

A Formalization of Assumptions and Guarantees for Compositional Noninterference

Sylvia Grewe, Heiko Mantel, Daniel Schoepe

September 13, 2023

Abstract

Research in information-flow security aims at developing methods to identify undesired information leaks within programs from private (high) sources to public (low) sinks. For a concurrent system, it is desirable to have compositional analysis methods that allow for analyzing each thread independently and that nevertheless guarantee that the parallel composition of successfully analyzed threads satisfies a global security guarantee. However, such a compositional analysis should not be overly pessimistic about what an environment might do with shared resources. Otherwise, the analysis will reject many intuitively secure programs.

The paper "Assumptions and Guarantees for Compositional Noninterference" by Mantel et. al. [MSS11] presents one solution for this problem: an approach for compositionally reasoning about noninterference in concurrent programs via rely-guarantee-style reasoning. We present an Isabelle/HOL formalization of the concepts and proofs of this approach.

The formalization includes the following parts:

- Notion of SIFUM-security and preliminary concepts:
`Preliminaries.thy`, `Security.thy`
- Compositionality proof: `Compositionality.thy`
- Example language: `Language.thy`
- Type system for ensuring SIFUM-security and soundness proof:
`TypeSystem.thy`
- Type system for ensuring sound use of modes and soundness proof: `LocallySoundUseOfModes.thy`

Contents

1	Preliminaries	2
2	Definition of the SIFUM-Security Property	4
2.1	Evaluation of Concurrent Programs	4
2.2	Low-equivalence and Strong Low Bisimulations	5

2.3	SIFUM-Security	6
2.4	Sound Mode Use	7
3	Compositionality Proof for SIFUM-Security Property	9
4	Language for Instantiating the SIFUM-Security Property	52
4.1	Syntax	52
4.2	Semantics	53
4.3	Semantic Properties	54
5	Type System for Ensuring SIFUM-Security of Commands	57
5.1	Typing Rules	57
5.2	Typing Soundness	59
6	Type System for Ensuring Locally Sound Use of Modes	84
6.1	Typing Rules	85
6.2	Soundness of the Type System	85

1 Preliminaries

```
theory Preliminaries
imports Main
begin
```

```
unbundle lattice-syntax
```

Possible modes for variables:

```
datatype Mode = AsmNoRead | AsmNoWrite | GuarNoRead | GuarNoWrite
```

We consider a two-element security lattice:

```
datatype Sec = High | Low
```

notation

```
less-eq (infix  $\sqsubseteq$  50) and
less (infix  $\sqsubset$  50)
```

Sec forms a (complete) lattice:

```
instantiation Sec :: complete-lattice
begin
```

definition *top-Sec-def*: $\top = High$

definition *sup-Sec-def*: $d1 \sqcup d2 = (if (d1 = High \vee d2 = High) then High else Low)$

definition *inf-Sec-def*: $d1 \sqcap d2 = (if (d1 = Low \vee d2 = Low) then Low else High)$

definition *bot-Sec-def*: $\perp = Low$
definition *less-eq-Sec-def*: $d1 \leq d2 = (d1 = d2 \vee d1 = Low)$
definition *less-Sec-def*: $d1 < d2 = (d1 = Low \wedge d2 = High)$
definition *Sup-Sec-def*: $\sqcup S = (if (High \in S) then High else Low)$
definition *Inf-Sec-def*: $\sqcap S = (if (Low \in S) then Low else High)$

instance

apply (*intro-classes*)
apply (*metis Sec.exhaust Sec.simps(2) less-Sec-def less-eq-Sec-def*)
apply (*metis less-eq-Sec-def*)
apply (*metis less-eq-Sec-def*)
apply (*metis less-eq-Sec-def*)
apply (*metis Sec.exhaust inf-Sec-def less-eq-Sec-def*)
apply (*metis Sec.exhaust inf-Sec-def less-eq-Sec-def*)
apply (*metis Sec.exhaust inf-Sec-def less-eq-Sec-def*)
apply (*metis Sec.exhaust less-eq-Sec-def sup-Sec-def*)
apply (*metis Sec.exhaust less-eq-Sec-def sup-Sec-def*)
apply (*metis Sec.exhaust Sec.simps(2) less-eq-Sec-def sup-Sec-def*)
apply (*metis (full-types) Inf-Sec-def Sec.exhaust less-eq-Sec-def*)
apply (*metis Inf-Sec-def Sec.exhaust less-eq-Sec-def*)
apply (*metis Sec.exhaust Sup-Sec-def less-eq-Sec-def*)
apply (*metis (full-types) Sup-Sec-def less-eq-Sec-def*)
apply (*metis (opaque-lifting, mono-tags) Inf-Sec-def empty-iff top-Sec-def*)
by (*metis (opaque-lifting, mono-tags) Sup-Sec-def bot-Sec-def empty-iff*)
end

Memories are mappings from variables to values

type-synonym (*'var, 'val*) *Mem* = *'var* \Rightarrow *'val*

A mode state maps modes to the set of variables for which the given mode is set.

type-synonym (*'var*) *Mds* = *Mode* \Rightarrow *'var set*

Local configurations:

type-synonym (*'com, 'var, 'val*) *LocalConf* = (*'com* \times *'var Mds*) \times (*'var, 'val*) *Mem*

Global configurations:

type-synonym (*'com, 'var, 'val*) *GlobalConf* = (*'com* \times *'var Mds*) *list* \times (*'var, 'val*) *Mem*

A locale to fix various parametric components in Mantel et. al, and assumptions about them:

locale *sifum-security* =
fixes *dma* :: *'Var* \Rightarrow *Sec*
fixes *stop* :: *'Com*
fixes *eval* :: (*'Com, 'Var, 'Val*) *LocalConf* *rel*
fixes *some-val* :: *'Val*

```

fixes some-val' :: 'Val
assumes stop-no-eval:  $\neg (((stop, mds), mem), ((c', mds'), mem')) \in eval$ 
assumes deterministic:  $\llbracket (lc, lc') \in eval; (lc, lc'') \in eval \rrbracket \implies lc' = lc''$ 
assumes finite-memory: finite  $\{(x::'Var). True\}$ 
assumes different-values: some-val  $\neq$  some-val'

```

end

2 Definition of the SIFUM-Security Property

```

theory Security
imports Main Preliminaries
begin

```

```

context sifum-security begin

```

2.1 Evaluation of Concurrent Programs

```

abbreviation eval-abv :: ('Com, 'Var, 'Val) LocalConf  $\Rightarrow$  (-, -, -) LocalConf  $\Rightarrow$ 
bool

```

```

  (infixl  $\rightsquigarrow$  70)

```

```

where

```

```

   $x \rightsquigarrow y \equiv (x, y) \in eval$ 

```

```

abbreviation conf-abv :: 'Com  $\Rightarrow$  'Var Mds  $\Rightarrow$  ('Var, 'Val) Mem  $\Rightarrow$  (-,-,-) Local-
Conf

```

```

  ( $\langle -, -, - \rangle [0, 0, 0] 1000$ )

```

```

where

```

```

   $\langle c, mds, mem \rangle \equiv ((c, mds), mem)$ 

```

```

inductive-set meval :: (-,-,-) GlobalConf rel

```

```

and meval-abv :: -  $\Rightarrow$  -  $\Rightarrow$  bool (infixl  $\rightarrow$  70)

```

```

where

```

```

   $conf \rightarrow conf' \equiv (conf, conf') \in meval$  |

```

```

  meval-intro [iff]:  $\llbracket (cms ! n, mem) \rightsquigarrow (cm', mem'); n < length cms \rrbracket \implies$ 

```

```

   $((cms, mem), (cms [n := cm'], mem')) \in meval$ 

```

```

inductive-cases meval-elim [elim!]:  $((cms, mem), (cms', mem')) \in meval$ 

```

```

abbreviation meval-clos :: -  $\Rightarrow$  -  $\Rightarrow$  bool (infixl  $\rightarrow^*$  70)

```

```

where

```

```

   $conf \rightarrow^* conf' \equiv (conf, conf') \in meval^*$ 

```

```

fun lc-set-var :: (-, -, -) LocalConf  $\Rightarrow$  'Var  $\Rightarrow$  'Val  $\Rightarrow$  (-, -, -) LocalConf

```

```

where

```

```

  lc-set-var (c, mem) x v = (c, mem (x := v))

```

fun *meval-k* :: *nat* \Rightarrow (*'Com*, *'Var*, *'Val*) *GlobalConf* \Rightarrow (-, -, -) *GlobalConf* \Rightarrow *bool*

where

meval-k 0 *c* *c'* = (*c* = *c'*) |

meval-k (*Suc* *n*) *c* *c'* = (\exists *c''*. *meval-k* *n* *c* *c''* \wedge *c''* \rightarrow *c'*)

abbreviation *meval-k-abv* :: *nat* \Rightarrow (-, -, -) *GlobalConf* \Rightarrow (-, -, -) *GlobalConf* \Rightarrow *bool*

(- \rightarrow_1 - [100, 100] 80)

where

gc \rightarrow_k *gc'* \equiv *meval-k* *k* *gc* *gc'*

2.2 Low-equivalence and Strong Low Bisimulations

definition *low-eq* :: (*'Var*, *'Val*) *Mem* \Rightarrow (-, -) *Mem* \Rightarrow *bool* (**infixl** $=^l$ 80)

where

*mem*₁ $=^l$ *mem*₂ \equiv (\forall *x*. *dma* *x* = *Low* \rightarrow *mem*₁ *x* = *mem*₂ *x*)

definition *low-mds-eq* :: *'Var* *Mds* \Rightarrow (*'Var*, *'Val*) *Mem* \Rightarrow (-, -) *Mem* \Rightarrow *bool*

(- $=^l$ - [100, 100] 80)

where

(*mem*₁ $=_{m_{ds}}^l$ *mem*₂) \equiv (\forall *x*. *dma* *x* = *Low* \wedge *x* \notin *m_{ds}* *AsmNoRead* \rightarrow *mem*₁ *x* = *mem*₂ *x*)

definition *m_{ds}_s* :: *'Var* *Mds* **where**

m_{ds}_s *x* = { }

lemma [*simp*]: *mem* $=^l$ *mem'* \implies *mem* $=_{m_{ds}}^l$ *mem'*

by (*simp* *add*: *low-mds-eq-def* *low-eq-def*)

lemma [*simp*]: (\forall *m_{ds}*. *mem* $=_{m_{ds}}^l$ *mem'*) \implies *mem* $=^l$ *mem'*

by (*auto* *simp*: *low-mds-eq-def* *low-eq-def*)

definition *closed-glob-consistent* :: ((*'Com*, *'Var*, *'Val*) *LocalConf*) *rel* \Rightarrow *bool*

where

closed-glob-consistent \mathcal{R} =

(\forall *c*₁ *m_{ds}* *mem*₁ *c*₂ *mem*₂. (\langle *c*₁, *m_{ds}*, *mem*₁ \rangle , \langle *c*₂, *m_{ds}*, *mem*₂ \rangle) \in \mathcal{R} \rightarrow

(\forall *x*. ((*dma* *x* = *High* \wedge *x* \notin *m_{ds}* *AsmNoWrite*) \rightarrow

(\forall *v*₁ *v*₂. (\langle *c*₁, *m_{ds}*, *mem*₁ (*x* := *v*₁) \rangle , \langle *c*₂, *m_{ds}*, *mem*₂ (*x* := *v*₂) \rangle) \in \mathcal{R})) \wedge

((*dma* *x* = *Low* \wedge *x* \notin *m_{ds}* *AsmNoWrite*) \rightarrow

(\forall *v*. (\langle *c*₁, *m_{ds}*, *mem*₁ (*x* := *v*) \rangle , \langle *c*₂, *m_{ds}*, *mem*₂ (*x* := *v*) \rangle) \in \mathcal{R})))))

definition *strong-low-bisim-mm* :: ((*'Com*, *'Var*, *'Val*) *LocalConf*) *rel* \Rightarrow *bool*

where

strong-low-bisim-mm $\mathcal{R} \equiv$
sym $\mathcal{R} \wedge$
closed-glob-consistent $\mathcal{R} \wedge$
 $(\forall c_1 \text{ mds } mem_1 c_2 \text{ mem}_2. (\langle c_1, \text{ mds}, mem_1 \rangle, \langle c_2, \text{ mds}, mem_2 \rangle) \in \mathcal{R} \longrightarrow$
 $(mem_1 =_{\text{mds}} mem_2) \wedge$
 $(\forall c_1' \text{ mds}' mem_1'. \langle c_1, \text{ mds}, mem_1 \rangle \rightsquigarrow \langle c_1', \text{ mds}', mem_1' \rangle \longrightarrow$
 $(\exists c_2' \text{ mem}_2'. \langle c_2, \text{ mds}, mem_2 \rangle \rightsquigarrow \langle c_2', \text{ mds}', mem_2' \rangle \wedge$
 $(\langle c_1', \text{ mds}', mem_1' \rangle, \langle c_2', \text{ mds}', mem_2' \rangle) \in \mathcal{R}))$

inductive-set *mm-equiv* :: (*'Com*, *'Var*, *'Val*) *LocalConf*) *rel*

and *mm-equiv-abv* :: (*'Com*, *'Var*, *'Val*) *LocalConf* \Rightarrow

(*'Com*, *'Var*, *'Val*) *LocalConf* \Rightarrow *bool* (**infix** ≈ 60)

where

mm-equiv-abv $x y \equiv (x, y) \in \text{mm-equiv} \mid$

mm-equiv-intro [*iff*]: $\llbracket \text{strong-low-bisim-mm } \mathcal{R} ; (lc_1, lc_2) \in \mathcal{R} \rrbracket \Longrightarrow (lc_1, lc_2) \in \text{mm-equiv}$

inductive-cases *mm-equiv-elim* [*elim*]: $\langle c_1, \text{ mds}, mem_1 \rangle \approx \langle c_2, \text{ mds}, mem_2 \rangle$

definition *low-indistinguishable* :: *'Var* *Mds* \Rightarrow *'Com* \Rightarrow *'Com* \Rightarrow *bool*

(- \sim_1 - [*100*, *100*] *80*)

where $c_1 \sim_{\text{mds}} c_2 = (\forall mem_1 mem_2. mem_1 =_{\text{mds}} mem_2 \longrightarrow$
 $\langle c_1, \text{ mds}, mem_1 \rangle \approx \langle c_2, \text{ mds}, mem_2 \rangle)$

2.3 SIFUM-Security

definition *com-sifum-secure* :: *'Com* \Rightarrow *bool*

where *com-sifum-secure* $c = c \sim_{\text{mds}_s} c$

definition *add-initial-modes* :: *'Com* *list* \Rightarrow (*'Com* \times *'Var* *Mds*) *list*

where *add-initial-modes* $cmds = \text{zip } cmds \text{ (replicate (length } cmds) \text{ mds}_s)$

definition *no-assumptions-on-termination* :: *'Com* *list* \Rightarrow *bool*

where *no-assumptions-on-termination* $cmds =$

($\forall mem \text{ mem}' \text{ cms}'.$

$(\text{add-initial-modes } cmds, mem) \rightarrow^* (cms', mem') \wedge$

$\text{list-all } (\lambda c. c = \text{stop}) (\text{map fst } cms') \longrightarrow$

$(\forall mds' \in \text{set } (\text{map snd } cms'). mds' \text{ AsmNoRead} = \{\} \wedge mds' \text{ AsmNoWrite} = \{\}))$

definition *prog-sifum-secure* :: *'Com* *list* \Rightarrow *bool*

where *prog-sifum-secure* $cmds =$

(*no-assumptions-on-termination* $cmds \wedge$

$(\forall mem_1 mem_2. mem_1 =^l mem_2 \longrightarrow$

$(\forall k \text{ cms}_1' mem_1'.$

$(\text{add-initial-modes } cmds, mem_1) \rightarrow_k (cms_1', mem_1') \longrightarrow$

