discardE*(*3*)]

$conjI[OF\ thin\text{-}locs\text{-}post\text{-}discardE(4)]$

lemma *thin-locs-post-keep-locsE*:
$[\![(L \longrightarrow P) \wedge R;\ R \Longrightarrow Q]\!] \Longrightarrow (L \longrightarrow P) \wedge Q$
$L \longrightarrow P \Longrightarrow L \longrightarrow P$
⟨*proof*⟩

lemma *thin-locs-post-keepE*:
$[\![P \wedge R;\ R \Longrightarrow Q]\!] \Longrightarrow (L \longrightarrow P) \wedge Q$
$P \Longrightarrow L \longrightarrow P$
⟨*proof*⟩

lemma *ni-thin-locs-discardE*:
$[\![at\ proc\ l\ s \longrightarrow P;\ AT\ s' = (AT\ s)(p := lfn,\ q := lfn');\ at\ proc\ l'\ s';\ l \neq l';\ proc \neq p;\ proc \neq q;\ Q]\!] \Longrightarrow Q$
$[\![at\ proc\ l\ s \longrightarrow P;\ AT\ s' = (AT\ s)(p := lfn);\ at\ proc\ l'\ s';\ l \neq l';\ proc \neq p;\ Q]\!] \Longrightarrow Q$
$[\![atS\ proc\ ls\ s \longrightarrow P;\ AT\ s' = (AT\ s)(p := lfn,\ q := lfn');\ at\ proc\ l'\ s';\ l' \notin ls;\ proc \neq p;\ proc \neq q;\ Q]\!] \Longrightarrow Q$
$[\![atS\ proc\ ls\ s \longrightarrow P;\ AT\ s' = (AT\ s)(p := lfn);\ at\ proc\ l'\ s';\ l' \notin ls;\ proc \neq p;\ Q]\!] \Longrightarrow Q$

$[\![at\ proc\ l\ s \longrightarrow P;\ AT\ s' = (AT\ s)(p := lfn,\ q := lfn');\ atS\ proc\ ls'\ s';\ l \notin ls';\ proc \neq p;\ proc \neq q;\ Q]\!] \Longrightarrow Q$
$[\![at\ proc\ l\ s \longrightarrow P;\ AT\ s' = (AT\ s)(p := lfn);\ atS\ proc\ ls'\ s';\ l \notin ls';\ proc \neq p;\ Q]\!] \Longrightarrow Q$

⟨*proof*⟩

lemma *ni-thin-locs-keep-atE*:
$[\![at\ proc\ l\ s \longrightarrow P;\ AT\ s' = (AT\ s)(p := lfn,\ q := lfn');\ at\ proc\ l\ s';\ proc \neq p;\ proc \neq q;\ P \Longrightarrow Q]\!] \Longrightarrow Q$
$[\![at\ proc\ l\ s \longrightarrow P;\ AT\ s' = (AT\ s)(p := lfn);\ at\ proc\ l\ s';\ proc \neq p;\ P \Longrightarrow Q]\!] \Longrightarrow Q$
⟨*proof*⟩

lemma *ni-thin-locs-keep-atSE*:
$[\![atS\ proc\ ls\ s \longrightarrow P;\ AT\ s' = (AT\ s)(p := lfn,\ q := lfn');\ at\ proc\ l'\ s';\ l' \in ls;\ proc \neq p;\ proc \neq q;\ P \Longrightarrow Q]\!] \Longrightarrow Q$
$[\![atS\ proc\ ls\ s \longrightarrow P;\ AT\ s' = (AT\ s)(p := lfn);\ at\ proc\ l'\ s';\ l' \in ls;\ proc \neq p;\ P \Longrightarrow Q]\!] \Longrightarrow Q$
$[\![atS\ proc\ ls\ s \longrightarrow P;\ AT\ s' = (AT\ s)(p := lfn,\ q := lfn');\ atS\ proc\ ls'\ s';\ ls' \subseteq ls;\ proc \neq p;\ proc \neq q;\ P \Longrightarrow Q]\!] \Longrightarrow Q$
$[\![atS\ proc\ ls\ s \longrightarrow P;\ AT\ s' = (AT\ s)(p := lfn);\ atS\ proc\ ls'\ s';\ ls' \subseteq ls;\ proc \neq p;\ P \Longrightarrow Q]\!] \Longrightarrow Q$
⟨*proof*⟩

lemma *loc-mem-tac-intros*:
$[\![c \notin A;\ c \notin B]\!] \Longrightarrow c \notin A \cup B$
$c \neq d \Longrightarrow c \notin \{d\}$
$c \notin A \Longrightarrow c \in -\ A$
$c \in A \Longrightarrow c \notin -\ A$
$A \subseteq A$
⟨*proof*⟩

lemmas *loc-mem-tac-elims* =
singletonE
UnE

lemmas *loc-mem-tac-simps* =
append.simps list.simps rev.simps — evaluate string equality
char.inject cong-exp-iff-simps — evaluate character equality
prefix-code suffix-to-prefix
simp-thms

Eq-FalseI
not-Cons-self

lemmas *vcg-fragments′-simps* =
valid-proc-def gc-coms.simps vcg-fragments′.simps atC.simps
ball-Un bool-simps if-False if-True

lemmas *vcg-sem-simps* =
lconst.simps
simp-thms
True-implies-equals
prod.simps fst-conv snd-conv
gc-phase.simps process-name.simps hs-type.simps hs-phase.simps
mem-store-action.simps mem-load-action.simps request-op.simps response.simps

lemmas *vcg-inv-simps* =
simp-thms

⟨*ML*⟩

7 Global invariants lemma bucket

declare *mut-m.mutator-phase-inv-aux.simps*[*simp*]
case-of-simps *mutator-phase-inv-aux-case*: *mut-m.mutator-phase-inv-aux.simps*
case-of-simps *sys-phase-inv-aux-case*: *sys-phase-inv-aux.simps*

7.1 TSO invariants

lemma *tso-store-inv-eq-imp*:
eq-imp (*λp s. mem-store-buffers* (*s sys*) *p*)
tso-store-inv
⟨*proof*⟩

lemmas *tso-store-inv-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF tso-store-inv-eq-imp*, *simplified eq-imp-simps*, *rule-format*]

lemma *tso-store-invD*[*simp*]:
tso-store-inv s ⟹ ¬*sys-mem-store-buffers gc s* = *mw-Mutate r f r′* # *ws*
tso-store-inv s ⟹ ¬*sys-mem-store-buffers gc s* = *mw-Mutate-Payload r f pl* # *ws*
tso-store-inv s ⟹ ¬*sys-mem-store-buffers* (*mutator m*) *s* = *mw-fA fl* # *ws*
tso-store-inv s ⟹ ¬*sys-mem-store-buffers* (*mutator m*) *s* = *mw-fM fl* # *ws*
tso-store-inv s ⟹ ¬*sys-mem-store-buffers* (*mutator m*) *s* = *mw-Phase ph* # *ws*
⟨*proof*⟩

lemma *mut-do-store-action*[*simp*]:
⟦ *sys-mem-store-buffers* (*mutator m*) *s* = *w* # *ws*; *tso-store-inv s* ⟧ ⟹ *fA* (*do-store-action w* (*s sys*)) = *sys-fA s*
⟦ *sys-mem-store-buffers* (*mutator m*) *s* = *w* # *ws*; *tso-store-inv s* ⟧ ⟹ *fM* (*do-store-action w* (*s sys*)) = *sys-fM s*
⟦ *sys-mem-store-buffers* (*mutator m*) *s* = *w* # *ws*; *tso-store-inv s* ⟧ ⟹ *phase* (*do-store-action w* (*s sys*)) = *sys-phase s*
⟨*proof*⟩

lemma *tso-store-inv-sys-load-Mut*[*simp*]:
assumes *tso-store-inv s*
assumes (*ract*, *v*) ∈ { (*mr-fM*, *mv-Mark* (*Some* (*sys-fM s*))), (*mr-fA*, *mv-Mark* (*Some* (*sys-fA s*))), (*mr-Phase*, *mv-Phase* (*sys-phase s*)) }
shows *sys-load* (*mutator m*) *ract* (*s sys*) = *v*
⟨*proof*⟩

lemma *tso-store-inv-sys-load-GC*[*simp*]:
  assumes *tso-store-inv s*
  shows *sys-load gc (mr-Ref r f) (s sys) = mv-Ref (Option.bind (sys-heap s r) (λobj. obj-fields obj f))* (**is** *?lhs = mv-Ref ?rhs*)
⟨*proof*⟩

lemma *tso-no-pending-marksD*[*simp*]:
  assumes *tso-pending-mark p s = []*
  shows *sys-load p (mr-Mark r) (s sys) = mv-Mark (map-option obj-mark (sys-heap s r))*
⟨*proof*⟩

lemma *no-pending-phase-sys-load*[*simp*]:
  assumes *tso-pending-phase p s = []*
  shows *sys-load p mr-Phase (s sys) = mv-Phase (sys-phase s)*
⟨*proof*⟩

lemma *gc-no-pending-fM-write*[*simp*]:
  assumes *tso-pending-fM gc s = []*
  shows *sys-load gc mr-fM (s sys) = mv-Mark (Some (sys-fM s))*
⟨*proof*⟩

lemma *tso-store-refs-simps*[*simp*]:
  mut-m.tso-store-refs m (s(mutator m' := s (mutator m')(|roots := roots'|)))
  = mut-m.tso-store-refs m s
  mut-m.tso-store-refs m (s(mutator m' := s (mutator m')(|ghost-honorary-root := {}|),
  sys := s sys(|mem-store-buffers := (mem-store-buffers (s sys))(mutator m' := sys-mem-store-buffers (mutator m') s @ [mw-Mutate r f opt-r'])|)))
  = mut-m.tso-store-refs m s ∪ (if m' = m then store-refs (mw-Mutate r f opt-r') else {})
  mut-m.tso-store-refs m (s(sys := s sys(|mem-store-buffers := (mem-store-buffers (s sys))(mutator m' := sys-mem-store-b
  (mutator m') s @ [mw-Mutate-Payload r f pl])|)))
  = mut-m.tso-store-refs m s ∪ (if m' = m then store-refs (mw-Mutate-Payload r f pl) else {})
  mut-m.tso-store-refs m (s(sys := s sys(|heap := (sys-heap s)(r' := None)|)))
  = mut-m.tso-store-refs m s
  mut-m.tso-store-refs m (s(mutator m' := s (mutator m')(|roots := insert r (roots (s (mutator m')))|), sys := s sys(|heap := (sys-heap s)(r ↦ obj)|)))
  = mut-m.tso-store-refs m s
  mut-m.tso-store-refs m (s(mutator m' := s (mutator m')(|ghost-honorary-root := Option.set-option opt-r', ref := opt-r'|)))
  = mut-m.tso-store-refs m s
  mut-m.tso-store-refs m (s(sys := s sys(|heap := (sys-heap s)(r := map-option (λobj. obj(|obj-fields := (obj-fields obj)(f := opt-r')|)) (sys-heap s r)),
  mem-store-buffers := (mem-store-buffers (s sys))(p := ws)|)))
  = (if p = mutator m then ⋃ w ∈ set ws. store-refs w else mut-m.tso-store-refs m s)
  mut-m.tso-store-refs m (s(sys := s sys(|heap := (sys-heap s)(r := map-option (λobj. obj(|obj-payload := (obj-payload obj)(f := pl)|)) (sys-heap s r)),
  mem-store-buffers := (mem-store-buffers (s sys))(p := ws)|)))
  = (if p = mutator m then ⋃ w ∈ set ws. store-refs w else mut-m.tso-store-refs m s)
  sys-mem-store-buffers p s = mw-Mark r fl # ws
⟹ mut-m.tso-store-refs m (s(sys := s sys(|heap := (sys-heap s)(r := map-option (obj-mark-update (λ-. fl)) (sys-heap s r)), mem-store-buffers := (mem-store-buffers (s sys))(p := ws)|)))
  = mut-m.tso-store-refs m s
⟨*proof*⟩

lemma *fold-stores-points-to*[*rule-format, simplified conj-explode*]:
  heap (fold-stores ws (s sys)) r = Some obj ∧ obj-fields obj f = Some r'
    ⟶ (r points-to r') s ∨ (∃ w ∈ set ws. r' ∈ store-refs w) (**is** *?P (fold-stores ws) obj*)
⟨*proof*⟩

lemma *points-to-Mutate*:
$(x\ \mathit{points\text{-}to}\ y)$
$(s(\mathit{sys} := (s\ \mathit{sys})(\!|\ \mathit{heap} := (\mathit{sys\text{-}heap}\ s)(r := \mathit{map\text{-}option}\ (\lambda \mathit{obj}.\ \mathit{obj}(\!|\mathit{obj\text{-}fields} := (\mathit{obj\text{-}fields}\ \mathit{obj})(f := \mathit{opt\text{-}r}')|\!))\ (\mathit{sys\text{-}heap}\ s\ r)),$
$\mathit{mem\text{-}store\text{-}buffers} := (\mathit{mem\text{-}store\text{-}buffers}\ (s\ \mathit{sys}))(p := \mathit{ws})\ |\!)))$
$\longleftrightarrow (r \neq x \wedge (x\ \mathit{points\text{-}to}\ y)\ s) \vee (r = x \wedge \mathit{valid\text{-}ref}\ r\ s \wedge (\mathit{opt\text{-}r}' = \mathit{Some}\ y \vee (\ (x\ \mathit{points\text{-}to}\ y)\ s \wedge \mathit{obj\text{-}at}\ (\lambda \mathit{obj}.\ \exists f'.\ \mathit{obj\text{-}fields}\ \mathit{obj}\ f' = \mathit{Some}\ y \wedge f \neq f')\ r\ s)))$
⟨*proof*⟩

7.2 FIXME mutator handshake facts

lemma — Sanity
$\mathit{hp}' = \mathit{hs\text{-}step}\ \mathit{hp} \Longrightarrow \exists\, \mathit{in}'\ \mathit{ht}.\ (\mathit{in}', \mathit{ht}, \mathit{hp}', \mathit{hp}) \in \mathit{hp\text{-}step\text{-}rel}$
⟨*proof*⟩

lemma — Sanity
$(\mathit{False}, \mathit{ht}, \mathit{hp}', \mathit{hp}) \in \mathit{hp\text{-}step\text{-}rel} \Longrightarrow \mathit{hp}' = \mathit{hp\text{-}step}\ \mathit{ht}\ \mathit{hp}$
⟨*proof*⟩

lemma (**in** *mut-m*) *handshake-phase-invD*:
assumes *handshake-phase-inv s*
shows $(\mathit{sys\text{-}ghost\text{-}hs\text{-}in\text{-}sync}\ m\ s, \mathit{sys\text{-}hs\text{-}type}\ s, \mathit{sys\text{-}ghost\text{-}hs\text{-}phase}\ s, \mathit{mut\text{-}ghost\text{-}hs\text{-}phase}\ s) \in \mathit{hp\text{-}step\text{-}rel}$
$\wedge (\mathit{sys\text{-}hs\text{-}pending}\ m\ s \longrightarrow \neg \mathit{sys\text{-}ghost\text{-}hs\text{-}in\text{-}sync}\ m\ s)$
⟨*proof*⟩

lemma *handshake-in-syncD*:
$[\![\ \mathit{All}\ (\mathit{ghost\text{-}hs\text{-}in\text{-}sync}\ (s\ \mathit{sys}));\ \mathit{handshake\text{-}phase\text{-}inv}\ s\ ]\!]$
$\Longrightarrow \forall\, m'.\ \mathit{mut\text{-}m.mut\text{-}ghost\text{-}hs\text{-}phase}\ m'\ s = \mathit{sys\text{-}ghost\text{-}hs\text{-}phase}\ s$
⟨*proof*⟩

lemmas *fM-rel-invD* = *iffD1*[*OF fun-cong*[*OF fM-rel-inv-def*[*simplified atomize-eq*]]]

Relate *sys-ghost-hs-phase*, *gc-phase*, *sys-phase* and writes to the phase in the GC's TSO buffer.

simps-of-case *handshake-phase-rel-simps*[*simp*]: *handshake-phase-rel-def* (*splits*: *hs-phase.split*)

lemma *phase-rel-invD*:
assumes *phase-rel-inv s*
shows $(\forall\, m.\ \mathit{sys\text{-}ghost\text{-}hs\text{-}in\text{-}sync}\ m\ s, \mathit{sys\text{-}ghost\text{-}hs\text{-}phase}\ s, \mathit{gc\text{-}phase}\ s, \mathit{sys\text{-}phase}\ s, \mathit{tso\text{-}pending\text{-}phase}\ \mathit{gc}\ s) \in \mathit{phase\text{-}rel}$
⟨*proof*⟩

lemma *mut-m-not-idle-no-fM-write*:
$[\![\ \mathit{ghost\text{-}hs\text{-}phase}\ (s\ (\mathit{mutator}\ m)) \neq \mathit{hp\text{-}Idle};\ \mathit{fM\text{-}rel\text{-}inv}\ s;\ \mathit{handshake\text{-}phase\text{-}inv}\ s;\ \mathit{tso\text{-}store\text{-}inv}\ s;\ p \neq \mathit{sys}\ ]\!]$
$\Longrightarrow \neg \mathit{sys\text{-}mem\text{-}store\text{-}buffers}\ p\ s = \mathit{mw\text{-}fM}\ \mathit{fl}\ \#\ \mathit{ws}$
⟨*proof*⟩

lemma (**in** *mut-m*) *mut-ghost-handshake-phase-idle*:
$[\![\ \mathit{mut\text{-}ghost\text{-}hs\text{-}phase}\ s = \mathit{hp\text{-}Idle};\ \mathit{handshake\text{-}phase\text{-}inv}\ s;\ \mathit{phase\text{-}rel\text{-}inv}\ s\ ]\!]$
$\Longrightarrow \mathit{sys\text{-}phase}\ s = \mathit{ph\text{-}Idle}$
⟨*proof*⟩

lemma *mut-m-not-idle-no-fM-writeD*:
$[\![\ \mathit{sys\text{-}mem\text{-}store\text{-}buffers}\ p\ s = \mathit{mw\text{-}fM}\ \mathit{fl}\ \#\ \mathit{ws};\ \mathit{ghost\text{-}hs\text{-}phase}\ (s\ (\mathit{mutator}\ m)) \neq \mathit{hp\text{-}Idle};\ \mathit{fM\text{-}rel\text{-}inv}\ s;\ \mathit{handshake\text{-}phase\text{-}inv}\ s;\ \mathit{tso\text{-}store\text{-}inv}\ s;\ p \neq \mathit{sys}\ ]\!]$
$\Longrightarrow \mathit{False}$
⟨*proof*⟩

