

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

Peter Gammie, Tony Hosking and Kai Engelhardt

September 13, 2023

## 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

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>  | <b>2</b>  |
| <b>2</b> | <b>A model of a Schism garbage collector</b>                     | <b>3</b>  |
| 2.1      | Object marking . . . . .   | 6         |
| 2.2      | Handshakes . . . . .   | 8         |
| 2.3      | The system process . . . . .                                     | 11        |
| 2.4      | Mutators . . . . .   | 13        |
| 2.5      | Garbage collector . . . . .                                      | 15        |
| <b>3</b> | <b>Proofs Basis</b>  | <b>17</b> |
| 3.1      | Model-specific functions and predicates . . . . .                | 19        |
| 3.2      | Object colours . . . . .   | 21        |
| 3.3      | Reachability . . . . .   | 22        |
| 3.4      | Sundry detritus . . . . .  | 22        |
| <b>4</b> | <b>Global Invariants</b>   | <b>26</b> |
| 4.1      | The valid references invariant . . . . .                         | 26        |
| 4.2      | The strong-tricolour invariant . . . . .                         | 26        |
| 4.3      | Phase invariants . . . . .                                       | 27        |
| 4.3.1    | Writes to shared GC variables . . . . .                          | 28        |
| 4.4      | Worklist invariants . . . . .                                    | 29        |
| 4.5      | Coarse invariants about the stores a process can issue . . . . . | 29        |
| 4.6      | The global invariants collected . . . . .                        | 30        |
| 4.7      | Initial conditions . . . . .                                     | 30        |
| <b>5</b> | <b>Local invariants</b>  | <b>31</b> |
| 5.1      | TSO invariants . . . . .   | 31        |
| 5.2      | Handshake phases . . . . .                                       | 31        |
| 5.3      | Mark Object . . . . .  | 35        |
| 5.4      | The infamous termination argument . . . . .                      | 38        |
| 5.5      | Sweep loop invariants . . . . .                                  | 39        |
| 5.6      | The local innvariants collected . . . . .                        | 40        |
| <b>6</b> | <b>CIMP specialisation</b>                                       | <b>40</b> |
| 6.1      | Hoare triples . . . . .  | 40        |
| 6.2      | Tactics . . . . .  | 41        |

|           |  |            |
|-----------|--|------------|
| 6.2.1     | Model-specific . . . . .                         | 41         |
| 6.2.2     | Locations . . . . .                              | 42         |
| <b>7</b>  | <b>Global invariants lemma bucket</b>            | <b>49</b>  |
| 7.1       | TSO invariants . . . . .                         | 49         |
| 7.2       | FIXME mutator handshake facts . . . . .          | 51         |
| 7.3       | points to, reaches, reachable mut . . . . .      | 52         |
| 7.4       | Colours . . . . .                                | 54         |
| 7.5       | <i>valid-W-inv</i> . . . . .                     | 58         |
| 7.6       | <i>grey-reachable</i> . . . . .                  | 59         |
| 7.7       | valid refs inv . . . . .                         | 59         |
| 7.8       | Location-specific simplification rules . . . . . | 62         |
| <b>8</b>  | <b>Local invariants lemma bucket</b>             | <b>71</b>  |
| 8.1       | Location facts . . . . .                         | 71         |
| 8.2       | <i>obj-fields-marked-inv</i> . . . . .           | 73         |
| 8.3       | mark object . . . . .                            | 74         |
| <b>9</b>  | <b>Initial conditions</b>                        | <b>75</b>  |
| <b>10</b> | <b>Noninterference</b>                           | <b>77</b>  |
| 10.1      | The infamous termination argument . . . . .      | 80         |
| <b>11</b> | <b>Global non-interference</b>                   | <b>87</b>  |
| <b>12</b> | <b>Mark Object</b>                               | <b>94</b>  |
| 12.1      | <i>obj-fields-marked-inv</i> . . . . .           | 96         |
| <b>13</b> | <b>Handshake phases</b>                          | <b>102</b> |
| 13.0.1    | sys phase inv . . . . .                          | 106        |
| 13.1      | Sweep loop invariants . . . . .                  | 110        |
| 13.2      | Mutator proofs . . . . .                         | 112        |
| <b>14</b> | <b>Coarse TSO invariants</b>                     | <b>118</b> |
| <b>15</b> | <b>Valid refs inv proofs</b>                     | <b>119</b> |
| <b>16</b> | <b>Worklist invariants</b>                       | <b>121</b> |
| <b>17</b> | <b>Top-level safety</b>                          | <b>125</b> |
| <b>18</b> | <b>A concrete system state</b>                   | <b>127</b> |
|           | <b>References</b>                                | <b>129</b> |

# 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 arbitrary (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*  $\Rightarrow$  *hs-phase*

### where

*hs-step ph* = (case *ph* of  
   *hp-Idle*  $\Rightarrow$  *hp-IdleInit*  
 | *hp-IdleInit*  $\Rightarrow$  *hp-InitMark*  
 | *hp-InitMark*  $\Rightarrow$  *hp-Mark*  
 | *hp-Mark*  $\Rightarrow$  *hp-IdleMarkSweep*  
 | *hp-IdleMarkSweep*  $\Rightarrow$  *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 *'fields* 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*  $\rightarrow$  *'ref*  
*obj-payload* :: *'field*  $\rightarrow$  *'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*  $\equiv$  *ro-Load* *mr-fM*

**abbreviation** *LoadMark* *r*  $\equiv$  *ro-Load* (*mr-Mark* *r*)

**abbreviation** *LoadPayload* *r* *f*  $\equiv$  *ro-Load* (*mr-Payload* *r* *f*)

**abbreviation** *LoadPhase*  $\equiv$  *ro-Load* *mr-Phase*

**abbreviation** *LoadRef* *r* *f*  $\equiv$  *ro-Load* (*mr-Ref* *r* *f*)

**abbreviation** *StorefA* *m*  $\equiv$  *ro-Store* (*mw-fA* *m*)

**abbreviation** *StorefM* *m*  $\equiv$  *ro-Store* (*mw-fM* *m*)

**abbreviation**  $\text{StoreMark } r \ m \equiv \text{ro-Store } (mw\text{-Mark } r \ m)$   
**abbreviation**  $\text{StorePayload } r \ f \ pl \equiv \text{ro-Store } (mw\text{-Mutate-Payload } r \ f \ pl)$   
**abbreviation**  $\text{StorePhase } ph \equiv \text{ro-Store } (mw\text{-Phase } ph)$   
**abbreviation**  $\text{StoreRef } r \ f \ r' \equiv \text{ro-Store } (mw\text{-Mutate } r \ f \ r')$

**type-synonym**  $(\text{'field}, \text{'mut}, \text{'payload}, \text{'ref}) \text{ request}$   
 $= \text{'mut process-name} \times (\text{'field}, \text{'mut}, \text{'payload}, \text{'ref}) \text{ request-op}$

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

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**  $(\text{'field}, \text{'mut}, \text{'payload}, \text{'ref}) \text{ local-state} =$   
— System-specific fields  
 $heap :: \text{'ref} \rightarrow (\text{'field}, \text{'payload}, \text{'ref}) \text{ object}$   
— TSO memory state  
 $mem\text{-store-buffers} :: \text{'mut process-name} \Rightarrow (\text{'field}, \text{'payload}, \text{'ref}) \text{ mem-store-action list}$   
 $mem\text{-lock} :: \text{'mut process-name } option$   
— Handshake state  
 $hs\text{-pending} :: \text{'mut} \Rightarrow bool$   
— Ghost state  
 $ghost\text{-hs-in-sync} :: \text{'mut} \Rightarrow bool$   
 $ghost\text{-hs-phase} :: hs\text{-phase}$   
  
— Mutator-specific temporaries  
 $new\text{-ref} :: \text{'ref } option$   
 $roots :: \text{'ref } set$   
 $ghost\text{-honorary-root} :: \text{'ref } set$   
 $payload\text{-value} :: \text{'payload } option$   
 $mutator\text{-data} :: \text{'field} \rightarrow \text{'payload}$   
 $mutator\text{-hs-pending} :: bool$   
  
— Garbage collector-specific temporaries  
 $field\text{-set} :: \text{'field } set$   
 $mut :: \text{'mut}$   
 $mut s :: \text{'mut } set$   
  
— Local variables used by multiple processes  
 $fA :: gc\text{-mark}$   
 $fM :: gc\text{-mark}$   
 $cas\text{-mark} :: gc\text{-mark } option$   
 $field :: \text{'field}$   
 $mark :: gc\text{-mark } option$   
 $phase :: gc\text{-phase}$   
 $tmp\text{-ref} :: \text{'ref}$   
 $ref :: \text{'ref } option$   
 $refs :: \text{'ref } set$   
 $W :: \text{'ref } set$   
— Handshake state

$hs\text{-}type :: hs\text{-}type$   
 — Ghost state  
 $ghost\text{-}honorary\text{-}grey :: 'ref\ set$

We instantiate CIMP's types as follows:

**type-synonym**  $(field, 'mut, 'payload, 'ref)\ gc\text{-}com$   
 $= ((field, 'payload, 'ref)\ response, location, (field, 'mut, 'payload, 'ref)\ request, (field, 'mut, 'payload, 'ref)\ local\text{-}state)\ com$   
**type-synonym**  $(field, 'mut, 'payload, 'ref)\ gc\text{-}loc\text{-}comp$   
 $= ((field, 'payload, 'ref)\ response, location, (field, 'mut, 'payload, 'ref)\ request, (field, 'mut, 'payload, 'ref)\ local\text{-}state)\ loc\text{-}comp$   
**type-synonym**  $(field, 'mut, 'payload, 'ref)\ gc\text{-}pred$   
 $= ((field, 'payload, 'ref)\ response, location, 'mut\ process\text{-}name, (field, 'mut, 'payload, 'ref)\ request, (field, 'mut, 'payload, 'ref)\ local\text{-}state)\ state\text{-}pred$   
**type-synonym**  $(field, 'mut, 'payload, 'ref)\ gc\text{-}system$   
 $= ((field, 'payload, 'ref)\ response, location, 'mut\ process\text{-}name, (field, 'mut, 'payload, 'ref)\ request, (field, 'mut, 'payload, 'ref)\ local\text{-}state)\ system$   
  
**type-synonym**  $(field, 'mut, 'payload, 'ref)\ gc\text{-}event$   
 $= (field, 'mut, 'payload, 'ref)\ request \times (field, 'payload, 'ref)\ response$   
**type-synonym**  $(field, 'mut, 'payload, 'ref)\ gc\text{-}history$   
 $= (field, 'mut, 'payload, 'ref)\ gc\text{-}event\ list$   
  
**type-synonym**  $(field, 'mut, 'payload, 'ref)\ lst\text{-}pred$   
 $= (field, 'mut, 'payload, 'ref)\ local\text{-}state \Rightarrow bool$   
  
**type-synonym**  $(field, 'mut, 'payload, 'ref)\ lsts$   
 $= 'mut\ process\text{-}name \Rightarrow (field, 'mut, 'payload, 'ref)\ local\text{-}state$   
  
**type-synonym**  $(field, 'mut, 'payload, 'ref)\ lsts\text{-}pred$   
 $= (field, 'mut, 'payload, 'ref)\ lsts \Rightarrow bool$

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**  $mut\text{-}m = \text{fixes } m :: 'mut$   
**locale**  $mut\text{-}m' = mut\text{-}m + \text{fixes } m' :: 'mut \text{ assumes } mm'[iff]: m \neq m'$   
**locale**  $gc$   
**locale**  $sys$

## 2.1 Object marking

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

**context**  
 $\text{fixes } p :: 'mut\ process\text{-}name$   
**begin**

**abbreviation**  $lock\text{-}syn :: location \Rightarrow (field, 'mut, 'payload, 'ref)\ gc\text{-}com \text{ where}$   
 $lock\text{-}syn\ l \equiv \{\!|l|\!\} Request\ (\lambda s. (p, ro\text{-}Lock))\ (\lambda\text{-} s. \{s\})$   
**notation**  $lock\text{-}syn\ (\{\!|-\!\})\ lock$

**abbreviation**  $unlock\text{-}syn :: location \Rightarrow (field, 'mut, 'payload, 'ref)\ gc\text{-}com \text{ where}$

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

**notation**  $unlock\text{-}syn\ (\{l\}\ unlock)$

### abbreviation

$load\text{-}mark\text{-}syn :: location \Rightarrow (('field, 'mut, 'payload, 'ref) local\text{-}state \Rightarrow 'ref)$   
 $\Rightarrow ((gc\text{-}mark\ option \Rightarrow gc\text{-}mark\ option)$   
 $\Rightarrow ('field, 'mut, 'payload, 'ref) local\text{-}state$   
 $\Rightarrow ('field, 'mut, 'payload, 'ref) local\text{-}state) \Rightarrow ('field, 'mut, 'payload, 'ref) gc\text{-}com$

### where

$load\text{-}mark\text{-}syn\ l\ r\ upd \equiv \{l\} Request\ (\lambda s. (p, LoadMark\ (r\ s)))\ (\lambda mv\ s. \{ upd\ \langle m \rangle\ s \mid m. mv = mv\text{-}Mark\ m \})$

**notation**  $load\text{-}mark\text{-}syn\ (\{l\}\ load'\text{-}mark)$

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

$load\text{-}fM\text{-}syn\ l \equiv \{l\} Request\ (\lambda s. (p, ro\text{-}Load\ mr\text{-}fM))\ (\lambda mv\ s. \{ s[fM := m] \mid m. mv = mv\text{-}Mark\ (Some\ m) \})$

**notation**  $load\text{-}fM\text{-}syn\ (\{l\}\ load'\text{-}fM)$

### abbreviation

$load\text{-}phase :: location \Rightarrow ('field, 'mut, 'payload, 'ref) gc\text{-}com$

### where

$load\text{-}phase\ l \equiv \{l\} Request\ (\lambda s. (p, LoadPhase))\ (\lambda mv\ s. \{ s[phase := ph] \mid ph. mv = mv\text{-}Phase\ ph \})$

**notation**  $load\text{-}phase\ (\{l\}\ load'\text{-}phase)$

### abbreviation

$store\text{-}mark\text{-}syn :: location \Rightarrow (('field, 'mut, 'payload, 'ref) local\text{-}state \Rightarrow 'ref) \Rightarrow (('field, 'mut, 'payload, 'ref) local\text{-}state \Rightarrow bool) \Rightarrow ('field, 'mut, 'payload, 'ref) gc\text{-}com$

### where

$store\text{-}mark\text{-}syn\ l\ r\ fl \equiv \{l\} Request\ (\lambda s. (p, StoreMark\ (r\ s)\ (fl\ s)))\ (\lambda s. \{ s[ghost\text{-}honorary\text{-}grey := \{r\ s\}] \})$

**notation**  $store\text{-}mark\text{-}syn\ (\{l\}\ store'\text{-}mark)$

### abbreviation

$add\text{-}to\text{-}W\text{-}syn :: location \Rightarrow (('field, 'mut, 'payload, 'ref) local\text{-}state \Rightarrow 'ref) \Rightarrow ('field, 'mut, 'payload, 'ref) gc\text{-}com$

### where

$add\text{-}to\text{-}W\text{-}syn\ l\ r \equiv \{l\} \lfloor \lambda s. s[W := W \cup \{r\ s\}, ghost\text{-}honorary\text{-}grey := \{\}] \rfloor$

**notation**  $add\text{-}to\text{-}W\text{-}syn\ (\{l\}\ add'\text{-}to'\text{-}W)$

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

$mark\text{-}object\text{-}fn :: location \Rightarrow ('field, 'mut, 'payload, 'ref) gc\text{-}com$

### where

$mark\text{-}object\text{-}fn\ l =$   
 $\{l @ \text{"-mo-null"}\} IF \neg (NULL\ ref) THEN$   
 $\{l @ \text{"-mo-mark"}\} load\text{-}mark\ (the \circ ref)\ mark\text{-}update\ ;;$   
 $\{l @ \text{"-mo-fM"}\} load\text{-}fM\ ;;$   
 $\{l @ \text{"-mo-mtest"}\} IF mark \neq Some \circ fM THEN$   
 $\{l @ \text{"-mo-phase"}\} load\text{-}phase\ ;;$   
 $\{l @ \text{"-mo-ptest"}\} IF phase \neq \langle ph\text{-}Idle \rangle THEN$   
 — CAS: claim object  
 $\{l @ \text{"-mo-co-lock"}\} lock\ ;;$   
 $\{l @ \text{"-mo-co-cmark"}\} load\text{-}mark\ (the \circ ref)\ cas\text{-}mark\text{-}update\ ;;$   
 $\{l @ \text{"-mo-co-ctest"}\} IF cas\text{-}mark = mark THEN$   
 $\{l @ \text{"-mo-co-mark"}\} store\text{-}mark\ (the \circ ref)\ fM$   
 $FI\ ;;$

```

    {l @ "-mo-co-unlock"} unlock ;;
    {l @ "-mo-co-won"} IF cas-mark = mark THEN
      {l @ "-mo-co-W"} add-to-W (the o ref)
    FI
  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**  $hp\text{-}step :: hs\text{-}type \Rightarrow hs\text{-}phase \Rightarrow hs\text{-}phase$  **where**

```

hp-step ht ≡
  case ht of
    ht-NOOP ⇒ hs-step
  | ht-GetRoots ⇒ hs-step
  | ht-GetWork ⇒ id

```

**context**  $sys$

**begin**

**definition**

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

**where**

```

handshake =
  { "sys-hs-gc-set-type" } Response
  (λ req 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
  (λ req 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
  (λ req s. { (s, mv-Bool (¬hs-pending s m))
    | m.req = (gc, ro-hs-gc-load-pending m) })
  ⊕ { "sys-hs-gc-load-W" } Response
  (λ req s. { (s { W := {} }, mv-Refs (W s))
    | ::unit.req = (gc, ro-hs-gc-load-W) })
  ⊕ { "sys-hs-mut-pending" } Response
  (λ req 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**  
 ⌈⌋ *mark-object* ≡ *mark-object-fn* (mutator *m*) *l*

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

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

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

**abbreviation** *hs-noop-done-syn* :: *location* ⇒ ('field, 'mut, 'payload, 'ref) *gc-com* (⌈⌋-⌋ *hs'-noop'-done'-*) **where**  
 ⌈⌋ *hs-noop-done-* ≡ ⌈⌋ *Request* (λs. (mutator *m*, ro-hs-mut-done {}))  
 (λ- s. {s⌊ ghost-hs-phase := hs-step (ghost-hs-phase s) ⌋})

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

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

**definition**

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

**where**

```

handshake =
  ⌈⌈"hs-load-pending"⌋⌋ hs-load-pending- ;;
  ⌈⌈"hs-pending"⌋⌋ IF mutator-hs-pending
  THEN
    ⌈⌈"hs-mfence"⌋⌋ MFENCE ;;
    ⌈⌈"hs-load-ht"⌋⌋ hs-load-type ;;
    ⌈⌈"hs-noop"⌋⌋ IF hs-type = ⟨ht-NOOP⟩
  THEN
    ⌈⌈"hs-noop-done"⌋⌋ hs-noop-done-

```

```

ELSE  $\{\{hs\text{-}get\text{-}roots'\}\}$  IF  $hs\text{-}type = \langle ht\text{-}GetRoots \rangle$ 
THEN
   $\{\{hs\text{-}get\text{-}roots\text{-}refs'\}\}$  'refs := 'roots ;;
   $\{\{hs\text{-}get\text{-}roots\text{-}loop'\}\}$  WHILE  $\neg EMPTY\ refs$  DO
     $\{\{hs\text{-}get\text{-}roots\text{-}loop\text{-}choose\text{-}ref'\}\}$  'ref := Some ' 'refs ;;
     $\{\{hs\text{-}get\text{-}roots\text{-}loop'\}\}$  mark-object ;;
     $\{\{hs\text{-}get\text{-}roots\text{-}loop\text{-}done'\}\}$  'refs := ('refs - {the 'ref})
  OD ;;
   $\{\{hs\text{-}get\text{-}roots\text{-}done'\}\}$  hs-get-roots-done- W
ELSE  $\{\{hs\text{-}get\text{-}work'\}\}$  IF  $hs\text{-}type = \langle ht\text{-}GetWork \rangle$ 
THEN
   $\{\{hs\text{-}get\text{-}work\text{-}done'\}\}$  hs-get-work-done W
FI FI FI
FI

```

end

The garbage collector's side of the interface.

**context** gc

**begin**

**abbreviation**  $set\text{-}hs\text{-}type :: location \Rightarrow hs\text{-}type \Rightarrow ('field, 'mut, 'payload, 'ref) gc\text{-}com (\{\{-\}\} set'\text{-}hs'\text{-}type)$  **where**  
 $\{\{l\}\} set\text{-}hs\text{-}type\ ht \equiv \{\{l\}\} Request (\lambda s. (gc, ro\text{-}hs\text{-}gc\text{-}store\text{-}type\ ht)) (\lambda\text{-} s. \{s\})$

**abbreviation**  $set\text{-}hs\text{-}pending :: location \Rightarrow (('field, 'mut, 'payload, 'ref) local\text{-}state \Rightarrow 'mut) \Rightarrow ('field, 'mut, 'payload, 'ref) gc\text{-}com (\{\{-\}\} set'\text{-}hs'\text{-}pending)$  **where**  
 $\{\{l\}\} set\text{-}hs\text{-}pending\ m \equiv \{\{l\}\} Request (\lambda s. (gc, ro\text{-}hs\text{-}gc\text{-}store\text{-}pending\ (m\ s))) (\lambda\text{-} s. \{s\})$

**abbreviation**  $load\text{-}W :: location \Rightarrow ('field, 'mut, 'payload, 'ref) gc\text{-}com (\{\{-\}\} load'\text{-}W)$  **where**  
 $\{\{l\}\} load\text{-}W \equiv \{\{l\} @ \text{"load-W"}'\} Request (\lambda s. (gc, ro\text{-}hs\text{-}gc\text{-}load\text{-}W))$   
 $(\lambda resp\ s. \{s \mid W := W'\} \mid W'. resp = mv\text{-}Refs\ W')$

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

**definition**

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

```

 $\{\{l\}\} handshake\text{-}init\ req =$ 
   $\{\{l\} @ \text{"init-type"}'\} set\text{-}hs\text{-}type\ req$  ;;
   $\{\{l\} @ \text{"init-muts"}'\} 'mut := UNIV$  ;;
   $\{\{l\} @ \text{"init-loop"}'\} WHILE \neg (EMPTY\ muts) DO$ 
     $\{\{l\} @ \text{"init-loop-choose-mut"}'\} 'mut := 'mut$  ;;
     $\{\{l\} @ \text{"init-loop-set-pending"}'\} set\text{-}hs\text{-}pending\ mut$  ;;
     $\{\{l\} @ \text{"init-loop-done"}'\} 'mut := ('mut - {'mut})$ 
  OD

```

**definition**

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

**where**

```

 $\{\{l\}\} handshake\text{-}done =$ 
   $\{\{l\} @ \text{"done-muts"}'\} 'mut := UNIV$  ;;
   $\{\{l\} @ \text{"done-loop"}'\} WHILE \neg EMPTY\ muts DO$ 
     $\{\{l\} @ \text{"done-loop-choose-mut"}'\} 'mut := 'mut$  ;;
     $\{\{l\} @ \text{"done-loop-rendezvous"}'\} Request$ 
       $(\lambda s. (gc, ro\text{-}hs\text{-}gc\text{-}load\text{-}pending\ (mut\ s)))$ 
       $(\lambda mv\ s. \{s \mid muts := muts\ s - \{mut\ s \mid done.\ mv = mv\text{-}Bool\ done \wedge done\} \})$ 
  OD

```

**definition**

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

**where**

$\{\} handshake-noop =$   
 $\{\} @ "-mfence" MFENCE ;;$   
 $\{\} handshake-init ht-NOOP ;;$   
 $\{\} handshake-done$

**definition**

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

**where**

$\{\} handshake-get-roots =$   
 $\{\} handshake-init ht-GetRoots ;;$   
 $\{\} handshake-done ;;$   
 $\{\} load-W$

**definition**

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

**where**

$\{\} handshake-get-work =$   
 $\{\} handshake-init ht-GetWork ;;$   
 $\{\} handshake-done ;;$   
 $\{\} load-W$

**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**

$not-blocked :: ('field, 'mut, 'payload, 'ref) local-state \Rightarrow 'mut process-name \Rightarrow bool$

**where**

$not-blocked s p = (case mem-lock s of None \Rightarrow True \mid Some p' \Rightarrow p = p')$

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

**definition**

$do-store-action :: ('field, 'payload, 'ref) mem-store-action \Rightarrow ('field, 'mut, 'payload, 'ref) local-state \Rightarrow ('field, 'mut, 'payload, 'ref) local-state$

**where**

$do-store-action wact =$   
 $(\lambda s. case wact of$   
 $mw-Mark r gc-mark \Rightarrow s(heap := (heap s)(r := map-option (\lambda obj. obj(obj-mark := gc-mark))) (heap s)))$   
 $\mid mw-Mutate r f new-r \Rightarrow s(heap := (heap s)(r := map-option (\lambda obj. obj(obj-fields := (obj-fields obj))(f := new-r)) (heap s r)))$   
 $\mid mw-Mutate-Payload r f pl \Rightarrow s(heap := (heap s)(r := map-option (\lambda obj. obj(obj-payload := (obj-payload obj)(f := pl)) (heap s r)))$   
 $\mid mw-fM gc-mark \Rightarrow s(fM := gc-mark)$   
 $\mid mw-fA gc-mark \Rightarrow s(fA := gc-mark)$   
 $\mid mw-Phase gc-phase \Rightarrow s(phase := gc-phase))$

**definition**

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

**where**

$fold-stores \text{ ws} = fold (\lambda w. (\circ) (do-store-action \text{ w})) \text{ ws id}$

**abbreviation**

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

**where**

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

**definition**

$do-load-action :: ('field, 'ref) \text{ mem-load-action} \\ \Rightarrow ('field, 'mut, 'payload, 'ref) \text{ local-state} \\ \Rightarrow ('field, 'payload, 'ref) \text{ response}$

**where**

$do-load-action \text{ ract} = \\ (\lambda s. \text{ case } \text{ ract of} \\ \quad \text{mr-Ref } r \text{ f} \Rightarrow mv\text{-Ref } (Option.\text{bind } (heap \text{ s } r) (\lambda obj. obj\text{-fields } obj \text{ f})) \\ \quad | \text{mr-Payload } r \text{ f} \Rightarrow mv\text{-Payload } (Option.\text{bind } (heap \text{ s } r) (\lambda obj. obj\text{-payload } obj \text{ f})) \\ \quad | \text{mr-Mark } r \Rightarrow mv\text{-Mark } (map-option \text{ obj-mark } (heap \text{ s } r)) \\ \quad | \text{mr-Phase} \Rightarrow mv\text{-Phase } (phase \text{ s}) \\ \quad | \text{mr-fM} \Rightarrow mv\text{-Mark } (Some (\text{fM } \text{ s})) \\ \quad | \text{mr-fA} \Rightarrow mv\text{-Mark } (Some (\text{fA } \text{ s})))$

**definition**

$sys-load :: 'mut \text{ process-name} \\ \Rightarrow ('field, 'ref) \text{ mem-load-action} \\ \Rightarrow ('field, 'mut, 'payload, 'ref) \text{ local-state} \\ \Rightarrow ('field, 'payload, 'ref) \text{ response}$

**where**

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

**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**

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

**where**

$mem-TSO = \\ \quad \{ "tso-load" \} \text{ Response } (\lambda req \text{ s. } \{ (s, sys-load \text{ p mr } s) \\ \quad \quad | p \text{ mr. req} = (p, ro\text{-Load } mr) \wedge not\text{-blocked } s \text{ p } \}) \\ \quad \oplus \{ "tso-store" \} \text{ Response } (\lambda req \text{ s. } \{ (s \parallel mem\text{-store-buffers} := (mem\text{-store-buffers } s) (p := mem\text{-store-buffers } s \text{ p } @ [w]) \parallel, mv\text{-Void}) \\ \quad \quad | p \text{ w. req} = (p, ro\text{-Store } w) \}) \\ \quad \oplus \{ "tso-mfence" \} \text{ Response } (\lambda req \text{ s. } \{ (s, mv\text{-Void}) \\ \quad \quad | p. req = (p, ro\text{-MFENCE}) \wedge mem\text{-store-buffers } s \text{ p} = [] \}) \\ \quad \oplus \{ "tso-lock" \} \text{ Response } (\lambda req \text{ s. } \{ (s \parallel mem\text{-lock} := Some \text{ p } \parallel, mv\text{-Void}) \\ \quad \quad | p. req = (p, ro\text{-Lock}) \wedge mem\text{-lock } s = None \}) \\ \quad \oplus \{ "tso-unlock" \} \text{ Response } (\lambda req \text{ s. } \{ (s \parallel mem\text{-lock} := None \parallel, mv\text{-Void}) \\ \quad \quad | p. req = (p, ro\text{-Unlock}) \wedge mem\text{-lock } s = Some \text{ p } \wedge mem\text{-store-buffers } s \text{ p} = [] \}) \\ \quad \oplus \{ "tso-dequeue-store-buffer" \} \text{ LocalOp } (\lambda s. \{ (do-store-action \text{ w } s) \parallel mem\text{-store-buffers} := (mem\text{-store-buffers } s \text{ p } @ [w]) \parallel, mv\text{-Void}) \})$

$s)(p := ws) \Downarrow$

$| p \text{ } w \text{ } ws. \text{ mem-store-buffers } s \text{ } p = w \# ws \wedge \text{ not-blocked } s \text{ } p \wedge p \neq \text{ sys } \}$

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

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

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**

$\text{alloc} :: ('field, 'mut, 'payload, 'ref) \text{ gc-com}$   
**where**  
 $\text{alloc} = \{\!\!| \text{"alloc"} \!\!\} \text{ Response } (\lambda \text{req } s.$   
 $\quad \text{if } \text{dom } (\text{heap } s) = \text{UNIV}$   
 $\quad \text{then } \{(s, \text{mv-Ref } \text{None}) \mid \text{::unit. snd req} = \text{ro-Alloc} \}$   
 $\quad \text{else } \{ (s \mid \text{heap} := (\text{heap } s)(r := \text{Some } (\mid \text{obj-mark} = \text{fA } s, \text{obj-fields} = \text{Map.empty}, \text{obj-payload} = \text{Map.empty}$   
 $\mid) \mid), \text{mv-Ref } (\text{Some } r) \}$   
 $\quad \mid r. r \notin \text{dom } (\text{heap } s) \wedge \text{snd req} = \text{ro-Alloc} \})$

References are freed by removing them from *heap*.