$(\exists cms_2' mem_2'. (\text{add-initial-modes } cmds, mem_2) \rightarrow_k (cms_2', mem_2') \wedge$

$$\begin{aligned}
& \text{map snd cms}_1' = \text{map snd cms}_2' \wedge \\
& \text{length cms}_2' = \text{length cms}_1' \wedge \\
& (\forall x. \text{dma } x = \text{Low} \wedge (\forall i < \text{length cms}_1'. \\
& \quad x \notin \text{snd} (\text{cms}_1' ! i) \text{AsmNoRead}) \longrightarrow \text{mem}_1' x = \text{mem}_2' x))
\end{aligned}$$

2.4 Sound Mode Use

definition *doesnt-read* :: 'Com ⇒ 'Var ⇒ bool

where

$$\begin{aligned}
& \text{doesnt-read } c \ x = (\forall \text{ mds mem } c' \text{ mds}' \text{ mem}'. \\
& \langle c, \text{ mds}, \text{ mem} \rangle \rightsquigarrow \langle c', \text{ mds}', \text{ mem}' \rangle \longrightarrow \\
& ((\forall v. \langle c, \text{ mds}, \text{ mem} (x := v) \rangle \rightsquigarrow \langle c', \text{ mds}', \text{ mem}' (x := v) \rangle) \vee \\
& (\forall v. \langle c, \text{ mds}, \text{ mem} (x := v) \rangle \rightsquigarrow \langle c', \text{ mds}', \text{ mem}' \rangle))
\end{aligned}$$

definition *doesnt-modify* :: 'Com ⇒ 'Var ⇒ bool

where

$$\begin{aligned}
& \text{doesnt-modify } c \ x = (\forall \text{ mds mem } c' \text{ mds}' \text{ mem}'. (\langle c, \text{ mds}, \text{ mem} \rangle \rightsquigarrow \langle c', \text{ mds}', \\
& \text{ mem}' \rangle) \longrightarrow \\
& \quad \text{mem } x = \text{mem}' x)
\end{aligned}$$

inductive-set *loc-reach* :: ('Com, 'Var, 'Val) LocalConf ⇒ ('Com, 'Var, 'Val) LocalConf set

for *lc* :: (-, -, -) LocalConf

where

$$\begin{aligned}
& \text{refl} : \langle \text{fst } (\text{fst } lc), \text{ snd } (\text{fst } lc), \text{ snd } lc \rangle \in \text{loc-reach } lc \mid \\
& \text{step} : \llbracket \langle c', \text{ mds}', \text{ mem}' \rangle \in \text{loc-reach } lc; \\
& \quad \langle c', \text{ mds}', \text{ mem}' \rangle \rightsquigarrow \langle c'', \text{ mds}'', \text{ mem}'' \rangle \rrbracket \Longrightarrow \\
& \quad \langle c'', \text{ mds}'', \text{ mem}'' \rangle \in \text{loc-reach } lc \mid \\
& \text{mem-diff} : \llbracket \langle c', \text{ mds}', \text{ mem}' \rangle \in \text{loc-reach } lc; \\
& \quad (\forall x \in \text{ mds}' \text{AsmNoWrite}. \text{ mem}' x = \text{ mem}'' x) \rrbracket \Longrightarrow \\
& \quad \langle c', \text{ mds}', \text{ mem}' \rangle \in \text{loc-reach } lc
\end{aligned}$$

definition *locally-sound-mode-use* :: (-, -, -) LocalConf ⇒ bool

where

$$\begin{aligned}
& \text{locally-sound-mode-use } lc = \\
& (\forall c' \text{ mds}' \text{ mem}'. \langle c', \text{ mds}', \text{ mem}' \rangle \in \text{loc-reach } lc \longrightarrow \\
& \quad (\forall x. (x \in \text{ mds}' \text{GuarNoRead} \longrightarrow \text{doesnt-read } c' x) \wedge \\
& \quad (x \in \text{ mds}' \text{GuarNoWrite} \longrightarrow \text{doesnt-modify } c' x)))
\end{aligned}$$

definition *compatible-modes* :: ('Var Mds) list ⇒ bool

where

$$\begin{aligned}
& \text{compatible-modes } \text{mdss} = (\forall (i :: \text{nat}) x. i < \text{length } \text{mdss} \longrightarrow \\
& \quad (x \in (\text{mdss} ! i) \text{AsmNoRead} \longrightarrow \\
& \quad (\forall j < \text{length } \text{mdss}. j \neq i \longrightarrow x \in (\text{mdss} ! j) \text{GuarNoRead})) \wedge \\
& \quad (x \in (\text{mdss} ! i) \text{AsmNoWrite} \longrightarrow \\
& \quad (\forall j < \text{length } \text{mdss}. j \neq i \longrightarrow x \in (\text{mdss} ! j) \text{GuarNoWrite})))
\end{aligned}$$

definition *reachable-mode-states* :: ('Com, 'Var, 'Val) GlobalConf ⇒ (('Var Mds)

list) set

where *reachable-mode-states* gc =
{mdss. (\exists cms' mem'. gc \rightarrow^* (cms', mem') \wedge map snd cms' = mdss)}

definition *globally-sound-mode-use* :: ('Com, 'Var, 'Val) GlobalConf \Rightarrow bool

where *globally-sound-mode-use* gc =
(\forall mdss. mdss \in *reachable-mode-states* gc \longrightarrow *compatible-modes* mdss)

primrec *sound-mode-use* :: (-, -, -) GlobalConf \Rightarrow bool

where

sound-mode-use (cms, mem) =
(list-all (λ cm. *locally-sound-mode-use* (cm, mem))) cms \wedge
globally-sound-mode-use (cms, mem))

lemma *mm-equiv-sym*:

assumes *equivalent*: $\langle c_1, mds_1, mem_1 \rangle \approx \langle c_2, mds_2, mem_2 \rangle$

shows $\langle c_2, mds_2, mem_2 \rangle \approx \langle c_1, mds_1, mem_1 \rangle$

proof –

from *equivalent* **obtain** \mathcal{R}

where *R-bisim*: *strong-low-bisim-mm* $\mathcal{R} \wedge (\langle c_1, mds_1, mem_1 \rangle, \langle c_2, mds_2, mem_2 \rangle) \in \mathcal{R}$

by (*metis mm-equiv.simps*)

hence *sym* \mathcal{R}

by (*auto simp: strong-low-bisim-mm-def*)

hence $(\langle c_2, mds_2, mem_2 \rangle, \langle c_1, mds_1, mem_1 \rangle) \in \mathcal{R}$

by (*metis R-bisim symE*)

thus *?thesis*

by (*metis R-bisim mm-equiv.intros*)

qed

lemma *low-indistinguishable-sym*: $lc \sim_{mds} lc' \Longrightarrow lc' \sim_{mds} lc$

by (*auto simp: mm-equiv-sym low-indistinguishable-def low-mds-eq-def*)

lemma *mm-equiv-glob-consistent*: *closed-glob-consistent mm-equiv*

unfolding *closed-glob-consistent-def*

apply *clarify*

apply (*erule mm-equiv-elim*)

by (*auto simp: strong-low-bisim-mm-def closed-glob-consistent-def*)

lemma *mm-equiv-strong-low-bisim*: *strong-low-bisim-mm mm-equiv*

unfolding *strong-low-bisim-mm-def*

proof (*auto*)

show *closed-glob-consistent mm-equiv* **by** (*rule mm-equiv-glob-consistent*)

next

fix c_1 mds mem_1 c_2 mem_2 x

assume $\langle c_1, mds, mem_1 \rangle \approx \langle c_2, mds, mem_2 \rangle$

then obtain \mathcal{R} **where**

strong-low-bisim-mm $\mathcal{R} \wedge (\langle c_1, mds, mem_1 \rangle, \langle c_2, mds, mem_2 \rangle) \in \mathcal{R}$


```

    by blast
  thus  $mem_1 =_{m_{ds}^l} mem_2$  by (auto simp: strong-low-bisim-mm-def)
next
  fix  $c_1 :: 'Com$ 
  fix  $m_{ds} mem_1 c_2 mem_2 c_1' m_{ds}' mem_1'$ 
  let  $?lc_1 = \langle c_1, m_{ds}, mem_1 \rangle$  and
       $?lc_1' = \langle c_1', m_{ds}', mem_1' \rangle$  and
       $?lc_2 = \langle c_2, m_{ds}, mem_2 \rangle$ 
  assume  $?lc_1 \approx ?lc_2$ 
  then obtain  $\mathcal{R}$  where strong-low-bisim-mm  $\mathcal{R} \wedge (?lc_1, ?lc_2) \in \mathcal{R}$ 
    by (rule mm-equiv-elim, blast)
  moreover assume  $?lc_1 \rightsquigarrow ?lc_1'$ 
  ultimately show  $\exists c_2' mem_2'. ?lc_2 \rightsquigarrow \langle c_2', m_{ds}', mem_2' \rangle \wedge ?lc_1' \approx \langle c_2',$ 
 $m_{ds}', mem_2' \rangle$ 
    by (simp add: strong-low-bisim-mm-def mm-equiv-sym, blast)
next
  show sym mm-equiv
    by (auto simp: sym-def mm-equiv-sym)
qed

end

end

```

3 Compositionality Proof for SIFUM-Security Property

```

theory Compositionality
imports Main Security
begin

```

```

context sifum-security
begin

```

```

definition differing-vars :: ('Var, 'Val) Mem  $\Rightarrow$  (-, -) Mem  $\Rightarrow$  'Var set
where
  differing-vars  $mem_1 mem_2 = \{x. mem_1 x \neq mem_2 x\}$ 

```

```

definition differing-vars-lists :: ('Var, 'Val) Mem  $\Rightarrow$  (-, -) Mem  $\Rightarrow$ 
  ((-, -) Mem  $\times$  (-, -) Mem) list  $\Rightarrow$  nat  $\Rightarrow$  'Var set
where
  differing-vars-lists  $mem_1 mem_2 mems i =$ 
  (differing-vars  $mem_1 (fst (mems ! i)) \cup$  differing-vars  $mem_2 (snd (mems ! i)))$ 

```

```

lemma differing-finite: finite (differing-vars  $mem_1 mem_2$ )
by (metis UNIV-def Un-UNIV-left finite-Un finite-memory)

```

lemma *differing-lists-finite*: *finite (differing-vars-lists mem₁ mem₂ mems i)*
by (*simp add: differing-finite differing-vars-lists-def*)

definition *subst* :: ('a \rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)
where
subst *f mem* = (λ *x*. *case* *f x of*
 None \Rightarrow *mem x* |
 Some v \Rightarrow *v*)

abbreviation *subst-abv* :: ('a \Rightarrow 'b) \Rightarrow ('a \rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) (- [\mapsto] [900, 0]
1000)
where
f [\mapsto σ] \equiv *subst* σ *f*

lemma *subst-not-in-dom* : $\llbracket x \notin \text{dom } \sigma \rrbracket \Longrightarrow \text{mem } [\mapsto \sigma] x = \text{mem } x$
by (*simp add: domIff subst-def*)

fun *makes-compatible* ::
('Com, 'Var, 'Val) *GlobalConf* \Rightarrow
('Com, 'Var, 'Val) *GlobalConf* \Rightarrow
((-, -) *Mem* \times (-, -) *Mem*) *list* \Rightarrow
bool
where
makes-compatible (*cms₁*, *mem₁*) (*cms₂*, *mem₂*) *mems* =
(*length cms₁* = *length cms₂* \wedge *length cms₁* = *length mems* \wedge
 $(\forall i. i < \text{length } cms_1 \longrightarrow$
 $(\forall \sigma. \text{dom } \sigma = \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i \longrightarrow$
 $(cms_1 ! i, (\text{fst } (mems ! i)) [\mapsto \sigma]) \approx (cms_2 ! i, (\text{snd } (mems ! i)) [\mapsto \sigma])) \wedge$
 $(\forall x. (mem_1 x = mem_2 x \vee \text{dma } x = \text{High}) \longrightarrow$
 $x \notin \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i)) \wedge$
 $((\text{length } cms_1 = 0 \wedge mem_1 =^l mem_2) \vee (\forall x. \exists i. i < \text{length } cms_1 \wedge$
 $x \notin \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i)))$

lemma *makes-compatible-intro* [*intro*]:
 $\llbracket \text{length } cms_1 = \text{length } cms_2 \wedge \text{length } cms_1 = \text{length } mems;$
 $(\wedge i \sigma. \llbracket i < \text{length } cms_1; \text{dom } \sigma = \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i \rrbracket$
 \Longrightarrow
 $(cms_1 ! i, (\text{fst } (mems ! i)) [\mapsto \sigma]) \approx (cms_2 ! i, (\text{snd } (mems ! i)) [\mapsto \sigma]);$
 $(\wedge i x. \llbracket i < \text{length } cms_1; mem_1 x = mem_2 x \vee \text{dma } x = \text{High} \rrbracket \Longrightarrow$
 $x \notin \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i);$
 $(\text{length } cms_1 = 0 \wedge mem_1 =^l mem_2) \vee$
 $(\forall x. \exists i. i < \text{length } cms_1 \wedge x \notin \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i) \rrbracket$
 \Longrightarrow
makes-compatible (*cms₁*, *mem₁*) (*cms₂*, *mem₂*) *mems*
by *auto*

lemma *compat-low*:

$\llbracket \text{makes-compatible } (cms_1, mem_1) (cms_2, mem_2) mems;$
 $i < \text{length } cms_1;$
 $x \in \text{differing-vars-lists } mem_1 mem_2 mems i \rrbracket \implies dma\ x = Low$

proof –

assume $i < \text{length } cms_1$ **and** $*$: $x \in \text{differing-vars-lists } mem_1 mem_2 mems i$ **and**
 $\text{makes-compatible } (cms_1, mem_1) (cms_2, mem_2) mems$

then have

$(mem_1\ x = mem_2\ x \vee dma\ x = High) \longrightarrow x \notin \text{differing-vars-lists } mem_1 mem_2 mems\ i$

by (*simp add: Let-def, blast*)

with $*$ **show** $dma\ x = Low$

by (*cases dma x*) *blast*

qed

lemma *compat-different*:

$\llbracket \text{makes-compatible } (cms_1, mem_1) (cms_2, mem_2) mems;$
 $i < \text{length } cms_1;$
 $x \in \text{differing-vars-lists } mem_1 mem_2 mems i \rrbracket \implies mem_1\ x \neq mem_2\ x \wedge dma\ x = Low$
by (*cases dma x, auto*)

lemma *sound-modes-no-read* :

$\llbracket \text{sound-mode-use } (cms, mem); x \in (\text{map snd } cms\ !\ i)\ \text{GuarNoRead}; i < \text{length } cms \rrbracket \implies$
 $\text{doesnt-read } (fst\ (cms\ !\ i))\ x$

proof –

fix $cms\ mem\ x\ i$

assume *sound-modes: sound-mode-use* (cms, mem) **and** $i < \text{length } cms$

hence *locally-sound-mode-use* $(cms\ !\ i, mem)$

by (*auto simp: sound-mode-use-def list-all-length*)

moreover

assume $x \in (\text{map snd } cms\ !\ i)\ \text{GuarNoRead}$

ultimately show $\text{doesnt-read } (fst\ (cms\ !\ i))\ x$

apply (*simp add: locally-sound-mode-use-def*)

by (*metis prod.exhaust <i < length cms> fst-conv loc-reach.refl nth-map snd-conv*)

qed

lemma *compat-different-vars*:

$\llbracket \text{fst } (mems\ !\ i)\ x = \text{snd } (mems\ !\ i)\ x;$
 $x \notin \text{differing-vars-lists } mem_1 mem_2 mems\ i \rrbracket \implies$
 $mem_1\ x = mem_2\ x$

proof –

assume $x \notin \text{differing-vars-lists } mem_1 mem_2 mems\ i$

hence $\text{fst } (mems\ !\ i)\ x = mem_1\ x \wedge \text{snd } (mems\ !\ i)\ x = mem_2\ x$

by (*simp add: differing-vars-lists-def differing-vars-def*)

moreover assume $\text{fst } (mems\ !\ i)\ x = \text{snd } (mems\ !\ i)\ x$

ultimately show $mem_1\ x = mem_2\ x$ **by** *auto*

qed

lemma *differing-vars-subst* [rule-format]:
assumes $dom\sigma: dom\ \sigma \supseteq \text{differing-vars}\ mem_1\ mem_2$
shows $mem_1 \ [\mapsto \sigma] = mem_2 \ [\mapsto \sigma]$
proof (rule ext)
fix x
from $dom\sigma$ **show** $mem_1 \ [\mapsto \sigma] x = mem_2 \ [\mapsto \sigma] x$
unfolding *subst-def differing-vars-def*
by (cases $\sigma\ x$, auto)
qed

lemma *mm-equiv-low-eq*:
 $\llbracket \langle c_1, mds, mem_1 \rangle \approx \langle c_2, mds, mem_2 \rangle \rrbracket \implies mem_1 =_{mds^l} mem_2$
unfolding *mm-equiv.simps strong-low-bisim-mm-def*
by *fast*

lemma *globally-sound-modes-compatible*:
 $\llbracket \text{globally-sound-mode-use}\ (cms, mem) \rrbracket \implies \text{compatible-modes}\ (\text{map}\ \text{snd}\ cms)$
by (*simp add: globally-sound-mode-use-def reachable-mode-states-def, auto*)

lemma *compatible-different-no-read* :
assumes *sound-modes: sound-mode-use* (cms_1, mem_1)
sound-mode-use (cms_2, mem_2)
assumes *compat: makes-compatible* (cms_1, mem_1) (cms_2, mem_2) *mems*
assumes *modes-eq: map snd* $cms_1 = \text{map}\ \text{snd}\ cms_2$
assumes *ile: i < length* cms_1
assumes $x: x \in \text{differing-vars-lists}\ mem_1\ mem_2\ mems\ i$
shows $\text{doesn't-read}\ (\text{fst}\ (cms_1\ !\ i))\ x \wedge \text{doesn't-read}\ (\text{fst}\ (cms_2\ !\ i))\ x$
proof –
from *compat* **have** *len: length* $cms_1 = \text{length}\ cms_2$
by *simp*

let $?X_i = \text{differing-vars-lists}\ mem_1\ mem_2\ mems\ i$

from *compat ile x* **have** $a: dma\ x = Low$
by (*metis compat-low*)

from *compat ile x* **have** $b: mem_1\ x \neq mem_2\ x$
by (*metis compat-different*)

with a **and** *compat ile x* **obtain** j **where**
 $jprop: j < \text{length}\ cms_1 \wedge x \notin \text{differing-vars-lists}\ mem_1\ mem_2\ mems\ j$
by *fastforce*

let $?X_j = \text{differing-vars-lists}\ mem_1\ mem_2\ mems\ j$
obtain $\sigma :: 'Var \rightarrow 'Val$ **where** $dom\sigma: dom\ \sigma = ?X_j$
proof
let $?\sigma = \lambda x. \text{if}\ (x \in ?X_j)\ \text{then}\ \text{Some}\ \text{some-val}\ \text{else}\ \text{None}$

```

show dom ?σ = ?Xj unfolding dom-def by auto
qed
let ?mdss = map snd cms1 and
    ?mems1j = fst (mems ! j) and
    ?mems2j = snd (mems ! j)

from jprop domσ have subst-eq:
  ?mems1j [↦ σ] x = ?mems1j x ∧ ?mems2j [↦ σ] x = ?mems2j x
by (metis subst-not-in-dom)

from compat jprop domσ
have (cms1 ! j, ?mems1j [↦ σ]) ≈ (cms2 ! j, ?mems2j [↦ σ])
by (auto simp: Let-def)

hence low-eq: ?mems1j [↦ σ] = ?mdss ! j! ?mems2j [↦ σ] using modes-eq
by (metis (no-types) jprop len mm-equiv-low-eq nth-map surjective-pairing)

with jprop and b have x ∈ (?mdss ! j) AsmNoRead
proof –
  { assume x ∉ (?mdss ! j) AsmNoRead
    then have mems-eq: ?mems1j x = ?mems2j x
      using ⟨dma x = Low⟩ low-eq subst-eq
      by (metis (full-types) low-mds-eq-def subst-eq)

    hence mem1 x = mem2 x
      by (metis compat-different-vars jprop)

    hence False by (metis b)
  }
thus ?thesis by metis
qed

hence x ∈ (?mdss ! i) GuarNoRead
using sound-modes jprop
by (metis compatible-modes-def globally-sound-modes-compatible
    length-map sound-mode-use.simps x ile)

thus doesnt-read (fst (cms1 ! i)) x ∧ doesnt-read (fst (cms2 ! i)) x using
sound-modes ile
by (metis len modes-eq sound-modes-no-read)
qed

definition func-le :: ('a → 'b) ⇒ ('a → 'b) ⇒ bool (infixl ≤ 60)
where f ≤ g = (∀ x ∈ dom f. f x = g x)

fun change-respecting ::
  ('Com, 'Var, 'Val) LocalConf ⇒
  ('Com, 'Var, 'Val) LocalConf ⇒
  'Var set ⇒

```

$((\text{'Var} \rightarrow \text{'Val}) \Rightarrow$
 $(\text{'Var} \rightarrow \text{'Val})) \Rightarrow \text{bool}$
where *change-respecting* $(\text{cms}, \text{mem}) (\text{cms}', \text{mem}') X g =$
 $((\text{cms}, \text{mem}) \rightsquigarrow (\text{cms}', \text{mem}') \wedge$
 $(\forall \sigma. \text{dom } \sigma = X \longrightarrow g \sigma \preceq \sigma) \wedge$
 $(\forall \sigma \sigma'. \text{dom } \sigma = X \wedge \text{dom } \sigma' = X \longrightarrow \text{dom } (g \sigma) = \text{dom } (g \sigma')) \wedge$
 $(\forall \sigma. \text{dom } \sigma = X \longrightarrow (\text{cms}, \text{mem} [\mapsto \sigma]) \rightsquigarrow (\text{cms}', \text{mem}' [\mapsto g \sigma])))$

lemma *change-respecting-dom-unique*:

$\llbracket \text{change-respecting } \langle c, \text{mds}, \text{mem} \rangle \langle c', \text{mds}', \text{mem}' \rangle X g \rrbracket \Longrightarrow$
 $\exists d. \forall f. \text{dom } f = X \longrightarrow d = \text{dom } (g f)$
by $(\text{metis } \text{change-respecting.simps})$

lemma *func-le-restrict*: $\llbracket f \preceq g; X \subseteq \text{dom } f \rrbracket \Longrightarrow f \upharpoonright X \preceq g$

by $(\text{auto simp: func-le-def})$

definition *to-partial* :: $(\text{'a} \Rightarrow \text{'b}) \Rightarrow (\text{'a} \rightarrow \text{'b})$

where *to-partial* $f = (\lambda x. \text{Some } (f x))$

lemma *func-le-dom*: $f \preceq g \Longrightarrow \text{dom } f \subseteq \text{dom } g$

by $(\text{auto simp add: func-le-def,metis domIff option.simps}(2))$

lemma *doesnt-read-mutually-exclusive*:

assumes *noread*: *doesnt-read* $c x$

assumes *eval*: $\langle c, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \rangle$

assumes *unchanged*: $\forall v. \langle c, \text{mds}, \text{mem } (x := v) \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' (x := v) \rangle$

shows $\neg (\forall v. \langle c, \text{mds}, \text{mem } (x := v) \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \rangle)$

using *assms*

apply $(\text{case-tac mem}' x = \text{some-val})$

apply $(\text{metis } (\text{full-types}) \text{prod.inject deterministic different-values fun-upd-same})$

by $(\text{metis } (\text{full-types}) \text{prod.inject deterministic fun-upd-same})$

lemma *doesnt-read-mutually-exclusive'*:

assumes *noread*: *doesnt-read* $c x$

assumes *eval*: $\langle c, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \rangle$

assumes *overwrite*: $\forall v. \langle c, \text{mds}, \text{mem } (x := v) \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \rangle$

shows $\neg (\forall v. \langle c, \text{mds}, \text{mem } (x := v) \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' (x := v) \rangle)$

by $(\text{metis } \text{assms } \text{doesnt-read-mutually-exclusive})$

lemma *change-respecting-dom*:

assumes *cr*: *change-respecting* $(\text{cms}, \text{mem}) (\text{cms}', \text{mem}') X g$

assumes *dom σ* : $\text{dom } \sigma = X$

shows $\text{dom } (g \sigma) \subseteq X$

by $(\text{metis } \text{assms } \text{change-respecting.simps } \text{func-le-dom})$

lemma *change-respecting-intro [iff]*:

$\llbracket \langle c, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \rangle;$

$\wedge f. \text{dom } f = X \Longrightarrow$

$g f \preceq f \wedge$

$(\forall f'. \text{dom } f' = X \longrightarrow \text{dom } (g f) = \text{dom } (g f')) \wedge$
 $(\langle c, \text{mds}, \text{mem} \ [\mapsto f] \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \ [\mapsto g f] \rangle)$]]
 $\implies \text{change-respecting } \langle c, \text{mds}, \text{mem} \rangle \langle c', \text{mds}', \text{mem}' \rangle X g$
unfolding *change-respecting.simps*
by *blast*

lemma *conjI3*: [A ; B ; C] $\implies A \wedge B \wedge C$
by *simp*

lemma *noread-exists-change-respecting*:

assumes *fin*: *finite* ($X :: 'Var \text{ set}$)

assumes *eval*: $\langle c, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \rangle$

assumes *noread*: $\forall x \in X. \text{doesnt-read } c \ x$

shows $\exists (g :: ('Var \rightarrow 'Val) \Rightarrow ('Var \rightarrow 'Val)). \text{change-respecting } \langle c, \text{mds}, \text{mem} \rangle \langle c', \text{mds}', \text{mem}' \rangle X g$

proof –

let $?lc = \langle c, \text{mds}, \text{mem} \rangle$ **and** $?lc' = \langle c', \text{mds}', \text{mem}' \rangle$

from *fin eval noread* **show** $\exists g. \text{change-respecting } \langle c, \text{mds}, \text{mem} \rangle \langle c', \text{mds}', \text{mem}' \rangle X g$

proof (*induct X arbitrary: mem mem' rule: finite-induct*)

case *empty*

let $?g = \lambda \sigma. \text{Map.empty}$

have $\text{mem} \ [\mapsto \text{Map.empty}] = \text{mem } \text{mem}' \ [\mapsto ?g \ \text{Map.empty}] = \text{mem}'$

unfolding *subst-def*

by *auto*

hence *change-respecting* $\langle c, \text{mds}, \text{mem} \rangle \langle c', \text{mds}', \text{mem}' \rangle \{\} ?g$

using *empty*

unfolding *change-respecting.simps func-le-def subst-def*

by *auto*

thus $?case$ **by** *auto*

next

case (*insert x X*)

then obtain g_X **where** *IH*: *change-respecting* $\langle c, \text{mds}, \text{mem} \rangle \langle c', \text{mds}', \text{mem}' \rangle X g_X$

by (*metis insert-iff*)

define g **where** $g \ \sigma =$

(*let* $\sigma' = \sigma \ \upharpoonright \ X$ *in*

(*if* ($\forall v. \langle c, \text{mds}, \text{mem} \ [\mapsto \sigma'] (x := v) \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \ [\mapsto g_X \ \sigma'] (x := v) \rangle$))

then ($\lambda y :: 'Var.$

if $x = y$

then $\sigma \ y$

else $g_X \ \sigma' \ y$)

else ($\lambda y. g_X \ \sigma' \ y$))

for $\sigma :: 'Var \rightarrow 'Val$

have *change-respecting* $\langle c, \text{mds}, \text{mem} \rangle \langle c', \text{mds}', \text{mem}' \rangle (\text{insert } x \ X) \ g$

proof

```

show  $\langle c, mds, mem \rangle \rightsquigarrow \langle c', mds', mem' \rangle$  using insert by auto
next — We first show that property (2) is satisfied.
fix  $\sigma :: 'Var \rightarrow 'Val$ 
let  $?\sigma_X = \sigma \upharpoonright X$ 
assume  $dom \sigma = insert\ x\ X$ 
hence  $dom\ ?\sigma_X = X$ 
  by (metis dom-restrict inf-absorb2 subset-insertI)
from insert have doesn't-read c x by auto
moreover
from IH have  $eval_X: \langle c, mds, mem \ [\mapsto\ ?\sigma_X] \rangle \rightsquigarrow \langle c', mds', mem' \ [\mapsto\ g_X\ ?\sigma_X] \rangle$ 
using  $\langle dom\ ?\sigma_X = X \rangle$ 
unfolding change-respecting.simps
by auto
ultimately have
  noreadx:
   $(\forall v. \langle c, mds, mem \ [\mapsto\ ?\sigma_X] (x := v) \rangle \rightsquigarrow \langle c', mds', mem' \ [\mapsto\ g_X\ ?\sigma_X] (x := v) \rangle) \vee$ 
   $(\forall v. \langle c, mds, mem \ [\mapsto\ ?\sigma_X] (x := v) \rangle \rightsquigarrow \langle c', mds', mem' \ [\mapsto\ g_X\ ?\sigma_X] \rangle)$ 
unfolding doesn't-read-def by auto
show  $g\ \sigma \preceq \sigma \wedge$ 
   $(\forall \sigma'. dom\ \sigma' = insert\ x\ X \longrightarrow dom\ (g\ \sigma) = dom\ (g\ \sigma') \wedge$ 
   $\langle c, mds, mem \ [\mapsto\ \sigma] \rangle \rightsquigarrow \langle c', mds', mem' \ [\mapsto\ g\ \sigma] \rangle)$ 
proof (rule conjI3)
from noreadx show  $g\ \sigma \preceq \sigma$ 
proof
assume nowrite:  $\forall v. \langle c, mds, mem \ [\mapsto\ ?\sigma_X] (x := v) \rangle \rightsquigarrow$ 
   $\langle c', mds', mem' \ [\mapsto\ g_X\ ?\sigma_X] (x := v) \rangle$ 
then have g-simp [simp]:  $g\ \sigma = (\lambda y. if\ y = x\ then\ \sigma\ y\ else\ g_X\ ?\sigma_X\ y)$ 
unfolding g-def
by auto
thus  $g\ \sigma \preceq \sigma$ 
using IH
unfolding g-simp func-le-def
by (auto, metis  $\langle dom\ (\sigma \upharpoonright X) = X \rangle$  domI func-le-def restrict-in)
next
assume overwrites:  $\forall v. \langle c, mds, mem \ [\mapsto\ ?\sigma_X] (x := v) \rangle \rightsquigarrow$ 
   $\langle c', mds', mem' \ [\mapsto\ g_X\ ?\sigma_X] \rangle$ 
hence
   $\neg (\forall v. \langle c, mds, mem \ [\mapsto\ ?\sigma_X] (x := v) \rangle \rightsquigarrow \langle c', mds', mem' \ [\mapsto\ g_X\ ?\sigma_X] (x := v) \rangle)$ 
by (metis  $\langle doesn't-read\ c\ x \rangle$  doesn't-read-mutually-exclusive evalX)
hence g-simp [simp]:  $g\ \sigma = g_X\ ?\sigma_X$ 
unfolding g-def
by (auto simp: Let-def)

also from IH have  $g_X\ ?\sigma_X \preceq ?\sigma_X$ 
by (metis  $\langle dom\ (\sigma \upharpoonright X) = X \rangle$  change-respecting.simps)

