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* ≠ *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 $l \equiv \{\!|l|\!\}$ *Request* ($\lambda s.$ (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* $\equiv \{\!|l|\!\}$ *Request* ($\lambda s.$ (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 $l \equiv \{\!|l|\!\}$ *Request* ($\lambda s.$ (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 $l \equiv \{\!|l|\!\}$ *Request* ($\lambda s.$ (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* $\equiv \{\!|l|\!\}$ *Request* ($\lambda s.$ (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 ≡ $\{\!|l|\!\}$ $\lfloor\!|\lambda s.$ $s(\!|$ W := W s ∪ {r s}, *ghost-honorary-grey* := {} $|\!|\rfloor$
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 @ ''-mo-co-unlock''|} unlock ;;
      {|l @ ''-mo-co-won''|} IF cas-mark = mark THEN
        {|l @ ''-mo-co-W''|} add-to-W (the ∘ ref)
      FI
    FI
  FI
FI
```

end

The worklists (field W) are not subject to TSO. As we later show (§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-hs-gc-set-type''|} Response
      (λreq s. { (s(| hs-type := ht,
                      ghost-hs-in-sync := ⟨False⟩,
                      ghost-hs-phase := hp-step ht (ghost-hs-phase s) |),
                  mv-Void)
               |ht. req = (gc, ro-hs-gc-store-type ht) })
  ⊕ {|''sys-hs-gc-mut-reqs''|} Response
      (λreq s. { (s(| hs-pending := (hs-pending s)(m := True) |), mv-Void)
               |m. req = (gc, ro-hs-gc-store-pending m) })
  ⊕ {|''sys-hs-gc-done''|} Response
      (λreq s. { (s, mv-Bool (¬hs-pending s m))
               |m. req = (gc, ro-hs-gc-load-pending m) })
  ⊕ {|''sys-hs-gc-load-W''|} Response
      (λreq s. { (s(| W := {} |), mv-Refs (W s))
               |-::unit. req = (gc, ro-hs-gc-load-W) })
  ⊕ {|''sys-hs-mut-pending''|} Response
      (λreq s. { (s, mv-Bool (hs-pending s m))
               |m. req = (mutator m, ro-hs-mut-load-pending) })
  ⊕ {|''sys-hs-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* (λ*s*. (*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* (λ*s*. (*mutator m*, *ro-hs-mut-load-pending*)) (λ*mv 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* (λ*s*. (*mutator m*, *ro-hs-mut-load-type*)) (λ*mv 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* (λ*s*. (*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* (λ*s*. (*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* (λ*s*. (*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 (λs. (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 (λs. (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 (λs. (gc, ro-hs-gc-load-W))
                                     (λresp s. {s(| W := W' |) | W'. resp = mv-Refs W'})

abbreviation mfence :: location ⇒ ('field, 'mut, 'payload, 'ref) gc-com (‹{|-|} MFENCE›) where
  {|l|} MFENCE ≡ {|l|} Request (λs. (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
            (λs. (gc, ro-hs-gc-load-pending (mut s)))
```

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

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

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

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

end
```

2.3 The system process

The system process models the environment in which the garbage collector and mutators execute. We translate the x86-TSO memory model due to Sewell et al. (2010) into a CIMP process. It is a reactive system: it receives requests and returns values, but initiates no communication itself. It can, however, autonomously commit a 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-blocked :: ('field, 'mut, 'payload, 'ref) local-state ⇒ 'mut process-name ⇒ bool
where
  not-blocked s p = (case mem-lock s of None ⇒ True | Some p' ⇒ p = p')
```

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

```
definition
  do-store-action :: ('field, 'payload, 'ref) mem-store-action ⇒ ('field, 'mut, 'payload, 'ref) local-state ⇒ ('field,
'mut, 'payload, 'ref) local-state
where
  do-store-action wact =
    (λs. case wact of
      mw-Mark r gc-mark     ⇒ s(|heap := (heap s)(r := map-option (λobj. obj(|obj-mark := gc-mark|)) (heap s
r))|)
    | mw-Mutate r f new-r  ⇒ s(|heap := (heap s)(r := map-option (λobj. obj(|obj-fields := (obj-fields obj)(f :=
new-r) |)) (heap s r))|)
    | mw-Mutate-Payload r f pl ⇒ s(|heap := (heap s)(r := map-option (λobj. obj(|obj-payload := (obj-payload
obj)(f := pl) |)) (heap s r))|)
    | mw-fM gc-mark         ⇒ s(|fM := gc-mark|)
    | mw-fA gc-mark         ⇒ s(|fA := gc-mark|)
```

```
  | mw-Phase gc-phase    ⇒ s(|phase := gc-phase|))

definition
  fold-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 (λw. (∘) (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 =
    (λs. case ract of
      mr-Ref r f ⇒ mv-Ref (Option.bind (heap s r) (λobj. obj-fields obj f))
    | mr-Payload r f ⇒ mv-Payload (Option.bind (heap s r) (λobj. 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 (λreq s. { (s, sys-load p mr s)
                                  |p mr. req = (p, ro-Load mr) ∧ not-blocked s p })
    ⊕ {|''tso-store''|} Response (λreq 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 (λreq s. { (s, mv-Void)
                                  |p. req = (p, ro-MFENCE) ∧ mem-store-buffers s p = [] })
    ⊕ {|''tso-lock''|} Response (λreq s. { (s(| mem-lock := Some p |), mv-Void)
                                  |p. req = (p, ro-Lock) ∧ mem-lock s = None })
    ⊕ {|''tso-unlock''|} Response (λreq s. { (s(| mem-lock := None |), mv-Void)
```

$|p.\ req = (p,\ ro\text{-}Unlock) \wedge mem\text{-}lock\ s = Some\ p \wedge mem\text{-}store\text{-}buffers\ s\ p = [] \})$
$\oplus \{\!|''tso\text{-}dequeue\text{-}store\text{-}buffer''|\!\}\ LocalOp\ (\lambda s.\ \{\ (do\text{-}store\text{-}action\ w\ s)(\!|\ mem\text{-}store\text{-}buffers := (mem\text{-}store\text{-}buffers\ s)(p := ws)\ |\!)$
$|\ p\ w\ ws.\ mem\text{-}store\text{-}buffers\ s\ p = w\ \#\ ws \wedge not\text{-}blocked\ s\ p \wedge p \neq sys\ \})$

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* ($\lambda req\ s.$
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 = $\{\!|''sys\text{-}free''|\!\}$ *Response* ($\lambda req\ s.$
{ (*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* ($\lambda s.$ (*mutator m*, *ro-Alloc*))
($\lambda mv\ s.$ { *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* (λ*s*. (*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* (λ*s*. (*gc*, *LoadfM*)) (λ*mv 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* (λ*s*. (*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* (λ*s*. (*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* (λ*s*. (*gc*, *ro-Free* (*r s*))) (λ- *s*. {*s*})

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

definition (**in** *gc*)
com :: (*'field*, *'mut*, *'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)
by (auto simp: Set.bind-def)

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
by (case-tac [!] f) simp-all

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 ≡ λ*s*.
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*
by (*auto simp*: *black-def white-def obj-at-def split*: *option.splits*)

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*
by (*auto simp*: *black-def grey-def white-def obj-at-def split*: *option.splits*)

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 = (∃ *x*. *root x* ∧ *x reaches y*)

definition *grey-reachable* :: *'ref* ⇒ (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* **where**
grey-reachable y = (∃ *g*. *grey g* ∧ *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 ≠ *sys* ⟷ *p* = *gc* ∨ (∃ *m*. *p* = *mutator m*)
by (*cases p*) *simp-all*

lemma (**in** *mut-m'*) *m'm*[*iff*]: *m'* ≠ *m*
using *mm'* **by** *blast*

obj at

lemma *obj-at-cong*[*cong*]:
⟦⋀*obj*. *sys-heap s r* = *Some obj* ⟹ *P obj* = *P' obj*; *r* = *r'*; *s* = *s'*⟧
⟹ *obj-at P r s* ⟷ *obj-at P' r' s'*
unfolding *obj-at-def* **by** (*simp cong*: *option.case-cong*)

lemma *obj-at-split*:
Q (*obj-at P r s*) = ((*sys-heap s r* = *None* ⟶ *Q False*) ∧ (∀ *obj*. *sys-heap s r* = *Some obj* ⟶ *Q* (*P obj*)))
by (*simp add*: *obj-at-def split*: *option.splits*)

lemma *obj-at-split-asm*:
Q (*obj-at P r s*) = (¬ ((*sys-heap s r* = *None* ∧ ¬*Q False*) ∨ (∃ *obj*. *sys-heap s r* = *Some obj* ∧ ¬ *Q* (*P obj*))))
by (*simp add*: *obj-at-def split*: *option.splits*)

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*)
by (*simp add*: *eq-imp-def obj-at-def split*: *option.splits*)

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

lemma *obj-at-simps*:
obj-at (λ*obj*. *P obj* ∧ *Q obj*) *r s* ⟷ *obj-at P r s* ∧ *obj-at Q r s*

by (*simp-all split*: *obj-at-splits*)

obj at field on heap

lemma *obj-at-field-on-heap-cong*[*cong*]:
⟦⋀*r' obj*. ⟦*sys-heap s r* = *Some obj*; *obj-fields obj f* = *Some r'*⟧⟹ *P r'* = *P' r'*; *r* = *r'*; *f* = *f'*; *s* = *s'*⟧
⟹ *obj-at-field-on-heap P r f s* ⟷ *obj-at-field-on-heap P' r' f' s'*
unfolding *obj-at-field-on-heap-def* **by** (*simp cong*: *option.case-cong*)

lemma *obj-at-field-on-heap-split*:
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*)

$\wedge$ ($\forall r'$ *obj*. *sys-heap s r* = *Some obj* $\wedge$ *obj-fields obj f* = *Some r'* $\longrightarrow$ *Q* (*P r'*)))
by (*simp add*: *obj-at-field-on-heap-def split*: *option.splits*)

lemma *obj-at-field-on-heap-split-asm*:
Q (*obj-at-field-on-heap P r f s*) $\longleftrightarrow$ ($\neg$ ((*sys-heap s r* = *None* $\wedge$ $\neg$*Q False*)
$\vee$ ($\exists$ *obj*. *sys-heap s r* = *Some obj* $\wedge$ *obj-fields obj f* = *None* $\wedge$ $\neg$ *Q True*)
$\vee$ ($\exists r'$ *obj*. *sys-heap s r* = *Some obj* $\wedge$ *obj-fields obj f* = *Some r'* $\wedge$ $\neg$ *Q* (*P r'*))))
by (*simp add*: *obj-at-field-on-heap-def split*: *option.splits*)

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*)
by (*simp add*: *eq-imp-def obj-at-field-on-heap-def*)

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 $\Longrightarrow$ *valid-ref r s*
obj-at-field-on-heap P r f s $\Longrightarrow$ *valid-null-ref* (*Some r*) *s*
by (*auto simp*: *obj-at-field-on-heap-def valid-null-ref-def split*: *obj-at-splits option.splits*)

lemma *obj-at-field-on-heapE*[*elim*]:
$[\![$ *obj-at-field-on-heap P r f s*; *sys-heap s' r* = *sys-heap s r*; $\bigwedge r'$. *P r'* $\Longrightarrow$ *P' r'* $]\!]$
$\Longrightarrow$ *obj-at-field-on-heap P' r f s'*
by (*simp add*: *obj-at-field-on-heap-def split*: *option.splits*)

lemma *valid-null-ref-eq-imp*:
eq-imp (λ(-::*unit*) *s*. *Option.bind r* (*map-option* $\langle$*True*$\rangle$ $\circ$ *sys-heap s*))
(*valid-null-ref r*)
by (*simp add*: *eq-imp-def obj-at-def valid-null-ref-def split*: *option.splits*)

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* $\longleftrightarrow$ *valid-ref r s*
unfolding *valid-null-ref-def* **by** *simp-all*

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* $\longleftrightarrow$ (*fM s* = *fl* $\wedge$ *w* $\neq$ *mw-fM* (¬*fM s*)) $\vee$ *w* = *mw-fM fl*
fl = *fM* (*do-store-action w s*) $\longleftrightarrow$ (*fl* = *fM s* $\wedge$ *w* $\neq$ *mw-fM* (¬*fM s*)) $\vee$ *w* = *mw-fM fl*
fA (*do-store-action w s*) = *fl* $\longleftrightarrow$ (*fA s* = *fl* $\wedge$ *w* $\neq$ *mw-fA* (¬*fA s*)) $\vee$ *w* = *mw-fA fl*
fl = *fA* (*do-store-action w s*) $\longleftrightarrow$ (*fl* = *fA s* $\wedge$ *w* $\neq$ *mw-fA* (¬*fA s*)) $\vee$ *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* $\longleftrightarrow$ *heap s r* = *None*
mem-lock (*do-store-action w s*) = *mem-lock s*

phase (*do-store-action w s*) = *ph* ⟷ (*phase s* = *ph* ∧ (∀ *ph'*. *w* ≠ *mw-Phase ph'*) ∨ *w* = *mw-Phase ph*)
ph = *phase* (*do-store-action w s*) ⟷ (*ph* = *phase s* ∧ (∀ *ph'*. *w* ≠ *mw-Phase ph'*) ∨ *w* = *mw-Phase ph*)
W (*do-store-action w s*) = *W s*
by (*auto simp*: *do-store-action-def fun-upd-apply split*: *mem-store-action.splits obj-at-splits*)

reaches

lemma *reaches-refl*[*iff*]:
(*r reaches r*) *s*
unfolding *reaches-def* **by** *blast*

lemma *reaches-step*[*intro*]:
⟦(*x reaches y*) *s*; (*y points-to z*) *s*⟧ ⟹ (*x reaches z*) *s*
⟦(*y reaches z*) *s*; (*x points-to y*) *s*⟧ ⟹ (*x reaches z*) *s*
unfolding *reaches-def*
apply (*simp add*: *rtranclp.rtrancl-into-rtrancl*)
apply (*simp add*: *converse-rtranclp-into-rtranclp*)
done

lemma *reaches-induct*[*consumes 1*, *case-names refl step*, *induct set*: *reaches*]:
assumes (*x reaches y*) *s*
assumes ⋀*x*. *P x x*
assumes ⋀*x y z*. ⟦(*x reaches y*) *s*; *P x y*; (*y points-to z*) *s*⟧ ⟹ *P x z*
shows *P x y*
using *assms* **unfolding** *reaches-def* **by** (*rule rtranclp.induct*)

lemma *converse-reachesE*[*consumes 1*, *case-names base step*]:
assumes (*x reaches z*) *s*
assumes *x* = *z* ⟹ *P*
assumes ⋀*y*. ⟦(*x points-to y*) *s*; (*y reaches z*) *s*⟧ ⟹ *P*
shows *P*
using *assms* **unfolding** *reaches-def* **by** (*blast elim*: *converse-rtranclpE*)

lemma *reaches-fields*: — Complicated condition takes care of *alloc*: collapses no object and object with no fields
assumes (*x reaches y*) *s'*
assumes ∀ *r'*. ⋃(*ran* ‘ *obj-fields* ‘ *set-option* (*sys-heap s' r'*)) = ⋃(*ran* ‘ *obj-fields* ‘ *set-option* (*sys-heap s r'*))
shows (*x reaches y*) *s*
using *assms*
proof *induct*
case (*step x y z*)
then have (*y points-to z*) *s*
by (*cases sys-heap s y*)
(*auto 10 10 simp*: *ran-def obj-at-def split*: *option.splits dest*!: *spec*[**where** *x*=*y*])
with *step* **show** *?case* **by** *blast*
qed *simp*

lemma *reaches-eq-imp*:
eq-imp (λ*r' s*. ⋃(*ran* ‘ *obj-fields* ‘ *set-option* (*sys-heap s r'*)))
(*x reaches y*)
unfolding *eq-imp-def* **by** (*metis reaches-fields*)

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-at P r (*s*(*sys* := *s sys*(| *heap* := (*sys-heap s*)(*r'* := *map-option* (*obj-mark-update* (λ-. *fl*)) (*sys-heap s r'*)), *mem-store-buffers* := *wb'* |)))
⟷ *obj-at* (λ*obj*. (*P* (*if r* = *r' then obj*(| *obj-mark* := *fl* |) *else obj*))) *r s*
by (*clarsimp simp*: *fun-upd-apply split*: *obj-at-splits*)

lemma *obj-at-field-on-heap-mw-simps*[*simp*]:
obj-at-field-on-heap P r0 f0
(*s*(*sys* := (*s sys*)(| *heap* := (*sys-heap s*)(*r* := *map-option* (λ*obj* :: ('*field*, '*payload*, '*ref*) *object*. *obj*(|*obj-fields* := (*obj-fields obj*)(*f* := *opt-r'*)|)) (*sys-heap s r*)),
mem-store-buffers := (*mem-store-buffers* (*s Sys*))(*p* := *ws*) |)))
⟷ ((*r* ≠ *r0* ∨ *f* ≠ *f0*) ∧ *obj-at-field-on-heap P r0 f0 s*)
∨ (*r* = *r0* ∧ *f* = *f0* ∧ *valid-ref r s* ∧ (*case opt-r'* *of Some r''* ⇒ *P r''* | - ⇒ *True*))
obj-at-field-on-heap P r f (*s*(*sys* := *s sys*(|*heap* := (*sys-heap s*)(*r'* := *map-option* (*obj-mark-update* (λ-. *fl*)) (*sys-heap s r'*)), *mem-store-buffers* := *sb'*|)))
⟷ *obj-at-field-on-heap P r f s*
by (*auto simp*: *obj-at-field-on-heap-def fun-upd-apply split*: *option.splits obj-at-splits*)

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* ⟧
⟹ *obj-at-field-on-heap* (λ*r*. *opt-r'* = *Some r*) *r f s*
unfolding *sys-load-def fold-stores-def*
apply *clarsimp*
apply (*rule fold-invariant*[**where** *P*=λ*fr*. *obj-at-field-on-heap* (λ*r'*. *Option.bind* (*heap* (*fr* (*s sys*)) *r*) (λ*obj*. *obj-fields obj f*) = *Some r'*) *r f s*
and *Q*=λ*w*. *w* ∈ *set* (*sys-mem-store-buffers* (*mutator m*) *s*)])
apply *fastforce*
apply (*fastforce simp*: *obj-at-field-on-heap-def split*: *option.splits obj-at-splits*)
apply (*auto simp*: *do-store-action-def map-option-case fun-upd-apply*
split: *obj-at-field-on-heap-splits option.splits obj-at-splits mem-store-action.splits*)
done

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* ⟹ *weak-tricolour-inv s*
by (*clarsimp simp*: *strong-tricolour-inv-def weak-tricolour-inv-def grey-protects-white-def*)

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*
∧ *phase-rel-inv*
∧ *strong-tricolour-inv*
∧ *sys-phase-inv*
∧ *tso-store-inv*
∧ *valid-refs-inv*
∧ *valid-W-inv*
∧ *mutators-phase-inv*
∧ *fA-rel-inv* ∧ *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*
∧ *phase s* = *ph-Idle*
∧ *ghost-honorary-grey s* = {}
∧ *W s* = {})

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

definition *sys-initial-state* :: ('*field*, '*mut*, '*payload*, '*ref*) *lst-pred* **where**
sys-initial-state s =
((∀ *m*. ¬*hs-pending s m* ∧ *ghost-hs-in-sync s m*)
∧ *ghost-hs-phase s* = *hp-IdleMarkSweep* ∧ *hs-type s* = *ht-GetRoots*
∧ *obj-mark* ‘ *ran* (*heap s*) ⊆ {*initial-mark*}
∧ *fA s* = *initial-mark*
∧ *fM s* = *initial-mark*
∧ *phase s* = *ph-Idle*
∧ *ghost-honorary-grey s* = {}
∧ *W s* = {}
∧ (∀ *p*. *mem-store-buffers s p* = [])
∧ *mem-lock s* = *None*)

abbreviation
root-reachable y ≡ ∃ *m x*. ⟨*x*⟩ ∈ *mut-m.mut-roots m* ∧ *x reaches y*

definition *valid-refs* :: ('*field*, '*mut*, '*payload*, '*ref*) *lsts-pred* **where**
valid-refs = (∀ *y*. *root-reachable y* ⟶ *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*⟩)

$\wedge$ *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* }
$\cup$ (*prefixed* ″*hs-noop*″)
$\cup$ (*prefixed* ″*hs-get-roots*″)
$\cup$ (*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*⟩)
$\wedge$ *atS-mut hs-get-roots-locs* (*sys-hs-type* = ⟨*ht-GetRoots*⟩)
$\wedge$ *atS-mut hs-get-work-locs* (*sys-hs-type* = ⟨*ht-GetWork*⟩)
$\wedge$ *atS-mut ht-loaded-locs* (*mut-hs-pending* $\longrightarrow$ *mut-hs-type* = *sys-hs-type*)
$\wedge$ *atS-mut hs-pending-loaded-locs* (*mut-hs-pending* $\longrightarrow$ *sys-hs-pending m*)
$\wedge$ *atS-mut hs-pending-locs* (*mut-hs-pending*)
$\wedge$ *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* }
$\cup$ *prefixed* ″*idle-noop*″
$\cup$ (*mark-locs* − { *mark-load-fM*, *mark-store-fA*, *mark-noop-mfence* })
$\cup$ *sweep-locs*

locset-definition
fA-neq-locs = { *idle-phase-init*, *idle-store-fM*, *mark-load-fM*, *mark-store-fA* }
$\cup$ *prefixed* ″*idle-flip-noop*″
$\cup$ *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*)
$\wedge$ *at-gc idle-store-fM* (*gc-fM* $\neq$ *sys-fM*)
$\wedge$ *at-gc idle-flip-noop-mfence* (*sys-fM* $\neq$ *gc-fM* $\longrightarrow$ $\neg$*LIST-NULL* (*tso-pending-fM gc*))
$\wedge$ *atS-gc fM-tso-empty-locs* (*LIST-NULL* (*tso-pending-fM gc*))

$\wedge$ *atS-gc fA-eq-locs* (*gc-fM* = *sys-fA*)
$\wedge$ *atS-gc fA-neq-locs* (*gc-fM* $\neq$ *sys-fA*)
$\wedge$ *at-gc mark-noop-mfence* (*gc-fM* $\neq$ *sys-fA* $\longrightarrow$ $\neg$*LIST-NULL* (*tso-pending-fA gc*))
$\wedge$ *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* $\equiv$ *cas-mark* (*s p*)
abbreviation (*input*) *p-mark s* $\equiv$ *mark* (*s p*)
abbreviation (*input*) *p-fM s* $\equiv$ *fM* (*s p*)
abbreviation (*input*) *p-ghost-hs-phase s* $\equiv$ *ghost-hs-phase* (*s p*)
abbreviation (*input*) *p-ghost-honorary-grey s* $\equiv$ *ghost-honorary-grey* (*s p*)
abbreviation (*input*) *p-ghost-hs-in-sync s* $\equiv$ *ghost-hs-in-sync* (*s p*)
abbreviation (*input*) *p-phase s* $\equiv$ *phase* (*s p*)
abbreviation (*input*) *p-ref s* $\equiv$ *ref* (*s p*)
abbreviation (*input*) *p-the-ref* $\equiv$ *the* $\circ$ *p-ref*
abbreviation (*input*) *p-W s* $\equiv$ *W* (*s p*)

abbreviation *at-p* :: *location* $\Rightarrow$ (*'field*, *'mut*, *'payload*, *'ref*) *lsts-pred* $\Rightarrow$ (*'field*, *'mut*, *'payload*, *'ref*) *gc-pred*
where
at-p l' P $\equiv$ *at p* (*l* @ *l'*) $\longrightarrow$ *LSTP P*

abbreviation (*input*) *p-en-cond P* $\equiv$ *p-ph-enabled* $\longrightarrow$ *P*

abbreviation (*input*) *p-valid-ref* $\equiv$ $\neg$*NULL p-ref* $\wedge$ *valid-ref* **$** *p-the-ref*
abbreviation (*input*) *p-tso-no-pending-mark* $\equiv$ *LIST-NULL* (*tso-pending-mark p*)
abbreviation (*input*) *p-tso-no-pending-mutate* $\equiv$ *LIST-NULL* (*tso-pending-mutate p*)

abbreviation (*input*)
p-valid-W-inv $\equiv$ ((*p-cas-mark* $\neq$ *p-mark* $\vee$ *p-tso-no-pending-mark*) $\longrightarrow$ *marked* **$** *p-the-ref*)
$\wedge$ (*tso-pending-mark p* $\in$ (λ*s*. {[], [*mw-Mark* (*p-the-ref s*) (*p-fM s*)]}))

abbreviation (*input*)
p-mark-inv $\equiv$ $\neg$*NULL p-mark*
$\wedge$ ((λ*s*. *obj-at* (λ*obj*. *Some* (*obj-mark obj*) = *p-mark s*) (*p-the-ref s*) *s*)
$\vee$ *marked* **$** *p-the-ref*)

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

abbreviation (*input*) *p-valid-fM* $\equiv$ *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*⟩
by *standard* (*simp add*: *eq-imp-def*)

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*
by *standard* (*simp add*: *eq-imp-def*)

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*
by *standard* (*simp add*: *eq-imp-def*)

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*
by *standard* (*simp add*: *eq-imp-def*)

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*⟩)

$\wedge$ *atS-mut hs-not-hp-Idle-locs* (*mut-ghost-hs-phase* $\neq \langle$*hp-Idle*$\rangle$)

$\wedge$ *atS-mut mo-ptest-locs* (*mut-phase* $= \langle$*ph-Idle*$\rangle \longrightarrow$ (*mut-ghost-hs-phase* $\in \langle\{$*hp-Idle*, *hp-IdleInit*$\}\rangle$
$\vee$ (*mut-ghost-hs-phase* $= \langle$*hp-IdleMarkSweep*$\rangle$
$\wedge$ *sys-phase* $= \langle$*ph-Idle*$\rangle$))))

$\wedge$ *atS-mut ghost-honorary-grey-empty-locs* (*EMPTY mut-ghost-honorary-grey*)

— insertion barrier

$\wedge$ *at-mut store-ins* ((*mut-ghost-hs-phase* $\in \langle\{$*hp-InitMark*, *hp-Mark*$\}\rangle$
$\vee$ (*mut-ghost-hs-phase* $= \langle$*hp-IdleMarkSweep*$\rangle \wedge$ *sys-phase* $\neq \langle$*ph-Idle*$\rangle$))
$\wedge \neg$*NULL mut-new-ref*
$\longrightarrow$ *marked* **$** *mut-the-new-ref*)

$\wedge$ *atS-mut ins-barrier-locs* (((*mut-ghost-hs-phase* $= \langle$*hp-Mark*$\rangle$
$\vee$ (*mut-ghost-hs-phase* $= \langle$*hp-IdleMarkSweep*$\rangle \wedge$ *sys-phase* $\neq \langle$*ph-Idle*$\rangle$))
$\wedge$ (λ*s*. $\forall$ *opt-r'*. $\neg$*tso-pending-store* (*mutator m*) (*mw-Mutate* (*mut-tmp-ref s*) (*mut-field s*) *opt-r'*) *s*)
$\longrightarrow$ (λ*s*. *obj-at-field-on-heap* (λ*r'*. *marked r' s*) (*mut-tmp-ref s*) (*mut-field s*) *s*))
$\wedge$ (*mut-ref* $=$ *mut-new-ref*))

— deletion barrier

$\wedge$ *atS-mut del-barrier1-locs* ((*mut-ghost-hs-phase* $= \langle$*hp-Mark*$\rangle$
$\vee$ (*mut-ghost-hs-phase* $= \langle$*hp-IdleMarkSweep*$\rangle \wedge$ *sys-phase* $\neq \langle$*ph-Idle*$\rangle$))
$\wedge$ (λ*s*. $\forall$ *opt-r'*. $\neg$*tso-pending-store* (*mutator m*) (*mw-Mutate* (*mut-tmp-ref s*) (*mut-field s*) *opt-r'*) *s*)
$\longrightarrow$ (λ*s*. *obj-at-field-on-heap* (λ*r*. *mut-ref s* $=$ *Some r* $\vee$ *marked r s*) (*mut-tmp-ref s*) (*mut-field s*) *s*))

$\wedge$ *at-mut lop-store-ins* ((*mut-ghost-hs-phase* $= \langle$*hp-Mark*$\rangle$
$\vee$ (*mut-ghost-hs-phase* $= \langle$*hp-IdleMarkSweep*$\rangle \wedge$ *sys-phase* $\neq \langle$*ph-Idle*$\rangle$))
$\wedge \neg$*NULL mut-ref*
$\longrightarrow$ *marked* **$** *mut-the-ref*)

— after *init-noop*. key: no pending white insertions *at-mut hs-noop-done* which we get from *handshake-invL*.