**definition**

$\text{free} :: ('field, 'mut, 'payload, 'ref) \text{ gc-com}$   
**where**  
 $\text{free} = \{\!\!| \text{"sys-free"} \!\!\} \text{ Response } (\lambda \text{req } s.$   
 $\quad \{ (s \mid \text{heap} := (\text{heap } s)(r := \text{None})), \text{mv-Void} \} \mid r. \text{snd req} = \text{ro-Free } r \})$

The top-level system process.

**definition**

$\text{com} :: ('field, 'mut, 'payload, 'ref) \text{ gc-com}$   
**where**  
 $\text{com} =$   
 $\text{LOOP DO}$   
 $\quad \text{mem-TSO}$   
 $\quad \oplus \text{ alloc}$   
 $\quad \oplus \text{ free}$   
 $\quad \oplus \text{ handshake}$   
 $\text{OD}$

**end**

## 2.4 Mutators

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

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

**context** *mut-m*

**begin**

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

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

$\text{alloc} \equiv$   
 $\{\!\!| \text{"alloc"} \!\!\} \text{ Request } (\lambda s. (\text{mutator } m, \text{ro-Alloc}))$   
 $(\lambda \text{mv } s. \{ s \mid \text{roots} := \text{roots } s \cup \text{set-option } \text{opt-r} \} \mid \text{opt-r. mv} = \text{mv-Ref } \text{opt-r} \})$

The mutator can always discard any references it holds.

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

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

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

Dereferencing a reference can increase the set of mutator roots.

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

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

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

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

**abbreviation**

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

**where**

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

**abbreviation**

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

**where**

$\{\text{l}\} \text{ store-ref } r f r' \equiv \{\text{l}\} \text{ Request } (\lambda s. (\text{mutator } m, \text{StoreRef } (r s) (f s) (r' s))) (\lambda s. \{ s \mid \text{ghost-honorary-root} :=$   
 $\{\text{-}\} \})$

**definition**

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

**where**

$\text{store} =$   
 — Choose vars for  $\text{ref} \rightarrow \text{field} := \text{new-ref}$   
 $\{\text{"store-choose"}\} \text{ LocalOp } (\lambda s. \{ s \mid \text{tmp-ref} := r, \text{field} := f, \text{new-ref} := r' \mid$   
 $\quad \mid r f r'. r \in \text{roots } s \wedge r' \in \text{Some } \text{'roots } s \cup \{\text{None}\} \}) ;;$   
 — Mark the reference we're about to overwrite. Does not update roots.  
 $\{\text{"deref-del"}\} \text{ deref tmp-ref field ref-update} ;;$   
 $\{\text{"store-del"}\} \text{ mark-object} ;;$   
 — Mark the reference we're about to insert.  
 $\{\text{"lop-store-ins"}\} \text{ 'ref} := \text{'new-ref} ;;$   
 $\{\text{"store-ins"}\} \text{ mark-object} ;;$   
 $\{\text{"store-ins"}\} \text{ store-ref tmp-ref field new-ref}$

Load and store payload data.

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

$\text{load-payload} \equiv$

```

{"mut-load-payload-choose"} LocalOp (λs. { s | tmp-ref := r, field := f | r f. r ∈ roots s } ) ;;
{"mut-load-payload"} Request (λs. (mutator m, LoadPayload (tmp-ref s) (field s)))
    (λmv 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 (λs. { s | tmp-ref := r, field := f, payload-value := pl s | r f pl. r ∈
roots s } ) ;;
{"mut-store-payload"} Request (λs. (mutator m, StorePayload (tmp-ref s) (field s) (payload-value s)))
    (λmv 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 (λs. { 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 (λs. (gc, LoadRef (r s) (f s)))
    (λmv 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 (λs. (gc, LoadMark (r s))) (λmv s. { upd ⟨m⟩ s | m. mv = mv-Mark m })

```

**syntax**

$-gc-fassign :: location \Rightarrow idt \Rightarrow 'ref \Rightarrow 'field \Rightarrow ('field, 'mut, 'payload, 'ref) gc-com (\{\!\!|\!\!|\} \text{' -} := \text{' -} \rightarrow - [0, 0, 0, 70] \ 71)$

$-gc-massign :: location \Rightarrow idt \Rightarrow 'ref \Rightarrow ('field, 'mut, 'payload, 'ref) gc-com (\{\!\!|\!\!|\} \text{' -} := \text{' -} \rightarrow flag [0, 0, 0] \ 71)$

**translations**

$\{\!\!|\!\!|\} \text{' } q := \text{' } r \rightarrow f \Rightarrow CONST gc-deref \ l \ r \ \langle f \rangle \ (-update-name \ q)$

$\{\!\!|\!\!|\} \text{' } m := \text{' } r \rightarrow flag \Rightarrow CONST gc-load-mark \ l \ r \ (-update-name \ m)$

**context**  $gc$

**begin**

**abbreviation**  $store-fA-syn :: location \Rightarrow (('field, 'mut, 'payload, 'ref) local-state \Rightarrow gc-mark) \Rightarrow ('field, 'mut, 'payload, 'ref) gc-com (\{\!\!|\!\!|\} store'\text{-}fA) \textbf{ where}$

$\{\!\!|\!\!|\} store\text{-}fA \ f \equiv \{\!\!|\!\!|\} Request \ (\lambda s. (gc, StorefA \ (f \ s))) \ (\lambda\text{-} \ s. \ \{s\})$

**abbreviation**  $load-fM-syn :: location \Rightarrow ('field, 'mut, 'payload, 'ref) gc-com (\{\!\!|\!\!|\} load'\text{-}fM) \textbf{ where}$

$\{\!\!|\!\!|\} load\text{-}fM \equiv \{\!\!|\!\!|\} Request \ (\lambda s. (gc, LoadfM)) \ (\lambda mv \ s. \ \{s \{\!\!|\!\!|\} fM := m \mid m. mv = mv\text{-}Mark \ (Some \ m) \ \})$

**abbreviation**  $store-fM-syn :: location \Rightarrow ('field, 'mut, 'payload, 'ref) gc-com (\{\!\!|\!\!|\} store'\text{-}fM) \textbf{ where}$

$\{\!\!|\!\!|\} store\text{-}fM \equiv \{\!\!|\!\!|\} Request \ (\lambda s. (gc, StorefM \ (fM \ s))) \ (\lambda\text{-} \ s. \ \{s\})$

**abbreviation**  $store-phase-syn :: location \Rightarrow gc-phase \Rightarrow ('field, 'mut, 'payload, 'ref) gc-com (\{\!\!|\!\!|\} store'\text{-}phase) \textbf{ where}$

$\{\!\!|\!\!|\} store\text{-}phase \ ph \equiv \{\!\!|\!\!|\} Request \ (\lambda s. (gc, StorePhase \ ph)) \ (\lambda\text{-} \ s. \ \{s \{\!\!|\!\!|\} phase := ph \ \})$

**abbreviation**  $mark-object-syn :: location \Rightarrow ('field, 'mut, 'payload, 'ref) gc-com (\{\!\!|\!\!|\} mark'\text{-}object) \textbf{ where}$

$\{\!\!|\!\!|\} mark\text{-}object \equiv mark\text{-}object\text{-}fn \ gc \ l$

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

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

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**

$com =$

$LOOP \ DO$

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

$\{\!\!|\!\!|\} \text{"idle-load-fM"} \ \} load\text{-}fM \ ;\ ;$

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

$\{\!\!|\!\!|\} \text{"idle-store-fM"} \ \} store\text{-}fM \ ;\ ;$

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

$\{\!\!|\!\!|\} \text{"idle-phase-init"} \ \} store\text{-}phase \ ph\text{-}Init \ ;\ ;$

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

$\{\!\!|\!\!|\} \text{"init-phase-mark"} \ \} store\text{-}phase \ ph\text{-}Mark \ ;\ ;$

$\{\!\!|\!\!|\} \text{"mark-load-fM"} \ \} load\text{-}fM \ ;\ ;$

$\{\!\!|\!\!|\} \text{"mark-store-fA"} \ \} store\text{-}fA \ fM \ ;\ ;$

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

$\{\!\!|\!\!|\} \text{"mark-loop-get-roots"} \ \} handshake\text{-}get\text{-}roots \ ;\ ;\text{ --- } hp\text{-}IdleMarkSweep$