7.3 points to, reaches, reachable mut

lemma (**in** *mut-m*) *reachable-eq-imp*:
eq-imp $(\lambda r'.\ \textit{mut-roots} \otimes \textit{mut-ghost-honorary-root} \otimes (\lambda s.\ \bigcup(\textit{ran}\ `\ \textit{obj-fields}\ `\ \textit{set-option}\ (\textit{sys-heap}\ s\ r')))$
$\otimes\ \textit{tso-pending-mutate}\ (\textit{mutator}\ m))$
$(\textit{reachable}\ r)$
⟨*proof*⟩

lemmas *reachable-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF mut-m.reachable-eq-imp*, *simplified eq-imp-simps*, *rule-format*]

lemma *reachableI*[*intro*]:
$x \in \textit{mut-m.mut-roots}\ m\ s \Longrightarrow \textit{mut-m.reachable}\ m\ x\ s$
$x \in \textit{mut-m.tso-store-refs}\ m\ s \Longrightarrow \textit{mut-m.reachable}\ m\ x\ s$
⟨*proof*⟩

lemma *reachable-points-to*[*elim*]:
$[\![\ (x\ \textit{points-to}\ y)\ s;\ \textit{mut-m.reachable}\ m\ x\ s\ ]\!] \Longrightarrow \textit{mut-m.reachable}\ m\ y\ s$
⟨*proof*⟩

lemma (**in** *mut-m*) *mut-reachableE*[*consumes 1*, *case-names mut-root tso-store-refs*]:
$[\![\ \textit{reachable}\ y\ s;$
$\bigwedge x.\ [\![\ (x\ \textit{reaches}\ y)\ s;\ x \in \textit{mut-roots}\ s\ ]\!] \Longrightarrow Q;$
$\bigwedge x.\ [\![\ (x\ \textit{reaches}\ y)\ s;\ x \in \textit{mut-ghost-honorary-root}\ s\ ]\!] \Longrightarrow Q;$
$\bigwedge x.\ [\![\ (x\ \textit{reaches}\ y)\ s;\ x \in \textit{tso-store-refs}\ s\ ]\!] \Longrightarrow Q\ ]\!] \Longrightarrow Q$
⟨*proof*⟩

lemma *reachable-induct*[*consumes 1*, *case-names root ghost-honorary-root tso-root reaches*]:
assumes *r*: $\textit{mut-m.reachable}\ m\ y\ s$
assumes *root*: $\bigwedge x.\ [\![\ x \in \textit{mut-m.mut-roots}\ m\ s\ ]\!] \Longrightarrow P\ x$
assumes *ghost-honorary-root*: $\bigwedge x.\ [\![\ x \in \textit{mut-m.mut-ghost-honorary-root}\ m\ s\ ]\!] \Longrightarrow P\ x$
assumes *tso-root*: $\bigwedge x.\ x \in \textit{mut-m.tso-store-refs}\ m\ s \Longrightarrow P\ x$
assumes *reaches*: $\bigwedge x\ y.\ [\![\ \textit{mut-m.reachable}\ m\ x\ s;\ (x\ \textit{points-to}\ y)\ s;\ P\ x\ ]\!] \Longrightarrow P\ y$
shows $P\ y$
⟨*proof*⟩

lemma *mutator-reachable-tso*:
$\textit{sys-mem-store-buffers}\ (\textit{mutator}\ m)\ s = \textit{mw-Mutate}\ r\ f\ \textit{opt-r}'\ \#\ ws$
$\Longrightarrow \textit{mut-m.reachable}\ m\ r\ s \wedge (\forall r'.\ \textit{opt-r}' = \textit{Some}\ r' \longrightarrow \textit{mut-m.reachable}\ m\ r'\ s)$
$\textit{sys-mem-store-buffers}\ (\textit{mutator}\ m)\ s = \textit{mw-Mutate-Payload}\ r\ f\ pl\ \#\ ws$
$\Longrightarrow \textit{mut-m.reachable}\ m\ r\ s$
⟨*proof*⟩

7.4 Colours

lemma *greyI*[*intro*]:
$r \in \textit{ghost-honorary-grey}\ (s\ p) \Longrightarrow \textit{grey}\ r\ s$
$r \in W\ (s\ p) \Longrightarrow \textit{grey}\ r\ s$
$r \in WL\ p\ s \Longrightarrow \textit{grey}\ r\ s$
⟨*proof*⟩

lemma *blackD*[*dest*]:
$\textit{black}\ r\ s \Longrightarrow \textit{marked}\ r\ s$
$\textit{black}\ r\ s \Longrightarrow r \notin WL\ p\ s$
⟨*proof*⟩

lemma *whiteI*[*intro*]:
$\textit{obj-at}\ (\lambda obj.\ \textit{obj-mark}\ obj = (\neg\ \textit{sys-fM}\ s))\ r\ s \Longrightarrow \textit{white}\ r\ s$
⟨*proof*⟩

lemma *marked-not-white*[*dest*]:
  white r s $\Longrightarrow$ $\neg$*marked r s*
⟨*proof*⟩

lemma *white-valid-ref*[*elim*!]:
  white r s $\Longrightarrow$ *valid-ref r s*
⟨*proof*⟩

lemma *not-white-marked*[*elim*!]:
  $[\![\neg$ *white r s*; *valid-ref r s*$]\!]$ $\Longrightarrow$ *marked r s*
⟨*proof*⟩

lemma *black-eq-imp*:
  eq-imp (λ-::*unit*. (λ*s*. *r* $\in$ ($\bigcup$*p*. *WL p s*)) $\otimes$ *sys-fM* $\otimes$ (λ*s*. *map-option obj-mark* (*sys-heap s r*)))
      (*black r*)
⟨*proof*⟩

lemma *grey-eq-imp*:
  eq-imp (λ-::*unit*. (λ*s*. *r* $\in$ ($\bigcup$*p*. *WL p s*)))
      (*grey r*)
⟨*proof*⟩

lemma *white-eq-imp*:
  eq-imp (λ-::*unit*. *sys-fM* $\otimes$ (λ*s*. *map-option obj-mark* (*sys-heap s r*)))
      (*white r*)
⟨*proof*⟩

lemmas *black-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF black-eq-imp*, *simplified eq-imp-simps*, *rule-format*]
lemmas *grey-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF grey-eq-imp*, *simplified eq-imp-simps*, *rule-format*]
lemmas *white-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF white-eq-imp*, *simplified eq-imp-simps*, *rule-format*]

coloured heaps

lemma *black-heap-eq-imp*:
  eq-imp ($\lambda r'$. (λ*s*. $\bigcup$*p*. *WL p s*) $\otimes$ *sys-fM* $\otimes$ (λ*s*. *map-option obj-mark* (*sys-heap s* r')))
      black-heap
⟨*proof*⟩

lemma *white-heap-eq-imp*:
  eq-imp ($\lambda r'$. *sys-fM* $\otimes$ (λ*s*. *map-option obj-mark* (*sys-heap s* r')))
      white-heap
⟨*proof*⟩

lemma *no-black-refs-eq-imp*:
  eq-imp ($\lambda r'$. (λ*s*. ($\bigcup$*p*. *WL p s*)) $\otimes$ *sys-fM* $\otimes$ (λ*s*. *map-option obj-mark* (*sys-heap s* r')))
      no-black-refs
⟨*proof*⟩

lemmas *black-heap-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF black-heap-eq-imp*, *simplified eq-imp-simps*, *rule-format*]
lemmas *white-heap-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF white-heap-eq-imp*, *simplified eq-imp-simps*, *rule-format*]
lemmas *no-black-refs-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF no-black-refs-eq-imp*, *simplified eq-imp-simps*, *rule-format*]

lemma *white-heap-imp-no-black-refs*[*elim*!]:
  white-heap s $\Longrightarrow$ *no-black-refs s*
⟨*proof*⟩

lemma *black-heap-no-greys*[*elim*]:
  $[\![$ *no-grey-refs s*; $\forall$ *r*. *marked r s* $\vee$ $\neg$*valid-ref r s* $]\!]$ $\Longrightarrow$ *black-heap s*
⟨*proof*⟩

lemma *heap-colours-colours*:
black-heap s $\Longrightarrow$ ¬*white r s*
white-heap s $\Longrightarrow$ ¬*black r s*
⟨*proof*⟩

The strong-tricolour invariant

lemma *strong-tricolour-invD*:
⟦ *black x s*; (*x points-to y*) *s*; *valid-ref y s*; *strong-tricolour-inv s* ⟧
$\Longrightarrow$ *marked y s*
⟨*proof*⟩

lemma *no-black-refsD*:
no-black-refs s $\Longrightarrow$ ¬*black r s*
⟨*proof*⟩

lemma *has-white-path-to-induct*[*consumes 1*, *case-names refl step*, *induct set*: *has-white-path-to*]:
assumes (*x has-white-path-to y*) *s*
assumes $\bigwedge$*x*. *P x x*
assumes $\bigwedge$*x y z*. ⟦(*x has-white-path-to y*) *s*; *P x y*; (*y points-to z*) *s*; *white z s*⟧ $\Longrightarrow$ *P x z*
shows *P x y*
⟨*proof*⟩

lemma *has-white-path-toD*[*dest*]:
(*x has-white-path-to y*) *s* $\Longrightarrow$ *white y s* ∨ *x* = *y*
⟨*proof*⟩

lemma *has-white-path-to-refl*[*iff*]:
(*x has-white-path-to x*) *s*
⟨*proof*⟩

lemma *has-white-path-to-step*[*intro*]:
⟦(*x has-white-path-to y*) *s*; (*y points-to z*) *s*; *white z s*⟧ $\Longrightarrow$ (*x has-white-path-to z*) *s*
⟦(*y has-white-path-to z*) *s*; (*x points-to y*) *s*; *white y s*⟧ $\Longrightarrow$ (*x has-white-path-to z*) *s*
⟨*proof*⟩

lemma *has-white-path-toE*[*elim*!]:
⟦ (*x points-to y*) *s*; *white y s* ⟧ $\Longrightarrow$ (*x has-white-path-to y*) *s*
⟨*proof*⟩

lemma *has-white-path-to-reaches*[*elim*]:
(*x has-white-path-to y*) *s* $\Longrightarrow$ (*x reaches y*) *s*
⟨*proof*⟩

lemma *has-white-path-to-blacken*[*simp*]:
(*x has-white-path-to w*) (*s*(*gc* := *s gc*⦇ *W* := *gc-W s* − *rs* ⦈)) $\longleftrightarrow$ (*x has-white-path-to w*) *s*
⟨*proof*⟩

lemma *has-white-path-to-eq-imp'*: — Complicated condition takes care of *alloc*: collapses no object and object with no fields
assumes (*x has-white-path-to y*) *s'*
assumes ∀ *r'*. ⋃(*ran* ‘ *obj-fields* ‘ *set-option* (*sys-heap s' r'*)) = ⋃(*ran* ‘ *obj-fields* ‘ *set-option* (*sys-heap s r'*))
assumes ∀ *r'*. *map-option obj-mark* (*sys-heap s' r'*) = *map-option obj-mark* (*sys-heap s r'*)
assumes *sys-fM s'* = *sys-fM s*
shows (*x has-white-path-to y*) *s*
⟨*proof*⟩

lemma *has-white-path-to-eq-imp*:

eq-imp (*λr′. sys-fM* ⊗ (*λs.* ⋃(*ran* ‘ *obj-fields* ‘ *set-option* (*sys-heap s r′*))) ⊗ (*λs. map-option obj-mark* (*sys-heap s r′*)))
(*x has-white-path-to y*)
⟨*proof*⟩

lemmas *has-white-path-to-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF has-white-path-to-eq-imp*, *simplified eq-imp-simps*, *rule-format*]

grey protects white

lemma *grey-protects-whiteD*[*dest*]:
(*g grey-protects-white w*) *s* ⟹ *grey g s* ∧ (*g* = *w* ∨ *white w s*)
⟨*proof*⟩

lemma *grey-protects-whiteI*[*iff*]:
grey g s ⟹ (*g grey-protects-white g*) *s*
⟨*proof*⟩

lemma *grey-protects-whiteE*[*elim*!]:
⟦ (*g points-to w*) *s*; *grey g s*; *white w s* ⟧ ⟹ (*g grey-protects-white w*) *s*
⟦ (*g grey-protects-white y*) *s*; (*y points-to w*) *s*; *white w s* ⟧ ⟹ (*g grey-protects-white w*) *s*
⟨*proof*⟩

lemma *grey-protects-white-reaches*[*elim*]:
(*g grey-protects-white w*) *s* ⟹ (*g reaches w*) *s*
⟨*proof*⟩

lemma *grey-protects-white-induct*[*consumes 1*, *case-names refl step*, *induct set*: *grey-protects-white*]:
assumes (*g grey-protects-white w*) *s*
assumes ⋀*x*. *grey x s* ⟹ *P x x*
assumes ⋀*x y z*. ⟦(*x has-white-path-to y*) *s*; *P x y*; (*y points-to z*) *s*; *white z s*⟧ ⟹ *P x z*
shows *P g w*
⟨*proof*⟩

7.5 *valid-W-inv*

lemma *valid-W-inv-sys-ghg-empty-iff*[*elim*!]:
valid-W-inv s ⟹ *sys-ghost-honorary-grey s* = {}
⟨*proof*⟩

lemma *WLI*[*intro*]:
r ∈ *W* (*s p*) ⟹ *r* ∈ *WL p s*
r ∈ *ghost-honorary-grey* (*s p*) ⟹ *r* ∈ *WL p s*
⟨*proof*⟩

lemma *WL-eq-imp*:
eq-imp (*λ*(*-::unit*) *s*. (*ghost-honorary-grey* (*s p*), *W* (*s p*)))
(*WL p*)
⟨*proof*⟩

lemmas *WL-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF WL-eq-imp*, *simplified eq-imp-simps*, *rule-format*]

lemma *valid-W-inv-eq-imp*:
eq-imp (*λ*(*p*, *r*). (*λs. W* (*s p*)) ⊗ (*λs. ghost-honorary-grey* (*s p*)) ⊗ *sys-fM* ⊗ (*λs. map-option obj-mark* (*sys-heap s r*)) ⊗ *sys-mem-lock* ⊗ *tso-pending-mark p*)
valid-W-inv
⟨*proof*⟩

lemmas *valid-W-inv-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF valid-W-inv-eq-imp*, *simplified eq-imp-simps*, *rule-format*]

lemma *valid-W-invE*[*elim*!]:
  $[\![r \in W\ (s\ p);\ valid\text{-}W\text{-}inv\ s\ ]\!] \Longrightarrow marked\ r\ s$
  $[\![r \in ghost\text{-}honorary\text{-}grey\ (s\ p);\ sys\text{-}mem\text{-}lock\ s \neq Some\ p;\ valid\text{-}W\text{-}inv\ s\ ]\!] \Longrightarrow marked\ r\ s$
  $[\![r \in W\ (s\ p);\ valid\text{-}W\text{-}inv\ s\ ]\!] \Longrightarrow valid\text{-}ref\ r\ s$
  $[\![r \in ghost\text{-}honorary\text{-}grey\ (s\ p);\ sys\text{-}mem\text{-}lock\ s \neq Some\ p;\ valid\text{-}W\text{-}inv\ s\ ]\!] \Longrightarrow valid\text{-}ref\ r\ s$
  $[\![mw\text{-}Mark\ r\ fl \in set\ (sys\text{-}mem\text{-}store\text{-}buffers\ p\ s);\ valid\text{-}W\text{-}inv\ s\ ]\!] \Longrightarrow r \in ghost\text{-}honorary\text{-}grey\ (s\ p)$
⟨*proof*⟩

lemma *valid-W-invD*:
  $[\![sys\text{-}mem\text{-}store\text{-}buffers\ p\ s = mw\text{-}Mark\ r\ fl\ \#\ ws;\ valid\text{-}W\text{-}inv\ s\ ]\!]$
    $\Longrightarrow fl = sys\text{-}fM\ s \wedge r \in ghost\text{-}honorary\text{-}grey\ (s\ p) \wedge tso\text{-}locked\text{-}by\ p\ s \wedge white\ r\ s \wedge filter\ is\text{-}mw\text{-}Mark\ ws = []$
  $[\![mw\text{-}Mark\ r\ fl \in set\ (sys\text{-}mem\text{-}store\text{-}buffers\ p\ s);\ valid\text{-}W\text{-}inv\ s\ ]\!]$
    $\Longrightarrow fl = sys\text{-}fM\ s \wedge r \in ghost\text{-}honorary\text{-}grey\ (s\ p) \wedge tso\text{-}locked\text{-}by\ p\ s \wedge white\ r\ s \wedge filter\ is\text{-}mw\text{-}Mark\ (sys\text{-}mem\text{-}store\text{-}buffers\ p\ s) = [mw\text{-}Mark\ r\ fl]$
⟨*proof*⟩

lemma *valid-W-inv-colours*:
  $[\![white\ x\ s;\ valid\text{-}W\text{-}inv\ s]\!] \Longrightarrow x \notin W\ (s\ p)$
⟨*proof*⟩

lemma *valid-W-inv-no-mark-stores-invD*:
  $[\![sys\text{-}mem\text{-}lock\ s \neq Some\ p;\ valid\text{-}W\text{-}inv\ s\ ]\!]$
    $\Longrightarrow tso\text{-}pending\ p\ is\text{-}mw\text{-}Mark\ s = []$
⟨*proof*⟩

lemma *valid-W-inv-sys-load*[*simp*]:
  $[\![sys\text{-}mem\text{-}lock\ s \neq Some\ p;\ valid\text{-}W\text{-}inv\ s\ ]\!]$
    $\Longrightarrow sys\text{-}load\ p\ (mr\text{-}Mark\ r)\ (s\ sys) = mv\text{-}Mark\ (map\text{-}option\ obj\text{-}mark\ (sys\text{-}heap\ s\ r))$
⟨*proof*⟩

7.6 *grey-reachable*

lemma *grey-reachable-eq-imp*:
  $eq\text{-}imp\ (\lambda r'.\ (\lambda s.\ \bigcup p.\ WL\ p\ s) \otimes (\lambda s.\ Set.bind\ (Option.set\text{-}option\ (sys\text{-}heap\ s\ r'))\ (ran \circ obj\text{-}fields)))$
    $(grey\text{-}reachable\ r)$
⟨*proof*⟩

lemmas *grey-reachable-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF grey-reachable-eq-imp*, *simplified eq-imp-simps*, *rule-format*]

lemma *grey-reachableI*[*intro*]:
  $grey\ g\ s \Longrightarrow grey\text{-}reachable\ g\ s$
⟨*proof*⟩

lemma *grey-reachableE*:
  $[\![(g\ points\text{-}to\ y)\ s;\ grey\text{-}reachable\ g\ s\ ]\!] \Longrightarrow grey\text{-}reachable\ y\ s$
⟨*proof*⟩