$\wedge$ *at-mut mut-load* (*mut-tmp-ref* $\in$ *mut-roots*)

$\wedge$ *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* $\wedge$ *sys-ghost-hs-in-sync m* $\wedge \neg$*EMPTY* (*WL* (*mutator m*)))
$\longrightarrow$ ($\exists$ *m'*. $\neg$*sys-ghost-hs-in-sync m'* $\wedge \neg$*EMPTY* (*WL* (*mutator m'*))))

context *gc*
begin

locset-definition *gc-W-empty-locs* :: *location set* **where**
gc-W-empty-locs =
idle-locs $\cup$ *init-locs* $\cup$ *sweep-locs* $\cup$ {*mark-load-fM*, *mark-store-fA*, *mark-end*}
$\cup$ *prefixed* ''*mark-noop*''
$\cup$ *prefixed* ''*mark-loop-get-roots*''
$\cup$ *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*⦄
using *assms*
apply (*clarsimp simp*: *valid-proc-def*)

apply (*drule* (*1*) *bspec*)
apply (*auto elim*: *vcg-pre*)
done

lemma *valid-conj-lift*:
  assumes *x*: $\{\!|P|\!\}\ p\ \{\!|Q|\!\}$
  assumes *y*: $\{\!|P'|\!\}\ p\ \{\!|Q'|\!\}$
  shows $\{\!|P \wedge P'|\!\}\ p\ \{\!|Q \wedge Q'|\!\}$
apply (*clarsimp simp*: *valid-proc-def*)
apply (*rule vcg-conj*)
apply (*rule vcg-pre*[*OF spec*[*OF spec*[*OF x*[*unfolded Ball-def valid-proc-def split-paired-All*]], *simplified*, *rule-format*]], *simp*, *simp*)
apply (*rule vcg-pre*[*OF spec*[*OF spec*[*OF y*[*unfolded Ball-def valid-proc-def split-paired-All*]], *simplified*, *rule-format*]], *simp*, *simp*)
done

lemma *valid-all-lift*:
  assumes $\bigwedge x.\ \{\!|P\ x|\!\}\ p\ \{\!|Q\ x|\!\}$
  shows $\{\!|\lambda s.\ \forall x.\ P\ x\ s|\!\}\ p\ \{\!|\lambda s.\ \forall x.\ Q\ x\ s|\!\}$
using *assms* **by** (*fastforce simp*: *valid-proc-def intro*: *vcg-all-lift*)

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*:
  $[\![\ (\{\!|l|\!\}\ Response\ action,\ afts) \in fragments\ (gc\text{-}coms\ p)\ \{\};\ v \in action\ x\ s;$
    $[\![\ p = sys;\ ?P\ ]\!] \Longrightarrow Q\ ]\!] \Longrightarrow Q$
apply (*cases p*)
apply (*simp-all add*: *all-com-interned-defs*)
apply *atomize*

apply (*drule-tac P*=$x \vee y$ **and** *Q*=$v \in action\ p\ k$ **for** *x y p k* **in** *conjI*, *assumption*)
apply (*thin-tac* $v \in action\ p\ k$ **for** *p k*)
apply (*simp only*: *conj-disj-distribR conj-assoc mem-Collect-eq cong*: *conj-cong*)

apply (*erule mp*)
apply (*thin-tac p = sys*)
apply (*assumption*)
done

schematic-goal *system-responds-action-caseE*:
  $[\![\ (\{\!|l|\!\}\ Response\ action,\ afts) \in fragments\ (gc\text{-}coms\ p)\ \{\};\ v \in action\ (pname,\ req)\ s;$
    $[\![\ p = sys;\ case\text{-}request\text{-}op\ ?P1\ ?P2\ ?P3\ ?P4\ ?P5\ ?P6\ ?P7\ ?P8\ ?P9\ ?P10\ ?P11\ ?P12\ ?P13\ ?P14\ req\ ]\!] \Longrightarrow Q\ ]\!] \Longrightarrow Q$
apply (*erule*(*1*) *system-responds-actionE*)
apply (*cases req*; *simp only*: *request-op.simps prod.inject simp-thms fst-conv snd-conv if-cancel empty-def*[*symmetric*] *empty-iff*)
apply (*drule meta-mp*[*OF - TrueI*], *erule meta-mp*, *erule-tac P*=$A \wedge B$ **for** *A B* **in** *triv*)+
done

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

$\Longrightarrow Q \rrbracket \Longrightarrow Q$
apply (*erule system-responds-action-caseE*[**where** *pname*=*fst x* **and** *req*=*snd x*])
apply *simp*
apply *assumption*
done

6.2.2 Locations

lemma *atS-dests*:
$\llbracket\ atS\ p\ ls\ s;\ atS\ p\ ls'\ s\ \rrbracket \Longrightarrow atS\ p\ (ls \cup ls')\ s$
$\llbracket\ \neg atS\ p\ ls\ s;\ \neg atS\ p\ ls'\ s\ \rrbracket \Longrightarrow \neg atS\ p\ (ls \cup ls')\ s$
$\llbracket\ \neg atS\ p\ ls\ s;\ atS\ p\ ls'\ s\ \rrbracket \Longrightarrow atS\ p\ (ls' - ls)\ s$
$\llbracket\ \neg atS\ p\ ls\ s;\ at\ p\ l\ s\ \rrbracket \Longrightarrow atS\ p\ (\{l\} - ls)\ s$
by (*auto simp*: *atS-def*)

lemma *schematic-prem*: $\llbracket Q \Longrightarrow P;\ Q \rrbracket \Longrightarrow P$
by *blast*

lemma *TrueE*: $\llbracket True;\ P \rrbracket \Longrightarrow P$
by *blast*

lemma *thin-locs-pre-discardE*:
$\llbracket at\ p\ l'\ s \longrightarrow P;\ at\ p\ l\ s;\ l' \neq l;\ Q \rrbracket \Longrightarrow Q$
$\llbracket atS\ p\ ls\ s \longrightarrow P;\ at\ p\ l\ s;\ l \notin ls;\ Q \rrbracket \Longrightarrow Q$
unfolding *atS-def* **by** *blast+*

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

lemma *thin-locs-pre-keep-atSE*:
$\llbracket atS\ p\ ls\ s \longrightarrow P;\ at\ p\ l\ s;\ l \in ls;\ P \Longrightarrow Q \rrbracket \Longrightarrow Q$
unfolding *atS-def* **by** *blast*

lemma *thin-locs-post-discardE*:
$\llbracket AT\ s' = (AT\ s)(p := lfn,\ q := lfn');\ l' \notin lfn;\ p \neq q \rrbracket \Longrightarrow at\ p\ l'\ s' \longrightarrow P$
$\llbracket AT\ s' = (AT\ s)(p := lfn);\ l' \notin lfn \rrbracket \Longrightarrow at\ p\ l'\ s' \longrightarrow P$
$\llbracket AT\ s' = (AT\ s)(p := lfn,\ q := lfn');\ \bigwedge l.\ l \in lfn \Longrightarrow l \notin ls;\ \ p \neq q \rrbracket \Longrightarrow atS\ p\ ls\ s' \longrightarrow P$
$\llbracket AT\ s' = (AT\ s)(p := lfn);\ \bigwedge l.\ l \in lfn \Longrightarrow l \notin ls \rrbracket \Longrightarrow atS\ p\ ls\ s' \longrightarrow P$
unfolding *atS-def* **by** (*auto simp*: *fun-upd-apply*)

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-locs-post-discardE*(*4*)]

lemma *thin-locs-post-keep-locsE*:
$\llbracket (L \longrightarrow P) \wedge R;\ R \Longrightarrow Q \rrbracket \Longrightarrow (L \longrightarrow P) \wedge Q$
$L \longrightarrow P \Longrightarrow L \longrightarrow P$
by *blast+*

lemma *thin-locs-post-keepE*:
$\llbracket P \wedge R;\ R \Longrightarrow Q \rrbracket \Longrightarrow (L \longrightarrow P) \wedge Q$
$P \Longrightarrow L \longrightarrow P$
by *blast+*

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

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

unfolding *atS-def* **by** *auto*

lemma *ni-thin-locs-keep-atE*:
$[\![at\ proc\ l\ s \longrightarrow P;\ AT\ s' = (AT\ s)(p := lfn,\ q := lfn');\ at\ proc\ l\ s';\ proc \neq p;\ proc \neq q;\ P \Longrightarrow Q]\!] \Longrightarrow Q$
$[\![at\ proc\ l\ s \longrightarrow P;\ AT\ s' = (AT\ s)(p := lfn);\ at\ proc\ l\ s';\ proc \neq p;\ P \Longrightarrow Q]\!] \Longrightarrow Q$
by (*auto simp*: *fun-upd-apply*)

lemma *ni-thin-locs-keep-atSE*:
$[\![atS\ proc\ ls\ s \longrightarrow P;\ AT\ s' = (AT\ s)(p := lfn,\ q := lfn');\ at\ proc\ l'\ s';\ l' \in ls;\ proc \neq p;\ proc \neq q;\ P \Longrightarrow Q]\!]$
$\Longrightarrow Q$
$[\![atS\ proc\ ls\ s \longrightarrow P;\ AT\ s' = (AT\ s)(p := lfn);\ at\ proc\ l'\ s';\ l' \in ls;\ proc \neq p;\ P \Longrightarrow Q]\!] \Longrightarrow Q$
$[\![atS\ proc\ ls\ s \longrightarrow P;\ AT\ s' = (AT\ s)(p := lfn,\ q := lfn');\ atS\ proc\ ls'\ s';\ ls' \subseteq ls;\ proc \neq p;\ proc \neq q;\ P \Longrightarrow Q]\!]$
$\Longrightarrow Q$
$[\![atS\ proc\ ls\ s \longrightarrow P;\ AT\ s' = (AT\ s)(p := lfn);\ atS\ proc\ ls'\ s';\ ls' \subseteq ls;\ proc \neq p;\ P \Longrightarrow Q]\!] \Longrightarrow Q$
unfolding *atS-def* **by** (*auto simp*: *fun-upd-apply*)

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

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 ‹

```
signature GC-VCG =
sig
  (* Internals *)
  val nuke-schematic-prems : Proof.context -> int -> tactic
  val loc-mem-tac : Proof.context -> int -> tactic
  val vcg-fragments-tac : Proof.context -> int -> tactic
  val vcg-sem-tac : Proof.context -> int -> tactic
  val thin-pre-inv-tac : Proof.context -> int -> tactic
  val thin-post-inv-tac : bool -> Proof.context -> int -> tactic
  val vcg-inv-tac : bool -> bool -> Proof.context -> int -> tactic
  (* End-user tactics *)
  val vcg-jackhammer-tac : bool -> bool -> Proof.context -> int -> tactic
  val vcg-chainsaw-tac : bool -> thm list -> Proof.context -> int -> tactic
  val vcg-name-cases-tac : term list -> thm list -> context-tactic
end

structure GC-VCG : GC-VCG =
struct

(* Identify and remove schematic premises. FIXME reverses the prems *)
fun nuke-schematic-prems ctxt =
  let
    fun is-schematic-prem t =
      case t of
        Const (HOL.Trueprop, -) $ t => is-schematic-prem t
      | t $ - => is-schematic-prem t
      | Var - => true
      | - => false
  in
    DETERM o filter-prems-tac ctxt (not o is-schematic-prem)
  end;

(* FIXME Want to unify only with a non-schematic prem. might get there with first order matching or some
existing variant of assume. *)
fun assume-non-schematic-prem-tac ctxt =
  (TRY o nuke-schematic-prems ctxt) THEN' assume-tac ctxt

fun vcg-fragments-tac ctxt =
 SELECT-GOAL (HEADGOAL (safe-simp-tac (ss-only (@{thms vcg-fragments'-simps} @ @{thms all-com-interned-def
ctxt)
                THEN' SELECT-GOAL (safe-tac ctxt))); (* FIXME split the goal, simplify the sets away. FIXME
try to nuke safe-tac *)

fun vcg-sem-tac ctxt =
  SELECT-GOAL (HEADGOAL (match-tac ctxt @{thms CIMP-vcg.vcg.intros}
              THEN' (TRY o (ematch-tac ctxt @{thms system-responds-action-specE} THEN' assume-tac ctxt))
              THEN' Rule-Insts.thin-tac ctxt HST s = h [(@{binding s}, NONE, NoSyn), (@{binding h}, NONE,
NoSyn)] (* discard history: we don't use it here *)
               THEN' clarsimp-tac (ss-only @{thms vcg-sem-simps} ctxt)
```

```
        THEN-ALL-NEW asm-simp-tac ctxt)); (* remove unused meta−bound vars FIXME subgoaler in HOL's
usual simplifier setup, somehow lost by ss-only *)

(* FIXME gingerly settle location set membership and (dis−)equalities *)
fun loc-mem-tac ctxt =
  let
    val loc-defs = Proof-Context.get-fact ctxt (Facts.named loc-defs)
  in
   SELECT-GOAL (HEADGOAL ( (TRY o REPEAT-ALL-NEW (ematch-tac ctxt @{thms loc-mem-tac-elims}))
            THEN-ALL-NEW (TRY o hyp-subst-tac ctxt)
            THEN-ALL-NEW (TRY o REPEAT-ALL-NEW (match-tac ctxt @{thms loc-mem-tac-intros}))
          THEN-ALL-NEW ( SOLVED' (match-tac ctxt (Named-Theorems.get ctxt named-theorems ‹locset-cache›)
                        ORELSE' safe-simp-tac (HOL-ss-only (@{thms loc-mem-tac-simps} @ loc-defs) ctxt) ) )))
  end;

fun thin-pre-inv-tac ctxt =
    SELECT-GOAL (HEADGOAL ( (* FIXME trying to scope the REPEAT-DETERM ala [1] *)
          (REPEAT-DETERM o ematch-tac ctxt @{thms conjE})
      THEN' (REPEAT-DETERM o ( (ematch-tac ctxt @{thms thin-locs-pre-discardE}  THEN' assume-tac ctxt
THEN' loc-mem-tac ctxt)
                    ORELSE' (ematch-tac ctxt @{thms thin-locs-pre-keep-atE}  THEN' assume-tac ctxt)
                    ORELSE' (ematch-tac ctxt @{thms thin-locs-pre-keep-atSE} THEN' assume-tac ctxt THEN'
loc-mem-tac ctxt) ))));

(* FIXME redo keep-postE: if at loc is provable, discard the at antecedent, otherwise keep it *)
(* if the post inv is an LSTP then the present fix is to say (no-thin-post-inv) −− would be good to automate that
*)
fun thin-post-inv-tac keep-locs ctxt =
  let
    val keep-postE-thms = if keep-locs then @{thms thin-locs-post-keep-locsE} else @{thms thin-locs-post-keepE}
    fun nail-discard-prems-tac ctxt = assume-non-schematic-prem-tac ctxt THEN' loc-mem-tac ctxt THEN' (TRY
o match-tac ctxt @{thms process-name.simps})
  in
    SELECT-GOAL (HEADGOAL ( (* FIXME trying to scope the REPEAT-DETERM ala [1] *)
          resolve-tac ctxt @{thms schematic-prem}
      THEN' REPEAT-DETERM o CHANGED o (* FIXME CHANGED? also check what happens for non−invL
post invs −− aim to fail the ^^^ resolve-tac too *)
              ( (                                match-tac ctxt @{thms thin-locs-post-discard-conjE} THEN'
nail-discard-prems-tac ctxt)
          ORELSE' (eresolve-tac ctxt @{thms TrueE} THEN' match-tac ctxt @{thms thin-locs-post-discardE}
THEN' nail-discard-prems-tac ctxt)
        ORELSE' eresolve-tac ctxt keep-postE-thms )
  ))
  end;

fun vcg-inv-tac keep-locs no-thin-post-inv ctxt =
  let
    val invs = Named-Theorems.get ctxt named-theorems ‹inv›
  in
         SELECT-GOAL (Local-Defs.unfold-tac ctxt invs) (* FIXME trying to say unfold in [1] only *)
    THEN' thin-pre-inv-tac ctxt
    THEN' ( if no-thin-post-inv
            then SELECT-GOAL all-tac (* full-simp-tac (ss-only @{thms vcg-inv-simps} ctxt) (* FIXME maybe
not? *) *)
            else full-simp-tac (Splitter.add-split @{thm lcond-split-asm} (ss-only @{thms vcg-inv-simps} ctxt))
    THEN-ALL-NEW thin-post-inv-tac keep-locs ctxt )
  end;
```

```
(* For showing local invariants. FIXME tack on (no-thin-post-inv) for universal/LSTP ones *)
fun vcg-jackhammer-tac keep-locs no-thin-post-inv ctxt =
    SELECT-GOAL (HEADGOAL (vcg-fragments-tac ctxt)
    THEN PARALLEL-ALLGOALS (
                vcg-sem-tac ctxt
      THEN-ALL-NEW vcg-inv-tac keep-locs no-thin-post-inv ctxt
      THEN-ALL-NEW (if keep-locs then SELECT-GOAL all-tac else Rule-Insts.thin-tac ctxt AT - = - [])
      THEN-ALL-NEW TRY o clarsimp-tac ctxt (* limply try to solve the remaining goals *)
    ));

(* For showing noninterference *)
fun vcg-chainsaw-tac no-thin unfold-foreign-inv-thms ctxt =
  let
    fun specialize-foreign-invs-tac ctxt =
                (* FIXME split goal: makes sense because local procs control locs have changed? *)
                    REPEAT-ALL-NEW (match-tac ctxt @{thms conjI})
          THEN-ALL-NEW TRY o ( match-tac ctxt @{thms impI} (* FIXME could tweak rules vvvv *)
                        (* thin out the invariant we're showing non-interference for *)
(* FIXME look for reasons to retain the invariant, then do a big thin-tac at the end.
Intuitively we don't have enough info to settle atS v atS questions and it's too hard/not informative enough to try.
Let the user do it.
Maybe add an info thing that tells what was thinned.

FIXME location-sensitive predicates are not amenable to
simplification: this is the cost of using projections on
pred-state. Instead use elimination rules ‹nie›.


*)
                                THEN' ( REPEAT-DETERM o ( ( ematch-tac ctxt @{thms ni-thin-locs-discardE}
THEN' assume-tac ctxt THEN' assume-tac ctxt THEN' loc-mem-tac ctxt THEN' match-tac ctxt @{thms pro-
cess-name.simps} THEN' TRY o match-tac ctxt @{thms process-name.simps} )
                                        ORELSE' ( ematch-tac ctxt @{thms ni-thin-locs-keep-atE} THEN' assume-tac
ctxt THEN' assume-tac ctxt THEN' match-tac ctxt @{thms process-name.simps} THEN' TRY o match-tac ctxt
@{thms process-name.simps} )
                                    ORELSE' ( ematch-tac ctxt @{thms ni-thin-locs-keep-atSE} THEN' assume-tac ctxt
THEN' assume-tac ctxt THEN' loc-mem-tac ctxt THEN' match-tac ctxt @{thms process-name.simps} THEN'
TRY o match-tac ctxt @{thms process-name.simps} ) ) ) )
  in
    SELECT-GOAL (HEADGOAL (vcg-fragments-tac ctxt)
    THEN PARALLEL-ALLGOALS (
                vcg-sem-tac ctxt
                (* nail cheaply with an nie fact + ambient clarsimp *)
     THEN-ALL-NEW ( SOLVED' (ematch-tac ctxt (Named-Theorems.get ctxt @{named-theorems nie}) THEN-ALL-NE
clarsimp-tac ctxt)
           ORELSE' ( (* do it the hard way: specialise any process-specific invariants. Similar to vcg-jackhammer
but not the same *)
                    vcg-inv-tac false true ctxt
                    (* unfold the foreign inv *)
                THEN' SELECT-GOAL (Local-Defs.unfold-tac ctxt unfold-foreign-inv-thms) (* FIXME trying to
say [1] *)
              THEN' (REPEAT-DETERM o ematch-tac ctxt @{thms conjE})
              THEN' specialize-foreign-invs-tac ctxt
                    (* limply try to solve the remaining goals; FIXME turn s' into s as much as easily possible *)
        THEN-ALL-NEW (TRY o clarsimp-tac ctxt)
                    (* FIXME discard loc info. It is useful, this is a stopgap *)
        THEN-ALL-NEW (if no-thin then SELECT-GOAL all-tac
                     else (Rule-Insts.thin-tac ctxt AT - = - []
            THEN-ALL-NEW (REPEAT-DETERM o Rule-Insts.thin-tac ctxt at - - - ⟶ - [])
```

```
        THEN-ALL-NEW (REPEAT-DETERM o Rule-Insts.thin-tac ctxt atS - - - ⟶ - []) ))
   ))))
  end;

(*

Scrutinise the goal state for an 'AT' fact that tells us which label the process is at.

It seems this is not kosher:
 − for reasons unknown (Eisbach?) this tactic gets called with a bogus TERM - and then the real goal state.
 − this tactic (sometimes) does not work if used with THEN-ALL-NEW ';' −−
   chk-label does not manage to uniquify labels −− so be sure to
   combine with ','.
 − if two goals have the same 'at' location then we disambiguate, but
   perhaps not stably. Could imagine creating subcases, but
   'Method.goal-cases-tac' is not yet capable of that.
 − at communication steps we could get unlucky and choose the label from the other process.

The user can supply a list of process names to somewhat address these issues.

See Pure/Tools/rule-insts.ML for structurally similar tactics (dynamic instantiation).

Limitations:
 − only works with 'Const' labels
 − brittle: assumes things are very well−formed

*)
fun vcg-name-cases-tac (proc-names: term list) -(*facts*) (ctxt, st) =
  if Thm.nprems-of st = 0
  then Seq.empty (* no-tac *)
  else
    let
      fun fst-ord ord ((x, -), (x', -)) = ord (x, x')
      fun snd-ord ord ((-, y), (-, y')) = ord (y, y')

      (* FIXME this search is drecky *)
      fun find-AT (t: term) : (term * string) option =
            ( (* tracing (scruting: " ^ Syntax.string-of-term ctxt t) ; *)
              case t of Const (HOL.Trueprop, -) $ (Const (@{const-name Set.member}, -) $ Const (l, -) $ (Const
(@{const-name CIMP-lang.AT},  -) $ - $ p)) => ((* tracing HIT; *) SOME (p, Long-Name.base-name l))
                      | Const (HOL.Trueprop, -) $ (Const (@{const-name CIMP-lang.atS}, -) $ p $ Const (ls, -) $ -)
=> ((* tracing HIT; *) SOME (p, Long-Name.base-name ls))
                      | - => NONE )

      (* FIXME Isabelle makes it awkward to use polymorphic process names; paper over that crack here *)
      val rec terms-eq-ignoring-types =
          fn (Const (c0, -), Const (c1, -)) => fast-string-ord (c0, c1) = EQUAL
           | (Free (f0, -),  Free (f1, -))  => fast-string-ord (f0, f1) = EQUAL
           | (Var (v0, -) , Var (v1, -)) => prod-ord fast-string-ord int-ord (v0, v1) = EQUAL
           | (Bound i0, Bound i1) => i0 = i1
         | (Abs (b0, -, t0), Abs (b1, -, t1)) => fast-string-ord (b0, b1) = EQUAL andalso terms-eq-ignoring-types
(t0, t1)
           | (t0 $ u0, t1 $ u1) => terms-eq-ignoring-types (t0, t1) andalso terms-eq-ignoring-types (u0, u1)
           | - => false

      fun mk-label (default: string) (ats : (term * string) list) : string =
            case (ats, proc-names) of
              ((-, l)::-, []) => ((* tracing (No proc-names, Using label: " ^ l); *) l)
```

```
        | - =>
          let
              val ls = List.mapPartial (fn p => List.find (fn (p', -) => terms-eq-ignoring-types (p, p')) ats)
proc-names
          in
            case ls of
              [] => (warning (vcg-name-cases: using the default name: ^ default); default)
            | - => ls |> List.map snd |> String.concatWith -
          end

    fun labels-for-cases (i: int) (acc: (int * string) list) : (int * string) list =
          case i of
            0 => acc
          | i => Thm.cprem-of st i |> Thm.term-of |> Logic.strip-assums-hyp
                |> List.mapPartial find-AT |> mk-label (Int.toString i)
                |> (fn l => labels-for-cases (i - 1) ((i, l) :: acc))

      (* Make the labels unique if need be *)
    fun uniquify (i: int) (ls: (int * string) list) : (int * string) list =
      case ls of
        [] => []
      | [l] => [l]
      | l :: l' :: ls => (case fast-string-ord (snd l, snd l') of
                            EQUAL => (fst l, snd l ^ Int.toString i) :: uniquify (i + 1) (l' :: ls)
                          | -     => l :: uniquify 0 (l' :: ls))
    val labels = labels-for-cases (Thm.nprems-of st) []
    val labels = labels
              |> sort (snd-ord fast-string-ord) |> uniquify 0 |> sort (fst-ord int-ord)
              |> List.map (fn (-, l) => ((* tracing (label: ^ l); *) l))
  in
    Method.goal-cases-tac labels (ctxt, st)
  end;

end

val - =
  Theory.setup (Method.setup @{binding loc-mem}
    (Scan.succeed (SIMPLE-METHOD' o GC-VCG.loc-mem-tac))
    solve location membership goals)

val - =
  Theory.setup (Method.setup @{binding vcg-fragments}
    (Scan.succeed (SIMPLE-METHOD' o GC-VCG.vcg-fragments-tac))
    unfold com defs, execute vcg-fragments' and split goals)

val - =
  Theory.setup (Method.setup @{binding vcg-sem}
    (Scan.succeed (SIMPLE-METHOD' o GC-VCG.vcg-sem-tac))
    reduce VCG goal to semantics and clarsimp)

val - =
  Theory.setup (Method.setup @{binding vcg-inv}
     (Scan.lift (Args.mode keep-locs -- Args.mode no-thin-post-inv) >> (fn (keep-locs, no-thin-post-inv) =>
SIMPLE-METHOD' o GC-VCG.vcg-inv-tac keep-locs no-thin-post-inv))
    specialise the invariants to the goal. (keep-locs) retains locs in post conds)

val - =
  Theory.setup (Method.setup @{binding vcg-jackhammer}
```

(*Scan.lift* (*Args.mode keep-locs* −− *Args.mode no-thin-post-inv*) >> (*fn* (*keep-locs*, *no-thin-post-inv*) => *SIMPLE-METHOD′ o GC-VCG.vcg-jackhammer-tac keep-locs no-thin-post-inv*))
VCG tactic. (*keep-locs*) *retains locs in post conds.* (*no-thin-post-inv*) *does not attempt to specalise the post condition.*)

val - =
Theory.setup (*Method.setup* @{*binding vcg-chainsaw*}
(*Scan.lift* (*Args.mode no-thin*) −− *Attrib.thms* >> (*fn* (*no-thin*, *thms*) => *SIMPLE-METHOD′ o GC-VCG.vcg-chains no-thin thms*))
VCG non−interference tactic. Tell it how to unfold the foreign local invs.)

val - =
Theory.setup (*Method.setup* @{*binding vcg-name-cases*}
(*Scan.repeat Args.term* >> (*fn proc-names* => *fn - =>* CONTEXT-METHOD (*GC-VCG.vcg-name-cases-tac proc-names*)))
divine canonical case names for outstanding VCG goals)

›

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 (λ*p s. mem-store-buffers* (*s sys*) *p*)
tso-store-inv
by (*simp add*: *eq-imp-def tso-store-inv-def*)

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*
by (*auto simp*: *tso-store-inv-def dest*!: *spec*[**where** *x*=*m*])

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*
by (*auto simp*: *do-store-action-def split*: *mem-store-action.splits*)

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*
using *assms*
apply (*clarsimp simp*: *sys-load-def fold-stores-def*)

apply (*rule fold-invariant*[**where** *P*=λ*fr*. *do-load-action ract* (*fr* (*s sys*)) = *v* **and** *Q*=*mut-writes*])
apply (*fastforce simp*: *tso-store-inv-def*)
apply (*auto simp*: *do-load-action-def split*: *mem-store-action.splits*)
done

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*) (λ*obj*. *obj-fields obj f*)) (**is** *?lhs* = *mv-Ref ?rhs*)
using *assms* **unfolding** *sys-load-def fold-stores-def*
apply *clarsimp*
apply (*rule fold-invariant*[**where** *P*=λ*fr*. *Option.bind* (*heap* (*fr* (*s sys*)) *r*) (λ*obj*. *obj-fields obj f*) = *?rhs*
and *Q*=λ*w*. ∀ *r f r'*. *w* ≠ *mw-Mutate r f r'*])
apply (*fastforce simp*: *tso-store-inv-def*)
apply (*auto simp*: *do-store-action-def map-option-case fun-upd-apply*
split: *mem-store-action.splits option.splits*)
done

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*))
using *assms* **unfolding** *sys-load-def fold-stores-def*
apply *clarsimp*
apply (*rule fold-invariant*[**where** *P*=λ*fr*. *map-option obj-mark* (*heap* (*fr* (*s sys*)) *r*) = *map-option obj-mark* (*sys-heap s r*)
and *Q*=λ*w*. ∀ *fl*. *w* ≠ *mw-Mark r fl*])
apply (*auto simp*: *map-option-case do-store-action-def filter-empty-conv fun-upd-apply*
split: *mem-store-action.splits option.splits*)
done

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*)
using *assms*
apply (*clarsimp simp*: *sys-load-def fold-stores-def*)
apply (*rule fold-invariant*[**where** *P*=λ*fr*. *phase* (*fr* (*s sys*)) = *sys-phase s* **and** *Q*=λ*w*. ∀ *ph*. *w* ≠ *mw-Phase ph*])
apply (*auto simp*: *do-store-action-def filter-empty-conv*
split: *mem-store-action.splits*)
done

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*))
using *assms*
apply (*clarsimp simp*: *sys-load-def fold-stores-def*)
apply (*rule fold-invariant*[**where** *P*=λ*fr*. *fM* (*fr* (*s sys*)) = *sys-fM s* **and** *Q*=λ*w*. ∀ *fl*. *w* ≠ *mw-fM fl*])
apply (*auto simp*: *do-store-action-def filter-empty-conv*
split: *mem-store-action.splits*)
done

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* (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*)⦈))
= (*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* (λ*obj*. *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*
unfolding *mut-m.tso-store-refs-def* **by** (*auto simp*: *fun-upd-apply*)

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*)
unfolding *fold-stores-def*
apply (*rule spec*[*OF fold-invariant*[**where** *P*=λ*fr*. ∀ *obj*. *?P fr obj* **and** *Q*=λ*w*. *w* ∈ *set ws*]])
apply *fastforce*
apply (*fastforce simp*: *ran-def split*: *obj-at-splits*)
apply *clarsimp*
apply (*drule* (*1*) *bspec*)
apply (*clarsimp simp*: *fun-upd-apply split*: *mem-store-action.split-asm if-splits*)
done

lemma *points-to-Mutate*:
(*x points-to 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*) ⦈))
⟷ (*r* ≠ *x* ∧ (*x points-to y*) *s*) ∨ (*r* = *x* ∧ *valid-ref r s* ∧ (*opt-r'* = *Some y* ∨ ((*x points-to y*) *s* ∧ *obj-at* (λ*obj*. ∃ *f'*. *obj-fields obj f'* = *Some y* ∧ *f* ≠ *f'*) *r s*)))
unfolding *ran-def* **by** (*auto simp*: *fun-upd-apply split*: *obj-at-splits*)

7.2 FIXME mutator handshake facts

lemma — Sanity
hp' = *hs-step hp* ⟹ ∃ *in' ht*. (*in'*, *ht*, *hp'*, *hp*) ∈ *hp-step-rel*
by (*cases hp*) (*auto simp*: *hp-step-rel-def*)

lemma — Sanity
(*False*, *ht*, *hp'*, *hp*) ∈ *hp-step-rel* ⟹ *hp'* = *hp-step ht hp*
by (*cases ht*) (*auto simp*: *hp-step-rel-def*)

lemma (**in** *mut-m*) *handshake-phase-invD*:
assumes *handshake-phase-inv s*
shows (*sys-ghost-hs-in-sync m s*, *sys-hs-type s*, *sys-ghost-hs-phase s*, *mut-ghost-hs-phase s*) ∈ *hp-step-rel*

$\wedge$ (*sys-hs-pending m s* $\longrightarrow$ $\neg$*sys-ghost-hs-in-sync m s*)
using *assms* **unfolding** *handshake-phase-inv-def* **by** *simp*

lemma *handshake-in-syncD*:
⟦ *All* (*ghost-hs-in-sync* (*s sys*)); *handshake-phase-inv s* ⟧
$\Longrightarrow$ $\forall$ *m'*. *mut-m.mut-ghost-hs-phase m' s* = *sys-ghost-hs-phase s*
by *clarsimp* (*auto simp*: *hp-step-rel-def dest*!: *mut-m.handshake-phase-invD*)

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*. *sys-ghost-hs-in-sync m s*, *sys-ghost-hs-phase s*, *gc-phase s*, *sys-phase s*, *tso-pending-phase gc s*) $\in$ *phase-rel*
using *assms* **unfolding** *phase-rel-inv-def* **by** *simp*

lemma *mut-m-not-idle-no-fM-write*:
⟦ *ghost-hs-phase* (*s* (*mutator m*)) $\neq$ *hp-Idle*; *fM-rel-inv s*; *handshake-phase-inv s*; *tso-store-inv s*; *p* $\neq$ *sys* ⟧
$\Longrightarrow$ $\neg$*sys-mem-store-buffers p s* = *mw-fM fl* # *ws*
apply (*drule mut-m.handshake-phase-invD*[**where** *m*=*m*])
apply (*drule fM-rel-invD*)
apply (*clarsimp simp*: *hp-step-rel-def fM-rel-def filter-empty-conv p-not-sys*)
apply (*metis list.set-intros*(*1*) *tso-store-invD*(*4*))
done

lemma (**in** *mut-m*) *mut-ghost-handshake-phase-idle*:
⟦ *mut-ghost-hs-phase s* = *hp-Idle*; *handshake-phase-inv s*; *phase-rel-inv s* ⟧
$\Longrightarrow$ *sys-phase s* = *ph-Idle*
apply (*drule phase-rel-invD*)
apply (*drule handshake-phase-invD*)
apply (*auto simp*: *phase-rel-def hp-step-rel-def*)
done

lemma *mut-m-not-idle-no-fM-writeD*:
⟦ *sys-mem-store-buffers p s* = *mw-fM fl* # *ws*; *ghost-hs-phase* (*s* (*mutator m*)) $\neq$ *hp-Idle*; *fM-rel-inv s*; *handshake-phase-inv s*; *tso-store-inv s*; *p* $\neq$ *sys* ⟧
$\Longrightarrow$ *False*
apply (*drule mut-m.handshake-phase-invD*[**where** *m*=*m*])
apply (*drule fM-rel-invD*)
apply (*clarsimp simp*: *hp-step-rel-def fM-rel-def filter-empty-conv p-not-sys*)
apply (*metis list.set-intros*(*1*) *tso-store-invD*(*4*))
done

7.3 points to, reaches, reachable mut

lemma (**in** *mut-m*) *reachable-eq-imp*:
eq-imp (λ*r'*. *mut-roots* $\otimes$ *mut-ghost-honorary-root* $\otimes$ (λ*s*. $\bigcup$(*ran* ‘ *obj-fields* ‘ *set-option* (*sys-heap s r'*)))
$\otimes$ *tso-pending-mutate* (*mutator m*))
(*reachable r*)
unfolding *eq-imp-def reachable-def tso-store-refs-def*
apply *clarsimp*
apply (*rename-tac s s'*)
apply (*subgoal-tac* $\forall$ *r'*. ($\exists$ *w*$\in$*set* (*sys-mem-store-buffers* (*mutator m*) *s*). *r'* $\in$ *store-refs w*) $\longleftrightarrow$ ($\exists$ *w*$\in$*set* (*sys-mem-store*
(*mutator m*) *s'*). *r'* $\in$ *store-refs w*))
apply (*subgoal-tac* $\forall$ *x*. (*x reaches r*) *s* $\longleftrightarrow$ (*x reaches r*) *s'*)

apply (*clarsimp*; *fail*)
apply (*auto simp*: *reaches-fields*; *fail*)[*1*]
apply (*drule arg-cong*[**where** *f*=*set*])
apply (*clarsimp simp*: *set-eq-iff*)
apply (*rule iffI*)
apply *clarsimp*
apply (*rename-tac s s′ r′ w*)
apply (*drule-tac x*=*w* **in** *spec*)
apply (*rule-tac x*=*w* **in** *bexI*)
apply (*clarsimp*; *fail*)
apply (*clarsimp split*: *mem-store-action.splits*; *fail*)
apply *clarsimp*
apply (*rename-tac s s′ r′ w*)
apply (*drule-tac x*=*w* **in** *spec*)
apply (*rule-tac x*=*w* **in** *bexI*)
apply (*clarsimp*; *fail*)
apply (*clarsimp split*: *mem-store-action.splits*; *fail*)
done