$\{\!\!|\!\!|\} \text{"mark-loop"} \ \} WHILE \neg EMPTY \ W \ DO$



```

{"mark-loop-inner"} WHILE  $\neg$ EMPTY W DO
  {"mark-loop-choose-ref"} 'tmp-ref  $\in$  'W ;;
  {"mark-loop-fields"} 'field-set := UNIV ;;
  {"mark-loop-mark-object-loop"} WHILE  $\neg$ EMPTY field-set DO
    {"mark-loop-mark-choose-field"} 'field  $\in$  'field-set ;;
    {"mark-loop-mark-deref"} 'ref := 'tmp-ref  $\rightarrow$  '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  $\neg$ EMPTY refs DO
  {"sweep-loop-choose-ref"} 'tmp-ref  $\in$  'refs ;;
  {"sweep-loop-load-mark"} 'mark := 'tmp-ref  $\rightarrow$  flag ;;
  {"sweep-loop-check"} IF  $\neg$ NULL mark  $\wedge$  the  $\circ$  mark  $\neq$  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  $\Rightarrow$  ('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*  $\cup$  (*Set.bind* *A* *B*)

**by** (auto simp: *Set.bind-def*)

**lemma** *option-bind-invE[elim]*:

$\llbracket \text{Option.bind } f \ g = \text{None}; \bigwedge a. \llbracket f = \text{Some } a; g \ a = \text{None} \rrbracket \Longrightarrow Q; f = \text{None} \Longrightarrow Q \rrbracket \Longrightarrow Q$

$\llbracket \text{Option.bind } f \ g = \text{Some } x; \bigwedge a. \llbracket f = \text{Some } a; g \ a = \text{Some } x \rrbracket \Longrightarrow Q \rrbracket \Longrightarrow Q$

**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*  $\Rightarrow$  *location set* **where**

*prefixed*  $p \equiv \{ l . \text{prefix } p \ l \}$

**abbreviation** *suffixed* :: *location*  $\Rightarrow$  *location set* **where**

*suffixed*  $p \equiv \{ l . \text{suffix } p \ l \}$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**abbreviation** *gc-W*  $s \equiv W \ (s \ gc)$

**abbreviation** *at-gc* :: *location*  $\Rightarrow$  ('field, 'mut, 'payload, 'ref) *lsts-pred*  $\Rightarrow$  ('field, 'mut, 'payload, 'ref) *gc-pred* **where**

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

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

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

**context** *mut-m*

**begin**

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

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

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

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

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

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

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

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

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

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

**abbreviation**  $mut\text{-}hs\text{-}pending\ s \equiv mutator\text{-}hs\text{-}pending\ (s\ (mutator\ m))$

**abbreviation**  $mut\text{-}hs\text{-}type\ s \equiv hs\text{-}type\ (s\ (mutator\ m))$

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

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

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

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

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

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

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

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

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

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

**end**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Projections on TSO buffers.

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

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

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

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

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

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

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

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

**abbreviation** (input) *tso-pending-mutate*  $p \equiv \text{tso-pending } p \text{ (is-mw-Mutate } \vee \text{ is-mw-Mutate-Payload)}$  — TSO makes it (mostly) not worth distinguishing these.

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

**abbreviation** (input) *tso-no-pending-marks*  $\equiv \forall p. \text{LIST-NULL } (\text{tso-pending-mark } p)$

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

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

**definition** *valid-null-ref*  $:: 'ref \text{ option} \Rightarrow ('field, 'mut, 'payload, 'ref) \text{ lsts-pred} \text{ where}$   
 $\text{valid-null-ref } r \equiv \text{case } r \text{ of None} \Rightarrow \langle \text{True} \rangle \mid \text{Some } r' \Rightarrow \text{valid-ref } r'$

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

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

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

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

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

## 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*  $:: 'ref \Rightarrow ('field, 'mut, 'payload, 'ref) \text{ lsts-pred} \text{ where}$   
 $\text{marked } r \text{ } s \equiv \text{obj-at } (\lambda obj. \text{obj-mark } obj = \text{sys-fM } s) \text{ } r \text{ } s$

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

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

**definition**  $WL :: 'mut\ process\text{-}name \Rightarrow ('field, 'mut, 'payload, 'ref)\ lsts \Rightarrow 'ref\ set$  **where**  
 $WL\ p = (\lambda s. W\ (s\ p) \cup ghost\text{-}honorary\text{-}grey\ (s\ p))$

**definition**  $grey :: 'ref \Rightarrow ('field, 'mut, 'payload, 'ref)\ lsts\text{-}pred$  **where**  
 $grey\ r = (\exists p. \langle r \rangle \in WL\ p)$

**definition**  $black :: 'ref \Rightarrow ('field, 'mut, 'payload, 'ref)\ lsts\text{-}pred$  **where**  
 $black\ r \equiv marked\ r \wedge \neg grey\ r$

These demonstrate the overlap in colours.

**lemma**  $colours\text{-}distinct[dest]:$

$black\ r\ s \implies \neg grey\ r\ s$

$black\ r\ s \implies \neg white\ r\ s$

$grey\ r\ s \implies \neg black\ r\ s$

$white\ r\ s \implies \neg black\ r\ s$

**by** ( $auto\ simp: black\text{-}def\ white\text{-}def\ obj\text{-}at\text{-}def\ split: option.splits$ )

**lemma**  $marked\text{-}imp\text{-}black\text{-}or\text{-}grey:$

$marked\ r\ s \implies black\ r\ s \vee grey\ r\ s$

$\neg white\ r\ s \implies \neg valid\text{-}ref\ r\ s \vee black\ r\ s \vee grey\ r\ s$

**by** ( $auto\ simp: black\text{-}def\ grey\text{-}def\ white\text{-}def\ obj\text{-}at\text{-}def\ split: option.splits$ )

In some phases the heap is monochrome.

**definition**  $black\text{-}heap :: ('field, 'mut, 'payload, 'ref)\ lsts\text{-}pred$  **where**  
 $black\text{-}heap = (\forall r. valid\text{-}ref\ r \longrightarrow black\ r)$

**definition**  $white\text{-}heap :: ('field, 'mut, 'payload, 'ref)\ lsts\text{-}pred$  **where**  
 $white\text{-}heap = (\forall r. valid\text{-}ref\ r \longrightarrow white\ r)$

**definition**  $no\text{-}black\text{-}refs :: ('field, 'mut, 'payload, 'ref)\ lsts\text{-}pred$  **where**  
 $no\text{-}black\text{-}refs = (\forall r. \neg black\ r)$

**definition**  $no\text{-}grey\text{-}refs :: ('field, 'mut, 'payload, 'ref)\ lsts\text{-}pred$  **where**  
 $no\text{-}grey\text{-}refs = (\forall r. \neg grey\ r)$

### 3.3 Reachability

We treat pending TSO heap mutations as extra mutator roots.

**abbreviation**  $store\text{-}refs :: ('field, 'payload, 'ref)\ mem\text{-}store\text{-}action \Rightarrow 'ref\ set$  **where**

$store\text{-}refs\ w \equiv case\ w\ of\ mw\text{-}Mutate\ r\ f\ r' \Rightarrow \{r\} \cup Option.set\ option\ r' \mid mw\text{-}Mutate\text{-}Payload\ r\ f\ pl \Rightarrow \{r\} \mid - \Rightarrow \{\}$

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

**abbreviation** (**in**  $mut\text{-}m$ )  $root :: 'ref \Rightarrow ('field, 'mut, 'payload, 'ref)\ lsts\text{-}pred$  **where**  
 $root\ x \equiv \langle x \rangle \in mut\text{-}roots \cup mut\text{-}ghost\text{-}honorary\text{-}root \cup tso\text{-}store\text{-}refs$

**definition** (**in**  $mut\text{-}m$ )  $reachable :: 'ref \Rightarrow ('field, 'mut, 'payload, 'ref)\ lsts\text{-}pred$  **where**  
 $reachable\ y = (\exists x. root\ x \wedge x\ reaches\ y)$

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

### 3.4 Sundry detritus

**lemmas** *eq-imp-simps* = — equations for deriving useful things from *eq-imp* facts

*eq-imp-def*

*all-conj-distrib*

*split-paired-All split-def fst-conv snd-conv prod-eq-iff*

*conj-explode*

*simp-thms*

**lemma** *p-not-sys*:

$p \neq \text{sys} \longleftrightarrow p = \text{gc} \vee (\exists m. p = \text{mutator } m)$

**by** (*cases p*) *simp-all*

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

**using** *mm'* **by** *blast*

obj at

**lemma** *obj-at-cong*[*cong*]:

$\llbracket \bigwedge \text{obj. sys-heap } s \ r = \text{Some } \text{obj} \implies P \ \text{obj} = P' \ \text{obj}; r = r'; s = s' \rrbracket$

$\implies \text{obj-at } P \ r \ s \longleftrightarrow \text{obj-at } P' \ r' \ s'$

**unfolding** *obj-at-def* **by** (*simp cong: option.case-cong*)

**lemma** *obj-at-split*:

$Q \ (\text{obj-at } P \ r \ s) = ((\text{sys-heap } s \ r = \text{None} \longrightarrow Q \ \text{False}) \wedge (\forall \text{obj. sys-heap } s \ r = \text{Some } \text{obj} \longrightarrow Q \ (P \ \text{obj})))$

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

**lemma** *obj-at-split-asm*:

$Q \ (\text{obj-at } P \ r \ s) = (\neg ((\text{sys-heap } s \ r = \text{None} \wedge \neg Q \ \text{False}) \vee (\exists \text{obj. sys-heap } s \ r = \text{Some } \text{obj} \wedge \neg Q \ (P \ \text{obj}))))$

**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*:

$\text{eq-imp } (\lambda(-::\text{unit}) \ s. \text{map-option } P \ (\text{sys-heap } s \ r))$

(*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*:

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

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

obj at field on heap

**lemma** *obj-at-field-on-heap-cong*[*cong*]:

$\llbracket \bigwedge r' \ \text{obj. } \llbracket \text{sys-heap } s \ r = \text{Some } \text{obj}; \text{obj-fields } \text{obj } f = \text{Some } r' \rrbracket \implies P \ r' = P' \ r'; r = r'; f = f'; s = s' \rrbracket$

$\implies \text{obj-at-field-on-heap } P \ r \ f \ s \longleftrightarrow \text{obj-at-field-on-heap } P' \ r' \ f' \ s'$

**unfolding** *obj-at-field-on-heap-def* **by** (*simp cong: option.case-cong*)

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

$Q \ (\text{obj-at-field-on-heap } P \ r \ f \ s) \longleftrightarrow ((\text{sys-heap } s \ r = \text{None} \longrightarrow Q \ \text{False})$

$\wedge (\forall \text{obj. sys-heap } s \ r = \text{Some } \text{obj} \wedge \text{obj-fields } \text{obj } f = \text{None} \longrightarrow Q \ \text{True})$

$\wedge (\forall r' \ \text{obj. sys-heap } s \ r = \text{Some } \text{obj} \wedge \text{obj-fields } \text{obj } f = \text{Some } r' \longrightarrow Q \ (P \ r'))$

**by** (*simp add: obj-at-field-on-heap-def split: option.splits*)

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

$Q \ (\text{obj-at-field-on-heap } P \ r \ f \ s) \longleftrightarrow (\neg ((\text{sys-heap } s \ r = \text{None} \wedge \neg Q \ \text{False})$

$\vee (\exists \text{obj. sys-heap } s \ r = \text{Some obj} \wedge \text{obj-fields obj } f = \text{None} \wedge \neg Q \ \text{True})$   
 $\vee (\exists r' \text{obj. sys-heap } s \ r = \text{Some obj} \wedge \text{obj-fields obj } f = \text{Some } r' \wedge \neg Q \ (P \ r'))$

**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:*

$\text{eq-imp } (\lambda(-::\text{unit}) \ s. \text{sys-heap } s \ r)$   
 $(\text{obj-at-field-on-heap } P \ r \ f)$

**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]:*

$\text{obj-at-field-on-heap } P \ r \ f \ s \implies \text{valid-ref } r \ s$   
 $\text{obj-at-field-on-heap } P \ r \ f \ s \implies \text{valid-null-ref } (\text{Some } r) \ s$

**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]:*

$\llbracket \text{obj-at-field-on-heap } P \ r \ f \ s; \text{sys-heap } s' \ r = \text{sys-heap } s \ r; \bigwedge r'. P \ r' \implies P' \ r' \rrbracket$   
 $\implies \text{obj-at-field-on-heap } P' \ r \ f \ s'$

**by** (*simp add: obj-at-field-on-heap-def split: option.splits*)

**lemma** *valid-null-ref-eq-imp:*

$\text{eq-imp } (\lambda(-::\text{unit}) \ s. \text{Option.bind } r \ (\text{map-option } \langle \text{True} \rangle \circ \text{sys-heap } s))$   
 $(\text{valid-null-ref } r)$

**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]:*

$\text{valid-null-ref } \text{None } s$   
 $\text{valid-null-ref } (\text{Some } r) \ s \longleftrightarrow \text{valid-ref } r \ s$

**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-action-split)*

**simps-of-case** *do-store-action-simps[simp]: fun-cong[OF do-store-action-def[simplified atomize-eq]] (splits: mem-store-action-split)*

**lemma** *do-store-action-prj-simps[simp]:*

$fM \ (\text{do-store-action } w \ s) = fl \longleftrightarrow (fM \ s = fl \wedge w \neq mw\text{-}fM \ (\neg fM \ s)) \vee w = mw\text{-}fM \ fl$   
 $fl = fM \ (\text{do-store-action } w \ s) \longleftrightarrow (fl = fM \ s \wedge w \neq mw\text{-}fM \ (\neg fM \ s)) \vee w = mw\text{-}fM \ fl$   
 $fA \ (\text{do-store-action } w \ s) = fl \longleftrightarrow (fA \ s = fl \wedge w \neq mw\text{-}fA \ (\neg fA \ s)) \vee w = mw\text{-}fA \ fl$   
 $fl = fA \ (\text{do-store-action } w \ s) \longleftrightarrow (fl = fA \ s \wedge w \neq mw\text{-}fA \ (\neg fA \ s)) \vee w = mw\text{-}fA \ fl$

$\text{ghost-hs-in-sync } (\text{do-store-action } w \ s) = \text{ghost-hs-in-sync } s$

$\text{ghost-hs-phase } (\text{do-store-action } w \ s) = \text{ghost-hs-phase } s$

$\text{ghost-honorary-grey } (\text{do-store-action } w \ s) = \text{ghost-honorary-grey } s$

$\text{hs-pending } (\text{do-store-action } w \ s) = \text{hs-pending } s$

$\text{hs-type } (\text{do-store-action } w \ s) = \text{hs-type } s$

$\text{heap } (\text{do-store-action } w \ s) \ r = \text{None} \longleftrightarrow \text{heap } s \ r = \text{None}$

$\text{mem-lock } (\text{do-store-action } w \ s) = \text{mem-lock } s$

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

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

$W \ (\text{do-store-action } w \ s) = W \ s$

**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*]:

$\llbracket (x \text{ reaches } y) \text{ } s; (y \text{ points-to } z) \text{ } s \rrbracket \implies (x \text{ reaches } z) \text{ } s$

$\llbracket (y \text{ reaches } z) \text{ } s; (x \text{ points-to } y) \text{ } s \rrbracket \implies (x \text{ reaches } z) \text{ } s$

**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**  $\bigwedge x. P \text{ } x \text{ } x$

**assumes**  $\bigwedge x \text{ } y \text{ } z. \llbracket (x \text{ reaches } y) \text{ } s; P \text{ } x \text{ } y; (y \text{ points-to } z) \text{ } s \rrbracket \implies P \text{ } x \text{ } z$

**shows**  $P \text{ } x \text{ } y$

**using** *assms* **unfolding** *reaches-def* **by** (*rule rtranclp.induct*)

**lemma** *converse-reachesE*[*consumes 1, case-names base step*]:

**assumes** (*x reaches z*) *s*

**assumes**  $x = z \implies P$

**assumes**  $\bigwedge y. \llbracket (x \text{ points-to } y) \text{ } s; (y \text{ reaches } z) \text{ } s \rrbracket \implies P$

**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**  $\forall r'. \bigcup (\text{ran } ' \text{obj-fields } ' \text{set-option } (\text{sys-heap } s' \text{ } r')) = \bigcup (\text{ran } ' \text{obj-fields } ' \text{set-option } (\text{sys-heap } s \text{ } r'))$

**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* ( $\lambda r' s. \bigcup (\text{ran } ' \text{obj-fields } ' \text{set-option } (\text{sys-heap } s \text{ } r'))$ )

(*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' ))*

$\longleftrightarrow \text{obj-at } (\lambda \text{obj}. (P \text{ (if } r = r' \text{ then obj() obj-mark := fl ) else obj))) \text{ } r \text{ } s$

**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 (λobj :: ('field, 'payload, 'ref) object. obj() obj-fields := (obj-fields obj)(f := opt-r')) (sys-heap s r))*),

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

$\llbracket sys\text{-}load\ (mutator\ m)\ (mr\text{-}Ref\ r\ f)\ (s\ sys) = mv\text{-}Ref\ opt\text{-}r'; \forall\ opt\text{-}r'.\ mw\text{-}Mutate\ r\ f\ opt\text{-}r' \notin set\ (sys\text{-}mem\text{-}store\text{-}buffers\ (mutator\ m)\ s); valid\text{-}ref\ r\ s \rrbracket$

$\implies obj\text{-}at\text{-}field\text{-}on\text{-}heap\ (\lambda r. opt\text{-}r' = Some\ r)\ r\ f\ s$

**unfolding** *sys-load-def fold-stores-def*

**apply** *clarsimp*

**apply** (*rule fold-invariant[where P= $\lambda fr. obj\text{-}at\text{-}field\text{-}on\text{-}heap\ (\lambda r'. Option.bind\ (heap\ (fr\ (s\ sys))\ r)\ (\lambda obj. obj\text{-}fields\ obj\ f) = Some\ r')$  r f s*

**and**  $Q = \lambda w. w \in set\ (sys\text{-}mem\text{-}store\text{-}buffers\ (mutator\ m)\ s)]$ )

**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 = ( $\forall m\ x. mut\text{-}m.reachable\ m\ x \vee grey\text{-}reachable\ x \longrightarrow valid\text{-}ref\ x$ )*

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

*x points-to-white y  $\equiv$  x points-to y  $\wedge$  white y*

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

*strong-tricolour-inv = ( $\forall b\ w. black\ b \longrightarrow \neg b\ points\text{-}to\text{-}white\ w$ )*

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

**definition** *has-white-path-to :: 'ref  $\Rightarrow$  'ref  $\Rightarrow$  ('field, 'mut, 'payload, 'ref) lsts-pred (infix has'-white'-path'-to 51) where*

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

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

*g grey-protects-white w = (grey g  $\wedge$  g has-white-path-to w)*

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

*weak-tricolour-inv =*

$$(\forall b w. \text{black } b \wedge b \text{ points-to-white } w \longrightarrow (\exists g. g \text{ grey-protects-white } w))$$

**lemma** *strong-tricolour-inv*  $s \implies \text{weak-tricolour-inv } s$

**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*  $:: ('ref \Rightarrow ('field, 'mut, 'payload, 'ref) \text{ lsts-pred } \mathbf{where}$   
*in-snapshot*  $r = (\text{black } r \vee (\exists g. g \text{ grey-protects-white } r))$

**definition** (*in mut-m*) *reachable-snapshot-inv*  $:: ('field, 'mut, 'payload, 'ref) \text{ lsts-pred } \mathbf{where}$   
*reachable-snapshot-inv*  $= (\forall r. \text{reachable } r \longrightarrow \text{in-snapshot } r)$

### 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*  $:: (\text{bool} \times \text{hs-type} \times \text{hs-phase} \times \text{hs-phase}) \text{ set } \mathbf{where}$   
*hp-step-rel*  $=$   
 $\{ \text{True} \} \times \{ (ht\text{-}NOOP, hp, hp) \mid hp. hp \in \{hp\text{-}Idle, hp\text{-}IdleInit, hp\text{-}InitMark, hp\text{-}Mark\} \}$   
 $\cup \{ (ht\text{-}GetRoots, hp\text{-}IdleMarkSweep, hp\text{-}IdleMarkSweep)$   
 $, (ht\text{-}GetWork, hp\text{-}IdleMarkSweep, hp\text{-}IdleMarkSweep) \}$   
 $\cup \{ \text{False} \} \times \{ (ht\text{-}NOOP, hp\text{-}Idle, hp\text{-}IdleMarkSweep)$   
 $, (ht\text{-}NOOP, hp\text{-}IdleInit, hp\text{-}Idle)$   
 $, (ht\text{-}NOOP, hp\text{-}InitMark, hp\text{-}IdleInit)$   
 $, (ht\text{-}NOOP, hp\text{-}Mark, hp\text{-}InitMark)$   
 $, (ht\text{-}GetRoots, hp\text{-}IdleMarkSweep, hp\text{-}Mark)$   
 $, (ht\text{-}GetWork, hp\text{-}IdleMarkSweep, hp\text{-}IdleMarkSweep) \}$

**definition** *handshake-phase-inv*  $:: ('field, 'mut, 'payload, 'ref) \text{ lsts-pred } \mathbf{where}$   
*handshake-phase-inv*  $= (\forall m.$   
 $\text{sys-ghost-hs-in-sync } m \otimes \text{sys-hs-type} \otimes \text{sys-ghost-hs-phase} \otimes \text{mut-m.mut-ghost-hs-phase } m \in \langle hp\text{-step-rel} \rangle$   
 $\wedge (\text{sys-hs-pending } m \longrightarrow \neg \text{sys-ghost-hs-in-sync } m))$

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*  $:: ('field, 'payload, 'ref) \text{ mem-store-action} \Rightarrow ('field, 'mut, 'payload, 'ref) \text{ lsts-pred}$   
**where**

$$\text{marked-insertion } w \equiv \lambda s. \text{case } w \text{ of } mw\text{-Mutate } r f (\text{Some } r') \Rightarrow \text{marked } r' s \mid - \Rightarrow \text{True}$$

**abbreviation** *marked-deletion*  $:: ('field, 'payload, 'ref) \text{ mem-store-action} \Rightarrow ('field, 'mut, 'payload, 'ref) \text{ lsts-pred}$   
**where**

$$\text{marked-deletion } w \equiv \lambda s. \text{case } w \text{ of } mw\text{-Mutate } r f \text{ opt-}r' \Rightarrow \text{obj-at-field-on-heap } (\lambda r'. \text{marked } r' s) r f s \mid - \Rightarrow \text{True}$$

**context** *mut-m*  
**begin**

**definition** *marked-insertions*  $:: ('field, 'mut, 'payload, 'ref) \text{ lsts-pred } \mathbf{where}$

$marked\_insertions = (\forall w. \text{tso-pending-store } (mutator\ m) \ w \longrightarrow \text{marked-insertion } w)$

**definition**  $marked\_deletions :: ('field, 'mut, 'payload, 'ref) \text{ lts-pred } \mathbf{where}$   
 $marked\_deletions = (\forall w. \text{tso-pending-store } (mutator\ m) \ w \longrightarrow \text{marked-deletion } w)$

**primrec**  $mutator\_phase\_inv\_aux :: hs\_phase \Rightarrow ('field, 'mut, 'payload, 'ref) \text{ lts-pred } \mathbf{where}$   
 $mutator\_phase\_inv\_aux\ hp\_Idle = \langle True \rangle$   
 $| mutator\_phase\_inv\_aux\ hp\_IdleInit = no\_black\_refs$   
 $| mutator\_phase\_inv\_aux\ hp\_InitMark = marked\_insertions$   
 $| mutator\_phase\_inv\_aux\ hp\_Mark = (marked\_insertions \wedge marked\_deletions)$   
 $| mutator\_phase\_inv\_aux\ hp\_IdleMarkSweep = (marked\_insertions \wedge marked\_deletions \wedge \text{reachable-snapshot-inv})$

**abbreviation**  $mutator\_phase\_inv :: ('field, 'mut, 'payload, 'ref) \text{ lts-pred } \mathbf{where}$   
 $mutator\_phase\_inv \equiv mutator\_phase\_inv\_aux\ \$\ mut\_ghost\_hs\_phase$

**end**

**abbreviation**  $mutators\_phase\_inv :: ('field, 'mut, 'payload, 'ref) \text{ lts-pred } \mathbf{where}$   
 $mutators\_phase\_inv \equiv (\forall m. mut\_m. mutator\_phase\_inv\ m)$

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

**primrec**  $sys\_phase\_inv\_aux :: hs\_phase \Rightarrow ('field, 'mut, 'payload, 'ref) \text{ lts-pred } \mathbf{where}$   
 $sys\_phase\_inv\_aux\ hp\_Idle = ( (If\ sys\_fA = sys\_fM\ Then\ black\_heap\ Else\ white\_heap) \wedge no\_grey\_refs )$   
 $| sys\_phase\_inv\_aux\ hp\_IdleInit = no\_black\_refs$   
 $| sys\_phase\_inv\_aux\ hp\_InitMark = (sys\_fA \neq sys\_fM \longrightarrow no\_black\_refs)$   
 $| sys\_phase\_inv\_aux\ hp\_Mark = \langle True \rangle$   
 $| sys\_phase\_inv\_aux\ hp\_IdleMarkSweep = ( (sys\_phase = \langle ph\_Idle \rangle \vee \text{tso-pending-store } gc\ (mw\_Phase\ ph\_Idle)) \longrightarrow no\_grey\_refs )$

**abbreviation**  $sys\_phase\_inv :: ('field, 'mut, 'payload, 'ref) \text{ lts-pred } \mathbf{where}$   
 $sys\_phase\_inv \equiv sys\_phase\_inv\_aux\ \$\ sys\_ghost\_hs\_phase$

#### 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**  $handshake\_phase\_rel :: hs\_phase \Rightarrow bool \Rightarrow gc\_phase \Rightarrow bool \mathbf{where}$   
 $handshake\_phase\_rel\ hp\ in\_sync\ ph =$   
 $(\text{case } hp \text{ of}$   
 $hp\_Idle \Rightarrow ph = ph\_Idle$   
 $| hp\_IdleInit \Rightarrow ph = ph\_Idle \vee (in\_sync \wedge ph = ph\_Init)$   
 $| hp\_InitMark \Rightarrow ph = ph\_Init \vee (in\_sync \wedge ph = ph\_Mark)$   
 $| hp\_Mark \Rightarrow ph = ph\_Mark$   
 $| hp\_IdleMarkSweep \Rightarrow ph = ph\_Mark \vee (in\_sync \wedge ph \in \{ ph\_Idle, ph\_Sweep \}))$

**definition**  $phase\_rel :: (bool \times hs\_phase \times gc\_phase \times gc\_phase \times ('field, 'payload, 'ref) \text{ mem-store-action list})$   
 $\text{set } \mathbf{where}$   
 $phase\_rel =$   
 $(\{ (in\_sync, hp, ph, ph, []) \mid in\_sync\ hp\ ph. \text{handshake-phase-rel } hp\ in\_sync\ ph \}$   
 $\cup (\{ True \} \times \{ (hp\_IdleInit, ph\_Init, ph\_Idle, [mw\_Phase\ ph\_Init]),$   
 $(hp\_InitMark, ph\_Mark, ph\_Init, [mw\_Phase\ ph\_Mark]),$   
 $(hp\_IdleMarkSweep, ph\_Sweep, ph\_Mark, [mw\_Phase\ ph\_Sweep]),$   
 $(hp\_IdleMarkSweep, ph\_Idle, ph\_Mark, [mw\_Phase\ ph\_Sweep, mw\_Phase\ ph\_Idle]),$   
 $(hp\_IdleMarkSweep, ph\_Idle, ph\_Sweep, [mw\_Phase\ ph\_Idle]) \} ))$

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

$\text{phase-rel-inv} = ((\forall m. \text{sys-ghost-hs-in-sync } m) \otimes \text{sys-ghost-hs-phase} \otimes \text{gc-phase} \otimes \text{sys-phase} \otimes \text{tso-pending-phase} \text{ gc} \in \langle \text{phase-rel} \rangle)$

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

**definition**  $\text{fM-rel} :: (\text{bool} \times \text{hs-phase} \times \text{gc-mark} \times \text{gc-mark} \times ('field, 'payload, 'ref) \text{ mem-store-action list} \times \text{bool}) \text{ set } \mathbf{where}$

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

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

$\text{fM-rel-inv} = ((\forall m. \text{sys-ghost-hs-in-sync } m) \otimes \text{sys-ghost-hs-phase} \otimes \text{gc-fM} \otimes \text{sys-fM} \otimes \text{tso-pending-fM } gc \otimes (\text{sys-mem-lock} = \langle \text{Some } gc \rangle) \in \langle \text{fM-rel} \rangle)$

**definition**  $\text{fA-rel} :: (\text{bool} \times \text{hs-phase} \times \text{gc-mark} \times \text{gc-mark} \times ('field, 'payload, 'ref) \text{ mem-store-action list}) \text{ set } \mathbf{where}$

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

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

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

#### 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  $\text{mark-object-fn}$ , the processes' worklists and  $\text{ghost-honorary-greys}$  are disjoint, and that pending marks are sensible.

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

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

**definition**  $\text{valid-W-inv} :: ('field, 'mut, 'payload, 'ref) \text{ lsts-pred } \mathbf{where}$

$\text{valid-W-inv} =$   
 $((\forall p \ r. r \text{ in-W } p \vee (\text{sys-mem-lock} \neq \langle \text{Some } p \rangle \wedge r \text{ in-ghost-honorary-grey } p) \longrightarrow \text{marked } r)$   
 $\wedge (\forall p \ q. \langle p \neq q \rangle \longrightarrow \text{WL } p \cap \text{WL } q = \langle \{\} \rangle)$   
 $\wedge (\forall p \ q \ r. \neg(r \text{ in-ghost-honorary-grey } p \wedge r \text{ in-W } q))$   
 $\wedge (\text{EMPTY } \text{sys-ghost-honorary-grey})$   
 $\wedge (\forall p \ r \ fl. \text{tso-pending-store } p \ (mw\text{-Mark } r \ fl)$   
 $\longrightarrow \langle fl \rangle = \text{sys-fM}$   
 $\wedge r \text{ in-ghost-honorary-grey } p$   
 $\wedge \text{tso-locked-by } p$   
 $\wedge \text{white } r$   
 $\wedge \text{tso-pending-mark } p = \langle [mw\text{-Mark } r \ fl] \rangle )$

#### 4.5 Coarse invariants about the stores a process can issue

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

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

**abbreviation**  $mut\text{-}writes :: ('field, 'payload, 'ref) mem\text{-}store\text{-}action \Rightarrow bool$  **where**  
 $mut\text{-}writes\ w \equiv case\ w\ of\ mw\text{-}Mutate\ -\ -\ - \Rightarrow True \mid mw\text{-}Mark\ -\ - \Rightarrow True \mid - \Rightarrow False$

**definition**  $tso\text{-}store\text{-}inv :: ('field, 'mut, 'payload, 'ref) lsts\text{-}pred$  **where**  
 $tso\text{-}store\text{-}inv =$   
 $((\forall w. \ tso\text{-}pending\text{-}store\ gc \quad w \longrightarrow \langle gc\text{-}writes\ w \rangle)$   
 $\wedge (\forall m\ w. \ tso\text{-}pending\text{-}store\ (mutator\ m)\ w \longrightarrow \langle mut\text{-}writes\ w \rangle))$

## 4.6 The global invariants collected

**definition**  $invs :: ('field, 'mut, 'payload, 'ref) lsts\text{-}pred$  **where**  
 $invs =$   
 $(handshake\text{-}phase\text{-}inv$   
 $\wedge phase\text{-}rel\text{-}inv$   
 $\wedge strong\text{-}tricolour\text{-}inv$   
 $\wedge sys\text{-}phase\text{-}inv$   
 $\wedge tso\text{-}store\text{-}inv$   
 $\wedge valid\text{-}refs\text{-}inv$   
 $\wedge valid\text{-}W\text{-}inv$   
 $\wedge mutators\text{-}phase\text{-}inv$   
 $\wedge fA\text{-}rel\text{-}inv \wedge fM\text{-}rel\text{-}inv)$

## 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  $sys\text{-}fM$ , 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**  $gc\text{-}system =$   
**fixes**  $initial\text{-}mark :: gc\text{-}mark$   
**begin**

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

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

**definition**  $sys\text{-}initial\text{-}state :: ('field, 'mut, 'payload, 'ref) lst\text{-}pred$  **where**  
 $sys\text{-}initial\text{-}state\ s =$   
 $((\forall m. \neg hs\text{-}pending\ s\ m \wedge ghost\text{-}hs\text{-}in\text{-}sync\ s\ m)$   
 $\wedge ghost\text{-}hs\text{-}phase\ s = hp\text{-}IdleMarkSweep \wedge hs\text{-}type\ s = ht\text{-}GetRoots$   
 $\wedge obj\text{-}mark\ 'ran\ (heap\ s) \subseteq \{initial\text{-}mark\})$

$\wedge fA\ s = \text{initial-mark}$   
 $\wedge fM\ s = \text{initial-mark}$   
 $\wedge \text{phase}\ s = \text{ph-Idle}$   
 $\wedge \text{ghost-honorary-grey}\ s = \{\}$   
 $\wedge W\ s = \{\}$   
 $\wedge (\forall p. \text{mem-store-buffers}\ s\ p = [])$   
 $\wedge \text{mem-lock}\ s = \text{None}$

## abbreviation

$\text{root-reachable}\ y \equiv \exists m\ x. \langle x \rangle \in \text{mut-m.mut-roots}\ m \wedge x \text{ reaches } y$

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

**definition**  $\text{gc-system-init} :: ('field, 'mut, 'payload, 'ref)\ \text{lsts-pred}$  **where**  
 $\text{gc-system-init} =$   
 $((\lambda s. \text{gc-initial-state}\ (s\ gc))$   
 $\wedge (\lambda s. \forall m. \text{mut-initial-state}\ (s\ (\text{mutator}\ m)))$   
 $\wedge (\lambda s. \text{sys-initial-state}\ (s\ sys))$   
 $\wedge \text{valid-refs})$

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

**abbreviation**  $\text{gc-system} :: ('field, 'mut, 'payload, 'ref)\ \text{gc-system}$  **where**  
 $\text{gc-system} \equiv (\text{PGMs} = \text{gc-coms}, \text{INIT} = \text{gc-system-init}, \text{FAIR} = \langle \text{True} \rangle)$

**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**  $\text{tso-lock-locs} :: \text{location set}$  **where**  
 $\text{tso-lock-locs} = (\bigcup l \in \{ \text{"mo-co-cmark"}, \text{"mo-co-ctest"}, \text{"mo-co-mark"}, \text{"mo-co-unlock"} \}. \text{suffixed } l)$

**definition**  $\text{tso-lock-invL} :: ('field, 'mut, 'payload, 'ref)\ \text{gc-pred}$  **where**  
 $[\text{inv}]: \text{tso-lock-invL} =$   
 $(\text{atS-gc}\ \text{tso-lock-locs} \quad (\text{tso-locked-by}\ gc))$   
 $\wedge \text{atS-gc}\ (\neg \text{tso-lock-locs})\ (\neg \text{tso-locked-by}\ gc))$

**end**

**context**  $\text{mut-m}$   
**begin**

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

**locset-definition**  $\text{tso-lock-locs} =$   
 $(\bigcup l \in \{ \text{"mo-co-cmark"}, \text{"mo-co-ctest"}, \text{"mo-co-mark"}, \text{"mo-co-unlock"} \}. \text{suffixed } l)$

**definition**  $\text{tso-lock-invL} :: ('field, 'mut, 'payload, 'ref)\ \text{gc-pred}$  **where**  
 $[\text{inv}]: \text{tso-lock-invL} =$   
 $(\text{atS-mut}\ \text{tso-lock-locs} \quad (\text{tso-locked-by}\ (\text{mutator}\ m)))$   
 $\wedge \text{atS-mut}\ (\neg \text{tso-lock-locs})\ (\neg \text{tso-locked-by}\ (\text{mutator}\ m)))$

**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* =

( $\bigcup l \in \text{hs-noop-prefixes. } \text{prefixed } l - (\text{suffixed } "-noop-mfence" \cup \text{suffixed } "-noop-init-type")$ )

**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* = ( $\bigcup l \in \text{hs-prefixes. } \text{prefixed } (l @ "-init-loop")$ )

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

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

**locset-definition** *hs-none-pending-locs* = – (*hs-init-loop-locs* ∪ *hs-done-locs*)

**locset-definition** *hs-in-sync-locs* =

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

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

**locset-definition** *hs-out-of-sync-locs* =

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



**locset-definition** *hs-mut-in-muts-locs* =  
 $(\bigcup l \in \text{hs-prefixes}. \{l @ \text{"-init-loop-set-pending"}, l @ \text{"-init-loop-done"}\})$

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

**locset-definition** *hs-init-loop-not-done-locs* =  
 $(\text{hs-init-loop-locs} - (\bigcup l \in \text{hs-prefixes}. \{l @ \text{"-init-loop-done"}\}))$

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

[inv]: *handshake-invL* =  
 $(\text{atS-gc hs-noop-locs} \quad (\text{sys-hs-type} = \langle \text{ht-NOOP} \rangle)$   
 $\wedge \text{atS-gc hs-get-roots-locs} \quad (\text{sys-hs-type} = \langle \text{ht-GetRoots} \rangle)$   
 $\wedge \text{atS-gc hs-get-work-locs} \quad (\text{sys-hs-type} = \langle \text{ht-GetWork} \rangle)$   
 $\wedge \text{atS-gc hs-mut-in-muts-locs} \quad (\text{gc-mut} \in \text{gc-muts})$   
 $\wedge \text{atS-gc hs-init-loop-locs} \quad (\forall m. \langle m \rangle \in \text{gc-muts} \longrightarrow \text{sys-hs-pending } m$   
 $\quad \vee \text{sys-ghost-hs-in-sync } m)$   
 $\wedge \text{atS-gc hs-init-loop-not-done-locs} \quad (\forall m. \langle m \rangle \in \text{gc-muts} \longrightarrow \neg \text{sys-hs-pending } m$   
 $\quad \wedge \neg \text{sys-ghost-hs-in-sync } m)$   
 $\wedge \text{atS-gc hs-init-loop-done-locs} \quad ( (\text{sys-hs-pending } \$ \text{gc-mut}$   
 $\quad \vee \text{sys-ghost-hs-in-sync } \$ \text{gc-mut})$   
 $\quad \wedge (\forall m. \langle m \rangle \in \text{gc-muts} \wedge \langle m \rangle \neq \text{gc-mut}$   
 $\quad \longrightarrow \neg \text{sys-hs-pending } m$   
 $\quad \wedge \neg \text{sys-ghost-hs-in-sync } m) )$   
 $\wedge \text{atS-gc hs-done-locs} \quad (\forall m. \text{sys-hs-pending } m \vee \text{sys-ghost-hs-in-sync } m)$   
 $\wedge \text{atS-gc hs-done-loop-locs} \quad (\forall m. \neg \langle m \rangle \in \text{gc-muts} \longrightarrow \neg \text{sys-hs-pending } m)$   
 $\wedge \text{atS-gc hs-none-pending-locs} \quad (\forall m. \neg \text{sys-hs-pending } m)$   
 $\wedge \text{atS-gc hs-in-sync-locs} \quad (\forall m. \text{sys-ghost-hs-in-sync } m)$   
 $\wedge \text{atS-gc hs-out-of-sync-locs} \quad (\forall m. \neg \text{sys-hs-pending } m$   
 $\quad \wedge \neg \text{sys-ghost-hs-in-sync } m)$   
 $\wedge \text{atS-gc hp-Idle-locs} \quad (\text{sys-ghost-hs-phase} = \langle \text{hp-Idle} \rangle)$   
 $\wedge \text{atS-gc hp-IdleInit-locs} \quad (\text{sys-ghost-hs-phase} = \langle \text{hp-IdleInit} \rangle)$   
 $\wedge \text{atS-gc hp-InitMark-locs} \quad (\text{sys-ghost-hs-phase} = \langle \text{hp-InitMark} \rangle)$   
 $\wedge \text{atS-gc hp-IdleMarkSweep-locs} \quad (\text{sys-ghost-hs-phase} = \langle \text{hp-IdleMarkSweep} \rangle)$   
 $\wedge \text{atS-gc hp-Mark-locs} \quad (\text{sys-ghost-hs-phase} = \langle \text{hp-Mark} \rangle))$

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* =  
 $(\text{idle-locs} - \{ \text{idle-noop-mfence} \})$   
 $\cup (\text{init-locs} - \{ \text{init-noop-mfence} \})$   
 $\cup (\text{mark-locs} - \{ \text{mark-load-fM}, \text{mark-store-fA}, \text{mark-noop-mfence} \})$

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

[inv]: *phase-invL* =  
 $(\text{atS-gc idle-locs} \quad (\text{gc-phase} = \langle \text{ph-Idle} \rangle)$   
 $\wedge \text{atS-gc init-locs} \quad (\text{gc-phase} = \langle \text{ph-Init} \rangle)$   
 $\wedge \text{atS-gc mark-locs} \quad (\text{gc-phase} = \langle \text{ph-Mark} \rangle)$   
 $\wedge \text{atS-gc sweep-locs} \quad (\text{gc-phase} = \langle \text{ph-Sweep} \rangle)$   
 $\wedge \text{atS-gc no-pending-phase-locs} \quad (\text{LIST-NULL } (\text{tso-pending-phase } \text{gc})))$

**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 ( $\lambda(-::unit) s. s\ p$ ) 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  $\equiv$  ( $(p\text{-cas-mark} \neq p\text{-mark} \vee p\text{-tso-no-pending-mark}) \longrightarrow \text{marked } \$ p\text{-the-ref}$ )
     $\wedge$  (tso-pending-mark p  $\in$  ( $\lambda s. \{[], [mw\text{-Mark } (p\text{-the-ref } s) (p\text{-fM } s)]\}$ ))

```

```

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

```

```

abbreviation (input)
  p-cas-mark-inv  $\equiv$  ( $\lambda s. \text{obj-at } (\lambda \text{obj}. \text{Some } (\text{obj-mark } \text{obj}) = p\text{-cas-mark } s) (p\text{-the-ref } s) s)$ 

```

```

abbreviation (input) p-valid-fM  $\equiv$  p-fM = sys-fM

```

```

abbreviation (input)
  p-ghg-eq-ref  $\equiv$  p-ghost-honorary-grey = pred-singleton (the  $\circ$  p-ref)

```

```

abbreviation (input)
  p-ghg-inv  $\equiv$  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"  $\langle \text{True} \rangle$ )

```

$\wedge \text{at-p } \text{"-mo-mark"} \quad (p\text{-valid-ref})$   
 $\wedge \text{at-p } \text{"-mo-fM"} \quad (p\text{-valid-ref} \wedge p\text{-en-cond } (p\text{-mark-inv}))$   
 $\wedge \text{at-p } \text{"-mo-mtest"} \quad (p\text{-valid-ref} \wedge p\text{-en-cond } (p\text{-mark-inv} \wedge p\text{-valid-fM}))$   
 $\wedge \text{at-p } \text{"-mo-phase"} \quad (p\text{-valid-ref} \wedge p\text{-mark} \neq \text{Some} \circ p\text{-fM} \wedge p\text{-en-cond } (p\text{-mark-inv} \wedge p\text{-valid-fM}))$   
 $\wedge \text{at-p } \text{"-mo-ptest"} \quad (p\text{-valid-ref} \wedge p\text{-mark} \neq \text{Some} \circ p\text{-fM} \wedge p\text{-en-cond } (p\text{-mark-inv} \wedge p\text{-valid-fM}))$   
 $\wedge \text{at-p } \text{"-mo-co-lock"} \quad (p\text{-valid-ref} \wedge p\text{-mark-inv} \wedge p\text{-valid-fM} \wedge p\text{-mark} \neq \text{Some} \circ p\text{-fM} \wedge p\text{-tso-no-pending-mark})$   
 $\wedge \text{at-p } \text{"-mo-co-cmark"} \quad (p\text{-valid-ref} \wedge p\text{-mark-inv} \wedge p\text{-valid-fM} \wedge p\text{-mark} \neq \text{Some} \circ p\text{-fM} \wedge p\text{-tso-no-pending-mark})$   
 $\wedge \text{at-p } \text{"-mo-co-ctest"} \quad (p\text{-valid-ref} \wedge p\text{-mark-inv} \wedge p\text{-valid-fM} \wedge p\text{-mark} \neq \text{Some} \circ p\text{-fM} \wedge p\text{-cas-mark-inv} \wedge$   
 $p\text{-tso-no-pending-mark})$   
 $\wedge \text{at-p } \text{"-mo-co-mark"} \quad (p\text{-cas-mark} = p\text{-mark} \wedge p\text{-valid-ref} \wedge p\text{-valid-fM} \wedge \text{white } \$ p\text{-the-ref} \wedge p\text{-tso-no-pending-mark})$   
 $\wedge \text{at-p } \text{"-mo-co-unlock"} \quad (p\text{-ghg-inv} \wedge p\text{-valid-ref} \wedge p\text{-valid-fM} \wedge p\text{-valid-W-inv})$   
 $\wedge \text{at-p } \text{"-mo-co-won"} \quad (p\text{-ghg-inv} \wedge p\text{-valid-ref} \wedge p\text{-valid-fM} \wedge \text{marked } \$ p\text{-the-ref} \wedge p\text{-tso-no-pending-mutate})$   
 $\wedge \text{at-p } \text{"-mo-co-W"} \quad (p\text{-ghg-eq-ref} \wedge p\text{-valid-ref} \wedge p\text{-valid-fM} \wedge \text{marked } \$ p\text{-the-ref} \wedge p\text{-tso-no-pending-mutate})$

**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*  $\langle \text{True} \rangle$

**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*  $\langle \text{True} \rangle$  **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*  $\neq \langle \text{hp-Idle} \rangle$  **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*  $\neq \langle \text{hp-Idle} \rangle$  **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"  $\cup \{ \text{hs-get-roots-loop-done} \}$*

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

**locset-definition** *hs-not-hp-Idle-locs* =  
 $(\bigcup_{\text{pref} \in \text{mut-async-mark-object-prefixes.}}$

$\bigcup l \in \{ "mo-co-lock", "mo-co-cmark", "mo-co-ctest", "mo-co-mark", "mo-co-unlock", "mo-co-won", "mo-co-W" \}.$   
 $\{pref @ "-" @ l\})$

**locset-definition** *async-mo-ptest-locs* =  
 $(\bigcup pref \in mut-async-mark-object-prefixes. \{pref @ "-mo-ptest"\})$

**locset-definition** *mo-ptest-locs* =  
 $(\bigcup pref \in mut-async-mark-object-prefixes. \{pref @ "-mo-ptest"\})$

**locset-definition** *mo-valid-ref-locs* =  
 $(prefixed "store-del" \cup prefixed "store-ins" \cup \{deref-del, lop-store-ins\})$