```


ultimately show $g \sigma \preceq \sigma$
unfolding *func-le-def*
by (*auto*, *metis* $\langle \text{dom } (\sigma \upharpoonright' X) = X \rangle$ *domI restrict-in*)
qed
next — This part proves that the domain of the family is unique
{
fix $\sigma' :: 'Var \rightarrow 'Val$
assume $\text{dom } \sigma' = \text{insert } x \ X$
let $?\sigma'_X = \sigma' \upharpoonright' X$
have $\text{dom } ?\sigma'_X = X$
by (*metis* $\langle \text{dom } (\sigma \upharpoonright' X) = X \rangle$ $\langle \text{dom } \sigma = \text{insert } x \ X \rangle$ $\langle \text{dom } \sigma' = \text{insert } x \ X \rangle$ *dom-restrict*)
— We first show, that we are always in the same case of the no read assumption:
have same-case:
 $((\forall v. \langle c, \text{mds}, \text{mem} \ [\mapsto ?\sigma_X] (x := v) \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \ [\mapsto g_X ?\sigma_X] (x := v) \rangle) \wedge$
 $(\forall v. \langle c, \text{mds}, \text{mem} \ [\mapsto ?\sigma'_X] (x := v) \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \ [\mapsto g_X ?\sigma'_X] (x := v) \rangle))$
 \vee
 $((\forall v. \langle c, \text{mds}, \text{mem} \ [\mapsto ?\sigma_X] (x := v) \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \ [\mapsto g_X ?\sigma_X] \rangle) \wedge$
 $(\forall v. \langle c, \text{mds}, \text{mem} \ [\mapsto ?\sigma'_X] (x := v) \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \ [\mapsto g_X ?\sigma'_X] \rangle))$
(is $(?N \wedge ?N') \vee (?O \wedge ?O')$ **)**
proof —
— By deriving a contradiction under the assumption that we are in different cases:
have not-different:
 $\bigwedge h \ h'. \llbracket \text{dom } h = \text{insert } x \ X; \text{dom } h' = \text{insert } x \ X;$
 $\forall v. \langle c, \text{mds}, \text{mem} \ [\mapsto h \upharpoonright' X] (x := v) \rangle \rightsquigarrow$
 $\langle c', \text{mds}', \text{mem}' \ [\mapsto g_X (h \upharpoonright' X)] (x := v) \rangle;$
 $\forall v. \langle c, \text{mds}, \text{mem} \ [\mapsto h' \upharpoonright' X] (x := v) \rangle \rightsquigarrow$
 $\langle c', \text{mds}', \text{mem}' \ [\mapsto g_X (h' \upharpoonright' X)] \rangle \rrbracket$
 $\implies \text{False}$
proof —
— Introduce new names to avoid clashes with functions in the outer scope.
fix $h \ h' :: 'Var \rightarrow 'Val$
assume *doms*: $\text{dom } h = \text{insert } x \ X$ $\text{dom } h' = \text{insert } x \ X$
assume *nowrite*: $\forall v. \langle c, \text{mds}, \text{mem} \ [\mapsto h \upharpoonright' X] (x := v) \rangle \rightsquigarrow$
 $\langle c', \text{mds}', \text{mem}' \ [\mapsto g_X (h \upharpoonright' X)] (x := v) \rangle$
assume *overwrite*: $\forall v. \langle c, \text{mds}, \text{mem} \ [\mapsto h' \upharpoonright' X] (x := v) \rangle \rightsquigarrow$
 $\langle c', \text{mds}', \text{mem}' \ [\mapsto g_X (h' \upharpoonright' X)] \rangle$

let $?h_X = h \upharpoonright' X$
let $?h'_X = h' \upharpoonright' X$

have $\text{dom } ?h_X = X$
by (*metis* $\langle \text{dom } (\sigma \upharpoonright' X) = X \rangle$ $\langle \text{dom } \sigma = \text{insert } x \ X \rangle$ *dom-restrict*)

$doms(1))$
have $dom \ ?h'_X = X$
by ($metis \ \langle dom (\sigma \mid' X) = X \rangle \ \langle dom \ \sigma = insert \ x \ X \rangle \ dom\text{-restrict}$)
 $doms(2))$
with IH **have** $eval_X': \langle c, mds, mem \ [\mapsto \ ?h'_X] \rangle \rightsquigarrow \langle c', mds', mem' \ [\mapsto \ g_X \ ?h'_X] \rangle$
unfolding $change\text{-respecting.simps}$
by $auto$
with $\langle doesn't\text{-read} \ c \ x \rangle$ **have** $noread_x'$:
 $(\forall v. \langle c, mds, mem \ [\mapsto \ ?h'_X] \ (x := v) \rangle \rightsquigarrow \langle c', mds', mem' \ [\mapsto \ g_X \ ?h'_X] \ (x := v) \rangle) \vee$
 $(\forall v. \langle c, mds, mem \ [\mapsto \ ?h'_X] \ (x := v) \rangle \rightsquigarrow \langle c', mds', mem' \ [\mapsto \ g_X \ ?h'_X] \ (x := v) \rangle)$
unfolding $doesn't\text{-read}\text{-def}$
by $auto$
from $overwrite$ **obtain** v **where**
 $\neg (\langle c, mds, mem \ [\mapsto \ h' \mid' X] \ (x := v) \rangle \rightsquigarrow \langle c', mds', mem' \ [\mapsto \ g_X \ (h' \mid' X)] \ (x := v) \rangle)$
by ($metis \ \langle doesn't\text{-read} \ c \ x \rangle \ doesn't\text{-read}\text{-mutually}\text{-exclusive}\text{-fun}\text{-upd}\text{-triv}$)
moreover
have $x \notin dom \ (?h'_X)$
by ($metis \ \langle dom \ (h' \mid' X) = X \rangle \ insert(2)$)
with IH **have** $x \notin dom \ (g_X \ ?h'_X)$
by ($metis \ \langle dom \ (h' \mid' X) = X \rangle \ change\text{-respecting.simps} \ func\text{-le}\text{-dom} \ rev\text{-subset}D$)
ultimately **have** $mem' \ x \neq v$
by ($metis \ fun\text{-upd}\text{-triv} \ overwrite \ subst\text{-not}\text{-in}\text{-dom}$)
let $?mem_v = mem \ (x := v)$
obtain mem_v' **where** $\langle c, mds, ?mem_v \rangle \rightsquigarrow \langle c', mds', mem_v' \rangle$
using $insert \ \langle doesn't\text{-read} \ c \ x \rangle$
unfolding $doesn't\text{-read}\text{-def}$
by ($auto, metis$)
also **have** $\forall x \in X. \ doesn't\text{-read} \ c \ x$
by ($metis \ insert(5) \ insert\text{-iff}$)
ultimately **obtain** g_v **where**
 $IH_v: \ change\text{-respecting} \ \langle c, mds, ?mem_v \rangle \ \langle c', mds', mem_v' \rangle \ X \ g_v$
by ($metis \ insert(3)$)
hence $eval_v: \langle c, mds, ?mem_v \ [\mapsto \ ?h_X] \rangle \rightsquigarrow \langle c', mds', mem_v' \ [\mapsto \ g_v \ ?h_X] \rangle$
 $\langle c, mds, ?mem_v \ [\mapsto \ ?h'_X] \rangle \rightsquigarrow \langle c', mds', mem_v' \ [\mapsto \ g_v \ ?h'_X] \rangle$
apply ($metis \ \langle dom \ (h \mid' X) = X \rangle \ change\text{-respecting.simps}$)
by ($metis \ IH_v \ \langle dom \ (h' \mid' X) = X \rangle \ change\text{-respecting.simps}$)

from $eval_v(1)$ **have** $mem_v' x = v$
proof –
 $?h_X]$ **assume** $\langle c, mds, mem (x := v) [\mapsto ?h_X] \rangle \rightsquigarrow \langle c', mds', mem_v' [\mapsto g_v$
have $?mem_v [\mapsto ?h_X] = mem [\mapsto ?h_X] (x := v)$
apply (*rule ext, rename-tac y*)
apply (*case-tac y = x*)
apply (*auto simp: subst-def*)
apply (*metis (full-types) <dom (h | ' X) = X> fun-upd-def*
insert(2) subst-def subst-not-in-dom)
by (*metis fun-upd-other*)

with nowrite have $mem_v' [\mapsto g_v ?h_X] = mem' [\mapsto g_X ?h_X] (x := v)$
using *deterministic*
by (*erule-tac x = v in allE, auto, metis eval_v(1)*)

hence $mem_v' [\mapsto g_v ?h_X] x = v$
by *simp*
also have $x \notin dom (g_v ?h_X)$
using $IH_v \langle dom ?h_X = X \rangle$ *change-respecting-dom*
by (*metis func-le-dom insert(2) rev-subsetD*)
ultimately show $mem_v' x = v$
by (*metis subst-not-in-dom*)
qed
moreover
from $eval_v(2)$ **have** $mem_v' x = mem' x$
proof –
assume $\langle c, mds, ?mem_v [\mapsto ?h'_X] \rangle \rightsquigarrow \langle c', mds', mem_v' [\mapsto g_v ?h'_X] \rangle$
moreover
from *overwrite have*
 $\langle c, mds, mem [\mapsto ?h'_X] (x := v) \rangle \rightsquigarrow \langle c', mds', mem' [\mapsto g_X ?h'_X] \rangle$
by *auto*
moreover
have $?mem_v [\mapsto ?h'_X] = mem [\mapsto ?h'_X] (x := v)$
apply (*rule ext, rename-tac y*)
apply (*case-tac y = x*)
apply (*metis <x \notin dom (h' | ' X)> fun-upd-apply subst-not-in-dom*)
apply (*auto simp: subst-def*)
by (*metis fun-upd-other*)
ultimately have $mem' [\mapsto g_X ?h'_X] = mem_v' [\mapsto g_v ?h'_X]$
using *deterministic*
by *auto*
also have $x \notin dom (g_v ?h'_X)$
using $IH_v \langle dom ?h'_X = X \rangle$ *change-respecting-dom*
by (*metis func-le-dom insert(2) subsetD*)
ultimately show $mem_v' x = mem' x$
using $\langle x \notin dom (g_X ?h'_X) \rangle$
by (*metis subst-not-in-dom*)

qed
ultimately show *False*
using $\langle \text{mem}' x \neq v \rangle$
by *auto*
qed

moreover
have $\text{dom } ?\sigma'_X = X$
by (*metis* $\langle \text{dom } (\sigma \upharpoonright' X) = X \rangle \langle \text{dom } \sigma = \text{insert } x X \rangle \langle \text{dom } \sigma' = \text{insert } x X \rangle \text{dom-restrict}$)

with *IH* **have** $\text{eval}_X': \langle c, \text{mds}, \text{mem } [\mapsto ?\sigma'_X] \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' [\mapsto g_X ?\sigma'_X] \rangle$
unfolding *change-respecting.simps*
by *auto*
with $\langle \text{doesnt-read } c x \rangle$ **have** noread_x' :
 $(\forall v. \langle c, \text{mds}, \text{mem } [\mapsto ?\sigma'_X] (x := v) \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' [\mapsto g_X ?\sigma'_X] (x := v) \rangle)$
 \vee
 $(\forall v. \langle c, \text{mds}, \text{mem } [\mapsto ?\sigma'_X] (x := v) \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' [\mapsto g_X ?\sigma'_X] \rangle)$
unfolding *doesnt-read-def*
by *auto*

ultimately show *?thesis*
using noread_x *not-different* $\langle \text{dom } \sigma = \text{insert } x X \rangle \langle \text{dom } \sigma' = \text{insert } x X \rangle$
by *auto*
qed
hence $\text{dom } (g \sigma) = \text{dom } (g \sigma')$
proof
assume
 $(\forall v. \langle c, \text{mds}, \text{mem } [\mapsto ?\sigma_X] (x := v) \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' [\mapsto g_X ?\sigma_X] (x := v) \rangle) \wedge$
 $(\forall v. \langle c, \text{mds}, \text{mem } [\mapsto ?\sigma'_X] (x := v) \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' [\mapsto g_X ?\sigma'_X] (x := v) \rangle)$
hence *g-simp* [*simp*]: $g \sigma = (\lambda y. \text{if } y = x \text{ then } \sigma y \text{ else } g_X ?\sigma_X y) \wedge$
 $g \sigma' = (\lambda y. \text{if } y = x \text{ then } \sigma' y \text{ else } g_X ?\sigma'_X y)$
unfolding *g-def*
by *auto*
thus *?thesis*
using *IH* $\langle \text{dom } \sigma = \text{insert } x X \rangle \langle \text{dom } \sigma' = \text{insert } x X \rangle$
unfolding *change-respecting.simps*
apply (*auto simp: domD*)
apply (*metis* $\langle \text{dom } (\sigma \upharpoonright' X) = X \rangle \langle \text{dom } (\sigma' \upharpoonright' X) = X \rangle \text{domD domI}$)
by (*metis* $\langle \text{dom } (\sigma \upharpoonright' X) = X \rangle \langle \text{dom } (\sigma' \upharpoonright' X) = X \rangle \text{domD domI}$)
next
assume
 $(\forall v. \langle c, \text{mds}, \text{mem } [\mapsto ?\sigma_X] (x := v) \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' [\mapsto g_X ?\sigma_X] \rangle)$
 \wedge