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 ∈ *mut-m.mut-roots m s* ⟹ *mut-m.reachable m x s*
x ∈ *mut-m.tso-store-refs m s* ⟹ *mut-m.reachable m x s*
by (*auto simp*: *mut-m.reachable-def reaches-def*)

lemma *reachable-points-to*[*elim*]:
⟦ (*x points-to y*) *s*; *mut-m.reachable m x s* ⟧ ⟹ *mut-m.reachable m y s*
by (*auto simp*: *mut-m.reachable-def reaches-def elim*: *rtranclp.intros*(*2*))

lemma (**in** *mut-m*) *mut-reachableE*[*consumes 1*, *case-names mut-root tso-store-refs*]:
⟦ *reachable y s*;
⋀*x*. ⟦ (*x reaches y*) *s*; *x* ∈ *mut-roots s* ⟧ ⟹ *Q*;
⋀*x*. ⟦ (*x reaches y*) *s*; *x* ∈ *mut-ghost-honorary-root s* ⟧ ⟹ *Q*;
⋀*x*. ⟦ (*x reaches y*) *s*; *x* ∈ *tso-store-refs s* ⟧ ⟹ *Q* ⟧ ⟹ *Q*
by (*auto simp*: *reachable-def*)

lemma *reachable-induct*[*consumes 1*, *case-names root ghost-honorary-root tso-root reaches*]:
assumes *r*: *mut-m.reachable m y s*
assumes *root*: ⋀*x*. ⟦ *x* ∈ *mut-m.mut-roots m s* ⟧ ⟹ *P x*
assumes *ghost-honorary-root*: ⋀*x*. ⟦ *x* ∈ *mut-m.mut-ghost-honorary-root m s* ⟧ ⟹ *P x*
assumes *tso-root*: ⋀*x*. *x* ∈ *mut-m.tso-store-refs m s* ⟹ *P x*
assumes *reaches*: ⋀*x y*. ⟦ *mut-m.reachable m x s*; (*x points-to y*) *s*; *P x* ⟧ ⟹ *P y*
shows *P y*
using *r* **unfolding** *mut-m.reachable-def*
proof(*clarify*)
fix *x*
assume (*x reaches y*) *s* **and** *x* ∈ *mut-m.mut-roots m s* ∪ *mut-m.mut-ghost-honorary-root m s* ∪ *mut-m.tso-store-refs m s*
then show *P y*
unfolding *reaches-def* **proof** *induct*
case *base* **with** *root ghost-honorary-root tso-root* **show** *?case* **by** *blast*
next
case (*step y z*) **with** *reaches* **show** *?case*
unfolding *mut-m.reachable-def reaches-def* **by** *meson*
qed
qed

lemma *mutator-reachable-tso*:
sys-mem-store-buffers (*mutator m*) *s* = *mw-Mutate r f opt-r′ # ws*
⟹ *mut-m.reachable m r s* ∧ (∀ *r′*. *opt-r′* = *Some r′* ⟶ *mut-m.reachable m r′ s*)
sys-mem-store-buffers (*mutator m*) *s* = *mw-Mutate-Payload r f pl # ws*
⟹ *mut-m.reachable m r s*
by (*auto simp*: *mut-m.tso-store-refs-def*)

7.4 Colours

lemma *greyI*[*intro*]:
r ∈ *ghost-honorary-grey* (*s p*) ⟹ *grey r s*
r ∈ *W* (*s p*) ⟹ *grey r s*
r ∈ *WL p s* ⟹ *grey r s*
unfolding *grey-def WL-def* **by** (*case-tac* [!] *p*) *auto*

lemma *blackD*[*dest*]:
black r s ⟹ *marked r s*
black r s ⟹ *r* ∉ *WL p s*
unfolding *black-def grey-def* **by** *simp-all*

lemma *whiteI*[*intro*]:
obj-at (λ*obj*. *obj-mark obj* = (¬ *sys-fM s*)) *r s* ⟹ *white r s*
unfolding *white-def* **by** *simp*

lemma *marked-not-white*[*dest*]:
white r s ⟹ ¬*marked r s*
unfolding *white-def* **by** (*simp-all split*: *obj-at-splits*)

lemma *white-valid-ref*[*elim*!]:
white r s ⟹ *valid-ref r s*
unfolding *white-def* **by** (*simp-all split*: *obj-at-splits*)

lemma *not-white-marked*[*elim*!]:
⟦¬ *white r s*; *valid-ref r s*⟧ ⟹ *marked r s*
unfolding *white-def* **by** (*simp split*: *obj-at-splits*)

lemma *black-eq-imp*:
eq-imp (λ-::*unit*. (λ*s*. *r* ∈ (⋃ *p*. *WL p s*)) ⊗ *sys-fM* ⊗ (λ*s*. *map-option obj-mark* (*sys-heap s r*)))
(*black r*)
unfolding *eq-imp-def black-def grey-def* **by** (*auto split*: *obj-at-splits*)

lemma *grey-eq-imp*:
eq-imp (λ-::*unit*. (λ*s*. *r* ∈ (⋃ *p*. *WL p s*)))
(*grey r*)
unfolding *eq-imp-def grey-def* **by** *auto*

lemma *white-eq-imp*:
eq-imp (λ-::*unit*. *sys-fM* ⊗ (λ*s*. *map-option obj-mark* (*sys-heap s r*)))
(*white r*)
unfolding *eq-imp-def white-def* **by** (*auto split*: *obj-at-splits*)

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 (λ*r′*. (λ*s*. ⋃ *p*. *WL p s*) ⊗ *sys-fM* ⊗ (λ*s*. *map-option obj-mark* (*sys-heap s r′*)))

black-heap
apply (*clarsimp simp*: *eq-imp-def black-heap-def black-def grey-def all-conj-distrib fun-eq-iff split*: *option.splits*)
apply (*rename-tac s s'*)
apply (*subgoal-tac* $\forall x.$ *marked x s* $\longleftrightarrow$ *marked x s'*)
apply (*subgoal-tac* $\forall x.$ *valid-ref x s* $\longleftrightarrow$ *valid-ref x s'*)
apply (*subgoal-tac* $\forall x.$ $(\forall p.\ x \notin WL\ p\ s) \longleftrightarrow (\forall p.\ x \notin WL\ p\ s')$)
apply *clarsimp*
apply (*auto simp*: *set-eq-iff*)[*1*]
apply *clarsimp*
apply (*rename-tac x*)
apply (*rule eq-impD*[*OF obj-at-eq-imp*])
apply (*drule-tac x=x* **in** *spec*)
apply (*drule-tac f=map-option* ⟨*True*⟩ **in** *arg-cong*)
apply *fastforce*
apply *clarsimp*
apply (*rule eq-impD*[*OF obj-at-eq-imp*])
apply *clarsimp*
apply (*rename-tac x*)
apply (*drule-tac x=x* **in** *spec*)
apply (*drule-tac f=map-option* (λ*fl*. *fl* = *sys-fM s*) **in** *arg-cong*)
apply *simp*
done

lemma *white-heap-eq-imp*:
eq-imp (λ*r'*. *sys-fM* ⊗ (λ*s*. *map-option obj-mark* (*sys-heap s r'*)))
white-heap
apply (*clarsimp simp*: *all-conj-distrib eq-imp-def white-def white-heap-def obj-at-def fun-eq-iff*
split: *option.splits*)
apply (*rule iffI*)
apply (*metis* (*opaque-lifting*, *no-types*) *map-option-eq-Some*)+
done

lemma *no-black-refs-eq-imp*:
eq-imp (λ*r'*. (λ*s*. ($\bigcup p.$ *WL p s*)) ⊗ *sys-fM* ⊗ (λ*s*. *map-option obj-mark* (*sys-heap s r'*)))
no-black-refs
apply (*clarsimp simp add*: *eq-imp-def no-black-refs-def black-def grey-def all-conj-distrib fun-eq-iff set-eq-iff split*: *option.splits*)
apply (*rename-tac s s'*)
apply (*subgoal-tac* $\forall x.$ *marked x s* $\longleftrightarrow$ *marked x s'*)
apply *clarsimp*
apply (*clarsimp split*: *obj-at-splits*)
apply (*rename-tac x*)
apply (*drule-tac x=x* **in** *spec*)+
apply (*auto split*: *obj-at-splits*)
done

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 ⟹ *no-black-refs s*
apply (*clarsimp simp*: *white-def white-heap-def no-black-refs-def black-def*)
apply (*rename-tac x*)
apply (*drule-tac x=x* **in** *spec*)
apply (*clarsimp split*: *obj-at-splits*)
done

lemma *black-heap-no-greys*[*elim*]:
⟦ *no-grey-refs s*; ∀ *r*. *marked r s* ∨ ¬*valid-ref r s* ⟧ ⟹ *black-heap s*
unfolding *black-def black-heap-def no-grey-refs-def* **by** *fastforce*

lemma *heap-colours-colours*:
black-heap s ⟹ ¬*white r s*
white-heap s ⟹ ¬*black r s*
by (*auto simp*: *black-heap-def white-def white-heap-def*
dest!: *spec*[**where** *x*=*r*]
split: *obj-at-splits*)

The strong-tricolour invariant

lemma *strong-tricolour-invD*:
⟦ *black x s*; (*x points-to y*) *s*; *valid-ref y s*; *strong-tricolour-inv s* ⟧
⟹ *marked y s*
unfolding *strong-tricolour-inv-def* **by** *fastforce*

lemma *no-black-refsD*:
no-black-refs s ⟹ ¬*black r s*
unfolding *no-black-refs-def* **by** *simp*

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 ⋀*x*. *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 x y*
using *assms* **unfolding** *has-white-path-to-def* **by** (*rule rtranclp.induct*; *blast*)

lemma *has-white-path-toD*[*dest*]:
(*x has-white-path-to y*) *s* ⟹ *white y s* ∨ *x* = *y*
unfolding *has-white-path-to-def* **by** (*fastforce elim*: *rtranclp.cases*)

lemma *has-white-path-to-refl*[*iff*]:
(*x has-white-path-to x*) *s*
unfolding *has-white-path-to-def* **by** *simp*

lemma *has-white-path-to-step*[*intro*]:
⟦(*x has-white-path-to y*) *s*; (*y points-to z*) *s*; *white z s*⟧ ⟹ (*x has-white-path-to z*) *s*
⟦(*y has-white-path-to z*) *s*; (*x points-to y*) *s*; *white y s*⟧ ⟹ (*x has-white-path-to z*) *s*
unfolding *has-white-path-to-def*
apply (*simp add*: *rtranclp.rtrancl-into-rtrancl*)
apply (*simp add*: *converse-rtranclp-into-rtranclp*)
done

lemma *has-white-path-toE*[*elim*!]:
⟦ (*x points-to y*) *s*; *white y s* ⟧ ⟹ (*x has-white-path-to y*) *s*
unfolding *has-white-path-to-def* **by** (*auto elim*: *rtranclp.intros*(*2*))

lemma *has-white-path-to-reaches*[*elim*]:
(*x has-white-path-to y*) *s* ⟹ (*x reaches y*) *s*
unfolding *has-white-path-to-def reaches-def*
by (*induct rule*: *rtranclp.induct*) (*auto intro*: *rtranclp.intros*(*2*))

lemma *has-white-path-to-blacken*[*simp*]:
(*x has-white-path-to w*) (*s*(*gc* := *s gc*(| *W* := *gc-W s* − *rs* |))) ⟷ (*x has-white-path-to w*) *s*
unfolding *has-white-path-to-def* **by** (*simp add*: *fun-upd-apply*)

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*
using *assms*
proof *induct*
case (*step x y z*)
then have (*y points-to z*) *s*
by (*cases sys-heap s y*)
(*auto 10 10 simp*: *ran-def obj-at-def split*: *option.splits dest*!: *spec*[**where** *x*=*y*])
with *step* **show** *?case*
apply −
apply (*rule has-white-path-to-step*, *assumption*, *assumption*)
apply (*clarsimp simp*: *white-def split*: *obj-at-splits*)
apply (*metis map-option-eq-Some option.sel*)
done
qed *simp*

lemma *has-white-path-to-eq-imp*:
eq-imp (λ*r'*. *sys-fM* ⊗ (λ*s*. ⋃(*ran* ‘ *obj-fields* ‘ *set-option* (*sys-heap s r'*))) ⊗ (λ*s*. *map-option obj-mark* (*sys-heap s r'*)))
(*x has-white-path-to y*)
unfolding *eq-imp-def*
apply (*clarsimp simp*: *all-conj-distrib*)
apply (*rule iffI*)
apply (*erule has-white-path-to-eq-imp'*; *auto*)
apply (*erule has-white-path-to-eq-imp'*; *auto*)
done

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*)
by (*auto simp*: *grey-protects-white-def*)

lemma *grey-protects-whiteI*[*iff*]:
grey g s ⟹ (*g grey-protects-white g*) *s*
by (*simp add*: *grey-protects-white-def*)

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*
by (*auto simp*: *grey-protects-white-def*)

lemma *grey-protects-white-reaches*[*elim*]:
(*g grey-protects-white w*) *s* ⟹ (*g reaches w*) *s*
by (*auto simp*: *grey-protects-white-def*)

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*
using *assms* **unfolding** *grey-protects-white-def*

apply −
apply (*elim conjE*)
apply (*rotate-tac −1*)
apply (*induct rule*: *has-white-path-to-induct*)
apply *blast+*
done

7.5 *valid-W-inv*

lemma *valid-W-inv-sys-ghg-empty-iff*[*elim*!]:
valid-W-inv s ⟹ *sys-ghost-honorary-grey s* = {}
unfolding *valid-W-inv-def* **by** *simp*

lemma *WLI*[*intro*]:
r ∈ *W* (*s p*) ⟹ *r* ∈ *WL p s*
r ∈ *ghost-honorary-grey* (*s p*) ⟹ *r* ∈ *WL p s*
unfolding *WL-def* **by** *simp-all*

lemma *WL-eq-imp*:
eq-imp (λ(-::*unit*) *s*. (*ghost-honorary-grey* (*s p*), *W* (*s p*)))
(*WL p*)
unfolding *eq-imp-def WL-def* **by** *simp*

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*). (λ*s*. *W* (*s p*)) ⊗ (λ*s*. *ghost-honorary-grey* (*s p*)) ⊗ *sys-fM* ⊗ (λ*s*. *map-option obj-mark* (*sys-heap s r*)) ⊗ *sys-mem-lock* ⊗ *tso-pending-mark p*)
valid-W-inv
apply (*clarsimp simp*: *eq-imp-def valid-W-inv-def fun-eq-iff all-conj-distrib white-def*)
apply (*rename-tac s s'*)
apply (*subgoal-tac* ∀ *p*. *WL p s* = *WL p s'*)
apply (*subgoal-tac* ∀ *x*. *marked x s* ⟷ *marked x s'*)
apply (*subgoal-tac* ∀ *x*. *obj-at* (λ*obj*. *obj-mark obj* = (¬*sys-fM s'*)) *x s* ⟷ *obj-at* (λ*obj*. *obj-mark obj* = (¬*sys-fM s'*)) *x s'*)
apply (*subgoal-tac* ∀ *x xa xb*. *mw-Mark xa xb* ∈ *set* (*sys-mem-store-buffers x s*) ⟷ *mw-Mark xa xb* ∈ *set* (*sys-mem-store-buffers x s'*))
apply (*simp*; *fail*)
apply *clarsimp*
apply (*rename-tac x xa xb*)
apply (*drule-tac x*=*x* **in** *spec*, *drule arg-cong*[**where** *f*=*set*], *fastforce*)
apply (*clarsimp split*: *obj-at-splits*)
apply (*rename-tac x*)
apply ((*drule-tac x*=*x* **in** *spec*)+)[*1*]
apply (*case-tac sys-heap s x*, *simp-all*)
apply (*case-tac sys-heap s' x*, *auto*)[*1*]
apply (*clarsimp split*: *obj-at-splits*)
apply (*rename-tac x*)
apply (*drule-tac x*=*x* **in** *spec*)
apply (*case-tac sys-heap s x*, *simp-all*)
apply (*case-tac sys-heap s' x*, *simp-all*)
apply (*simp add*: *WL-def*)
done

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* ∈ *W* (*s p*); *valid-W-inv s* ⟧ ⟹ *marked r s*

⟦ *r* ∈ *ghost-honorary-grey* (*s p*); *sys-mem-lock s* ≠ *Some p*; *valid-W-inv s* ⟧ ⟹ *marked r s*
⟦ *r* ∈ *W* (*s p*); *valid-W-inv s* ⟧ ⟹ *valid-ref r s*
⟦ *r* ∈ *ghost-honorary-grey* (*s p*); *sys-mem-lock s* ≠ *Some p*; *valid-W-inv s* ⟧ ⟹ *valid-ref r s*
⟦ *mw-Mark r fl* ∈ *set* (*sys-mem-store-buffers p s*); *valid-W-inv s* ⟧ ⟹ *r* ∈ *ghost-honorary-grey* (*s p*)
unfolding *valid-W-inv-def*
apply (*simp-all add*: *split*: *obj-at-splits*)
apply *blast*+
done

lemma *valid-W-invD*:
⟦ *sys-mem-store-buffers p s* = *mw-Mark r fl* # *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* ∈ *set* (*sys-mem-store-buffers p s*); *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* (*sys-mem-store-buffers p s*) = [*mw-Mark r fl*]
unfolding *valid-W-inv-def white-def* **by** (*clarsimp dest*!: *spec*[**where** *x*=*p*], *blast*)+

lemma *valid-W-inv-colours*:
⟦*white x s*; *valid-W-inv s*⟧ ⟹ *x* ∉ *W* (*s p*)
using *marked-not-white valid-W-invE*(*1*) **by** *force*

lemma *valid-W-inv-no-mark-stores-invD*:
⟦ *sys-mem-lock s* ≠ *Some p*; *valid-W-inv s* ⟧
⟹ *tso-pending p is-mw-Mark s* = []
by (*auto dest*: *valid-W-invD*(*2*) *intro*!: *filter-False*)

lemma *valid-W-inv-sys-load*[*simp*]:
⟦ *sys-mem-lock s* ≠ *Some p*; *valid-W-inv s* ⟧
⟹ *sys-load p* (*mr-Mark r*) (*s sys*) = *mv-Mark* (*map-option obj-mark* (*sys-heap s r*))
unfolding *sys-load-def fold-stores-def*
apply *clarsimp*
apply (*rule fold-invariant*[**where** *P*=λ*fr*. *map-option obj-mark* (*heap* (*fr* (*s sys*)) *r*) = *map-option obj-mark* (*sys-heap s r*)
and *Q*=λ*w*. ∀ *r fl*. *w* ≠ *mw-Mark r fl*])
apply (*auto simp*: *map-option-case do-store-action-def filter-empty-conv fun-upd-apply*
dest: *valid-W-invD*(*2*)
split: *mem-store-action.splits option.splits*)
done

7.6 *grey-reachable*

lemma *grey-reachable-eq-imp*:
eq-imp (λ*r*′. (λ*s*. ⋃*p*. *WL p s*) ⊗ (λ*s*. *Set.bind* (*Option.set-option* (*sys-heap s r*′)) (*ran* ∘ *obj-fields*)))
(*grey-reachable r*)
by (*auto simp*: *eq-imp-def grey-reachable-def grey-def set-eq-iff reaches-fields*)

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 ⟹ *grey-reachable g s*
unfolding *grey-reachable-def reaches-def* **by** *blast*

lemma *grey-reachableE*:
⟦ (*g points-to y*) *s*; *grey-reachable g s* ⟧ ⟹ *grey-reachable y s*
unfolding *grey-reachable-def reaches-def* **by** (*auto elim*: *rtranclp.intros*(*2*))

7.7 valid refs inv

lemma *valid-refs-invI*:
⟦ ⋀*m x y*. ⟦ (*x reaches y*) *s*; *mut-m.root m x s* ∨ *grey x s* ⟧ ⟹ *valid-ref y s*
⟧ ⟹ *valid-refs-inv s*
by (*auto simp*: *valid-refs-inv-def mut-m.reachable-def grey-reachable-def*)

lemma *valid-refs-inv-eq-imp*:
eq-imp (λ(*m′*, *r′*). (λ*s*. *roots* (*s* (*mutator m′*))) ⊗ (λ*s*. *ghost-honorary-root* (*s* (*mutator m′*))) ⊗ (λ*s*. *map-option obj-fields* (*sys-heap s r′*)) ⊗ *tso-pending-mutate* (*mutator m′*) ⊗ (λ*s*. ⋃*p*. *WL p s*))
valid-refs-inv
apply (*clarsimp simp*: *eq-imp-def valid-refs-inv-def grey-reachable-def all-conj-distrib*)
apply (*rename-tac s s′*)
apply (*subgoal-tac* ∀ *r′*. *valid-ref r′ s* ⟷ *valid-ref r′ s′*)
apply (*subgoal-tac* ∀ *r′*. ⋃(*ran* ‘ *obj-fields* ‘ *set-option* (*sys-heap s r′*)) = ⋃(*ran* ‘ *obj-fields* ‘ *set-option* (*sys-heap s′ r′*)))
apply (*subst eq-impD*[*OF mut-m.reachable-eq-imp*])
defer
apply (*subst eq-impD*[*OF grey-eq-imp*])
defer
apply (*subst eq-impD*[*OF reaches-eq-imp*])
defer
apply *force*
apply (*metis option.set-map*)
apply (*clarsimp split*: *obj-at-splits*)
apply (*metis* (*no-types*, *opaque-lifting*) *None-eq-map-option-iff option.exhaust*)
apply *clarsimp*
apply *clarsimp*
apply *clarsimp*
done

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* ∈ *mut-m.mut-roots m s*; (*x reaches y*) *s*; *valid-refs-inv s* ⟧ ⟹ *valid-ref y s*
⟦ *x* ∈ *mut-m.mut-roots m s*; (*x reaches y*) *s*; *valid-refs-inv s* ⟧ ⟹ ∃ *obj*. *sys-heap s y* = *Some obj*
⟦ *x* ∈ *mut-m.tso-store-refs m s*; (*x reaches y*) *s*; *valid-refs-inv s* ⟧ ⟹ *valid-ref y s*
⟦ *x* ∈ *mut-m.tso-store-refs m s*; (*x reaches y*) *s*; *valid-refs-inv s* ⟧ ⟹ ∃ *obj*. *sys-heap s y* = *Some obj*
⟦ *w* ∈ *set* (*sys-mem-store-buffers* (*mutator m*) *s*); *x* ∈ *store-refs w*; (*x reaches y*) *s*; *valid-refs-inv s* ⟧ ⟹ *valid-ref y s*
⟦ *w* ∈ *set* (*sys-mem-store-buffers* (*mutator m*) *s*); *x* ∈ *store-refs w*; (*x reaches y*) *s*; *valid-refs-inv s* ⟧ ⟹ ∃ *obj*. *sys-heap s y* = *Some obj*
⟦ *grey x s*; (*x reaches y*) *s*; *valid-refs-inv s* ⟧ ⟹ *valid-ref y s*
⟦ *mut-m.reachable m x s*; *valid-refs-inv s* ⟧ ⟹ *valid-ref x s*
⟦ *mut-m.reachable m x s*; *valid-refs-inv s* ⟧ ⟹ ∃ *obj*. *sys-heap s x* = *Some obj*
⟦ *x* ∈ *mut-m.mut-ghost-honorary-root m s*; (*x reaches y*) *s*; *valid-refs-inv s* ⟧ ⟹ *valid-ref y s*
⟦ *x* ∈ *mut-m.mut-ghost-honorary-root m s*; (*x reaches y*) *s*; *valid-refs-inv s* ⟧ ⟹ ∃ *obj*. *sys-heap s y* = *Some obj*
apply (*simp-all add*: *valid-refs-inv-def grey-reachable-def mut-m.reachable-def mut-m.tso-store-refs-def*
split: *obj-at-splits*)
apply *blast*+
done

reachable snapshot inv

context *mut-m*
begin

lemma *reachable-snapshot-invI*[*intro*]:
(⋀*y*. *reachable y s* ⟹ *in-snapshot y s*) ⟹ *reachable-snapshot-inv s*

by (*simp add*: *reachable-snapshot-inv-def*)

lemma *reachable-snapshot-inv-eq-imp*:
eq-imp (λ*r'*. *mut-roots* ⊗ *mut-ghost-honorary-root* ⊗ (λ*s*. *r'* ∈ (⋃ *p*. *WL p s*)) ⊗ *sys-fM*
⊗ (λ*s*. ⋃(*ran* ‘ *obj-fields* ‘ *set-option* (*sys-heap s r'*))) ⊗ (λ*s*. *map-option obj-mark* (*sys-heap s r'*))
⊗ *tso-pending-mutate* (*mutator m*))
reachable-snapshot-inv
unfolding *eq-imp-def mut-m.reachable-snapshot-inv-def in-snapshot-def grey-protects-white-def black-def grey-def*
apply (*clarsimp simp*: *all-conj-distrib*)
apply (*rename-tac s s'*)
apply (*subst* (*1*) *eq-impD*[*OF has-white-path-to-eq-imp*])
apply *force*
apply (*subst eq-impD*[*OF reachable-eq-imp*])
apply *force*
apply (*subgoal-tac* ∀ *x*. *obj-at* (λ*obj*. *obj-mark obj* = *sys-fM s'*) *x s* ⟷ *obj-at* (λ*obj*. *obj-mark obj* = *sys-fM s'*) *x s'*)
apply *force*
apply (*clarsimp split*: *obj-at-splits*)
apply (*rename-tac x*)
apply (*drule-tac x*=*x* **in** *spec*)+
apply (*case-tac sys-heap s x*, *simp-all*)
apply (*case-tac sys-heap s' x*, *simp-all*)
done

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 ⟹ *in-snapshot r s*
grey r s ⟹ *in-snapshot r s*
⟦ *white w s*; (*g grey-protects-white w*) *s* ⟧ ⟹ *in-snapshot w s*
by (*auto simp*: *in-snapshot-def*)

lemma — Sanity
in-snapshot r s ⟹ *black r s* ∨ *grey r s* ∨ *white r s*
by (*auto simp*: *in-snapshot-def*)

lemma *in-snapshot-valid-ref*:
⟦ *in-snapshot r s*; *valid-refs-inv s* ⟧ ⟹ *valid-ref r s*
by (*metis blackD*(*1*) *grey-protects-whiteD grey-protects-white-reaches in-snapshot-def obj-at-cong obj-at-def option.case*(*2*) *valid-refs-invD*(*7*))

lemma *reachableI2*[*intro*]:
x ∈ *mut-m.mut-ghost-honorary-root m s* ⟹ *mut-m.reachable m x s*
unfolding *mut-m.reachable-def reaches-def* **by** *blast*

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'*);
⋀*r f r'*. *P r f r'* ⟷ *Q r f r'* ⟧
⟹ (∀ *r f r'*. *mw-Mutate r f r'* ∈ *set* (*sys-mem-store-buffers p s*) ⟶ *P r f r'*)
⟷ (∀ *r f r'*. *mw-Mutate r f r'* ∈ *set* (*sys-mem-store-buffers p s'*) ⟶ *Q r f r'*)
by (*intro iff-allI*) (*auto dest*!: *arg-cong*[**where** *f*=*set*])

lemma (**in** *mut-m*) *marked-insertions-eq-imp*:
eq-imp (λ*r'*. *sys-fM* ⊗ (λ*s*. *map-option obj-mark* (*sys-heap s r'*)) ⊗ *tso-pending-mw-mutate* (*mutator m*))
marked-insertions

unfolding *eq-imp-def marked-insertions-def obj-at-def*
apply (*clarsimp split*: *mem-store-action.splits*)
apply (*erule tso-pending-mw-mutate-cong*)
apply (*clarsimp split*: *option.splits obj-at-splits*)
apply (*rename-tac s s′ opt x*)
apply (*drule-tac x=x* **in** *spec*)
apply *auto*
done

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* ⟧
⟹ *marked r′ s*
by (*auto simp*: *mut-m.marked-insertions-def*)

lemma *marked-insertions-store-buffer-empty*[*intro*]:
tso-pending-mutate (*mutator m*) *s* = [] ⟹ *mut-m.marked-insertions m s*
unfolding *mut-m.marked-insertions-def* **by** (*auto simp*: *filter-empty-conv split*: *mem-store-action.splits*)

lemma (**in** *mut-m*) *marked-deletions-eq-imp*:
eq-imp (λ*r′*. *sys-fM* ⊗ (λ*s*. *map-option obj-fields* (*sys-heap s r′*)) ⊗ (λ*s*. *map-option obj-mark* (*sys-heap s r′*))
⊗ *tso-pending-mw-mutate* (*mutator m*))
marked-deletions
unfolding *eq-imp-def marked-deletions-def obj-at-field-on-heap-def ran-def*
apply (*clarsimp simp*: *all-conj-distrib*)
apply (*drule arg-cong*[**where** *f*=*set*])
apply (*subgoal-tac* ∀ *x*. *marked x s* ⟷ *marked x s′*)
apply (*clarsimp cong*: *option.case-cong*)
apply (*rule iffI*; *clarsimp simp*: *set-eq-iff split*: *option.splits mem-store-action.splits*; *blast*)
apply *clarsimp*
apply (*rename-tac s s′ x*)
apply (*drule-tac x=x* **in** *spec*)+
apply (*force split*: *obj-at-splits*)
done

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* = [] ⟹ *mut-m.marked-deletions m s*
unfolding *mut-m.marked-deletions-def* **by** (*auto simp*: *filter-empty-conv split*: *mem-store-action.splits*)

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*)|))) ⟷ *obj-at P r s* ∧ *r* ≠ *r′*
by (*clarsimp simp*: *fun-upd-apply split*: *obj-at-splits*)

lemma *obj-at-alloc*[*simp*]:
sys-heap s r′ = *None*
⟹ *obj-at P r* (*s*(*m* := *mut-m-s′*, *sys* := (*s sys*)(| *heap* := (*sys-heap s*)(*r′* ↦ *obj*) |)))
⟷ (*obj-at P r s* ∨ (*r* = *r′* ∧ *P obj*))
unfolding *ran-def* **by** (*simp add*: *fun-upd-apply split*: *obj-at-splits*)

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*)|))) ⟷ *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*)|))) ⟷ *valid-null-ref r' s*
valid-null-ref r' (*s*(*mutator m* := *mut-s'*, *sys* := (*s sys*)(| *heap* := (*heap* (*s sys*))(*r''* ↦ *obj*) |))) ⟷ *valid-null-ref r' s* ∨ *r'* = *Some r''*
unfolding *do-store-action-def valid-null-ref-def*
by (*auto simp*: *fun-upd-apply*
split: *mem-store-action.splits obj-at-splits option.splits*)

context *mut-m*
begin

lemma *reachable-load*[*simp*]:
assumes *sys-load* (*mutator m*) (*mr-Ref r f*) (*s sys*) = *mv-Ref r'*
assumes *r* ∈ *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(*cases m'* = *m*)
case *True* **show** *?thesis*
proof(*rule iffI*)
assume *?lhs* **with** *assms True* **show** *?rhs*
unfolding *sys-load-def*
apply *clarsimp*
apply (*clarsimp simp*: *reachable-def reaches-def tso-store-refs-def sys-load-def fold-stores-def fun-upd-apply*)
apply (*elim disjE*)
apply *blast*
defer
apply *blast*
apply *blast*

apply (*fold fold-stores-def*)
apply *clarsimp*
apply (*drule* (*1*) *fold-stores-points-to*)
apply (*erule disjE*)
apply (*fastforce elim*!: *converse-rtranclp-into-rtranclp*[*rotated*] *split*: *obj-at-splits intro*!: *ranI*)
apply (*clarsimp split*: *mem-store-action.splits*)
apply *meson*
done
next
assume *?rhs* **with** *True* **show** *?lhs* **unfolding** *mut-m.reachable-def* **by** (*fastforce simp*: *fun-upd-apply*)
qed
qed (*simp add*: *fun-upd-apply*)