7.7 valid refs inv

lemma *valid-refs-invI*:
  $[\![\bigwedge m\ x\ y.\ [\![(x\ reaches\ y)\ s;\ mut\text{-}m.root\ m\ x\ s \vee grey\ x\ s\ ]\!] \Longrightarrow valid\text{-}ref\ y\ s$
   $]\!] \Longrightarrow valid\text{-}refs\text{-}inv\ s$
⟨*proof*⟩

lemma *valid-refs-inv-eq-imp*:
  $eq\text{-}imp\ (\lambda(m', r').\ (\lambda s.\ roots\ (s\ (mutator\ m'))) \otimes (\lambda s.\ ghost\text{-}honorary\text{-}root\ (s\ (mutator\ m'))) \otimes (\lambda s.\ map\text{-}option\ obj\text{-}fields\ (sys\text{-}heap\ s\ r')) \otimes tso\text{-}pending\text{-}mutate\ (mutator\ m') \otimes (\lambda s.\ \bigcup p.\ WL\ p\ s))$

valid-refs-inv
⟨*proof*⟩

lemmas *valid-refs-inv-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF valid-refs-inv-eq-imp*, *simplified eq-imp-simps*, *rule-format*]

lemma *valid-refs-invD*[*elim*]:
⟦ $x \in$ *mut-m.mut-roots m s*; (*x reaches y*) *s*; *valid-refs-inv s* ⟧ $\Longrightarrow$ *valid-ref y s*
⟦ $x \in$ *mut-m.mut-roots m s*; (*x reaches y*) *s*; *valid-refs-inv s* ⟧ $\Longrightarrow$ $\exists$ *obj. sys-heap s y* = *Some obj*
⟦ $x \in$ *mut-m.tso-store-refs m s*; (*x reaches y*) *s*; *valid-refs-inv s* ⟧ $\Longrightarrow$ *valid-ref y s*
⟦ $x \in$ *mut-m.tso-store-refs m s*; (*x reaches y*) *s*; *valid-refs-inv s* ⟧ $\Longrightarrow$ $\exists$ *obj. sys-heap s y* = *Some obj*
⟦ $w \in$ *set* (*sys-mem-store-buffers* (*mutator m*) *s*); $x \in$ *store-refs w*; (*x reaches y*) *s*; *valid-refs-inv s* ⟧ $\Longrightarrow$ *valid-ref y s*
⟦ $w \in$ *set* (*sys-mem-store-buffers* (*mutator m*) *s*); $x \in$ *store-refs w*; (*x reaches y*) *s*; *valid-refs-inv s* ⟧ $\Longrightarrow$ $\exists$ *obj. sys-heap s y* = *Some obj*
⟦ *grey x s*; (*x reaches y*) *s*; *valid-refs-inv s* ⟧ $\Longrightarrow$ *valid-ref y s*
⟦ *mut-m.reachable m x s*; *valid-refs-inv s* ⟧ $\Longrightarrow$ *valid-ref x s*
⟦ *mut-m.reachable m x s*; *valid-refs-inv s* ⟧ $\Longrightarrow$ $\exists$ *obj. sys-heap s x* = *Some obj*
⟦ $x \in$ *mut-m.mut-ghost-honorary-root m s*; (*x reaches y*) *s*; *valid-refs-inv s* ⟧ $\Longrightarrow$ *valid-ref y s*
⟦ $x \in$ *mut-m.mut-ghost-honorary-root m s*; (*x reaches y*) *s*; *valid-refs-inv s* ⟧ $\Longrightarrow$ $\exists$ *obj. sys-heap s y* = *Some obj*
⟨*proof*⟩

reachable snapshot inv

context *mut-m*
begin

lemma *reachable-snapshot-invI*[*intro*]:
($\bigwedge$*y. reachable y s* $\Longrightarrow$ *in-snapshot y s*) $\Longrightarrow$ *reachable-snapshot-inv s*
⟨*proof*⟩

lemma *reachable-snapshot-inv-eq-imp*:
eq-imp ($\lambda r'$. *mut-roots* $\otimes$ *mut-ghost-honorary-root* $\otimes$ ($\lambda s.\ r' \in (\bigcup p.$ *WL p s*)) $\otimes$ *sys-fM*
$\otimes$ ($\lambda s. \bigcup$(*ran* ‘ *obj-fields* ‘ *set-option* (*sys-heap s r'*))) $\otimes$ ($\lambda s.$ *map-option obj-mark* (*sys-heap s r'*))
$\otimes$ *tso-pending-mutate* (*mutator m*))
reachable-snapshot-inv
⟨*proof*⟩

end

lemmas *reachable-snapshot-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF mut-m.reachable-snapshot-inv-eq-imp*, *simplified eq-imp-simps*, *rule-format*]

lemma *in-snapshotI*[*intro*]:
black r s $\Longrightarrow$ *in-snapshot r s*
grey r s $\Longrightarrow$ *in-snapshot r s*
⟦ *white w s*; (*g grey-protects-white w*) *s* ⟧ $\Longrightarrow$ *in-snapshot w s*
⟨*proof*⟩

lemma — Sanity
in-snapshot r s $\Longrightarrow$ *black r s* $\vee$ *grey r s* $\vee$ *white r s*
⟨*proof*⟩

lemma *in-snapshot-valid-ref*:
⟦ *in-snapshot r s*; *valid-refs-inv s* ⟧ $\Longrightarrow$ *valid-ref r s*
⟨*proof*⟩

lemma *reachableI2*[*intro*]:
$x \in$ *mut-m.mut-ghost-honorary-root m s* $\Longrightarrow$ *mut-m.reachable m x s*
⟨*proof*⟩

lemma *tso-pending-mw-mutate-cong*:
  $[\![$ *filter is-mw-Mutate* (*sys-mem-store-buffers p s*) = *filter is-mw-Mutate* (*sys-mem-store-buffers p s'*);
    $\bigwedge r\, f\, r'.\ P\ r\ f\ r' \longleftrightarrow Q\ r\ f\ r'$ $]\!]$
    $\Longrightarrow (\forall\, r\, f\, r'.$ *mw-Mutate* $r\ f\ r' \in$ *set* (*sys-mem-store-buffers p s*) $\longrightarrow P\ r\ f\ r')$
    $\longleftrightarrow (\forall\, r\, f\, r'.$ *mw-Mutate* $r\ f\ r' \in$ *set* (*sys-mem-store-buffers p s'*) $\longrightarrow Q\ r\ f\ r')$
⟨*proof*⟩

lemma (**in** *mut-m*) *marked-insertions-eq-imp*:
  eq-imp ($\lambda r'$. *sys-fM* $\otimes$ (λs. *map-option obj-mark* (*sys-heap s r'*)) $\otimes$ *tso-pending-mw-mutate* (*mutator m*))
      marked-insertions
⟨*proof*⟩

lemmas *marked-insertions-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF mut-m.marked-insertions-eq-imp*, *simplified eq-imp-simp*
rule-format]

lemma *marked-insertionD*[*elim*!]:
  $[\![$ *sys-mem-store-buffers* (*mutator m*) *s* = *mw-Mutate r f* (*Some r'*) # *ws*; *mut-m.marked-insertions m s* $]\!]$
    $\Longrightarrow$ *marked r' s*
⟨*proof*⟩

lemma *marked-insertions-store-buffer-empty*[*intro*]:
  tso-pending-mutate (*mutator m*) *s* = [] $\Longrightarrow$ *mut-m.marked-insertions m s*
⟨*proof*⟩

lemma (**in** *mut-m*) *marked-deletions-eq-imp*:
  eq-imp ($\lambda r'$. *sys-fM* $\otimes$ (λs. *map-option obj-fields* (*sys-heap s r'*)) $\otimes$ (λs. *map-option obj-mark* (*sys-heap s r'*))
$\otimes$ *tso-pending-mw-mutate* (*mutator m*))
      marked-deletions
⟨*proof*⟩

lemmas *marked-deletions-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF mut-m.marked-deletions-eq-imp*, *simplified eq-imp-simps*,
rule-format]

lemma *marked-deletions-store-buffer-empty*[*intro*]:
  tso-pending-mutate (*mutator m*) *s* = [] $\Longrightarrow$ *mut-m.marked-deletions m s*
⟨*proof*⟩

7.8 Location-specific simplification rules

lemma *obj-at-ref-sweep-loop-free*[*simp*]:
  obj-at P r (*s*(*sys* := (*s sys*)(| *heap* := (*sys-heap s*)(*r'* := *None*)|))) $\longleftrightarrow$ *obj-at P r s* $\wedge\ r \neq r'$
⟨*proof*⟩

lemma *obj-at-alloc*[*simp*]:
  sys-heap s r' = *None*
  $\Longrightarrow$ *obj-at P r* (*s*(*m* := *mut-m-s'*, *sys* := (*s sys*)(| *heap* := (*sys-heap s*)(*r'* $\mapsto$ *obj*) |)))
  $\longleftrightarrow$ (*obj-at P r s* $\vee$ (*r* = *r'* $\wedge$ *P obj*))
⟨*proof*⟩

lemma *valid-ref-valid-null-ref-simps*[*simp*]:
  valid-ref r (*s*(*sys* := *do-store-action w* (*s sys*)(|*mem-store-buffers* := (*mem-store-buffers* (*s sys*))(*p* := *ws*)|))) $\longleftrightarrow$
valid-ref r s
  valid-null-ref r' (*s*(*sys* := *do-store-action w* (*s sys*)(|*mem-store-buffers* := (*mem-store-buffers* (*s sys*))(*p* := *ws*)|)))
$\longleftrightarrow$ *valid-null-ref r' s*
  valid-null-ref r' (*s*(*mutator m* := *mut-s'*, *sys* := (*s sys*)(| *heap* := (*heap* (*s sys*))(*r''* $\mapsto$ *obj*) |))) $\longleftrightarrow$ *valid-null-ref*

$r'\ s \lor r' = Some\ r''$
⟨*proof*⟩

context *mut-m*
begin

lemma *reachable-load*[*simp*]:
assumes *sys-load* (*mutator m*) (*mr-Ref r f*) (*s sys*) = *mv-Ref r′*
assumes $r \in$ *mut-roots s*
shows *mut-m.reachable m′ y* (*s*(*mutator m* := *s* (*mutator m*)⦇ *roots* := *mut-roots s* ∪ *Option.set-option r′* ⦈)) ⟷ *mut-m.reachable m′ y s* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

end

WL

lemma *WL-blacken*[*simp*]:
gc-ghost-honorary-grey s = {}
⟹ *WL p* (*s*(*gc* := *s gc*⦇ *W* := *gc-W s* − *rs* ⦈)) = *WL p s* − { *r* |*r*. *p* = *gc* ∧ *r* ∈ *rs* }
⟨*proof*⟩

lemma *WL-hs-done*[*simp*]:
ghost-honorary-grey (*s* (*mutator m*)) = {}
⟹ *WL p* (*s*(*mutator m* := *s* (*mutator m*)⦇ *W* := {}, *ghost-hs-phase* := *hp′* ⦈,
sys := *s sys*⦇ *hs-pending* := *hsp′*, *W* := *sys-W s* ∪ *W* (*s* (*mutator m*)),
ghost-hs-in-sync := *in′* ⦈))
= (*case p of gc* ⇒ *WL gc s* | *mutator m′* ⇒ (*if m′* = *m then* {} *else WL* (*mutator m′*) *s*) | *sys* ⇒ *WL sys s* ∪ *WL* (*mutator m*) *s*)
ghost-honorary-grey (*s* (*mutator m*)) = {}
⟹ *WL p* (*s*(*mutator m* := *s* (*mutator m*)⦇ *W* := {} ⦈,
sys := *s sys*⦇ *hs-pending* := *hsp′*, *W* := *sys-W s* ∪ *W* (*s* (*mutator m*)),
ghost-hs-in-sync := *in′* ⦈))
= (*case p of gc* ⇒ *WL gc s* | *mutator m′* ⇒ (*if m′* = *m then* {} *else WL* (*mutator m′*) *s*) | *sys* ⇒ *WL sys s* ∪ *WL* (*mutator m*) *s*)
⟨*proof*⟩

lemma *colours-load-W*[*iff*]:
gc-W s = {} ⟹ *black r* (*s*(*gc* := (*s gc*)⦇*W* := *W* (*s sys*)⦈, *sys* := (*s sys*)⦇*W* := {}⦈)) ⟷ *black r s*
gc-W s = {} ⟹ *grey r* (*s*(*gc* := (*s gc*)⦇*W* := *W* (*s sys*)⦈, *sys* := (*s sys*)⦇*W* := {}⦈)) ⟷ *grey r s*
⟨*proof*⟩

lemma *WL-load-W*[*simp*]:
gc-W s = {}
⟹ (*WL p* (*s*(*gc* := (*s gc*)⦇*W* := *sys-W s*⦈, *sys* := (*s sys*)⦇*W* := {}⦈)))
= (*case p of gc* ⇒ *WL gc s* ∪ *sys-W s* | *mutator m* ⇒ *WL* (*mutator m*) *s* | *sys* ⇒ *sys-ghost-honorary-grey s*)
⟨*proof*⟩

no grey refs

lemma *no-grey-refs-eq-imp*:
eq-imp (λ(-::*unit*). (λ*s*. ⋃*p*. *WL p s*))
no-grey-refs
⟨*proof*⟩

lemmas *no-grey-refs-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF no-grey-refs-eq-imp*, *simplified eq-imp-simps*, *rule-format*]

lemma *no-grey-refs-no-pending-marks*:
⟦ *no-grey-refs s*; *valid-W-inv s* ⟧ ⟹ *tso-no-pending-marks s*
⟨*proof*⟩

lemma *no-grey-refs-not-grey-reachableD*:
no-grey-refs s $\Longrightarrow$ $\neg$*grey-reachable x s*
⟨*proof*⟩

lemma *no-grey-refsD*:
no-grey-refs s $\Longrightarrow$ *r* $\notin$ *W* (*s p*)
no-grey-refs s $\Longrightarrow$ *r* $\notin$ *WL p s*
no-grey-refs s $\Longrightarrow$ *r* $\notin$ *ghost-honorary-grey* (*s p*)
⟨*proof*⟩

lemma *no-grey-refs-marked*[*dest*]:
⟦ *marked r s*; *no-grey-refs s* ⟧ $\Longrightarrow$ *black r s*
⟨*proof*⟩

lemma *no-grey-refs-bwD*[*dest*]:
⟦ *heap* (*s sys*) *r* = *Some obj*; *no-grey-refs s* ⟧ $\Longrightarrow$ *black r s* $\vee$ *white r s*
⟨*proof*⟩

context *mut-m*
begin

lemma *reachable-blackD*:
⟦ *no-grey-refs s*; *reachable-snapshot-inv s*; *reachable r s* ⟧ $\Longrightarrow$ *black r s*
⟨*proof*⟩

lemma *no-grey-refs-not-reachable*:
⟦ *no-grey-refs s*; *reachable-snapshot-inv s*; *white r s* ⟧ $\Longrightarrow$ $\neg$*reachable r s*
⟨*proof*⟩

lemma *no-grey-refs-not-rootD*:
⟦ *no-grey-refs s*; *reachable-snapshot-inv s*; *white r s* ⟧
$\Longrightarrow$ *r* $\notin$ *mut-roots s* $\wedge$ *r* $\notin$ *mut-ghost-honorary-root s* $\wedge$ *r* $\notin$ *tso-store-refs s*
⟨*proof*⟩

lemma *reachable-snapshot-inv-white-root*:
⟦ *white w s*; *w* $\in$ *mut-roots s* $\vee$ *w* $\in$ *mut-ghost-honorary-root s*; *reachable-snapshot-inv s* ⟧ $\Longrightarrow$ $\exists$ *g*. (*g grey-protects-white w*) *s*
⟨*proof*⟩

end

lemma *black-dequeue-Mark*[*simp*]:
black b (*s*(*sys* := (*s sys*)⦇ *heap* := (*sys-heap s*)(*r* := *map-option* (*obj-mark-update* (λ-. *fl*)) (*sys-heap s r*)), *mem-store-buffers* := (*mem-store-buffers* (*s sys*))(*p* := *ws*) ⦈))
$\longleftrightarrow$ (*black b s* $\wedge$ *b* $\neq$ *r*) $\vee$ (*b* = *r* $\wedge$ *fl* = *sys-fM s* $\wedge$ *valid-ref r s* $\wedge$ $\neg$*grey r s*)
⟨*proof*⟩

lemma *colours-sweep-loop-free*[*iff*]:
black r (*s*(*sys* := *s sys*⦇*heap* := (*heap* (*s sys*))(*r'* := *None*)⦈)) $\longleftrightarrow$ (*black r s* $\wedge$ *r* $\neq$ *r'*)
grey r (*s*(*sys* := *s sys*⦇*heap* := (*heap* (*s sys*))(*r'* := *None*)⦈)) $\longleftrightarrow$ (*grey r s*)
white r (*s*(*sys* := *s sys*⦇*heap* := (*heap* (*s sys*))(*r'* := *None*)⦈)) $\longleftrightarrow$ (*white r s* $\wedge$ *r* $\neq$ *r'*)
⟨*proof*⟩

lemma *colours-get-work-done*[*simp*]:
black r (*s*(*mutator m* := (*s* (*mutator m*))⦇ *W* := {}⦈),

$sys := (s\ sys)(\!|\ hs\text{-}pending := hp',\ W := W\ (s\ sys) \cup W\ (s\ (mutator\ m)),$
$ghost\text{-}hs\text{-}in\text{-}sync := his'\ |\!)) \longleftrightarrow black\ r\ s$
$grey\ r\ (s(mutator\ m := (s\ (mutator\ m))(\!|\ W := \{\}\ |\!),$
$sys := (s\ sys)(\!|\ hs\text{-}pending := hp',\ W := W\ (s\ sys) \cup W\ (s\ (mutator\ m)),$
$ghost\text{-}hs\text{-}in\text{-}sync := his'\ |\!)) \longleftrightarrow grey\ r\ s$
$white\ r\ (s(mutator\ m := (s\ (mutator\ m))(\!|\ W := \{\}\ |\!),$
$sys := (s\ sys)(\!|\ hs\text{-}pending := hp',\ W := W\ (s\ sys) \cup W\ (s\ (mutator\ m)),$
$ghost\text{-}hs\text{-}in\text{-}sync := his'\ |\!)) \longleftrightarrow white\ r\ s$
⟨*proof*⟩

lemma *colours-get-roots-done*[*simp*]:
$black\ r\ (s(mutator\ m := (s\ (mutator\ m))(\!|\ W := \{\},\ ghost\text{-}hs\text{-}phase := hs'\ |\!),$
$sys := (s\ sys)(\!|\ hs\text{-}pending := hp',\ W := W\ (s\ sys) \cup W\ (s\ (mutator\ m)),$
$ghost\text{-}hs\text{-}in\text{-}sync := his'\ |\!)) \longleftrightarrow black\ r\ s$
$grey\ r\ (s(mutator\ m := (s\ (mutator\ m))(\!|\ W := \{\},\ ghost\text{-}hs\text{-}phase := hs'\ |\!),$
$sys := (s\ sys)(\!|\ hs\text{-}pending := hp',\ W := W\ (s\ sys) \cup W\ (s\ (mutator\ m)),$
$ghost\text{-}hs\text{-}in\text{-}sync := his'\ |\!)) \longleftrightarrow grey\ r\ s$
$white\ r\ (s(mutator\ m := (s\ (mutator\ m))(\!|\ W := \{\},\ ghost\text{-}hs\text{-}phase := hs'\ |\!),$
$sys := (s\ sys)(\!|\ hs\text{-}pending := hp',\ W := W\ (s\ sys) \cup W\ (s\ (mutator\ m)),$
$ghost\text{-}hs\text{-}in\text{-}sync := his'\ |\!)) \longleftrightarrow white\ r\ s$
⟨*proof*⟩

lemma *colours-flip-fM*[*simp*]:
$fl \neq sys\text{-}fM\ s \Longrightarrow black\ b\ (s(sys := (s\ sys)(\!|fM := fl,\ mem\text{-}store\text{-}buffers := (mem\text{-}store\text{-}buffers\ (s\ sys))(p := ws)|\!))) \longleftrightarrow white\ b\ s \wedge \neg grey\ b\ s$
⟨*proof*⟩

lemma *colours-alloc*[*simp*]:
$heap\ (s\ sys)\ r' = None$
$\Longrightarrow black\ r\ (s(mutator\ m := (s\ (mutator\ m))(\!|\ roots := roots'\ |\!),\ sys := (s\ sys)(\!|heap := (heap\ (s\ sys))(r' \mapsto (\!|obj\text{-}mark = fl,\ obj\text{-}fields = Map.empty,\ obj\text{-}payload = Map.empty|\!))|\!)))$
$\longleftrightarrow black\ r\ s \vee (r' = r \wedge fl = sys\text{-}fM\ s \wedge \neg grey\ r'\ s)$
$grey\ r\ (s(mutator\ m := (s\ (mutator\ m))(\!|\ roots := roots'\ |\!),\ sys := (s\ sys)(\!|heap := (heap\ (s\ sys))(r' \mapsto (\!|obj\text{-}mark = fl,\ obj\text{-}fields = Map.empty,\ obj\text{-}payload = Map.empty|\!))|\!)))$
$\longleftrightarrow grey\ r\ s$
$heap\ (s\ sys)\ r' = None$
$\Longrightarrow white\ r\ (s(mutator\ m := (s\ (mutator\ m))(\!|\ roots := roots'\ |\!),\ sys := (s\ sys)(\!|heap := (heap\ (s\ sys))(r' \mapsto (\!|obj\text{-}mark = fl,\ obj\text{-}fields = Map.empty,\ obj\text{-}payload = Map.empty|\!))|\!)))$
$\longleftrightarrow white\ r\ s \vee (r' = r \wedge fl \neq sys\text{-}fM\ s)$
⟨*proof*⟩

lemma *heap-colours-alloc*[*simp*]:
$[\![\ heap\ (s\ sys)\ r' = None;\ valid\text{-}refs\text{-}inv\ s\ ]\!]$
$\Longrightarrow black\text{-}heap\ (s(mutator\ m := s\ (mutator\ m)(\!|roots := roots'|\!),\ sys := s\ sys(\!|heap := (sys\text{-}heap\ s)(r' \mapsto (\!|obj\text{-}mark = fl,\ obj\text{-}fields = Map.empty,\ obj\text{-}payload = Map.empty|\!))|\!)))$
$\longleftrightarrow black\text{-}heap\ s \wedge fl = sys\text{-}fM\ s$
$heap\ (s\ sys)\ r' = None$
$\Longrightarrow white\text{-}heap\ (s(mutator\ m := s\ (mutator\ m)(\!|roots := roots'|\!),\ sys := s\ sys(\!|heap := (sys\text{-}heap\ s)(r' \mapsto (\!|obj\text{-}mark = fl,\ obj\text{-}fields = Map.empty,\ obj\text{-}payload = Map.empty|\!))|\!)))$
$\longleftrightarrow white\text{-}heap\ s \wedge fl \neq sys\text{-}fM\ s$
⟨*proof*⟩