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* =  
 $(- (\bigcup pref \in \{ "hs-get-roots-loop", "store-del", "store-ins" \}.$   
 $\bigcup l \in \{ "mo-co-unlock", "mo-co-won", "mo-co-W" \}. \{pref @ "-" @ l\}))$

**locset-definition**  
*ghost-honorary-root-empty-locs* =  
 $(- (prefixed "store-del" \cup \{lop-store-ins\} \cup prefixed "store-ins"))$

**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* =  $prefixed "store-del-mo" \cup \{lop-store-ins\}$

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

[inv]: *mark-object-invL* =

$(atS-mut\ hs-get-roots-loop-locs \quad (mut-refs \subseteq mut-roots \wedge (\forall r. \langle r \rangle \in mut-roots - mut-refs \longrightarrow marked\ r))$   
 $\wedge atS-mut\ hs-get-roots-loop-mo-locs \quad (\neg NULL\ mut-ref \wedge mut-the-ref \in mut-roots)$   
 $\wedge at-mut\ hs-get-roots-loop-done \quad (marked\ \$\ mut-the-ref)$   
 $\wedge at-mut\ hs-get-roots-loop-mo-ptest \quad (mut-phase \neq \langle ph-Idle \rangle)$   
 $\wedge at-mut\ hs-get-roots-done \quad (\forall r. \langle r \rangle \in mut-roots \longrightarrow marked\ r)$

$\wedge atS-mut\ mo-valid-ref-locs \quad ( (\neg NULL\ mut-new-ref \longrightarrow mut-the-new-ref \in mut-roots)$   
 $\wedge (mut-tmp-ref \in mut-roots) )$

$\wedge at-mut\ store-del-mo-null \quad (\neg NULL\ mut-ref \longrightarrow mut-the-ref \in mut-ghost-honorary-root)$   
 $\wedge atS-mut\ ghost-honorary-root-nonempty-locs \quad (mut-the-ref \in mut-ghost-honorary-root)$

$\wedge atS-mut\ not-idle-locs \quad (mut-phase \neq \langle ph-Idle \rangle \longrightarrow mut-ghost-hs-phase \neq \langle hp-Idle \rangle)$   
 $\wedge atS-mut\ hs-not-hp-Idle-locs \quad (mut-ghost-hs-phase \neq \langle hp-Idle \rangle)$

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

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

— insertion barrier

$\wedge$  *at-mut store-ins*  $(\langle \text{mut-ghost-hs-phase} \in \langle \{hp\text{-InitMark}, hp\text{-Mark}\} \rangle \vee \langle \text{mut-ghost-hs-phase} = \langle hp\text{-IdleMarkSweep} \rangle \wedge \text{sys-phase} \neq \langle ph\text{-Idle} \rangle \rangle) \wedge \neg \text{NULL mut-new-ref} \longrightarrow \text{marked } \$ \text{ mut-the-new-ref} )$   
 $\wedge$  *atS-mut ins-barrier-locs*  $(\langle \langle \text{mut-ghost-hs-phase} = \langle hp\text{-Mark} \rangle \vee \langle \text{mut-ghost-hs-phase} = \langle hp\text{-IdleMarkSweep} \rangle \wedge \text{sys-phase} \neq \langle ph\text{-Idle} \rangle \rangle \wedge (\lambda s. \forall \text{opt-r}'. \neg \text{tso-pending-store (mutator } m) (mw\text{-Mutate (mut-tmp-ref } s) (mut\text{-field } s) \text{ opt-r}') } s) \longrightarrow (\lambda s. \text{obj-at-field-on-heap } (\lambda r'. \text{marked } r' s) (mut\text{-tmp-ref } s) (mut\text{-field } s) s) )$   
 $\wedge$  *(mut-ref = mut-new-ref)* )  
— deletion barrier  
 $\wedge$  *atS-mut del-barrier1-locs*  $(\langle \langle \text{mut-ghost-hs-phase} = \langle hp\text{-Mark} \rangle \vee \langle \text{mut-ghost-hs-phase} = \langle hp\text{-IdleMarkSweep} \rangle \wedge \text{sys-phase} \neq \langle ph\text{-Idle} \rangle \rangle \wedge (\lambda s. \forall \text{opt-r}'. \neg \text{tso-pending-store (mutator } m) (mw\text{-Mutate (mut-tmp-ref } s) (mut\text{-field } s) \text{ opt-r}') } s) \longrightarrow (\lambda s. \text{obj-at-field-on-heap } (\lambda r. \text{mut-ref } s = \text{Some } r \vee \text{marked } r s) (mut\text{-tmp-ref } s) (mut\text{-field } s) s) )$   
 $\wedge$  *at-mut lop-store-ins*  $(\langle \langle \text{mut-ghost-hs-phase} = \langle hp\text{-Mark} \rangle \vee \langle \text{mut-ghost-hs-phase} = \langle hp\text{-IdleMarkSweep} \rangle \wedge \text{sys-phase} \neq \langle ph\text{-Idle} \rangle \rangle \wedge \neg \text{NULL mut-ref} \longrightarrow \text{marked } \$ \text{ mut-the-ref} )$   
— after *init-noop*. key: no pending white insertions *at-mut hs-noop-done* which we get from *handshake-invL*.  
 $\wedge$  *at-mut mut-load*  $(\text{mut-tmp-ref} \in \text{mut-roots})$   
 $\wedge$  *atS-mut ghost-honorary-root-empty-locs*  $(\text{EMPTY mut-ghost-honorary-root})$  )

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*) *lst-pred* **where**

*gc-W-empty-mut-inv* =  
 $((\text{EMPTY sys-W} \wedge \text{sys-ghost-hs-in-sync } m \wedge \neg \text{EMPTY (WL (mutator } m)))$   
 $\longrightarrow (\exists m'. \neg \text{sys-ghost-hs-in-sync } m' \wedge \neg \text{EMPTY (WL (mutator } m'))))$

**context** *gc*

**begin**

**locset-definition** *gc-W-empty-locs* :: *location set* **where**

*gc-W-empty-locs* =  
 $\text{idle-locs} \cup \text{init-locs} \cup \text{sweep-locs} \cup \{\text{mark-load-fM}, \text{mark-store-fA}, \text{mark-end}\}$   
 $\cup \text{prefixed "mark-noop"}$   
 $\cup \text{prefixed "mark-loop-get-roots"}$   
 $\cup \text{prefixed "mark-loop-get-work"}$

**locset-definition** *get-roots-UN-get-work-locs* = *hs-get-roots-locs*  $\cup$  *hs-get-work-locs*

**locset-definition** *black-heap-locs* =  $\{\text{sweep-idle}, \text{idle-noop-mfence}, \text{idle-noop-init-type}\}$

**locset-definition** *no-grey-refs-locs* = *black-heap-locs*  $\cup$  *sweep-locs*  $\cup$   $\{\text{mark-end}\}$

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

[*inv*]: *gc-W-empty-invL* =

$(atS\text{-}gc\text{ } get\text{-}roots\text{-}UN\text{-}get\text{-}work\text{-}locs \quad (\forall m. mut\text{-}m.gc\text{-}W\text{-}empty\text{-}mut\text{-}inv\text{-}m))$   
 $\wedge at\text{-}gc\text{ } mark\text{-}loop\text{-}get\text{-}roots\text{-}load\text{-}W \quad (EMPTY\text{ } sys\text{-}W \longrightarrow no\text{-}grey\text{-}refs)$   
 $\wedge at\text{-}gc\text{ } mark\text{-}loop\text{-}get\text{-}work\text{-}load\text{-}W \quad (EMPTY\text{ } sys\text{-}W \longrightarrow no\text{-}grey\text{-}refs)$   
 $\wedge at\text{-}gc\text{ } mark\text{-}loop \quad (EMPTY\text{ } gc\text{-}W \longrightarrow no\text{-}grey\text{-}refs)$   
 $\wedge atS\text{-}gc\text{ } no\text{-}grey\text{-}refs\text{-}locs \quad no\text{-}grey\text{-}refs$   
 $\wedge atS\text{-}gc\text{ } gc\text{-}W\text{-}empty\text{-}locs \quad (EMPTY\text{ } gc\text{-}W))$

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**

$[inv]: sweep\text{-}loop\text{-}invL =$   
 $(at\text{-}gc\text{ } sweep\text{-}loop\text{-}check \quad ( (\neg NULL\text{ } gc\text{-}mark \longrightarrow (\lambda s. obj\text{-}at\text{ } (\lambda obj. Some\text{ } (obj\text{-}mark\text{ } obj) = gc\text{-}mark\text{ } s)$   
 $(gc\text{-}tmp\text{-}ref\text{ } s)\text{ } s))$   
 $\wedge ( NULL\text{ } gc\text{-}mark \wedge valid\text{-}ref\text{ } \$\text{ } gc\text{-}tmp\text{-}ref \longrightarrow marked\text{ } \$\text{ } gc\text{-}tmp\text{-}ref ) )$   
 $\wedge at\text{-}gc\text{ } sweep\text{-}loop\text{-}free \quad ( \neg NULL\text{ } gc\text{-}mark \wedge the\text{ } \circ\text{ } gc\text{-}mark \neq gc\text{-}fM \wedge (\lambda s. obj\text{-}at\text{ } (\lambda obj. Some$   
 $(obj\text{-}mark\text{ } obj) = gc\text{-}mark\text{ } s)\text{ } (gc\text{-}tmp\text{-}ref\text{ } s)\text{ } s) )$   
 $\wedge at\text{-}gc\text{ } sweep\text{-}loop\text{-}ref\text{-}done \quad (valid\text{-}ref\text{ } \$\text{ } gc\text{-}tmp\text{-}ref \longrightarrow marked\text{ } \$\text{ } gc\text{-}tmp\text{-}ref)$   
 $\wedge atS\text{-}gc\text{ } sweep\text{-}loop\text{-}locs \quad (\forall r. \neg \langle r \rangle \in gc\text{-}refs \wedge valid\text{-}ref\text{ } r \longrightarrow marked\text{ } r)$   
 $\wedge atS\text{-}gc\text{ } black\text{-}heap\text{-}locs \quad (\forall r. valid\text{-}ref\text{ } r \longrightarrow marked\text{ } r)$   
 $\wedge atS\text{-}gc\text{ } sweep\text{-}loop\text{-}not\text{-}choose\text{-}ref\text{-}locs\text{ } (gc\text{-}tmp\text{-}ref \in gc\text{-}refs))$

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**

$obj\text{-}fields\text{-}marked =$   
 $(\forall f. \langle f \rangle \in (-\text{ } gc\text{-}field\text{-}set) \longrightarrow (\lambda s. obj\text{-}at\text{-}field\text{-}on\text{-}heap\text{ } (\lambda r. marked\text{ } r\text{ } s)\text{ } (gc\text{-}tmp\text{-}ref\text{ } s)\text{ } f\text{ } s))$

**locset-definition** *mark-loop-mo-locs* = *prefixed "mark-loop-mo"*

**locset-definition** *obj-fields-marked-good-ref-locs* = *mark-loop-mo-locs*  $\cup$  {*mark-loop-mark-field-done*}

**locset-definition**

$ghost\text{-}honorary\text{-}grey\text{-}empty\text{-}locs =$   
 $(-\text{ } \{ mark\text{-}loop\text{-}mo\text{-}co\text{-}unlock, mark\text{-}loop\text{-}mo\text{-}co\text{-}won, mark\text{-}loop\text{-}mo\text{-}co\text{-}W \})$

**locset-definition**

$obj\text{-}fields\text{-}marked\text{-}locs =$   
 $\{ mark\text{-}loop\text{-}mark\text{-}object\text{-}loop, mark\text{-}loop\text{-}mark\text{-}choose\text{-}field, mark\text{-}loop\text{-}mark\text{-}deref, mark\text{-}loop\text{-}mark\text{-}field\text{-}done,$   
 $mark\text{-}loop\text{-}blacken \}$   
 $\cup mark\text{-}loop\text{-}mo\text{-}locs$

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

$[inv]: obj\text{-}fields\text{-}marked\text{-}invL =$   
 $(atS\text{-}gc\text{ } obj\text{-}fields\text{-}marked\text{-}locs \quad (obj\text{-}fields\text{-}marked \wedge gc\text{-}tmp\text{-}ref \in gc\text{-}W)$   
 $\wedge atS\text{-}gc\text{ } obj\text{-}fields\text{-}marked\text{-}good\text{-}ref\text{-}locs \quad (\lambda s. obj\text{-}at\text{-}field\text{-}on\text{-}heap\text{ } (\lambda r. gc\text{-}ref\text{ } s = Some\text{ } r \vee marked\text{ } r\text{ } s)\text{ } (gc\text{-}tmp\text{-}ref$   
 $s)\text{ } (gc\text{-}field\text{ } s)\text{ } s)$   
 $\wedge atS\text{-}gc\text{ } mark\text{-}loop\text{-}mo\text{-}locs \quad (\forall y. \neg NULL\text{ } gc\text{-}ref \wedge (\lambda s. ((gc\text{-}the\text{-}ref\text{ } s)\text{ } reaches\text{ } y)\text{ } s) \longrightarrow valid\text{-}ref\text{ } y)$   
 $\wedge at\text{-}gc\text{ } mark\text{-}loop\text{-}fields \quad (gc\text{-}tmp\text{-}ref \in gc\text{-}W)$   
 $\wedge at\text{-}gc\text{ } mark\text{-}loop\text{-}mark\text{-}field\text{-}done \quad (\neg NULL\text{ } gc\text{-}ref \longrightarrow marked\text{ } \$\text{ } gc\text{-}the\text{-}ref)$   
 $\wedge at\text{-}gc\text{ } mark\text{-}loop\text{-}blacken \quad (EMPTY\text{ } gc\text{-}field\text{-}set)$

$\wedge \text{atS-gc ghost-honorary-grey-empty-locs } (EMPTY \text{ gc-ghost-honorary-grey}))$

**end**

## 5.6 The local innvariants collected

**definition** (in *gc*) *invsL* :: ('field, 'mut, 'payload, 'ref) *gc-pred* **where**

*invsL* =  
 (fM-fA-invL  
 $\wedge$  gc-mark.mark-object-invL  
 $\wedge$  gc-W-empty-invL  
 $\wedge$  handshake-invL  
 $\wedge$  obj-fields-marked-invL  
 $\wedge$  phase-invL  
 $\wedge$  sweep-loop-invL  
 $\wedge$  tso-lock-invL)

**definition** (in *mut-m*) *invsL* :: ('field, 'mut, 'payload, 'ref) *gc-pred* **where**

*invsL* =  
 (mark-object-invL  
 $\wedge$  mut-get-roots.mark-object-invL *m*  
 $\wedge$  mut-store-ins.mark-object-invL *m*  
 $\wedge$  mut-store-del.mark-object-invL *m*  
 $\wedge$  handshake-invL  
 $\wedge$  tso-lock-invL)

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

*invsL* = (*gc.invsL*  $\wedge$  ( $\forall m. \text{mut-m.invsL } m$ ))

## 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*  $\Rightarrow$  'mut *process-name*  $\Rightarrow$  ('field, 'mut, 'payload, 'ref) *gc-pred*  $\Rightarrow$  bool ( $\{\{-\} - \{\{-\}\}$ )

**where**

$\{P\} p \{Q\} = (\forall (c, \text{afts}) \in \text{vcg-fragments } (gc\text{-coms } p). gc\text{-coms}, p, \text{afts} \vdash \{P\} c \{Q\})$

**abbreviation**

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

**where**

$\{P\} p \equiv \{P\} p \{P\}$

**lemma** *valid-pre*:

**assumes**  $\{Q\} p \{R\}$   
**assumes**  $\bigwedge s. P s \Longrightarrow Q s$   
**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  $\llbracket P \wedge P' \rrbracket p \llbracket Q \wedge Q' \rrbracket$ 
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  $\bigwedge x. \llbracket P x \rrbracket p \llbracket Q x \rrbracket$ 
shows  $\llbracket \lambda s. \forall x. P x s \rrbracket p \llbracket \lambda s. \forall x. Q x s \rrbracket$ 
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*:

```

 $\llbracket (\llbracket l \rrbracket \text{Response action, afts}) \in \text{fragments (gc-coms p) } \{\}; v \in \text{action } x s;$ 
 $\llbracket p = \text{sys}; ?P \rrbracket \implies Q \rrbracket \implies Q$ 
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*:

```

 $\llbracket (\llbracket l \rrbracket \text{Response action, afts}) \in \text{fragments (gc-coms p) } \{\}; v \in \text{action (pname, req) } s;$ 
 $\llbracket p = \text{sys}; \text{case-request-op } ?P1 ?P2 ?P3 ?P4 ?P5 ?P6 ?P7 ?P8 ?P9 ?P10 ?P11 ?P12 ?P13 ?P14 \text{ req} \rrbracket \implies$ 
 $Q \rrbracket \implies Q$ 
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*:

```

 $\llbracket (\llbracket l \rrbracket \text{Response action, afts}) \in \text{fragments (gc-coms p) } \{\}; v \in \text{action } x s;$ 
 $\llbracket p = \text{sys}; \text{case-request-op } ?P1 ?P2 ?P3 ?P4 ?P5 ?P6 ?P7 ?P8 ?P9 ?P10 ?P11 ?P12 ?P13 ?P14 \text{ (snd } x) \rrbracket$ 
 $\implies Q \rrbracket \implies Q$ 
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*:

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

**lemma** *schematic-prem*:  $\llbracket Q \implies P; Q \rrbracket \implies P$

**by** *blast*

**lemma** *TrueE*:  $\llbracket True; P \rrbracket \implies P$

**by** *blast*

**lemma** *thin-locs-pre-discardE*:

$\llbracket at\ p\ l'\ s \longrightarrow P; at\ p\ l\ s; l' \neq l; Q \rrbracket \implies Q$   
 $\llbracket atS\ p\ ls\ s \longrightarrow P; at\ p\ l\ s; l \notin ls; Q \rrbracket \implies Q$

**unfolding** *atS-def* **by** *blast+*

**lemma** *thin-locs-pre-keep-atE*:

$\llbracket at\ p\ l\ s \longrightarrow P; at\ p\ l\ s; P \implies Q \rrbracket \implies Q$   
**by** *blast*

**lemma** *thin-locs-pre-keep-atSE*:

$\llbracket atS\ p\ ls\ s \longrightarrow P; at\ p\ l\ s; l \in ls; P \implies Q \rrbracket \implies Q$   
**unfolding** *atS-def* **by** *blast*

**lemma** *thin-locs-post-discardE*:

$\llbracket AT\ s' = (AT\ s)(p := lfn, q := lfn'); l' \notin lfn; p \neq q \rrbracket \implies at\ p\ l'\ s' \longrightarrow P$   
 $\llbracket AT\ s' = (AT\ s)(p := lfn); l' \notin lfn \rrbracket \implies at\ p\ l'\ s' \longrightarrow P$   
 $\llbracket AT\ s' = (AT\ s)(p := lfn, q := lfn'); \bigwedge l. l \in lfn \implies l \notin ls; p \neq q \rrbracket \implies atS\ p\ ls\ s' \longrightarrow P$   
 $\llbracket AT\ s' = (AT\ s)(p := lfn); \bigwedge l. l \in lfn \implies l \notin ls \rrbracket \implies atS\ p\ ls\ s' \longrightarrow P$

**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*:

$\llbracket (L \longrightarrow P) \wedge R; R \implies Q \rrbracket \implies (L \longrightarrow P) \wedge Q$   
 $L \longrightarrow P \implies L \longrightarrow P$   
**by** *blast+*

**lemma** *thin-locs-post-keepE*:

$\llbracket P \wedge R; R \implies Q \rrbracket \implies (L \longrightarrow P) \wedge Q$   
 $P \implies L \longrightarrow P$   
**by** *blast+*

**lemma** *ni-thin-locs-discardE*:

$\llbracket at\ proc\ l\ s \longrightarrow P; AT\ s' = (AT\ s)(p := lfn, q := lfn'); at\ proc\ l'\ s'; l \neq l'; proc \neq p; proc \neq q; Q \rrbracket \implies Q$

$\llbracket \text{at proc } l \ s \longrightarrow P; AT \ s' = (AT \ s)(p := \text{lf}n); \text{at proc } l' \ s'; l \neq l'; \text{proc} \neq p; Q \rrbracket \Longrightarrow Q$   
 $\llbracket \text{atS proc } ls \ s \longrightarrow P; AT \ s' = (AT \ s)(p := \text{lf}n, q := \text{lf}n'); \text{at proc } l' \ s'; l' \notin ls; \text{proc} \neq p; \text{proc} \neq q; Q \rrbracket \Longrightarrow Q$   
 $\llbracket \text{atS proc } ls \ s \longrightarrow P; AT \ s' = (AT \ s)(p := \text{lf}n); \text{at proc } l' \ s'; l' \notin ls; \text{proc} \neq p; Q \rrbracket \Longrightarrow Q$   
 $\llbracket \text{at proc } l \ s \longrightarrow P; AT \ s' = (AT \ s)(p := \text{lf}n, q := \text{lf}n'); \text{atS proc } ls' \ s'; l \notin ls'; \text{proc} \neq p; \text{proc} \neq q; Q \rrbracket \Longrightarrow Q$   
 $\llbracket \text{at proc } l \ s \longrightarrow P; AT \ s' = (AT \ s)(p := \text{lf}n); \text{atS proc } ls' \ s'; l \notin ls'; \text{proc} \neq p; Q \rrbracket \Longrightarrow Q$

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

**lemma** *ni-thin-locs-keep-atE*:

$\llbracket \text{at proc } l \ s \longrightarrow P; AT \ s' = (AT \ s)(p := \text{lf}n, q := \text{lf}n'); \text{at proc } l \ s'; \text{proc} \neq p; \text{proc} \neq q; P \Longrightarrow Q \rrbracket \Longrightarrow Q$   
 $\llbracket \text{at proc } l \ s \longrightarrow P; AT \ s' = (AT \ s)(p := \text{lf}n); \text{at proc } l \ s'; \text{proc} \neq p; P \Longrightarrow Q \rrbracket \Longrightarrow Q$

**by** (*auto simp: fun-upd-apply*)

**lemma** *ni-thin-locs-keep-atSE*:

$\llbracket \text{atS proc } ls \ s \longrightarrow P; AT \ s' = (AT \ s)(p := \text{lf}n, q := \text{lf}n'); \text{at proc } l' \ s'; l' \in ls; \text{proc} \neq p; \text{proc} \neq q; P \Longrightarrow Q \rrbracket \Longrightarrow Q$   
 $\llbracket \text{atS proc } ls \ s \longrightarrow P; AT \ s' = (AT \ s)(p := \text{lf}n); \text{at proc } l' \ s'; l' \in ls; \text{proc} \neq p; P \Longrightarrow Q \rrbracket \Longrightarrow Q$   
 $\llbracket \text{atS proc } ls \ s \longrightarrow P; AT \ s' = (AT \ s)(p := \text{lf}n, q := \text{lf}n'); \text{atS proc } ls' \ s'; ls' \subseteq ls; \text{proc} \neq p; \text{proc} \neq q; P \Longrightarrow Q \rrbracket \Longrightarrow Q$   
 $\llbracket \text{atS proc } ls \ s \longrightarrow P; AT \ s' = (AT \ s)(p := \text{lf}n); \text{atS proc } ls' \ s'; ls' \subseteq ls; \text{proc} \neq p; P \Longrightarrow Q \rrbracket \Longrightarrow Q$

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

**lemma** *loc-mem-tac-intros*:

$\llbracket c \notin A; c \notin B \rrbracket \Longrightarrow c \notin A \cup B$   
 $c \neq d \Longrightarrow c \notin \{d\}$   
 $c \notin A \Longrightarrow c \in - \ A$   
 $c \in A \Longrightarrow c \notin - \ A$   
 $A \subseteq A$

**by** *blast+*

**lemmas** *loc-mem-tac-elim* =

*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 subgoal 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-elim}))
      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) ))))
    end;

  (* 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 (* imply 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 process-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-NEW 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
            (* imply 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)
            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 specialise 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-chain.
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 (λp 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=λfr. 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) (λobj. 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**  $P = \lambda fr. Option.bind (heap (fr (s sys)) r) (\lambda obj. obj\text{-}fields\ obj\ f) = ?rhs$   
**and**  $Q = \lambda w. \forall r\ f\ r'. w \neq mw\text{-}Mutate\ r\ f\ r'$ ])  
**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**  $P = \lambda fr. map\text{-}option\ obj\text{-}mark (heap (fr (s sys)) r) = map\text{-}option\ obj\text{-}mark (sys\text{-}heap\ s\ r)$   
**and**  $Q = \lambda w. \forall fl. w \neq mw\text{-}Mark\ r\ fl$ ])  
**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**  $P = \lambda fr. phase (fr (s sys)) = sys\text{-}phase\ s$  **and**  $Q = \lambda w. \forall ph. w \neq mw\text{-}Phase\ ph$ ])  
**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**  $P = \lambda fr. fM (fr (s sys)) = sys\text{-}fM\ s$  **and**  $Q = \lambda w. \forall fl. w \neq mw\text{-}fM\ fl$ ])  
**apply** (*auto simp: do-store-action-def filter-empty-conv*  
*split: mem-store-action.splits*)  
**done**

**lemma** *tso-store-refs-simps[simp]:*  
 $mut\text{-}m.tso\text{-}store\text{-}refs\ m\ (s(mutator\ m' := s\ (mutator\ m')(\text{roots} := \text{roots}')))$   
 $= mut\text{-}m.tso\text{-}store\text{-}refs\ m\ s$   
 $mut\text{-}m.tso\text{-}store\text{-}refs\ m\ (s(mutator\ m' := s\ (mutator\ m')(\text{ghost-honorary-root} := \{\}),$   
 $sys := s\ sys(\text{mem-store-buffers} := (\text{mem-store-buffers}\ (s\ sys))(mutator\ m' :=$   
 $sys\text{-}mem\text{-}store\text{-}buffers\ (mutator\ m')\ s\ @\ [mw\text{-}Mutate\ r\ f\ opt\text{-}r'])))$   
 $= mut\text{-}m.tso\text{-}store\text{-}refs\ m\ s\ \cup\ (if\ m' = m\ then\ store\text{-}refs\ (mw\text{-}Mutate\ r\ f\ opt\text{-}r')\ else\ \{\})$   
 $mut\text{-}m.tso\text{-}store\text{-}refs\ m\ (s(sys := s\ sys(\text{mem-store-buffers} := (\text{mem-store-buffers}\ (s\ sys))(mutator\ m' := sys\text{-}mem\text{-}store\text{-}buffers\ (mutator\ m')\ s\ @\ [mw\text{-}Mutate\ Payload\ r\ f\ pl])))$   
 $= mut\text{-}m.tso\text{-}store\text{-}refs\ m\ s\ \cup\ (if\ m' = m\ then\ store\text{-}refs\ (mw\text{-}Mutate\ Payload\ r\ f\ pl)\ else\ \{\})$   
 $mut\text{-}m.tso\text{-}store\text{-}refs\ m\ (s(sys := s\ sys(\text{heap} := (\text{sys-heap}\ s)(r' := None))))$   
 $= mut\text{-}m.tso\text{-}store\text{-}refs\ m\ s$   
 $mut\text{-}m.tso\text{-}store\text{-}refs\ m\ (s(mutator\ m' := s\ (mutator\ m')(\text{roots} := insert\ r\ (\text{roots}\ (s\ (mutator\ m')))), sys := s$   
 $sys(\text{heap} := (\text{sys-heap}\ s)(r \mapsto obj))))$

$= \text{mut-m.tso-store-refs } m \ s$   
 $\text{mut-m.tso-store-refs } m \ (s(\text{mutator } m' := s(\text{mutator } m')(\text{ghost-honorary-root} := \text{Option.set-option } \text{opt-r}', \text{ref} := \text{opt-r}')))$   
 $= \text{mut-m.tso-store-refs } m \ s$   
 $\text{mut-m.tso-store-refs } m \ (s(\text{sys} := s \ \text{sys}(\text{heap} := (\text{sys-heap } s)(r := \text{map-option } (\lambda \text{obj. } \text{obj}(\text{obj-fields} := (\text{obj-fields } \text{obj})(f := \text{opt-r}')))(\text{sys-heap } s \ r))),$   
 $\text{mem-store-buffers} := (\text{mem-store-buffers } (s \ \text{sys}))(p := \text{ws}))$   
 $= (\text{if } p = \text{mutator } m \text{ then } \bigcup w \in \text{set } \text{ws. store-refs } w \text{ else } \text{mut-m.tso-store-refs } m \ s)$   
 $\text{mut-m.tso-store-refs } m \ (s(\text{sys} := s \ \text{sys}(\text{heap} := (\text{sys-heap } s)(r := \text{map-option } (\lambda \text{obj. } \text{obj}(\text{obj-payload} := (\text{obj-payload } \text{obj})(f := \text{pl}')))(\text{sys-heap } s \ r))),$   
 $\text{mem-store-buffers} := (\text{mem-store-buffers } (s \ \text{sys}))(p := \text{ws}))$   
 $= (\text{if } p = \text{mutator } m \text{ then } \bigcup w \in \text{set } \text{ws. store-refs } w \text{ else } \text{mut-m.tso-store-refs } m \ s)$   
 $\text{sys-mem-store-buffers } p \ s = \text{mw-Mark } r \ \text{fl} \ \# \ \text{ws}$   
 $\Rightarrow \text{mut-m.tso-store-refs } m \ (s(\text{sys} := s \ \text{sys}(\text{heap} := (\text{sys-heap } s)(r := \text{map-option } (\text{obj-mark-update } (\lambda \cdot. \text{fl})))$   
 $(\text{sys-heap } s \ r)), \text{mem-store-buffers} := (\text{mem-store-buffers } (s \ \text{sys}))(p := \text{ws})))$   
 $= \text{mut-m.tso-store-refs } m \ s$   
**unfolding**  $\text{mut-m.tso-store-refs-def}$  **by**  $(\text{auto simp: fun-upd-apply})$

**lemma**  $\text{fold-stores-points-to}[\text{rule-format, simplified conj-explode}]$ :

$\text{heap } (\text{fold-stores } \text{ws } (s \ \text{sys})) \ r = \text{Some } \text{obj} \wedge \text{obj-fields } \text{obj} \ f = \text{Some } r'$   
 $\rightarrow (r \text{ points-to } r') \ s \vee (\exists w \in \text{set } \text{ws. } r' \in \text{store-refs } w) \ (\text{is } ?P \ (\text{fold-stores } \text{ws}) \ \text{obj})$

**unfolding**  $\text{fold-stores-def}$

**apply**  $(\text{rule spec}[OF \text{fold-invariant}[\text{where } P = \lambda \text{fr. } \forall \text{obj. } ?P \ \text{fr } \text{obj} \text{ and } Q = \lambda w. w \in \text{set } \text{ws}]])$

**apply**  $\text{fastforce}$

**apply**  $(\text{fastforce simp: ran-def split: obj-at-splits})$

**apply**  $\text{clarsimp}$

**apply**  $(\text{drule } (1) \ \text{bspec})$

**apply**  $(\text{clarsimp simp: fun-upd-apply split: mem-store-action.split-asm if-splits})$

**done**

**lemma**  $\text{points-to-Mutate}$ :

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

**unfolding**  $\text{ran-def}$  **by**  $(\text{auto simp: fun-upd-apply split: obj-at-splits})$

## 7.2 FIXME mutator handshake facts

**lemma** —  $\text{Sanity}$

$hp' = \text{hs-step } hp \Rightarrow \exists in' \ ht. (in', ht, hp', hp) \in \text{hp-step-rel}$

**by**  $(\text{cases } hp) \ (\text{auto simp: hp-step-rel-def})$

**lemma** —  $\text{Sanity}$

$(\text{False}, ht, hp', hp) \in \text{hp-step-rel} \Rightarrow hp' = \text{hp-step } ht \ hp$

**by**  $(\text{cases } ht) \ (\text{auto simp: hp-step-rel-def})$

**lemma**  $(\text{in } \text{mut-m}) \ \text{handshake-phase-invD}$ :

**assumes**  $\text{handshake-phase-inv } s$

**shows**  $(\text{sys-ghost-hs-in-sync } m \ s, \text{sys-hs-type } s, \text{sys-ghost-hs-phase } s, \text{mut-ghost-hs-phase } s) \in \text{hp-step-rel}$

$\wedge (\text{sys-hs-pending } m \ s \rightarrow \neg \text{sys-ghost-hs-in-sync } m \ s)$

**using**  $\text{assms}$  **unfolding**  $\text{handshake-phase-inv-def}$  **by**  $\text{simp}$

**lemma**  $\text{handshake-in-syncD}$ :

$\llbracket \text{All } (\text{ghost-hs-in-sync } (s \ \text{sys})); \text{handshake-phase-inv } s \rrbracket$

$\Rightarrow \forall m'. \text{mut-m.mut-ghost-hs-phase } m' \ s = \text{sys-ghost-hs-phase } s$

**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** ( $\forall m. \text{sys-ghost-hs-in-sync } m \ s, \text{sys-ghost-hs-phase } s, \text{gc-phase } s, \text{sys-phase } s, \text{tso-pending-phase } gc \ s) \in \text{phase-rel}$

**using** *assms* **unfolding** *phase-rel-inv-def* **by** *simp*

**lemma** *mut-m-not-idle-no-fM-write*:

$\llbracket \text{ghost-hs-phase } (s \text{ (mutator } m)) \neq \text{hp-Idle}; \text{fM-rel-inv } s; \text{handshake-phase-inv } s; \text{tso-store-inv } s; p \neq \text{sys} \rrbracket$   
 $\implies \neg \text{sys-mem-store-buffers } p \ s = \text{mw-fM } fl \ \# \ ws$

**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*:

$\llbracket \text{mut-ghost-hs-phase } s = \text{hp-Idle}; \text{handshake-phase-inv } s; \text{phase-rel-inv } s \rrbracket$   
 $\implies \text{sys-phase } s = \text{ph-Idle}$

**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*:

$\llbracket \text{sys-mem-store-buffers } p \ s = \text{mw-fM } fl \ \# \ ws; \text{ghost-hs-phase } (s \text{ (mutator } m)) \neq \text{hp-Idle}; \text{fM-rel-inv } s; \text{handshake-phase-inv } s; \text{tso-store-inv } s; p \neq \text{sys} \rrbracket$

$\implies \text{False}$

**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*:

$\text{eq-imp } (\lambda r'. \text{mut-roots} \otimes \text{mut-ghost-honorary-root} \otimes (\lambda s. \bigcup (\text{ran } \text{'obj-fields' set-option (sys-heap } s \ r'))$   
 $\otimes \text{tso-pending-mutate (mutator } m))$   
 $(\text{reachable } r)$

**unfolding** *eq-imp-def* *reachable-def* *tso-store-refs-def*

**apply** *clarsimp*

**apply** (rename-tac *s s'*)

**apply** (subgoal-tac  $\forall r'. (\exists w \in \text{set (sys-mem-store-buffers (mutator } m) \ s). r' \in \text{store-refs } w) \longleftrightarrow (\exists w \in \text{set (sys-mem-store-refs (mutator } m) \ s'). r' \in \text{store-refs } w)$ )

**apply** (subgoal-tac  $\forall x. (x \text{ reaches } r) \ s \longleftrightarrow (x \text{ reaches } r) \ s'$ )

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