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* }
unfolding *WL-def* **by** (*auto simp*: *fun-upd-apply*)

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'* |)))

$= (case\ p\ of\ gc \Rightarrow WL\ gc\ s \mid mutator\ m' \Rightarrow (if\ m' = m\ then\ \{\}\ else\ WL\ (mutator\ m')\ s) \mid sys \Rightarrow WL\ sys\ s \cup WL\ (mutator\ m)\ s)$
$ghost\text{-}honorary\text{-}grey\ (s\ (mutator\ m)) = \{\}$
$\Longrightarrow WL\ p\ (s(mutator\ m := s\ (mutator\ m)(\!|\ W := \{\}\ |\!),$
$sys := s\ sys(\!|\ hs\text{-}pending := hsp',\ W := sys\text{-}W\ s \cup W\ (s\ (mutator\ m)),$
$ghost\text{-}hs\text{-}in\text{-}sync := in'\ |\!)))$
$= (case\ p\ of\ gc \Rightarrow WL\ gc\ s \mid mutator\ m' \Rightarrow (if\ m' = m\ then\ \{\}\ else\ WL\ (mutator\ m')\ s) \mid sys \Rightarrow WL\ sys\ s \cup WL\ (mutator\ m)\ s)$
unfolding *WL-def* **by** (*auto simp*: *fun-upd-apply split*: *process-name.splits*)

lemma *colours-load-W*[*iff*]:
$gc\text{-}W\ s = \{\} \Longrightarrow black\ r\ (s(gc := (s\ gc)(\!|W := W\ (s\ sys)|\!),\ sys := (s\ sys)(\!|W := \{\}|\!))) \longleftrightarrow black\ r\ s$
$gc\text{-}W\ s = \{\} \Longrightarrow grey\ r\ (s(gc := (s\ gc)(\!|W := W\ (s\ sys)|\!),\ sys := (s\ sys)(\!|W := \{\}|\!))) \longleftrightarrow grey\ r\ s$
unfolding *black-def grey-def WL-def*
apply (*simp-all add*: *fun-upd-apply*)
apply *safe*
apply (*case-tac* [!] *x*)
apply *blast*+
done

lemma *WL-load-W*[*simp*]:
$gc\text{-}W\ s = \{\}$
$\Longrightarrow (WL\ p\ (s(gc := (s\ gc)(\!|W := sys\text{-}W\ s|\!),\ sys := (s\ sys)(\!|W := \{\}|\!))))$
$= (case\ p\ of\ gc \Rightarrow WL\ gc\ s \cup sys\text{-}W\ s \mid mutator\ m \Rightarrow WL\ (mutator\ m)\ s \mid sys \Rightarrow sys\text{-}ghost\text{-}honorary\text{-}grey\ s)$
unfolding *WL-def* **by** (*auto simp*: *fun-upd-apply split*: *process-name.splits*)

no grey refs

lemma *no-grey-refs-eq-imp*:
$eq\text{-}imp\ (\lambda(\text{-}::unit).\ (\lambda s.\ \bigcup p.\ WL\ p\ s))$
$no\text{-}grey\text{-}refs$
by (*auto simp add*: *eq-imp-def grey-def no-grey-refs-def set-eq-iff*)

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*:
$[\![\ no\text{-}grey\text{-}refs\ s;\ valid\text{-}W\text{-}inv\ s\ ]\!] \Longrightarrow tso\text{-}no\text{-}pending\text{-}marks\ s$
unfolding *no-grey-refs-def* **by** (*auto intro*!: *filter-False dest*: *valid-W-invD*(*2*))

lemma *no-grey-refs-not-grey-reachableD*:
$no\text{-}grey\text{-}refs\ s \Longrightarrow \neg grey\text{-}reachable\ x\ s$
by (*clarsimp simp*: *no-grey-refs-def grey-reachable-def*)

lemma *no-grey-refsD*:
$no\text{-}grey\text{-}refs\ s \Longrightarrow r \notin W\ (s\ p)$
$no\text{-}grey\text{-}refs\ s \Longrightarrow r \notin WL\ p\ s$
$no\text{-}grey\text{-}refs\ s \Longrightarrow r \notin ghost\text{-}honorary\text{-}grey\ (s\ p)$
by (*auto simp*: *no-grey-refs-def*)

lemma *no-grey-refs-marked*[*dest*]:
$[\![\ marked\ r\ s;\ no\text{-}grey\text{-}refs\ s\ ]\!] \Longrightarrow black\ r\ s$
by (*auto simp*: *no-grey-refs-def black-def*)

lemma *no-grey-refs-bwD*[*dest*]:
$[\![\ heap\ (s\ sys)\ r = Some\ obj;\ no\text{-}grey\text{-}refs\ s\ ]\!] \Longrightarrow black\ r\ s \vee white\ r\ s$
by (*clarsimp simp*: *black-def grey-def no-grey-refs-def white-def split*: *obj-at-splits*)

context *mut-m*
begin

lemma *reachable-blackD*:
⟦ *no-grey-refs s*; *reachable-snapshot-inv s*; *reachable r s* ⟧ ⟹ *black r s*
by (*simp add*: *no-grey-refs-def reachable-snapshot-inv-def in-snapshot-def grey-protects-white-def*)

lemma *no-grey-refs-not-reachable*:
⟦ *no-grey-refs s*; *reachable-snapshot-inv s*; *white r s* ⟧ ⟹ ¬*reachable r s*
by (*fastforce simp*: *no-grey-refs-def reachable-snapshot-inv-def in-snapshot-def grey-protects-white-def*
split: *obj-at-splits*)

lemma *no-grey-refs-not-rootD*:
⟦ *no-grey-refs s*; *reachable-snapshot-inv s*; *white r s* ⟧
⟹ *r* ∉ *mut-roots s* ∧ *r* ∉ *mut-ghost-honorary-root s* ∧ *r* ∉ *tso-store-refs s*
apply (*drule* (*2*) *no-grey-refs-not-reachable*)
apply (*force simp*: *reachable-def reaches-def*)
done

lemma *reachable-snapshot-inv-white-root*:
⟦ *white w s*; *w* ∈ *mut-roots s* ∨ *w* ∈ *mut-ghost-honorary-root s*; *reachable-snapshot-inv s* ⟧ ⟹ ∃ *g*. (*g grey-protects-white w*) *s*
unfolding *reachable-snapshot-inv-def in-snapshot-def reachable-def grey-protects-white-def reaches-def* **by** *auto*

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*) |)))
⟷ (*black b s* ∧ *b* ≠ *r*) ∨ (*b* = *r* ∧ *fl* = *sys-fM s* ∧ *valid-ref r s* ∧ ¬*grey r s*)
unfolding *black-def* **by** (*auto simp*: *fun-upd-apply split*: *obj-at-splits*)

lemma *colours-sweep-loop-free*[*iff*]:
black r (*s*(*sys* := *s sys*(|*heap* := (*heap* (*s sys*))(*r'* := *None*)|))) ⟷ (*black r s* ∧ *r* ≠ *r'*)
grey r (*s*(*sys* := *s sys*(|*heap* := (*heap* (*s sys*))(*r'* := *None*)|))) ⟷ (*grey r s*)
white r (*s*(*sys* := *s sys*(|*heap* := (*heap* (*s sys*))(*r'* := *None*)|))) ⟷ (*white r s* ∧ *r* ≠ *r'*)
unfolding *black-def grey-def white-def* **by** (*auto simp*: *fun-upd-apply split*: *obj-at-splits*)

lemma *colours-get-work-done*[*simp*]:
black r (*s*(*mutator m* := (*s* (*mutator m*))(| *W* := {} |),
sys := (*s sys*)(| *hs-pending* := *hp'*, *W* := *W* (*s sys*) ∪ *W* (*s* (*mutator m*)),
ghost-hs-in-sync := *his'* |))) ⟷ *black r s*
grey r (*s*(*mutator m* := (*s* (*mutator m*))(| *W* := {} |),
sys := (*s sys*)(| *hs-pending* := *hp'*, *W* := *W* (*s sys*) ∪ *W* (*s* (*mutator m*)),
ghost-hs-in-sync := *his'* |))) ⟷ *grey r s*
white r (*s*(*mutator m* := (*s* (*mutator m*))(| *W* := {} |),
sys := (*s sys*)(| *hs-pending* := *hp'*, *W* := *W* (*s sys*) ∪ *W* (*s* (*mutator m*)),
ghost-hs-in-sync := *his'* |))) ⟷ *white r s*
unfolding *black-def grey-def WL-def*
apply (*simp-all add*: *fun-upd-apply split*: *obj-at-splits*)
apply *blast*
apply (*metis process-name.distinct*(*3*))
done

lemma *colours-get-roots-done*[*simp*]:
black r (*s*(*mutator m* := (*s* (*mutator m*))(| *W* := {}, *ghost-hs-phase* := *hs'* |),
sys := (*s sys*)(| *hs-pending* := *hp'*, *W* := *W* (*s sys*) ∪ *W* (*s* (*mutator m*)),
ghost-hs-in-sync := *his'* |))) ⟷ *black r s*

grey r (*s*(*mutator m* := (*s* (*mutator m*))(| *W* := {}, *ghost-hs-phase* := *hs'* |),
sys := (*s sys*)(| *hs-pending* := *hp'*, *W* := *W* (*s sys*) ∪ *W* (*s* (*mutator m*)),
ghost-hs-in-sync := *his'* |))) ⟷ *grey r s*
white r (*s*(*mutator m* := (*s* (*mutator m*))(| *W* := {}, *ghost-hs-phase* := *hs'* |),
sys := (*s sys*)(| *hs-pending* := *hp'*, *W* := *W* (*s sys*) ∪ *W* (*s* (*mutator m*)),
ghost-hs-in-sync := *his'* |))) ⟷ *white r s*
unfolding *black-def grey-def WL-def*
apply (*simp-all add*: *fun-upd-apply split*: *obj-at-splits*)
apply *blast*
apply (*metis process-name.distinct*(*3*))
done

lemma *colours-flip-fM*[*simp*]:
fl ≠ *sys-fM s* ⟹ *black b* (*s*(*sys* := (*s sys*)(|*fM* := *fl*, *mem-store-buffers* := (*mem-store-buffers* (*s sys*))(*p* := *ws*)|))) ⟷ *white b s* ∧ ¬*grey b s*
unfolding *black-def white-def* **by** (*simp add*: *fun-upd-apply*)

lemma *colours-alloc*[*simp*]:
heap (*s sys*) *r'* = *None*
⟹ *black r* (*s*(*mutator m* := (*s* (*mutator m*))(| *roots* := *roots'* |), *sys* := (*s sys*)(|*heap* := (*heap* (*s sys*))(*r'* ↦ (|*obj-mark* = *fl*, *obj-fields* = *Map.empty*, *obj-payload* = *Map.empty*|))|)))
⟷ *black r s* ∨ (*r'* = *r* ∧ *fl* = *sys-fM s* ∧ ¬*grey r' s*)
grey r (*s*(*mutator m* := (*s* (*mutator m*))(| *roots* := *roots'* |), *sys* := (*s sys*)(|*heap* := (*heap* (*s sys*))(*r'* ↦ (|*obj-mark* = *fl*, *obj-fields* = *Map.empty*, *obj-payload* = *Map.empty*|))|)))
⟷ *grey r s*
heap (*s sys*) *r'* = *None*
⟹ *white r* (*s*(*mutator m* := (*s* (*mutator m*))(| *roots* := *roots'* |), *sys* := (*s sys*)(|*heap* := (*heap* (*s sys*))(*r'* ↦ (|*obj-mark* = *fl*, *obj-fields* = *Map.empty*, *obj-payload* = *Map.empty*|))|)))
⟷ *white r s* ∨ (*r'* = *r* ∧ *fl* ≠ *sys-fM s*)
unfolding *black-def white-def* **by** (*auto simp*: *fun-upd-apply split*: *obj-at-splits*)

lemma *heap-colours-alloc*[*simp*]:
⟦ *heap* (*s sys*) *r'* = *None*; *valid-refs-inv s* ⟧
⟹ *black-heap* (*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*|))|)))
⟷ *black-heap s* ∧ *fl* = *sys-fM s*
heap (*s sys*) *r'* = *None*
⟹ *white-heap* (*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*|))|)))
⟷ *white-heap s* ∧ *fl* ≠ *sys-fM s*
unfolding *black-heap-def white-def white-heap-def*
apply (*simp-all add*: *fun-upd-apply split*: *obj-at-splits*)
apply (*rule iffI*)
apply (*intro allI conjI impI*)
apply (*rename-tac x*)
apply (*drule-tac x*=*x* **in** *spec*)
apply *clarsimp*
apply (*drule spec*[**where** *x*=*r'*], *auto simp*: *reaches-def dest*!: *valid-refs-invD split*: *obj-at-splits*)[*2*]
apply (*rule iffI*)
apply (*intro allI conjI impI*)
apply (*rename-tac x obj*)
apply (*drule-tac x*=*x* **in** *spec*)
apply *clarsimp*
apply (*drule spec*[**where** *x*=*r'*], *auto dest*!: *valid-refs-invD split*: *obj-at-splits*)[*2*]
done

lemma *grey-protects-white-hs-done*[*simp*]:
(*g grey-protects-white w*) (*s*(*mutator m* := *s* (*mutator m*)(| *W* := {}, *ghost-hs-phase* := *hs'* |),

sys := *s sys*(| *hs-pending* := *hp'*, *W* := *sys-W s* ∪ *W* (*s* (*mutator m*)),
ghost-hs-in-sync := *his'* |))
⟷ (*g grey-protects-white w*) *s*
unfolding *grey-protects-white-def* **by** (*simp add*: *fun-upd-apply*)

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*
unfolding *grey-protects-white-def has-white-path-to-def* **by** *simp*

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
fix *y* :: '*ref*
assume *rx*: *reachable y ?s'*
then have *black y s* ∧ *y* ≠ *r*
proof(*induct rule*: *reachable-induct*)
case (*root x*) **with** *ngs nmr rsi* **show** *?case*
by (*auto simp*: *fun-upd-apply dest*: *reachable-blackD*)
next
case (*ghost-honorary-root x*) **with** *ngs nmr rsi* **show** *?case*
unfolding *reachable-def reaches-def* **by** (*auto simp*: *fun-upd-apply dest*: *reachable-blackD*)
next
case (*tso-root x*) **with** *ngs nmr rsi* **show** *?case*
unfolding *reachable-def reaches-def* **by** (*auto simp*: *fun-upd-apply dest*: *reachable-blackD*)
next
case (*reaches x y*) **with** *ngs nmr rsi* **show** *?case*
unfolding *reachable-def reaches-def*
apply (*clarsimp simp*: *fun-upd-apply*)
apply (*drule predicate2D*[*OF rtranclp-mono*[**where** *s*=λ*x y*. (*x points-to y*) *s*, *OF predicate2I*], *rotated*])
apply (*clarsimp split*: *obj-at-splits if-splits*)
apply (*rule conjI*)
apply (*rule reachable-blackD*, *assumption*, *assumption*)
apply (*simp add*: *reachable-def reaches-def*)
apply (*blast intro*: *rtranclp.intros*(*2*))
apply *clarsimp*
apply (*frule* (*1*) *reachable-blackD*[**where** *r*=*r*])
apply (*simp add*: *reachable-def reaches-def*)
apply (*blast intro*: *rtranclp.intros*(*2*))
apply *auto*
done
qed
then show *in-snapshot y ?s'*
unfolding *in-snapshot-def* **by** *simp*
qed

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(*rule iffI*)
assume *?lhs* **from** *this assms* **show** *?rhs*
proof(*induct rule*: *reachable-induct*)
case (*reaches x y*) **then show** *?case* **by** *clarsimp* (*fastforce simp*: *mut-m.reachable-def reaches-def elim*: *rtranclp.intros*(*2*) *split*: *obj-at-splits*)
qed (*auto simp*: *fun-upd-apply split*: *if-splits*)
next
assume *?rhs* **then show** *?lhs*
proof(*rule disjE*)
assume *mut-m.reachable m r' s* **then show** *?thesis*
proof(*induct rule*: *reachable-induct*)
case (*tso-root x*) **then show** *?case*
unfolding *mut-m.reachable-def* **by** *fastforce*
next
case (*reaches x y*) **with** *rn* **show** *?case*
unfolding *mut-m.reachable-def* **by** *fastforce*
qed (*auto simp*: *fun-upd-apply*)
next
assume $m' = m \wedge r' = r$ **with** *rn* **show** *?thesis*
unfolding *mut-m.reachable-def* **by** (*fastforce simp*: *fun-upd-apply*)
qed
qed

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'*)
using *assms* **unfolding** *reachable-snapshot-inv-def in-snapshot-def*
by (*auto simp del*: *reachable-fun-upd*)

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'*|)))
unfolding *reachable-snapshot-inv-def reachable-def in-snapshot-def grey-protects-white-def* **by** (*auto simp*: *fun-upd-apply*

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'* |)))
unfolding *reachable-snapshot-inv-def in-snapshot-def grey-protects-white-def* **by** (*simp add*: *fun-upd-apply*)

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'*]) |)))
unfolding *reachable-snapshot-inv-def in-snapshot-def grey-protects-white-def reachable-def*
apply *clarsimp*
apply (*drule-tac x*=*x* **in** *spec*)
apply (*auto simp*: *fun-upd-apply*)

apply (*subst* (*asm*) *tso-store-refs-simps*; *force*)+
done

end

lemma *WL-mo-co-mark*[*simp*]:
ghost-honorary-grey (*s p*) = {}
⟹ *WL p'* (*s*(*p* := *s p*(| *ghost-honorary-grey* := *rs* |))) = *WL p' s* ∪ { *r* |*r*. *p'* = *p* ∧ *r* ∈ *rs*}
unfolding *WL-def* **by** (*simp add*: *fun-upd-apply*)

lemma *ghost-honorary-grey-mo-co-mark*[*simp*]:
⟦ *ghost-honorary-grey* (*s p*) = {} ⟧ ⟹ *black b* (*s*(*p* := *s p*(|*ghost-honorary-grey* := {*r*}|))) ⟷ *black b s* ∧ *b* ≠ *r*
⟦ *ghost-honorary-grey* (*s p*) = {} ⟧ ⟹ *grey g* (*s*(*p* := (*s p*)(|*ghost-honorary-grey* := {*r*}|))) ⟷ *grey g s* ∨ *g* = *r*
⟦ *ghost-honorary-grey* (*s p*) = {} ⟧ ⟹ *white w* (*s*(*p* := *s p*(|*ghost-honorary-grey* := {*r*}|))) ⟷ *white w s*
unfolding *black-def grey-def* **by** (*auto simp*: *fun-upd-apply*)

lemma *ghost-honorary-grey-mo-co-W*[*simp*]:
ghost-honorary-grey (*s p'*) = {*r*}
⟹ (*WL p* (*s*(*p'* := (*s p'*)(| *W* := *insert r* (*W* (*s p'*)), *ghost-honorary-grey* := {}|)))) = (*WL p s*)
ghost-honorary-grey (*s p'*) = {*r*}
⟹ *grey g* (*s*(*p'* := (*s p'*)(| *W* := *insert r* (*W* (*s p'*)), *ghost-honorary-grey* := {}|))) ⟷ *grey g s*
unfolding *grey-def WL-def* **by** (*auto simp*: *fun-upd-apply split*: *process-name.splits if-splits*)

lemma *reachable-sweep-loop-free*:
mut-m.reachable m r (*s*(*sys* := *s sys*(|*heap* := (*sys-heap s*)(*r'* := *None*)|)))
⟹ *mut-m.reachable m r s*
unfolding *mut-m.reachable-def reaches-def* **by** (*clarsimp simp*: *fun-upd-apply*) (*metis* (*no-types*, *lifting*) *mono-rtranclp*)

lemma *reachable-deref-del*[*simp*]:
⟦ *sys-load* (*mutator m*) (*mr-Ref r f*) (*s sys*) = *mv-Ref opt-r'*; *r* ∈ *mut-m.mut-roots m s*; *mut-m.mut-ghost-honorary-root m s* = {} ⟧
⟹ *mut-m.reachable m' y* (*s*(*mutator m* := *s* (*mutator m*)(| *ghost-honorary-root* := *Option.set-option opt-r'*, *ref* := *opt-r'* |)))
⟷ *mut-m.reachable m' y s*
unfolding *mut-m.reachable-def reaches-def sys-load-def*
apply (*clarsimp simp*: *fun-upd-apply*)
apply (*rule iffI*)
apply *clarsimp*
apply (*elim disjE*)
apply *metis*
apply (*erule option-bind-invE*; *auto dest*!: *fold-stores-points-to*)
apply (*auto elim*!: *converse-rtranclp-into-rtranclp*[*rotated*]
simp: *mut-m.tso-store-refs-def*)
done

lemma *no-black-refs-dequeue*[*simp*]:
⟦ *sys-mem-store-buffers p s* = *mw-Mark r fl* # *ws*; *no-black-refs s*; *valid-W-inv s* ⟧
⟹ *no-black-refs* (*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*)|)))
⟦ *sys-mem-store-buffers p s* = *mw-Mutate r f r'* # *ws*; *no-black-refs s* ⟧
⟹ *no-black-refs* (*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*)|)))
unfolding *no-black-refs-def* **by** (*auto simp*: *fun-upd-apply dest*: *valid-W-invD*)

lemma *colours-blacken*[*simp*]:
valid-W-inv s ⟹ *black b* (*s*(*gc* := *s gc*(| *W* := *gc-W s* − {*r*}|))) ⟷ *black b s* ∨ (*r* ∈ *gc-W s* ∧ *b* = *r*)
⟦ *r* ∈ *gc-W s*; *valid-W-inv s* ⟧ ⟹ *grey g* (*s*(*gc* := *s gc*(| *W* := *gc-W s* − {*r*}|))) ⟷ (*grey g s* ∧ *g* ≠ *r*)

unfolding *black-def grey-def valid-W-inv-def*
apply (*simp-all add*: *all-conj-distrib split*: *obj-at-splits if-splits*)
apply *safe*
apply (*simp-all add*: *WL-def fun-upd-apply split*: *if-splits*)
apply (*metis option.distinct*(*1*))
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *metis*
done

lemma *no-black-refs-alloc*[*simp*]:
⟦ *heap* (*s sys*) *r'* = *None*; *no-black-refs s* ⟧
⟹ *no-black-refs* (*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*|))|)))
⟷ *fl* ≠ *sys-fM s* ∨ *grey r' s*
unfolding *no-black-refs-def* **by** *simp*

lemma *no-black-refs-mo-co-mark*[*simp*]:
⟦ *ghost-honorary-grey* (*s p*) = {}; *white r s* ⟧
⟹ *no-black-refs* (*s*(*p* := *s p*(|*ghost-honorary-grey* := {*r*}|))) ⟷ *no-black-refs s*
unfolding *no-black-refs-def* **by** *auto*

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
assume *?lhs*
then obtain *g* **where** (*g grey-protects-white w*) (*s*(*p* := *s p*(|*ghost-honorary-grey* := {*r*}|))) **by** *blast*
from *this ghg* **show** *?rhs* **by** *induct* (*auto simp*: *fun-upd-apply*)
next
assume *?rhs* **then show** *?lhs*
proof(*safe*)
fix *g* **assume** (*g grey-protects-white w*) *s*
from *this ghg* **show** *?thesis*
apply *induct*
apply *force*
unfolding *grey-protects-white-def*
apply (*auto simp*: *fun-upd-apply*)
done
next
assume (*r has-white-path-to w*) *s* **with** *ghg* **show** *?thesis*
unfolding *grey-protects-white-def has-white-path-to-def* **by** (*auto simp*: *fun-upd-apply*)
qed
qed

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(*rule valid-refs-invI*)
fix *m*
let *?root* = λ*m x*. *mut-m.root m x* ∨ *grey x*
fix *x y* **assume** *xy*: (*x reaches y*) *?s'* **and** *x*: *?root m x ?s'*
from *xy* **have** (∃ *m x*. *?root m x s* ∧ (*x reaches y*) *s*) ∧ *valid-ref y ?s'*
unfolding *reaches-def* **proof** *induct*
case *base* **with** *x sb vri* **show** *?case*
apply −
apply (*subst obj-at-fun-upd*)
apply (*auto simp*: *mut-m.tso-store-refs-def reaches-def fun-upd-apply split*: *if-splits intro*: *valid-refs-invD*(*5*)[**where** *m*=*m*])
apply (*metis list.set-intros*(*2*) *rtranclp.rtrancl-refl*)
done
next
case (*step y z*)
with *sb vri* **show** *?case*
apply −
apply (*subst obj-at-fun-upd*, *clarsimp simp*: *fun-upd-apply*)
apply (*subst* (*asm*) *obj-at-fun-upd*, *fastforce simp*: *fun-upd-apply*)
apply (*clarsimp simp*: *points-to-Mutate fun-upd-apply*)
apply (*fastforce elim*: *rtranclp.intros*(*2*) *simp*: *mut-m.tso-store-refs-def reaches-def fun-upd-apply intro*: *exI*[**where** *x*=*m'*] *valid-refs-invD*(*5*)[**where** *m*=*m'*])
done
qed
then show *valid-ref y ?s'* **by** *blast*
qed

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'*)
apply (*rule valid-refs-invI*)
using *assms*
apply (*clarsimp simp*: *valid-refs-invD fun-upd-apply split*: *obj-at-splits mem-store-action.splits*)
apply *auto*
apply (*metis* (*mono-tags*, *lifting*) *UN-insert Un-iff list.simps*(*15*) *mut-m.tso-store-refs-def valid-refs-invD*(*4*))
apply (*metis case-optionE obj-at-def valid-refs-invD*(*7*))
done

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*
unfolding *hs-get-roots-loop-locs-def hs-get-roots-locs-def* **by** (*fastforce intro*: *append-prefixD*)

lemma *hs-pending-locs-subseteq-hs-pending-loaded-locs*:
hs-pending-locs ⊆ *hs-pending-loaded-locs*
unfolding *hs-pending-locs-def hs-pending-loaded-locs-def* **by** (*fastforce intro*: *append-prefixD*)

lemma *ht-loaded-locs-subseteq-hs-pending-loaded-locs*:
ht-loaded-locs ⊆ *hs-pending-loaded-locs*
unfolding *ht-loaded-locs-def hs-pending-loaded-locs-def* **by** (*fastforce intro*: *append-prefixD*)

lemma *hs-noop-locs-subseteq-hs-pending-loaded-locs*:
hs-noop-locs ⊆ *hs-pending-loaded-locs*
unfolding *hs-noop-locs-def hs-pending-loaded-locs-def loc-defs* **by** (*fastforce intro*: *append-prefixD*)

lemma *hs-noop-locs-subseteq-hs-pending-locs*:
hs-noop-locs ⊆ *hs-pending-locs*
unfolding *hs-noop-locs-def hs-pending-locs-def loc-defs* **by** (*fastforce intro*: *append-prefixD*)

lemma *hs-noop-locs-subseteq-ht-loaded-locs*:
hs-noop-locs ⊆ *ht-loaded-locs*
unfolding *hs-noop-locs-def ht-loaded-locs-def loc-defs* **by** (*fastforce intro*: *append-prefixD*)

lemma *hs-get-roots-locs-subseteq-hs-pending-loaded-locs*:
hs-get-roots-locs ⊆ *hs-pending-loaded-locs*
unfolding *hs-get-roots-locs-def hs-pending-loaded-locs-def loc-defs* **by** (*fastforce intro*: *append-prefixD*)

lemma *hs-get-roots-locs-subseteq-hs-pending-locs*:
hs-get-roots-locs ⊆ *hs-pending-locs*
unfolding *hs-get-roots-locs-def hs-pending-locs-def loc-defs* **by** (*fastforce intro*: *append-prefixD*)

lemma *hs-get-roots-locs-subseteq-ht-loaded-locs*:
hs-get-roots-locs ⊆ *ht-loaded-locs*
unfolding *hs-get-roots-locs-def ht-loaded-locs-def loc-defs* **by** (*fastforce intro*: *append-prefixD*)

lemma *hs-get-work-locs-subseteq-hs-pending-loaded-locs*:
hs-get-work-locs ⊆ *hs-pending-loaded-locs*
unfolding *hs-get-work-locs-def hs-pending-loaded-locs-def loc-defs* **by** (*fastforce intro*: *append-prefixD*)

lemma *hs-get-work-locs-subseteq-hs-pending-locs*:
hs-get-work-locs ⊆ *hs-pending-locs*
unfolding *hs-get-work-locs-def hs-pending-locs-def loc-defs* **by** (*fastforce intro*: *append-prefixD*)

lemma *hs-get-work-locs-subseteq-ht-loaded-locs*:
hs-get-work-locs ⊆ *ht-loaded-locs*
unfolding *hs-get-work-locs-def ht-loaded-locs-def loc-defs* **by** (*fastforce intro*: *append-prefixD*)

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*
unfolding *get-roots-UN-get-work-locs-def ghost-honorary-grey-empty-locs-def hs-get-roots-locs-def hs-get-work-locs-def loc-defs*
by (*fastforce intro*: *append-prefixD*)

lemma *hs-get-roots-locs-subseteq-hp-IdleMarkSweep-locs*:
hs-get-roots-locs ⊆ *hp-IdleMarkSweep-locs*
by (*auto simp*: *hs-get-roots-locs-def hp-IdleMarkSweep-locs-def mark-loop-locs-def*
intro: *append-prefixD*)

lemma *hs-get-work-locs-subseteq-hp-IdleMarkSweep-locs*:
hs-get-work-locs ⊆ *hp-IdleMarkSweep-locs*
apply (*simp add*: *hs-get-work-locs-def hp-IdleMarkSweep-locs-def mark-loop-locs-def loc-defs*)
apply *clarsimp*
apply (*drule mp*)
apply (*auto intro*: *append-prefixD*)[*1*]
apply *auto*
done