$(\forall v. \langle c, mds, mem \ [\vdash \ ?\sigma'_X] \ (x := v) \rangle \rightsquigarrow \langle c', mds', mem' \ [\vdash \ g_X \ ?\sigma'_X] \rangle)$
hence
 $\neg (\forall v. \langle c, mds, mem \ [\vdash \ ?\sigma_X] \ (x := v) \rangle \rightsquigarrow \langle c', mds', mem' \ [\vdash \ g_X \ ?\sigma_X] \ (x := v) \rangle)$
 \wedge
 $\neg (\forall v. \langle c, mds, mem \ [\vdash \ ?\sigma'_X] \ (x := v) \rangle \rightsquigarrow \langle c', mds', mem' \ [\vdash \ g_X \ ?\sigma'_X] \ (x := v) \rangle)$
by (*metis* $\langle \text{doesnt-read } c \ x \rangle \text{ doesnt-read-mutually-exclusive' fun-upd-triv}$)

hence *g-simp* [*simp*]: $g \ \sigma = g_X \ ?\sigma_X \wedge g \ \sigma' = g_X \ ?\sigma'_X$
unfolding *g-def*
by (*auto simp: Let-def*)
with IH show *?thesis*
unfolding *change-respecting.simps*
by (*metis* $\langle \text{dom } (\sigma \ |' \ X) = X \rangle \langle \text{dom } (\sigma' \ |' \ X) = X \rangle$)
qed

thus $\forall \sigma'. \text{dom } \sigma' = \text{insert } x \ X \longrightarrow \text{dom } (g \ \sigma) = \text{dom } (g \ \sigma')$ **by** *blast*
next

from *noread_x* **show** $\langle c, mds, mem \ [\vdash \ \sigma] \rangle \rightsquigarrow \langle c', mds', mem' \ [\vdash \ g \ \sigma] \rangle$
proof
assume *nowrite*:
 $\forall v. \langle c, mds, mem \ [\vdash \ ?\sigma_X] \ (x := v) \rangle \rightsquigarrow \langle c', mds', mem' \ [\vdash \ g_X \ ?\sigma_X] \ (x := v) \rangle$
then have *g-simp* [*simp*]: $g \ \sigma = (\lambda y. \text{if } y = x \text{ then } \sigma \ y \text{ else } g_X \ ?\sigma_X \ y)$
unfolding *g-def*
by *auto*
obtain *v* **where** $\sigma \ x = \text{Some } v$
by (*metis* $\langle \text{dom } \sigma = \text{insert } x \ X \rangle \text{ domD insertI1}$)

from *nowrite* **have**
 $\langle c, mds, mem \ [\vdash \ ?\sigma_X] \ (x := v) \rangle \rightsquigarrow \langle c', mds', mem' \ [\vdash \ g_X \ ?\sigma_X] \ (x := v) \rangle$
by *auto*
moreover
have $mem \ [\vdash \ ?\sigma_X] \ (x := v) = mem \ [\vdash \ \sigma]$
apply (*rule ext, rename-tac y*)
apply (*case-tac y = x*)
apply (*auto simp: subst-def*)
apply (*metis* $\langle \sigma \ x = \text{Some } v \rangle \text{ option.simps(5)}$)
by (*metis* $\langle \text{dom } (\sigma \ |' \ X) = X \rangle \langle \text{dom } \sigma = \text{insert } x \ X \rangle \text{ insertE}$
restrict-in subst-def subst-not-in-dom)

moreover
have $mem' \ [\vdash \ g_X \ ?\sigma_X] \ (x := v) = mem' \ [\vdash \ g \ \sigma]$
apply (*rule ext, rename-tac y*)
apply (*case-tac y = x*)
by (*auto simp: subst-def option.simps* $\langle \sigma \ x = \text{Some } v \rangle$)
ultimately show *?thesis*
by *auto*

next
assume *overwrites*:
 $\forall v. \langle c, mds, mem \ [\mapsto \ ?\sigma_X] \ (x := v) \rangle \rightsquigarrow \langle c', mds', mem' \ [\mapsto \ g_X \ ?\sigma_X] \rangle$
hence
 $\neg (\forall v. \langle c, mds, mem \ [\mapsto \ ?\sigma_X] \ (x := v) \rangle \rightsquigarrow \langle c', mds', mem' \ [\mapsto \ g_X \ ?\sigma_X] \ (x := v) \rangle)$
by (*metis* $\langle \text{doesnt-read } c \ x \rangle \text{ doesnt-read-mutually-exclusive' eval}_X$)
hence *g-simp* [*simp*]: $g \ \sigma = g_X \ ?\sigma_X$
unfolding *g-def*
by (*auto simp: Let-def*)
obtain *v* **where** $\sigma \ x = \text{Some } v$
by (*metis* $\langle \text{dom } \sigma = \text{insert } x \ X \rangle \text{ domD insertI1}$)
have $mem \ [\mapsto \ ?\sigma_X] \ (x := v) = mem \ [\mapsto \ \sigma]$
apply (*rule ext, rename-tac y*)
apply (*case-tac y = x*)
apply (*auto simp: subst-def*)
apply (*metis* $\langle \sigma \ x = \text{Some } v \rangle \text{ option.simps(5)}$)
by (*metis* $\langle \text{dom } (\sigma \ |' \ X) = X \rangle \langle \text{dom } \sigma = \text{insert } x \ X \rangle \text{ insertE}$
restrict-in subst-def subst-not-in-dom)
moreover
from *overwrites* **have** $\langle c, mds, mem \ [\mapsto \ ?\sigma_X] \ (x := v) \rangle \rightsquigarrow \langle c', mds', mem' \ [\mapsto \ g \ \sigma] \rangle$
by (*metis g-simp*)
ultimately show $\langle c, mds, mem \ [\mapsto \ \sigma] \rangle \rightsquigarrow \langle c', mds', mem' \ [\mapsto \ g \ \sigma] \rangle$
by *auto*
qed
qed
qed
thus $\exists g. \text{change-respecting } \langle c, mds, mem \rangle \langle c', mds', mem' \rangle \ (\text{insert } x \ X) \ g$
by *metis*
qed
qed

lemma *differing-vars-neg*: $x \notin \text{differing-vars-lists } mem1 \ mem2 \ mems \ i \implies$
 $(fst \ (mems \ ! \ i) \ x = mem1 \ x \wedge \text{snd} \ (mems \ ! \ i) \ x = mem2 \ x)$
by (*simp add: differing-vars-lists-def differing-vars-def*)

lemma *differing-vars-neg-intro*:
 $\llbracket mem_1 \ x = fst \ (mems \ ! \ i) \ x;$
 $mem_2 \ x = \text{snd} \ (mems \ ! \ i) \ x \rrbracket \implies x \notin \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i$
by (*auto simp: differing-vars-lists-def differing-vars-def*)

lemma *differing-vars-elim* [*elim*]:
 $x \in \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i \implies$
 $(fst \ (mems \ ! \ i) \ x \neq mem_1 \ x) \vee (\text{snd} \ (mems \ ! \ i) \ x \neq mem_2 \ x)$
by (*auto simp: differing-vars-lists-def differing-vars-def*)

lemma *subst-overrides*: $\text{dom } \sigma = \text{dom } \tau \implies mem \ [\mapsto \ \tau] \ [\mapsto \ \sigma] = mem \ [\mapsto \ \sigma]$
unfolding *subst-def*

by (metis domIff option.exhaust option.simps(4) option.simps(5))

lemma *dom-restrict-total*: $\text{dom } (\text{to-partial } f \mid' X) = X$
unfolding *to-partial-def*
by (metis Int-UNIV-left dom-const dom-restrict)

lemma *update-nth-eq*:
 $\llbracket xs = ys; n < \text{length } xs \rrbracket \implies xs = ys [n := xs ! n]$
by (metis list-update-id)

This property is obvious, so an unreadable apply-style proof is acceptable here:

lemma *mm-equiv-step*:
assumes *bisim*: $(\text{cms}_1, \text{mem}_1) \approx (\text{cms}_2, \text{mem}_2)$
assumes *modes-eq*: $\text{snd } \text{cms}_1 = \text{snd } \text{cms}_2$
assumes *step*: $(\text{cms}_1, \text{mem}_1) \rightsquigarrow (\text{cms}_1', \text{mem}_1')$
shows $\exists c_2' \text{ mem}_2'. (\text{cms}_2, \text{mem}_2) \rightsquigarrow \langle c_2', \text{snd } \text{cms}_1', \text{mem}_2' \rangle \wedge$
 $(\text{cms}_1', \text{mem}_1') \approx \langle c_2', \text{snd } \text{cms}_1', \text{mem}_2' \rangle$
using *assms mm-equiv-strong-low-bisim*
unfolding *strong-low-bisim-mm-def*
apply *auto*
apply (erule-tac $x = \text{fst } \text{cms}_1$ **in** *allE*)
apply (erule-tac $x = \text{snd } \text{cms}_1$ **in** *allE*)
by (metis surjective-pairing)

lemma *change-respecting-doesnt-modify*:
assumes *cr*: *change-respecting* $(\text{cms}, \text{mem}) (\text{cms}', \text{mem}') X g$
assumes *eval*: $(\text{cms}, \text{mem}) \rightsquigarrow (\text{cms}', \text{mem}')$
assumes *domf*: $\text{dom } f = X$
assumes *x-in-dom*: $x \in \text{dom } (g f)$
assumes *noread*: *doesnt-read* $(\text{fst } \text{cms}) x$
shows $\text{mem } x = \text{mem}' x$

proof –
let $?f' = \text{to-partial } \text{mem} \mid' X$
have $\text{dom } ?f' = X$
by (metis dom-restrict-total)

from *cr* **and** *eval* **have** $\forall f. \text{dom } f = X \longrightarrow (\text{cms}, \text{mem} [\mapsto f]) \rightsquigarrow (\text{cms}', \text{mem}' [\mapsto g f])$
 $[\mapsto g f])$
unfolding *change-respecting.simps*
by *metis*
hence $\text{eval}' : (\text{cms}, \text{mem} [\mapsto ?f']) \rightsquigarrow (\text{cms}', \text{mem}' [\mapsto g ?f'])$
by (metis domf')

have *mem-eq*: $\text{mem} [\mapsto ?f'] = \text{mem}$
proof
fix x
show $\text{mem} [\mapsto ?f'] x = \text{mem } x$
unfolding *subst-def*

```

    apply (cases x ∈ X)
    apply (metis option.simps(5) restrict-in to-partial-def)
    by (metis domf' subst-def subst-not-in-dom)
qed

then have mem'-eq: mem' [↦ g ?f'] = mem'
  using eval eval' deterministic
  by (metis Pair-inject)

moreover
have dom (g ?f') = dom (g f)
  by (metis change-respecting.simps cr domf domf')
hence x-in-dom': x ∈ dom (g ?f')
  by (metis x-in-dom)
have x ∈ X
  by (metis change-respecting.simps cr domf func-le-dom in-mono x-in-dom)
hence ?f' x = Some (mem x)
  by (metis restrict-in to-partial-def)
hence g ?f' x = Some (mem x)
  using cr func-le-def
  by (metis change-respecting.simps domf' x-in-dom')

hence mem' [↦ g ?f'] x = mem x
  using subst-def x-in-dom'
  by (metis option.simps(5))
thus mem x = mem' x
  by (metis mem'-eq)
qed

```

type-synonym ('var, 'val) adaptation = 'var → ('val × 'val)

definition apply-adaptation ::
 bool ⇒ ('Var, 'Val) Mem ⇒ ('Var, 'Val) adaptation ⇒ ('Var, 'Val) Mem
 where apply-adaptation first mem A =
 (λ x. case (A x) of
 Some (v₁, v₂) ⇒ if first then v₁ else v₂
 | None ⇒ mem x)

abbreviation apply-adaptation₁ ::
 ('Var, 'Val) Mem ⇒ ('Var, 'Val) adaptation ⇒ ('Var, 'Val) Mem
 (- [[₁ -] [900, 0] 1000)
 where mem [[₁ A] ≡ apply-adaptation True mem A

abbreviation apply-adaptation₂ ::
 ('Var, 'Val) Mem ⇒ ('Var, 'Val) adaptation ⇒ ('Var, 'Val) Mem
 (- [[₂ -] [900, 0] 1000)
 where mem [[₂ A] ≡ apply-adaptation False mem A

definition *restrict-total* :: ('a ⇒ 'b) ⇒ 'a set ⇒ 'a → 'b
where *restrict-total* f A = *to-partial* f |' A

lemma *differing-empty-eq*:
[[*differing-vars* mem mem' = {}]] ⇒ mem = mem'
unfolding *differing-vars-def*
by *auto*

definition *globally-consistent-var* :: ('Var, 'Val) adaptation ⇒ 'Var Mds ⇒ 'Var
⇒ bool
where *globally-consistent-var* A mds x ≡
(case A x of
Some (v, v') ⇒ x ∉ mds AsmNoWrite ∧ (dma x = Low → v = v')
| None ⇒ True)

definition *globally-consistent* :: ('Var, 'Val) adaptation ⇒ 'Var Mds ⇒ bool
where *globally-consistent* A mds ≡ finite (dom A) ∧
(∀ x ∈ dom A. *globally-consistent-var* A mds x)

definition *gc2* :: ('Var, 'Val) adaptation ⇒ 'Var Mds ⇒ bool
where *gc2* A mds = (∀ x ∈ dom A. *globally-consistent-var* A mds x)

lemma *globally-consistent-dom*:
[[*globally-consistent* A mds; X ⊆ dom A]] ⇒ *globally-consistent* (A |' X) mds
unfolding *globally-consistent-def* *globally-consistent-var-def*
by (*metis* (*no-types*) *IntE* *dom-restrict* *inf-absorb2* *restrict-in* *rev-finite-subset*)

lemma *globally-consistent-writable*:
[[x ∈ dom A; *globally-consistent* A mds]] ⇒ x ∉ mds AsmNoWrite
unfolding *globally-consistent-def* *globally-consistent-var-def*
by (*metis* (*no-types*) *domD* *option.simps(5)* *split-part*)

lemma *globally-consistent-loweq*:
assumes *globally-consistent*: *globally-consistent* A mds
assumes *some*: A x = Some (v, v')
assumes *low*: dma x = Low
shows v = v'

proof –
from *some* **have** x ∈ dom A
by (*metis* *domI*)
hence case A x of None ⇒ True | Some (v, v') ⇒ (dma x = Low → v = v')
using *globally-consistent*
unfolding *globally-consistent-def* *globally-consistent-var-def*
by (*metis* *option.simps(5)* *some* *split-part*)
with ⟨dma x = Low⟩ **show** ?thesis
unfolding *some*
by *auto*

qed

lemma *globally-consistent-adapt-bisim*:
assumes *bisim*: $\langle c_1, mds, mem_1 \rangle \approx \langle c_2, mds, mem_2 \rangle$
assumes *globally-consistent*: *globally-consistent A mds*
shows $\langle c_1, mds, mem_1 [\llbracket_1 A \rrbracket] \rangle \approx \langle c_2, mds, mem_2 [\llbracket_2 A \rrbracket] \rangle$
proof –
from *globally-consistent* **have** *finite (dom A)*
by (*auto simp: globally-consistent-def*)
thus *?thesis*
using *globally-consistent*
proof (*induct dom A arbitrary: A rule: finite-induct*)
case *empty*
hence $\bigwedge x. A\ x = None$
by *auto*
hence $mem_1 [\llbracket_1 A \rrbracket] = mem_1$ **and** $mem_2 [\llbracket_2 A \rrbracket] = mem_2$
unfolding *apply-adaptation-def*
by *auto*
with *bisim* **show** *?case*
by *auto*
next
case (*insert x X*)
define *A'* **where** $A' = A \upharpoonright^c X$
hence $dom\ A' = X$
by (*metis Int-insert-left-if0 dom-restrict inf-absorb2 insert(2) insert(4) order-refl*)
moreover
from *insert* **have** *globally-consistent A' mds*
by (*metis A'-def globally-consistent-dom subset-insertI*)
ultimately **have** *bisim'*: $\langle c_1, mds, mem_1 [\llbracket_1 A' \rrbracket] \rangle \approx \langle c_2, mds, mem_2 [\llbracket_2 A' \rrbracket] \rangle$
using *insert*
by *metis*
with *insert* **have** *writable: x ∉ mds AsmNoWrite*
by (*metis globally-consistent-writable insertI1*)
from *insert* **obtain** *v v'* **where** $A\ x = Some\ (v, v')$
unfolding *globally-consistent-def globally-consistent-var-def*
by (*metis (no-types) domD insert-iff option.simps(5) case-prodE*)

have $A-A': \bigwedge y. y \neq x \implies A\ y = A'\ y$
unfolding *A'-def*
by (*metis domIff insert(4) insert-iff restrict-in restrict-out*)

have *eq1*: $mem_1 [\llbracket_1 A' \rrbracket] (x := v) = mem_1 [\llbracket_1 A \rrbracket]$
unfolding *apply-adaptation-def A'-def*
apply (*rule ext, rename-tac y*)
apply (*case-tac x = y*)
apply *auto*
apply (*metis ⟨A x = Some (v, v')⟩ option.simps(5) split-conv*)
by (*metis A'-def A-A'*)

```

have eq2: mem2 [|2 A'] (x := v') = mem2 [|2 A]
  unfolding apply-adaptation-def A'-def
  apply (rule ext, rename-tac y)
  apply (case-tac x = y)
  apply auto
  apply (metis ⟨A x = Some (v, v')⟩ option.simps(5) split-conv)
  by (metis A'-def A-A')

show ?case
proof (cases dma x)
  assume dma x = High
  hence ⟨c1, mds, mem1 [|1 A'] (x := v)⟩ ≈ ⟨c2, mds, mem2 [|2 A'] (x := v')⟩
    using mm-equiv-glob-consistent
    unfolding closed-glob-consistent-def
    by (metis bisim' ⟨x ∉ mds AsmNoWrite⟩)
  thus ?case using eq1 eq2
    by auto
next
  assume dma x = Low
  hence v = v'
    by (metis ⟨A x = Some (v, v')⟩ globally-consistent-loweq insert.prem)
  moreover
  from writable and bisim have
    ⟨c1, mds, mem1 [|1 A'] (x := v)⟩ ≈ ⟨c2, mds, mem2 [|2 A'] (x := v)⟩
      using mm-equiv-glob-consistent
      unfolding closed-glob-consistent-def
      by (metis ⟨dma x = Low⟩ bisim')
  ultimately show ?case using eq1 eq2
    by auto
qed
qed
qed