```

**lemma** *reachable-points-to[elim]:*

```

   $\llbracket (x \text{ points-to } y) \text{ } s; \text{ mut-m.reachable } m \text{ } x \text{ } s \rrbracket \implies \text{mut-m.reachable } m \text{ } y \text{ } s$ 
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]:*

```

   $\llbracket \text{reachable } y \text{ } s; \bigwedge x. \llbracket (x \text{ reaches } y) \text{ } s; x \in \text{mut-roots } s \rrbracket \implies Q; \bigwedge x. \llbracket (x \text{ reaches } y) \text{ } s; x \in \text{mut-ghost-honorary-root } s \rrbracket \implies Q; \bigwedge x. \llbracket (x \text{ reaches } y) \text{ } s; x \in \text{tso-store-refs } s \rrbracket \implies Q \rrbracket \implies Q$ 
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:  $\bigwedge x. \llbracket x \in \text{mut-m.mut-roots } m \text{ } s \rrbracket \implies P \text{ } x$ 
  assumes ghost-honorary-root:  $\bigwedge x. \llbracket x \in \text{mut-m.mut-ghost-honorary-root } m \text{ } s \rrbracket \implies P \text{ } x$ 
  assumes tso-root:  $\bigwedge x. x \in \text{mut-m.tso-store-refs } m \text{ } s \implies P \text{ } x$ 
  assumes reaches:  $\bigwedge x \text{ } y. \llbracket \text{mut-m.reachable } m \text{ } x \text{ } s; (x \text{ points-to } y) \text{ } s; P \text{ } x \rrbracket \implies P \text{ } y$ 
  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
   $\implies \text{mut-m.reachable } m \text{ } r \text{ } s \wedge (\forall r'. \text{opt-r}' = \text{Some } r' \longrightarrow \text{mut-m.reachable } m \text{ } r' \text{ } s)$ 
  sys-mem-store-buffers (mutator m) s = mw-Mutate-Payload r f pl # ws
   $\implies \text{mut-m.reachable } m \text{ } r \text{ } s$ 
by (auto simp: mut-m.tso-store-refs-def)

```

## 7.4 Colours

**lemma** *greyI*[intro]:

$r \in \text{ghost-honorary-grey } (s \ p) \implies \text{grey } r \ s$

$r \in W \ (s \ p) \implies \text{grey } r \ s$

$r \in WL \ p \ s \implies \text{grey } r \ s$

**unfolding** *grey-def WL-def* **by** (*case-tac* [!] *p*) *auto*

**lemma** *blackD*[dest]:

$\text{black } r \ s \implies \text{marked } r \ s$

$\text{black } r \ s \implies r \notin WL \ p \ s$

**unfolding** *black-def grey-def* **by** *simp-all*

**lemma** *whiteI*[intro]:

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

**unfolding** *white-def* **by** *simp*

**lemma** *marked-not-white*[dest]:

$\text{white } r \ s \implies \neg \text{marked } r \ s$

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

**lemma** *white-valid-ref*[elim!]:

$\text{white } r \ s \implies \text{valid-ref } r \ s$

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

**lemma** *not-white-marked*[elim!]:

$\llbracket \neg \text{white } r \ s; \text{valid-ref } r \ s \rrbracket \implies \text{marked } r \ s$

**unfolding** *white-def* **by** (*simp split: obj-at-splits*)

**lemma** *black-eq-imp*:

$\text{eq-imp } (\lambda :: \text{unit. } (\lambda s. r \in (\bigcup p. WL \ p \ s)) \otimes \text{sys-fM} \otimes (\lambda s. \text{map-option obj-mark } (\text{sys-heap } s \ r)))$   
 $(\text{black } r)$

**unfolding** *eq-imp-def black-def grey-def* **by** (*auto split: obj-at-splits*)

**lemma** *grey-eq-imp*:

$\text{eq-imp } (\lambda :: \text{unit. } (\lambda s. r \in (\bigcup p. WL \ p \ s)))$   
 $(\text{grey } r)$

**unfolding** *eq-imp-def grey-def* **by** *auto*

**lemma** *white-eq-imp*:

$\text{eq-imp } (\lambda :: \text{unit. } \text{sys-fM} \otimes (\lambda s. \text{map-option obj-mark } (\text{sys-heap } s \ r)))$   
 $(\text{white } r)$

**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*:

$\text{eq-imp } (\lambda r'. (\lambda s. \bigcup p. WL \ p \ s) \otimes \text{sys-fM} \otimes (\lambda s. \text{map-option obj-mark } (\text{sys-heap } s \ r')))$   
 $\text{black-heap}$

**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*  $\forall x. \text{marked } x \ s \longleftrightarrow \text{marked } x \ s'$ )

**apply** (*subgoal-tac*  $\forall x. \text{valid-ref } x \ s \longleftrightarrow \text{valid-ref } x \ s'$ )

**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 (λfl. fl = sys-fM s) in arg-cong)
apply simp
done