lemma *grey-protects-white-hs-done*[*simp*]:
$(g\ grey\text{-}protects\text{-}white\ w)\ (s(mutator\ m := s\ (mutator\ m)(\!|\ W := \{\},\ ghost\text{-}hs\text{-}phase := hs'\ |\!),$
$sys := s\ sys(\!|\ hs\text{-}pending := hp',\ W := sys\text{-}W\ s \cup W\ (s\ (mutator\ m)),$
$ghost\text{-}hs\text{-}in\text{-}sync := his'\ |\!)))$
$\longleftrightarrow (g\ grey\text{-}protects\text{-}white\ w)\ s$
⟨*proof*⟩

lemma *grey-protects-white-alloc*[*simp*]:
⟦ *fl* = *sys-fM s*; *sys-heap s r* = *None* ⟧
⟹ (*g grey-protects-white w*) (*s*(*mutator m* := *s* (*mutator m*)⦇*roots* := *roots'*⦈, *sys* := *s sys*⦇*heap* := (*sys-heap s*)(*r* ↦ ⦇*obj-mark* = *fl*, *obj-fields* = *Map.empty*, *obj-payload* = *Map.empty*⦈)⦈))
⟷ (*g grey-protects-white w*) *s*
⟨*proof*⟩

lemma (**in** *mut-m*) *reachable-snapshot-inv-sweep-loop-free*:
fixes *s* :: (*'field*, *'mut*, *'payload*, *'ref*) *lsts*
assumes *nmr*: *white r s*
assumes *ngs*: *no-grey-refs s*
assumes *rsi*: *reachable-snapshot-inv s*
shows *reachable-snapshot-inv* (*s*(*sys* := (*s sys*)⦇*heap* := (*heap* (*s sys*))(*r* := *None*)⦈)) (**is** *reachable-snapshot-inv ?s'*)
⟨*proof*⟩

lemma *reachable-alloc*[*simp*]:
assumes *rn*: *sys-heap s r* = *None*
shows *mut-m.reachable m r'* (*s*(*mutator m'* := (*s* (*mutator m'*))⦇*roots* := *insert r* (*roots* (*s* (*mutator m'*)))⦈, *sys* := (*s sys*)⦇*heap* := (*sys-heap s*)(*r* ↦ ⦇*obj-mark* = *fl*, *obj-fields* = *Map.empty*, *obj-payload* = *Map.empty*⦈)⦈))
⟷ *mut-m.reachable m r' s* ∨ (*m'* = *m* ∧ *r'* = *r*) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

context *mut-m*
begin

lemma *reachable-snapshot-inv-alloc*[*simp*, *elim*!]:
fixes *s* :: (*'field*, *'mut*, *'payload*, *'ref*) *lsts*
assumes *rsi*: *reachable-snapshot-inv s*
assumes *rn*: *sys-heap s r* = *None*
assumes *fl*: *fl* = *sys-fM s*
assumes *vri*: *valid-refs-inv s*
shows *reachable-snapshot-inv* (*s*(*mutator m'* := (*s* (*mutator m'*))⦇*roots* := *insert r* (*roots* (*s* (*mutator m'*)))⦈, *sys* := (*s sys*)⦇*heap* := (*sys-heap s*)(*r* ↦ ⦇*obj-mark* = *fl*, *obj-fields* = *Map.empty*, *obj-payload* = *Map.empty*⦈)⦈))
(**is** *reachable-snapshot-inv ?s'*)
⟨*proof*⟩

lemma *reachable-snapshot-inv-discard-roots*[*simp*]:
⟦ *reachable-snapshot-inv s*; *roots'* ⊆ *roots* (*s* (*mutator m*)) ⟧
⟹ *reachable-snapshot-inv* (*s*(*mutator m* := (*s* (*mutator m*))⦇*roots* := *roots'*⦈))
⟨*proof*⟩

lemma *reachable-snapshot-inv-load*[*simp*]:
⟦ *reachable-snapshot-inv s*; *sys-load* (*mutator m*) (*mr-Ref r f*) (*s sys*) = *mv-Ref r'*; *r* ∈ *mut-roots s* ⟧
⟹ *reachable-snapshot-inv* (*s*(*mutator m* := *s* (*mutator m*)⦇ *roots* := *mut-roots s* ∪ *Option.set-option r'* ⦈))
⟨*proof*⟩

lemma *reachable-snapshot-inv-store-ins*[*simp*]:
⟦ *reachable-snapshot-inv s*; *r* ∈ *mut-roots s*; (∃ *r'*. *opt-r'* = *Some r'*) ⟶ *the opt-r'* ∈ *mut-roots s* ⟧
⟹ *reachable-snapshot-inv* (*s*(*mutator m* := *s* (*mutator m*)⦇*ghost-honorary-root* := {}⦈,
sys := *s sys*⦇ *mem-store-buffers* := (*mem-store-buffers* (*s sys*))(*mutator m* := *sys-mem-store-buffers* (*mutator m*) *s* @ [*mw-Mutate r f opt-r'*]) ⦈))
⟨*proof*⟩

end

lemma *WL-mo-co-mark*[*simp*]:

ghost-honorary-grey $(s\ p) = \{\}$
$\Longrightarrow$ *WL* p' $(s(p := s\ p(\!|\ ghost\text{-}honorary\text{-}grey := rs\ |\!))) = WL\ p'\ s \cup \{\ r\ |r.\ p' = p \wedge r \in rs\}$
⟨*proof*⟩

lemma *ghost-honorary-grey-mo-co-mark*[*simp*]:
$[\![\ ghost\text{-}honorary\text{-}grey\ (s\ p) = \{\}\ ]\!] \Longrightarrow black\ b\ (s(p := s\ p(\!|ghost\text{-}honorary\text{-}grey := \{r\}|\!))) \longleftrightarrow black\ b\ s \wedge b \neq r$
$[\![\ ghost\text{-}honorary\text{-}grey\ (s\ p) = \{\}\ ]\!] \Longrightarrow grey\ g\ (s(p := (s\ p)(\!|ghost\text{-}honorary\text{-}grey := \{r\}|\!))) \longleftrightarrow grey\ g\ s \vee g = r$
$[\![\ ghost\text{-}honorary\text{-}grey\ (s\ p) = \{\}\ ]\!] \Longrightarrow white\ w\ (s(p := s\ p(\!|ghost\text{-}honorary\text{-}grey := \{r\}|\!))) \longleftrightarrow white\ w\ s$
⟨*proof*⟩

lemma *ghost-honorary-grey-mo-co-W*[*simp*]:
ghost-honorary-grey $(s\ p') = \{r\}$
$\Longrightarrow (WL\ p\ (s(p' := (s\ p')(\!|\ W := insert\ r\ (W\ (s\ p')),\ ghost\text{-}honorary\text{-}grey := \{\}|\!)))) = (WL\ p\ s)$
ghost-honorary-grey $(s\ p') = \{r\}$
$\Longrightarrow grey\ g\ (s(p' := (s\ p')(\!|\ W := insert\ r\ (W\ (s\ p')),\ ghost\text{-}honorary\text{-}grey := \{\}|\!))) \longleftrightarrow grey\ g\ s$
⟨*proof*⟩

lemma *reachable-sweep-loop-free*:
mut-m.reachable $m\ r\ (s(sys := s\ sys(\!|heap := (sys\text{-}heap\ s)(r' := None)|\!)))$
$\Longrightarrow$ *mut-m.reachable* $m\ r\ s$
⟨*proof*⟩

lemma *reachable-deref-del*[*simp*]:
$[\![\ sys\text{-}load\ (mutator\ m)\ (mr\text{-}Ref\ r\ f)\ (s\ sys) = mv\text{-}Ref\ opt\text{-}r';\ r \in mut\text{-}m.mut\text{-}roots\ m\ s;\ mut\text{-}m.mut\text{-}ghost\text{-}honorary\text{-}root\ m\ s = \{\}\ ]\!]$
$\Longrightarrow mut\text{-}m.reachable\ m'\ y\ (s(mutator\ m := s\ (mutator\ m)(\!|\ ghost\text{-}honorary\text{-}root := Option.set\text{-}option\ opt\text{-}r',\ ref := opt\text{-}r'\ |\!)))$
$\longleftrightarrow mut\text{-}m.reachable\ m'\ y\ s$
⟨*proof*⟩

lemma *no-black-refs-dequeue*[*simp*]:
$[\![\ sys\text{-}mem\text{-}store\text{-}buffers\ p\ s = mw\text{-}Mark\ r\ fl\ \#\ ws;\ no\text{-}black\text{-}refs\ s;\ valid\text{-}W\text{-}inv\ s\ ]\!]$
$\Longrightarrow no\text{-}black\text{-}refs\ (s(sys := s\ sys(\!|heap := (sys\text{-}heap\ s)(r := map\text{-}option\ (obj\text{-}mark\text{-}update\ (\lambda\text{-}.\ fl))\ (sys\text{-}heap\ s\ r)),\ mem\text{-}store\text{-}buffers := (mem\text{-}store\text{-}buffers\ (s\ sys))(p := ws)|\!)))$
$[\![\ sys\text{-}mem\text{-}store\text{-}buffers\ p\ s = mw\text{-}Mutate\ r\ f\ r'\ \#\ ws;\ no\text{-}black\text{-}refs\ s\ ]\!]$
$\Longrightarrow no\text{-}black\text{-}refs\ (s(sys := s\ sys(\!|heap := (sys\text{-}heap\ s)(r := map\text{-}option\ (\lambda obj.\ obj(\!|obj\text{-}fields := (obj\text{-}fields\ obj)(f := r')|\!))\ (sys\text{-}heap\ s\ r)),$
$mem\text{-}store\text{-}buffers := (mem\text{-}store\text{-}buffers\ (s\ sys))(p := ws)|\!)))$
⟨*proof*⟩

lemma *colours-blacken*[*simp*]:
$valid\text{-}W\text{-}inv\ s \Longrightarrow black\ b\ (s(gc := s\ gc(\!|\ W := gc\text{-}W\ s - \{r\}|\!))) \longleftrightarrow black\ b\ s \vee (r \in gc\text{-}W\ s \wedge b = r)$
$[\![\ r \in gc\text{-}W\ s;\ valid\text{-}W\text{-}inv\ s\ ]\!] \Longrightarrow grey\ g\ (s(gc := s\ gc(\!|\ W := gc\text{-}W\ s - \{r\}|\!))) \longleftrightarrow (grey\ g\ s \wedge g \neq r)$

⟨*proof*⟩

lemma *no-black-refs-alloc*[*simp*]:
$[\![\ heap\ (s\ sys)\ r' = None;\ no\text{-}black\text{-}refs\ s\ ]\!]$
$\Longrightarrow no\text{-}black\text{-}refs\ (s(mutator\ m' := s\ (mutator\ m')(\!|roots := roots'|\!),\ sys := s\ sys(\!|heap := (sys\text{-}heap\ s)(r' \mapsto (\!|obj\text{-}mark = fl,\ obj\text{-}fields = Map.empty,\ obj\text{-}payload = Map.empty|\!))|\!)))$
$\longleftrightarrow fl \neq sys\text{-}fM\ s \vee grey\ r'\ s$
⟨*proof*⟩

lemma *no-black-refs-mo-co-mark*[*simp*]:
$[\![\ ghost\text{-}honorary\text{-}grey\ (s\ p) = \{\};\ white\ r\ s\ ]\!]$
$\Longrightarrow no\text{-}black\text{-}refs\ (s(p := s\ p(\!|ghost\text{-}honorary\text{-}grey := \{r\}|\!))) \longleftrightarrow no\text{-}black\text{-}refs\ s$
⟨*proof*⟩

lemma *grey-protects-white-mark*[*simp*]:
  assumes *ghg*: *ghost-honorary-grey* (*s p*) = {}
  shows (∃ *g*. (*g grey-protects-white w*) (*s*(*p* := *s p*⦇ *ghost-honorary-grey* := {*r*} ⦈)))
    ⟷ (∃ *g*′. (*g*′ *grey-protects-white w*) *s*) ∨ (*r has-white-path-to w*) *s* (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *valid-refs-inv-dequeue-Mutate*:
  fixes *s* :: (′*field*, ′*mut*, ′*payload*, ′*ref*) *lsts*
  assumes *vri*: *valid-refs-inv s*
  assumes *sb*: *sys-mem-store-buffers* (*mutator m*′) *s* = *mw-Mutate r f opt-r*′ # *ws*
  shows *valid-refs-inv* (*s*(*sys* := *s sys*⦇*heap* := (*sys-heap s*)(*r* := *map-option* (λ*obj*. *obj*⦇*obj-fields* := (*obj-fields obj*)(*f* := *opt-r*′)⦈) (*sys-heap s r*)),
    mem-store-buffers := (*mem-store-buffers* (*s sys*))(*mutator m*′ := *ws*)⦈)) (**is** *valid-refs-inv ?s*′)
⟨*proof*⟩

lemma *valid-refs-inv-dequeue-Mutate-Payload*:
  notes *if-split-asm*[*split del*]
  fixes *s* :: (′*field*, ′*mut*, ′*payload*, ′*ref*) *lsts*
  assumes *vri*: *valid-refs-inv s*
  assumes *sb*: *sys-mem-store-buffers* (*mutator m*′) *s* = *mw-Mutate-Payload r f pl* # *ws*
  shows *valid-refs-inv* (*s*(*sys* := *s sys*⦇*heap* := (*sys-heap s*)(*r* := *map-option* (λ*obj*. *obj*⦇*obj-payload* := (*obj-payload obj*)(*f* := *pl*)⦈) (*sys-heap s r*)),
    mem-store-buffers := (*mem-store-buffers* (*s sys*))(*mutator m* := *ws*)⦈)) (**is** *valid-refs-inv ?s*′)
⟨*proof*⟩

8 Local invariants lemma bucket

8.1 Location facts

context *mut-m*
begin

lemma *hs-get-roots-loop-locs-subseteq-hs-get-roots-locs*:
  hs-get-roots-loop-locs ⊆ *hs-get-roots-locs*
⟨*proof*⟩

lemma *hs-pending-locs-subseteq-hs-pending-loaded-locs*:
  hs-pending-locs ⊆ *hs-pending-loaded-locs*
⟨*proof*⟩

lemma *ht-loaded-locs-subseteq-hs-pending-loaded-locs*:
  ht-loaded-locs ⊆ *hs-pending-loaded-locs*
⟨*proof*⟩

lemma *hs-noop-locs-subseteq-hs-pending-loaded-locs*:
  hs-noop-locs ⊆ *hs-pending-loaded-locs*
⟨*proof*⟩

lemma *hs-noop-locs-subseteq-hs-pending-locs*:
  hs-noop-locs ⊆ *hs-pending-locs*
⟨*proof*⟩

lemma *hs-noop-locs-subseteq-ht-loaded-locs*:
  hs-noop-locs ⊆ *ht-loaded-locs*
⟨*proof*⟩

lemma *hs-get-roots-locs-subseteq-hs-pending-loaded-locs*:
hs-get-roots-locs ⊆ *hs-pending-loaded-locs*
⟨*proof*⟩

lemma *hs-get-roots-locs-subseteq-hs-pending-locs*:
hs-get-roots-locs ⊆ *hs-pending-locs*
⟨*proof*⟩

lemma *hs-get-roots-locs-subseteq-ht-loaded-locs*:
hs-get-roots-locs ⊆ *ht-loaded-locs*
⟨*proof*⟩

lemma *hs-get-work-locs-subseteq-hs-pending-loaded-locs*:
hs-get-work-locs ⊆ *hs-pending-loaded-locs*
⟨*proof*⟩

lemma *hs-get-work-locs-subseteq-hs-pending-locs*:
hs-get-work-locs ⊆ *hs-pending-locs*
⟨*proof*⟩

lemma *hs-get-work-locs-subseteq-ht-loaded-locs*:
hs-get-work-locs ⊆ *ht-loaded-locs*
⟨*proof*⟩

end

declare
mut-m.hs-get-roots-loop-locs-subseteq-hs-get-roots-locs[*locset-cache*]
mut-m.hs-pending-locs-subseteq-hs-pending-loaded-locs[*locset-cache*]
mut-m.ht-loaded-locs-subseteq-hs-pending-loaded-locs[*locset-cache*]
mut-m.hs-noop-locs-subseteq-hs-pending-loaded-locs[*locset-cache*]
mut-m.hs-noop-locs-subseteq-hs-pending-locs[*locset-cache*]
mut-m.hs-noop-locs-subseteq-ht-loaded-locs[*locset-cache*]
mut-m.hs-get-roots-locs-subseteq-hs-pending-loaded-locs[*locset-cache*]
mut-m.hs-get-roots-locs-subseteq-hs-pending-locs[*locset-cache*]
mut-m.hs-get-roots-locs-subseteq-ht-loaded-locs[*locset-cache*]
mut-m.hs-get-work-locs-subseteq-hs-pending-loaded-locs[*locset-cache*]
mut-m.hs-get-work-locs-subseteq-hs-pending-locs[*locset-cache*]
mut-m.hs-get-work-locs-subseteq-ht-loaded-locs[*locset-cache*]

context *gc*
begin

lemma *get-roots-UN-get-work-locs-subseteq-ghost-honorary-grey-empty-locs*:
get-roots-UN-get-work-locs ⊆ *ghost-honorary-grey-empty-locs*
⟨*proof*⟩

lemma *hs-get-roots-locs-subseteq-hp-IdleMarkSweep-locs*:
hs-get-roots-locs ⊆ *hp-IdleMarkSweep-locs*
⟨*proof*⟩

lemma *hs-get-work-locs-subseteq-hp-IdleMarkSweep-locs*:
hs-get-work-locs ⊆ *hp-IdleMarkSweep-locs*
⟨*proof*⟩

end

declare
gc.get-roots-UN-get-work-locs-subseteq-ghost-honorary-grey-empty-locs[*locset-cache*]
gc.hs-get-roots-locs-subseteq-hp-IdleMarkSweep-locs[*locset-cache*]
gc.hs-get-work-locs-subseteq-hp-IdleMarkSweep-locs[*locset-cache*]

8.2 *obj-fields-marked-inv*

context *gc*
begin

lemma *obj-fields-marked-eq-imp*:
eq-imp ($\lambda r'$. *gc-field-set* ⊗ *gc-tmp-ref* ⊗ (λs. *map-option obj-fields* (*sys-heap s r'*)) ⊗ (λs. *map-option obj-mark* (*sys-heap s r'*)) ⊗ *sys-fM* ⊗ *tso-pending-mutate gc*)
obj-fields-marked
⟨*proof*⟩

lemma *obj-fields-marked-UNIV*[*iff*]:
obj-fields-marked (*s*(*gc* := (*s gc*)⦇ *field-set* := *UNIV* ⦈))
⟨*proof*⟩

lemma *obj-fields-marked-invL-eq-imp*:
eq-imp ($\lambda r'$ *s*. (*AT s gc*, *s*↓ *gc*, *map-option obj-fields* (*sys-heap s*↓ *r'*), *map-option obj-mark* (*sys-heap s*↓ *r'*), *sys-fM s*↓, *sys-W s*↓, *tso-pending-mutate gc s*↓))
obj-fields-marked-invL
⟨*proof*⟩

lemma *obj-fields-marked-mark-field-done*[*iff*]:
⟦ *obj-at-field-on-heap* (λr. *marked r s*) (*gc-tmp-ref s*) (*gc-field s*) *s*; *obj-fields-marked s* ⟧
⟹ *obj-fields-marked* (*s*(*gc* := (*s gc*)⦇*field-set* := *gc-field-set s* − {*gc-field s*}⦈))
⟨*proof*⟩

end

lemmas *gc-obj-fields-marked-inv-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF gc.obj-fields-marked-eq-imp*, *simplified eq-imp-sim
rule-format*]
lemmas *gc-obj-fields-marked-invL-niE*[*nie*] = *iffD1*[*OF gc.obj-fields-marked-invL-eq-imp*[*simplified eq-imp-simps*, *rule-format*, *unfolded conj-explode*], *rotated* −*1*]