```

lemma *makes-compatible-invariant*:

```

assumes sound-modes: sound-mode-use (cms1, mem1)
  sound-mode-use (cms2, mem2)
assumes compat: makes-compatible (cms1, mem1) (cms2, mem2) mems
assumes modes-eq: map snd cms1 = map snd cms2
assumes eval: (cms1, mem1) → (cms1', mem1')
obtains cms2' mem2' mems' where
  map snd cms1' = map snd cms2' ∧
  (cms2, mem2) → (cms2', mem2') ∧
  makes-compatible (cms1', mem1') (cms2', mem2') mems'
proof –
let ?X = λ i. differing-vars-lists mem1 mem2 mems i
from sound-modes compat modes-eq have
  a: ∀ i < length cms1. ∀ x ∈ (?X i). doesnt-read (fst (cms1 ! i)) x ∧
    doesnt-read (fst (cms2 ! i)) x

```

by (*metis compatible-different-no-read*)
from *eval* **obtain** k **where**
 $b: k < \text{length } cms_1 \wedge (cms_1 ! k, mem_1) \rightsquigarrow (cms_1' ! k, mem_1') \wedge$
 $cms_1' = cms_1 [k := cms_1' ! k]$
by (*metis meval-elim nth-list-update-eq*)

from *modes-eq* **have** *equal-size*: $\text{length } cms_1 = \text{length } cms_2$
by (*metis length-map*)

let $?mds_k = \text{snd } (cms_1 ! k)$ **and**
 $?mds_{k'} = \text{snd } (cms_1' ! k)$ **and**
 $?mems_1 k = \text{fst } (mems ! k)$ **and**
 $?mems_2 k = \text{snd } (mems ! k)$ **and**
 $?n = \text{length } cms_1$

have *finite* ($?X k$)
by (*metis differing-lists-finite*)

then obtain $g1$ **where**
 $c: \text{change-respecting } (cms_1 ! k, mem_1) (cms_1' ! k, mem_1') (?X k) g1$
using *noread-exists-change-respecting* $b a$
by (*metis surjective-pairing*)

from *compat* **have** $\bigwedge \sigma. \text{dom } \sigma = ?X k \implies ?mems_1 k [\mapsto \sigma] = mem_1 [\mapsto \sigma]$
by (*metis (no-types) Un-upper1 differing-vars-lists-def differing-vars-subst*)

with b **and** c **have**
 $\text{eval}_\sigma: \bigwedge \sigma. \text{dom } \sigma = ?X k \implies (cms_1 ! k, ?mems_1 k [\mapsto \sigma]) \rightsquigarrow (cms_1' ! k, mem_1' [\mapsto g1 \sigma])$
by *auto*

moreover
with b **and** *compat* **have**
 $\text{bisim}_\sigma: \bigwedge \sigma. \text{dom } \sigma = ?X k \implies (cms_1 ! k, ?mems_1 k [\mapsto \sigma]) \approx (cms_2 ! k, ?mems_2 k [\mapsto \sigma])$
by *auto*

moreover have $\text{snd } (cms_1 ! k) = \text{snd } (cms_2 ! k)$
by (*metis b equal-size modes-eq nth-map*)

ultimately have $d: \bigwedge \sigma. \text{dom } \sigma = ?X k \implies \exists c_f' mem_f'.$
 $(cms_2 ! k, ?mems_2 k [\mapsto \sigma]) \rightsquigarrow \langle c_f', ?mds_{k'}, mem_f' \rangle \wedge$
 $(cms_1' ! k, mem_1' [\mapsto g1 \sigma]) \approx \langle c_f', ?mds_{k'}, mem_f' \rangle$
by (*metis mm-equiv-step*)

obtain $h :: 'Var \rightarrow 'Val$ **where** *domh*: $\text{dom } h = ?X k$
by (*metis dom-restrict-total*)

then obtain $c_h \text{ mem}_h$ **where** $h\text{-prop}$:
 $(cms_2 ! k, ?mems_2k [\mapsto h]) \rightsquigarrow \langle c_h, ?mds_k', mem_h \rangle \wedge$
 $(cms_1' ! k, mem_1' [\mapsto g1 h]) \approx \langle c_h, ?mds_k', mem_h \rangle$
using d
by $metis$

then obtain g^2 **where** e :
 $change\text{-respecting} (cms_2 ! k, ?mems_2k [\mapsto h]) \langle c_h, ?mds_k', mem_h \rangle (?X k) g^2$
using $a b \text{ noread-exists-change-respecting}$
by $(metis \text{differing-lists-finite surjective-pairing})$

— The following statements are universally quantified since they are reused later:

with $h\text{-prop}$ **have**
 $\forall \sigma. \text{dom } \sigma = ?X k \longrightarrow$
 $(cms_2 ! k, ?mems_2k [\mapsto h] [\mapsto \sigma]) \rightsquigarrow \langle c_h, ?mds_k', mem_h [\mapsto g^2 \sigma] \rangle$
unfolding $change\text{-respecting.simps}$
by $auto$

with $domh$ **have** f :
 $\forall \sigma. \text{dom } \sigma = ?X k \longrightarrow$
 $(cms_2 ! k, ?mems_2k [\mapsto \sigma]) \rightsquigarrow \langle c_h, ?mds_k', mem_h [\mapsto g^2 \sigma] \rangle$
by $(auto \text{ simp: subst-overrides})$

from d **and** f **have** g : $\bigwedge \sigma. \text{dom } \sigma = ?X k \implies$
 $(cms_2 ! k, ?mems_2k [\mapsto \sigma]) \rightsquigarrow \langle c_h, ?mds_k', mem_h [\mapsto g^2 \sigma] \rangle \wedge$
 $(cms_1' ! k, mem_1' [\mapsto g1 \sigma]) \approx \langle c_h, ?mds_k', mem_h [\mapsto g^2 \sigma] \rangle$
using $h\text{-prop}$
by $(metis \text{deterministic})$

let $? \sigma\text{-mem}_2 = \text{to-partial } mem_2 \mid' ?X k$
define mem_2' **where** $mem_2' = mem_h [\mapsto g^2 ? \sigma\text{-mem}_2]$
define c_2' **where** $c_2' = c_h$

have $dom\sigma\text{-mem}_2$: $dom ? \sigma\text{-mem}_2 = ?X k$
by $(metis \text{dom-restrict-total})$

have $mem_2 = ?mems_2k [\mapsto ? \sigma\text{-mem}_2]$

proof $(rule \text{ext})$

fix x
show $mem_2 x = ?mems_2k [\mapsto ? \sigma\text{-mem}_2] x$
using $dom\sigma\text{-mem}_2$
unfolding $\text{to-partial-def subst-def}$
apply $(cases \ x \in ?X k)$
apply $auto$
by $(metis \text{differing-vars-neg})$

qed

with $f \text{ dom}\sigma\text{-mem}_2$ **have** i : $(cms_2 ! k, mem_2) \rightsquigarrow \langle c_2', ?mds_k', mem_2' \rangle$
unfolding $mem_2'\text{-def } c_2'\text{-def}$
by $metis$

define cms_2' **where** $cms_2' = cms_2 [k := (c_2', ?m_{ds_k}')]]$

with i b *equal-size* **have** $(cms_2, mem_2) \rightarrow (cms_2', mem_2')$
by (*metis meval-intro*)

moreover

from *equal-size* **have** *new-length*: $length\ cms_1' = length\ cms_2'$
unfolding cms_2' -*def*
by (*metis eval length-list-update meval-elim*)

with *modes-eq* **have** $map\ snd\ cms_1' = map\ snd\ cms_2'$
unfolding cms_2' -*def*
by (*metis b map-update snd-conv*)

moreover

from c **and** e **obtain** *dom-g1 dom-g2* **where**
dom-uniq: $\bigwedge \sigma. dom\ \sigma = ?X\ k \implies dom-g1 = dom\ (g1\ \sigma)$
 $\bigwedge \sigma. dom\ \sigma = ?X\ k \implies dom-g2 = dom\ (g2\ \sigma)$
by (*metis change-respecting.simps domh*)

— This is the complicated part of the proof.

obtain $mems'$ **where** *makes-compatible* $(cms_1', mem_1') (cms_2', mem_2')\ mems'$

proof

define $mems'$ - k
where $mems'$ - $k\ x =$
(if $x \notin ?X\ k$
then $(mem_1'\ x, mem_2'\ x)$
else if $(x \notin dom-g1) \vee (x \notin dom-g2)$
then $(mem_1'\ x, mem_2'\ x)$
else $(?mems_1k\ x, ?mems_2k\ x)$ **for** x

— This is used in two of the following cases, so we prove it beforehand:

have *x-unchanged*: $\bigwedge x. \llbracket x \in ?X\ k; x \in dom-g1; x \in dom-g2 \rrbracket \implies$
 $mem_1\ x = mem_1'\ x \wedge mem_2\ x = mem_2'\ x$

proof

fix x

assume $x \in ?X\ k$ **and** $x \in dom-g1$

thus $mem_1\ x = mem_1'\ x$

using $a\ b\ c$

by (*metis change-respecting-doesnt-modify dom-uniq(1) domh*)

next

fix x

assume $x \in ?X\ k$ **and** $x \in dom-g2$

hence *eq-mem₂*: $?σ-mem_2\ x = Some\ (mem_2\ x)$

by (*metis restrict-in to-partial-def*)

hence $?mems_2k\ [\vdash h]\ [\vdash ?σ-mem_2]\ x = mem_2\ x$

by (*auto simp: subst-def*)

moreover
from $\langle x \in \text{dom-}g2 \rangle \text{ dom-uniq } e$ **have** $g\text{-eq}: g2 \text{ ?}\sigma\text{-mem}_2 x = \text{Some } (\text{mem}_2 x)$
unfolding *change-respecting.simps func-le-def*
by (*metis dom-restrict-total eq-mem₂*)
hence $\text{mem}_h [\mapsto g2 \text{ ?}\sigma\text{-mem}_2] x = \text{mem}_2 x$
by (*auto simp: subst-def*)

ultimately have $\text{?mems}_2 k [\mapsto h] [\mapsto \text{?}\sigma\text{-mem}_2] x = \text{mem}_h [\mapsto g2 \text{ ?}\sigma\text{-mem}_2]$
 x
by *auto*
thus $\text{mem}_2 x = \text{mem}_2' x$
by (*metis $\langle \text{mem}_2 = \text{?mems}_2 k [\mapsto \text{?}\sigma\text{-mem}_2] \rangle \text{ dom}\sigma\text{-mem}_2 \text{ dom}h \text{ mem}_2'\text{-def}$*
subst-overrides)
qed

define *mems'-i*
where *mems'-i* $i x =$
(if ((mem₁ x ≠ mem₁' x ∨ mem₂ x ≠ mem₂' x) ∧
(mem₁' x = mem₂' x ∨ dma x = High))
then (mem₁' x, mem₂' x)
else if ((mem₁ x ≠ mem₁' x ∨ mem₂ x ≠ mem₂' x) ∧
(mem₁' x ≠ mem₂' x ∧ dma x = Low))
then (some-val, some-val)
else (fst (mems ! i) x, snd (mems ! i) x)) **for** $i x$

define *mems'*
where *mems'* =
map $(\lambda i.$
if $i = k$
then $(\text{fst} \circ \text{mems}'\text{-}k, \text{snd} \circ \text{mems}'\text{-}k)$
else $(\text{fst} \circ \text{mems}'\text{-}i, \text{snd} \circ \text{mems}'\text{-}i)$
 $[0..< \text{length cms}_1]$
from b **have** *mems'-k-simp: mems' ! k = (fst ∘ mems'-k, snd ∘ mems'-k)*
unfolding *mems'-def*
by *auto*

have *mems'-simp2: $\llbracket i \neq k; i < \text{length cms}_1 \rrbracket \implies$*
mems' ! i = (fst ∘ mems'-i, snd ∘ mems'-i)
unfolding *mems'-def*
by *auto*

have *mems'-k-1 [simp]: $\bigwedge x. \llbracket x \notin \text{?}X k \rrbracket \implies$*
fst (mems' ! k) x = mem₁' x ∧ snd (mems' ! k) x = mem₂' x
unfolding *mems'-k-simp mems'-k-def*
by *auto*

have *mems'-k-2 [simp]: $\bigwedge x. \llbracket x \in \text{?}X k; x \notin \text{dom-}g1 \vee x \notin \text{dom-}g2 \rrbracket \implies$*
fst (mems' ! k) x = mem₁' x ∧ snd (mems' ! k) x = mem₂' x
unfolding *mems'-k-simp mems'-k-def*
by *auto*

have $\text{mems}'\text{-}k\text{-}3$ [simp]: $\bigwedge x. \llbracket x \in ?X\ k; x \in \text{dom}\text{-}g1; x \in \text{dom}\text{-}g2 \rrbracket \implies$
 $\text{fst}(\text{mems}'!k)\ x = \text{fst}(\text{mems}\ !k)\ x \wedge \text{snd}(\text{mems}'!k)\ x = \text{snd}(\text{mems}\ !k)\ x$
unfolding $\text{mems}'\text{-}k\text{-}\text{simp}$ $\text{mems}'\text{-}k\text{-}\text{def}$
by *auto*

have $\text{mems}'\text{-}k\text{-}\text{cases}$:

$\bigwedge P\ x.$
 \llbracket
 $\llbracket x \notin ?X\ k \vee x \notin \text{dom}\text{-}g1 \vee x \notin \text{dom}\text{-}g2;$
 $\text{fst}(\text{mems}'!k)\ x = \text{mem}_1'\ x;$
 $\text{snd}(\text{mems}'!k)\ x = \text{mem}_2'\ x \rrbracket \implies P\ x;$
 $\llbracket x \in ?X\ k; x \in \text{dom}\text{-}g1; x \in \text{dom}\text{-}g2;$
 $\text{fst}(\text{mems}'!k)\ x = \text{fst}(\text{mems}\ !k)\ x;$
 $\text{snd}(\text{mems}'!k)\ x = \text{snd}(\text{mems}\ !k)\ x \rrbracket \implies P\ x \rrbracket \implies P\ x$
using $\text{mems}'\text{-}k\text{-}1$ $\text{mems}'\text{-}k\text{-}2$ $\text{mems}'\text{-}k\text{-}3$
by *blast*

have $\text{mems}'\text{-}i\text{-}\text{simp}$:

$\bigwedge i. \llbracket i < \text{length}\ \text{cms}_1; i \neq k \rrbracket \implies \text{mems}'!i = (\text{fst} \circ \text{mems}'\text{-}i\ i, \text{snd} \circ$
 $\text{mems}'\text{-}i\ i)$
unfolding $\text{mems}'\text{-}\text{def}$
by *auto*

have $\text{mems}'\text{-}i\text{-}1$ [simp]:

$\bigwedge i\ x. \llbracket i \neq k; i < \text{length}\ \text{cms}_1;$
 $\text{mem}_1\ x \neq \text{mem}_1'\ x \vee \text{mem}_2\ x \neq \text{mem}_2'\ x;$
 $\text{mem}_1'\ x = \text{mem}_2'\ x \vee \text{dma}\ x = \text{High} \rrbracket \implies$
 $\text{fst}(\text{mems}'!i)\ x = \text{mem}_1'\ x \wedge \text{snd}(\text{mems}'!i)\ x = \text{mem}_2'\ x$
unfolding $\text{mems}'\text{-}i\text{-}\text{def}$ $\text{mems}'\text{-}i\text{-}\text{simp}$
by *auto*

have $\text{mems}'\text{-}i\text{-}2$ [simp]:

$\bigwedge i\ x. \llbracket i \neq k; i < \text{length}\ \text{cms}_1;$
 $\text{mem}_1\ x \neq \text{mem}_1'\ x \vee \text{mem}_2\ x \neq \text{mem}_2'\ x;$
 $\text{mem}_1'\ x \neq \text{mem}_2'\ x; \text{dma}\ x = \text{Low} \rrbracket \implies$
 $\text{fst}(\text{mems}'!i)\ x = \text{some}\text{-}\text{val} \wedge \text{snd}(\text{mems}'!i)\ x = \text{some}\text{-}\text{val}$
unfolding $\text{mems}'\text{-}i\text{-}\text{def}$ $\text{mems}'\text{-}i\text{-}\text{simp}$
by *auto*

have $\text{mems}'\text{-}i\text{-}3$ [simp]:

$\bigwedge i\ x. \llbracket i \neq k; i < \text{length}\ \text{cms}_1;$
 $\text{mem}_1\ x = \text{mem}_1'\ x; \text{mem}_2\ x = \text{mem}_2'\ x \rrbracket \implies$
 $\text{fst}(\text{mems}'!i)\ x = \text{fst}(\text{mems}\ !i)\ x \wedge \text{snd}(\text{mems}'!i)\ x = \text{snd}(\text{mems}$
 $\ !i)\ x$
unfolding $\text{mems}'\text{-}i\text{-}\text{def}$ $\text{mems}'\text{-}i\text{-}\text{simp}$
by *auto*

have $\text{mems}'\text{-}i\text{-}\text{cases}$:

$\bigwedge P\ i\ x.$
 $\llbracket i \neq k; i < \text{length}\ \text{cms}_1;$


```

    [[ mem1 x ≠ mem1' x ∨ mem2 x ≠ mem2' x;
      mem1' x = mem2' x ∨ dma x = High;
      fst (mems' ! i) x = mem1' x; snd (mems' ! i) x = mem2' x ]] ⇒ P x;
  [[ mem1 x ≠ mem1' x ∨ mem2 x ≠ mem2' x;
    mem1' x ≠ mem2' x; dma x = Low;
    fst (mems' ! i) x = some-val; snd (mems' ! i) x = some-val ]] ⇒ P x;
  [[ mem1 x = mem1' x; mem2 x = mem2' x;
    fst (mems' ! i) x = fst (mems ! i) x; snd (mems' ! i) x = snd (mems ! i) x
  ]] ⇒ P x
  ⇒ P x
  using mems'-i-1 mems'-i-2 mems'-i-3
  by (metis (full-types) Sec.exhaust)

```

```

let ?X' = λ i. differing-vars-lists mem1' mem2' mems' i
show makes-compatible (cms1', mem1' ) (cms2', mem2' ) mems'

```

proof

```

  have length cms1' = length cms1
  by (metis cms2'-def equal-size length-list-update new-length)
  then show length cms1' = length cms2' ∧ length cms1' = length mems'
  using compat new-length
  unfolding mems'-def
  by auto

```

next

```

  fix i
  fix σ :: 'Var → 'Val
  let ?mems1'i = fst (mems' ! i)
  let ?mems2'i = snd (mems' ! i)
  assume i-le: i < length cms1'
  assume domσ: dom σ = ?X' i
  show (cms1' ! i, (fst (mems' ! i)) [↦ σ]) ≈ (cms2' ! i, (snd (mems' ! i)) [↦

```

σ])

proof (cases i = k)

assume [simp]: i = k

— We define another function from this and reuse the universally quantified statements from the first part of the proof.

define σ'

```

  where σ' x =
    (if x ∈ ?X k
     then if x ∈ ?X' k
          then σ x
        else if (x ∈ dom (g1 h))
              then Some (?mems1'i x)
            else if (x ∈ dom (g2 h))
              then Some (?mems2'i x)
            else Some some-val
    else None) for x

```

then have domσ': dom σ' = ?X k

by (auto, metis domI domIff, metis ⟨i = k⟩ domD domσ)

have *diff-vars-impl* [*simp*]: $\bigwedge x. x \in ?X' k \implies x \in ?X k$
proof (*rule ccontr*)
fix x
assume $x \notin ?X k$
hence $mem_1 x = ?mems_1 k x \wedge mem_2 x = ?mems_2 k x$
by (*metis differing-vars-neg*)
from $\langle x \notin ?X k \rangle$ **have** $?mems_1' i x = mem_1' x \wedge ?mems_2' i x = mem_2' x$
by *auto*
moreover
assume $x \in ?X' k$
hence $mem_1' x \neq ?mems_1' i x \vee mem_2' x \neq ?mems_2' i x$
by (*metis* $\langle i = k \rangle$ *differing-vars-elim*)
ultimately show *False*
by *auto*
qed

have *differing-in-dom*: $\bigwedge x. \llbracket x \in ?X k; x \in ?X' k \rrbracket \implies x \in dom-g1 \wedge x \in dom-g2$

proof (*rule ccontr*)
fix x
assume $x \in ?X k$
assume $\neg (x \in dom-g1 \wedge x \in dom-g2)$
hence *not-in-dom*: $x \notin dom-g1 \vee x \notin dom-g2$ **by** *auto*
hence $?mems_1' i x = mem_1' x \wedge ?mems_2' i x = mem_2' x$
using $\langle i = k \rangle \langle x \in ?X k \rangle$ *mems'-k-2*
by *auto*

moreover assume $x \in ?X' k$
ultimately show *False*
by (*metis* $\langle i = k \rangle$ *differing-vars-elim*)
qed

have $?mems_1' i [\mapsto \sigma] = mem_1' [\mapsto g1 \sigma]$

proof (*rule ext*)

fix x

show $?mems_1' i [\mapsto \sigma] x = mem_1' [\mapsto g1 \sigma] x$

proof (*cases* $x \in ?X' k$)

assume *x-in-X'k*: $x \in ?X' k$

then obtain v **where** $\sigma x = Some v$

by (*metis* *dom σ domD* $\langle i = k \rangle$)

hence $?mems_1' i [\mapsto \sigma] x = v$

using $\langle x \in ?X' k \rangle$ *dom σ*

by (*auto simp: subst-def*)

moreover

from c **have** le : $g1 \sigma' \preceq \sigma'$

using *dom σ'*

by *auto*
from $\text{dom}\sigma'$ **and** $\langle x \in ?X' k \rangle$ **have** $x \in \text{dom} (g1 \sigma')$
 by (*metis diff-vars-impl differing-in-dom dom-uniq(1)*)

hence $\text{mem}_1' [\mapsto g1 \sigma'] x = v$
 using *dom σ' c le*
unfolding *func-le-def subst-def*
by (*metis σ' -def $\langle \sigma x = \text{Some } v \rangle$ diff-vars-impl option.simps(5) x-in-X'k*)

ultimately show $?mems_1'i [\mapsto \sigma] x = \text{mem}_1' [\mapsto g1 \sigma'] x ..$
next
assume $x \notin ?X' k$

hence $?mems_1'i [\mapsto \sigma] x = ?mems_1'i x$
 using *dom σ*
by (*metis $\langle i = k \rangle$ subst-not-in-dom*)
show *?thesis*
proof (*cases x \in dom-g1*)
assume $x \in \text{dom-g1}$
hence $x \in \text{dom} (g1 \sigma')$
 using *dom σ' dom-uniq*
by *auto*
hence $g1 \sigma' x = \sigma' x$
 using *c dom σ'*
by (*metis change-respecting.simps func-le-def*)
then have $\sigma' x = \text{Some} (?mems_1'i x)$
unfolding *σ' -def*
using *dom σ' domh*
by (*metis $\langle g1 \sigma' x = \sigma' x \rangle$ $\langle x \in \text{dom} (g1 \sigma') \rangle$ $\langle x \notin ?X' k \rangle$ domIff dom-uniq(1)*)

hence $\text{mem}_1' [\mapsto g1 \sigma'] x = ?mems_1'i x$
unfolding *subst-def*
by (*metis $\langle g1 \sigma' x = \sigma' x \rangle$ option.simps(5)*)
thus *?thesis*
by (*metis $\langle ?mems_1'i [\mapsto \sigma] x = ?mems_1'i x \rangle$*)
next
assume $x \notin \text{dom-g1}$
then have $\text{mem}_1' [\mapsto g1 \sigma'] x = \text{mem}_1' x$
by (*metis dom σ' dom-uniq(1) subst-not-in-dom*)
moreover
have $?mems_1'i x = \text{mem}_1' x$
by (*metis $\langle i = k \rangle$ $\langle x \notin ?X' k \rangle$ differing-vars-neg*)
ultimately show *?thesis*
by (*metis $\langle ?mems_1'i [\mapsto \sigma] x = ?mems_1'i x \rangle$*)
qed
qed
qed

moreover have $?mems_2'i [\mapsto \sigma] = mem_h [\mapsto g_2 \sigma']$
proof (*rule ext*)
fix x

show $?mems_2'i [\mapsto \sigma] x = mem_h [\mapsto g_2 \sigma'] x$
proof (*cases* $x \in ?X' k$)
assume $x \in ?X' k$

then obtain v **where** $\sigma x = Some\ v$
using $dom\sigma$
by (*metis* $domD \langle i = k \rangle$)
hence $?mems_2'i [\mapsto \sigma] x = v$
using $\langle x \in ?X' k \rangle dom\sigma$
unfolding $subst-def$
by (*metis* $option.simps(5)$)

moreover
from e **have** $le: g_2 \sigma' \preceq \sigma'$
using $dom\sigma'$
by *auto*

from $\langle x \in ?X' k \rangle$ **have** $x \in ?X\ k$
by *auto*

hence $x \in dom\ (g_2 \sigma')$
by (*metis* $differing-in-dom\ dom\sigma'\ dom-uniq(2) \langle x \in ?X' k \rangle$)
hence $mem_2' [\mapsto g_2 \sigma'] x = v$
using $dom\sigma' c\ le$
unfolding $func-le-def\ subst-def$
by (*metis* $\sigma'-def \langle \sigma x = Some\ v \rangle diff-vars-impl\ option.simps(5) \langle x \in ?X' k \rangle$)

ultimately show $?thesis$
by (*metis* $dom\sigma'\ dom-restrict-total\ dom-uniq(2)\ mem_2'-def\ subst-overrides$)
next
assume $x \notin ?X' k$

hence $?mems_2'i [\mapsto \sigma] x = ?mems_2'i x$
using $dom\sigma$
by (*metis* $\langle i = k \rangle\ subst-not-in-dom$)

show $?thesis$
proof (*cases* $x \in dom-g_2$)
assume $x \in dom-g_2$
hence $x \in dom\ (g_2 \sigma')$
using $dom\sigma'$
by (*metis* $dom-uniq$)
hence $g_2 \sigma' x = \sigma' x$
using $e\ dom\sigma'$
by (*metis* $change-respecting.simps\ func-le-def$)

then have $\sigma' x = Some\ (?mems_2'i x)$
proof (*cases* $x \in dom-g_1$)
— This can't happen, so derive a contradiction.

```

assume  $x \in \text{dom-g1}$ 

have  $x \notin ?X\ k$ 
proof (rule ccontr)
  assume  $\neg (x \notin ?X\ k)$ 
  hence  $x \in ?X\ k$  by auto
  have  $\text{mem}_1\ x = \text{mem}_1'\ x \wedge \text{mem}_2\ x = \text{mem}_2'\ x$ 
    by (metis  $\sigma'$ -def  $\langle g2\ \sigma'\ x = \sigma'\ x \rangle \langle x \in \text{dom}\ (g2\ \sigma') \rangle$ 
       $\langle x \in \text{dom-g1} \rangle \langle x \in \text{dom-g2} \rangle \text{domIff}\ x\text{-unchanged}$ )
  moreover
  from  $\langle x \notin ?X'\ k \rangle$  have
     $?mems_1'\ i\ x = ?mems_1\ k\ x \wedge ?mems_2'\ i\ x = ?mems_2\ k\ x$ 
    using  $\langle x \in ?X\ k \rangle \langle x \in \text{dom-g1} \rangle \langle x \in \text{dom-g2} \rangle$ 
    by auto
  ultimately show False
    using  $\langle x \in ?X\ k \rangle \langle x \notin ?X'\ k \rangle$ 
    by (metis  $\langle i = k \rangle$  differing-vars-elim differing-vars-neg)
qed
hence False
  by (metis  $\sigma'$ -def  $\langle g2\ \sigma'\ x = \sigma'\ x \rangle \langle x \in \text{dom}\ (g2\ \sigma') \rangle \text{domIff}$ )
thus ?thesis
  by blast
next
assume  $x \notin \text{dom-g1}$ 
thus ?thesis
  unfolding  $\sigma'$ -def
  by (metis  $\langle g2\ \sigma'\ x = \sigma'\ x \rangle \langle x \in \text{dom}\ (g2\ \sigma') \rangle \langle x \notin ?X'\ k \rangle$ 
    domIff dom $\sigma'$  dom-uniq domh)
qed
hence  $\text{mem}_2'\ [\mapsto g2\ \sigma']\ x = ?mems_2'\ i\ x$ 
  unfolding subst-def
  by (metis  $\langle g2\ \sigma'\ x = \sigma'\ x \rangle$  option.simps(5))
thus ?thesis
  using  $\langle x \notin ?X'\ k \rangle \text{dom}\sigma\ \text{dom}\sigma'$ 
  by (metis  $\langle i = k \rangle$  dom-restrict-total dom-uniq(2)
    mem $_2'$ -def subst-not-in-dom subst-overrides)
next
assume  $x \notin \text{dom-g2}$ 
then have  $\text{mem}_h\ [\mapsto g2\ \sigma']\ x = \text{mem}_h\ x$ 
  by (metis dom $\sigma'$  dom-uniq(2) subst-not-in-dom)
moreover
have  $?mems_2'\ i\ x = \text{mem}_2'\ x$ 
  by (metis  $\langle i = k \rangle \langle x \notin \text{dom-g2} \rangle$  mems'-k-1 mems'-k-2)

hence  $?mems_2'\ i\ x = \text{mem}_h\ x$ 
  unfolding mem $_2'$ -def
  by (metis  $\langle x \notin \text{dom-g2} \rangle \text{dom}\sigma\text{-mem}_2\ \text{dom-uniq(2)}\ \text{subst-not-in-dom}$ )
ultimately show ?thesis
  by (metis  $\langle ?mems_2'\ i\ [\mapsto \sigma]\ x = ?mems_2'\ i\ x \rangle$ )