```

```

lemma white-heap-eq-imp:
  eq-imp (λr'. sys-fM ⊗ (λs. 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 (λr'. (λs. (⋃ p. WL p s)) ⊗ sys-fM ⊗ (λs. 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 ∀ x. marked x s ⟷ 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]:
   $\llbracket \text{no-grey-refs } s; \forall r. \text{marked } r \ s \vee \neg \text{valid-ref } r \ s \rrbracket \Longrightarrow \text{black-heap } s$ 
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*:

$\llbracket \text{black } x \text{ } s; (x \text{ points-to } y) \text{ } s; \text{valid-ref } y \text{ } s; \text{strong-tricolour-inv } s \rrbracket$   
 $\implies \text{marked } y \text{ } s$

**unfolding** *strong-tricolour-inv-def* **by** *fastforce*

**lemma** *no-black-refsD*:

*no-black-refs s*  $\implies \neg \text{black } r \text{ } s$

**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**  $(x \text{ has-white-path-to } y) \text{ } s$

**assumes**  $\bigwedge x. P \text{ } x$

**assumes**  $\bigwedge x \text{ } y \text{ } z. \llbracket (x \text{ has-white-path-to } y) \text{ } s; P \text{ } x \text{ } y; (y \text{ points-to } z) \text{ } s; \text{white } z \text{ } s \rrbracket \implies P \text{ } x \text{ } z$

**shows**  $P \text{ } x \text{ } y$

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

**lemma** *has-white-path-toD*[*dest*]:

$(x \text{ has-white-path-to } y) \text{ } s \implies \text{white } y \text{ } s \vee x = y$

**unfolding** *has-white-path-to-def* **by** (*fastforce elim: rtranclp.cases*)

**lemma** *has-white-path-to-refl*[*iff*]:

$(x \text{ has-white-path-to } x) \text{ } s$

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

**lemma** *has-white-path-to-step*[*intro*]:

$\llbracket (x \text{ has-white-path-to } y) \text{ } s; (y \text{ points-to } z) \text{ } s; \text{white } z \text{ } s \rrbracket \implies (x \text{ has-white-path-to } z) \text{ } s$

$\llbracket (y \text{ has-white-path-to } z) \text{ } s; (x \text{ points-to } y) \text{ } s; \text{white } y \text{ } s \rrbracket \implies (x \text{ has-white-path-to } z) \text{ } s$

**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!*]:

$\llbracket (x \text{ points-to } y) \text{ } s; \text{white } y \text{ } s \rrbracket \implies (x \text{ has-white-path-to } y) \text{ } s$

**unfolding** *has-white-path-to-def* **by** (*auto elim: rtranclp.intros(2)*)

**lemma** *has-white-path-to-reaches*[*elim*]:

$(x \text{ has-white-path-to } y) \text{ } s \implies (x \text{ reaches } y) \text{ } s$

**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*]:

$(x \text{ has-white-path-to } w) (s(\text{gc} := s \text{ gc} \llbracket W := \text{gc-}W \text{ } s - \text{rs} \rrbracket)) \longleftrightarrow (x \text{ has-white-path-to } w) \text{ } s$

**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**  $(x \text{ has-white-path-to } y) \text{ } s'$

**assumes**  $\forall r'. \bigcup (\text{ran } \text{'obj-fields' set-option (sys-heap } s' \text{ } r')) = \bigcup (\text{ran } \text{'obj-fields' set-option (sys-heap } s \text{ } r'))$

**assumes**  $\forall r'. \text{map-option obj-mark (sys-heap } s' \text{ } r') = \text{map-option obj-mark (sys-heap } s \text{ } r')$

**assumes**  $\text{sys-fM } s' = \text{sys-fM } s$

**shows**  $(x \text{ has-white-path-to } y) \text{ } s$

**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 ( $\lambda r'. \text{sys-fM} \otimes (\lambda s. \bigcup (\text{ran } \text{'obj-fields' set-option (sys-heap s r')}) \otimes (\lambda s. \text{map-option obj-mark (sys-heap s r')}))$ )
    (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  $\implies$  grey g s  $\wedge$  (g = w  $\vee$  white w s)
by (auto simp: grey-protects-white-def)

lemma grey-protects-whiteI[iff]:
  grey g s  $\implies$  (g grey-protects-white g) s
by (simp add: grey-protects-white-def)

lemma grey-protects-whiteE[elim!]:
   $\llbracket (g \text{ points-to } w) s; \text{grey } g s; \text{white } w s \rrbracket \implies (g \text{ grey-protects-white } w) s$ 
   $\llbracket (g \text{ grey-protects-white } y) s; (y \text{ points-to } w) s; \text{white } w s \rrbracket \implies (g \text{ grey-protects-white } w) s$ 
by (auto simp: grey-protects-white-def)

lemma grey-protects-white-reaches[elim]:
  (g grey-protects-white w) s  $\implies$  (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  $\bigwedge x. \text{grey } x s \implies P x x$ 
  assumes  $\bigwedge x y z. \llbracket (x \text{ has-white-path-to } y) s; P x y; (y \text{ points-to } z) s; \text{white } z s \rrbracket \implies P x z$ 
  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*  $\implies$  *sys-ghost-honorary-grey s* = {}

**unfolding** *valid-W-inv-def* **by** *simp*

**lemma** *WLI*[intro]:

$r \in W (s p) \implies r \in WL p s$

$r \in \text{ghost-honorary-grey} (s p) \implies r \in WL p s$

**unfolding** *WL-def* **by** *simp-all*

**lemma** *WL-eq-imp*:

*eq-imp* ( $\lambda(-::\text{unit}) s. (\text{ghost-honorary-grey} (s p), W (s p)))$   
 $(WL p)$

**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* ( $\lambda(p, r). (\lambda s. W (s p)) \otimes (\lambda s. \text{ghost-honorary-grey} (s p)) \otimes \text{sys-fM} \otimes (\lambda s. \text{map-option obj-mark} (\text{sys-heap } s r)) \otimes \text{sys-mem-lock} \otimes \text{tso-pending-mark } p)$   
*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*  $\forall p. WL p s = WL p s'$ )

**apply** (*subgoal-tac*  $\forall x. \text{marked } x s \longleftrightarrow \text{marked } x s'$ )

**apply** (*subgoal-tac*  $\forall x. \text{obj-at} (\lambda \text{obj}. \text{obj-mark obj} = (\neg \text{sys-fM } s')) x s \longleftrightarrow \text{obj-at} (\lambda \text{obj}. \text{obj-mark obj} = (\neg \text{sys-fM } s')) x s'$ )

**apply** (*subgoal-tac*  $\forall x xa xb. \text{mw-Mark } xa xb \in \text{set} (\text{sys-mem-store-buffers } x s) \longleftrightarrow \text{mw-Mark } xa xb \in \text{set} (\text{sys-mem-store-buffers } x s')$ )

**apply** (*simp; fail*)

**apply** *clarsimp*

**apply** (*rename-tac x xa xb*)

**apply** (*drule-tac*  $x=x$  **in** *spec*, *drule arg-cong*[**where**  $f=\text{set}$ ], *fastforce*)

**apply** (*clarsimp split: obj-at-splits*)

**apply** (*rename-tac x*)

**apply** ( $(\text{drule-tac } x=x \text{ in } \text{spec})+ ) [1]$ )

**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!]:

$\llbracket r \in W (s p); \text{valid-W-inv } s \rrbracket \implies \text{marked } r s$

$\llbracket r \in \text{ghost-honorary-grey} (s p); \text{sys-mem-lock } s \neq \text{Some } p; \text{valid-W-inv } s \rrbracket \implies \text{marked } r s$

$\llbracket r \in W (s p); \text{valid-W-inv } s \rrbracket \implies \text{valid-ref } r s$

$\llbracket r \in \text{ghost-honorary-grey} (s p); \text{sys-mem-lock } s \neq \text{Some } p; \text{valid-W-inv } s \rrbracket \implies \text{valid-ref } r s$

$\llbracket \text{mw-Mark } r \text{ fl} \in \text{set} (\text{sys-mem-store-buffers } p s); \text{valid-W-inv } s \rrbracket \implies r \in \text{ghost-honorary-grey} (s p)$

**unfolding** *valid-W-inv-def*

**apply** (*simp-all add: split: obj-at-splits*)

**apply** *blast+*

done

**lemma** *valid-W-invD*:

$\llbracket \text{sys-mem-store-buffers } p \ s = \text{mw-Mark } r \ fl \ \# \ ws; \text{valid-W-inv } s \rrbracket$   
 $\implies fl = \text{sys-fM } s \wedge r \in \text{ghost-honorary-grey } (s \ p) \wedge \text{tso-locked-by } p \ s \wedge \text{white } r \ s \wedge \text{filter is-mw-Mark } ws = []$   
 $\llbracket \text{mw-Mark } r \ fl \in \text{set } (\text{sys-mem-store-buffers } p \ s); \text{valid-W-inv } s \rrbracket$   
 $\implies fl = \text{sys-fM } s \wedge r \in \text{ghost-honorary-grey } (s \ p) \wedge \text{tso-locked-by } p \ s \wedge \text{white } r \ s \wedge \text{filter is-mw-Mark } (\text{sys-mem-store-buffers } p \ s) = [\text{mw-Mark } r \ fl]$   
**unfolding** *valid-W-inv-def white-def* **by** (*clarsimp dest!*: *spec[where x=p]*, *blast*)+

**lemma** *valid-W-inv-colours*:

$\llbracket \text{white } x \ s; \text{valid-W-inv } s \rrbracket \implies x \notin W \ (s \ p)$   
**using** *marked-not-white valid-W-invE(1)* **by** *force*

**lemma** *valid-W-inv-no-mark-stores-invD*:

$\llbracket \text{sys-mem-lock } s \neq \text{Some } p; \text{valid-W-inv } s \rrbracket$   
 $\implies \text{tso-pending } p \text{ is-mw-Mark } s = []$   
**by** (*auto dest: valid-W-invD(2) intro!*: *filter-False*)

**lemma** *valid-W-inv-sys-load[simp]*:

$\llbracket \text{sys-mem-lock } s \neq \text{Some } p; \text{valid-W-inv } s \rrbracket$   
 $\implies \text{sys-load } p \ (\text{mr-Mark } r) \ (s \ \text{sys}) = \text{mv-Mark } (\text{map-option obj-mark } (\text{sys-heap } s \ r))$

**unfolding** *sys-load-def fold-stores-def*

**apply** *clarsimp*

**apply** (*rule fold-invariant[where P= $\lambda fr. \text{map-option obj-mark } (\text{heap } (fr \ (s \ \text{sys})) \ r) = \text{map-option obj-mark } (\text{sys-heap } s \ r)$*

**and** *Q= $\lambda w. \forall r \ fl. w \neq \text{mw-Mark } r \ fl$* )

**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* ( $\lambda r'. (\lambda s. \bigcup p. WL \ p \ s) \otimes (\lambda s. \text{Set.bind } (\text{Option.set-option } (\text{sys-heap } s \ r')) \ (\text{ran} \circ \text{obj-fields}))$ )  
*(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*  $\implies \text{grey-reachable } g \ s$   
**unfolding** *grey-reachable-def reaches-def* **by** *blast*

**lemma** *grey-reachableE*:

$\llbracket (g \ \text{points-to } y) \ s; \text{grey-reachable } g \ s \rrbracket \implies \text{grey-reachable } y \ s$   
**unfolding** *grey-reachable-def reaches-def* **by** (*auto elim: rtranclp.intros(2)*)

## 7.7 valid refs inv

**lemma** *valid-refs-invI*:

$\llbracket \bigwedge m \ x \ y. \llbracket (x \ \text{reaches } y) \ s; \text{mut-m.root } m \ x \ s \vee \text{grey } x \ s \rrbracket \implies \text{valid-ref } y \ s$   
 $\rrbracket \implies \text{valid-refs-inv } s$

**by** (*auto simp: valid-refs-inv-def mut-m.reachable-def grey-reachable-def*)

**lemma** *valid-refs-inv-eq-imp*:

*eq-imp* ( $\lambda (m', r'). (\lambda s. \text{roots } (s \ (\text{mutator } m')) \otimes (\lambda s. \text{ghost-honorary-root } (s \ (\text{mutator } m')) \otimes (\lambda s. \text{map-option$

$obj\text{-}fields\ (sys\text{-}heap\ s\ r') \otimes tso\text{-}pending\text{-}mutate\ (mutator\ m') \otimes (\lambda s. \bigcup p. WL\ p\ s))$   
 $valid\text{-}refs\text{-}inv$   
**apply** (*clarsimp simp: eq-imp-def valid-refs-inv-def grey-reachable-def all-conj-distrib*)  
**apply** (*rename-tac s s'*)  
**apply** (*subgoal-tac  $\forall r'. valid\text{-}ref\ r'\ s \longleftrightarrow valid\text{-}ref\ r'\ s'$* )  
**apply** (*subgoal-tac  $\forall r'. \bigcup (ran\ 'obj\text{-}fields\ 'set\text{-}option\ (sys\text{-}heap\ s\ r')) = \bigcup (ran\ 'obj\text{-}fields\ 'set\text{-}option\ (sys\text{-}heap\ s'\ r'))$* )  
**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]:*  
 $\llbracket x \in mut\text{-}m.mut\text{-}roots\ m\ s; (x\ reaches\ y)\ s; valid\text{-}refs\text{-}inv\ s \rrbracket \implies valid\text{-}ref\ y\ s$   
 $\llbracket x \in mut\text{-}m.mut\text{-}roots\ m\ s; (x\ reaches\ y)\ s; valid\text{-}refs\text{-}inv\ s \rrbracket \implies \exists obj. sys\text{-}heap\ s\ y = Some\ obj$   
 $\llbracket x \in mut\text{-}m.tso\text{-}store\text{-}refs\ m\ s; (x\ reaches\ y)\ s; valid\text{-}refs\text{-}inv\ s \rrbracket \implies valid\text{-}ref\ y\ s$   
 $\llbracket x \in mut\text{-}m.tso\text{-}store\text{-}refs\ m\ s; (x\ reaches\ y)\ s; valid\text{-}refs\text{-}inv\ s \rrbracket \implies \exists obj. sys\text{-}heap\ s\ y = Some\ obj$   
 $\llbracket w \in set\ (sys\text{-}mem\text{-}store\text{-}buffers\ (mutator\ m)\ s); x \in store\text{-}refs\ w; (x\ reaches\ y)\ s; valid\text{-}refs\text{-}inv\ s \rrbracket \implies valid\text{-}ref\ y\ s$   
 $\llbracket w \in set\ (sys\text{-}mem\text{-}store\text{-}buffers\ (mutator\ m)\ s); x \in store\text{-}refs\ w; (x\ reaches\ y)\ s; valid\text{-}refs\text{-}inv\ s \rrbracket \implies \exists obj. sys\text{-}heap\ s\ y = Some\ obj$   
 $\llbracket grey\ x\ s; (x\ reaches\ y)\ s; valid\text{-}refs\text{-}inv\ s \rrbracket \implies valid\text{-}ref\ y\ s$   
 $\llbracket mut\text{-}m.reachable\ m\ x\ s; valid\text{-}refs\text{-}inv\ s \rrbracket \implies valid\text{-}ref\ x\ s$   
 $\llbracket mut\text{-}m.reachable\ m\ x\ s; valid\text{-}refs\text{-}inv\ s \rrbracket \implies \exists obj. sys\text{-}heap\ s\ x = Some\ obj$   
 $\llbracket x \in mut\text{-}m.mut\text{-}ghost\text{-}honorary\text{-}root\ m\ s; (x\ reaches\ y)\ s; valid\text{-}refs\text{-}inv\ s \rrbracket \implies valid\text{-}ref\ y\ s$   
 $\llbracket x \in mut\text{-}m.mut\text{-}ghost\text{-}honorary\text{-}root\ m\ s; (x\ reaches\ y)\ s; valid\text{-}refs\text{-}inv\ s \rrbracket \implies \exists obj. sys\text{-}heap\ s\ y = Some\ obj$   
**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]:*  
 $(\bigwedge y. reachable\ y\ s \implies in\text{-}snapshot\ y\ s) \implies reachable\text{-}snapshot\text{-}inv\ s$   
**by** (*simp add: reachable-snapshot-inv-def*)

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

**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  $\forall x. \text{obj-at } (\lambda \text{obj}. \text{obj-mark obj} = \text{sys-fM } s') x s \longleftrightarrow \text{obj-at } (\lambda \text{obj}. \text{obj-mark obj} = \text{sys-fM } s') x s')$ 
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  $\implies$  in-snapshot r s  
 grey r s  $\implies$  in-snapshot r s  
 $\llbracket \text{white } w s; (g \text{ grey-protects-white } w) s \rrbracket \implies \text{in-snapshot } w s$   
**by** (auto simp: in-snapshot-def)

**lemma** — Sanity  
 in-snapshot r s  $\implies$  black r s  $\vee$  grey r s  $\vee$  white r s  
**by** (auto simp: in-snapshot-def)

**lemma** in-snapshot-valid-ref:  
 $\llbracket \text{in-snapshot } r s; \text{valid-refs-inv } s \rrbracket \implies \text{valid-ref } r s$   
**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]:  
 $x \in \text{mut-m.mut-ghost-honorary-root } m s \implies \text{mut-m.reachable } m x s$   
**unfolding** mut-m.reachable-def reaches-def **by** blast

**lemma** tso-pending-mw-mutate-cong:  
 $\llbracket \text{filter is-mw-Mutate } (\text{sys-mem-store-buffers } p s) = \text{filter is-mw-Mutate } (\text{sys-mem-store-buffers } p s'); \bigwedge r f r'. P r f r' \longleftrightarrow Q r f r' \rrbracket$   
 $\implies (\forall r f r'. \text{mw-Mutate } r f r' \in \text{set } (\text{sys-mem-store-buffers } p s) \longrightarrow P r f r')$   
 $\longleftrightarrow (\forall r f r'. \text{mw-Mutate } r f r' \in \text{set } (\text{sys-mem-store-buffers } p s') \longrightarrow Q r f r')$   
**by** (intro iff-allI) (auto dest!: arg-cong[where f=set])

**lemma** (in mut-m) marked-insertions-eq-imp:  
 eq-imp ( $\lambda r'. \text{sys-fM } \otimes (\lambda s. \text{map-option obj-mark } (\text{sys-heap } s r')) \otimes \text{tso-pending-mw-mutate } (\text{mutator } m))$   
 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-simps*, *rule-format*]

**lemma** *marked-insertionD*[elim!]:

$\llbracket \text{sys-mem-store-buffers } (\text{mutator } m) \ s = \text{mw-Mutate } r \ f \ (\text{Some } r') \ \# \ ws; \text{mut-m.marked-insertions } m \ s \rrbracket$   
 $\implies \text{marked } r' \ s$

**by** (*auto simp: mut-m.marked-insertions-def*)

**lemma** *marked-insertions-store-buffer-empty*[intro]:

*tso-pending-mutate* (*mutator m*) *s* = []  $\implies$  *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* ( $\lambda r'. \text{sys-fM} \otimes (\lambda s. \text{map-option obj-fields } (\text{sys-heap } s \ r')) \otimes (\lambda s. \text{map-option obj-mark } (\text{sys-heap } s \ r'))$ )  
 $\otimes \text{tso-pending-mw-mutate } (\text{mutator } m)$   
*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*  $\forall x. \text{marked } x \ s \longleftrightarrow \text{marked } x \ s'$ )

**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* = []  $\implies$  *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)⟧))*  $\longleftrightarrow$  *obj-at P r s*  $\wedge$  *r*  $\neq$  *r'*

**by** (*clarsimp simp: fun-upd-apply split: obj-at-splits*)

**lemma** *obj-at-alloc*[simp]:

*sys-heap s r' = None*

$\implies \text{obj-at } P \ r \ (s(m := \text{mut-m-s}', \text{sys} := (s \ \text{sys}) \llbracket \text{heap} := (\text{sys-heap } s)(r' \mapsto \text{obj}) \rrbracket))$

$\longleftrightarrow (\text{obj-at } P \ r \ s \vee (r = r' \wedge P \ \text{obj}))$

**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)⟧))*  $\longleftrightarrow$   
*valid-ref r s*

*valid-null-ref r' (s(sys := do-store-action w (s sys)⟦mem-store-buffers := (mem-store-buffers (s sys))(p := ws)⟧))*  
 $\longleftrightarrow \text{valid-null-ref } r' \ s$

*valid-null-ref r' (s(mutator m := mut-s', sys := (s sys)⟦heap := (heap (s sys))(r''  $\mapsto$  obj)⟧))*  $\longleftrightarrow$  *valid-null-ref*  
*r' s*  $\vee$  *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'*()))  
 $\longleftrightarrow$  *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* = {}

$\implies$  *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*)) = {}

$\implies$  *WL p* (*s*(*mutator m* := *s* (*mutator m*)() *W* := {}, *ghost-hs-phase* := *hp'*),

*sys* := *s sys*() *hs-pending* := *hsp'*, *W* := *sys-W s* ∪ *W* (*s* (*mutator m*)),

*ghost-hs-in-sync* := *in'*()))

= (*case p of gc* ⇒ *WL gc s* | *mutator m'* ⇒ (*if m' = m* then {} else *WL* (*mutator m'*) *s*) | *sys* ⇒ *WL sys s* ∪ *WL* (*mutator m*) *s*)

*ghost-honorary-grey* (*s* (*mutator m*)) = {}

$\implies$  *WL p* (*s*(*mutator m* := *s* (*mutator m*)() *W* := {},

*sys* := *s sys*() *hs-pending* := *hsp'*, *W* := *sys-W s* ∪ *W* (*s* (*mutator m*)),

*ghost-hs-in-sync* := *in'*()))

= (*case p of gc* ⇒ *WL gc s* | *mutator m'* ⇒ (*if m' = m* then {} else *WL* (*mutator m'*) *s*) | *sys* ⇒ *WL sys s* ∪ *WL* (*mutator m*) *s*)

**unfolding** *WL-def* **by** (*auto simp: fun-upd-apply split: process-name.splits*)

**lemma** *colours-load-W*[*iff*]:

$gc-W\ s = \{\} \implies black\ r\ (s(gc := (s\ gc)\ W := W\ (s\ sys))\ sys := (s\ sys)\ W := \{\})) \longleftrightarrow black\ r\ s$   
 $gc-W\ s = \{\} \implies grey\ r\ (s(gc := (s\ gc)\ W := W\ (s\ sys))\ sys := (s\ sys)\ W := \{\})) \longleftrightarrow grey\ r\ s$   
**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*]:

$gc-W\ s = \{\}$   
 $\implies (WL\ p\ (s(gc := (s\ gc)\ W := sys-W\ s)\ sys := (s\ sys)\ W := \{\}))$   
 $= (case\ p\ of\ gc \Rightarrow WL\ gc\ s \cup sys-W\ s \mid mutator\ m \Rightarrow WL\ (mutator\ m)\ s \mid sys \Rightarrow sys-ghost-honorary-grey\ s)$   
**unfolding** *WL-def* **by** (*auto simp: fun-upd-apply split: process-name.splits*)

no grey refs

**lemma** *no-grey-refs-eq-imp*:

$eq-imp\ (\lambda(-::unit). (\lambda s. \bigcup p. WL\ p\ s))$   
*no-grey-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*:

$\llbracket no-grey-refs\ s; valid-W-inv\ s \rrbracket \implies tso-no-pending-marks\ s$   
**unfolding** *no-grey-refs-def* **by** (*auto intro!: filter-False dest: valid-W-invD(2)*)

**lemma** *no-grey-refs-not-grey-reachableD*:

$no-grey-refs\ s \implies \neg grey-reachable\ x\ s$   
**by** (*clarsimp simp: no-grey-refs-def grey-reachable-def*)

**lemma** *no-grey-refsD*:

$no-grey-refs\ s \implies r \notin W\ (s\ p)$   
 $no-grey-refs\ s \implies r \notin WL\ p\ s$   
 $no-grey-refs\ s \implies r \notin ghost-honorary-grey\ (s\ p)$   
**by** (*auto simp: no-grey-refs-def*)

**lemma** *no-grey-refs-marked*[*dest*]:

$\llbracket marked\ r\ s; no-grey-refs\ s \rrbracket \implies black\ r\ s$   
**by** (*auto simp: no-grey-refs-def black-def*)

**lemma** *no-grey-refs-bwD*[*dest*]:

$\llbracket heap\ (s\ sys)\ r = Some\ obj; no-grey-refs\ s \rrbracket \implies black\ r\ s \vee white\ r\ s$   
**by** (*clarsimp simp: black-def grey-def no-grey-refs-def white-def split: obj-at-splits*)

**context** *mut-m*

**begin**

**lemma** *reachable-blackD*:

$\llbracket no-grey-refs\ s; reachable-snapshot-inv\ s; reachable\ r\ s \rrbracket \implies black\ r\ s$   
**by** (*simp add: no-grey-refs-def reachable-snapshot-inv-def in-snapshot-def grey-protects-white-def*)

**lemma** *no-grey-refs-not-reachable*:

$\llbracket no-grey-refs\ s; reachable-snapshot-inv\ s; white\ r\ s \rrbracket \implies \neg reachable\ r\ s$   
**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* ]]  
 $\implies r \notin \text{mut-roots } s \wedge r \notin \text{mut-ghost-honorary-root } s \wedge r \notin \text{tso-store-refs } s$   
**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*;  $w \in \text{mut-roots } s \vee w \in \text{mut-ghost-honorary-root } s$ ; *reachable-snapshot-inv s* ]] $\implies \exists g. (g \text{ grey-protects-white } w) s$   
**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 ( $\lambda \cdot$  fl)) (sys-heap s r))),*  
*mem-store-buffers := (mem-store-buffers (s sys))(p := ws) )*)  
 $\longleftrightarrow (black\ b\ s \wedge b \neq r) \vee (b = r \wedge fl = \text{sys-fM } s \wedge \text{valid-ref } r\ s \wedge \neg \text{grey } r\ s)$   
**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)))*)  $\longleftrightarrow (black\ r\ s \wedge r \neq r')$   
*grey r (s(sys := s sys(heap := (heap (s sys))(r' := None)))*)  $\longleftrightarrow (grey\ r\ s)$   
*white r (s(sys := s sys(heap := (heap (s sys))(r' := None)))*)  $\longleftrightarrow (white\ r\ s \wedge r \neq r')$   
**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 := {}))*,  
 $sys := (s\ sys)(hs\text{-}pending := hp', W := W\ (s\ sys) \cup W\ (s\ (mutator\ m)),$   
 $ghost\text{-}hs\text{-}in\text{-}sync := his' )$ )  $\longleftrightarrow black\ r\ s$   
*grey r (s(mutator m := (s (mutator m))(W := {}))*,  
 $sys := (s\ sys)(hs\text{-}pending := hp', W := W\ (s\ sys) \cup W\ (s\ (mutator\ m)),$   
 $ghost\text{-}hs\text{-}in\text{-}sync := his' )$ )  $\longleftrightarrow grey\ r\ s$   
*white r (s(mutator m := (s (mutator m))(W := {}))*,  
 $sys := (s\ sys)(hs\text{-}pending := hp', W := W\ (s\ sys) \cup W\ (s\ (mutator\ m)),$   
 $ghost\text{-}hs\text{-}in\text{-}sync := his' )$ )  $\longleftrightarrow white\ r\ s$

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

**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*]:

$fl \neq sys\text{-}fM\ s \implies black\ b\ (s(sys := (s\ sys)(\!|fM := fl, mem\text{-}store\text{-}buffers := (mem\text{-}store\text{-}buffers\ (s\ sys))(p := ws)\!))) \longleftrightarrow white\ b\ s \wedge \neg grey\ b\ s$

**unfolding** *black-def white-def* **by** (*simp add: fun-upd-apply*)

**lemma** *colours-alloc*[*simp*]:

$heap\ (s\ sys)\ r' = None$   
 $\implies black\ r\ (s(mutator\ m := (s\ (mutator\ m))(\!|roots := roots'\!|), sys := (s\ sys)(\!|heap := (heap\ (s\ sys))(r' \mapsto (\!|obj\text{-}mark = fl, obj\text{-}fields = Map.empty, obj\text{-}payload = Map.empty\!|)\!))) \longleftrightarrow black\ r\ s \vee (r' = r \wedge fl = sys\text{-}fM\ s \wedge \neg grey\ r'\ s)$   
 $grey\ r\ (s(mutator\ m := (s\ (mutator\ m))(\!|roots := roots'\!|), sys := (s\ sys)(\!|heap := (heap\ (s\ sys))(r' \mapsto (\!|obj\text{-}mark = fl, obj\text{-}fields = Map.empty, obj\text{-}payload = Map.empty\!|)\!))) \longleftrightarrow grey\ r\ s$   
 $heap\ (s\ sys)\ r' = None$   
 $\implies white\ r\ (s(mutator\ m := (s\ (mutator\ m))(\!|roots := roots'\!|), sys := (s\ sys)(\!|heap := (heap\ (s\ sys))(r' \mapsto (\!|obj\text{-}mark = fl, obj\text{-}fields = Map.empty, obj\text{-}payload = Map.empty\!|)\!))) \longleftrightarrow white\ r\ s \vee (r' = r \wedge fl \neq sys\text{-}fM\ s)$

**unfolding** *black-def white-def* **by** (*auto simp: fun-upd-apply split: obj-at-splits*)

**lemma** *heap-colours-alloc*[*simp*]:

$\llbracket heap\ (s\ sys)\ r' = None; valid\text{-}refs\text{-}inv\ s \rrbracket$   
 $\implies black\text{-}heap\ (s(mutator\ m := s\ (mutator\ m)(\!|roots := roots'\!|), sys := s\ sys(\!|heap := (sys\text{-}heap\ s)(r' \mapsto (\!|obj\text{-}mark = fl, obj\text{-}fields = Map.empty, obj\text{-}payload = Map.empty\!|)\!))) \longleftrightarrow black\text{-}heap\ s \wedge fl = sys\text{-}fM\ s$   
 $heap\ (s\ sys)\ r' = None$   
 $\implies white\text{-}heap\ (s(mutator\ m := s\ (mutator\ m)(\!|roots := roots'\!|), sys := s\ sys(\!|heap := (sys\text{-}heap\ s)(r' \mapsto (\!|obj\text{-}mark = fl, obj\text{-}fields = Map.empty, obj\text{-}payload = Map.empty\!|)\!))) \longleftrightarrow white\text{-}heap\ s \wedge fl \neq sys\text{-}fM\ s$

**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*]:

$(g\ grey\text{-}protects\text{-}white\ w)\ (s(mutator\ m := s\ (mutator\ m)(\!|W := \{\}, ghost\text{-}hs\text{-}phase := hs'\!|), sys := s\ sys(\!|hs\text{-}pending := hp', W := sys\text{-}W\ s \cup W\ (s\ (mutator\ m)), ghost\text{-}hs\text{-}in\text{-}sync := his'\!|)))$

$\longleftrightarrow (g\ grey\text{-}protects\text{-}white\ w)\ s$

**unfolding** *grey-protects-white-def* **by** (*simp add: fun-upd-apply*)

**lemma** *grey-protects-white-alloc*[*simp*]:

$\llbracket fl = sys\text{-}fM\ s; sys\text{-}heap\ s\ r = None \rrbracket$   
 $\implies (g\ grey\text{-}protects\text{-}white\ w)\ (s(mutator\ m := s\ (mutator\ m)(\!|roots := roots'\!|), sys := s\ sys(\!|heap := (sys\text{-}heap\ s)(r \mapsto (\!|obj\text{-}mark = fl, obj\text{-}fields = Map.empty, obj\text{-}payload = Map.empty\!|)\!)))$

$\longleftrightarrow (g \text{ grey-protects-white } w) \ s$   
**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** *n timer*: *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*  $\wedge$  *y*  $\neq$  *r*  
**proof**(*induct rule: reachable-induct*)  
**case** (*root x*) **with** *ngs n timer rsi* **show** ?*case*  
**by** (*auto simp: fun-upd-apply dest: reachable-blackD*)  
**next**  
**case** (*ghost-honorary-root x*) **with** *ngs n timer rsi* **show** ?*case*  
**unfolding** *reachable-def reaches-def* **by** (*auto simp: fun-upd-apply dest: reachable-blackD*)  
**next**  
**case** (*tso-root x*) **with** *ngs n timer rsi* **show** ?*case*  
**unfolding** *reachable-def reaches-def* **by** (*auto simp: fun-upd-apply dest: reachable-blackD*)  
**next**  
**case** (*reaches x y*) **with** *ngs n timer rsi* **show** ?*case*  
**unfolding** *reachable-def reaches-def*  
**apply** (*clarsimp simp: fun-upd-apply*)  
**apply** (*drule predicate2D[OF rtranclp-mono[where s= $\lambda x y. (x \text{ points-to } y) \ s, OF predicate2I], rotated]$* )  
**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  $\mapsto$  (obj-mark = fl, obj-fields = Map.empty, obj-payload = Map.empty))))*  
 $\longleftrightarrow$  *mut-m.reachable m r' s*  $\vee$  (*m' = m*  $\wedge$  *r' = r*) (**is** ?*lhs*  $\longleftrightarrow$  ?*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]:
   $\llbracket \text{reachable-snapshot-inv } s; \text{roots}' \subseteq \text{roots } (s \text{ (mutator } m)) \rrbracket$ 
 $\implies \text{reachable-snapshot-inv } (s(\text{mutator } m := (s \text{ (mutator } m))(\text{roots} := \text{roots}')))$ 
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]:
   $\llbracket \text{reachable-snapshot-inv } s; \text{sys-load } (\text{mutator } m) \text{ (mr-Ref } r \text{ f)} (s \text{ sys}) = \text{mv-Ref } r'; r \in \text{mut-roots } s \rrbracket$ 
 $\implies \text{reachable-snapshot-inv } (s(\text{mutator } m := s \text{ (mutator } m))(\text{roots} := \text{mut-roots } s \cup \text{Option.set-option } r' \text{ })))$ 
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]:
   $\llbracket \text{reachable-snapshot-inv } s; r \in \text{mut-roots } s; (\exists r'. \text{opt-r}' = \text{Some } r') \longrightarrow \text{the opt-r}' \in \text{mut-roots } s \rrbracket$ 
 $\implies \text{reachable-snapshot-inv } (s(\text{mutator } m := s \text{ (mutator } m))(\text{ghost-honorary-root} := \{\}),$ 
 $\text{sys} := s \text{ sys}(\text{mem-store-buffers} := (\text{mem-store-buffers } (s \text{ sys}))(\text{mutator } m :=$ 
 $\text{sys-mem-store-buffers } (\text{mutator } m) \text{ s } @ [\text{mw-Mutate } r \text{ f opt-r}'] \text{ })))$ 
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) = {}
 $\implies \text{WL } p' (s(p := s p(\text{ghost-honorary-grey} := rs))) = \text{WL } p' s \cup \{ r \mid r. p' = p \wedge r \in rs \}$ 
unfolding WL-def by (simp add: fun-upd-apply)

```

**lemma** *ghost-honorary-grey-mo-co-mark*[simp]:

$\llbracket \text{ghost-honorary-grey } (s \ p) = \{\} \rrbracket \implies \text{black } b \ (s(p := s \ p \llbracket \text{ghost-honorary-grey} := \{r\} \rrbracket)) \longleftrightarrow \text{black } b \ s \wedge b \neq r$   
 $\llbracket \text{ghost-honorary-grey } (s \ p) = \{\} \rrbracket \implies \text{grey } g \ (s(p := (s \ p) \llbracket \text{ghost-honorary-grey} := \{r\} \rrbracket)) \longleftrightarrow \text{grey } g \ s \vee g = r$   
 $\llbracket \text{ghost-honorary-grey } (s \ p) = \{\} \rrbracket \implies \text{white } w \ (s(p := s \ p \llbracket \text{ghost-honorary-grey} := \{r\} \rrbracket)) \longleftrightarrow \text{white } w \ s$

**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*}  
 $\implies (WL \ p \ (s(p' := (s \ p') \llbracket W := \text{insert } r \ (W \ (s \ p')), \text{ghost-honorary-grey} := \{r\} \rrbracket))) = (WL \ p \ s)$   
*ghost-honorary-grey* (*s p'*) = {*r*}  
 $\implies \text{grey } g \ (s(p' := (s \ p') \llbracket W := \text{insert } r \ (W \ (s \ p')), \text{ghost-honorary-grey} := \{r\} \rrbracket)) \longleftrightarrow \text{grey } g \ s$

**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* ( $\llbracket \text{heap} := (\text{sys-heap } s)(r' := \text{None}) \rrbracket$ )))  
 $\implies \text{mut-m.reachable } m \ r \ s$

**unfolding** *mut-m.reachable-def reaches-def* **by** (*clarsimp simp: fun-upd-apply*) (*metis (no-types, lifting) mono-rtranclp*)

**lemma** *reachable-deref-del*[simp]:

$\llbracket \text{sys-load } (\text{mutator } m) \ (\text{mr-Ref } r \ f) \ (s \ \text{sys}) = \text{mv-Ref } \text{opt-r'}; r \in \text{mut-m.mut-roots } m \ s; \text{mut-m.mut-ghost-honorary-root } m \ s = \{\} \rrbracket$   
 $\implies \text{mut-m.reachable } m' \ y \ (s(\text{mutator } m := s \ (\text{mutator } m) \llbracket \text{ghost-honorary-root} := \text{Option.set-option } \text{opt-r'}, \text{ref} := \text{opt-r'} \rrbracket))$   
 $\longleftrightarrow \text{mut-m.reachable } m' \ y \ s$

**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]:

$\llbracket \text{sys-mem-store-buffers } p \ s = \text{mw-Mark } r \ fl \ \# \ ws; \text{no-black-refs } s; \text{valid-W-inv } s \rrbracket$   
 $\implies \text{no-black-refs } (s(\text{sys} := s \ \text{sys} \llbracket \text{heap} := (\text{sys-heap } s)(r := \text{map-option } (\text{obj-mark-update } (\lambda \cdot. fl)) \ (\text{sys-heap } s \ r)), \text{mem-store-buffers} := (\text{mem-store-buffers } (s \ \text{sys}))(p := ws) \rrbracket))$   
 $\llbracket \text{sys-mem-store-buffers } p \ s = \text{mw-Mutate } r \ f \ r' \ \# \ ws; \text{no-black-refs } s \rrbracket$   
 $\implies \text{no-black-refs } (s(\text{sys} := s \ \text{sys} \llbracket \text{heap} := (\text{sys-heap } s)(r := \text{map-option } (\lambda \text{obj. obj} \llbracket \text{obj-fields} := (\text{obj-fields } \text{obj})(f := r') \rrbracket) \ (\text{sys-heap } s \ r)),$   
 $\text{mem-store-buffers} := (\text{mem-store-buffers } (s \ \text{sys}))(p := ws) \rrbracket))$

**unfolding** *no-black-refs-def* **by** (*auto simp: fun-upd-apply dest: valid-W-invD*)

**lemma** *colours-blacken*[simp]:

*valid-W-inv s*  $\implies \text{black } b \ (s(gc := s \ gc \llbracket W := gc-W \ s - \{r\} \rrbracket)) \longleftrightarrow \text{black } b \ s \vee (r \in gc-W \ s \wedge b = r)$   
 $\llbracket r \in gc-W \ s; \text{valid-W-inv } s \rrbracket \implies \text{grey } g \ (s(gc := s \ gc \llbracket W := gc-W \ s - \{r\} \rrbracket)) \longleftrightarrow (\text{grey } g \ s \wedge g \neq r)$

**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]*:

$\llbracket \text{heap } (s \text{ sys}) \text{ } r' = \text{None}; \text{no-black-refs } s \rrbracket$   
 $\implies \text{no-black-refs } (s(\text{mutator } m' := s(\text{mutator } m')(\text{roots} := \text{roots}')), \text{sys} := s \text{ sys}(\text{heap} := (\text{sys-heap } s)(r' \mapsto$   
 $(\text{obj-mark} = \text{fl}, \text{obj-fields} = \text{Map.empty}, \text{obj-payload} = \text{Map.empty}))(\text{obj}))$   
 $\iff \text{fl} \neq \text{sys-fM } s \vee \text{grey } r' s$

**unfolding** *no-black-refs-def* **by** *simp*

**lemma** *no-black-refs-mo-co-mark[simp]*:

$\llbracket \text{ghost-honorary-grey } (s \text{ } p) = \{\}; \text{white } r \text{ } s \rrbracket$   
 $\implies \text{no-black-refs } (s(p := s \text{ } p(\text{ghost-honorary-grey} := \{r\}))) \iff \text{no-black-refs } s$

**unfolding** *no-black-refs-def* **by** *auto*

**lemma** *grey-protects-white-mark[simp]*:

**assumes** *ghg*: *ghost-honorary-grey*  $(s \text{ } p) = \{\}$   
**shows**  $(\exists g. (g \text{ grey-protects-white } w) (s(p := s \text{ } p(\text{ghost-honorary-grey} := \{r\}))))$   
 $\iff (\exists g'. (g' \text{ grey-protects-white } w) s) \vee (r \text{ has-white-path-to } w) s (\text{is } ?lhs \iff ?rhs)$

**proof**

**assume** *?lhs*  
**then obtain** *g* **where**  $(g \text{ grey-protects-white } w) (s(p := s \text{ } p(\text{ghost-honorary-grey} := \{r\})))$  **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**  $(g \text{ grey-protects-white } w) s$   
**from this ghg show** *?thesis*

**apply** *induct*

**apply** *force*

**unfolding** *grey-protects-white-def*

**apply**  $(\text{auto simp: fun-upd-apply})$

**done**

**next**

**assume**  $(r \text{ has-white-path-to } w) s$  **with** *ghg* **show** *?thesis*

**unfolding** *grey-protects-white-def* *has-white-path-to-def* **by**  $(\text{auto simp: fun-upd-apply})$

**qed**

**qed**

**lemma** *valid-refs-inv-dequeue-Mutate*:

**fixes**  $s :: ('field, 'mut, 'payload, 'ref) \text{ lsts}$

**assumes** *vri*: *valid-refs-inv*  $s$

**assumes** *sb*: *sys-mem-store-buffers*  $(\text{mutator } m') s = \text{mw-Mutate } r \text{ } f \text{ } \text{opt-r}' \# ws$

**shows** *valid-refs-inv*  $(s(\text{sys} := s \text{ sys}(\text{heap} := (\text{sys-heap } s)(r := \text{map-option } (\lambda \text{obj. obj}(\text{obj-fields} := (\text{obj-fields } \text{obj})(f := \text{opt-r}')))) (\text{sys-heap } s \text{ } r)))$ ,

$\text{mem-store-buffers} := (\text{mem-store-buffers } (s \text{ sys}))(\text{mutator } m' := ws))$  **(is**

*valid-refs-inv ?s')*

**proof**(*rule valid-refs-invI*)

**fix** *m*

**let** *?root* =  $\lambda m \text{ } x. \text{mut-m.root } m \text{ } x \vee \text{grey } x$

**fix**  $x \text{ } y$  **assume**  $xy: (x \text{ reaches } y) ?s'$  **and**  $x: ?root \text{ } m \text{ } x ?s'$

**from**  $xy$  **have**  $(\exists m \text{ } x. ?root \text{ } m \text{ } x s \wedge (x \text{ reaches } y) s) \wedge \text{valid-ref } y ?s'$

**unfolding** *reaches-def* **proof** *induct*

**case** *base* **with**  $x \text{ } sb \text{ } vri$  **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 (λobj. 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:*

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

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

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

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

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

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

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

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

**lemma**  $hs-get-work-locs-subseteq-ht-loaded-locs$ :  
 $hs-get-work-locs \subseteq ht-loaded-locs$   
**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**  $get-roots-UN-get-work-locs-subseteq-ghost-honorary-grey-empty-locs$ :  
 $get-roots-UN-get-work-locs \subseteq ghost-honorary-grey-empty-locs$   
**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*  $\subseteq$  *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*  $\subseteq$  *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* ( $\lambda r'. gc\text{-field-set} \otimes gc\text{-tmp-ref} \otimes (\lambda s. map\text{-option } obj\text{-fields } (sys\text{-heap } s \downarrow r')) \otimes (\lambda s. map\text{-option } obj\text{-mark } (sys\text{-heap } s \downarrow r')) \otimes sys\text{-fM} \otimes tso\text{-pending-mutate } gc$ )  
*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*) $\downarrow$  *field-set* := *UNIV*  $\downarrow$ )))

**unfolding** *obj-fields-marked-def* **by** (*simp add: fun-upd-apply*)

**lemma** *obj-fields-marked-invL-eq-imp:*

*eq-imp* ( $\lambda r' s. (AT \ s \ gc, s \downarrow gc, map\text{-option } obj\text{-fields } (sys\text{-heap } s \downarrow r'), map\text{-option } obj\text{-mark } (sys\text{-heap } s \downarrow r'), sys\text{-fM } s \downarrow, sys\text{-W } s \downarrow, tso\text{-pending-mutate } gc \ s \downarrow)$ )

*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 (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]:

$\llbracket \text{obj-at-field-on-heap } (\lambda r. \text{marked } r \ s) \ (gc\text{-tmp-ref } s) \ (gc\text{-field } s) \ s; \text{obj-fields-marked } s \rrbracket$   
 $\implies \text{obj-fields-marked } (s(gc := (s \ gc) \lfloor \text{field-set} := gc\text{-field-set } s - \{gc\text{-field } s\} \rfloor))$

**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:

$\text{eq-imp } (\lambda(-::\text{unit}) \ s. \ (AT \ s \ p, \ s \downarrow \ p, \ \text{sys-heap } s \downarrow, \ \text{sys-fM } s \downarrow, \ \text{sys-mem-store-buffers } p \ s \downarrow))$   
 $\text{mark-object-invL}$