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 (λ*r'*. *gc-field-set* ⊗ *gc-tmp-ref* ⊗ (λ*s*. *map-option obj-fields* (*sys-heap s r'*)) ⊗ (λ*s*. *map-option obj-mark* (*sys-heap s r'*)) ⊗ *sys-fM* ⊗ *tso-pending-mutate gc*)
obj-fields-marked
unfolding *eq-imp-def obj-fields-marked-def obj-at-field-on-heap-def obj-at-def*
apply (*clarsimp simp*: *all-conj-distrib*)
apply (*rule iffI*; *clarsimp split*: *option.splits*)
apply (*intro allI conjI impI*)
apply *simp-all*
apply (*metis* (*no-types*, *opaque-lifting*) *option.distinct*(*1*) *option.map-disc-iff*)
apply (*metis* (*no-types*, *lifting*) *option.distinct*(*1*) *option.map-sel option.sel*)
apply (*intro allI conjI impI*)
apply *simp-all*
apply (*metis* (*no-types*, *opaque-lifting*) *option.distinct*(*1*) *option.map-disc-iff*)
apply (*metis* (*no-types*, *lifting*) *option.distinct*(*1*) *option.map-sel option.sel*)
done

lemma *obj-fields-marked-UNIV*[*iff*]:
obj-fields-marked (*s*(*gc* := (*s gc*)(| *field-set* := *UNIV* |)))
unfolding *obj-fields-marked-def* **by** (*simp add*: *fun-upd-apply*)

lemma *obj-fields-marked-invL-eq-imp*:
eq-imp (λ*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
unfolding *eq-imp-def inv obj-at-def obj-at-field-on-heap-def*
apply (*clarsimp simp*: *all-conj-distrib cong*: *option.case-cong*)
apply (*rule iffI*)
apply (*intro conjI impI*; *clarsimp*)
apply (*subst eq-impD*[*OF obj-fields-marked-eq-imp*]; *force*)
apply (*clarsimp split*: *option.split-asm*)
apply (*metis* (*no-types*, *lifting*) *None-eq-map-option-iff option.simps*(*3*))
apply (*metis* (*no-types*, *lifting*) *option.distinct*(*1*) *option.map-sel option.sel*)
apply (*metis* (*no-types*, *lifting*) *None-eq-map-option-iff option.simps*(*3*))
apply (*metis* (*no-types*, *lifting*) *option.distinct*(*1*) *option.map-sel option.sel*)
apply (*subst* (*asm*) (*2*) *eq-impD*[*OF reaches-eq-imp*])
prefer *2* **apply** (*drule spec*, *drule mp*, *assumption*)
apply (*metis* (*no-types*) *option.disc-eq-case*(*2*) *option.map-disc-iff*)
apply (*metis option.set-map*)
apply (*clarsimp split*: *option.splits*)
apply (*metis* (*no-types*, *opaque-lifting*) *atS-simps*(*2*) *atS-un obj-fields-marked-good-ref-locs-def*)
apply (*metis* (*no-types*, *opaque-lifting*) *map-option-eq-Some option.inject option.simps*(*9*))
apply (*metis* (*no-types*, *opaque-lifting*) *map-option-eq-Some option.inject option.simps*(*9*))
apply (*metis* (*no-types*, *opaque-lifting*) *map-option-eq-Some option.inject option.simps*(*9*))

apply (*intro conjI impI*; *clarsimp*)
apply (*subst eq-impD*[*OF obj-fields-marked-eq-imp*]; *force*)
apply (*clarsimp split*: *option.split-asm*)
apply (*metis* (*no-types*, *lifting*) *None-eq-map-option-iff option.simps*(*3*))
apply (*metis* (*no-types*, *lifting*) *option.distinct*(*1*) *option.map-sel option.sel*)
apply (*metis* (*no-types*, *lifting*) *None-eq-map-option-iff option.simps*(*3*))
apply (*metis* (*no-types*, *lifting*) *option.distinct*(*1*) *option.map-sel option.sel*)
apply (*subst* (*asm*) (*2*) *eq-impD*[*OF reaches-eq-imp*])
prefer *2* **apply** (*drule spec*, *drule mp*, *assumption*)
apply (*metis* (*no-types*, *lifting*) *None-eq-map-option-iff option.case-eq-if*)
apply (*metis option.set-map*)
apply (*clarsimp split*: *option.splits*)
apply (*metis* (*no-types*, *opaque-lifting*) *atS-simps*(*2*) *atS-un obj-fields-marked-good-ref-locs-def*)
apply (*metis* (*no-types*, *opaque-lifting*) *map-option-eq-Some option.inject option.simps*(*9*))
apply (*metis* (*no-types*, *opaque-lifting*) *map-option-eq-Some option.inject option.simps*(*9*))
apply (*metis* (*no-types*, *opaque-lifting*) *map-option-eq-Some option.inject option.simps*(*9*))
done

lemma *obj-fields-marked-mark-field-done*[*iff*]:
⟦ *obj-at-field-on-heap* (λ*r*. *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*}|)))
unfolding *obj-fields-marked-def obj-at-field-on-heap-def* **by** (*fastforce simp*: *fun-upd-apply split*: *option.splits obj-at-splits*

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
unfolding *eq-imp-def*
apply *clarsimp*
apply (*rename-tac s s'*)
apply (*cut-tac s=s*↓ **and** *s'=s'*↓ **in** *eq-impD*[*OF p-ph-enabled-eq-imp*], *simp*)
apply (*clarsimp simp*: *mark-object-invL-def obj-at-def white-def*
cong: *option.case-cong*)
done

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 (λ*r 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*)
apply (*clarsimp simp*: *eq-imp-def mut-m.mark-object-invL-def fun-eq-iff*[*symmetric*] *obj-at-field-on-heap-def*
cong: *option.case-cong*)
apply (*rename-tac s s'*)
apply (*subgoal-tac* ∀ *r*. *marked r s*↓ ⟷ *marked r s'*↓)
apply (*subgoal-tac* ∀ *r*. *valid-null-ref r s*↓ ⟷ *valid-null-ref r s'*↓)
apply (*subgoal-tac* ∀ *r f opt-r'*. *mw-Mutate r f opt-r'* ∉ *set* (*sys-mem-store-buffers* (*mutator m*) *s*↓)
⟷ *mw-Mutate r f opt-r'* ∉ *set* (*sys-mem-store-buffers* (*mutator m*) *s'*↓))
apply (*clarsimp cong*: *option.case-cong*)
apply (*metis* (*mono-tags*, *lifting*) *filter-set member-filter*)
apply (*clarsimp simp*: *obj-at-def valid-null-ref-def split*: *option.splits*)
apply (*clarsimp simp*: *obj-at-def valid-null-ref-def split*: *option.splits*)
done

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* = []|)↓
unfolding *strong-tricolour-inv-def ran-def white-def* **by** (*auto split*: *obj-at-splits*)

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* = []|)↓
unfolding *no-grey-refs-def grey-def WL-def* **by** (*metis equals0D process-name.exhaust sup-bot.left-neutral*)

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*
unfolding *valid-refs-inv-def valid-refs-def mut-m.reachable-def mut-m.tso-store-refs-def*
using *no-grey-refs-not-grey-reachableD* **by** *fastforce*

lemma *no-grey-refs-imp-valid-W-inv*:
⟦ *no-grey-refs s*; ∀ *p*. *sys-mem-store-buffers p s* = [] ⟧
⟹ *valid-W-inv s*
unfolding *valid-W-inv-def no-grey-refs-def grey-def WL-def* **by** *auto*

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*
unfolding *mut-m.reachable-snapshot-inv-def in-snapshot-def valid-refs-def black-def mut-m.reachable-def mut-m.tso-stor*
apply *clarsimp*
apply (*auto simp*: *image-subset-iff ran-def split*: *obj-at-splits*)
done

lemma *init-inv-sys*: ∀ *s*. *initial-state gc-system s* ⟶ *invs* (| *GST* = *s*, *HST* = [] |)↓
apply (*clarsimp dest*!: *initial-stateD*
simp: *gc-system-init-def invs-def gc-initial-state-def mut-initial-state-def sys-initial-state-def*
inv
handshake-phase-rel-def handshake-phase-inv-def hp-step-rel-def phase-rel-inv-def phase-rel-def
tso-store-inv-def
init-no-grey-refs init-strong-tricolour-inv no-grey-refs-imp-valid-W-inv
valid-refs-imp-reachable-snapshot-inv
valid-refs-imp-valid-refs-inv
mut-m.marked-deletions-def mut-m.marked-insertions-def
fA-rel-inv-def fA-rel-def fM-rel-inv-def fM-rel-def
all-conj-distrib)
done

lemma *init-inv-mut*: ∀ *s*. *initial-state gc-system s* ⟶ *mut-m.invsL m* (| *GST* = *s*, *HST* = [] |)
apply (*clarsimp dest*!: *initial-stateD*)
apply (*drule fun-cong*[**where** *x*=*mutator m*])
apply (*clarsimp simp*: *all-com-interned-defs*)
unfolding *mut-m.invsL-def mut-m.mut-get-roots-mark-object-invL-def2 mut-m.mut-store-del-mark-object-invL-def2 mut-m.mut-store-ins-mark-object-invL-def2*
mut-m.mark-object-invL-def mut-m.handshake-invL-def mut-m.tso-lock-invL-def
gc-system-init-def mut-initial-state-def sys-initial-state-def
apply (*intro conjI*; *simp add*: *locset-cache atS-simps*; *simp add*: *mut-m.loc-defs*)
done

lemma *init-inv-gc*: ∀ *s*. *initial-state gc-system s* ⟶ *gc.invsL* (| *GST* = *s*, *HST* = [] |)
apply (*clarsimp dest*!: *initial-stateD*)
apply (*drule fun-cong*[**where** *x*=*gc*])
apply (*clarsimp simp*: *all-com-interned-defs*)
unfolding *gc.invsL-def gc.fM-fA-invL-def gc.handshake-invL-def gc.obj-fields-marked-invL-def gc.phase-invL-def gc.sweep-loop-invL-def*
gc.tso-lock-invL-def gc.gc-W-empty-invL-def gc.gc-mark-mark-object-invL-def2
apply (*intro conjI*; *simp add*: *locset-cache atS-simps init-no-grey-refs*; *simp add*: *gc.loc-defs*)
apply (*simp-all add*: *gc-system-init-def gc-initial-state-def mut-initial-state-def sys-initial-state-def*
gc-system.init-no-grey-refs)
apply *blast*

apply (*clarsimp simp*: *image-subset-iff ranI split*: *obj-at-splits*)
done

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* = []|)
unfolding *I-def invsL-def* **by** (*simp add*: *init-inv-sys init-inv-gc init-inv-mut*)

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*
unfolding *mut-m.del-barrier1-locs-def mut-m.mo-valid-ref-locs-def* **by** (*auto intro*: *append-prefixD*)

lemma *mut-del-barrier2-subseteq-mut-mo-valid-ref*[*locset-cache*]:
mut-m.ins-barrier-locs ⊆ *mut-m.mo-valid-ref-locs*
unfolding *mut-m.ins-barrier-locs-def mut-m.mo-valid-ref-locs-def* **by** (*auto intro*: *append-prefixD*)

context *gc*
begin

lemma *obj-fields-marked-locs-subseteq-hp-IdleMarkSweep-locs*:
obj-fields-marked-locs ⊆ *hp-IdleMarkSweep-locs*
unfolding *gc.obj-fields-marked-locs-def gc.hp-IdleMarkSweep-locs-def gc.mark-loop-locs-def gc.mark-loop-mo-locs-def*
apply (*clarsimp simp*: *locset-cache loc-defs*)
apply (*drule mp*)
apply (*auto intro*: *append-prefixD*)
done

lemma *obj-fields-marked-locs-subseteq-hs-in-sync-locs*:
obj-fields-marked-locs ⊆ *hs-in-sync-locs*
unfolding *obj-fields-marked-locs-def hs-in-sync-locs-def hs-done-locs-def mark-loop-mo-locs-def*
by (*auto simp*: *loc-defs dest*: *prefix-same-cases*)

lemma *obj-fields-marked-good-ref-subseteq-hp-IdleMarkSweep-locs*:
obj-fields-marked-good-ref-locs ⊆ *hp-IdleMarkSweep-locs*
unfolding *obj-fields-marked-good-ref-locs-def mark-loop-locs-def hp-IdleMarkSweep-locs-def mark-loop-mo-locs-def*
apply (*clarsimp simp*: *loc-defs*)
apply (*drule mp*)
apply (*auto intro*: *append-prefixD*)
done

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

unfolding *obj-fields-marked-good-ref-locs-def hs-in-sync-locs-def mark-loop-mo-locs-def hs-done-locs-def*
by (*auto simp*: *loc-defs dest*: *prefix-same-cases*)

lemma *no-grey-refs-locs-subseteq-hs-in-sync-locs*:
no-grey-refs-locs ⊆ *hs-in-sync-locs*
by (*auto simp*: *no-grey-refs-locs-def black-heap-locs-def hs-in-sync-locs-def hs-done-locs-def sweep-locs-def loc-defs*
dest: *prefix-same-cases*)

lemma *get-roots-UN-get-work-locs-subseteq-gc-W-empty-locs*:
get-roots-UN-get-work-locs ⊆ *gc-W-empty-locs*
unfolding *get-roots-UN-get-work-locs-def*
by (*auto simp*: *hs-get-roots-locs-def hs-get-work-locs-def gc-W-empty-locs-def*)

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*))
unfolding *gc.handshake-invL-def*
by (*metis* (*no-types*, *lifting*) *atS-mono gc.obj-fields-marked-locs-subseteq-hp-IdleMarkSweep-locs gc.obj-fields-marked-locs*

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*))
unfolding *gc.handshake-invL-def*
by (*metis* (*no-types*, *lifting*) *atS-mono gc.mark-loop-mo-mark-loop-field-done-subseteq-hs-in-sync-locs gc.obj-fields-marke*

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 −
from *xy xys*
have ∃ *z*. *z* ∈ {*x*} ∪ *mut-m.tso-store-refs m s* ∧ (*z reaches y*) *s* ∧ *valid-ref y s*
proof *induct*
case (*refl x*) **then show** *?case* **by** *auto*
next
case (*step x y z*) **with** *sb vri* **show** *?case*
apply (*clarsimp simp*: *points-to-Mutate*)
apply (*elim disjE*)
apply (*metis* (*no-types*, *lifting*) *obj-at-cong reaches-def rtranclp.rtrancl-into-rtrancl*)
apply (*metis* (*no-types*, *lifting*) *obj-at-def option.case*(*2*) *reaches-def rtranclp.rtrancl-into-rtrancl valid-refs-invD*(*4*)
apply *clarsimp*
apply (*elim disjE*)
apply (*rule exI*[**where** *x*=*z*])
apply (*clarsimp simp*: *mut-m.tso-store-refs-def*)

apply (*rule valid-refs-invD*(*3*)[**where** *m*=*m* **and** *x*=*z*], *auto simp*: *mut-m.tso-store-refs-def*; *fail*)[*1*]
apply (*metis* (*no-types*, *lifting*) *obj-at-cong reaches-def rtranclp.rtrancl-into-rtrancl*)
apply *clarsimp*
apply (*elim disjE*)
apply (*rule exI*[**where** *x*=*z*])
apply (*clarsimp simp*: *mut-m.tso-store-refs-def*)
apply (*rule valid-refs-invD*(*3*)[**where** *m*=*m* **and** *x*=*z*], *auto simp*: *mut-m.tso-store-refs-def*)[*1*]
apply (*metis* (*no-types*, *lifting*) *obj-at-def option.case*(*2*) *reaches-def rtranclp.rtrancl-into-rtrancl valid-refs-invD*(*4*))
done
qed
then show *?thesis* **by** *blast*
qed

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(*vcg-jackhammer* (*keep-locs*) (*no-thin-post-inv*), *vcg-name-cases*)
case (*tso-dequeue-store-buffer s s′ p w ws*) **show** *?case*
proof(*cases w*)
case (*mw-Mark ref mark*) **with** *tso-dequeue-store-buffer* **show** *?thesis*
apply −
apply (*clarsimp simp*: *p-not-sys gc.obj-fields-marked-invL-def*)
apply (*intro conjI impI*; *clarsimp*)

apply (*frule* (*1*) *handshake-obj-fields-markedD*)
apply (*clarsimp simp*: *gc.obj-fields-marked-def*)
apply (*frule* (*1*) *valid-W-invD*)
apply (*drule-tac x*=*x* **in** *spec*)
apply *clarsimp*
apply (*erule obj-at-field-on-heapE*)
apply (*force split*: *obj-at-splits*)
apply (*force split*: *obj-at-splits*)

apply (*erule obj-at-field-on-heapE*)
apply (*clarsimp split*: *obj-at-splits*; *fail*)
apply (*clarsimp split*: *obj-at-splits*)
apply (*metis valid-W-invD*(*1*))
apply (*metis valid-W-invD*(*1*))

apply (*force simp*: *valid-W-invD*(*1*) *split*: *obj-at-splits*)
done
next case (*mw-Mutate r f opt-r′*) **with** *tso-dequeue-store-buffer* **show** *?thesis*
apply −
apply (*clarsimp simp*: *p-not-sys gc.obj-fields-marked-invL-def*)
apply (*erule disjE*; *clarsimp*)
apply (*rename-tac m*)
apply (*drule-tac m*=*m* **in** *mut-m.handshake-phase-invD*; *clarsimp simp*: *hp-step-rel-def*)
apply (*drule-tac x*=*m* **in** *spec*)
apply (*intro conjI impI*; *clarsimp simp*: *obj-at-field-on-heap-imp-valid-ref gc-marking-reaches-Mutate split*: *option.splits*)

subgoal for *m*

apply (*frule* (*1*) *handshake-obj-fields-markedD*)
apply (*elim disjE*; *auto simp*: *gc.obj-fields-marked-def split*: *option.splits*)
done

subgoal for *m r′*
apply (*frule* (*1*) *obj-fields-marked-good-ref-locs-hp-phaseD*)
apply (*elim disjE*; *clarsimp simp*: *marked-insertionD*)
done
done
next case (*mw-Mutate-Payload r f pl*) **with** *tso-dequeue-store-buffer* **show** *?thesis* **by** − (*erule gc-obj-fields-marked-inv clarsimp*)
next case (*mw-fA mark*) **with** *tso-dequeue-store-buffer* **show** *?thesis* **by** − (*erule gc-obj-fields-marked-invL-niE*; *clarsimp*)
next case (*mw-fM mark*) **with** *tso-dequeue-store-buffer* **show** *?thesis*
apply −
apply (*clarsimp simp*: *p-not-sys fM-rel-inv-def fM-rel-def gc.obj-fields-marked-invL-def*)
apply (*erule disjE*; *clarsimp*)
apply (*intro conjI impI*; *clarsimp*)
apply (*metis* (*no-types*, *lifting*) *handshake-obj-fields-markedD hs-phase.distinct*(*7*))
apply (*metis* (*no-types*, *lifting*) *hs-phase.distinct*(*7*) *obj-fields-marked-good-ref-locs-hp-phaseD*)
apply (*metis* (*no-types*, *lifting*) *UnCI elem-set hs-phase.distinct*(*7*) *gc.obj-fields-marked-good-ref-locs-def obj-fields-marked-good-ref-locs-hp-phaseD option.simps*(*15*) *thin-locs-pre-keep-atSE*)
done
next case (*mw-Phase ph*) **with** *tso-dequeue-store-buffer* **show** *?thesis*
by − (*erule gc-obj-fields-marked-invL-niE*; *clarsimp*)
qed
qed

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
by (*simp add*: *eq-imp-def gc-W-empty-mut-inv-def*)

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
by (*simp add*: *eq-imp-def gc-W-empty-invL-def mut-m.gc-W-empty-mut-inv-def no-grey-refs-def grey-def*)

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*
apply (*clarsimp simp*: *mut-m.gc-W-empty-mut-inv-def no-grey-refs-def grey-def*)
apply (*rename-tac x xa*)
apply (*case-tac xa*)
apply (*simp-all add*: *WL-def*)
done

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*′
|)))
by (*simp-all add*: *mut-m.gc-W-empty-mut-inv-def*)

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* |}
apply (*vcg-jackhammer*; (*clarsimp elim*: *gc-W-empty-mut-inv-load-W simp*: *WL-def*) *?*)
proof *vcg-name-cases*
case (*mark-loop-get-work-done-loop s s*′) **then show** *?case*
by (*simp add*: *WL-def gc-W-empty-mut-inv-load-W valid-W-inv-sys-ghg-empty-iff*)
next case (*mark-loop-get-roots-done-loop s s*′) **then show** *?case*
by (*simp add*: *WL-def gc-W-empty-mut-inv-load-W valid-W-inv-sys-ghg-empty-iff*)
qed

end

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

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*↓ = {}
unfolding *gc.obj-fields-marked-invL-def*
using *atS-mono*[*OF - gc.get-roots-UN-get-work-locs-subseteq-ghost-honorary-grey-empty-locs*]
apply *metis*
done

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*}
unfolding *gc.handshake-invL-def*
apply *clarsimp*
apply (*metis* (*no-types*, *lifting*) *atS-mono atS-un gc.get-roots-UN-get-work-locs-def gc.hs-get-roots-locs-subseteq-hp-IdleM
gc.hs-get-work-locs-subseteq-hp-IdleMarkSweep-locs*)
done

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*))
apply (*simp add*: *gc.handshake-invL-def*)
apply (*elim conjE*)
apply (*drule mp*, *erule atS-mono*[*OF - gc.no-grey-refs-locs-subseteq-hs-in-sync-locs*])
apply (*drule mut-m.handshake-phase-invD*)
apply (*simp only*: *gc.no-grey-refs-locs-def cong del*: *atS-state-weak-cong*)
apply (*clarsimp simp*: *atS-un*)

apply (*elim disjE*)
 apply (*drule mp*, *erule atS-mono*[**where** *ls′=gc.hp-IdleMarkSweep-locs*])
  apply (*clarsimp simp*: *gc.black-heap-locs-def locset-cache*)
 apply (*clarsimp simp*: *hp-step-rel-def*)
 apply *blast*
apply (*drule mp*, *erule atS-mono*[**where** *ls′=gc.hp-IdleMarkSweep-locs*])
 apply (*clarsimp simp*: *hp-IdleMarkSweep-locs-def hp-step-rel-def*)
apply (*clarsimp simp*: *hp-step-rel-def*)
apply *blast*
apply (*clarsimp simp*: *atS-simps locset-cache hp-step-rel-def*)
apply *blast*
done

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*}|)))
apply (*frule* (*1*) *gc-hs-get-roots-get-workD*)
apply (*frule-tac m=m* **in** *mut-m.handshake-phase-invD*)
apply (*clarsimp simp*: *hp-step-rel-def simp del*: *Un-iff*)
apply (*elim disjE*, *simp-all*)
proof(*goal-cases before-get-work past-get-work before-get-roots after-get-roots*)
 case *before-get-work* **then show** *?thesis*
  apply (*clarsimp simp*: *mut-m.gc-W-empty-mut-inv-def*)
  apply *blast*
  done
next case *past-get-work* **then show** *?thesis*
  apply (*clarsimp simp*: *mut-m.gc-W-empty-mut-inv-def*)
  apply (*frule spec*[**where** *x=m*], *clarsimp*)
  apply (*frule* (*2*) *mut-m.reachable-snapshot-inv-white-root*)
  apply *clarsimp*
  apply (*drule grey-protects-whiteD*)
  apply (*clarsimp simp*: *grey-def*)
  apply (*rename-tac g p*)
  apply (*case-tac p*; *clarsimp*)

   apply *blast*

  apply (*frule* (*1*) *empty-WL-GC*)
  apply (*drule mp*, *erule atS-mono*[*OF - get-roots-UN-get-work-locs-subseteq-gc-W-empty-locs*])
  apply (*clarsimp simp*: *WL-def*; *fail*)

 apply (*clarsimp simp*: *WL-def valid-W-inv-sys-ghg-empty-iff*; *fail*)
 done
next case *before-get-roots* **then show** *?case*
 apply (*clarsimp simp*: *mut-m.gc-W-empty-mut-inv-def*)
 apply *blast*
 done
next case *after-get-roots* **then show** *?case*
 apply (*clarsimp simp*: *mut-m.gc-W-empty-mut-inv-def*)
 apply (*frule spec*[**where** *x=m*], *clarsimp*)
 apply (*frule* (*2*) *mut-m.reachable-snapshot-inv-white-root*)
 apply *clarsimp*
 apply (*drule grey-protects-whiteD*)

apply (*clarsimp simp*: *grey-def*)
apply (*rename-tac g p*)
apply (*case-tac p*; *clarsimp*)

apply *blast*

apply (*frule* (*1*) *empty-WL-GC*)
apply (*drule mp*, *erule atS-mono*[*OF - get-roots-UN-get-work-locs-subseteq-gc-W-empty-locs*])
apply (*clarsimp simp*: *WL-def*; *fail*)

apply (*clarsimp simp*: *WL-def valid-W-inv-sys-ghg-empty-iff*; *fail*)
done
qed

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*}|)))
apply (*elim disjE*)
apply (*clarsimp simp*: *atS-simps gc.handshake-invL-def locset-cache*)
apply (*frule mut-m.handshake-phase-invD*)
apply (*clarsimp simp*: *hp-step-rel-def*)
apply (*drule spec*[**where** *x*=*m*])
apply (*clarsimp simp*: *conj-disj-distribR*[*symmetric*])
apply (*simp add*: *handshake-in-syncD mut-m.no-grey-refs-not-rootD*; *fail*)
apply (*clarsimp simp*: *atS-simps gc.handshake-invL-def locset-cache*)
apply (*frule mut-m.handshake-phase-invD*)
apply (*clarsimp simp*: *hp-step-rel-def*)
apply (*drule spec*[**where** *x*=*m*])
apply (*simp add*: *handshake-in-syncD mut-m.no-grey-refs-not-rootD*; *fail*)
apply (*clarsimp simp*: *atS-simps gc.handshake-invL-def locset-cache*)
apply (*frule mut-m.handshake-phase-invD*)
apply (*clarsimp simp*: *hp-step-rel-def*)
apply (*drule spec*[**where** *x*=*m*])
apply (*simp add*: *handshake-in-syncD mut-m.no-grey-refs-not-rootD*; *fail*)
apply (*frule* (*2*) *handshake-sweep-mark-endD*)
apply (*drule spec*[**where** *x*=*m*])
apply *clarsimp*
apply (*simp add*: *handshake-in-syncD mut-m.no-grey-refs-not-rootD*; *fail*)
done

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(vcg-chainsaw gc.gc-W-empty-invL-def, vcg-name-cases)
    case (hs-noop-done s s' x)      then show ?case
     unfolding gc.handshake-invL-def
     by (metis atS-un gc.get-roots-UN-get-work-locs-def hs-type.distinct(1) hs-type.distinct(3))
next case (hs-get-roots-done0 s s' x) then show ?case
     apply (clarsimp simp: mut-m.gc-W-empty-mut-inv-def WL-def)
     apply (metis (no-types, lifting))
     done
next case (hs-get-work-done0 s s' x)  then show ?case
     apply (clarsimp simp: mut-m.gc-W-empty-mut-inv-def WL-def)
     apply (metis (no-types, lifting))
     done
qed (simp-all add: no-grey-refs-def)

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 ⦄
apply (vcg-chainsaw mut-m.mark-object-invL-def mut-m.mut-store-del-mark-object-invL-def2) — at gc sweep-loop-free
s
 apply (metis (no-types, lifting) handshake-in-syncD mut-m.mutator-phase-inv-aux.simps(5) mut-m.no-grey-refs-not-ro
obj-at-cong white-def)+
done

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 ⦄
apply (vcg-chainsaw mut-m.mark-object-invL-def mut-m.mut-store-ins-mark-object-invL-def2) — at gc sweep-loop-free
s
 apply (metis (no-types, lifting) handshake-in-syncD mut-m.mutator-phase-inv-aux.simps(5) mut-m.no-grey-refs-not-ro
obj-at-cong white-def)+
done

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(*vcg-chainsaw mut-m.handshake-invL-def mut-m.mark-object-invL-def*, *vcg-name-cases mutator m*) — *at gc sweep-loop-free s*
case (*ins-barrier-locs s s'*) **then show** *?case*
apply −
apply (*drule-tac x=m* **in** *spec*)
apply (*clarsimp simp*: *fun-upd-apply dest*!: *handshake-in-syncD split*: *obj-at-field-on-heap-splits*)
apply (*metis* (*no-types*, *lifting*) *mut-m.no-grey-refs-not-rootD obj-at-cong white-def*)
apply (*metis* (*no-types*) *marked-not-white mut-m.no-grey-refs-not-rootD whiteI*)
done
next case (*del-barrier1-locs s s'*) **then show** *?case*
apply −
apply (*drule-tac x=m* **in** *spec*)
apply (*clarsimp simp*: *fun-upd-apply dest*!: *handshake-in-syncD split*: *obj-at-field-on-heap-splits*)
apply (*metis* (*no-types*, *lifting*) *mut-m.no-grey-refs-not-rootD obj-at-cong white-def*)
apply (*metis* (*no-types*, *lifting*) *marked-not-white mut-m.no-grey-refs-not-rootD obj-at-cong white-def*)
done
qed *blast*+

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*
unfolding *mut-m.handshake-invL-def*
apply (*elim conjE*)
apply (*drule mut-m.handshake-phase-invD*[**where** *m=m*])
apply (*drule fM-rel-invD*)
apply (*clarsimp simp*: *hp-step-rel-def fM-rel-def filter-empty-conv p-not-sys*)
apply (*metis* (*full-types*) *hs-phase.distinct*(*7*) *list.set-intros*(*1*) *tso-store-invD*(*4*))
done

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(*vcg-chainsaw mut-m.mark-object-invL-def*, *vcg-name-cases mutator m*)
case (*hs-get-roots-loop-locs s s' p w ws x*) **then show** *?case*
apply −
apply (*cases w*; *clarsimp split*: *obj-at-splits*)
apply (*meson valid-W-invD*(*1*))
apply (*simp add*: *atS-mono mut-m.hs-get-roots-loop-locs-subseteq-hs-get-roots-locs mut-m-get-roots-no-fM-write*)
done
next case (*hs-get-roots-loop-done s s' p w ws y*) **then show** *?case*
apply −
apply (*cases w*; *clarsimp simp*: *p-not-sys valid-W-invD split*: *obj-at-splits*)
apply (*rename-tac fl obj*)
apply (*drule-tac fl=fl* **and** *p=p* **and** *ws=ws* **in** *mut-m-get-roots-no-fM-write*; *clarsimp*)
apply (*drule mp*, *erule atS-simps*, *loc-mem*)
apply *blast*
done

next case (*hs-get-roots-done s s′ p w ws x*) **then show** *?case*
apply −
apply (*cases w*; *clarsimp simp*: *p-not-sys valid-W-invD split*: *obj-at-splits*)
apply *blast*
apply (*rename-tac fl*)
apply (*drule-tac fl=fl* **and** *p=p* **and** *ws=ws* **in** *mut-m-get-roots-no-fM-write*; *clarsimp*)
apply (*drule mp*, *erule atS-simps*, *loc-mem*)
apply *blast*
done
next case (*mo-ptest-locs s s′ p ws ph′*) **then show** *?case* **by** (*clarsimp simp*: *p-not-sys*; *elim disjE*; *clarsimp simp*: *phase-rel-def handshake-in-syncD dest*!: *phase-rel-invD*)
next case (*store-ins s s′ p w ws y*) **then show** *?case*
apply −
apply (*cases w*; *clarsimp simp*: *p-not-sys valid-W-invD split*: *obj-at-splits*)
apply (*metis* (*no-types*, *lifting*) *hs-phase.distinct*(*3*, *5*) *mut-m.mut-ghost-handshake-phase-idle mut-m-not-idle-no-f store-ins*(*9*))
using *valid-refs-invD*(*9*) **apply** *fastforce*
apply (*elim disjE*; *clarsimp simp*: *phase-rel-def handshake-in-syncD dest*!: *phase-rel-invD*)
done
next case (*del-barrier1-locs s s′ p w ws*) **then show** *?case*
proof(*cases w*)
case (*mw-Mutate r f opt-r′*) **with** *del-barrier1-locs* **show** *?thesis*
apply (*clarsimp simp*: *p-not-sys*; *elim disjE*; *clarsimp*)
apply (*intro conjI impI*; *clarsimp simp*: *obj-at-field-on-heap-imp-valid-ref split*: *option.splits*)
apply (*intro conjI impI*; *clarsimp*)
apply (*smt* (*z3*) *reachableI*(*1*) *valid-refs-invD*(*8*))
apply (*metis* (*no-types*, *lifting*) *marked-insertionD mut-m.mutator-phase-inv-aux.simps*(*4*) *mut-m.mutator-phase-inv-a obj-at-cong reachableI*(*1*) *valid-refs-invD*(*8*))

apply (*rename-tac ma x2*)
apply (*frule-tac m=m* **in** *mut-m.handshake-phase-invD*)
apply (*frule-tac m=ma* **in** *mut-m.handshake-phase-invD*)
apply (*frule spec*[**where** *x=m*])
apply (*drule-tac x=ma* **in** *spec*)
apply (*clarsimp simp*: *hp-step-rel-def*)
apply (*elim disjE*; *clarsimp simp*: *marked-insertionD mut-m.mut-ghost-handshake-phase-idle*)
done
next case (*mw-fM fl*) **with** *del-barrier1-locs mut-m-not-idle-no-fM-writeD* **show** *?thesis* **by** *fastforce*
next case (*mw-Phase ph*) **with** *del-barrier1-locs* **show** *?thesis* **by** (*clarsimp simp*: *p-not-sys*; *elim disjE*; *clarsimp simp*: *phase-rel-def handshake-in-syncD dest*!: *phase-rel-invD*)
qed (*fastforce simp*: *valid-W-invD split*: *obj-at-field-on-heap-splits obj-at-splits*)+
next case (*ins-barrier-locs s s′ p w ws*) **then show** *?case*
proof(*cases w*)
case (*mw-Mutate r f opt-r′*) **with** *ins-barrier-locs* **show** *?thesis*
apply (*clarsimp simp*: *p-not-sys*; *elim disjE*; *clarsimp*)
apply (*intro conjI impI*; *clarsimp simp*: *obj-at-field-on-heap-imp-valid-ref split*: *option.splits*)
apply (*intro conjI impI*; *clarsimp*)
apply (*smt* (*z3*) *reachableI*(*1*) *valid-refs-invD*(*8*))
apply (*metis* (*no-types*, *lifting*) *marked-insertionD mut-m.mutator-phase-inv-aux.simps*(*4*) *mut-m.mutator-phase-inv-a obj-at-cong reachableI*(*1*) *valid-refs-invD*(*8*))

apply (*rename-tac ma x2*)
apply (*frule-tac m=m* **in** *mut-m.handshake-phase-invD*)
apply (*frule-tac m=ma* **in** *mut-m.handshake-phase-invD*)
apply (*frule spec*[**where** *x=m*])
apply (*drule-tac x=ma* **in** *spec*)
apply (*clarsimp simp*: *hp-step-rel-def*)
apply (*elim disjE*; *clarsimp simp*: *marked-insertionD mut-m.mut-ghost-handshake-phase-idle*)

done
next case (*mw-fM fl*) **with** *ins-barrier-locs mut-m-not-idle-no-fM-writeD* **show** *?thesis* **by** *fastforce*
next case (*mw-Phase ph*) **with** *ins-barrier-locs* **show** *?thesis* **by** (*clarsimp simp*: *p-not-sys*; *elim disjE*; *clarsimp simp*: *phase-rel-def handshake-in-syncD dest*!: *phase-rel-invD*)
qed (*fastforce simp*: *valid-W-invD split*: *obj-at-field-on-heap-splits obj-at-splits*)+
next case (*lop-store-ins s s′ p w ws y*) **then show** *?case*
apply −
apply (*cases w*; *clarsimp simp*: *valid-W-invD*(*1*) *split*: *obj-at-splits*)
apply (*metis* (*no-types*, *opaque-lifting*) *hs-phase.distinct*(*5*,*7*) *mut-m-not-idle-no-fM-write*)
apply (*clarsimp simp*: *p-not-sys*; *elim disjE*; *clarsimp simp*: *phase-rel-def handshake-in-syncD dest*!: *phase-rel-invD*; *fail*)+
done
qed