8.3 mark object

context *mark-object*
begin

lemma *mark-object-invL-eq-imp*:
eq-imp (λ(-::*unit*) *s*. (*AT s p*, *s*↓ *p*, *sys-heap s*↓, *sys-fM s*↓, *sys-mem-store-buffers p s*↓))
mark-object-invL
⟨*proof*⟩

lemmas *mark-object-invL-niE*[*nie*] =
iffD1[*OF mark-object-invL-eq-imp*[*simplified eq-imp-simps*, *rule-format*, *unfolded conj-explode*], *rotated* −*1*]

end

lemma *mut-m-mark-object-invL-eq-imp*:
eq-imp (λr *s*. (*AT s* (*mutator m*), *s*↓ (*mutator m*), *sys-heap s*↓ *r*, *sys-fM s*↓, *sys-phase s*↓, *tso-pending-mutate* (*mutator m*) *s*↓))
(*mut-m.mark-object-invL m*)
⟨*proof*⟩

lemmas *mut-m-mark-object-invL-niE*[*nie*] =
iffD1[*OF mut-m-mark-object-invL-eq-imp*[*simplified eq-imp-simps*, *rule-format*, *unfolded conj-explode*], *rotated* -1]

9 Initial conditions

context *gc-system*
begin

lemma *init-strong-tricolour-inv*:
⟦ *obj-mark* ‘ *ran* (*sys-heap* (|*GST* = *s*, *HST* = []|)↓) ⊆ {*gc-fM* (|*GST* = *s*, *HST* = []|)↓}; *sys-fM* (|*GST* = *s*, *HST* = []|)↓ = *gc-fM* (|*GST* = *s*, *HST* = []|)↓ ⟧
⟹ *strong-tricolour-inv* (|*GST* = *s*, *HST* = []|)↓
⟨*proof*⟩

lemma *init-no-grey-refs*:
⟦ *gc-W* (|*GST* = *s*, *HST* = []|)↓ = {}; ∀ *m*. *W* ((|*GST* = *s*, *HST* = []|)↓ (*mutator m*)) = {}; *sys-W* (|*GST* = *s*, *HST* = []|)↓ = {};
gc-ghost-honorary-grey (|*GST* = *s*, *HST* = []|)↓ = {}; ∀ *m*. *ghost-honorary-grey* ((|*GST* = *s*, *HST* = []|)↓ (*mutator m*)) = {}; *sys-ghost-honorary-grey* (|*GST* = *s*, *HST* = []|)↓ = {} ⟧
⟹ *no-grey-refs* (|*GST* = *s*, *HST* = []|)↓
⟨*proof*⟩

lemma *valid-refs-imp-valid-refs-inv*:
⟦ *valid-refs s*; *no-grey-refs s*; ∀ *p*. *sys-mem-store-buffers p s* = []; ∀ *m*. *ghost-honorary-root* (*s* (*mutator m*)) = {} ⟧
⟹ *valid-refs-inv s*
⟨*proof*⟩

lemma *no-grey-refs-imp-valid-W-inv*:
⟦ *no-grey-refs s*; ∀ *p*. *sys-mem-store-buffers p s* = [] ⟧
⟹ *valid-W-inv s*
⟨*proof*⟩

lemma *valid-refs-imp-reachable-snapshot-inv*:
⟦ *valid-refs s*; *obj-mark* ‘ *ran* (*sys-heap s*) ⊆ {*sys-fM s*}; ∀ *p*. *sys-mem-store-buffers p s* = []; ∀ *m*. *ghost-honorary-root* (*s* (*mutator m*)) = {} ⟧
⟹ *mut-m.reachable-snapshot-inv m s*
⟨*proof*⟩

lemma *init-inv-sys*: ∀ *s*. *initial-state gc-system s* ⟶ *invs* (|*GST* = *s*, *HST* = []|)↓
⟨*proof*⟩

lemma *init-inv-mut*: ∀ *s*. *initial-state gc-system s* ⟶ *mut-m.invsL m* (|*GST* = *s*, *HST* = []|)
⟨*proof*⟩

lemma *init-inv-gc*: ∀ *s*. *initial-state gc-system s* ⟶ *gc.invsL* (|*GST* = *s*, *HST* = []|)
⟨*proof*⟩

end

definition *I* :: (*'field*, *'mut*, *'payload*, *'ref*) *gc-pred* **where**
I = (*invsL* ∧ *LSTP invs*)

lemmas *I-defs* = *gc.invsL-def mut-m.invsL-def invsL-def invs-def I-def*

context *gc-system*
begin

theorem *init-inv*: ∀ *s*. *initial-state gc-system s* ⟶ *I* ⦇*GST* = *s*, *HST* = []⦈
⟨*proof*⟩

end

10 Noninterference

lemma *mut-del-barrier1-subseteq-mut-mo-valid-ref-locs*[*locset-cache*]:
mut-m.del-barrier1-locs ⊆ *mut-m.mo-valid-ref-locs*
⟨*proof*⟩

lemma *mut-del-barrier2-subseteq-mut-mo-valid-ref*[*locset-cache*]:
mut-m.ins-barrier-locs ⊆ *mut-m.mo-valid-ref-locs*
⟨*proof*⟩

context *gc*
begin

lemma *obj-fields-marked-locs-subseteq-hp-IdleMarkSweep-locs*:
obj-fields-marked-locs ⊆ *hp-IdleMarkSweep-locs*
⟨*proof*⟩

lemma *obj-fields-marked-locs-subseteq-hs-in-sync-locs*:
obj-fields-marked-locs ⊆ *hs-in-sync-locs*
⟨*proof*⟩

lemma *obj-fields-marked-good-ref-subseteq-hp-IdleMarkSweep-locs*:
obj-fields-marked-good-ref-locs ⊆ *hp-IdleMarkSweep-locs*
⟨*proof*⟩

lemma *mark-loop-mo-mark-loop-field-done-subseteq-hs-in-sync-locs*:
obj-fields-marked-good-ref-locs ⊆ *hs-in-sync-locs*
⟨*proof*⟩

lemma *no-grey-refs-locs-subseteq-hs-in-sync-locs*:
no-grey-refs-locs ⊆ *hs-in-sync-locs*
⟨*proof*⟩

lemma *get-roots-UN-get-work-locs-subseteq-gc-W-empty-locs*:
get-roots-UN-get-work-locs ⊆ *gc-W-empty-locs*
⟨*proof*⟩

end

declare
gc.obj-fields-marked-locs-subseteq-hp-IdleMarkSweep-locs[*locset-cache*]
gc.obj-fields-marked-locs-subseteq-hs-in-sync-locs[*locset-cache*]
gc.obj-fields-marked-good-ref-subseteq-hp-IdleMarkSweep-locs[*locset-cache*]
gc.mark-loop-mo-mark-loop-field-done-subseteq-hs-in-sync-locs[*locset-cache*]
gc.no-grey-refs-locs-subseteq-hs-in-sync-locs[*locset-cache*]

gc.get-roots-UN-get-work-locs-subseteq-gc-W-empty-locs[*locset-cache*]

lemma *handshake-obj-fields-markedD*:
⟦ *atS gc gc.obj-fields-marked-locs s*; *gc.handshake-invL s* ⟧ ⟹ *sys-ghost-hs-phase s*↓ = *hp-IdleMarkSweep* ∧ *All* (*ghost-hs-in-sync* (*s*↓ *sys*))
⟨*proof*⟩

lemma *obj-fields-marked-good-ref-locs-hp-phaseD*:
⟦ *atS gc gc.obj-fields-marked-good-ref-locs s*; *gc.handshake-invL s* ⟧
⟹ *sys-ghost-hs-phase s*↓ = *hp-IdleMarkSweep* ∧ *All* (*ghost-hs-in-sync* (*s*↓ *sys*))
⟨*proof*⟩

lemma *gc-marking-reaches-Mutate*:
assumes *xys*: ∀ *y*. (*x reaches y*) *s* ⟶ *valid-ref y s*
assumes *xy*: (*x reaches y*) (*s*(*sys* := *s sys*⦇*heap* := (*sys-heap s*)(*r* := *map-option* (λ*obj*. *obj*⦇*obj-fields* := (*obj-fields obj*)(*f* := *opt-r*′)⦈) (*sys-heap s r*)),
mem-store-buffers := (*mem-store-buffers* (*s sys*))(*p* := *ws*)⦈))
assumes *sb*: *sys-mem-store-buffers* (*mutator m*) *s* = *mw-Mutate r f opt-r*′ # *ws*
assumes *vri*: *valid-refs-inv s*
shows *valid-ref y s*
⟨*proof*⟩

lemma (**in** *sys*) *gc-obj-fields-marked-invL*[*intro*]:
notes *filter-empty-conv*[*simp*]
notes *fun-upd-apply*[*simp*]
shows
⦃ *gc.fM-fA-invL* ∧ *gc.handshake-invL* ∧ *gc.obj-fields-marked-invL*
∧ *LSTP* (*fM-rel-inv* ∧ *handshake-phase-inv* ∧ *mutators-phase-inv* ∧ *tso-store-inv* ∧ *valid-refs-inv* ∧ *valid-W-inv*) ⦄
sys
⦃ *gc.obj-fields-marked-invL* ⦄
⟨*proof*⟩

10.1 The infamous termination argument

lemma (**in** *mut-m*) *gc-W-empty-mut-inv-eq-imp*:
eq-imp (λ*m*′. *sys-W* ⊗ *WL* (*mutator m*′) ⊗ *sys-ghost-hs-in-sync m*′)
gc-W-empty-mut-inv
⟨*proof*⟩

lemmas *gc-W-empty-mut-inv-fun-upd*[*simp*] = *eq-imp-fun-upd*[*OF mut-m.gc-W-empty-mut-inv-eq-imp*, *simplified eq-imp-simps*, *rule-format*]

lemma (**in** *gc*) *gc-W-empty-invL-eq-imp*:
eq-imp (λ(*m*′, *p*) *s*. (*AT s gc*, *s*↓ *gc*, *sys-W s*↓, *WL p s*↓, *sys-ghost-hs-in-sync m*′ *s*↓))
gc-W-empty-invL
⟨*proof*⟩

lemmas *gc-W-empty-invL-niE*[*nie*] =
iffD1[*OF gc.gc-W-empty-invL-eq-imp*[*simplified eq-imp-simps*, *rule-format*, *unfolded conj-explode*, *rule-format*], *rotated* −*1*]

lemma *gc-W-empty-mut-inv-load-W*:
⟦ ∀ *m*. *mut-m.gc-W-empty-mut-inv m s*; ∀ *m*. *sys-ghost-hs-in-sync m s*; *WL gc s* = {}; *WL sys s* = {} ⟧
⟹ *no-grey-refs s*
⟨*proof*⟩

context *gc*

begin

lemma *gc-W-empty-mut-inv-hs-init*[*iff*]:
mut-m.gc-W-empty-mut-inv m (*s*(*sys* := *s sys*(|*hs-type* := *ht*, *ghost-hs-in-sync* := ⟨*False*⟩|)))
mut-m.gc-W-empty-mut-inv m (*s*(*sys* := *s sys*(|*hs-type* := *ht*, *ghost-hs-in-sync* := ⟨*False*⟩, *ghost-hs-phase* := *hp'* |)))
⟨*proof*⟩

lemma *gc-W-empty-invL*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
{| *handshake-invL* ∧ *obj-fields-marked-invL* ∧ *gc-W-empty-invL* ∧ *LSTP valid-W-inv* |}
gc
{| *gc-W-empty-invL* |}
⟨*proof*⟩

end

lemma (**in** *sys*) *gc-gc-W-empty-invL*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
{| *gc.gc-W-empty-invL* |} *sys*
⟨*proof*⟩

lemma *empty-WL-GC*:
⟦ *atS gc gc.get-roots-UN-get-work-locs s*; *gc.obj-fields-marked-invL s* ⟧ ⟹ *gc-ghost-honorary-grey s*↓ = {}
⟨*proof*⟩

lemma *gc-hs-get-roots-get-workD*:
⟦ *atS gc gc.get-roots-UN-get-work-locs s*; *gc.handshake-invL s* ⟧
⟹ *sys-ghost-hs-phase s*↓ = *hp-IdleMarkSweep* ∧ *sys-hs-type s*↓ ∈ {*ht-GetWork*, *ht-GetRoots*}
⟨*proof*⟩

context *gc*
begin

lemma *handshake-sweep-mark-endD*:
⟦ *atS gc no-grey-refs-locs s*; *handshake-invL s*; *handshake-phase-inv s*↓ ⟧
⟹ *mut-m.mut-ghost-hs-phase m s*↓ = *hp-IdleMarkSweep* ∧ *All* (*ghost-hs-in-sync* (*s*↓ *sys*))
⟨*proof*⟩

lemma *gc-W-empty-mut-mo-co-mark*:
⟦ ∀ *x*. *mut-m.gc-W-empty-mut-inv x s*↓; *mutators-phase-inv s*↓;
mut-m.mut-ghost-honorary-grey m s↓ = {};
r ∈ *mut-m.mut-roots m s*↓ ∪ *mut-m.mut-ghost-honorary-root m s*↓; *white r s*↓;
atS gc get-roots-UN-get-work-locs s; *gc.handshake-invL s*; *gc.obj-fields-marked-invL s*;
atS gc gc-W-empty-locs s ⟶ *gc-W s*↓ = {};
handshake-phase-inv s↓; *valid-W-inv s*↓ ⟧
⟹ *mut-m.gc-W-empty-mut-inv m'* (*s*↓(*mutator m* := *s*↓ (*mutator m*)(|*ghost-honorary-grey* := {*r*}|)))
⟨*proof*⟩

lemma *no-grey-refs-mo-co-mark*:
⟦ *mutators-phase-inv s*↓;
no-grey-refs s↓;
gc.handshake-invL s;
at gc mark-loop s ∨ *at gc mark-loop-get-roots-load-W s* ∨ *at gc mark-loop-get-work-load-W s* ∨ *atS gc no-grey-refs-locs s*;

r ∈ *mut-m.mut-roots m s*↓ ∪ *mut-m.mut-ghost-honorary-root m s*↓; *white r s*↓;
handshake-phase-inv s↓ ⟧
⟹ *no-grey-refs* (*s*↓(*mutator m* := *s*↓ (*mutator m*)⦇*ghost-honorary-grey* := {*r*}⦈))
⟨*proof*⟩

end

context *mut-m*
begin

lemma *gc-W-empty-invL*[*intro*]:
notes *gc.gc-W-empty-mut-mo-co-mark*[*simp*]
notes *gc.no-grey-refs-mo-co-mark*[*simp*]
notes *fun-upd-apply*[*simp*]
shows
⦃ *handshake-invL* ∧ *mark-object-invL* ∧ *tso-lock-invL*
∧ *mut-get-roots.mark-object-invL m*
∧ *mut-store-del.mark-object-invL m*
∧ *mut-store-ins.mark-object-invL m*
∧ *gc.handshake-invL* ∧ *gc.obj-fields-marked-invL*
∧ *gc.gc-W-empty-invL*
∧ *LSTP* (*handshake-phase-inv* ∧ *mutators-phase-inv* ∧ *valid-W-inv*) ⦄
mutator m
⦃ *gc.gc-W-empty-invL* ⦄
⟨*proof*⟩

end

context *gc*
begin

lemma *mut-store-old-mark-object-invL*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
⦃ *fM-fA-invL* ∧ *handshake-invL* ∧ *sweep-loop-invL* ∧ *gc-W-empty-invL*
∧ *mut-m.mark-object-invL m*
∧ *mut-store-del.mark-object-invL m*
∧ *LSTP* (*handshake-phase-inv* ∧ *mut-m.mutator-phase-inv m*) ⦄
gc
⦃ *mut-store-del.mark-object-invL m* ⦄
⟨*proof*⟩

lemma *mut-store-ins-mark-object-invL*[*intro*]:
⦃ *fM-fA-invL* ∧ *handshake-invL* ∧ *sweep-loop-invL* ∧ *gc-W-empty-invL*
∧ *mut-m.mark-object-invL m*
∧ *mut-store-ins.mark-object-invL m*
∧ *LSTP* (*handshake-phase-inv* ∧ *mut-m.mutator-phase-inv m*) ⦄
gc
⦃ *mut-store-ins.mark-object-invL m* ⦄
⟨*proof*⟩

lemma *mut-mark-object-invL*[*intro*]:
⦃ *fM-fA-invL* ∧ *gc-W-empty-invL* ∧ *handshake-invL* ∧ *sweep-loop-invL*
∧ *mut-m.handshake-invL m* ∧ *mut-m.mark-object-invL m*
∧ *LSTP* (*fM-rel-inv* ∧ *handshake-phase-inv* ∧ *mutators-phase-inv* ∧ *sys-phase-inv*) ⦄
gc
⦃ *mut-m.mark-object-invL m* ⦄
⟨*proof*⟩

end

lemma *mut-m-get-roots-no-fM-write*:
⟦ *mut-m.handshake-invL m s*; *handshake-phase-inv s*↓; *fM-rel-inv s*↓; *tso-store-inv s*↓ ⟧
⟹ *atS* (*mutator m*) *mut-m.hs-get-roots-locs s* ∧ *p* ≠ *sys* ⟶ ¬*sys-mem-store-buffers p s*↓ = *mw-fM fl* # *ws*
⟨*proof*⟩

lemma (**in** *sys*) *mut-mark-object-invL*[*intro*]:
notes *filter-empty-conv*[*simp*]
notes *fun-upd-apply*[*simp*]
shows
⦃ *mut-m.handshake-invL m* ∧ *mut-m.mark-object-invL m*
∧ *LSTP* (*fA-rel-inv* ∧ *fM-rel-inv* ∧ *handshake-phase-inv* ∧ *mutators-phase-inv* ∧ *phase-rel-inv* ∧ *valid-refs-inv* ∧ *valid-W-inv* ∧ *tso-store-inv*) ⦄
sys
⦃ *mut-m.mark-object-invL m* ⦄
⟨*proof*⟩