**unfolding** eq-imp-def

**apply** clarsimp

**apply** (rename-tac s s')

**apply** (cut-tac s=s $\downarrow$  and s'=s' $\downarrow$  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* ( $\lambda r s. (AT\ s\ (mutator\ m),\ s\downarrow\ (mutator\ m),\ sys\text{-}heap\ s\downarrow\ r,\ sys\text{-}fM\ s\downarrow,\ sys\text{-}phase\ s\downarrow,\ tso\text{-}pending\text{-}mutate\ (mutator\ m)\ s\downarrow))$   
*(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*  $\forall r. marked\ r\ s\downarrow \longleftrightarrow marked\ r\ s'\downarrow$ )

**apply** (*subgoal-tac*  $\forall r. valid\text{-}null\text{-}ref\ r\ s\downarrow \longleftrightarrow valid\text{-}null\text{-}ref\ r\ s'\downarrow$ )

**apply** (*subgoal-tac*  $\forall r\ f\ opt\text{-}r'. mw\text{-}Mutate\ r\ f\ opt\text{-}r' \notin set\ (sys\text{-}mem\text{-}store\text{-}buffers\ (mutator\ m)\ s\downarrow) \longleftrightarrow mw\text{-}Mutate\ r\ f\ opt\text{-}r' \notin set\ (sys\text{-}mem\text{-}store\text{-}buffers\ (mutator\ m)\ s'\downarrow)$ )

**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*:  
 $\llbracket obj\text{-}mark\ 'ran\ (sys\text{-}heap\ (\downarrow GST = s, HST = \downarrow)) \subseteq \{gc\text{-}fM\ (\downarrow GST = s, HST = \downarrow); sys\text{-}fM\ (\downarrow GST = s, HST = \downarrow) = gc\text{-}fM\ (\downarrow GST = s, HST = \downarrow) \rrbracket$   
 $\implies strong\text{-}tricolour\text{-}inv\ (\downarrow GST = s, HST = \downarrow)$

**unfolding** *strong-tricolour-inv-def ran-def white-def* **by** (*auto split: obj-at-splits*)

**lemma** *init-no-grey-refs*:  
 $\llbracket gc\text{-}W\ (\downarrow GST = s, HST = \downarrow) = \{\}; \forall m. W\ (\downarrow GST = s, HST = \downarrow\ (mutator\ m)) = \{\}; sys\text{-}W\ (\downarrow GST = s, HST = \downarrow) = \{\};$   
 $gc\text{-}ghost\text{-}honorary\text{-}grey\ (\downarrow GST = s, HST = \downarrow) = \{\}; \forall m. ghost\text{-}honorary\text{-}grey\ (\downarrow GST = s, HST = \downarrow\ (mutator\ m)) = \{\}; sys\text{-}ghost\text{-}honorary\text{-}grey\ (\downarrow GST = s, HST = \downarrow) = \{\} \rrbracket$   
 $\implies no\text{-}grey\text{-}refs\ (\downarrow GST = s, HST = \downarrow)$

**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*:  
 $\llbracket valid\text{-}refs\ s; no\text{-}grey\text{-}refs\ s; \forall p. sys\text{-}mem\text{-}store\text{-}buffers\ p\ s = \{\}; \forall m. ghost\text{-}honorary\text{-}root\ (s\ (mutator\ m)) = \{\} \rrbracket$   
 $\implies valid\text{-}refs\text{-}inv\ s$

**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*:

$\llbracket \text{no-grey-refs } s; \forall p. \text{sys-mem-store-buffers } p \ s = [] \rrbracket$   
 $\implies \text{valid-W-inv } s$   
**unfolding** *valid-W-inv-def no-grey-refs-def grey-def WL-def* **by** *auto*

**lemma** *valid-refs-imp-reachable-snapshot-inv*:

$\llbracket \text{valid-refs } s; \text{obj-mark } \text{'ran } (\text{sys-heap } s) \subseteq \{\text{sys-fM } s\}; \forall p. \text{sys-mem-store-buffers } p \ s = []; \forall m. \text{ghost-honorary-root } (s \ (\text{mutator } m)) = \{\} \rrbracket$   
 $\implies \text{mut-m.reachable-snapshot-inv } m \ s$

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

**lemma** *init-inv-sys*:  $\forall s. \text{initial-state gc-system } s \longrightarrow \text{invs } (\text{GST} = s, \text{HST} = []) \downarrow$

**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*:  $\forall s. \text{initial-state gc-system } s \longrightarrow \text{mut-m.invsL } m \ (\text{GST} = s, \text{HST} = [])$

**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*:  $\forall s. \text{initial-state gc-system } s \longrightarrow \text{gc.invsL } (\text{GST} = s, \text{HST} = [])$

**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  $\wedge$  LSTP invs)*

**lemmas**  $I\text{-defs} = gc.invsL\text{-def } mut\text{-m}.invsL\text{-def } invsL\text{-def } invs\text{-def } I\text{-def}$

**context**  $gc\text{-system}$   
**begin**

**theorem**  $init\text{-inv}$ :  $\forall s. \text{initial-state } gc\text{-system } s \longrightarrow I \ (\! \mid GST = s, HST = [] \!)$

**unfolding**  $I\text{-def } invsL\text{-def}$  **by** ( $simp \text{ add: } init\text{-inv-sys } init\text{-inv-gc } init\text{-inv-mut}$ )

**end**

## 10 Noninterference

**lemma**  $mut\text{-del-barrier1-subseteq-mut-mo-valid-ref-locs[locset-cache]$ :

$mut\text{-m}.del\text{-barrier1-locs} \subseteq mut\text{-m}.mo\text{-valid-ref-locs}$

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

**lemma**  $mut\text{-del-barrier2-subseteq-mut-mo-valid-ref[locset-cache]$ :

$mut\text{-m}.ins\text{-barrier-locs} \subseteq mut\text{-m}.mo\text{-valid-ref-locs}$

**unfolding**  $mut\text{-m}.ins\text{-barrier-locs-def } mut\text{-m}.mo\text{-valid-ref-locs-def}$  **by** ( $auto \text{ intro: } append\text{-prefixD}$ )

**context**  $gc$   
**begin**

**lemma**  $obj\text{-fields-marked-locs-subseteq-hp-IdleMarkSweep-locs}$ :

$obj\text{-fields-marked-locs} \subseteq hp\text{-IdleMarkSweep-locs}$

**unfolding**  $gc.obj\text{-fields-marked-locs-def } gc.hp\text{-IdleMarkSweep-locs-def } gc.mark\text{-loop-locs-def } gc.mark\text{-loop-mo-locs-def}$

**apply** ( $clarsimp \text{ simp: } locset\text{-cache } loc\text{-defs}$ )

**apply** ( $drule \text{ mp}$ )

**apply** ( $auto \text{ intro: } append\text{-prefixD}$ )

**done**

**lemma**  $obj\text{-fields-marked-locs-subseteq-hs-in-sync-locs}$ :

$obj\text{-fields-marked-locs} \subseteq hs\text{-in-sync-locs}$

**unfolding**  $obj\text{-fields-marked-locs-def } hs\text{-in-sync-locs-def } hs\text{-done-locs-def } mark\text{-loop-mo-locs-def}$

**by** ( $auto \text{ simp: } loc\text{-defs } dest: \text{prefix-same-cases}$ )

**lemma**  $obj\text{-fields-marked-good-ref-subseteq-hp-IdleMarkSweep-locs}$ :

$obj\text{-fields-marked-good-ref-locs} \subseteq hp\text{-IdleMarkSweep-locs}$

**unfolding**  $obj\text{-fields-marked-good-ref-locs-def } mark\text{-loop-locs-def } hp\text{-IdleMarkSweep-locs-def } mark\text{-loop-mo-locs-def}$

**apply** ( $clarsimp \text{ simp: } loc\text{-defs}$ )

**apply** ( $drule \text{ mp}$ )

**apply** ( $auto \text{ intro: } append\text{-prefixD}$ )

**done**

**lemma**  $mark\text{-loop-mo-mark-loop-field-done-subseteq-hs-in-sync-locs}$ :

$obj\text{-fields-marked-good-ref-locs} \subseteq hs\text{-in-sync-locs}$

**unfolding**  $obj\text{-fields-marked-good-ref-locs-def } hs\text{-in-sync-locs-def } mark\text{-loop-mo-locs-def } hs\text{-done-locs-def}$

**by** ( $auto \text{ simp: } loc\text{-defs } dest: \text{prefix-same-cases}$ )

**lemma**  $no\text{-grey-refs-locs-subseteq-hs-in-sync-locs}$ :

$no\text{-grey-refs-locs} \subseteq hs\text{-in-sync-locs}$

**by** ( $auto \text{ simp: } no\text{-grey-refs-locs-def } black\text{-heap-locs-def } hs\text{-in-sync-locs-def } hs\text{-done-locs-def } sweep\text{-locs-def } loc\text{-defs}$   
 $dest: \text{prefix-same-cases}$ )

**lemma**  $get\text{-roots-UN-get-work-locs-subseteq-gc-W-empty-locs}$ :

$get\text{-}roots\text{-}UN\text{-}get\text{-}work\text{-}locs \subseteq gc\text{-}W\text{-}empty\text{-}locs$   
**unfolding**  $get\text{-}roots\text{-}UN\text{-}get\text{-}work\text{-}locs\text{-}def$   
**by** (*auto simp: hs-get-roots-locs-def hs-get-work-locs-def gc-W-empty-locs-def*)  
**end**

**declare**

$gc.obj\text{-}fields\text{-}marked\text{-}locs\text{-}subseq\text{-}hp\text{-}IdleMarkSweep\text{-}locs[locset\text{-}cache]$   
 $gc.obj\text{-}fields\text{-}marked\text{-}locs\text{-}subseq\text{-}hs\text{-}in\text{-}sync\text{-}locs[locset\text{-}cache]$   
 $gc.obj\text{-}fields\text{-}marked\text{-}good\text{-}ref\text{-}subseq\text{-}hp\text{-}IdleMarkSweep\text{-}locs[locset\text{-}cache]$   
 $gc.mark\text{-}loop\text{-}mo\text{-}mark\text{-}loop\text{-}field\text{-}done\text{-}subseq\text{-}hs\text{-}in\text{-}sync\text{-}locs[locset\text{-}cache]$   
 $gc.no\text{-}grey\text{-}refs\text{-}locs\text{-}subseq\text{-}hs\text{-}in\text{-}sync\text{-}locs[locset\text{-}cache]$   
 $gc.get\text{-}roots\text{-}UN\text{-}get\text{-}work\text{-}locs\text{-}subseq\text{-}gc\text{-}W\text{-}empty\text{-}locs[locset\text{-}cache]$

**lemma** *handshake-obj-fields-markedD:*

$\llbracket atS\ gc\ gc.obj\text{-}fields\text{-}marked\text{-}locs\ s;\ gc.handshake\text{-}invL\ s \rrbracket \implies sys\text{-}ghost\text{-}hs\text{-}phase\ s\downarrow = hp\text{-}IdleMarkSweep \wedge All$   
 $(ghost\text{-}hs\text{-}in\text{-}sync\ (s\downarrow\ sys))$

**unfolding**  $gc.handshake\text{-}invL\text{-}def$

**by** (*metis (no-types, lifting) atS-mono gc.obj-fields-marked-locs-subseq-hp-IdleMarkSweep-locs gc.obj-fields-marked-locs*)

**lemma** *obj-fields-marked-good-ref-locs-hp-phaseD:*

$\llbracket atS\ gc\ gc.obj\text{-}fields\text{-}marked\text{-}good\text{-}ref\text{-}locs\ s;\ gc.handshake\text{-}invL\ s \rrbracket$   
 $\implies sys\text{-}ghost\text{-}hs\text{-}phase\ s\downarrow = hp\text{-}IdleMarkSweep \wedge All\ (ghost\text{-}hs\text{-}in\text{-}sync\ (s\downarrow\ sys))$

**unfolding**  $gc.handshake\text{-}invL\text{-}def$

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

**lemma** *gc-marking-reaches-Mutate:*

**assumes**  $xy: \forall y. (x\ reaches\ y)\ s \longrightarrow valid\text{-}ref\ y\ s$

**assumes**  $xy: (x\ reaches\ y)\ (s(sys := s\ sys(heap := (sys\text{-}heap\ s)(r := map\text{-}option\ (\lambda obj. obj(heap := (obj\text{-}fields\ obj)(f := opt\text{-}r)))\ (sys\text{-}heap\ s\ r))),$   
 $mem\text{-}store\text{-}buffers := (mem\text{-}store\text{-}buffers\ (s\ sys))(p := ws)))$

**assumes**  $sb: sys\text{-}mem\text{-}store\text{-}buffers\ (mutator\ m)\ s = mw\text{-}Mutate\ r\ f\ opt\text{-}r'\ \# ws$

**assumes**  $vri: valid\text{-}refs\text{-}inv\ s$

**shows**  $valid\text{-}ref\ y\ s$

**proof** –

**from**  $xy\ xys$

**have**  $\exists z. z \in \{x\} \cup mut\text{-}m.tso\text{-}store\text{-}refs\ m\ s \wedge (z\ reaches\ y)\ s \wedge valid\text{-}ref\ y\ s$

**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
     $\{\!|$   $gc.fM\text{-}fA\text{-}invL \wedge gc.handshake\text{-}invL \wedge gc.obj\text{-}fields\text{-}marked\text{-}invL$ 
       $\wedge LSTP (fM\text{-}rel\text{-}inv \wedge handshake\text{-}phase\text{-}inv \wedge mutators\text{-}phase\text{-}inv \wedge tso\text{-}store\text{-}inv \wedge valid\text{-}refs\text{-}inv \wedge$ 
 $valid\text{-}W\text{-}inv) \!\!|\}$ 
    sys
     $\{\!|$   $gc.obj\text{-}fields\text{-}marked\text{-}invL \!\!|\}$ 
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 ( $\lambda m'. \text{sys-}W \otimes WL \text{ (mutator } m') \otimes \text{sys-ghost-hs-in-sync } m'$ )
    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 ( $\lambda(m', p) s. (AT \ s \ gc, \ s \downarrow \ gc, \ \text{sys-}W \ s \downarrow, \ WL \ p \ s \downarrow, \ \text{sys-ghost-hs-in-sync } m' \ s \downarrow)$ )
    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:
   $\llbracket \forall m. \text{mut-m.gc-W-empty-mut-inv } m \ s; \forall m. \text{sys-ghost-hs-in-sync } m \ s; WL \ gc \ s = \{\}; WL \ \text{sys } s = \{\} \rrbracket$ 
   $\implies \text{no-grey-refs } s$ 
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**

$\{ \text{handshake-invL} \wedge \text{obj-fields-marked-invL} \wedge \text{gc-W-empty-invL} \wedge \text{LSTP valid-W-inv} \}$   
 $\text{gc}$   
 $\{ \text{gc-W-empty-invL} \}$

**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**

$\{ \text{gc.gc-W-empty-invL} \} \text{ sys}$

**by** *vcg-chainsaw*

**lemma** *empty-WL-GC*:

$\llbracket \text{atS gc gc.get-roots-UN-get-work-locs s; gc.obj-fields-marked-invL s} \rrbracket \implies \text{gc-ghost-honorary-grey s} \downarrow = \{\}$

**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*:

$\llbracket \text{atS gc gc.get-roots-UN-get-work-locs s; gc.handshake-invL s} \rrbracket$

$\implies \text{sys-ghost-hs-phase s} \downarrow = \text{hp-IdleMarkSweep} \wedge \text{sys-hs-type s} \downarrow \in \{\text{ht-GetWork}, \text{ht-GetRoots}\}$

**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-IdleMarkSweep-locs*)

*gc.hs-get-work-locs-subseteq-hp-IdleMarkSweep-locs*)

**done**

**context** *gc*

**begin**

**lemma** *handshake-sweep-mark-endD*:

$\llbracket \text{atS gc no-grey-refs-locs s; handshake-invL s; handshake-phase-inv s} \downarrow \rrbracket$

$\implies \text{mut-m.mut-ghost-hs-phase m s} \downarrow = \text{hp-IdleMarkSweep} \wedge \text{All (ghost-hs-in-sync (s} \downarrow \text{ sys))}$

**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:*

$\llbracket \forall x. \text{mut-m.gc-W-empty-mut-inv } x \text{ s}\downarrow; \text{mutators-phase-inv } s\downarrow;$

$\text{mut-m.mut-ghost-honorary-grey } m \text{ s}\downarrow = \{\};$

$r \in \text{mut-m.mut-roots } m \text{ s}\downarrow \cup \text{mut-m.mut-ghost-honorary-root } m \text{ s}\downarrow; \text{white } r \text{ s}\downarrow;$

$\text{atS gc get-roots-UN-get-work-locs } s; \text{gc.handshake-invL } s; \text{gc.obj-fields-marked-invL } s;$

$\text{atS gc gc-W-empty-locs } s \longrightarrow \text{gc-W } s\downarrow = \{\};$

$\text{handshake-phase-inv } s\downarrow; \text{valid-W-inv } s\downarrow \rrbracket$

$\implies \text{mut-m.gc-W-empty-mut-inv } m' (s\downarrow(\text{mutator } m := s\downarrow(\text{mutator } m)(\text{ghost-honorary-grey} := \{r\})))$

**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-rootD
      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-rootD
      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*:

$\llbracket \text{sys-mem-store-buffers } p \text{ } s = \text{mw-Mark } r \text{ } fl \# ws; \text{mut-m.marked-deletions } m \text{ } s; \text{tso-store-inv } s; \text{valid-W-inv } s \rrbracket$   
 $\implies \text{mut-m.marked-deletions } m \text{ } (s(\text{sys} := s \text{ sys} \setminus \text{heap} := (\text{sys-heap } s)(r := \text{map-option } (\text{obj-mark-update } (\lambda-. fl)) (\text{sys-heap } s \text{ } r))), \text{mem-store-buffers} := (\text{mem-store-buffers } (s \text{ sys}))(p := ws)))$

**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*:

$\llbracket \text{sys-mem-store-buffers } (\text{mutator } m') \text{ } s = \text{mw-Mutate } r \text{ } f \text{ } \text{opt-r}' \# ws; \text{mut-m.marked-deletions } m \text{ } s; \text{mut-m.marked-insertions } m' \text{ } s \rrbracket$

$\implies \text{mut-m.marked-deletions } m \text{ } (s(\text{sys} := s \text{ sys} \setminus \text{heap} := (\text{sys-heap } s)(r := \text{map-option } (\lambda \text{obj. obj} \setminus \text{obj-fields} := (\text{obj-fields } \text{obj})(f := \text{opt-r}')) (\text{sys-heap } s \text{ } r))),$   
 $\text{mem-store-buffers} := (\text{mem-store-buffers } (s \text{ sys}))( (\text{mutator } m') := ws)))$

**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**  $(\exists g. (g \text{ grey-protects-white } w) (s(\text{sys} := s \text{ sys} \setminus \text{heap} := (\text{sys-heap } s)(r := \text{map-option } (\text{obj-mark-update } (\lambda-. fl)) (\text{sys-heap } s \text{ } r))), \text{mem-store-buffers} := (\text{mem-store-buffers } (s \text{ sys}))(p := ws))))$

$\longleftrightarrow (\exists g. (g \text{ grey-protects-white } w) \text{ } s) \text{ (is } (\exists g. (g \text{ grey-protects-white } w) \text{ } ?s') \longleftrightarrow ?rhs)$

**proof** (*rule iffI*)

**assume**  $\exists g. (g \text{ grey-protects-white } w) \text{ } ?s'$

**then obtain** *g* **where**  $(g \text{ grey-protects-white } w) \text{ } ?s'$  **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**  $(g' \text{ grey-protects-white } w) \text{ } s \text{ ..}$

**then show**  $\exists g. (g \text{ grey-protects-white } w) \text{ } ?s'$

**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:*

$\llbracket \text{sys-mem-store-buffers } p \text{ } s = \text{mw-Mark } r \text{ } fl \# ws; \text{mut-m.reachable-snapshot-inv } m \text{ } s; \text{valid-W-inv } s \rrbracket$   
 $\implies \text{mut-m.reachable-snapshot-inv } m \text{ } (s(\text{sys} := s \text{ sys}(\text{heap} := (\text{sys-heap } s)(r := \text{map-option } (\text{obj-mark-update } (\lambda-. fl)) (\text{sys-heap } s \text{ } r))), \text{mem-store-buffers} := (\text{mem-store-buffers } (s \text{ sys}))(p := ws)))$

**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-distinct valid-W-invD(1) fun-upd-apply*)

**done**

**lemma** *marked-insertions-dequeue-Mark:*

$\llbracket \text{sys-mem-store-buffers } p \text{ } s = \text{mw-Mark } r \text{ } fl \# ws; \text{mut-m.marked-insertions } m \text{ } s; \text{tso-writes-inv } s; \text{valid-W-inv } s \rrbracket$   
 $\implies \text{mut-m.marked-insertions } m \text{ } (s(\text{sys} := s \text{ sys}(\text{heap} := (\text{sys-heap } s)(r := \text{map-option } (\text{obj-mark-update } (\lambda-. fl)) (\text{sys-heap } s \text{ } r))), \text{mem-store-buffers} := (\text{mem-store-buffers } (s \text{ sys}))(p := ws)))$

**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:*

$\llbracket \text{sys-mem-store-buffers } p \text{ } s = \text{mw-Mutate } r \text{ } f \text{ } r' \# ws; \text{mut-m.marked-insertions } m \text{ } s \rrbracket$   
 $\implies \text{mut-m.marked-insertions } m \text{ } (s(\text{sys} := s \text{ sys}(\text{heap} := (\text{sys-heap } s)(r := \text{map-option } (\lambda \text{obj. obj}(\text{obj-fields} := (\text{obj-fields } \text{obj})(f := r')))) (\text{sys-heap } s \text{ } r))),$

$\text{mem-store-buffers} := (\text{mem-store-buffers } (s \text{ sys}))(p := ws)))$

**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  $(\exists g. (g \text{ grey-protects-white } w) (s(sys := s \text{ sys} \parallel heap := (sys\text{-heap } s)(r := \text{map-option } (\lambda obj. obj \parallel obj\text{-fields} := (obj\text{-fields } obj)(f := \text{opt-r}') \parallel) (sys\text{-heap } s \ r))),$ 

$$mem\text{-store-buffers} := (mem\text{-store-buffers } (s \ sys)) (mutator \ m := \ ws) \parallel))$$


$$\longleftrightarrow (\exists g. (g \text{ grey-protects-white } w) \ s) \ (\text{is } (\exists g. (g \text{ grey-protects-white } w) \ ?s') \longleftrightarrow ?rhs)$$

proof
  assume  $(\exists g. (g \text{ grey-protects-white } w) \ ?s')$ 
  then obtain g where  $(g \text{ grey-protects-white } w) \ ?s'$  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  $(\exists g. (g \text{ grey-protects-white } w) \ ?s')$ 
  proof(clarsimp)
    fix g assume  $(g \text{ grey-protects-white } w) \ s$ 
    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  $y$  uri have valid-ref  $y$   $s$  by auto
with reaches  $mi$   $md$   $rsi$   $sb$   $sti$   $y$  have ( $black\ y\ s \vee (\exists x. (x\ grey\ protects\ white\ y)\ s)$ )
  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 ( $\exists$ ) 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 ( $\exists$ ) 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 ( $fA\ rel\ inv \wedge fM\ rel\ inv \wedge handshake\ phase\ inv \wedge mutators\ phase\ inv \wedge strong\ tricolour\ inv \wedge sys\ phase\ inv$ 
 $\wedge tso\ store\ inv \wedge valid\ refs\ inv \wedge valid\ W\ inv$ )  $\!\!\}$ 
  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 ( $s\downarrow$  (mutator  $m$ )))
    case hp-IdleInit
    with  $\langle sys\ mem\ store\ buffers\ p\ s\downarrow = w\ \# \ ws \rangle\ spec[OF\ \langle mutators\ phase\ inv\ s\downarrow \rangle, \mathbf{where}\ x=m]$  mw-Mutate
    show ?thesis by simp
  next case hp-InitMark
  with  $\langle sys\ mem\ store\ buffers\ p\ s\downarrow = w\ \# \ ws \rangle\ spec[OF\ \langle mutators\ phase\ inv\ s\downarrow \rangle, \mathbf{where}\ x=m]$  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]:

$\{ \text{mut-store-del.mark-object-invL } m \wedge \text{mark-object-invL} \wedge \text{handshake-invL} \wedge \text{tso-lock-invL} \\
\wedge \text{LSTP } (\text{handshake-phase-inv} \wedge \text{valid-W-inv} \wedge \text{tso-store-inv} \wedge \text{valid-refs-inv}) \}$   
*mutator m*  
 $\{ \text{mut-store-del.mark-object-invL } m \}$

**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]:

$\{ \text{mut-get-roots.mark-object-invL } m \wedge \text{mark-object-invL} \wedge \text{handshake-invL} \wedge \text{tso-lock-invL} \\
\wedge \text{LSTP } (\text{handshake-phase-inv} \wedge \text{valid-W-inv} \wedge \text{tso-store-inv} \wedge \text{valid-refs-inv}) \}$   
*mutator m*  
 $\{ \text{mut-get-roots.mark-object-invL } m \}$

**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  
 $\{ \text{mark-object-invL} \}$  *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]:

$\{ fM-fA-invL \wedge gc-mark.mark-object-invL \wedge \text{obj-fields-marked-invL} \wedge \text{tso-lock-invL} \\
\wedge \text{LSTP } \text{valid-W-inv} \}$   
*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]:

$\{ fM-fA-invL \wedge \text{phase-invL} \wedge \text{obj-fields-marked-invL} \wedge gc-mark.mark-object-invL \\
\wedge \text{LSTP } (\text{tso-store-inv} \wedge \text{valid-W-inv} \wedge \text{valid-refs-inv}) \}$   
*gc*  
 $\{ \text{obj-fields-marked-invL} \}$



```

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.splits
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.splits
obj-at-splits)
  done

```

```

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.splits
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.splits
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**

$\{ \mid \text{mut-m.tso-lock-invL } m \wedge \text{mut-m.handshake-invL } m \wedge \text{mut-get-roots.mark-object-invL } m$   
 $\wedge \text{LSTP } (fM\text{-rel-inv} \wedge \text{handshake-phase-inv} \wedge \text{valid-W-inv} \wedge \text{tso-store-inv}) \}$   
*sys*

$\{ \mid \text{mut-get-roots.mark-object-invL } m \}$

**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-invL
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  $s\downarrow$ )  $s\downarrow$ )
apply (drule-tac  $y=gc-tmp-ref\ s\downarrow$  in valid-refs-invD( $\gamma$ ), 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 ( $\lambda(-::unit)$  s. (AT s gc,  $s\downarrow$  gc, tso-pending-phase gc s $\downarrow$ ))
    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  $\wedge$  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  $\wedge$  phase-invL  $\wedge$  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  $\wedge$  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  $\wedge$  LSTP (handshake-phase-inv  $\wedge$  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 ( $\lambda(-::\text{unit})\ s.\ (AT\ s\ gc,\ s\downarrow\ gc,\ sys\text{-ghost-hs-phase}\ s\downarrow,\ hs\text{-pending}\ (s\downarrow\ sys),\ ghost\text{-hs-in-sync}\ (s\downarrow\ sys),\ sys\text{-hs-type}\ s\downarrow))$ 
    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  $\wedge$  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* ( $\lambda(::unit) s. (AT\ s\ (mutator\ m),\ s\downarrow\ (mutator\ m),\ sys\text{-}hs\text{-}type\ s\downarrow,\ sys\text{-}hs\text{-}pending\ m\ s\downarrow,\ mem\text{-}store\text{-}buffers\ (s\downarrow\ sys)\ (mutator\ m)))$ )

*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]*:

$\{\!\!| handshake\text{-}invL \!\!\}$  *mutator m*

**by** *vcg-jackhammer*

**lemma** (**in** *mut-m'*) *handshake-invL[intro]*:

$\{\!\!| handshake\text{-}invL \!\!\}$  *mutator m'*

**by** *vcg-chainsaw*

**lemma** (**in** *gc*) *mut-handshake-invL[intro]*:

**notes** *fun-upd-apply[simp]*

**shows**

$\{\!\!| handshake\text{-}invL \wedge mut\text{-}m.handshake\text{-}invL\ m \!\!\} gc\ \{\!\!| mut\text{-}m.handshake\text{-}invL\ m \!\!\}$

**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**

$\{\!\!| mut\text{-}m.handshake\text{-}invL\ m \!\!\} sys$

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

**lemma** (**in** *mut-m*) *gc-handshake-invL[intro]*:

**notes** *fun-upd-apply[simp]*

**shows**

$\{\!\!| handshake\text{-}invL \wedge gc.handshake\text{-}invL \!\!\} mutator\ m\ \{\!\!| gc.handshake\text{-}invL \!\!\}$

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

**lemma** (**in** *mut-m*) *handshake-phase-inv[intro]*:

**notes** *fun-upd-apply[simp]*

**shows**

$\{\!\!| handshake\text{-}invL \wedge LSTP\ handshake\text{-}phase\text{-}inv \!\!\} mutator\ m\ \{\!\!| LSTP\ handshake\text{-}phase\text{-}inv \!\!\}$

**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* ( $\lambda(::unit) s. (AT\ s\ gc,\ s\downarrow\ gc,\ sys\text{-}fA\ s\downarrow,\ sys\text{-}fM\ s\downarrow,\ sys\text{-}mem\text{-}store\text{-}buffers\ gc\ s\downarrow))$ )

*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]*:

$\{\!\!| fM\text{-}fA\text{-}invL \!\!\} gc$

**by** *vcg-jackhammer*