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* ‖}
unfolding *strong-tricolour-inv-def*
proof(*vcg-jackhammer* (*no-thin-post-inv*), *vcg-name-cases*)
case (*tso-dequeue-store-buffer s s′ p w ws x xa*) **then show** *?case*
proof(*cases w*)
case (*mw-Mark ref field*) **with** *tso-dequeue-store-buffer* **show** *?thesis*
apply −
apply *clarsimp*
apply (*frule* (*1*) *valid-W-invD*)
apply *clarsimp*
apply (*cases x* = *ref*; *clarsimp simp*: *grey-def white-def WL-def split*: *if-splits*)
apply (*drule-tac x*=*x* **in** *spec*; *force split*: *obj-at-splits*)
done
next case (*mw-Mutate ref field opt-r′*) **with** *tso-dequeue-store-buffer* **show** *?thesis*
apply −
apply (*clarsimp simp*: *fM-rel-inv-def p-not-sys*)
apply (*elim disjE*; *clarsimp simp*: *points-to-Mutate*)
apply (*elim disjE*; *clarsimp*)
apply (*case-tac sys-ghost-hs-phase s*↓; *clarsimp simp*: *hp-step-rel-def heap-colours-colours no-black-refsD*)
proof(*goal-cases hp-InitMark hp-Mark hp-IdleMarkSweep*)
case (*hp-InitMark m*) **then show** *?case*
apply −
apply (*drule mut-m.handshake-phase-invD*[**where** *m*=*m*])
apply (*drule-tac x*=*m* **in** *spec*)
apply (*elim disjE*; *clarsimp simp*: *hp-step-rel-def*)
apply (*elim disjE*; *clarsimp simp*: *mut-m.marked-insertions-def no-black-refsD marked-not-white*)
done
next case (*hp-Mark m*) **then show** *?case*
apply −
apply (*drule mut-m.handshake-phase-invD*[**where** *m*=*m*])
apply (*drule-tac x*=*m* **in** *spec*)
apply (*elim disjE*; *clarsimp simp*: *hp-step-rel-def*)

```
        apply (elim disjE; clarsimp simp: mut-m.marked-insertions-def no-black-refsD)
        apply blast+
        done
    next case (hp-IdleMarkSweep m) then show ?case
        apply −
        apply (drule mut-m.handshake-phase-invD[where m=m])
        apply (drule-tac x=m in spec)
        apply (elim disjE; clarsimp simp: hp-step-rel-def)
        apply (elim disjE; clarsimp simp: marked-not-white mut-m.marked-insertions-def)
        done
    qed
  next case (mw-fM fM) with tso-dequeue-store-buffer show ?thesis
    apply −
    apply (clarsimp simp: fM-rel-inv-def p-not-sys)
    apply (erule disjE)
     apply (clarsimp simp: fM-rel-def black-heap-def split: if-splits)
      apply (metis colours-distinct(2) white-valid-ref)
     apply (clarsimp simp: white-heap-def)
     apply ( (drule-tac x=xa in spec)+ )[1]
     apply (clarsimp simp: white-def split: obj-at-splits)
    apply (fastforce simp: white-def)
    done
  qed (clarsimp simp: fM-rel-inv-def p-not-sys)+
qed

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
using assms
apply (induct rule: reachable-induct)
apply (simp-all add: black-heap-def valid-refs-invD)
apply (metis (full-types) reachable-points-to valid-refs-inv-def)
done

lemma black-heap-valid-ref-marked-insertions:
  ⟦ black-heap s; valid-refs-inv s ⟧ ⟹ mut-m.marked-insertions m s
by (auto simp: mut-m.marked-insertions-def black-heap-def black-def
        split: mem-store-action.splits option.splits
         dest: valid-refs-invD)

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 (λobj. 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′)
apply (rule mut-m.reachable-snapshot-invI)
apply (rule in-snapshotI)
apply (erule black-heap-reachable)
 using bh vri
```

apply (*simp add*: *black-heap-def fun-upd-apply*; *fail*)
using *bh ngr sb vri*
apply (*subst valid-refs-inv-def*)
apply (*clarsimp simp add*: *no-grey-refs-def grey-reachable-def fun-upd-apply*)
apply (*drule black-heap-reachable*)
apply (*simp add*: *black-heap-def fun-upd-apply*; *fail*)
apply (*clarsimp simp*: *valid-refs-inv-dequeue-Mutate*; *fail*)
apply (*clarsimp simp*: *in-snapshot-def in-snapshot-valid-ref fun-upd-apply*)
done

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*)|)))
unfolding *mut-m.marked-deletions-def*
by (*auto simp*: *fun-upd-apply obj-at-field-on-heap-def*
split: *obj-at-splits option.splits mem-store-action.splits*
dest: *valid-W-invD*)

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*)|)))
unfolding *mut-m.marked-insertions-def mut-m.marked-deletions-def*
apply (*clarsimp simp*: *fun-upd-apply split*: *mem-store-action.splits option.splits*)
apply (*metis list.set-intros*(*2*) *obj-at-field-on-heap-imp-valid-ref*(*1*))+
done

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(*rule iffI*)
assume ∃ *g*. (*g grey-protects-white w*) *?s'*
then obtain *g* **where** (*g grey-protects-white w*) *?s'* **by** *blast*
from *this assms* **show** *?rhs*
proof *induct*
case (*step x y z*) **then show** *?case*
apply (*cases y* = *r*; *clarsimp simp*: *fun-upd-apply*)
apply (*metis black-dequeue-Mark colours-distinct*(*2*) *do-store-action-simps*(*1*) *greyI*(*1*) *grey-protects-whiteE*(*1*) *grey-protects-whiteI marked-imp-black-or-grey*(*2*) *valid-ref-valid-null-ref-simps*(*1*) *white-valid-ref*)
apply (*metis black-dequeue-Mark colours-distinct*(*2*) *do-store-action-simps*(*1*) *grey-protects-whiteE*(*2*) *grey-protects-marked-imp-black-or-grey*(*2*) *valid-ref-valid-null-ref-simps*(*1*) *white-valid-ref*)
done
qed (*fastforce simp*: *fun-upd-apply*)
next
assume *?rhs*
then obtain *g'* **where** (*g' grey-protects-white w*) *s* **..**
then show ∃ *g*. (*g grey-protects-white w*) *?s'*
proof *induct*
case (*refl g*) **with** *assms* **show** *?case*
apply −
apply (*rule exI*[**where** *x*=*g*])
apply (*rule grey-protects-whiteI*)

apply (*subst grey-fun-upd*; *simp add*: *fun-upd-apply*)
done
next
case (*step x y z*) **with** *assms* **show** *?case*
apply *clarsimp*
apply (*rename-tac g*)
apply (*clarsimp simp add*: *grey-protects-white-def*)
apply (*case-tac z = r*)
apply (*rule exI*[**where** *x*=*r*])
apply (*clarsimp simp add*: *grey-protects-white-def*)
apply (*subst grey-fun-upd*; *force simp*: *fun-upd-apply*)
apply (*rule-tac x*=*g* **in** *exI*)
apply (*fastforce elim*!: *has-white-path-to-step*)
done
qed
qed

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*)|)))
unfolding *mut-m.reachable-snapshot-inv-def in-snapshot-def*
apply *clarsimp*
apply (*rename-tac x*)
apply (*drule-tac x*=*x* **in** *spec*)
apply (*subst* (*asm*) *arg-cong*[**where** *f*=*Not*, *OF grey-protects-white-dequeue-Mark*, *simplified*]; *simp add*: *colours-distinc valid-W-invD*(*1*) *fun-upd-apply*)
done

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*)|)))
apply (*clarsimp simp*: *mut-m.marked-insertions-def*)
apply (*cases mutator m = p*)
apply *clarsimp*
apply (*rename-tac x*)
apply (*drule-tac x*=*x* **in** *spec*)
apply (*auto simp*: *valid-W-invD split*: *mem-store-action.splits option.splits obj-at-splits*; *fail*)
apply *clarsimp*
apply (*rename-tac x*)
apply (*drule-tac x*=*x* **in** *spec*)
apply (*auto simp*: *valid-W-invD split*: *mem-store-action.splits option.splits obj-at-splits*)
done