11 Global non-interference

proofs that depend only on global invariants + lemmas

lemma (**in** *sys*) *strong-tricolour-inv*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
⦃ *LSTP* (*fM-rel-inv* ∧ *handshake-phase-inv* ∧ *mutators-phase-inv* ∧ *strong-tricolour-inv* ∧ *sys-phase-inv* ∧ *tso-store-inv* ∧ *valid-W-inv*) ⦄
sys
⦃ *LSTP strong-tricolour-inv* ⦄
⟨*proof*⟩

lemma *black-heap-reachable*:
assumes *mut-m.reachable m y s*
assumes *bh*: *black-heap s*
assumes *vri*: *valid-refs-inv s*
shows *black y s*
⟨*proof*⟩

lemma *black-heap-valid-ref-marked-insertions*:
⟦ *black-heap s*; *valid-refs-inv s* ⟧ ⟹ *mut-m.marked-insertions m s*
⟨*proof*⟩

context *sys*
begin

lemma *reachable-snapshot-inv-black-heap-no-grey-refs-dequeue-Mutate*:
assumes *sb*: *sys-mem-store-buffers* (*mutator m'*) *s* = *mw-Mutate r f opt-r'* # *ws*
assumes *bh*: *black-heap s*
assumes *ngr*: *no-grey-refs s*
assumes *vri*: *valid-refs-inv s*
shows *mut-m.reachable-snapshot-inv m* (*s*(*sys* := *s sys*⦇*heap* := (*sys-heap s*)(*r* := *map-option* (λ*obj*. *obj*⦇*obj-fields* := (*obj-fields obj*)(*f* := *opt-r'*)⦈) (*sys-heap s r*)),
mem-store-buffers := (*mem-store-buffers* (*s sys*))(*mutator m'* := *ws*)⦈)) (**is** *mut-m.reachable-snapshot-inv m ?s'*)
⟨*proof*⟩

lemma *marked-deletions-dequeue-Mark*:
⟦ *sys-mem-store-buffers p s* = *mw-Mark r fl* # *ws*; *mut-m.marked-deletions m s*; *tso-store-inv s*; *valid-W-inv s* ⟧
⟹ *mut-m.marked-deletions m* (*s*(*sys* := *s sys*⦇*heap* := (*sys-heap s*)(*r* := *map-option* (*obj-mark-update* (λ-. *fl*)) (*sys-heap s r*)), *mem-store-buffers* := (*mem-store-buffers* (*s sys*))(*p* := *ws*)⦈))
⟨*proof*⟩

lemma *marked-deletions-dequeue-Mutate*:
⟦ *sys-mem-store-buffers* (*mutator m′*) *s* = *mw-Mutate r f opt-r′* # *ws*; *mut-m.marked-deletions m s*; *mut-m.marked-inser*
m′ s ⟧
⟹ *mut-m.marked-deletions m* (*s*(*sys* := *s sys*⦇*heap* := (*sys-heap s*)(*r* := *map-option* (λ*obj*. *obj*⦇*obj-fields* := (*obj-fields obj*)(*f* := *opt-r′*)⦈) (*sys-heap s r*)),
mem-store-buffers := (*mem-store-buffers* (*s sys*))((*mutator m′*) := *ws*)⦈))
⟨*proof*⟩

lemma *grey-protects-white-dequeue-Mark*:
assumes *fl*: *fl* = *sys-fM s*
assumes *r* ∈ *ghost-honorary-grey* (*s p*)
shows (∃ *g*. (*g grey-protects-white w*) (*s*(*sys* := *s sys*⦇*heap* := (*sys-heap s*)(*r* := *map-option* (*obj-mark-update* (λ-. *fl*)) (*sys-heap s r*)), *mem-store-buffers* := (*mem-store-buffers* (*s sys*))(*p* := *ws*)⦈)))
⟷ (∃ *g*. (*g grey-protects-white w*) *s*) (**is** (∃ *g*. (*g grey-protects-white w*) *?s′*) ⟷ *?rhs*)
⟨*proof*⟩

lemma *reachable-snapshot-inv-dequeue-Mark*:
⟦ *sys-mem-store-buffers p s* = *mw-Mark r fl* # *ws*; *mut-m.reachable-snapshot-inv m s*; *valid-W-inv s* ⟧
⟹ *mut-m.reachable-snapshot-inv m* (*s*(*sys* := *s sys*⦇*heap* := (*sys-heap s*)(*r* := *map-option* (*obj-mark-update* (λ-. *fl*)) (*sys-heap s r*)), *mem-store-buffers* := (*mem-store-buffers* (*s sys*))(*p* := *ws*)⦈))
⟨*proof*⟩

lemma *marked-insertions-dequeue-Mark*:
⟦ *sys-mem-store-buffers p s* = *mw-Mark r fl* # *ws*; *mut-m.marked-insertions m s*; *tso-writes-inv s*; *valid-W-inv s* ⟧
⟹ *mut-m.marked-insertions m* (*s*(*sys* := *s sys*⦇*heap* := (*sys-heap s*)(*r* := *map-option* (*obj-mark-update* (λ-. *fl*)) (*sys-heap s r*)), *mem-store-buffers* := (*mem-store-buffers* (*s sys*))(*p* := *ws*)⦈))
⟨*proof*⟩

lemma *marked-insertions-dequeue-Mutate*:
⟦ *sys-mem-store-buffers p s* = *mw-Mutate r f r′* # *ws*; *mut-m.marked-insertions m s* ⟧
⟹ *mut-m.marked-insertions m* (*s*(*sys* := *s sys*⦇*heap* := (*sys-heap s*)(*r* := *map-option* (λ*obj*. *obj*⦇*obj-fields* := (*obj-fields obj*)(*f* := *r′*)⦈) (*sys-heap s r*)),
mem-store-buffers := (*mem-store-buffers* (*s sys*))(*p* := *ws*)⦈))
⟨*proof*⟩

lemma *grey-protects-white-dequeue-Mutate*:
assumes *sb*: *sys-mem-store-buffers* (*mutator m*) *s* = *mw-Mutate r f opt-r′* # *ws*
assumes *mi*: *mut-m.marked-insertions m s*
assumes *md*: *mut-m.marked-deletions m s*
shows (∃ *g*. (*g grey-protects-white w*) (*s*(*sys* := *s sys*⦇*heap* := (*sys-heap s*)(*r* := *map-option* (λ*obj*. *obj*⦇*obj-fields* := (*obj-fields obj*)(*f* := *opt-r′*)⦈) (*sys-heap s r*)),
mem-store-buffers := (*mem-store-buffers* (*s sys*))(*mutator m* := *ws*)⦈)))
⟷ (∃ *g*. (*g grey-protects-white w*) *s*) (**is** (∃ *g*. (*g grey-protects-white w*) *?s′*) ⟷ *?rhs*)
⟨*proof*⟩

lemma *reachable-snapshot-inv-dequeue-Mutate*:
notes *grey-protects-white-dequeue-Mutate*[*simp*]
fixes *s* :: (*′field*, *′mut*, *′payload*, *′ref*) *lsts*
assumes *sb*: *sys-mem-store-buffers* (*mutator m′*) *s* = *mw-Mutate r f opt-r′* # *ws*

assumes *mi*: *mut-m.marked-insertions m' s*
assumes *md*: *mut-m.marked-deletions m' s*
assumes *rsi*: *mut-m.reachable-snapshot-inv m s*
assumes *sti*: *strong-tricolour-inv s*
assumes *vri*: *valid-refs-inv s*
shows *mut-m.reachable-snapshot-inv m* (*s*(*sys* := *s sys*(|*heap* := (*sys-heap s*)(*r* := *map-option* (λ*obj*. *obj*(|*obj-fields* := (*obj-fields obj*)(*f* := *opt-r'*)|)) (*sys-heap s r*)),
mem-store-buffers := (*mem-store-buffers* (*s sys*))(*mutator m'* := *ws*)|))) (**is** *mut-m.reachable-snapshot-inv m ?s'*)
⟨*proof*⟩

lemma *mutator-phase-inv*[*intro*]:
{| *LSTP* (*fA-rel-inv* ∧ *fM-rel-inv* ∧ *handshake-phase-inv* ∧ *mutators-phase-inv* ∧ *strong-tricolour-inv* ∧ *sys-phase-inv* ∧ *tso-store-inv* ∧ *valid-refs-inv* ∧ *valid-W-inv*) |}
sys
{| *LSTP* (*mut-m.mutator-phase-inv m*) |}
⟨*proof*⟩

end

12 Mark Object

These are the most intricate proofs in this development.

context *mut-m*
begin

lemma *mark-object-invL*[*intro*]:
{| *handshake-invL* ∧ *mark-object-invL*
∧ *mut-get-roots.mark-object-invL m*
∧ *mut-store-del.mark-object-invL m*
∧ *mut-store-ins.mark-object-invL m*
∧ *LSTP* (*phase-rel-inv* ∧ *handshake-phase-inv* ∧ *phase-rel-inv* ∧ *tso-store-inv* ∧ *valid-refs-inv*) |}
mutator m
{| *mark-object-invL* |}
⟨*proof*⟩

lemma *mut-store-ins-mark-object-invL*[*intro*]:
{| *mut-store-ins.mark-object-invL m* ∧ *mark-object-invL* ∧ *handshake-invL* ∧ *tso-lock-invL*
∧ *LSTP* (*handshake-phase-inv* ∧ *valid-W-inv* ∧ *tso-store-inv* ∧ *valid-refs-inv*) |}
mutator m
{| *mut-store-ins.mark-object-invL m* |}
⟨*proof*⟩

lemma *mut-store-del-mark-object-invL*[*intro*]:
{| *mut-store-del.mark-object-invL m* ∧ *mark-object-invL* ∧ *handshake-invL* ∧ *tso-lock-invL*
∧ *LSTP* (*handshake-phase-inv* ∧ *valid-W-inv* ∧ *tso-store-inv* ∧ *valid-refs-inv*) |}
mutator m
{| *mut-store-del.mark-object-invL m* |}
⟨*proof*⟩

lemma *mut-get-roots-mark-object-invL*[*intro*]:
{| *mut-get-roots.mark-object-invL m* ∧ *mark-object-invL* ∧ *handshake-invL* ∧ *tso-lock-invL*
∧ *LSTP* (*handshake-phase-inv* ∧ *valid-W-inv* ∧ *tso-store-inv* ∧ *valid-refs-inv*) |}
mutator m
{| *mut-get-roots.mark-object-invL m* |}
⟨*proof*⟩

end

lemma (**in** *mut-m'*) *mut-mark-object-invL*[*intro*]:
  notes *obj-at-field-on-heap-splits*[*split*]
  notes *fun-upd-apply*[*simp*]
  shows
  ⦃ *mark-object-invL* ⦄ *mutator m'*
⟨*proof*⟩

12.1 *obj-fields-marked-inv*

context *gc*
begin

lemma *gc-mark-mark-object-invL*[*intro*]:
  ⦃ *fM-fA-invL* ∧ *gc-mark.mark-object-invL* ∧ *obj-fields-marked-invL* ∧ *tso-lock-invL*
      ∧ *LSTP valid-W-inv* ⦄
    gc
  ⦃ *gc-mark.mark-object-invL* ⦄
⟨*proof*⟩

lemma *obj-fields-marked-invL*[*intro*]:
  ⦃ *fM-fA-invL* ∧ *phase-invL* ∧ *obj-fields-marked-invL* ∧ *gc-mark.mark-object-invL*
      ∧ *LSTP* (*tso-store-inv* ∧ *valid-W-inv* ∧ *valid-refs-inv*) ⦄
    gc
  ⦃ *obj-fields-marked-invL* ⦄
⟨*proof*⟩

end

context *sys*
begin

lemma *mut-store-ins-mark-object-invL*[*intro*]:
  notes *mut-m-not-idle-no-fM-writeD*[**where** *m*=*m*, *dest*!]
  notes *not-blocked-def*[*simp*]
  notes *fun-upd-apply*[*simp*]
  notes *if-split-asm*[*split del*]
  shows
  ⦃ *mut-m.tso-lock-invL m* ∧ *mut-m.mark-object-invL m* ∧ *mut-store-ins.mark-object-invL m*
      ∧ *LSTP* (*fM-rel-inv* ∧ *handshake-phase-inv* ∧ *valid-W-inv* ∧ *tso-store-inv*) ⦄
    sys
  ⦃ *mut-store-ins.mark-object-invL m* ⦄
⟨*proof*⟩

lemma *mut-store-del-mark-object-invL*[*intro*]:
  notes *mut-m-not-idle-no-fM-writeD*[**where** *m*=*m*, *dest*!]
  notes *not-blocked-def*[*simp*]
  notes *fun-upd-apply*[*simp*]
  notes *if-split-asm*[*split del*]
  shows
  ⦃ *mut-m.tso-lock-invL m* ∧ *mut-m.mark-object-invL m* ∧ *mut-store-del.mark-object-invL m*
      ∧ *LSTP* (*fM-rel-inv* ∧ *handshake-phase-inv* ∧ *valid-W-inv* ∧ *tso-store-inv*) ⦄
    sys
  ⦃ *mut-store-del.mark-object-invL m* ⦄
⟨*proof*⟩

lemma *mut-get-roots-mark-object-invL*[*intro*]:
  notes *not-blocked-def*[*simp*]
  notes *p-not-sys*[*simp*]
  notes *mut-m.handshake-phase-invD*[**where** *m*=*m*, *dest*!]
  notes *fun-upd-apply*[*simp*]
  notes *if-split-asm*[*split del*]
  shows
  ⦃ *mut-m.tso-lock-invL m* ∧ *mut-m.handshake-invL m* ∧ *mut-get-roots.mark-object-invL m*
      ∧ *LSTP* (*fM-rel-inv* ∧ *handshake-phase-inv* ∧ *valid-W-inv* ∧ *tso-store-inv*) ⦄
    sys
   ⦃ *mut-get-roots.mark-object-invL m* ⦄
⟨*proof*⟩

lemma *gc-mark-mark-object-invL*[*intro*]:
  notes *fun-upd-apply*[*simp*]
  notes *if-split-asm*[*split del*]
  shows
  ⦃ *gc.fM-fA-invL* ∧ *gc.handshake-invL* ∧ *gc.phase-invL* ∧ *gc-mark.mark-object-invL* ∧ *gc.tso-lock-invL*
      ∧ *LSTP* (*handshake-phase-inv* ∧ *phase-rel-inv* ∧ *valid-W-inv* ∧ *tso-store-inv*) ⦄
    sys
   ⦃ *gc-mark.mark-object-invL* ⦄
⟨*proof*⟩

end

lemma (**in** *mut-m′*) *mut-get-roots-mark-object-invL*[*intro*]:
  ⦃ *mut-get-roots.mark-object-invL m* ⦄ *mutator m′*
⟨*proof*⟩

lemma (**in** *mut-m′*) *mut-store-ins-mark-object-invL*[*intro*]:
  ⦃ *mut-store-ins.mark-object-invL m* ⦄ *mutator m′*
⟨*proof*⟩

lemma (**in** *mut-m′*) *mut-store-del-mark-object-invL*[*intro*]:
  ⦃ *mut-store-del.mark-object-invL m* ⦄ *mutator m′*
⟨*proof*⟩

lemma (**in** *gc*) *mut-get-roots-mark-object-invL*[*intro*]:
 ⦃ *handshake-invL* ∧ *mut-m.handshake-invL m* ∧ *mut-get-roots.mark-object-invL m* ⦄ *gc* ⦃ *mut-get-roots.mark-object-inv*
m ⦄
⟨*proof*⟩

lemma (**in** *mut-m*) *gc-obj-fields-marked-invL*[*intro*]:
  ⦃ *handshake-invL* ∧ *gc.handshake-invL* ∧ *gc.obj-fields-marked-invL*
    ∧ *LSTP* (*tso-store-inv* ∧ *valid-refs-inv*) ⦄
    mutator m
   ⦃ *gc.obj-fields-marked-invL* ⦄
⟨*proof*⟩

lemma (**in** *mut-m*) *gc-mark-mark-object-invL*[*intro*]:
  ⦃ *gc-mark.mark-object-invL* ⦄ *mutator m*
⟨*proof*⟩

13 Handshake phases

Reasoning about phases, handshakes.

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

lemma (**in** *gc*) *phase-invL-eq-imp*:
eq-imp (λ(-::*unit*) *s*. (*AT s gc*, *s*↓ *gc*, *tso-pending-phase gc s*↓))
phase-invL
⟨*proof*⟩

lemmas *gc-phase-invL-niE*[*nie*] =
iffD1[*OF gc.phase-invL-eq-imp*[*simplified eq-imp-simps*, *rule-format*, *unfolded conj-explode*], *rotated* −*1*]

lemma (**in** *gc*) *phase-invL*[*intro*]:
⦃ *phase-invL* ∧ *LSTP phase-rel-inv* ⦄ *gc* ⦃ *phase-invL* ⦄
⟨*proof*⟩

lemma (**in** *sys*) *gc-phase-invL*[*intro*]:
notes *fun-upd-apply*[*simp*]
notes *if-splits*[*split*]
shows
⦃ *gc.phase-invL* ⦄ *sys*
⟨*proof*⟩

lemma (**in** *mut-m*) *gc-phase-invL*[*intro*]:
⦃ *gc.phase-invL* ⦄ *mutator m*
⟨*proof*⟩

lemma (**in** *gc*) *phase-rel-inv*[*intro*]:
⦃ *handshake-invL* ∧ *phase-invL* ∧ *LSTP phase-rel-inv* ⦄ *gc* ⦃ *LSTP phase-rel-inv* ⦄
⟨*proof*⟩

lemma (**in** *sys*) *phase-rel-inv*[*intro*]:
notes *gc.phase-invL-def*[*inv*]
notes *phase-rel-inv-def*[*inv*]
notes *fun-upd-apply*[*simp*]
shows
⦃ *LSTP* (*phase-rel-inv* ∧ *tso-store-inv*) ⦄ *sys* ⦃ *LSTP phase-rel-inv* ⦄
⟨*proof*⟩

lemma (**in** *mut-m*) *phase-rel-inv*[*intro*]:
⦃ *handshake-invL* ∧ *LSTP* (*handshake-phase-inv* ∧ *phase-rel-inv*) ⦄
mutator m
⦃ *LSTP phase-rel-inv* ⦄
⟨*proof*⟩

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

lemma *gc-handshake-invL-eq-imp*:
eq-imp (λ(-::*unit*) *s*. (*AT s gc*, *s*↓ *gc*, *sys-ghost-hs-phase s*↓, *hs-pending* (*s*↓ *sys*), *ghost-hs-in-sync* (*s*↓ *sys*), *sys-hs-type s*↓))
gc.handshake-invL
⟨*proof*⟩

lemmas *gc-handshake-invL-niE*[*nie*] =
iffD1[*OF gc-handshake-invL-eq-imp*[*simplified eq-imp-simps*, *rule-format*, *unfolded conj-explode*], *rotated* −*1*]

lemma (**in** *sys*) *gc-handshake-invL*[*intro*]:
⦃ *gc.handshake-invL* ⦄ *sys*
⟨*proof*⟩

lemma (**in** *sys*) *handshake-phase-inv*[*intro*]:
⦃ *LSTP handshake-phase-inv* ⦄ *sys*
⟨*proof*⟩

lemma (**in** *gc*) *handshake-invL*[*intro*]:
  notes *fun-upd-apply*[*simp*]
  shows
  ⦃ *handshake-invL* ⦄ *gc*
⟨*proof*⟩

lemma (**in** *gc*) *handshake-phase-inv*[*intro*]:
  notes *fun-upd-apply*[*simp*]
  shows
  ⦃ *handshake-invL* ∧ *LSTP handshake-phase-inv* ⦄ *gc* ⦃ *LSTP handshake-phase-inv* ⦄
⟨*proof*⟩

Local handshake phase invariant for the mutators.

lemma (**in** *mut-m*) *handshake-invL-eq-imp*:
  eq-imp (λ(-::*unit*) *s*. (*AT s* (*mutator m*), *s*↓ (*mutator m*), *sys-hs-type s*↓, *sys-hs-pending m s*↓, *mem-store-buffers* (*s*↓ *sys*) (*mutator m*)))
          handshake-invL
⟨*proof*⟩

lemmas *mut-m-handshake-invL-niE*[*nie*] =
  iffD1[*OF mut-m.handshake-invL-eq-imp*[*simplified eq-imp-simps*, *rule-format*, *unfolded conj-explode*], *rotated* −*1*]

lemma (**in** *mut-m*) *handshake-invL*[*intro*]:
  ⦃ *handshake-invL* ⦄ *mutator m*
⟨*proof*⟩

lemma (**in** *mut-m′*) *handshake-invL*[*intro*]:
  ⦃ *handshake-invL* ⦄ *mutator m′*
⟨*proof*⟩

lemma (**in** *gc*) *mut-handshake-invL*[*intro*]:
  notes *fun-upd-apply*[*simp*]
  shows
  ⦃ *handshake-invL* ∧ *mut-m.handshake-invL m* ⦄ *gc* ⦃ *mut-m.handshake-invL m* ⦄
⟨*proof*⟩

lemma (**in** *sys*) *mut-handshake-invL*[*intro*]:
  notes *if-splits*[*split*]
  notes *fun-upd-apply*[*simp*]
  shows
  ⦃ *mut-m.handshake-invL m* ⦄ *sys*
⟨*proof*⟩

lemma (**in** *mut-m*) *gc-handshake-invL*[*intro*]:
  notes *fun-upd-apply*[*simp*]
  shows
  ⦃ *handshake-invL* ∧ *gc.handshake-invL* ⦄ *mutator m* ⦃ *gc.handshake-invL* ⦄
⟨*proof*⟩

lemma (**in** *mut-m*) *handshake-phase-inv*[*intro*]:
  notes *fun-upd-apply*[*simp*]
  shows
  ⦃ *handshake-invL* ∧ *LSTP handshake-phase-inv* ⦄ *mutator m* ⦃ *LSTP handshake-phase-inv* ⦄
⟨*proof*⟩