```

lemma fM-rel-inv[intro]:
  notes fun-upd-apply[simp]
  shows
     $\{\!\!| fM\text{-}fA\text{-}invL \wedge handshake\text{-}invL \wedge tso\text{-}lock\text{-}invL \wedge LSTP\ fM\text{-}rel\text{-}inv \!\!\}$ 
    gc
     $\{\!\!| LSTP\ fM\text{-}rel\text{-}inv \!\!\}$ 
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
     $\{\!\!| fM\text{-}fA\text{-}invL \wedge handshake\text{-}invL \wedge LSTP\ fA\text{-}rel\text{-}inv \!\!\}$ 
    gc
     $\{\!\!| LSTP\ fA\text{-}rel\text{-}inv \!\!\}$ 
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]:
   $\{\!\!| gc.fM\text{-}fA\text{-}invL \!\!\}$  mutator m
by (vcg-chainsaw gc.fM-fA-invL-def)

lemma fM-rel-inv[intro]:
  notes fun-upd-apply[simp]
  shows
     $\{\!\!| LSTP\ fM\text{-}rel\text{-}inv \!\!\}$  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
     $\{\!\!| LSTP\ fA\text{-}rel\text{-}inv \!\!\}$  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]:

```

```

 $\{\!\!|$   $gc.fM-fA-invL \wedge LSTP (fA-rel-inv \wedge fM-rel-inv \wedge tso-store-inv)$   $\!\!\}$ 
   $sys$ 
 $\{\!\!|$   $gc.fM-fA-invL$   $\!\!\}$ 
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
     $\{\!\!| LSTP (fM-rel-inv \wedge tso-store-inv) \!\!\} sys \{\!\!| LSTP fM-rel-inv \!\!\}$ 
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
     $\{\!\!| LSTP (fA-rel-inv \wedge tso-store-inv) \!\!\} sys \{\!\!| LSTP fA-rel-inv \!\!\}$ 
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
     $\{\!\!| handshake-invL$ 
       $\wedge mark-object-invL$ 
       $\wedge mut-get-roots.mark-object-invL m$ 
       $\wedge mut-store-del.mark-object-invL m$ 
       $\wedge mut-store-ins.mark-object-invL m$ 
       $\wedge LSTP (fA-rel-inv \wedge fM-rel-inv \wedge handshake-phase-inv \wedge mutators-phase-inv \wedge phase-rel-inv \wedge$ 
 $sys-phase-inv \wedge valid-refs-inv) \!\!\}$ 
       $mutator m$ 

```

```

  { 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↓
  ⇒ 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↓ }
  ⇒ 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
     $\{ \mid LSTP (fA\text{-rel-inv} \wedge fM\text{-rel-inv} \wedge \text{handshake-phase-inv} \wedge \text{mutators-phase-inv} \wedge \text{phase-rel-inv} \wedge \text{sys-phase-inv} \wedge \text{tso-store-inv} \wedge \text{valid-W-inv}) \}$ 
    sys
     $\{ \mid LSTP \text{ sys-phase-inv} \}$ 
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]:
   $\llbracket \text{marked-insertions } s; (\exists r'. \text{opt-r}' = \text{Some } r') \longrightarrow \text{marked (the opt-r')} s \rrbracket$ 
   $\implies \text{marked-insertions}$ 
    (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]:
   $\llbracket \text{heap } (s \text{ sys}) \text{ } r' = \text{None}; \text{valid-refs-inv } s \rrbracket$ 
   $\implies \text{marked-insertions } (s(\text{mutator } m' := s(\text{mutator } m'))(\text{roots} := \text{roots}'), \text{ sys} := s \text{ sys}(\text{heap} := (\text{sys-heap } s)(r' \mapsto \text{obj}'))))$ 
   $\longleftrightarrow \text{marked-insertions } s$ 
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]:
   $\llbracket \text{marked-deletions } s; \text{obj-at-field-on-heap } (\lambda r'. \text{marked } r' s) \text{ } r \text{ } f \text{ } s \rrbracket$ 

```

```

 $\implies$  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 ]
 $\implies$  marked-deletions (s(mutator m' := s (mutator m') (roots := roots')), sys := s sys (heap := (sys-heap s) (r'  $\mapsto$ 
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 ( $\lambda$ (::unit) s. (AT s gc, s $\downarrow$  gc, sys-fM s $\downarrow$ , map-option obj-mark  $\circ$  sys-heap s $\downarrow$ ))
    sweep-loop-invL
apply (clarsimp simp: eq-imp-def inv)
apply (rename-tac s s')
apply (subgoal-tac  $\forall r$ . valid-ref r s $\downarrow$   $\longleftrightarrow$  valid-ref r s' $\downarrow$ )
  apply (subgoal-tac  $\forall P r$ . obj-at ( $\lambda$ obj. P (obj-mark obj)) r s $\downarrow$   $\longleftrightarrow$  obj-at ( $\lambda$ obj. P (obj-mark obj)) r s' $\downarrow$ )
    apply (frule-tac x= $\lambda$ mark. Some mark = gc-mark s' $\downarrow$  in spec)
    apply (frule-tac x= $\lambda$ mark. mark = sys-fM s' $\downarrow$  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  $\wedge$  phase-invL  $\wedge$  sweep-loop-invL  $\wedge$  tso-lock-invL
     $\wedge$  LSTP (phase-rel-inv  $\wedge$  mutators-phase-inv  $\wedge$  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  $\subseteq$  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  $\subseteq$  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  $\subseteq$  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  $\subseteq$  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  $\subseteq$  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  $\subseteq$  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  $\subseteq$  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:
   $\llbracket \text{atS } gc \text{ } ls \text{ } s; fM\text{-}fA\text{-}invL \text{ } s; ls \subseteq fM\text{-}tso\text{-}empty\text{-}locs \rrbracket \implies tso\text{-}pending\text{-}fM \text{ } gc \text{ } s \downarrow = []$ 
by (auto simp: fM-fA-invL-def dest: atS-mono)

lemma gc-sweep-loop-invL-locsE[rule-format]:
  (atS gc (sweep-locs  $\cup$  black-heap-locs) s  $\longrightarrow$  False)  $\implies$  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]:
   $\llbracket gc.fM\text{-}fA\text{-}invL \wedge gc.gc\text{-}W\text{-}empty\text{-}invL \wedge gc.sweep\text{-}loop\text{-}invL$ 
     $\wedge LSTP (tso\text{-}store\text{-}inv \wedge valid\text{-}W\text{-}inv) \rrbracket$ 
  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 - - -  $\longrightarrow$  -)+
  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-niE;
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-invL-niE;
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  $\wedge$  gc.handshake-invL  $\wedge$  gc.sweep-loop-invL
     $\wedge$  LSTP (mutators-phase-inv  $\wedge$  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 ]
   $\implies$  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:  $\forall r \in \text{mut-roots } s. \text{marked } r \ s$ 
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))  $\parallel$  W :=  $\{\}$ , ghost-hs-phase := ghp'),
  (sys := s sys (hs-pending := hp', W := sys-W s  $\cup$  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*

$\implies$  *reachable-snapshot-inv*

(*s*(*mutator m* := *s* (*mutator m*))  $\parallel$  *W* :=  $\{\}$ ),  
*sys* := *s sys* (*hs-pending* := *pending'*, *W* := *sys-W s*  $\cup$  *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*:

$\llbracket$  *reachable-snapshot-inv s; sys-load (mutator m) (mr-Ref r f) (s sys) = mv-Ref opt-r'; r  $\in$  mut-roots s; mut-ghost-honorary-root s = \{\}*  $\rrbracket$

$\implies$  *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
     $\wedge$  mark-object-invL
     $\wedge$  mut-get-roots.mark-object-invL m
     $\wedge$  mut-store-del.mark-object-invL m
     $\wedge$  mut-store-ins.mark-object-invL m
     $\wedge$  LSTP (handshake-phase-inv  $\wedge$  mutators-phase-inv  $\wedge$  phase-rel-inv  $\wedge$  sys-phase-inv  $\wedge$  fA-rel-inv  $\wedge$ 
fM-rel-inv  $\wedge$  valid-refs-inv  $\wedge$  strong-tricolour-inv  $\wedge$  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 ( $\forall$  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) }
    { 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*  $\implies$  *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]:  
 $\llbracket \text{no-black-refs } s; \text{gc-}W \text{ } s = \{\} \rrbracket$   
 $\implies \text{no-black-refs } (s(\text{gc} := s \text{gc} \llbracket W := \text{sys-}W \text{ } s \rrbracket, \text{sys} := s \text{sys} \llbracket W := \{\} \rrbracket))$   
**unfolding** *no-black-refs-def* **by** *simp*

**lemma** *marked-insertions-sweep-loop-free*[simp]:  
 $\llbracket \text{mut-m.marked-insertions } m \text{ } s; \text{white } r \text{ } s \rrbracket$   
 $\implies \text{mut-m.marked-insertions } m \text{ } (s(\text{sys} := (s \text{sys}) \llbracket \text{heap} := (\text{heap } (s \text{sys})) (r := \text{None}) \rrbracket))$   
**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**  
 $\llbracket \text{mut-m.marked-deletions } m \text{ } s; \text{mut-m.reachable-snapshot-inv } m \text{ } s; \text{no-grey-refs } s; \text{white } r \text{ } s \rrbracket$   
 $\implies \text{mut-m.marked-deletions } m \text{ } (s(\text{sys} := s \text{sys} \llbracket \text{heap} := (\text{sys-heap } s)(r := \text{None}) \rrbracket))$   
**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*:  
 $\llbracket \text{gc-field-set } s = \{\}; \text{obj-fields-marked } s; (\text{gc-tmp-ref } s \text{ points-to } w) \text{ } s; \text{white } w \text{ } s \rrbracket \implies \text{False}$   
**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*:  
 $\llbracket \text{gc-field-set } s = \{\}; \text{gc-tmp-ref } s \in \text{gc-}W \text{ } s; (\text{gc-tmp-ref } s \text{ has-white-path-to } w) \text{ } s; \text{obj-fields-marked } s; \text{valid-}W\text{-inv } s \rrbracket \implies w = \text{gc-tmp-ref } s$   
**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-snapshot-inv-def)
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

```

```

     $\wedge$  LSTP (fA-rel-inv  $\wedge$  fM-rel-inv  $\wedge$  handshake-phase-inv  $\wedge$  mutators-phase-inv  $\wedge$  strong-tricolour-inv  $\wedge$ 
sys-phase-inv  $\wedge$  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:  $\forall m'. \text{mut-m.reachable-snapshot-inv } m' s$*   
**assumes** *white r' s*  
**shows** *valid-refs-inv (s(sys := s sys( $\lfloor$ 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*  $\wedge$  *handshake-invL*  $\wedge$  *gc-W-empty-invL*  $\wedge$  *gc-mark.mark-object-invL*  $\wedge$  *obj-fields-marked-invL*  $\wedge$

$phase-invL \wedge sweep-loop-invL$   
 $\wedge LSTP (handshake-phase-inv \wedge mutators-phase-inv \wedge sys-phase-inv \wedge valid-refs-inv \wedge valid-W-inv) \} \}$   
 $gc$   
 $\{ \{ LSTP valid-refs-inv \} \}$   
**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**  $valid-refs-inv-discard-roots$ :  
 $\llbracket valid-refs-inv s; roots' \subseteq mut-roots s \rrbracket$   
 $\implies valid-refs-inv (s(mutator m := s (mutator m)(\llbracket roots := roots' \rrbracket)))$   
**unfolding**  $valid-refs-inv-def mut-m.reachable-def$  **by** ( $auto simp: fun-upd-apply$ )

**lemma**  $valid-refs-inv-load$ :  
 $\llbracket valid-refs-inv s; sys-load (mutator m) (mr-Ref r f) (s sys) = mv-Ref r'; r \in mut-roots s \rrbracket$   
 $\implies valid-refs-inv (s(mutator m := s (mutator m)(\llbracket roots := mut-roots s \cup Option.set-option r' \rrbracket)))$   
**unfolding**  $valid-refs-inv-def$  **by** ( $simp add: fun-upd-apply$ )

**lemma**  $valid-refs-inv-alloc$ :  
 $\llbracket valid-refs-inv s; sys-heap s r' = None \rrbracket$   
 $\implies valid-refs-inv (s(mutator m := s (mutator m)(\llbracket roots := insert r' (mut-roots s) \rrbracket), sys := s sys(\llbracket heap := (sys-heap s)(r' \mapsto (\llbracket obj-mark = fl, obj-fields = Map.empty, obj-payload = Map.empty \rrbracket)) \rrbracket)))$   
**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**  $valid-refs-inv-store-ins$ :  
 $\llbracket valid-refs-inv s; r \in mut-roots s; (\exists r'. opt-r' = Some r') \longrightarrow the opt-r' \in mut-roots s \rrbracket$   
 $\implies valid-refs-inv (s(mutator m := s (mutator m)(\llbracket ghost-honorary-root := \{\} \rrbracket,$   
 $sys := s sys(\llbracket mem-store-buffers := (mem-store-buffers (s sys))(mutator m := sys-mem-store-buffers$   
 $(mutator m) s @ [mw-Mutate r f opt-r'] \rrbracket) \rrbracket)))$   
**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**  $valid-refs-inv-deref-del$ :  
 $\llbracket valid-refs-inv s; sys-load (mutator m) (mr-Ref r f) (s sys) = mv-Ref opt-r'; r \in mut-roots s; mut-ghost-honorary-root s = \{\} \rrbracket$   
 $\implies valid-refs-inv (s(mutator m := s (mutator m)(\llbracket ghost-honorary-root := Option.set-option opt-r', ref := opt-r' \rrbracket)))$   
**unfolding**  $valid-refs-inv-def$  **by** ( $simp add: fun-upd-apply$ )

**lemma**  $valid-refs-inv-mo-co-mark$ :  
 $\llbracket r \in mut-roots s \cup mut-ghost-honorary-root s; mut-ghost-honorary-grey s = \{\}; valid-refs-inv s \rrbracket$   
 $\implies valid-refs-inv (s(mutator m := s (mutator m)(\llbracket ghost-honorary-grey := \{r\} \rrbracket)))$   
**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
     $\wedge$  mut-get-roots.mark-object-invL m
     $\wedge$  mut-store-del.mark-object-invL m
     $\wedge$  mut-store-ins.mark-object-invL m
     $\wedge$  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  $\wedge$  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:
   $\llbracket r \in W(s\ p); \text{valid-W-inv } s; p \neq q \rrbracket \implies r \notin WL\ q\ s$ 
   $\llbracket r \in W(s\ p); \text{valid-W-inv } s \rrbracket \implies r \notin \text{ghost-honorary-grey}(s\ q)$ 
   $\llbracket r \in \text{ghost-honorary-grey}(s\ p); \text{valid-W-inv } s \rrbracket \implies r \notin W(s\ q)$ 
   $\llbracket r \in \text{ghost-honorary-grey}(s\ p); \text{valid-W-inv } s; p \neq q \rrbracket \implies r \notin WL\ q\ s$ 
using marked-not-white unfolding valid-W-inv-def WL-def by (auto 0 5 split: obj-at-splits)

```

```

lemma valid-W-distinct-simps:
   $\llbracket r \in \text{ghost-honorary-grey}(s\ p); \text{valid-W-inv } s \rrbracket \implies (r \in \text{ghost-honorary-grey}(s\ q)) \longleftrightarrow (p = q)$ 
   $\llbracket r \in W(s\ p); \text{valid-W-inv } s \rrbracket \implies (r \in W(s\ q)) \longleftrightarrow (p = q)$ 
   $\llbracket r \in WL\ p\ s; \text{valid-W-inv } s \rrbracket \implies (r \in WL\ q\ s) \longleftrightarrow (p = q)$ 
  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:

```

$\llbracket \text{sys-mem-store-buffers } p \ s = \text{mw-Mutate } r' \ f \ r'' \ \# \ ws; \text{mw-Mark } r \ fl \in \text{set } ws; \text{valid-W-inv } s \rrbracket$   
 $\implies fl = \text{sys-fM } s \wedge r \in \text{ghost-honorary-grey } (s \ p) \wedge \text{tso-locked-by } p \ s \wedge \text{white } r \ s \wedge \text{filter is-mw-Mark } ws =$   
 $[\text{mw-Mark } r \ fl]$   
 $\llbracket \text{sys-mem-store-buffers } p \ s = \text{mw-fA } fl' \ \# \ ws; \text{mw-Mark } r \ fl \in \text{set } ws; \text{valid-W-inv } s \rrbracket$   
 $\implies fl = \text{sys-fM } s \wedge r \in \text{ghost-honorary-grey } (s \ p) \wedge \text{tso-locked-by } p \ s \wedge \text{white } r \ s \wedge \text{filter is-mw-Mark } ws =$   
 $[\text{mw-Mark } r \ fl]$   
 $\llbracket \text{sys-mem-store-buffers } p \ s = \text{mw-fM } fl' \ \# \ ws; \text{mw-Mark } r \ fl \in \text{set } ws; \text{valid-W-inv } s \rrbracket$   
 $\implies fl = \text{sys-fM } s \wedge r \in \text{ghost-honorary-grey } (s \ p) \wedge \text{tso-locked-by } p \ s \wedge \text{white } r \ s \wedge \text{filter is-mw-Mark } ws =$   
 $[\text{mw-Mark } r \ fl]$   
 $\llbracket \text{sys-mem-store-buffers } p \ s = \text{mw-Phase } ph \ \# \ ws; \text{mw-Mark } r \ fl \in \text{set } ws; \text{valid-W-inv } s \rrbracket$   
 $\implies fl = \text{sys-fM } s \wedge r \in \text{ghost-honorary-grey } (s \ p) \wedge \text{tso-locked-by } p \ s \wedge \text{white } r \ s \wedge \text{filter is-mw-Mark } ws =$   
 $[\text{mw-Mark } r \ fl]$   
**unfolding** *valid-W-inv-def white-def* **by** (*clarsimp dest!:* *spec[where x=p], blast*)**+**

**lemma** *valid-W-invE2*:

$\llbracket r \in W \ (s \ p); \text{valid-W-inv } s; \bigwedge \text{obj. obj-mark obj} = \text{sys-fM } s \implies P \ \text{obj} \rrbracket \implies \text{obj-at } P \ r \ s$   
 $\llbracket r \in \text{ghost-honorary-grey } (s \ p); \text{sys-mem-lock } s \neq \text{Some } p; \text{valid-W-inv } s; \bigwedge \text{obj. obj-mark obj} = \text{sys-fM } s \implies$   
 $P \ \text{obj} \rrbracket \implies \text{obj-at } P \ r \ s$

**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**

$\{ \text{LSTP } (fM\text{-rel-inv} \wedge \text{sys-phase-inv} \wedge \text{tso-store-inv} \wedge \text{valid-refs-inv} \wedge \text{valid-W-inv}) \}$   
 $\text{sys}$

$\{ \text{LSTP } \text{valid-W-inv} \}$

**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*:

$\llbracket \text{white } y \text{ } s; \text{sys-mem-lock } s = \text{Some } p; \text{valid-W-inv } s; p0 \neq p1 \rrbracket$   
 $\implies \text{WL } p0 \ (s(p := s \text{ } p \llbracket \text{ghost-honorary-grey} := \{y\} \rrbracket)) \cap \text{WL } p1 \ (s(p := s \text{ } p \llbracket \text{ghost-honorary-grey} := \{y\} \rrbracket))$   
 $= \{\}$

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

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

$\llbracket \text{valid-W-inv } s; \text{white } y \text{ } s; \text{sys-mem-lock } s = \text{Some } p; \text{filter is-mw-Mark } (\text{sys-mem-store-buffers } p \text{ } s) = \llbracket; p \neq \text{sys} \rrbracket$   
 $\implies \text{valid-W-inv } (s(p := s \text{ } p \llbracket \text{ghost-honorary-grey} := \{y\} \rrbracket), \text{sys} := s \text{ } \text{sys} \llbracket \text{mem-store-buffers} := (\text{mem-store-buffers } (s \text{ } \text{sys})) (p := \text{sys-mem-store-buffers } p \text{ } s \text{ } @ \llbracket \text{mw-Mark } y \text{ } (\text{sys-fM } s) \rrbracket) \rrbracket))$

**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-empty valid-W-inv-colours*)

**done**

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

$\llbracket \text{valid-W-inv } s; \text{sys-mem-lock } s = \text{None} \rrbracket$   
 $\implies \text{valid-W-inv } (s(\text{sys} := s \text{ } \text{sys} \llbracket \text{mem-lock} := \text{Some } p \rrbracket))$

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

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

$\llbracket \text{valid-W-inv } s; \text{marked } y \text{ } s; \text{ghost-honorary-grey } (s \text{ } p) = \{y\}; p \neq \text{sys} \rrbracket$   
 $\implies \text{valid-W-inv } (s(p := s \text{ } p \llbracket W := \text{insert } y \text{ } (W \text{ } (s \text{ } p)), \text{ghost-honorary-grey} := \{y\} \rrbracket))$

**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*:

$\llbracket \text{sys-mem-lock } s = \text{Some } p; \text{sys-mem-store-buffers } p \text{ } s = \llbracket;$

$\bigwedge r. r \in \text{ghost-honorary-grey } (s \text{ } p) \implies \text{marked } r \text{ } s;$

$\text{valid-W-inv } s$

$\rrbracket \implies \text{valid-W-inv } (s(\text{sys} := \text{mem-lock-update Map.empty } (s \text{ } \text{sys})))$

**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**

$\{ \text{gc-mark.mark-object-invL} \wedge \text{gc-W-empty-invL}$

$\wedge \text{obj-fields-marked-invL}$

$\wedge \text{sweep-loop-invL} \wedge \text{tso-lock-invL}$

$\wedge \text{LSTP valid-W-inv} \}$

*gc*

$\{ \text{LSTP valid-W-inv} \}$

**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]
done
qed

```

**lemma** (in mut-m) valid-W-inv[intro]:

**notes** if-split-asm[split del]

**notes** fun-upd-apply[simp]

**shows**

$\{ \text{handshake-invL} \wedge \text{mark-object-invL} \wedge \text{tso-lock-invL} \\ \wedge \text{mut-get-roots.mark-object-invL } m \\ \wedge \text{mut-store-del.mark-object-invL } m \\ \wedge \text{mut-store-ins.mark-object-invL } m \\ \wedge \text{LSTP } (fM\text{-rel-inv} \wedge \text{sys-phase-inv} \wedge \text{valid-refs-inv} \wedge \text{valid-W-inv}) \}$   
 mutator  $m$

$\{ \text{LSTP valid-W-inv} \}$

**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; assump-
tion+)
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)
  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

```

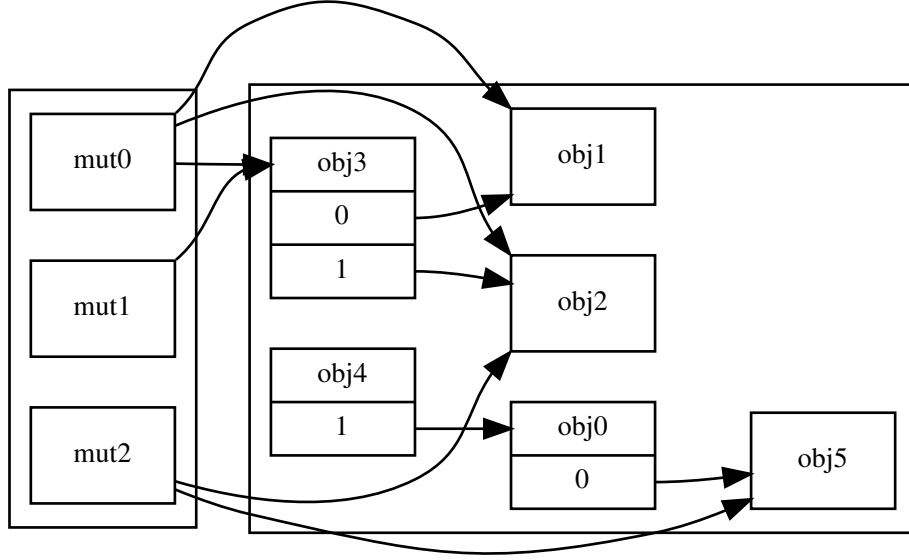


Figure 1: A concrete system state.

```

using I unfolding I-def invs-def valid-refs-def prerun-valid-def
apply clarsimp
apply (drule-tac x= $\sigma$  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  $\equiv$  undefined $\lfloor$  ghost-hs-phase := hp-IdleMarkSweep, ghost-honorary-grey := {}, ghost-honorary-r
:= {}, roots := {}, W := {}  $\rfloor$ 

```

```

context gc-system
begin

```

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

*sys-init-heap*  $\equiv$

```
[ 0  $\mapsto$  ( $\emptyset$  obj-mark = initial-mark,
    obj-fields = [ 0  $\mapsto$  5 ],
    obj-payload = Map.empty  $\emptyset$ ),
  1  $\mapsto$  ( $\emptyset$  obj-mark = initial-mark,
    obj-fields = Map.empty,
    obj-payload = Map.empty  $\emptyset$ ),
  2  $\mapsto$  ( $\emptyset$  obj-mark = initial-mark,
    obj-fields = Map.empty,
    obj-payload = Map.empty  $\emptyset$ ),
  3  $\mapsto$  ( $\emptyset$  obj-mark = initial-mark,
    obj-fields = [ 0  $\mapsto$  1 , 1  $\mapsto$  2 ],
    obj-payload = Map.empty  $\emptyset$ ),
  4  $\mapsto$  ( $\emptyset$  obj-mark = initial-mark,
    obj-fields = [ 1  $\mapsto$  0 ],
    obj-payload = Map.empty  $\emptyset$ ),
  5  $\mapsto$  ( $\emptyset$  obj-mark = initial-mark,
    obj-fields = Map.empty,
    obj-payload = Map.empty  $\emptyset$ )
]
```

**abbreviation** *mut-init-state0* :: *concrete-local-state* **where**

*mut-init-state0*  $\equiv$  *mut-common-init-state* ( $\emptyset$  *roots* := {1, 2, 3}  $\emptyset$ )

**abbreviation** *mut-init-state1* :: *concrete-local-state* **where**

*mut-init-state1*  $\equiv$  *mut-common-init-state* ( $\emptyset$  *roots* := {3}  $\emptyset$ )

**abbreviation** *mut-init-state2* :: *concrete-local-state* **where**

*mut-init-state2*  $\equiv$  *mut-common-init-state* ( $\emptyset$  *roots* := {2, 5}  $\emptyset$ )

**end**

**context** *gc-system*

**begin**

**abbreviation** *sys-init-state* :: *concrete-local-state* **where**

*sys-init-state*  $\equiv$

```
undefined( $\emptyset$  fA := initial-mark
    , fM := initial-mark
    , heap := sys-init-heap
    , hs-pending :=  $\langle$ False $\rangle$ 
    , hs-type := ht-GetRoots
    , mem-lock := None
    , mem-store-buffers :=  $\langle$  $\emptyset$  $\rangle$ 
    , phase := ph-Idle
    , W := {}
    , ghost-honorary-grey := {}
    , ghost-hs-in-sync :=  $\langle$ True $\rangle$ 
    , ghost-hs-phase := hp-IdleMarkSweep  $\emptyset$ )
```

**abbreviation** *gc-init-state* :: *concrete-local-state* **where**

*gc-init-state*  $\equiv$

```
undefined( $\emptyset$  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** *mut-init-states* :: (mut × concrete-local-state) list **where**  
*mut-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* mut-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.