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*)|)))
unfolding *mut-m.marked-insertions-def*
apply (*cases mutator m = p*)
apply *clarsimp*
apply (*rename-tac x*)
apply (*drule-tac x*=*x* **in** *spec*)
apply (*auto simp*: *fun-upd-apply split*: *mem-store-action.splits option.splits obj-at-splits*; *fail*)[*1*]
apply *clarsimp*

```
apply (rename-tac x)
apply (drule-tac x=x in spec)
apply (auto simp: fun-upd-apply split: mem-store-action.splits option.splits obj-at-splits)[1]
done


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 (λobj. 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
  assume (∃ g. (g grey-protects-white w) ?s')
  then obtain g where (g grey-protects-white w) ?s' by blast
  from this mi sb show ?rhs
  proof(induct rule: grey-protects-white-induct)
      case (refl x) then show ?case by (fastforce simp: fun-upd-apply)
  next case (step x y z) then show ?case
       unfolding white-def
       apply (clarsimp simp: points-to-Mutate grey-protects-white-def)
       apply (auto dest: marked-insertionD simp: marked-not-white whiteI fun-upd-apply)
       done
  qed
next
  assume ?rhs then show (∃ g. (g grey-protects-white w) ?s')
  proof(clarsimp)
    fix g assume (g grey-protects-white w) s
    from this show ?thesis
    proof(induct rule: grey-protects-white-induct)
        case (refl x) then show ?case
         apply -
         apply (rule exI[where x=x])
         apply (clarsimp simp: grey-protects-white-def)
         apply (subst grey-fun-upd; simp add: fun-upd-apply)
         done
    next case (step x y z) with md sb show ?case
      apply clarsimp
      apply (clarsimp simp: grey-protects-white-def)
      apply (rename-tac g)

      apply (case-tac y = r)
       defer
       apply (auto simp: points-to-Mutate fun-upd-apply elim!: has-white-path-to-step; fail)[1]
      apply (clarsimp simp: ran-def fun-upd-apply split: obj-at-split-asm)
      apply (rename-tac g obj aa)
      apply (case-tac aa = f)
       defer
       apply (rule-tac x=g in exI)
       apply clarsimp
       apply (clarsimp simp: has-white-path-to-def fun-upd-apply)
       apply (erule rtranclp.intros)
       apply (auto simp: fun-upd-apply ran-def split: obj-at-splits; fail)[1]

      apply (clarsimp simp: has-white-path-to-def)
      apply (clarsimp simp: mut-m.marked-deletions-def)
```

```
    apply (drule spec[where x=mw-Mutate r f opt-r'])
    apply (clarsimp simp: obj-at-field-on-heap-def)
    apply (simp add: white-def split: obj-at-splits)
    done
  qed
 qed
qed


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 (λobj. 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(rule mut-m.reachable-snapshot-invI)
 fix y assume y: mut-m.reachable m y ?s'
 then have (mut-m.reachable m y s ∨ mut-m.reachable m' y s) ∧ in-snapshot y ?s'
 proof(induct rule: reachable-induct)
   case (root x) with mi md rsi sb show ?case
    apply (clarsimp simp: mut-m.reachable-snapshot-inv-def in-snapshot-def)
    apply (auto simp: fun-upd-apply)
    done
 next
   case (ghost-honorary-root x) with mi md rsi sb show ?case
    unfolding mut-m.reachable-snapshot-inv-def in-snapshot-def by (auto simp: fun-upd-apply)
 next
   case (tso-root x) with mi md rsi sb show ?case
    apply (clarsimp simp: mut-m.reachable-snapshot-inv-def in-snapshot-def)
     apply (rename-tac w)
     apply (case-tac w; simp)
      apply (rename-tac ref field option)
      apply (clarsimp simp: mut-m.marked-deletions-def mut-m.marked-insertions-def fun-upd-apply)
      apply (drule-tac x=mw-Mutate ref field option in spec)
      apply (drule-tac x=mw-Mutate ref field option in spec)
      apply (clarsimp simp: fun-upd-apply)
      apply (frule spec[where x=x])
      apply (subgoal-tac mut-m.reachable m x s)
       apply (force simp: fun-upd-apply)
      apply (rule reachableI(2))
     apply (force simp: mut-m.tso-store-refs-def)
     apply (rename-tac ref field pl)
     apply (clarsimp simp: mut-m.marked-deletions-def mut-m.marked-insertions-def fun-upd-apply)
      apply (drule-tac x=mw-Mutate-Payload x field pl in spec)
      apply (drule-tac x=mw-Mutate-Payload x field pl in spec)
      apply (clarsimp simp: fun-upd-apply)
      apply (frule spec[where x=x])
      apply (subgoal-tac mut-m.reachable m x s)
       apply (force simp: fun-upd-apply)
      apply (rule reachableI(2))
     apply (force simp: mut-m.tso-store-refs-def)
```

```
    apply (auto simp: fun-upd-apply)
    done
  next
    case (reaches x y)
    from reaches sb have y: mut-m.reachable m y s ∨ mut-m.reachable m' y s
      apply (clarsimp simp: points-to-Mutate mut-m.reachable-snapshot-inv-def in-snapshot-def)
      apply (elim disjE, (force dest!: reachable-points-to mutator-reachable-tso)+)[1]
      done
    moreover
    from y vri have valid-ref y s by auto
    with reaches mi md rsi sb sti y have (black y s ∨ (∃ x. (x grey-protects-white y) s))
      apply (clarsimp simp: mut-m.reachable-snapshot-inv-def in-snapshot-def)
      apply (clarsimp simp: fun-upd-apply)
      apply (drule spec[where x=y])
      apply (clarsimp simp: points-to-Mutate mut-m.marked-insertions-def mut-m.marked-deletions-def)
      apply (drule spec[where x=mw-Mutate r f opt-r'])+
      apply clarsimp
      apply (elim disjE; clarsimp simp: reachable-points-to)
       apply (drule (3) strong-tricolour-invD)
       apply (metis (no-types) grey-protects-whiteI marked-imp-black-or-grey(1))

       apply (metis (no-types) grey-protects-whiteE(2) grey-protects-whiteI marked-imp-black-or-grey(2))

       apply (elim disjE; clarsimp simp: reachable-points-to)
       apply (force simp: black-def)

       apply (elim disjE; clarsimp simp: reachable-points-to)
       apply (force simp: black-def)

       apply (elim disjE; clarsimp simp: reachable-points-to)
       apply (force simp: black-def)

       apply (drule (3) strong-tricolour-invD)
       apply (force simp: black-def)

       apply (elim disjE; clarsimp)
        apply (force simp: black-def fun-upd-apply)
       apply (metis (no-types) grey-protects-whiteE(2) grey-protects-whiteI marked-imp-black-or-grey(2))
       done
    moreover note mi md rsi sb
    ultimately show ?case
      apply (clarsimp simp: mut-m.reachable-snapshot-inv-def in-snapshot-def)
      apply (clarsimp simp: fun-upd-apply)
      done
  qed
  then show in-snapshot y ?s' by blast
qed

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(vcg-jackhammer (no-thin-post-inv), vcg-name-cases)
  case (tso-dequeue-store-buffer s s' p w ws) show ?case
  proof(cases w)
      case (mw-Mark ref field) with tso-dequeue-store-buffer show ?thesis
       by (clarsimp simp: mutator-phase-inv-aux-case
```

```
                marked-deletions-dequeue-Mark marked-insertions-dequeue-Mark reachable-snapshot-inv-dequeue-Mark
                split: hs-phase.splits)
  next case (mw-Mutate ref field opt-r') show ?thesis
    proof(cases ghost-hs-phase (s↓ (mutator m)))
      case hp-IdleInit
      with ‹sys-mem-store-buffers p s↓ = w # ws› spec[OF ‹mutators-phase-inv s↓›, where x=m] mw-Mutate
       show ?thesis by simp
    next case hp-InitMark
      with ‹sys-mem-store-buffers p s↓ = w # ws› spec[OF ‹mutators-phase-inv s↓›, where x=m] mw-Mutate
       show ?thesis by (simp add: marked-insertions-dequeue-Mutate)
    next case hp-Mark with tso-dequeue-store-buffer mw-Mutate show ?thesis
        apply −
        apply (clarsimp simp: mutator-phase-inv-aux-case p-not-sys split: hs-phase.splits)
        apply (erule disjE; clarsimp simp: marked-insertions-dequeue-Mutate)
        apply (rename-tac m')
        apply (frule mut-m.handshake-phase-invD[where m=m])
        apply (rule marked-deletions-dequeue-Mutate, simp-all)
        apply (drule-tac m=m' in mut-m.handshake-phase-invD, clarsimp simp: hp-step-rel-def)
        using hs-phase.distinct(11) hs-phase.distinct(15) hs-type.distinct(1) apply presburger
        done
    next case hp-IdleMarkSweep with tso-dequeue-store-buffer mw-Mutate show ?thesis
        apply −
        apply (clarsimp simp: mutator-phase-inv-aux-case p-not-sys
                    split: hs-phase.splits)
        apply (intro allI conjI impI; erule disjE; clarsimp simp: sys.marked-insertions-dequeue-Mutate)
         apply (rename-tac m')
         apply (rule marked-deletions-dequeue-Mutate, simp-all)[1]
         apply (drule-tac x=m' in spec)
         apply (frule mut-m.handshake-phase-invD[where m=m])
         apply (drule-tac m=m' in mut-m.handshake-phase-invD, clarsimp simp: hp-step-rel-def)
         apply (elim disjE; clarsimp split del: if-split-asm)
         apply (clarsimp simp: fA-rel-inv-def fM-rel-inv-def fA-rel-def fM-rel-def split del: if-split-asm)
         apply (meson black-heap-valid-ref-marked-insertions; fail)
        apply (rename-tac m')
        apply (frule-tac m=m  in mut-m.handshake-phase-invD)
        apply (drule-tac m=m' in mut-m.handshake-phase-invD, clarsimp simp: hp-step-rel-def)
          apply (elim disjE; clarsimp simp: reachable-snapshot-inv-black-heap-no-grey-refs-dequeue-Mutate
reachable-snapshot-inv-dequeue-Mutate)
        apply (clarsimp simp: fA-rel-inv-def fM-rel-inv-def fA-rel-def fM-rel-def)
        apply blast
        done
    qed simp
  next case (mw-Mutate-Payload r f pl) with tso-dequeue-store-buffer show ?thesis
apply (clarsimp simp: mutator-phase-inv-aux-case fun-upd-apply split: hs-phase.splits)
apply (subst reachable-snapshot-fun-upd)
apply (simp-all add: fun-upd-apply)
apply (metis (no-types, lifting) list.set-intros(1) mem-store-action.simps(39) tso-store-inv-def)
done
  next case (mw-fA mark)     with tso-dequeue-store-buffer show ?thesis
    by (clarsimp simp: mutator-phase-inv-aux-case fun-upd-apply split: hs-phase.splits)
  next case (mw-fM mark)     with tso-dequeue-store-buffer show ?thesis
    using mut-m-not-idle-no-fM-writeD by fastforce
  next case (mw-Phase phase) with tso-dequeue-store-buffer show ?thesis
    by (clarsimp simp: mutator-phase-inv-aux-case fun-upd-apply split: hs-phase.splits)
  qed
qed

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 (*vcg-jackhammer*, *vcg-name-cases*)
 case (*store-ins-mo-ptest s s′ obj*) **then show** *?case*
  apply −
  apply (*drule handshake-phase-invD*)
  apply (*drule phase-rel-invD*)
  apply (*clarsimp simp*: *phase-rel-def*)
  apply (*cases sys-ghost-hs-phase s*↓; *simp add*: *hp-step-rel-def*; *elim disjE*; *simp*; *force*)
  done
next case (*store-ins-mo-phase s s′*) **then show** *?case*
  apply −
  apply (*drule handshake-phase-invD*)
  apply (*drule phase-rel-invD*)
  apply (*clarsimp simp*: *phase-rel-def*)
  apply (*case-tac sys-ghost-hs-phase s*↓; *simp add*: *hp-step-rel-def*; *elim disjE*; *simp*; *force*)
  done
next case (*store-del-mo-phase s s′ y*) **then show** *?case*
  apply −
  apply (*drule handshake-phase-invD*)
  apply (*drule phase-rel-invD*)
  apply (*clarsimp simp*: *phase-rel-def*)
  apply (*case-tac sys-ghost-hs-phase s*↓; *simp add*: *hp-step-rel-def*; *elim disjE*; *simp*; *force*)
  done
next case (*deref-del s s′ opt-r′*) **then show** *?case*
  apply −
  apply (*rule obj-at-field-on-heapE*[*OF obj-at-field-on-heap-no-pending-stores*[**where** *m*=*m*]])
  apply *auto*
  done
next case (*hs-get-roots-loop-mo-phase s s′ y*) **then show** *?case*
  apply −
  apply (*drule handshake-phase-invD*)
  apply (*drule phase-rel-invD*)
  apply (*clarsimp simp*: *phase-rel-def hp-step-rel-def*)
  done
qed *fastforce*+

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(*vcg-jackhammer*, *vcg-name-cases*)

case *store-ins-mo-null* **then show** *?case* **by** (*metis reachableI*(*1*) *valid-refs-invD*(*8*))
next case *store-ins-mo-mark* **then show** *?case* **by** (*clarsimp split*: *obj-at-splits*)
next case (*store-ins-mo-ptest s s′ obj*) **then show** *?case* **by** (*simp add*: *valid-W-inv-no-mark-stores-invD filter-empty-conv*) *metis*
next case *store-ins-mo-co-won* **then show** *?case* **by** *metis*
next case *store-ins-mo-mtest* **then show** *?case* **by** *metis*
next case *store-ins-mo-co-ctest0* **then show** *?case* **by** (*metis whiteI*)
next case (*store-ins-mo-co-ctest s s′ obj*) **then show** *?case*
apply (*elim disjE*; *clarsimp split*: *obj-at-splits*)
apply *metis*
done
qed

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(*vcg-jackhammer*, *vcg-name-cases*)
case *store-del-mo-co-ctest0* **then show** *?case* **by** *blast*
next case *store-del-mo-co-ctest* **then show** *?case* **by** (*clarsimp split*: *obj-at-splits*)
next case *store-del-mo-ptest* **then show** *?case* **by** (*auto dest*: *valid-W-inv-no-mark-stores-invD*)
next case *store-del-mo-mark* **then show** *?case* **by** (*clarsimp split*: *obj-at-splits*)
next case *store-del-mo-null* **then show** *?case* **by** (*auto dest*: *valid-refs-invD*)
qed

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(*vcg-jackhammer*, *vcg-name-cases*)
case *hs-get-roots-loop-mo-co-ctest0* **then show** *?case* **by** *blast*
next case *hs-get-roots-loop-mo-co-ctest* **then show** *?case* **by** (*clarsimp split*: *obj-at-splits*)
next case *hs-get-roots-loop-mo-ptest* **then show** *?case* **by** (*auto dest*: *valid-W-inv-no-mark-stores-invD split*: *obj-at-splits*)
next case *hs-get-roots-loop-mo-mark* **then show** *?case* **by** (*clarsimp split*: *obj-at-splits*)
next case *hs-get-roots-loop-mo-null* **then show** *?case* **by** (*auto dest*: *valid-W-inv-no-mark-stores-invD split*: *obj-at-splits*)
qed

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′*
by (*vcg-chainsaw mark-object-invL-def*)

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* ⦈}
by *vcg-jackhammer* (*auto dest*: *valid-W-inv-no-mark-stores-invD split*: *obj-at-splits*)

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(*vcg-jackhammer*, *vcg-name-cases*)
case (*mark-loop-mark-field-done s s'*) **then show** *?case* **by** − (*rule obj-fields-marked-mark-field-done*, *auto*)
next case (*mark-loop-mark-deref s s'*)
then have *grey* (*gc-tmp-ref s*↓) *s*↓ **by** *blast*
with *mark-loop-mark-deref* **show** *?case*
apply (*clarsimp split*: *obj-at-field-on-heap-splits*)
apply (*rule conjI*)
apply (*metis* (*no-types*) *case-optionE obj-at-def valid-W-invE*(*3*))
apply *clarsimp*
apply (*erule valid-refs-invD*; *auto simp*: *obj-at-def ranI reaches-step*(*2*))
done
qed

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(*vcg-chainsaw mut-m.mark-object-invL-def mut-m.tso-lock-invL-def mut-m.mut-store-ins-mark-object-invL-def2*, *vcg-name-cases mutator m*)
case (*store-ins-mo-fM s s' p w ws ref fl*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write valid-W-invD split*: *mem-store-action.splits obj-at-splits if-splits*)
apply (*metis* (*no-types*, *lifting*) *valid-W-invD*(*1*))
done
next case (*store-ins-mo-mtest s s' p w ws y ya*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write valid-W-invD split*: *mem-store-action.splits obj-at-splits if-splits*)
done
next case (*store-ins-mo-phase s s' p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write valid-W-invD split*: *mem-store-action.splits obj-at-splits if-splits*)
done
next case (*store-ins-mo-ptest s s' p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write valid-W-invD split*: *mem-store-action.splits obj-at-splits*)
done
next case (*store-ins-mo-co-lock s s' p w ws y*) **then show** *?case*
apply (*intro conjI impI notI*; *clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write*

split: *mem-store-action.splits obj-at-splits*)
apply *metis*
done
next case (*store-ins-mo-co-cmark s s' w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write split*: *mem-store-action.split obj-at-splits*)
done
next case (*store-ins-mo-co-ctest s s' w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write split*: *mem-store-action.split obj-at-splits*)
done
next case (*store-ins-mo-co-mark s s' w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write split*: *mem-store-action.split obj-at-splits*)
done
next case (*store-ins-mo-co-unlock s s' w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write split*: *mem-store-action.split obj-at-splits*)
done
next case (*store-ins-mo-co-won s s' p w ws y*) **then show** *?case*
apply (*intro conjI impI notI*; *clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write valid-W-invD*(*1*) *split*: *mem-store-action.splits obj-at-splits*)
done
next case (*store-ins-mo-co-W s s' p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write valid-W-invD*(*1*) *split*: *mem-store-action.splits obj-at-splits*)
apply *auto*
done
qed

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(*vcg-chainsaw mut-m.mark-object-invL-def mut-m.tso-lock-invL-def mut-m.mut-store-del-mark-object-invL-def2*, *vcg-name-cases mutator m*)
case (*store-del-mo-fM s s' p w ws y ya*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write valid-W-invD split*: *mem-store-action.splits obj-at-splits if-splits*)
apply (*metis* (*no-types*, *lifting*) *valid-W-invD*(*1*))
done
next case (*store-del-mo-mtest s s' p w ws y ya*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write valid-W-invD split*: *mem-store-action.splits obj-at-splits if-splits*)
done
next case (*store-del-mo-phase s s' p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write valid-W-invD split*: *mem-store-action.splits obj-at-splits*)
done
next case (*store-del-mo-ptest s s' p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write valid-W-invD split*: *mem-store-action.splits obj-at-splits*)

done
next case (*store-del-mo-co-lock s s′ p w ws y*) **then show** *?case*
apply (*intro conjI impI notI*; *clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write split*: *mem-store-action.splits obj-at-splits*)
apply *metis*
done
next case (*store-del-mo-co-cmark s s′ w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write valid-W-invD split*: *mem-store-action.splits obj-at-splits*)
done
next case (*store-del-mo-co-ctest s s′ w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write valid-W-invD split*: *mem-store-action.splits obj-at-splits*)
done
next case (*store-del-mo-co-mark s s′ w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write valid-W-invD split*: *mem-store-action.splits obj-at-splits*)
done
next case (*store-del-mo-co-unlock s s′ w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write valid-W-invD split*: *mem-store-action.splits obj-at-splits*)
apply (*metis* (*mono-tags*, *lifting*) *filter-empty-conv valid-W-invD*(*1*))
done
next case (*store-del-mo-co-won s s′ p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write valid-W-invD split*: *mem-store-action.splits obj-at-splits*)
apply *auto*
done
next case (*store-del-mo-co-W s s′ p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def filter-empty-conv mut-m-not-idle-no-fM-write valid-W-invD split*: *mem-store-action.splits obj-at-splits*)
apply *auto*
done
qed

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(*vcg-chainsaw mut-m.tso-lock-invL-def mut-m.handshake-invL-def mut-m.mut-get-roots-mark-object-invL-def2*, *vcg-name-cases mutator m*)
case (*hs-get-roots-loop-mo-fM s s′ p w ws y ya*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def valid-W-invD split*: *mem-store-action.splits obj-at-splits if-splits*)
apply (*metis* (*no-types*, *lifting*) *valid-W-invD*(*1*))
apply (*force simp*: *fM-rel-inv-def fM-rel-def hp-step-rel-def filter-empty-conv valid-W-invD*)+
done
next case (*hs-get-roots-loop-mo-mtest s s′ p w ws y ya*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def valid-W-invD split*: *mem-store-action.splits obj-at-splits*)
apply (*force simp*: *fM-rel-inv-def fM-rel-def hp-step-rel-def filter-empty-conv valid-W-invD*)+
done
next case (*hs-get-roots-loop-mo-phase s s′ p w ws y*) **then show** *?case*

apply (*clarsimp simp*: *do-store-action-def valid-W-invD split*: *mem-store-action.splits obj-at-splits*)
apply (*force simp*: *fM-rel-inv-def fM-rel-def hp-step-rel-def filter-empty-conv valid-W-invD*)+
done
next case (*hs-get-roots-loop-mo-ptest s s' p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def valid-W-invD split*: *mem-store-action.splits obj-at-splits*)
apply (*force simp*: *fM-rel-inv-def fM-rel-def hp-step-rel-def filter-empty-conv valid-W-invD*)+
done
next case (*hs-get-roots-loop-mo-co-lock s s' p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def valid-W-invD split*: *mem-store-action.splits obj-at-splits*)
apply (*force simp*: *fM-rel-inv-def fM-rel-def hp-step-rel-def filter-empty-conv valid-W-invD*)+
done
next case (*hs-get-roots-loop-mo-co-cmark s s' w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def valid-W-invD split*: *mem-store-action.splits obj-at-splits*)
done
next case (*hs-get-roots-loop-mo-co-ctest s s' w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def valid-W-invD split*: *mem-store-action.splits obj-at-splits*)
done
next case (*hs-get-roots-loop-mo-co-mark s s' w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def valid-W-invD split*: *mem-store-action.splits obj-at-splits*)
done
next case (*hs-get-roots-loop-mo-co-unlock s s' w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def valid-W-invD split*: *mem-store-action.splits obj-at-splits*)
done
next case (*hs-get-roots-loop-mo-co-won s s' p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def valid-W-invD split*: *mem-store-action.splits obj-at-splits*)
apply (*force simp*: *fM-rel-inv-def fM-rel-def hp-step-rel-def filter-empty-conv valid-W-invD*)+
done
next case (*hs-get-roots-loop-mo-co-W s s' p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def valid-W-invD split*: *mem-store-action.splits obj-at-splits*)
apply (*force simp*: *fM-rel-inv-def fM-rel-def hp-step-rel-def filter-empty-conv valid-W-invD*)+
done
qed

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(*vcg-chainsaw gc.gc-mark-mark-object-invL-def2 gc.tso-lock-invL-def gc.phase-invL-def gc.fM-fA-invL-def gc.handshake-invL-def*, *vcg-name-cases gc*)
case (*mark-loop-mo-fM s s' p w ws y ya*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def not-blocked-def fM-rel-def filter-empty-conv p-not-sys valid-W-invD*
split: *mem-store-action.splits if-splits*)
apply (*auto split*: *obj-at-splits*)
done
next case (*mark-loop-mo-mtest s s' p w ws y ya*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def not-blocked-def fM-rel-def filter-empty-conv p-not-sys valid-W-invD*
split: *mem-store-action.splits*)
apply (*auto split*: *obj-at-splits*)
done
next case (*mark-loop-mo-phase s s' p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def not-blocked-def fM-rel-def filter-empty-conv p-not-sys valid-W-invD*
split: *mem-store-action.splits*)
apply (*auto split*: *obj-at-splits*)
done

next case (*mark-loop-mo-ptest s s′ p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def not-blocked-def fM-rel-def filter-empty-conv p-not-sys valid-W-invD*
split: *mem-store-action.splits*)
apply (*auto split*: *obj-at-splits*)
done
next case (*mark-loop-mo-co-lock s s′ p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def not-blocked-def fM-rel-def filter-empty-conv p-not-sys valid-W-invD*
split: *mem-store-action.splits*)
apply (*auto split*: *obj-at-splits*)
done
next case (*mark-loop-mo-co-cmark s s′ p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def not-blocked-def fM-rel-def filter-empty-conv p-not-sys valid-W-invD*
split: *mem-store-action.splits*)
done
next case (*mark-loop-mo-co-ctest s s′ p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def not-blocked-def fM-rel-def filter-empty-conv p-not-sys valid-W-invD*
split: *mem-store-action.splits*)
done
next case (*mark-loop-mo-co-mark s s′ p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def not-blocked-def fM-rel-def filter-empty-conv p-not-sys valid-W-invD*
split: *mem-store-action.splits*)
done
next case (*mark-loop-mo-co-unlock s s′ p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def not-blocked-def fM-rel-def filter-empty-conv p-not-sys*
split: *mem-store-action.splits*)
apply *auto*
done
next case (*mark-loop-mo-co-won s s′ p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def not-blocked-def fM-rel-def filter-empty-conv p-not-sys valid-W-invD*
split: *mem-store-action.splits*)
apply (*auto split*: *obj-at-splits*)
done
next case (*mark-loop-mo-co-W s s′ p w ws y*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def not-blocked-def fM-rel-def filter-empty-conv p-not-sys valid-W-invD*
split: *mem-store-action.splits*)
apply (*auto split*: *obj-at-splits*)
done
qed

end

lemma (**in** *mut-m′*) *mut-get-roots-mark-object-invL*[*intro*]:
⦃ *mut-get-roots.mark-object-invL m* ⦄ *mutator m′*
by *vcg-chainsaw*

lemma (**in** *mut-m′*) *mut-store-ins-mark-object-invL*[*intro*]:
⦃ *mut-store-ins.mark-object-invL m* ⦄ *mutator m′*
by *vcg-chainsaw*

lemma (**in** *mut-m′*) *mut-store-del-mark-object-invL*[*intro*]:
⦃ *mut-store-del.mark-object-invL m* ⦄ *mutator m′*
by *vcg-chainsaw*

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 ⦄
by (*vcg-chainsaw mut-m.handshake-invL-def mut-m.mut-get-roots-mark-object-invL-def2*)

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* |}
apply (*vcg-chainsaw gc.obj-fields-marked-invL-def gc.handshake-invL-def*)

 apply (*clarsimp simp*: *gc.obj-fields-marked-def fun-upd-apply*)
 apply (*rename-tac s s′ ra x*)
 apply (*drule-tac x=x* **in** *spec*)
 apply *clarsimp*
 apply (*erule obj-at-field-on-heapE*)
 apply (*subgoal-tac grey* (*gc-tmp-ref s*↓) *s*↓)
 apply (*drule-tac y=gc-tmp-ref s*↓ **in** *valid-refs-invD*(7), *simp*+)
 apply (*clarsimp simp*: *fun-upd-apply split*: *obj-at-splits*; *fail*)
 apply (*erule greyI*)
 apply (*clarsimp simp*: *fun-upd-apply split*: *obj-at-splits*; *fail*)
apply (*clarsimp simp*: *fun-upd-apply split*: *obj-at-field-on-heap-splits*; *fail*)
apply (*clarsimp simp*: *fun-upd-apply*)
done

lemma (**in** *mut-m*) *gc-mark-mark-object-invL*[*intro*]:
 {| *gc-mark.mark-object-invL* |} *mutator m*
by (*vcg-chainsaw gc.gc-mark-mark-object-invL-def2*)

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
by (*clarsimp simp*: *eq-imp-def inv*)

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* |}
by *vcg-jackhammer* (*fastforce dest*!: *phase-rel-invD simp*: *phase-rel-def*)

lemma (**in** *sys*) *gc-phase-invL*[*intro*]:
 notes *fun-upd-apply*[*simp*]
 notes *if-splits*[*split*]
 shows
 {| *gc.phase-invL* |} *sys*
by (*vcg-chainsaw gc.phase-invL-def*)

lemma (**in** *mut-m*) *gc-phase-invL*[*intro*]:
 {| *gc.phase-invL* |} *mutator m*
by (*vcg-chainsaw gc.phase-invL-def*[*inv*])

lemma (**in** *gc*) *phase-rel-inv*[*intro*]:
 {| *handshake-invL* ∧ *phase-invL* ∧ *LSTP phase-rel-inv* |} *gc* {| *LSTP phase-rel-inv* |}
unfolding *phase-rel-inv-def* **by** (*vcg-jackhammer* (*no-thin-post-inv*); *simp add*: *phase-rel-def*; *blast*)

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(*vcg-jackhammer* (*no-thin-post-inv*), *vcg-name-cases*)
case (*tso-dequeue-store-buffer s s′ p w ws*) **then show** *?case*
apply (*simp add*: *phase-rel-def p-not-sys split*: *if-splits*)
apply (*elim disjE*; *auto split*: *if-splits*)
done
qed

lemma (**in** *mut-m*) *phase-rel-inv*[*intro*]:
⦃ *handshake-invL* ∧ *LSTP* (*handshake-phase-inv* ∧ *phase-rel-inv*) ⦄
mutator m
⦃ *LSTP phase-rel-inv* ⦄
unfolding *phase-rel-inv-def*
proof (*vcg-jackhammer* (*no-thin-post-inv*), *vcg-name-cases*)
case (*hs-noop-done s s′*) **then show** *?case*
by (*auto dest*!: *handshake-phase-invD*
simp: *handshake-phase-rel-def phase-rel-def hp-step-rel-def*
split: *hs-phase.splits*)
next case (*hs-get-roots-done s s′*) **then show** *?case*
by (*auto dest*!: *handshake-phase-invD*
simp: *handshake-phase-rel-def phase-rel-def hp-step-rel-def*
split: *hs-phase.splits*)
next case (*hs-get-work-done s s′*) **then show** *?case*
by (*auto dest*!: *handshake-phase-invD*
simp: *handshake-phase-rel-def phase-rel-def hp-step-rel-def*
split: *hs-phase.splits*)
qed

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
by (*simp add*: *gc.handshake-invL-def eq-imp-def*)

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*
by (*vcg-chainsaw gc.handshake-invL-def*)

lemma (**in** *sys*) *handshake-phase-inv*[*intro*]:
⦃ *LSTP handshake-phase-inv* ⦄ *sys*
unfolding *handshake-phase-inv-def* **by** (*vcg-jackhammer* (*no-thin-post-inv*))

lemma (**in** *gc*) *handshake-invL*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
⦃ *handshake-invL* ⦄ *gc*
by *vcg-jackhammer fastforce*+

lemma (**in** *gc*) *handshake-phase-inv*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
⦃ *handshake-invL* ∧ *LSTP handshake-phase-inv* ⦄ *gc* ⦃ *LSTP handshake-phase-inv* ⦄
unfolding *handshake-phase-inv-def* **by** (*vcg-jackhammer* (*no-thin-post-inv*)) (*auto simp*: *handshake-phase-inv-def hp-step-rel-def*)

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
unfolding *eq-imp-def handshake-invL-def* **by** *simp*

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*
by *vcg-jackhammer*

lemma (**in** *mut-m′*) *handshake-invL*[*intro*]:
⦃ *handshake-invL* ⦄ *mutator m′*
by *vcg-chainsaw*

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* ⦄
by (*vcg-chainsaw mut-m.handshake-invL-def*)

lemma (**in** *sys*) *mut-handshake-invL*[*intro*]:
notes *if-splits*[*split*]
notes *fun-upd-apply*[*simp*]
shows
⦃ *mut-m.handshake-invL m* ⦄ *sys*
by (*vcg-chainsaw mut-m.handshake-invL-def*)

lemma (**in** *mut-m*) *gc-handshake-invL*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
⦃ *handshake-invL* ∧ *gc.handshake-invL* ⦄ *mutator m* ⦃ *gc.handshake-invL* ⦄
by (*vcg-chainsaw gc.handshake-invL-def*)

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* ⦄
unfolding *handshake-phase-inv-def* **by** (*vcg-jackhammer* (*no-thin-post-inv*)) (*auto simp*: *hp-step-rel-def*)

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
by (*simp add*: *gc.fM-fA-invL-def eq-imp-def*)

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*
by *vcg-jackhammer*

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* ‖}
by (*vcg-jackhammer* (*no-thin-post-inv*); *simp add*: *fM-rel-inv-def fM-rel-def*)

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* ‖}
by (*vcg-jackhammer* (*no-thin-post-inv*); *simp add*: *fA-rel-inv-def*; *auto simp*: *fA-rel-def*)

end

context *mut-m*
begin

lemma *gc-fM-fA-invL*[*intro*]:
{‖ *gc.fM-fA-invL* ‖} *mutator m*
by (*vcg-chainsaw gc.fM-fA-invL-def*)

lemma *fM-rel-inv*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
{‖ *LSTP fM-rel-inv* ‖} *mutator m*
unfolding *fM-rel-inv-def* **by** (*vcg-jackhammer* (*no-thin-post-inv*); *simp add*: *fM-rel-def*; *elim disjE*; *auto split*: *if-splits*)

lemma *fA-rel-inv*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
{‖ *LSTP fA-rel-inv* ‖} *mutator m*
unfolding *fA-rel-inv-def* **by** (*vcg-jackhammer* (*no-thin-post-inv*); *simp add*: *fA-rel-def*; *elim disjE*; *auto split*: *if-splits*)

end

context *gc*
begin

lemma *fA-neq-locs-diff-fA-tso-empty-locs*:
fA-neq-locs − *fA-tso-empty-locs* = {}
apply (*simp add*: *fA-neq-locs-def fA-tso-empty-locs-def locset-cache*)

apply (*simp add*: *loc-defs*)
done

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* ⦄
apply(*vcg-chainsaw* (*no-thin*) *gc.fM-fA-invL-def*
; (*simp add*: *p-not-sys*)*?*; (*erule disjE*)*?*; *clarsimp split*: *if-splits*)
proof(*vcg-name-cases sys gc*)
case (*tso-dequeue-store-buffer-mark-noop-mfence s s′ ws*) **then show** *?case* **by** (*clarsimp simp*: *fA-rel-inv-def fA-rel-def*)
next case (*tso-dequeue-store-buffer-fA-neq-locs s s′ ws*) **then show** *?case*
apply (*clarsimp simp*: *fA-rel-inv-def fA-rel-def fM-rel-inv-def fM-rel-def*)
apply (*drule* (*1*) *atS-dests*(*3*), *fastforce simp*: *atS-simps gc.fA-neq-locs-diff-fA-tso-empty-locs*)
done
next case (*tso-dequeue-store-buffer-fA-eq-locs s s′ ws*) **then show** *?case* **by** (*clarsimp simp*: *fA-rel-inv-def fA-rel-def*)
next case (*tso-dequeue-store-buffer-idle-flip-noop-mfence s s′ ws*) **then show** *?case* **by** (*clarsimp simp*: *fM-rel-inv-def fM-rel-def*)
next case (*tso-dequeue-store-buffer-fM-eq-locs s s′ ws*) **then show** *?case* **by** (*clarsimp simp*: *fM-rel-inv-def fM-rel-def*)
qed

lemma *fM-rel-inv*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
⦃ *LSTP* (*fM-rel-inv* ∧ *tso-store-inv*) ⦄ *sys* ⦃ *LSTP fM-rel-inv* ⦄
apply (*vcg-jackhammer* (*no-thin-post-inv*))
apply (*clarsimp simp*: *do-store-action-def fM-rel-inv-def fM-rel-def p-not-sys*
split: *mem-store-action.splits*)
apply (*intro allI conjI impI*; *clarsimp*)
done

lemma *fA-rel-inv*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
⦃ *LSTP* (*fA-rel-inv* ∧ *tso-store-inv*) ⦄ *sys* ⦃ *LSTP fA-rel-inv* ⦄
apply (*vcg-jackhammer* (*no-thin-post-inv*))
apply (*clarsimp simp*: *do-store-action-def fA-rel-inv-def fA-rel-def p-not-sys*
split: *mem-store-action.splits*)
apply (*intro allI conjI impI*; *clarsimp*)
done

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((*vcg-jackhammer* (*no-thin-post-inv*)
; *clarsimp simp*: *fA-rel-inv-def fM-rel-inv-def sys-phase-inv-aux-case heap-colours-colours*
split: *hs-phase.splits if-splits*)
, *vcg-name-cases*)
case (*alloc s s′ rb*) **then show** *?case*
by (*clarsimp simp*: *fA-rel-def fM-rel-def no-black-refs-def*
dest!: *handshake-phase-invD phase-rel-invD*
split: *hs-phase.splits*)
next case (*store-ins-mo-co-mark0 s s′ y*) **then show** *?case*
by (*fastforce simp*: *fA-rel-def fM-rel-def hp-step-rel-def*
dest!: *handshake-phase-invD phase-rel-invD*)
next case (*store-ins-mo-co-mark s s′ y*) **then show** *?case*
apply −
apply (*drule spec*[**where** *x*=*m*])
apply (*rule conjI*)
apply (*clarsimp simp*: *hp-step-rel-def phase-rel-def conj-disj-distribR*[*symmetric*]
dest!: *handshake-phase-invD phase-rel-invD*)
apply (*elim disjE*, *simp-all add*: *no-grey-refs-not-rootD*; *fail*)
apply (*clarsimp simp*: *hp-step-rel-def phase-rel-def*
dest!: *handshake-phase-invD phase-rel-invD*)
apply (*elim disjE*, *simp-all add*: *no-grey-refs-not-rootD*)[*1*]
apply *clarsimp*
apply (*elim disjE*, *simp-all add*: *no-grey-refs-not-rootD filter-empty-conv*)[*1*]
apply *fastforce*
done
next case (*store-del-mo-co-mark0 s s′ y*) **then show** *?case*
apply (*clarsimp simp*: *hp-step-rel-def dest*!: *handshake-phase-invD phase-rel-invD*)
apply (*metis* (*no-types*, *lifting*) *mut-m.no-grey-refs-not-rootD mutator-phase-inv-aux.simps*(*5*))
done
next case (*store-del-mo-co-mark s s′ y*) **then show** *?case*
apply −
apply (*drule spec*[**where** *x*=*m*])
apply (*rule conjI*)
apply (*clarsimp simp*: *hp-step-rel-def phase-rel-def conj-disj-distribR*[*symmetric*] *no-grey-refs-not-rootD*
dest!: *handshake-phase-invD phase-rel-invD*; *fail*)
apply (*clarsimp simp*: *hp-step-rel-def phase-rel-def*
dest!: *handshake-phase-invD phase-rel-invD*)
apply (*elim disjE*, *simp-all add*: *no-grey-refs-not-rootD*)
apply *clarsimp*
apply (*elim disjE*, *simp-all add*: *no-grey-refs-not-rootD filter-empty-conv*)
apply *fastforce*
done
next case (*hs-get-roots-done s s′*) **then show** *?case*
apply (*clarsimp simp*: *hp-step-rel-def phase-rel-def filter-empty-conv*
dest!: *handshake-phase-invD phase-rel-invD*)
apply *auto*
done
next case (*hs-get-roots-loop-mo-co-mark s s′ y*) **then show** *?case*

apply −
apply (*drule spec*[**where** *x*=*m*])
apply (*rule conjI*)
apply (*clarsimp simp*: *hp-step-rel-def phase-rel-def conj-disj-distribR*[*symmetric*]
dest!: *handshake-phase-invD phase-rel-invD*; *fail*)
apply (*clarsimp simp*: *hp-step-rel-def phase-rel-def*
dest!: *handshake-phase-invD phase-rel-invD*)
apply (*elim disjE*, *simp-all add*: *no-grey-refs-not-rootD*)
apply *clarsimp*
apply (*elim disjE*, *simp-all add*: *no-grey-refs-not-rootD filter-empty-conv*)[*1*]
apply *fastforce*
done
next case (*hs-get-work-done s s'*) **then show** *?case*
apply (*clarsimp simp*: *hp-step-rel-def phase-rel-def filter-empty-conv*
dest!: *handshake-phase-invD phase-rel-invD*)
apply *auto*
done
qed (*clarsimp simp*: *hp-step-rel-def dest*!: *handshake-phase-invD phase-rel-invD*)+

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*
∧ *LSTP* (*phase-rel-inv* ∧ *sys-phase-inv* ∧ *valid-W-inv* ∧ *tso-store-inv*) ⦈}
gc
{⦇ *LSTP sys-phase-inv* ⦈}
proof(*vcg-jackhammer* (*no-thin-post-inv*), *vcg-name-cases*)
case (*mark-loop-get-work-load-W s s'*) **then show** *?case* **by** (*fastforce dest*!: *phase-rel-invD simp*: *phase-rel-def no-grey-refsD filter-empty-conv*)
next case (*mark-loop-blacken s s'*) **then show** *?case* **by** (*meson no-grey-refsD*(*1*))
next case (*mark-loop-mo-co-W s s'*) **then show** *?case* **by** (*meson no-grey-refsD*(*1*))
next case (*mark-loop-mo-co-mark s s'*) **then show** *?case* **by** (*meson no-grey-refsD*(*1*))
next case (*mark-loop-get-roots-load-W s s'*) **then show** *?case* **by** (*fastforce dest*!: *phase-rel-invD simp*: *phase-rel-def no-grey-refsD filter-empty-conv*)
next case (*mark-loop-get-roots-init-type s s'*) **then show** *?case* **by** (*fastforce dest*!: *phase-rel-invD simp*: *phase-rel-def no-grey-refsD filter-empty-conv*)
next case (*idle-noop-init-type s s'*) **then show** *?case* **using** *black-heap-no-greys* **by** *blast*
qed

lemma *no-grey-refs-no-marks*[*simp*]:
⟦ *no-grey-refs s*; *valid-W-inv s* ⟧ ⟹ ¬*sys-mem-store-buffers p s* = *mw-Mark r fl* # *ws*
unfolding *no-grey-refs-def* **by** (*metis greyI*(*1*) *list.set-intros*(*1*) *valid-W-invE*(*5*))

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* ⟧
⟹ *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*)|)))
unfolding *black-heap-def* **by** (*metis colours-distinct*(*4*) *valid-W-invD*(*1*) *white-valid-ref*)

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*)|))))
unfolding *black-heap-def white-heap-def black-def white-def* **by** *clarsimp*

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*)|)))
unfolding *black-heap-def white-heap-def no-grey-refs-def* **by** *auto*

lemma *sys-phase-inv*[*intro*]:
notes *if-split-asm*[*split del*]
notes *fun-upd-apply*[*simp*]
shows
{| *LSTP* (*fA-rel-inv* ∧ *fM-rel-inv* ∧ *handshake-phase-inv* ∧ *mutators-phase-inv* ∧ *phase-rel-inv* ∧ *sys-phase-inv* ∧ *tso-store-inv* ∧ *valid-W-inv*) |}
sys
{| *LSTP sys-phase-inv* |}
proof(*vcg-jackhammer* (*no-thin-post-inv*)
, *clarsimp simp*: *fA-rel-inv-def fM-rel-inv-def p-not-sys*
, *vcg-name-cases*)
case (*tso-dequeue-store-buffer s s′ p w ws*) **then show** *?case*
apply (*clarsimp simp*: *do-store-action-def sys-phase-inv-aux-case*
split: *mem-store-action.splits hs-phase.splits if-splits*)
apply (*clarsimp simp*: *fA-rel-def fM-rel-def*; *erule disjE*; *clarsimp simp*: *fA-rel-def fM-rel-def*)+
apply (*metis* (*mono-tags*, *lifting*) *filter.simps*(*2*) *list.discI tso-store-invD*(*4*))
apply *auto*
done
qed

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′*])|)))
by (*auto simp*: *marked-insertions-def*
split: *mem-store-action.splits option.splits*)

lemma *marked-insertions-alloc*[*simp*]:
⟦ *heap* (*s sys*) *r′* = *None*; *valid-refs-inv s* ⟧
$\Longrightarrow$ *marked-insertions* (*s*(*mutator m′* := *s* (*mutator m′*)(|*roots* := *roots′*|), *sys* := *s sys*(|*heap* := (*sys-heap s*)(*r′* ↦ *obj′*)|)))
⟷ *marked-insertions s*
apply (*clarsimp simp*: *marked-insertions-def split*: *mem-store-action.splits option.splits*)
apply (*rule iffI*)
apply *clarsimp*
apply (*rename-tac ref field x*)

apply (*drule-tac x=ref* **in** *spec*, *drule-tac x=field* **in** *spec*, *drule-tac x=x* **in** *spec*, *clarsimp*)
apply (*drule valid-refs-invD(6)*[**where** *x=r′* **and** *y=r′*], *simp-all*)
done

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′*])|)))
by (*auto simp*: *marked-deletions-def*
split: *mem-store-action.splits option.splits*)

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′*)|)))
apply (*clarsimp simp*: *marked-deletions-def split*: *mem-store-action.splits*)
apply (*rename-tac ref field option*)
apply (*drule-tac x=mw-Mutate ref field option* **in** *spec*)
apply *clarsimp*
apply (*case-tac ref* = *r′*)
apply (*auto simp*: *obj-at-field-on-heap-def split*: *option.splits*)
done

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
apply (*clarsimp simp*: *eq-imp-def inv*)
apply (*rename-tac s s′*)
apply (*subgoal-tac* ∀ *r*. *valid-ref r s*↓ ⟷ *valid-ref r s′*↓)
apply (*subgoal-tac* ∀ *P r*. *obj-at* (λ*obj*. *P* (*obj-mark obj*)) *r s*↓ ⟷ *obj-at* (λ*obj*. *P* (*obj-mark obj*)) *r s′*↓)
apply (*frule-tac x=*λ*mark*. *Some mark* = *gc-mark s′*↓ **in** *spec*)
apply (*frule-tac x=*λ*mark*. *mark* = *sys-fM s′*↓ **in** *spec*)
apply *clarsimp*
apply (*clarsimp simp*: *fun-eq-iff split*: *obj-at-splits*)
apply (*rename-tac r*)
apply ((*drule-tac x=r* **in** *spec*)+, *auto*)[*1*]
apply (*clarsimp simp*: *fun-eq-iff split*: *obj-at-splits*)
apply (*rename-tac r*)
apply (*drule-tac x=r* **in** *spec*, *auto*)[*1*]
apply (*metis map-option-eq-Some*)+
done

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(vcg-jackhammer, vcg-name-cases)
    case sweep-loop-ref-done  then show ?case by blast
next case sweep-loop-check     then show ?case
     apply (clarsimp split: obj-at-splits)
     apply (metis (no-types, lifting) option.collapse option.inject)
     done
next case sweep-loop-load-mark then show ?case by (clarsimp split: obj-at-splits)
qed

context gc
begin

lemma sweep-loop-locs-subseteq-sweep-locs:
  sweep-loop-locs ⊆ sweep-locs
by (auto simp: sweep-loop-locs-def sweep-locs-def intro: append-prefixD)

lemma sweep-locs-subseteq-fM-tso-empty-locs:
  sweep-locs ⊆ fM-tso-empty-locs
by (auto simp: sweep-locs-def fM-tso-empty-locs-def loc-defs)

lemma sweep-loop-locs-fM-eq-locs:
  sweep-loop-locs ⊆ fM-eq-locs
by (auto simp: sweep-loop-locs-def fM-eq-locs-def sweep-locs-def loc-defs)

lemma sweep-loop-locs-fA-eq-locs:
  sweep-loop-locs ⊆ fA-eq-locs
apply (simp add: sweep-loop-locs-def fA-eq-locs-def sweep-locs-def)
apply (intro subset-insertI2)
apply (auto intro: append-prefixD)
done

lemma black-heap-locs-subseteq-fM-tso-empty-locs:
  black-heap-locs ⊆ fM-tso-empty-locs
by (auto simp: black-heap-locs-def fM-tso-empty-locs-def loc-defs)

lemma black-heap-locs-fM-eq-locs:
  black-heap-locs ⊆ fM-eq-locs
by (simp add: black-heap-locs-def fM-eq-locs-def loc-defs)

lemma black-heap-locs-fA-eq-locs:
  black-heap-locs ⊆ fA-eq-locs
by (simp add: black-heap-locs-def fA-eq-locs-def sweep-locs-def loc-defs)

lemma fM-fA-invL-tso-emptyD:
  ⟦ atS gc ls s; fM-fA-invL s; ls ⊆ fM-tso-empty-locs ⟧ ⟹ tso-pending-fM gc s↓ = []
by (auto simp: fM-fA-invL-def dest: atS-mono)

lemma gc-sweep-loop-invL-locsE[rule-format]:
  (atS gc (sweep-locs ∪ black-heap-locs) s ⟶ False) ⟹ gc.sweep-loop-invL s
apply (simp add: gc.sweep-loop-invL-def atS-un)
apply (auto simp: locset-cache atS-simps dest: atS-mono)
 apply (simp add: atS-mono gc.sweep-loop-locs-subseteq-sweep-locs; fail)
apply (clarsimp simp: atS-def)
apply (rename-tac x)
apply (drule-tac x=x in bspec)
```

apply (*auto simp*: *sweep-locs-def sweep-loop-not-choose-ref-locs-def intro*: *append-prefixD*)
done

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(*vcg-jackhammer* (*keep-locs*) (*no-thin-post-inv*), *vcg-name-cases*)
 case (*tso-dequeue-store-buffer s s′ p w ws*) **then show** *?case*
 proof(*cases w*)
 case (*mw-Mark r fl*) **with** *tso-dequeue-store-buffer* **show** *?thesis*
 apply −
 apply (*rule gc.gc-sweep-loop-invL-locsE*)
 apply (*simp only*: *gc.gc-W-empty-invL-def gc.no-grey-refs-locs-def cong del*: *atS-state-weak-cong*)
 apply (*clarsimp simp*: *atS-un*)
 apply (*thin-tac AT - = -*)
 apply (*thin-tac at - - - ⟶ -*)+
 apply (*metis* (*mono-tags*, *lifting*) *filter.simps*(*2*) *loc-mem-tac-simps*(*4*) *no-grey-refs-no-pending-marks*)
 done
 next case (*mw-Mutate r f opt-r′*) **with** *tso-dequeue-store-buffer* **show** *?thesis* **by** *clarsimp* (*erule gc-sweep-loop-invL-ni*
simp add: *fun-eq-iff fun-upd-apply*)
 next case (*mw-Mutate-Payload r f pl*) **with** *tso-dequeue-store-buffer* **show** *?thesis* **by** *clarsimp* (*erule gc-sweep-loop-inv*
simp add: *fun-eq-iff fun-upd-apply*)
 next case (*mw-fA fl*) **with** *tso-dequeue-store-buffer* **show** *?thesis* **by** − (*erule gc-sweep-loop-invL-niE*; *simp*
add: *fun-eq-iff*)
 next case (*mw-fM fl*) **with** *tso-dequeue-store-buffer* **show** *?thesis*
 apply −
 apply (*rule gc.gc-sweep-loop-invL-locsE*)
 apply (*case-tac p*; *clarsimp*)
 apply (*drule* (*1*) *gc.fM-fA-invL-tso-emptyD*)
 apply *simp-all*
 using *gc.black-heap-locs-subseteq-fM-tso-empty-locs gc.sweep-locs-subseteq-fM-tso-empty-locs* **apply** *blast*
 done
 next case (*mw-Phase ph*) **with** *tso-dequeue-store-buffer* **show** *?thesis* **by** − (*erule gc-sweep-loop-invL-niE*;
simp add: *fun-eq-iff*)
 qed
qed

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(*vcg-chainsaw* (*no-thin*) *gc.fM-fA-invL-def gc.sweep-loop-invL-def gc.handshake-invL-def*, *vcg-name-cases*
gc)
 case (*sweep-loop-locs s s′ rb*) **then show** *?case* **by** (*metis* (*no-types*, *lifting*) *atS-mono gc.sweep-loop-locs-fA-eq-locs*
gc.sweep-loop-locs-fM-eq-locs)
next case (*black-heap-locs s s′ rb*) **then show** *?case* **by** (*metis* (*no-types*, *lifting*) *atS-mono gc.black-heap-locs-fA-eq-locs*
gc.black-heap-locs-fM-eq-locs)
qed

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} |)))
unfolding in-snapshot-def reachable-snapshot-inv-def by (auto simp: fun-upd-apply)

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(rule, clarsimp)
  fix r assume reachable r s
  then show in-snapshot r ?s'
  proof (induct rule: reachable-induct)
    case (root x) with m show ?case
      apply (clarsimp simp: in-snapshot-def)
      apply (auto dest: marked-imp-black-or-grey)
      done
  next
    case (ghost-honorary-root x) with ghr show ?case by simp
  next
    case (tso-root x) with t show ?case
      apply (clarsimp simp: filter-empty-conv tso-store-refs-def)
      apply (rename-tac w; case-tac w; fastforce)
      done
  next
    case (reaches x y)
    from reaches vri have valid-ref x s valid-ref y s
      using reachable-points-to by fastforce+
    with reaches sti vri show ?case
      apply (clarsimp simp: in-snapshot-def)
      apply (elim disjE)
       apply (clarsimp simp: strong-tricolour-inv-def)
       apply (drule spec[where x=x])
       apply clarsimp
       apply (auto dest!: marked-imp-black-or-grey)[1]
      apply (cases white y s)
      apply (auto dest: grey-protects-whiteE
                  dest!: marked-imp-black-or-grey)
       done
    qed
qed