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

lemma *gc-fM-fA-invL-eq-imp*:
  eq-imp (λ(-::*unit*) *s*. (*AT s gc*, *s*↓ *gc*, *sys-fA s*↓, *sys-fM s*↓, *sys-mem-store-buffers gc s*↓))
        gc.fM-fA-invL
⟨*proof*⟩

lemmas *gc-fM-fA-invL-niE*[*nie*] =
  iffD1[*OF gc-fM-fA-invL-eq-imp*[*simplified eq-imp-simps*, *rule-format*, *unfolded conj-explode*], *rotated* −*1*]

context *gc*
begin

lemma *fM-fA-invL*[*intro*]:
  ⦃ *fM-fA-invL* ⦄ *gc*
⟨*proof*⟩

lemma *fM-rel-inv*[*intro*]:
  notes *fun-upd-apply*[*simp*]
  shows
  ⦃ *fM-fA-invL* ∧ *handshake-invL* ∧ *tso-lock-invL* ∧ *LSTP fM-rel-inv* ⦄
    gc
  ⦃ *LSTP fM-rel-inv* ⦄
⟨*proof*⟩

lemma *fA-rel-inv*[*intro*]:
  notes *fun-upd-apply*[*simp*]
  shows
  ⦃ *fM-fA-invL* ∧ *handshake-invL* ∧ *LSTP fA-rel-inv* ⦄
    gc
  ⦃ *LSTP fA-rel-inv* ⦄
⟨*proof*⟩

end

context *mut-m*
begin

lemma *gc-fM-fA-invL*[*intro*]:
  ⦃ *gc.fM-fA-invL* ⦄ *mutator m*
⟨*proof*⟩

lemma *fM-rel-inv*[*intro*]:
  notes *fun-upd-apply*[*simp*]
  shows
  ⦃ *LSTP fM-rel-inv* ⦄ *mutator m*
⟨*proof*⟩

lemma *fA-rel-inv*[*intro*]:
  notes *fun-upd-apply*[*simp*]
  shows
  ⦃ *LSTP fA-rel-inv* ⦄ *mutator m*
⟨*proof*⟩

end

context *gc*

begin

lemma *fA-neq-locs-diff-fA-tso-empty-locs*:
fA-neq-locs − *fA-tso-empty-locs* = {}
⟨*proof*⟩

end

context *sys*
begin

lemma *gc-fM-fA-invL*[*intro*]:
⦃ *gc.fM-fA-invL* ∧ *LSTP* (*fA-rel-inv* ∧ *fM-rel-inv* ∧ *tso-store-inv*) ⦄
sys
⦃ *gc.fM-fA-invL* ⦄
⟨*proof*⟩

lemma *fM-rel-inv*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
⦃ *LSTP* (*fM-rel-inv* ∧ *tso-store-inv*) ⦄ *sys* ⦃ *LSTP fM-rel-inv* ⦄
⟨*proof*⟩

lemma *fA-rel-inv*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
⦃ *LSTP* (*fA-rel-inv* ∧ *tso-store-inv*) ⦄ *sys* ⦃ *LSTP fA-rel-inv* ⦄
⟨*proof*⟩

end

13.0.1 sys phase inv

context *mut-m*
begin

lemma *sys-phase-inv*[*intro*]:
notes *if-split-asm*[*split del*]
notes *fun-upd-apply*[*simp*]
shows
⦃ *handshake-invL*
∧ *mark-object-invL*
∧ *mut-get-roots.mark-object-invL m*
∧ *mut-store-del.mark-object-invL m*
∧ *mut-store-ins.mark-object-invL m*
∧ *LSTP* (*fA-rel-inv* ∧ *fM-rel-inv* ∧ *handshake-phase-inv* ∧ *mutators-phase-inv* ∧ *phase-rel-inv* ∧ *sys-phase-inv* ∧ *valid-refs-inv*) ⦄
mutator m
⦃ *LSTP sys-phase-inv* ⦄
⟨*proof*⟩

end

lemma (**in** *gc*) *sys-phase-inv*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
⦃ *fM-fA-invL* ∧ *gc-W-empty-invL* ∧ *handshake-invL* ∧ *obj-fields-marked-invL*
∧ *phase-invL* ∧ *sweep-loop-invL*

$\wedge$ *LSTP* (*phase-rel-inv* $\wedge$ *sys-phase-inv* $\wedge$ *valid-W-inv* $\wedge$ *tso-store-inv*) ⦄
gc
⦃ *LSTP sys-phase-inv* ⦄
⟨*proof*⟩

lemma *no-grey-refs-no-marks*[*simp*]:
⟦ *no-grey-refs s*; *valid-W-inv s* ⟧ $\Longrightarrow$ ¬*sys-mem-store-buffers p s* = *mw-Mark r fl* # *ws*
⟨*proof*⟩

context *sys*
begin

lemma *black-heap-dequeue-mark*[*iff*]:
⟦ *sys-mem-store-buffers p s* = *mw-Mark r fl* # *ws*; *black-heap s*; *valid-W-inv s* ⟧
$\Longrightarrow$ *black-heap* (*s*(*sys* := *s sys*⦇*heap* := (*sys-heap s*)(*r* := *map-option* (*obj-mark-update* (λ-. *fl*)) (*sys-heap s r*)), *mem-store-buffers* := (*mem-store-buffers* (*s sys*))(*p* := *ws*)⦈))
⟨*proof*⟩

lemma *white-heap-dequeue-fM*[*iff*]:
black-heap s↓
$\Longrightarrow$ *white-heap* (*s*↓(*sys* := *s*↓ *sys*⦇*fM* := ¬ *sys-fM s*↓, *mem-store-buffers* := (*mem-store-buffers* (*s*↓ *sys*))(*gc* := *ws*)⦈))
⟨*proof*⟩

lemma *black-heap-dequeue-fM*[*iff*]:
⟦ *white-heap s*↓; *no-grey-refs s*↓ ⟧
$\Longrightarrow$ *black-heap* (*s*↓(*sys* := *s*↓ *sys*⦇*fM* := ¬ *sys-fM s*↓, *mem-store-buffers* := (*mem-store-buffers* (*s*↓ *sys*))(*gc* := *ws*)⦈))
⟨*proof*⟩

lemma *sys-phase-inv*[*intro*]:
notes *if-split-asm*[*split del*]
notes *fun-upd-apply*[*simp*]
shows
⦃ *LSTP* (*fA-rel-inv* $\wedge$ *fM-rel-inv* $\wedge$ *handshake-phase-inv* $\wedge$ *mutators-phase-inv* $\wedge$ *phase-rel-inv* $\wedge$ *sys-phase-inv* $\wedge$ *tso-store-inv* $\wedge$ *valid-W-inv*) ⦄
sys
⦃ *LSTP sys-phase-inv* ⦄
⟨*proof*⟩

end

context *mut-m*
begin

lemma *marked-insertions-store-ins*[*simp*]:
⟦ *marked-insertions s*; (∃ *r'*. *opt-r'* = *Some r'*) ⟶ *marked* (*the opt-r'*) *s* ⟧
$\Longrightarrow$ *marked-insertions*
(*s*(*mutator m* := *s* (*mutator m*)⦇*ghost-honorary-root* := {}⦈,
sys := *s sys*
⦇*mem-store-buffers* := (*mem-store-buffers* (*s sys*))(*mutator m* := *sys-mem-store-buffers* (*mutator m*) *s* @ [*mw-Mutate r f opt-r'*])⦈))
⟨*proof*⟩

lemma *marked-insertions-alloc*[*simp*]:
  ⟦ *heap* (*s sys*) *r*′ = *None*; *valid-refs-inv s* ⟧
  ⟹ *marked-insertions* (*s*(*mutator m*′ := *s* (*mutator m*′)⦇*roots* := *roots*′⦈, *sys* := *s sys*⦇*heap* := (*sys-heap s*)(*r*′ ↦ *obj*′)⦈))
  ⟷ *marked-insertions s*
⟨*proof*⟩

lemma *marked-deletions-store-ins*[*simp*]:
  ⟦ *marked-deletions s*; *obj-at-field-on-heap* (λ*r*′. *marked r*′ *s*) *r f s* ⟧
    ⟹ *marked-deletions*
        (*s*(*mutator m* := *s* (*mutator m*)⦇*ghost-honorary-root* := {}⦈,
          sys := *s sys*
          ⦇*mem-store-buffers* := (*mem-store-buffers* (*s sys*))(*mutator m* := *sys-mem-store-buffers* (*mutator m*) *s* @ [*mw-Mutate r f opt-r*′])⦈))
⟨*proof*⟩

lemma *marked-deletions-alloc*[*simp*]:
  ⟦ *marked-deletions s*; *heap* (*s sys*) *r*′ = *None*; *valid-refs-inv s* ⟧
  ⟹ *marked-deletions* (*s*(*mutator m*′ := *s* (*mutator m*′)⦇*roots* := *roots*′⦈, *sys* := *s sys*⦇*heap* := (*sys-heap s*)(*r*′ ↦ *obj*′)⦈))
⟨*proof*⟩

end

13.1 Sweep loop invariants

lemma (**in** *gc*) *sweep-loop-invL-eq-imp*:
  eq-imp (λ(-::*unit*) *s*. (*AT s gc*, *s*↓ *gc*, *sys-fM s*↓, *map-option obj-mark* ∘ *sys-heap s*↓))
      sweep-loop-invL
⟨*proof*⟩

lemmas *gc-sweep-loop-invL-niE*[*nie*] =
  iffD1[*OF gc.sweep-loop-invL-eq-imp*[*simplified eq-imp-simps*, *rule-format*, *unfolded conj-explode*, *rule-format*], *rotated* −*1*]

lemma (**in** *gc*) *sweep-loop-invL*[*intro*]:
  ⦃ *fM-fA-invL* ∧ *phase-invL* ∧ *sweep-loop-invL* ∧ *tso-lock-invL*
      ∧ *LSTP* (*phase-rel-inv* ∧ *mutators-phase-inv* ∧ *valid-W-inv*) ⦄
    gc
  ⦃ *sweep-loop-invL* ⦄
⟨*proof*⟩

context *gc*
begin

lemma *sweep-loop-locs-subseteq-sweep-locs*:
  sweep-loop-locs ⊆ *sweep-locs*
⟨*proof*⟩

lemma *sweep-locs-subseteq-fM-tso-empty-locs*:
  sweep-locs ⊆ *fM-tso-empty-locs*
⟨*proof*⟩

lemma *sweep-loop-locs-fM-eq-locs*:

sweep-loop-locs ⊆ *fM-eq-locs*
⟨*proof*⟩

lemma *sweep-loop-locs-fA-eq-locs*:
sweep-loop-locs ⊆ *fA-eq-locs*
⟨*proof*⟩

lemma *black-heap-locs-subseteq-fM-tso-empty-locs*:
black-heap-locs ⊆ *fM-tso-empty-locs*
⟨*proof*⟩

lemma *black-heap-locs-fM-eq-locs*:
black-heap-locs ⊆ *fM-eq-locs*
⟨*proof*⟩

lemma *black-heap-locs-fA-eq-locs*:
black-heap-locs ⊆ *fA-eq-locs*
⟨*proof*⟩

lemma *fM-fA-invL-tso-emptyD*:
⟦ *atS gc ls s*; *fM-fA-invL s*; *ls* ⊆ *fM-tso-empty-locs* ⟧ ⟹ *tso-pending-fM gc s*↓ = []
⟨*proof*⟩

lemma *gc-sweep-loop-invL-locsE*[*rule-format*]:
(*atS gc* (*sweep-locs* ∪ *black-heap-locs*) *s* ⟶ *False*) ⟹ *gc.sweep-loop-invL s*
⟨*proof*⟩

end

lemma (**in** *sys*) *gc-sweep-loop-invL*[*intro*]:
⦃ *gc.fM-fA-invL* ∧ *gc.gc-W-empty-invL* ∧ *gc.sweep-loop-invL*
∧ *LSTP* (*tso-store-inv* ∧ *valid-W-inv*) ⦄
sys
⦃ *gc.sweep-loop-invL* ⦄
⟨*proof*⟩

lemma (**in** *mut-m*) *gc-sweep-loop-invL*[*intro*]:
⦃ *gc.fM-fA-invL* ∧ *gc.handshake-invL* ∧ *gc.sweep-loop-invL*
∧ *LSTP* (*mutators-phase-inv* ∧ *valid-refs-inv*) ⦄
mutator m
⦃ *gc.sweep-loop-invL* ⦄
⟨*proof*⟩

13.2 Mutator proofs

context *mut-m*
begin

lemma *reachable-snapshot-inv-mo-co-mark*[*simp*]:
⟦ *ghost-honorary-grey* (*s p*) = {}; *reachable-snapshot-inv s* ⟧
⟹ *reachable-snapshot-inv* (*s*(*p* := *s p*⦇ *ghost-honorary-grey* := {*r*} ⦈))
⟨*proof*⟩

lemma *reachable-snapshot-inv-hs-get-roots-done*:
assumes *sti*: *strong-tricolour-inv s*
assumes *m*: ∀ *r* ∈ *mut-roots s*. *marked r s*

assumes *ghr*: *mut-ghost-honorary-root s* = {}
assumes *t*: *tso-pending-mutate* (*mutator m*) *s* = []
assumes *vri*: *valid-refs-inv s*
shows *reachable-snapshot-inv*
(*s*(*mutator m* := *s* (*mutator m*)(| *W* := {}, *ghost-hs-phase* := *ghp'*|),
sys := *s sys*(| *hs-pending* := *hp'*, *W* := *sys-W s* ∪ *mut-W s*, *ghost-hs-in-sync* := *in'*|)))
(**is** *reachable-snapshot-inv ?s'*)
⟨*proof*⟩

lemma *reachable-snapshot-inv-hs-get-work-done*:
reachable-snapshot-inv s
⟹ *reachable-snapshot-inv*
(*s*(*mutator m* := *s* (*mutator m*)(| *W* := {}|),
sys := *s sys*(| *hs-pending* := *pending'*, *W* := *sys-W s* ∪ *mut-W s*,
ghost-hs-in-sync := (*ghost-hs-in-sync* (*s sys*))(*m* := *True*)|)))
⟨*proof*⟩

lemma *reachable-snapshot-inv-deref-del*:
⟦ *reachable-snapshot-inv s*; *sys-load* (*mutator m*) (*mr-Ref r f*) (*s sys*) = *mv-Ref opt-r'*; *r* ∈ *mut-roots s*; *mut-ghost-honorary-root s* = {} ⟧
⟹ *reachable-snapshot-inv* (*s*(*mutator m* := *s* (*mutator m*)(| *ghost-honorary-root* := *Option.set-option opt-r'*, *ref* := *opt-r'*|)))
⟨*proof*⟩

lemma *mutator-phase-inv*[*intro*]:
notes *fun-upd-apply*[*simp*]
notes *reachable-snapshot-inv-deref-del*[*simp*]
notes *if-split-asm*[*split del*]
shows
{| *handshake-invL*
∧ *mark-object-invL*
∧ *mut-get-roots.mark-object-invL m*
∧ *mut-store-del.mark-object-invL m*
∧ *mut-store-ins.mark-object-invL m*
∧ *LSTP* (*handshake-phase-inv* ∧ *mutators-phase-inv* ∧ *phase-rel-inv* ∧ *sys-phase-inv* ∧ *fA-rel-inv* ∧ *fM-rel-inv* ∧ *valid-refs-inv* ∧ *strong-tricolour-inv* ∧ *valid-W-inv*) |}
mutator m
{| *LSTP mutator-phase-inv* |}
⟨*proof*⟩

end

lemma (**in** *mut-m'*) *mutator-phase-inv*[*intro*]:
notes *mut-m.mark-object-invL-def*[*inv*]
notes *mut-m.handshake-invL-def*[*inv*]
notes *fun-upd-apply*[*simp*]
shows
{| *handshake-invL* ∧ *mut-m.handshake-invL m'*
∧ *mut-m.mark-object-invL m'*
∧ *mut-get-roots.mark-object-invL m'*
∧ *mut-store-del.mark-object-invL m'*
∧ *mut-store-ins.mark-object-invL m'*
∧ *LSTP* (*fA-rel-inv* ∧ *fM-rel-inv* ∧ *handshake-phase-inv* ∧ *mutators-phase-inv* ∧ *valid-refs-inv*) |}
mutator m'
{| *LSTP mutator-phase-inv* |}
⟨*proof*⟩

lemma *no-black-refs-sweep-loop-free*[*simp*]:
no-black-refs s $\Longrightarrow$ *no-black-refs* (*s*(*sys* := *s sys*⦇*heap* := (*sys-heap s*)(*gc-tmp-ref s* := *None*)⦈))
⟨*proof*⟩

lemma *no-black-refs-load-W*[*simp*]:
⟦ *no-black-refs s*; *gc-W s* = {} ⟧
$\Longrightarrow$ *no-black-refs* (*s*(*gc* := *s gc*⦇*W* := *sys-W s*⦈, *sys* := *s sys*⦇*W* := {}⦈))
⟨*proof*⟩

lemma *marked-insertions-sweep-loop-free*[*simp*]:
⟦ *mut-m.marked-insertions m s*; *white r s* ⟧
$\Longrightarrow$ *mut-m.marked-insertions m* (*s*(*sys* := (*s sys*)⦇*heap* := (*heap* (*s sys*))(*r* := *None*)⦈))
⟨*proof*⟩

lemma *marked-deletions-sweep-loop-free*[*simp*]:
notes *fun-upd-apply*[*simp*]
shows
⟦ *mut-m.marked-deletions m s*; *mut-m.reachable-snapshot-inv m s*; *no-grey-refs s*; *white r s* ⟧
$\Longrightarrow$ *mut-m.marked-deletions m* (*s*(*sys* := *s sys*⦇*heap* := (*sys-heap s*)(*r* := *None*)⦈))
⟨*proof*⟩

context *gc*
begin

lemma *obj-fields-marked-inv-blacken*:
⟦ *gc-field-set s* = {}; *obj-fields-marked s*; (*gc-tmp-ref s points-to w*) *s*; *white w s* ⟧ $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *obj-fields-marked-inv-has-white-path-to-blacken*:
⟦ *gc-field-set s* = {}; *gc-tmp-ref s* ∈ *gc-W s*; (*gc-tmp-ref s has-white-path-to w*) *s*; *obj-fields-marked s*; *valid-W-inv s* ⟧ $\Longrightarrow$ *w* = *gc-tmp-ref s*
⟨*proof*⟩

lemma *mutator-phase-inv*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
⦃ *fM-fA-invL* ∧ *gc-W-empty-invL* ∧ *handshake-invL* ∧ *obj-fields-marked-invL* ∧ *sweep-loop-invL*
∧ *gc-mark.mark-object-invL*
∧ *LSTP* (*handshake-phase-inv* ∧ *mutators-phase-inv* ∧ *valid-refs-inv* ∧ *valid-W-inv*) ⦄
gc
⦃ *LSTP* (*mut-m.mutator-phase-inv m*) ⦄
⟨*proof*⟩

end

lemma (**in** *gc*) *strong-tricolour-inv*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
⦃ *fM-fA-invL* ∧ *gc-W-empty-invL* ∧ *gc-mark.mark-object-invL* ∧ *obj-fields-marked-invL* ∧ *sweep-loop-invL*
∧ *LSTP* (*strong-tricolour-inv* ∧ *valid-W-inv*) ⦄
gc
⦃ *LSTP strong-tricolour-inv* ⦄
⟨*proof*⟩

lemma (**in** *mut-m*) *strong-tricolour*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows