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)|)))
by (simp add: reachable-snapshot-inv-def in-snapshot-def grey-protects-white-def)
```

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′*|)))
unfolding *reachable-snapshot-inv-def in-snapshot-def grey-protects-white-def* **by** (*clarsimp simp*: *fun-upd-apply*)

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(*vcg-jackhammer* (*no-thin-post-inv*)
, *simp-all add*: *mutator-phase-inv-aux-case split*: *hs-phase.splits*
, *vcg-name-cases*)
case *alloc* **then show** *?case*
apply (*drule-tac x*=*m* **in** *spec*)
apply (*drule handshake-phase-invD*)
apply (*clarsimp simp*: *fA-rel-inv-def fM-rel-inv-def fM-rel-def hp-step-rel-def split*: *if-split-asm*)
apply (*intro conjI impI*; *simp*)
apply (*elim disjE*; *force simp*: *fA-rel-def*)
apply (*rule reachable-snapshot-inv-alloc*, *simp-all*)
apply (*elim disjE*; *force simp*: *fA-rel-def*)
done
next case (*store-ins s s′*) **then show** *?case*
apply (*drule-tac x*=*m* **in** *spec*)
apply (*drule handshake-phase-invD*)
apply (*intro conjI impI*; *clarsimp*)
apply (*rule marked-deletions-store-ins*, *assumption*)
apply (*cases* (∀ *opt-r′*. *mw-Mutate* (*mut-tmp-ref s*↓) (*mut-field s*↓) *opt-r′* ∉ *set* (*sys-mem-store-buffers* (*mutator m*) *s*↓)); *clarsimp*)
apply (*force simp*: *marked-deletions-def*)
apply (*erule marked-insertions-store-ins*)
apply (*drule phase-rel-invD*)
apply (*clarsimp simp*: *phase-rel-def hp-step-rel-def*; *elim disjE*; *fastforce dest*: *reachable-blackD elim*: *blackD*; *fail*)
apply (*rule marked-deletions-store-ins*; *clarsimp*)
apply (*erule disjE*; *clarsimp*)
apply (*drule phase-rel-invD*)
apply (*clarsimp simp*: *phase-rel-def*)
apply (*elim disjE*; *clarsimp*)
apply (*fastforce simp*: *hp-step-rel-def*)
apply (*clarsimp simp*: *hp-step-rel-def*)
apply (*case-tac sys-ghost-hs-phase s*↓; *clarsimp*)
apply (*clarsimp simp*: *obj-at-field-on-heap-def split*: *option.splits*)
apply (*rule conjI*, *fast*, *clarsimp*)
apply (*frule-tac r*=*x2a* **in** *blackD*(*1*)[*OF reachable-blackD*], *simp-all*)[*1*]
apply (*rule-tac x*=*mut-tmp-ref s*↓ **in** *reachable-points-to*; *auto simp*: *ran-def split*: *obj-at-splits*; *fail*)

```
    apply (clarsimp simp: obj-at-field-on-heap-def split: option.splits)
    apply (rule conjI, fast, clarsimp)
    apply (frule-tac r=x2a in blackD(1)[OF reachable-blackD], simp-all)[1]
    apply (rule-tac x=mut-tmp-ref s↓ in reachable-points-to; auto simp: ran-def split: obj-at-splits; fail)
   apply (force simp: marked-deletions-def)
   done
next case (hs-noop-done s s') then show ?case
   apply −
   apply (drule-tac x=m in spec)
   apply (drule handshake-phase-invD)
   apply (simp add: fA-rel-def fM-rel-def hp-step-rel-def)
   apply (cases mut-ghost-hs-phase s↓)
   apply auto
   done
next case (hs-get-roots-done s s') then show ?case
   apply −
   apply (drule-tac x=m in spec)
   apply (drule handshake-phase-invD)
   apply (force simp: hp-step-rel-def reachable-snapshot-inv-hs-get-roots-done)
   done
next case (hs-get-work-done s s') then show ?case
   apply (drule-tac x=m in spec)
   apply (drule handshake-phase-invD)
   apply (force simp add: hp-step-rel-def reachable-snapshot-inv-hs-get-work-done)
   done
qed

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( vcg-jackhammer (no-thin-post-inv)
    , simp-all add: mutator-phase-inv-aux-case split: hs-phase.splits
    , vcg-name-cases)
     case (alloc s s' rb) then show ?case
      apply −
      apply (clarsimp simp: fA-rel-inv-def fM-rel-inv-def white-def)
      apply (drule spec[where x=m])
      apply (intro conjI impI; clarsimp)
       apply (clarsimp simp: hp-step-rel-def simp: fA-rel-def fM-rel-def dest!: handshake-phase-invD)
       apply (elim disjE, auto; fail)
      apply (rule reachable-snapshot-inv-alloc, simp-all)
      apply (clarsimp simp: hp-step-rel-def simp: fA-rel-def fM-rel-def dest!: handshake-phase-invD)
      apply (cases sys-ghost-hs-phase s↓; clarsimp; blast)
      done
next case (hs-get-roots-done s s') then show ?case
      apply −
```

apply (*drule spec*[**where** *x*=*m*])
apply (*simp add*: *no-black-refs-def reachable-snapshot-inv-def in-snapshot-def*)
done
next case (*hs-get-work-done s s'*) **then show** *?case*
apply −
apply (*drule spec*[**where** *x*=*m*])
apply (*clarsimp simp*: *no-black-refs-def reachable-snapshot-inv-def in-snapshot-def grey-protects-white-def*)
done
qed

lemma *no-black-refs-sweep-loop-free*[*simp*]:
no-black-refs s ⟹ *no-black-refs* (*s*(*sys* := *s sys*(|*heap* := (*sys-heap s*)(*gc-tmp-ref s* := *None*)|)))
unfolding *no-black-refs-def* **by** *simp*

lemma *no-black-refs-load-W*[*simp*]:
⟦ *no-black-refs s*; *gc-W s* = {} ⟧
⟹ *no-black-refs* (*s*(*gc* := *s gc*(| *W* := *sys-W s*|), *sys* := *s sys*(| *W* := {}|)))
unfolding *no-black-refs-def* **by** *simp*

lemma *marked-insertions-sweep-loop-free*[*simp*]:
⟦ *mut-m.marked-insertions m s*; *white r s* ⟧
⟹ *mut-m.marked-insertions m* (*s*(*sys* := (*s sys*)(|*heap* := (*heap* (*s sys*))(*r* := *None*)|)))
unfolding *mut-m.marked-insertions-def* **by** (*fastforce simp*: *fun-upd-apply split*: *mem-store-action.splits obj-at-splits option.splits*)

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* ⟧
⟹ *mut-m.marked-deletions m* (*s*(*sys* := *s sys*(|*heap* := (*sys-heap s*)(*r* := *None*)|)))
unfolding *mut-m.marked-deletions-def*
apply (*clarsimp split*: *mem-store-action.splits*)
apply (*rename-tac ref field option*)
apply (*drule-tac x*=*mw-Mutate ref field option* **in** *spec*)
apply (*clarsimp simp*: *obj-at-field-on-heap-def split*: *option.splits*)
apply (*rule conjI*)
apply (*clarsimp simp*: *mut-m.reachable-snapshot-inv-def*)
apply (*drule spec*[**where** *x*=*r*], *clarsimp simp*: *in-snapshot-def*)
apply (*drule mp*, *auto simp*: *mut-m.reachable-def mut-m.tso-store-refs-def split*: *mem-store-action.splits*)[*1*]
apply (*drule grey-protects-whiteD*)
apply (*clarsimp simp*: *no-grey-refs-def*)
apply (*clarsimp*; *fail*)
apply (*rule conjI*; *clarsimp*)
apply (*rule conjI*)
apply (*clarsimp simp*: *mut-m.reachable-snapshot-inv-def*)
apply (*drule spec*[**where** *x*=*r*], *clarsimp simp*: *in-snapshot-def*)
apply (*drule mp*, *auto simp*: *mut-m.reachable-def mut-m.tso-store-refs-def split*: *mem-store-action.splits*)[*1*]
apply (*drule grey-protects-whiteD*)
apply (*clarsimp simp*: *no-grey-refs-def*)
unfolding *white-def* **apply** (*clarsimp split*: *obj-at-splits*)
done

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* ⟧ ⟹ *False*
by (*simp add*: *obj-fields-marked-def obj-at-field-on-heap-def ran-def white-def split*: *option.splits obj-at-splits*)

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* ⟧ ⟹ *w = gc-tmp-ref s*
by (*metis* (*mono-tags*, *lifting*) *converse-rtranclpE gc.obj-fields-marked-inv-blacken has-white-path-to-def*)

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(*vcg-jackhammer* (*no-thin-post-inv*)
, *simp-all add*: *mutator-phase-inv-aux-case white-def split*: *hs-phase.splits*
, *vcg-name-cases*)
case (*sweep-loop-free s s'*) **then show** *?case*
apply (*intro allI conjI impI*)
apply (*drule mut-m.handshake-phase-invD*[**where** *m=m*], *clarsimp simp*: *hp-step-rel-def*; *fail*)
apply (*rule mut-m.reachable-snapshot-inv-sweep-loop-free*, *simp-all add*: *white-def*)
done
next case (*mark-loop-get-work-load-W s s'*) **then show** *?case*
apply *clarsimp*
apply (*drule spec*[**where** *x=m*])
apply (*clarsimp simp*: *mut-m.reachable-snapshot-inv-def in-snapshot-def grey-protects-white-def*)
done
next case (*mark-loop-blacken s s'*) **then show** *?case*
apply −
apply (*drule spec*[**where** *x=m*])
apply *clarsimp*
apply (*intro allI conjI impI*; *clarsimp*)
apply (*drule mut-m.handshake-phase-invD*[**where** *m=m*], *clarsimp simp*: *hp-step-rel-def*)
apply (*clarsimp simp*: *mut-m.reachable-snapshot-inv-def in-snapshot-def grey-protects-white-def*)
apply (*metis* (*no-types*, *opaque-lifting*) *obj-fields-marked-inv-has-white-path-to-blacken*)
done
next case (*mark-loop-mo-co-mark s s' y*) **then show** *?case* **by** (*clarsimp simp*: *handshake-in-syncD mut-m.reachable-sn*
next case (*mark-loop-get-roots-load-W s s'*) **then show** *?case*
apply *clarsimp*
apply (*drule spec*[**where** *x=m*])
apply (*clarsimp simp*: *mut-m.reachable-snapshot-inv-def in-snapshot-def grey-protects-white-def*)
done
qed

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* ⦈}
unfolding *strong-tricolour-inv-def*
proof(*vcg-jackhammer* (*no-thin-post-inv*), *vcg-name-cases*)
case (*mark-loop-blacken s s' x xa*) **then show** *?case* **by** (*fastforce elim*!: *obj-fields-marked-inv-blacken*)

qed

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* ‖}
unfolding *strong-tricolour-inv-def*
proof(*vcg-jackhammer* (*no-thin-post-inv*), *vcg-name-cases*)
case (*alloc s s' x xa rb*) **then show** *?case*
apply (*clarsimp simp*: *fA-rel-inv-def fM-rel-inv-def*)
apply (*drule handshake-phase-invD*)
apply (*drule spec*[**where** *x*=*m*])
apply (*clarsimp simp*: *sys-phase-inv-aux-case*
split: *hs-phase.splits if-splits*)

apply (*blast dest*: *heap-colours-colours*)

apply (*metis* (*no-types*, *lifting*) *black-def no-black-refsD obj-at-cong option.simps*(*3*))
apply (*metis* (*no-types*, *lifting*) *black-def no-black-refsD obj-at-cong option.distinct*(*1*))

apply (*clarsimp simp*: *hp-step-rel-def*)
apply (*elim disjE*; *force simp*: *fA-rel-def fM-rel-def split*: *obj-at-splits*)

apply (*clarsimp simp*: *hp-step-rel-def*)
apply (*elim disjE*; *force simp*: *fA-rel-def fM-rel-def split*: *obj-at-splits*)
done
qed

14 Coarse TSO invariants

context *gc*
begin

lemma *tso-lock-invL*[*intro*]:
{‖ *tso-lock-invL* ‖} *gc*
by *vcg-jackhammer*

lemma *tso-store-inv*[*intro*]:
{‖ *LSTP tso-store-inv* ‖} *gc*
unfolding *tso-store-inv-def* **by** *vcg-jackhammer*

lemma *mut-tso-lock-invL*[*intro*]:
{‖ *mut-m.tso-lock-invL m* ‖} *gc*
by (*vcg-chainsaw mut-m.tso-lock-invL-def*)

end

context *mut-m*

begin

lemma *tso-store-inv*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
{| *LSTP tso-store-inv* |} *mutator m*
unfolding *tso-store-inv-def* **by** *vcg-jackhammer*

lemma *gc-tso-lock-invL*[*intro*]:
{| *gc.tso-lock-invL* |} *mutator m*
by (*vcg-chainsaw gc.tso-lock-invL-def*)

lemma *tso-lock-invL*[*intro*]:
{| *tso-lock-invL* |} *mutator m*
by *vcg-jackhammer*

end

context *mut-m′*
begin

lemma *tso-lock-invL*[*intro*]:
{| *tso-lock-invL* |} *mutator m′*
by (*vcg-chainsaw tso-lock-invL*)

end

context *sys*
begin

lemma *tso-gc-store-inv*[*intro*]:
notes *fun-upd-apply*[*simp*]
shows
{| *LSTP tso-store-inv* |} *sys*
apply (*vcg-chainsaw tso-store-inv-def*)
apply (*metis* (*no-types*) *list.set-intros*(*2*))
done

lemma *gc-tso-lock-invL*[*intro*]:
{| *gc.tso-lock-invL* |} *sys*
by (*vcg-chainsaw gc.tso-lock-invL-def*)

lemma *mut-tso-lock-invL*[*intro*]:
{| *mut-m.tso-lock-invL m* |} *sys*
by (*vcg-chainsaw mut-m.tso-lock-invL-def*)

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*: ∀ *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*)|)))
using *assms* **unfolding** *valid-refs-inv-def grey-reachable-def no-grey-refs-def*

apply (*clarsimp dest*!: *reachable-sweep-loop-free*)
apply (*drule mut-m.reachable-blackD*[*OF ngr spec*[*OF rsi*]])
apply (*auto split*: *obj-at-splits*)
done

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(*vcg-jackhammer* (*no-thin-post-inv*), *vcg-name-cases*)
case (*sweep-loop-free s s'*) **then show** *?case*
apply −
apply (*drule* (*1*) *handshake-in-syncD*)
apply (*rule valid-refs-inv-sweep-loop-free*, *assumption*, *assumption*)
apply (*simp*; *fail*)
apply (*simp add*: *white-def*)
done
qed (*auto simp*: *valid-refs-inv-def grey-reachable-def*)

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'*|)))
unfolding *valid-refs-inv-def mut-m.reachable-def* **by** (*auto simp*: *fun-upd-apply*)

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'*|)))
unfolding *valid-refs-inv-def* **by** (*simp add*: *fun-upd-apply*)

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*|)|)))
unfolding *valid-refs-inv-def mut-m.reachable-def*
apply (*clarsimp simp*: *fun-upd-apply*)
apply (*auto elim*: *converse-reachesE split*: *obj-at-splits*)
done

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'*]) |)))
apply (*subst valid-refs-inv-def*)
apply (*auto simp*: *grey-reachable-def mut-m.reachable-def fun-upd-apply*)

apply (*subst* (*asm*) *tso-store-refs-simps*; *force*)+
done

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′*|)))
unfolding *valid-refs-inv-def* **by** (*simp add*: *fun-upd-apply*)

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*}|)))
unfolding *valid-refs-inv-def*
apply (*clarsimp simp*: *grey-reachable-def fun-upd-apply*)
apply (*metis grey-reachable-def valid-refs-invD*(*1*) *valid-refs-invD*(*10*) *valid-refs-inv-def*)
done

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(*vcg-jackhammer* (*keep-locs*) (*no-thin-post-inv*), *vcg-name-cases*)
next case (*hs-get-roots-done s s′*) **then show** *?case* **by** (*clarsimp simp*: *valid-refs-inv-def grey-reachable-def*)
next case (*hs-get-work-done s s′*) **then show** *?case* **by** (*clarsimp simp*: *valid-refs-inv-def grey-reachable-def*)
qed

end

lemma (**in** *sys*) *valid-refs-inv*[*intro*]:
{| *LSTP* (*valid-refs-inv* ∧ *tso-store-inv*) |} *sys* {| *LSTP valid-refs-inv* |}
proof(*vcg-jackhammer* (*no-thin-post-inv*), *vcg-name-cases*)
case (*tso-dequeue-store-buffer s s′ p w ws*) **then show** *?case*
unfolding *do-store-action-def*
apply (*auto simp*: *p-not-sys valid-refs-inv-dequeue-Mutate valid-refs-inv-dequeue-Mutate-Payload fun-upd-apply split*: *mem-store-action.splits*)
done
qed

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*
using *marked-not-white* **unfolding** *valid-W-inv-def WL-def* **by** (*auto 0 5 split*: *obj-at-splits*)

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)
using *valid-W-invD0*(*4*) **apply** *fastforce*
using *valid-W-invD0*(*1*) **apply** *fastforce*
apply (*metis UnE WL-def valid-W-invD0*(*1*) *valid-W-invD0*(*4*))
done

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*]
unfolding *valid-W-inv-def white-def* **by** (*clarsimp dest*!: *spec*[**where** *x*=*p*], *blast*)+

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*
unfolding *valid-W-inv-def*
apply (*simp-all add*: *split*: *obj-at-splits*)
apply *blast*+
done

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(*vcg-jackhammer* (*no-thin-post-inv*), *vcg-name-cases*)
case (*tso-dequeue-store-buffer s s' p w ws*) **then show** *?case*
proof(*cases w*)
case (*mw-Mark r fl*) **with** *tso-dequeue-store-buffer* **show** *?thesis*
apply (*subst valid-W-inv-def*)
apply *clarsimp*
apply (*frule* (*1*) *valid-W-invD*(*1*))
apply (*clarsimp simp*: *all-conj-distrib white-def valid-W-inv-sys-ghg-empty-iff filter-empty-conv obj-at-simps*)
apply (*intro allI conjI impI*)
apply (*auto elim*: *valid-W-invE2*)[*3*]
apply (*meson Int-emptyI valid-W-distinct-simps*(*3*))
apply (*meson valid-W-invD0*(*2*))
apply (*meson valid-W-invD0*(*2*))
using *valid-W-invD*(*2*) **apply** *fastforce*
apply *auto*[*1*]
using *valid-W-invD*(*2*) **apply** *fastforce*
done
next case (*mw-fM fl*) **with** *tso-dequeue-store-buffer* **show** *?thesis*
apply (*clarsimp simp*: *fM-rel-inv-def fM-rel-def p-not-sys*)

apply (*elim disjE*; *clarsimp*)
apply (*frule* (*1*) *no-grey-refs-no-pending-marks*)
apply (*subst valid-W-inv-def*)
apply *clarsimp*
apply (*meson Int-emptyI no-grey-refsD*(*1*) *no-grey-refsD*(*3*) *valid-W-distinct-simps*(*3*) *valid-W-invD*(*2*) *valid-W-inv-sys-ghg-empty-iff valid-W-inv-sys-mem-store-buffersD*(*3*))
done
qed *simp-all*
qed

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*}|))) = {}
unfolding *valid-W-inv-def WL-def* **by** (*auto 5 5 simp*: *fun-upd-apply*)

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*)])|)))
apply (*subst valid-W-inv-def*)
apply (*clarsimp simp*: *all-conj-distrib fun-upd-apply*)
apply (*intro allI conjI impI*)
apply (*auto simp*: *valid-W-invD valid-W-distinct-simps*(*3*) *valid-W-inv-sys-ghg-empty-iff valid-W-invD0 valid-W-inv-ghg valid-W-inv-colours*)
done

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*|)))
by (*auto simp*: *valid-W-inv-def fun-upd-apply*)

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* := {}|)))
apply (*subst valid-W-inv-def*)
apply (*clarsimp simp*: *all-conj-distrib valid-W-invD0*(*2*) *fun-upd-apply*)
apply (*intro allI conjI impI*)
apply (*auto simp*: *valid-W-invD valid-W-invD0*(*2*) *valid-W-distinct-simps*(*3*))
using *valid-W-distinct-simps*(*1*) **apply** *fastforce*
apply (*metis marked-not-white singletonD valid-W-invD*(*2*))
done

lemma *valid-W-inv-mo-co-unlock*:
⟦ *sys-mem-lock s* = *Some p*; *sys-mem-store-buffers p s* = [];
⋀*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*)))
unfolding *valid-W-inv-def* **by** (*clarsimp simp*: *fun-upd-apply*) (*metis emptyE empty-set*)

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(*vcg-jackhammer* (*no-thin-post-inv*), *vcg-name-cases*)
case (*sweep-loop-free s s′*) **then show** *?case*
apply (*subst valid-W-inv-def*)
apply (*clarsimp simp*: *all-conj-distrib white-def valid-W-inv-sys-ghg-empty-iff*)
apply (*meson disjoint-iff-not-equal no-grey-refsD*(*1*) *no-grey-refsD*(*2*) *no-grey-refsD*(*3*) *valid-W-invE*(*5*))
done
next case (*mark-loop-get-work-load-W s s′*) **then show** *?case*
apply (*subst valid-W-inv-def*)
apply (*clarsimp simp*: *all-conj-distrib*)
apply (*intro allI conjI impI*; *auto dest*: *valid-W-invD0 valid-W-invD simp*: *valid-W-distinct-simps split*: *if-splits process-name.splits*)
done
next case (*mark-loop-blacken s s′*) **then show** *?case*
apply (*subst valid-W-inv-def*)
apply (*clarsimp simp*: *all-conj-distrib*)
apply (*intro allI conjI impI*; *auto dest*: *valid-W-invD0 valid-W-invD simp*: *valid-W-distinct-simps split*: *if-splits process-name.splits*)
done
next case (*mark-loop-mo-co-W s s′ y*) **then show** *?case* **by** − (*erule valid-W-inv-mo-co-W*; *blast*)
next case (*mark-loop-mo-co-unlock s s′ y*) **then show** *?case* **by** − (*erule valid-W-inv-mo-co-unlock*; *simp split*: *if-splits*)
next case (*mark-loop-mo-co-mark s s′ y*) **then show** *?case* **by** − (*erule valid-W-inv-mo-co-mark*; *blast*)
next case (*mark-loop-mo-co-lock s s′ y*) **then show** *?case* **by** − (*erule valid-W-inv-mo-co-lock*; *assumption+*)
next case (*mark-loop-get-roots-load-W s s′*) **then show** *?case*

apply (*subst valid-W-inv-def*)
apply (*clarsimp simp*: *all-conj-distrib valid-W-inv-sys-ghg-empty-iff*)
apply (*intro allI conjI impI*)
apply (*auto simp*: *valid-W-invD valid-W-invD0 valid-W-inv-sys-ghg-empty-iff split*: *process-name.splits*; *fail*)[*1*]
apply (*auto simp*: *valid-W-invD valid-W-invD0 valid-W-inv-sys-ghg-empty-iff split*: *process-name.splits*; *fail*)[*1*]
apply (*auto simp*: *valid-W-invD valid-W-invD0 valid-W-inv-sys-ghg-empty-iff split*: *process-name.splits*; *fail*)[*1*]
apply (*clarsimp split*: *process-name.splits*)
apply (*meson Int-emptyI Un-iff process-name.distinct*(*4*) *valid-W-distinct-simps*(*3*) *valid-W-invD0*(*1*))
apply (*auto simp*: *valid-W-invD valid-W-invD0 split*: *process-name.splits*; *fail*)[*1*]
apply (*auto simp*: *valid-W-invD valid-W-invD0 valid-W-inv-sys-ghg-empty-iff split*: *process-name.splits*; *fail*)[*1*]
using *valid-W-invD*(*2*) *valid-W-inv-sys-ghg-empty-iff* **apply** *fastforce*
apply (*auto simp*: *valid-W-invD valid-W-invD0 valid-W-inv-sys-ghg-empty-iff split*: *process-name.splits*; *fail*)[*1*]
apply (*auto simp*: *valid-W-invD valid-W-invD0 valid-W-inv-sys-ghg-empty-iff split*: *process-name.splits*; *fail*)[*1*]
apply (*auto simp*: *valid-W-invD valid-W-invD0 valid-W-inv-sys-ghg-empty-iff split*: *process-name.splits*; *fail*)[*1*]
apply (*auto simp*: *valid-W-invD valid-W-invD0 valid-W-inv-sys-ghg-empty-iff split*: *process-name.splits*; *fail*)[*1*]
apply (*auto simp*: *valid-W-invD valid-W-invD0 valid-W-inv-sys-ghg-empty-iff split*: *process-name.splits*; *fail*)[*1*]
apply (*auto simp*: *valid-W-invD valid-W-invD0 valid-W-inv-sys-ghg-empty-iff split*: *process-name.splits*; *fail*)[*1*]
done
qed

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*

$\wedge$ *LSTP* (*fM-rel-inv* $\wedge$ *sys-phase-inv* $\wedge$ *valid-refs-inv* $\wedge$ *valid-W-inv*) ⦄
mutator m
⦃ *LSTP valid-W-inv* ⦄
proof(*vcg-jackhammer* (*no-thin-post-inv*), *vcg-name-cases*)
case (*alloc s s′ r*) **then show** *?case*
apply (*subst valid-W-inv-def*)
apply (*clarsimp simp*: *all-conj-distrib split*: *if-split-asm*)
apply (*intro allI conjI impI*)
apply (*auto simp*: *valid-W-distinct-simps valid-W-invD0*(*3*) *valid-W-invD*(*2*))
done
next case (*store-ins-mo-co-W s s′ y*) **then show** *?case* **by** − (*erule valid-W-inv-mo-co-W*; *blast*+)
next case (*store-ins-mo-co-unlock s s′ y*) **then show** *?case* **by** − (*erule valid-W-inv-mo-co-unlock*; *simp split*: *if-splits*)
next case (*store-ins-mo-co-mark s s′ y*) **then show** *?case* **by** − (*erule valid-W-inv-mo-co-mark*; *blast*+)
next case (*store-ins-mo-co-lock s s′ y*) **then show** *?case* **by** − (*erule valid-W-inv-mo-co-lock*; *assumption*+)
next case (*store-del-mo-co-W s s′ y*) **then show** *?case* **by** − (*erule valid-W-inv-mo-co-W*; *blast*+)
next case (*store-del-mo-co-unlock s s′ y*) **then show** *?case* **by** − (*erule valid-W-inv-mo-co-unlock*; *simp split*: *if-splits*)
next case (*store-del-mo-co-mark s s′ y*) **then show** *?case* **by** − (*erule valid-W-inv-mo-co-mark*; *blast*+)
next case (*store-del-mo-co-lock s s′ y*) **then show** *?case* **by** − (*erule valid-W-inv-mo-co-lock*; *assumption*+)
next case (*hs-get-roots-loop-mo-co-W s s′ y*) **then show** *?case* **by** − (*erule valid-W-inv-mo-co-W*; *blast*+)
next case (*hs-get-roots-loop-mo-co-unlock s s′ y*) **then show** *?case* **by** − (*erule valid-W-inv-mo-co-unlock*; *simp split*: *if-splits*)
next case (*hs-get-roots-loop-mo-co-mark s s′ y*) **then show** *?case* **by** − (*erule valid-W-inv-mo-co-mark*; *blast*+)
next case (*hs-get-roots-loop-mo-co-lock s s′ y*) **then show** *?case* **by** − (*erule valid-W-inv-mo-co-lock*; *assumption*+)
next case (*hs-get-roots-done s s′*) **then show** *?case*
apply (*subst valid-W-inv-def*)
apply (*simp add*: *all-conj-distrib*)
apply (*intro allI conjI impI*; *clarsimp simp*: *valid-W-inv-sys-ghg-empty-iff valid-W-invD*(*2*) *valid-W-invD0*(*2*,*3*) *filter-empty-conv dest*!: *valid-W-invE*(*5*))
apply (*fastforce simp*: *valid-W-distinct-simps split*: *process-name.splits if-splits*)
done
next case (*hs-get-work-done s s′*) **then show** *?case*
apply (*subst valid-W-inv-def*)
apply (*simp add*: *all-conj-distrib*)
apply (*intro allI conjI impI*; *clarsimp simp*: *valid-W-inv-sys-ghg-empty-iff valid-W-invD*(*2*) *valid-W-invD0*(*2*,*3*) *filter-empty-conv dest*!: *valid-W-invE*(*5*))
apply (*fastforce simp*: *valid-W-distinct-simps split*: *process-name.splits if-splits*)
done
qed

17 Top-level safety

lemma (**in** *gc*) *I*:
⦃ *I* ⦄ *gc*
apply (*simp add*: *I-defs*)
apply (*rule valid-pre*)
apply (*rule valid-conj-lift valid-all-lift* | *fastforce*)+
done

lemma (**in** *sys*) *I*:
⦃ *I* ⦄ *sys*
apply (*simp add*: *I-defs*)
apply (*rule valid-pre*)
apply (*rule valid-conj-lift valid-all-lift* | *fastforce*)+

done

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(*cases m = m'*)
  case *True*
  interpret *mut-m m'* **by** *unfold-locales*
  from *True* **show** *?thesis*
    apply (*simp add*: *I-defs*)
    apply (*rule valid-pre*)
    apply (*rule valid-conj-lift* | *fastforce*)+
    done
next
  case *False*
  then interpret *mut-m' m' m* **by** *unfold-locales blast*
  from *False* **show** *?thesis*
    apply (*simp add*: *I-defs*)
    apply (*rule valid-pre*)
    apply (*rule valid-conj-lift* | *fastforce*)+
    done
qed

lemma *mutators-phase-inv*[*intro*]:
  $\{\!|$ *I* $|\!\}$ *mutator m* $\{\!|$ *LSTP* (*mut-m.mutator-phase-inv m'*) $|\!\}$
proof(*cases m = m'*)
  case *True*
  interpret *mut-m m'* **by** *unfold-locales*
  from *True* **show** *?thesis*
    apply (*simp add*: *I-defs*)
    apply (*rule valid-pre*)
    apply (*rule valid-conj-lift valid-all-lift* | *fastforce*)+
    done
next
  case *False*
  then interpret *mut-m' m' m* **by** *unfold-locales blast*
  from *False* **show** *?thesis*
    apply (*simp add*: *I-defs*)
    apply (*rule valid-pre*)
    apply (*rule valid-conj-lift valid-all-lift* | *fastforce*)+
    done
qed

lemma (**in** *mut-m*) *I*:
  $\{\!|$ *I* $|\!\}$ *mutator m*
apply (*simp add*: *I-def gc.invsL-def invs-def Local-Invariants.invsL-def*)
apply (*rule valid-pre*)
apply (*rule valid-conj-lift valid-all-lift* | *fastforce*)+
apply (*simp add*: *I-defs*)
done

context *gc-system*
begin

theorem *I*: *gc-system* $\models_{pre}$ *I*
apply (*rule VCG*)

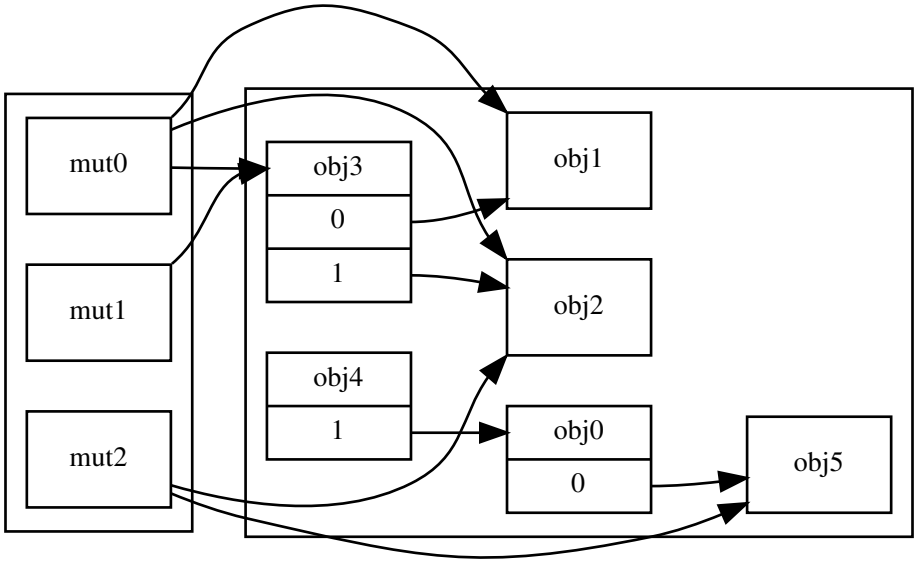

Figure 1: A concrete system state.

apply (*rule init-inv*)
apply (*rename-tac p*)
apply (*case-tac p*, *simp-all*)
apply (*rule mut-m.I*[*unfolded valid-proc-def*, *simplified*])
apply (*rule gc.I*[*unfolded valid-proc-def*, *simplified*])
apply (*rule sys.I*[*unfolded valid-proc-def*, *simplified*])
done

Our headline safety result follows directly.

corollary *safety*: *gc-system* $\models_{pre}$ *LSTP valid-refs*
using *I* **unfolding** *I-def invs-def valid-refs-def prerun-valid-def*
apply *clarsimp*
apply (*drule-tac x*=σ **in** *spec*)
apply (*drule* (*1*) *mp*)
apply (*rule alwaysI*)
apply (*erule-tac i*=*i* **in** *alwaysE*)
apply (*clarsimp simp*: *valid-refs-invD*(*1*))
done

end

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

interpretation *gc-system-interpretation*: *gc-system undefined* **.**

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

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

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

lemma
  gc-system-init init-state

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.