⦃ *mark-object-invL*
∧ *mut-get-roots.mark-object-invL m*
∧ *mut-store-del.mark-object-invL m*
∧ *mut-store-ins.mark-object-invL m*
∧ *LSTP* (*fA-rel-inv* ∧ *fM-rel-inv* ∧ *handshake-phase-inv* ∧ *mutators-phase-inv* ∧ *strong-tricolour-inv* ∧ *sys-phase-inv* ∧ *valid-refs-inv*) ⦄
mutator m
⦃ *LSTP strong-tricolour-inv* ⦄
⟨*proof*⟩

14 Coarse TSO invariants

context *gc*
begin

lemma *tso-lock-invL*[*intro*]:
⦃ *tso-lock-invL* ⦄ *gc*
⟨*proof*⟩

lemma *tso-store-inv*[*intro*]:
⦃ *LSTP tso-store-inv* ⦄ *gc*
⟨*proof*⟩

lemma *mut-tso-lock-invL*[*intro*]:
⦃ *mut-m.tso-lock-invL m* ⦄ *gc*
⟨*proof*⟩

end

context *mut-m*
begin

lemma *tso-store-inv*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
⦃ *LSTP tso-store-inv* ⦄ *mutator m*
⟨*proof*⟩

lemma *gc-tso-lock-invL*[*intro*]:
⦃ *gc.tso-lock-invL* ⦄ *mutator m*
⟨*proof*⟩

lemma *tso-lock-invL*[*intro*]:
⦃ *tso-lock-invL* ⦄ *mutator m*
⟨*proof*⟩

end

context *mut-m′*
begin

lemma *tso-lock-invL*[*intro*]:
⦃ *tso-lock-invL* ⦄ *mutator m′*
⟨*proof*⟩

end

context *sys*
begin

lemma *tso-gc-store-inv*[*intro*]:
  notes *fun-upd-apply*[*simp*]
  shows
  ⦃ *LSTP tso-store-inv* ⦄ *sys*
⟨*proof*⟩

lemma *gc-tso-lock-invL*[*intro*]:
  ⦃ *gc.tso-lock-invL* ⦄ *sys*
⟨*proof*⟩

lemma *mut-tso-lock-invL*[*intro*]:
  ⦃ *mut-m.tso-lock-invL m* ⦄ *sys*
⟨*proof*⟩

end

15 Valid refs inv proofs

lemma *valid-refs-inv-sweep-loop-free*:
  assumes *valid-refs-inv s*
  assumes *ngr*: *no-grey-refs s*
  assumes *rsi*: $\forall m'$. *mut-m.reachable-snapshot-inv* m' *s*
  assumes *white* r' *s*
  shows *valid-refs-inv* (*s*(*sys* := *s sys*⦇*heap* := (*sys-heap s*)(r' := *None*)⦈))
⟨*proof*⟩

lemma (**in** *gc*) *valid-refs-inv*[*intro*]:
  notes *fun-upd-apply*[*simp*]
  shows
  ⦃ *fM-fA-invL* ∧ *handshake-invL* ∧ *gc-W-empty-invL* ∧ *gc-mark.mark-object-invL* ∧ *obj-fields-marked-invL* ∧ *phase-invL* ∧ *sweep-loop-invL*
      ∧ *LSTP* (*handshake-phase-inv* ∧ *mutators-phase-inv* ∧ *sys-phase-inv* ∧ *valid-refs-inv* ∧ *valid-W-inv*) ⦄
    gc
   ⦃ *LSTP valid-refs-inv* ⦄
⟨*proof*⟩

context *mut-m*
begin

lemma *valid-refs-inv-discard-roots*:
  ⟦ *valid-refs-inv s*; *roots*$'$ ⊆ *mut-roots s* ⟧
    ⟹ *valid-refs-inv* (*s*(*mutator m* := *s* (*mutator m*)⦇*roots* := *roots*$'$⦈))
⟨*proof*⟩

lemma *valid-refs-inv-load*:
  ⟦ *valid-refs-inv s*; *sys-load* (*mutator m*) (*mr-Ref r f*) (*s sys*) = *mv-Ref* r'; *r* ∈ *mut-roots s* ⟧
    ⟹ *valid-refs-inv* (*s*(*mutator m* := *s* (*mutator m*)⦇*roots* := *mut-roots s* ∪ *Option.set-option* r'⦈))
⟨*proof*⟩

lemma *valid-refs-inv-alloc*:
  ⟦ *valid-refs-inv s*; *sys-heap s* r' = *None* ⟧
    ⟹ *valid-refs-inv* (*s*(*mutator m* := *s* (*mutator m*)⦇*roots* := *insert* r' (*mut-roots s*)⦈, *sys* := *s sys*⦇*heap* := (*sys-heap s*)(r' ↦ ⦇*obj-mark* = *fl*, *obj-fields* = *Map.empty*, *obj-payload* = *Map.empty*⦈)⦈))
⟨*proof*⟩

lemma *valid-refs-inv-store-ins*:
⟦ *valid-refs-inv s*; *r* ∈ *mut-roots s*; (∃ *r'*. *opt-r'* = *Some r'*) ⟶ *the opt-r'* ∈ *mut-roots s* ⟧
⟹ *valid-refs-inv* (*s*(*mutator m* := *s* (*mutator m*)⦇ *ghost-honorary-root* := {} ⦈,
sys := *s sys*⦇ *mem-store-buffers* := (*mem-store-buffers* (*s sys*))(*mutator m* := *sys-mem-store-buffers* (*mutator m*) *s* @ [*mw-Mutate r f opt-r'*]) ⦈))
⟨*proof*⟩

lemma *valid-refs-inv-deref-del*:
⟦ *valid-refs-inv s*; *sys-load* (*mutator m*) (*mr-Ref r f*) (*s sys*) = *mv-Ref opt-r'*; *r* ∈ *mut-roots s*; *mut-ghost-honorary-root s* = {} ⟧
⟹ *valid-refs-inv* (*s*(*mutator m* := *s* (*mutator m*)⦇*ghost-honorary-root* := *Option.set-option opt-r'*, *ref* := *opt-r'*⦈))
⟨*proof*⟩

lemma *valid-refs-inv-mo-co-mark*:
⟦ *r* ∈ *mut-roots s* ∪ *mut-ghost-honorary-root s*; *mut-ghost-honorary-grey s* = {}; *valid-refs-inv s* ⟧
⟹ *valid-refs-inv* (*s*(*mutator m* := *s* (*mutator m*)⦇*ghost-honorary-grey* := {*r*}⦈))
⟨*proof*⟩

lemma *valid-refs-inv*[*intro*]:
notes *fun-upd-apply*[*simp*]
notes *valid-refs-inv-discard-roots*[*simp*]
notes *valid-refs-inv-load*[*simp*]
notes *valid-refs-inv-alloc*[*simp*]
notes *valid-refs-inv-store-ins*[*simp*]
notes *valid-refs-inv-deref-del*[*simp*]
notes *valid-refs-inv-mo-co-mark*[*simp*]
shows
⦃ *mark-object-invL*
∧ *mut-get-roots.mark-object-invL m*
∧ *mut-store-del.mark-object-invL m*
∧ *mut-store-ins.mark-object-invL m*
∧ *LSTP valid-refs-inv* ⦄
mutator m
⦃ *LSTP valid-refs-inv* ⦄
⟨*proof*⟩

end

lemma (**in** *sys*) *valid-refs-inv*[*intro*]:
⦃ *LSTP* (*valid-refs-inv* ∧ *tso-store-inv*) ⦄ *sys* ⦃ *LSTP valid-refs-inv* ⦄
⟨*proof*⟩

16 Worklist invariants

lemma *valid-W-invD0*:
⟦ *r* ∈ *W* (*s p*); *valid-W-inv s*; *p* ≠ *q* ⟧ ⟹ *r* ∉ *WL q s*
⟦ *r* ∈ *W* (*s p*); *valid-W-inv s* ⟧ ⟹ *r* ∉ *ghost-honorary-grey* (*s q*)
⟦ *r* ∈ *ghost-honorary-grey* (*s p*); *valid-W-inv s* ⟧ ⟹ *r* ∉ *W* (*s q*)
⟦ *r* ∈ *ghost-honorary-grey* (*s p*); *valid-W-inv s*; *p* ≠ *q* ⟧ ⟹ *r* ∉ *WL q s*
⟨*proof*⟩

lemma *valid-W-distinct-simps*:
⟦*r* ∈ *ghost-honorary-grey* (*s p*); *valid-W-inv s*⟧ ⟹ (*r* ∈ *ghost-honorary-grey* (*s q*)) ⟷ (*p* = *q*)
⟦*r* ∈ *W* (*s p*); *valid-W-inv s*⟧ ⟹ (*r* ∈ *W* (*s q*)) ⟷ (*p* = *q*)
⟦*r* ∈ *WL p s*; *valid-W-inv s*⟧ ⟹ (*r* ∈ *WL q s*) ⟷ (*p* = *q*)

⟨*proof*⟩

lemma *valid-W-inv-sys-mem-store-buffersD*:
⟦ *sys-mem-store-buffers p s* = *mw-Mutate r′ f r″ # ws*; *mw-Mark r fl* ∈ *set ws*; *valid-W-inv s* ⟧
⟹ *fl* = *sys-fM s* ∧ *r* ∈ *ghost-honorary-grey* (*s p*) ∧ *tso-locked-by p s* ∧ *white r s* ∧ *filter is-mw-Mark ws* = [*mw-Mark r fl*]
⟦ *sys-mem-store-buffers p s* = *mw-fA fl′ # ws*; *mw-Mark r fl* ∈ *set ws*; *valid-W-inv s* ⟧
⟹ *fl* = *sys-fM s* ∧ *r* ∈ *ghost-honorary-grey* (*s p*) ∧ *tso-locked-by p s* ∧ *white r s* ∧ *filter is-mw-Mark ws* = [*mw-Mark r fl*]
⟦ *sys-mem-store-buffers p s* = *mw-fM fl′ # ws*; *mw-Mark r fl* ∈ *set ws*; *valid-W-inv s* ⟧
⟹ *fl* = *sys-fM s* ∧ *r* ∈ *ghost-honorary-grey* (*s p*) ∧ *tso-locked-by p s* ∧ *white r s* ∧ *filter is-mw-Mark ws* = [*mw-Mark r fl*]
⟦ *sys-mem-store-buffers p s* = *mw-Phase ph # ws*; *mw-Mark r fl* ∈ *set ws*; *valid-W-inv s* ⟧
⟹ *fl* = *sys-fM s* ∧ *r* ∈ *ghost-honorary-grey* (*s p*) ∧ *tso-locked-by p s* ∧ *white r s* ∧ *filter is-mw-Mark ws* = [*mw-Mark r fl*]
⟨*proof*⟩

lemma *valid-W-invE2*:
⟦ *r* ∈ *W* (*s p*); *valid-W-inv s*; ⋀*obj*. *obj-mark obj* = *sys-fM s* ⟹ *P obj*⟧ ⟹ *obj-at P r s*
⟦ *r* ∈ *ghost-honorary-grey* (*s p*); *sys-mem-lock s* ≠ *Some p*; *valid-W-inv s*; ⋀*obj*. *obj-mark obj* = *sys-fM s* ⟹ *P obj* ⟧ ⟹ *obj-at P r s*
⟨*proof*⟩

lemma (**in** *sys*) *valid-W-inv*[*intro*]:
notes *if-split-asm*[*split del*]
notes *fun-upd-apply*[*simp*]
shows
⦃ *LSTP* (*fM-rel-inv* ∧ *sys-phase-inv* ∧ *tso-store-inv* ∧ *valid-refs-inv* ∧ *valid-W-inv*) ⦄
sys
⦃ *LSTP valid-W-inv* ⦄
⟨*proof*⟩

lemma *valid-W-inv-ghg-disjoint*:
⟦ *white y s*; *sys-mem-lock s* = *Some p*; *valid-W-inv s*; *p0* ≠ *p1* ⟧
⟹ *WL p0* (*s*(*p* := *s p*⦇*ghost-honorary-grey* := {*y*}⦈)) ∩ *WL p1* (*s*(*p* := *s p*⦇*ghost-honorary-grey* := {*y*}⦈)) = {}
⟨*proof*⟩

lemma *valid-W-inv-mo-co-mark*:
⟦ *valid-W-inv s*; *white y s*; *sys-mem-lock s* = *Some p*; *filter is-mw-Mark* (*sys-mem-store-buffers p s*) = []; *p* ≠ *sys* ⟧
⟹ *valid-W-inv* (*s*(*p* := *s p*⦇*ghost-honorary-grey* := {*y*}⦈, *sys* := *s sys*⦇*mem-store-buffers* := (*mem-store-buffers* (*s sys*))(*p* := *sys-mem-store-buffers p s* @ [*mw-Mark y* (*sys-fM s*)])⦈))
⟨*proof*⟩

lemma *valid-W-inv-mo-co-lock*:
⟦ *valid-W-inv s*; *sys-mem-lock s* = *None* ⟧
⟹ *valid-W-inv* (*s*(*sys* := *s sys*⦇*mem-lock* := *Some p*⦈))
⟨*proof*⟩

lemma *valid-W-inv-mo-co-W*:
⟦ *valid-W-inv s*; *marked y s*; *ghost-honorary-grey* (*s p*) = {*y*}; *p* ≠ *sys* ⟧
⟹ *valid-W-inv* (*s*(*p* := *s p*⦇*W* := *insert y* (*W* (*s p*)), *ghost-honorary-grey* := {}⦈))
⟨*proof*⟩

lemma *valid-W-inv-mo-co-unlock*:

⟦ *sys-mem-lock s = Some p*; *sys-mem-store-buffers p s* = [];
  $\bigwedge$*r. r* ∈ *ghost-honorary-grey* (*s p*) ⟹ *marked r s*;
  valid-W-inv s
⟧ ⟹ *valid-W-inv* (*s*(*sys* := *mem-lock-update Map.empty* (*s sys*)))
⟨*proof*⟩

lemma (**in** *gc*) *valid-W-inv*[*intro*]:
notes *if-split-asm*[*split del*]
notes *fun-upd-apply*[*simp*]
shows
⦃ *gc-mark.mark-object-invL* ∧ *gc-W-empty-invL*
  ∧ *obj-fields-marked-invL*
  ∧ *sweep-loop-invL* ∧ *tso-lock-invL*
  ∧ *LSTP valid-W-inv* ⦄
  gc
⦃ *LSTP valid-W-inv* ⦄
⟨*proof*⟩

lemma (**in** *mut-m*) *valid-W-inv*[*intro*]:
notes *if-split-asm*[*split del*]
notes *fun-upd-apply*[*simp*]
shows
⦃ *handshake-invL* ∧ *mark-object-invL* ∧ *tso-lock-invL*
  ∧ *mut-get-roots.mark-object-invL m*
  ∧ *mut-store-del.mark-object-invL m*
  ∧ *mut-store-ins.mark-object-invL m*
  ∧ *LSTP* (*fM-rel-inv* ∧ *sys-phase-inv* ∧ *valid-refs-inv* ∧ *valid-W-inv*) ⦄
  mutator m
⦃ *LSTP valid-W-inv* ⦄
⟨*proof*⟩

17 Top-level safety

lemma (**in** *gc*) *I*:
⦃ *I* ⦄ *gc*
⟨*proof*⟩

lemma (**in** *sys*) *I*:
⦃ *I* ⦄ *sys*
⟨*proof*⟩

We need to separately treat the two cases of a single mutator and multiple mutators. In the latter case we have the additional obligation of showing mutual non-interference amongst mutators.

lemma *mut-invsL*[*intro*]:
⦃*I*⦄ *mutator m* ⦃*mut-m.invsL m'*⦄
⟨*proof*⟩

lemma *mutators-phase-inv*[*intro*]:
⦃ *I* ⦄ *mutator m* ⦃ *LSTP* (*mut-m.mutator-phase-inv m'*) ⦄
⟨*proof*⟩

lemma (**in** *mut-m*) *I*:
⦃ *I* ⦄ *mutator m*
⟨*proof*⟩

context *gc-system*
begin

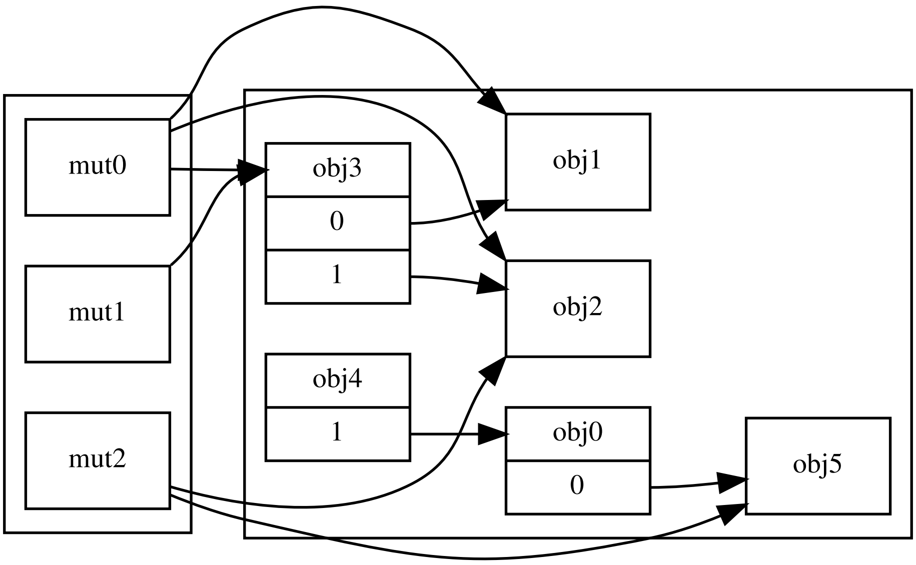

Figure 1: A concrete system state.

theorem *I*: *gc-system* $\models_{pre}$ *I*
⟨*proof*⟩

Our headline safety result follows directly.

corollary *safety*: *gc-system* $\models_{pre}$ *LSTP valid-refs*
⟨*proof*⟩

end

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

interpretation *gc-system-interpretation*: *gc-system undefined* ⟨*proof*⟩

18 A concrete system state

We demonstrate that our definitions are not vacuous by exhibiting a concrete initial state that satisfies the initial conditions. The heap is shown in Figure 1. We use Isabelle's notation for types of a given size.

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

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

abbreviation *mut-common-init-state* :: *concrete-local-state* **where**
mut-common-init-state ≡ *undefined*⦇ *ghost-hs-phase* := *hp-IdleMarkSweep*, *ghost-honorary-grey* := {}, *ghost-honorary-r*
:= {}, *roots* := {}, *W* := {} ⦈

context *gc-system*
begin

abbreviation *sys-init-heap* :: *ref* ⇒ (*field*, *payload*, *ref*) *object option* **where**
sys-init-heap ≡
[*0* ↦ (| *obj-mark* = *initial-mark*,
obj-fields = [*0* ↦ *5*],
obj-payload = *Map.empty* |),
1 ↦ (| *obj-mark* = *initial-mark*,
obj-fields = *Map.empty*,
obj-payload = *Map.empty* |),
2 ↦ (| *obj-mark* = *initial-mark*,
obj-fields = *Map.empty*,
obj-payload = *Map.empty* |),
3 ↦ (| *obj-mark* = *initial-mark*,
obj-fields = [*0* ↦ *1* , *1* ↦ *2*],
obj-payload = *Map.empty* |),
4 ↦ (| *obj-mark* = *initial-mark*,
obj-fields = [*1* ↦ *0*],
obj-payload = *Map.empty* |),
5 ↦ (| *obj-mark* = *initial-mark*,
obj-fields = *Map.empty*,
obj-payload = *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

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*
, *hs-pending* := ⟨*False*⟩
, *hs-type* := *ht-GetRoots*
, *mem-lock* := *None*
, *mem-store-buffers* := ⟨[]⟩
, *phase* := *ph-Idle*
, *W* := {}
, *ghost-honorary-grey* := {}
, *ghost-hs-in-sync* := ⟨*True*⟩
, *ghost-hs-phase* := *hp-IdleMarkSweep* |)

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

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

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

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

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

end

References

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

D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL'1994*, pages 70–83. ACM Press, 1994. doi: 10.1145/174675.174673.

P. 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.