```

```

    qed
  qed
qed

ultimately show
  (cms1' ! i, (fst (mems' ! i)) [↦ σ]) ≈ (cms2' ! i, (snd (mems' ! i)) [↦ σ])
  using domσ domσ' g b ⟨i = k⟩
  by (metis c2'-def cms2'-def equal-size nth-list-update-eq)

next
  assume i ≠ k
  define σ'
    where σ' x = (if x ∈ ?X i
                  then if x ∈ ?X' i
                       then σ x
                       else Some (mem1' x)
                  else None) for x
  let ?mems1i = fst (mems ! i) and
      ?mems2i = snd (mems ! i)
  have dom σ' = ?X i
    unfolding σ'-def
    apply auto
    apply (metis option.simps(2))
    by (metis domD domσ)
  have o: ∧ x.
    (?mems1'i [↦ σ] x ≠ ?mems1i [↦ σ'] x ∨
     ?mems2'i [↦ σ] x ≠ ?mems2i [↦ σ'] x)
    → (mem1' x ≠ mem1 x ∨ mem2' x ≠ mem2 x)
  proof -
    fix x
    {
      assume eq-mem: mem1' x = mem1 x ∧ mem2' x = mem2 x
      hence mems'-simp [simp]: ?mems1'i x = ?mems1i x ∧ ?mems2'i x =
?mems2i x
      using mems'-i-3
      by (metis ⟨i ≠ k⟩ b i-le length-list-update)
      have
        ?mems1'i [↦ σ] x = ?mems1i [↦ σ'] x ∧ ?mems2'i [↦ σ] x = ?mems2i
[↦ σ'] x
      proof (cases x ∈ ?X' i)
        assume x ∈ ?X' i
        hence ?mems1'i x ≠ mem1' x ∨ ?mems2'i x ≠ mem2' x
          by (metis differing-vars-neg-intro)
        hence x ∈ ?X i
          using eq-mem mems'-simp
          by (metis differing-vars-neg)
        hence σ' x = σ x
          by (metis σ'-def ⟨x ∈ ?X' i⟩)
        thus ?thesis
      }
    }
  }

```

```

      apply (auto simp: subst-def)
      apply (metis mems'-simp)
      by (metis mems'-simp)
    next
      assume  $x \notin ?X' i$ 
      hence  $?mems_1' i x = mem_1' x \wedge ?mems_2' i x = mem_2' x$ 
      by (metis differing-vars-neg)
      hence  $x \notin ?X i$ 
      using eq-mem mems'-simp
      by (auto simp: differing-vars-neg-intro)
      thus ?thesis
      by (metis  $\langle dom \sigma' = ?X i \rangle \langle x \notin ?X' i \rangle dom \sigma mems'-simp$ 
subst-not-in-dom)
    qed
  }
  thus ?thesis x by blast
qed

from o have
   $p: \bigwedge x. [\![?mems_1' i \mapsto \sigma] x \neq ?mems_1 i \mapsto \sigma^\frown] x \vee$ 
     $[\![?mems_2' i \mapsto \sigma] x \neq ?mems_2 i \mapsto \sigma^\frown] x \implies$ 
     $x \notin snd (cms_1 ! i) AsmNoWrite$ 
proof
  fix x
  assume mems-neg:
     $[\![?mems_1' i \mapsto \sigma] x \neq ?mems_1 i \mapsto \sigma^\frown] x \vee [\![?mems_2' i \mapsto \sigma] x \neq ?mems_2 i$ 
 $\mapsto \sigma^\frown] x$ 
  hence modified:
     $\neg (doesnt-modify (fst (cms_1 ! k)) x) \vee \neg (doesnt-modify (fst (cms_2 ! k))$ 
x)
    using b i o
    unfolding doesnt-modify-def
    by (metis surjective-pairing)
  moreover
  from sound-modes have loc-modes:
    locally-sound-mode-use (cms_1 ! k, mem_1)  $\wedge$ 
    locally-sound-mode-use (cms_2 ! k, mem_2)
    unfolding sound-mode-use.simps
    by (metis b equal-size list-all-length)
  moreover
  have  $snd (cms_1 ! k) = snd (cms_2 ! k)$ 
    by (metis b equal-size modes-eq nth-map)
  have  $(cms_1 ! k, mem_1) \in loc-reach (cms_1 ! k, mem_1)$ 
    by (metis loc-reach.refl prod.collapse)
  hence guar:
     $x \in snd (cms_1 ! k) GuarNoWrite \implies doesnt-modify (fst (cms_1 ! k))$ 
x  $\wedge$ 
     $x \in snd (cms_2 ! k) GuarNoWrite \implies doesnt-modify (fst (cms_1 ! k)) x$ 
    using loc-modes

```

unfolding *locally-sound-mode-use-def* $\langle \text{snd } (cms_1 ! k) = \text{snd } (cms_2 ! k) \rangle$
by (*metis loc-reach.refl surjective-pairing*)

hence $x \notin \text{snd } (cms_1 ! k)$ *GuarNoWrite*
using *modified loc-modes locally-sound-mode-use-def*
by (*metis* $\langle \text{snd } (cms_1 ! k) = \text{snd } (cms_2 ! k) \rangle$ *loc-reach.refl prod.collapse*)
moreover
from *sound-modes* **have** *compatible-modes* (*map snd cms₁*)
by (*metis globally-sound-modes-compatible sound-mode-use.simps*)

ultimately show $x \notin \text{snd } (cms_1 ! i)$ *AsmNoWrite*
unfolding *compatible-modes-def*
using $\langle i \neq k \rangle$ *i-le*
by (*metis* (*no-types*) *b length-list-update length-map nth-map*)

qed

have *q*:

$\bigwedge x. \llbracket \text{dma } x = \text{Low};$
 $\quad ?\text{mems}_1' i \llbracket \mapsto \sigma \rrbracket x \neq ?\text{mems}_1 i \llbracket \mapsto \sigma' \rrbracket x \vee$
 $\quad ?\text{mems}_2' i \llbracket \mapsto \sigma \rrbracket x \neq ?\text{mems}_2 i \llbracket \mapsto \sigma' \rrbracket x;$
 $\quad x \notin ?X' i \rrbracket \implies$
 $\quad \text{mem}_1' x = \text{mem}_2' x$

by (*metis* $\langle i \neq k \rangle$ *b compat-different-vars i-le length-list-update mems'-i-2*)

o)

have $i < \text{length } cms_1$

by (*metis cms₂'-def equal-size i-le length-list-update new-length*)

with *compat* **and** $\langle \text{dom } \sigma' = ?X i \rangle$ **have**

bisim: $(cms_1 ! i, ?\text{mems}_1 i \llbracket \mapsto \sigma' \rrbracket) \approx (cms_2 ! i, ?\text{mems}_2 i \llbracket \mapsto \sigma' \rrbracket)$

by *auto*

let $? \Delta = \text{differing-vars } (? \text{mems}_1 i \llbracket \mapsto \sigma' \rrbracket) (? \text{mems}_1' i \llbracket \mapsto \sigma \rrbracket) \cup$
 $\text{differing-vars } (? \text{mems}_2 i \llbracket \mapsto \sigma' \rrbracket) (? \text{mems}_2' i \llbracket \mapsto \sigma \rrbracket)$

have Δ -*finite*: *finite* $? \Delta$

by (*metis* (*no-types*) *differing-finite finite-UnI*)

— We first define the adaptation, then prove that it does the right thing.

define *A* **where** $A x =$

$(\text{if } x \in ? \Delta$
 $\quad \text{then if } \text{dma } x = \text{High}$
 $\quad \quad \text{then } \text{Some } (? \text{mems}_1' i \llbracket \mapsto \sigma \rrbracket x, ? \text{mems}_2' i \llbracket \mapsto \sigma \rrbracket x)$
 $\quad \quad \text{else if } x \in ?X' i$
 $\quad \quad \quad \text{then (case } \sigma x \text{ of}$
 $\quad \quad \quad \quad \text{Some } v \Rightarrow \text{Some } (v, v)$
 $\quad \quad \quad \quad | \text{None} \Rightarrow \text{None})$
 $\quad \quad \quad \quad \text{else } \text{Some } (\text{mem}_1' x, \text{mem}_1' x)$
 $\quad \text{else None) for } x$

have *domA*: $\text{dom } A = ? \Delta$

proof

show $\text{dom } A \subseteq ? \Delta$


```

    using A-def
    apply (auto simp: domD)
    by (metis option.simps(2))
next
show ? $\Delta \subseteq \text{dom } A$ 
  unfolding A-def
  apply auto
  apply (metis (no-types) domIff dom $\sigma$  option.exhaust option.simps(5))
  by (metis (no-types) domIff dom $\sigma$  option.exhaust option.simps(5))
qed

have A-correct:
   $\bigwedge x.$ 
  globally-consistent-var A (snd (cms1 ! i)) x  $\wedge$ 
  ?mems1i [math>\mapsto \sigma'] [|1 A] x = ?mems1'i [math>\mapsto \sigma] x  $\wedge$ 
  ?mems2i [math>\mapsto \sigma'] [|2 A] x = ?mems2'i [math>\mapsto \sigma] x
proof -
fix x
show ?thesis x
  (is ?A  $\wedge$  ?Eq1  $\wedge$  ?Eq2)
proof (cases x  $\in$  ? $\Delta$ )
assume x  $\in$  ? $\Delta$ 
hence diff:
  ?mems1'i [math>\mapsto \sigma] x  $\neq$  ?mems1i [math>\mapsto \sigma'] x  $\vee$  ?mems2'i [math>\mapsto \sigma] x  $\neq$  ?mems2i
[math>\mapsto \sigma'] x
  by (auto simp: differing-vars-def)
from p and diff have writable: x  $\notin$  snd (cms1 ! i) AsmNoWrite
  by auto
show ?thesis
proof (cases dma x)
assume dma x = High
from  $\langle \text{dma } x = \text{High} \rangle$  have A-simp [simp]:
  A x = Some (?mems1'i [math>\mapsto \sigma] x, ?mems2'i [math>\mapsto \sigma] x)
  unfolding A-def
  by (metis  $\langle x \in ?\Delta \rangle$ )
from writable have ?A
  unfolding globally-consistent-var-def A-simp
  using  $\langle \text{dma } x = \text{High} \rangle$ 
  by auto
moreover
from A-simp have ?Eq1 ?Eq2
  unfolding A-def apply-adaptation-def
  by auto
ultimately show ?thesis
  by auto
next
assume dma x = Low
show ?thesis
proof (cases x  $\in$  ?X' i)

```

```

assume  $x \in ?X' i$ 
then obtain  $v$  where  $\sigma x = \text{Some } v$ 
  by (metis domD dom $\sigma$ )
hence eq:  $?mems_1' i [\mapsto \sigma] x = v \wedge ?mems_2' i [\mapsto \sigma] x = v$ 
  unfolding subst-def
  by auto
moreover
from  $\langle x \in ?X' i \rangle$  and  $\langle dma\ x = Low \rangle$  have  $A\text{-simp}$  [simp]:
   $A\ x = (\text{case } \sigma\ x\ \text{of}$ 
     $\quad \text{Some } v \Rightarrow \text{Some } (v, v)$ 
     $\quad | \text{None} \Rightarrow \text{None})$ 
  unfolding A-def
  by (metis Sec.simps(1) \langle x \in ?\Delta \rangle)
with writable eq  $\langle \sigma\ x = \text{Some } v \rangle$  have  $?A$ 
  unfolding globally-consistent-var-def
  by auto
ultimately show ?thesis
  using domA \langle x \in ?\Delta \rangle \langle \sigma\ x = \text{Some } v \rangle
  by (auto simp: apply-adaptation-def)

next
assume  $x \notin ?X' i$ 
hence  $A\text{-simp}$  [simp]:  $A\ x = \text{Some } (mem_1' x, mem_1' x)$ 
  unfolding A-def
  using  $\langle x \in ?\Delta \rangle \langle dma\ x = Low \rangle$ 
  by auto
from  $q$  have  $mem_1' x = mem_2' x$ 
  by (metis \langle dma\ x = Low \rangle diff \langle x \notin ?X' i \rangle)
with writable have  $?A$ 
  unfolding globally-consistent-var-def
  by auto

moreover
from  $\langle x \notin ?X' i \rangle$  have
   $?mems_1' i [\mapsto \sigma] x = ?mems_1' i x \wedge ?mems_2' i [\mapsto \sigma] x = ?mems_2' i x$ 
  by (metis dom $\sigma$  subst-not-in-dom)
moreover
from  $\langle x \notin ?X' i \rangle$  have  $?mems_1' i x = mem_1' x \wedge ?mems_2' i x =$ 
 $mem_2' x$ 
  by (metis differing-vars-neg)
ultimately show ?thesis
  using  $\langle mem_1' x = mem_2' x \rangle$ 
  by (auto simp: apply-adaptation-def)
qed
qed
next
assume  $x \notin ?\Delta$ 
hence  $A\ x = None$ 
  by (metis domA domIff)

```

```

hence globally-consistent-var  $A$  (snd ( $cms_1 ! i$ ))  $x$ 
  by (auto simp: globally-consistent-var-def)
moreover
from  $\langle A \ x = None \rangle$  have  $x \notin \text{dom } A$ 
  by (metis domIff)
from  $\langle x \notin ?\Delta \rangle$  have  $?mems_1 i [\mapsto \sigma'] [\|_1 A] \ x = ?mems_1' i [\mapsto \sigma] \ x \wedge$ 
   $?mems_2 i [\mapsto \sigma'] [\|_2 A] \ x = ?mems_2' i [\mapsto \sigma] \ x$ 
  using  $\langle A \ x = None \rangle$ 
  unfolding differing-vars-def apply-adaptation-def
  by auto

ultimately show ?thesis
  by auto
qed
qed
hence  $?mems_1 i [\mapsto \sigma'] [\|_1 A] = ?mems_1' i [\mapsto \sigma] \wedge$ 
   $?mems_2 i [\mapsto \sigma'] [\|_2 A] = ?mems_2' i [\mapsto \sigma]$ 
  by auto
moreover
from A-correct have globally-consistent  $A$  (snd ( $cms_1 ! i$ ))
  by (metis  $\Delta$ -finite globally-consistent-def dom.A)

have snd ( $cms_1 ! i$ ) = snd ( $cms_2 ! i$ )
  by (metis  $\langle i < \text{length } cms_1 \rangle$  equal-size modes-eq nth-map)

with bisim have ( $cms_1 ! i, ?mems_1 i [\mapsto \sigma'] [\|_1 A]$ )  $\approx$  ( $cms_2 ! i, ?mems_2 i$ 
 $[\mapsto \sigma'] [\|_2 A]$ )
  using  $\langle \text{globally-consistent } A \ (\text{snd } (cms_1 ! i)) \rangle$ 
  apply (subst surjective-pairing[of  $cms_1 ! i$ ])
  apply (subst surjective-pairing[of  $cms_2 ! i$ ])
  by (metis surjective-pairing globally-consistent-adapt-bisim)

ultimately show ?thesis
  by (metis  $\langle i \neq k \rangle$  b cms_2'-def nth-list-update-neq)
qed
next
fix  $i \ x$ 

let  $?mems_1' i = \text{fst } (mems' ! i)$ 
let  $?mems_2' i = \text{snd } (mems' ! i)$ 
assume i-le:  $i < \text{length } cms_1'$ 
assume mem-eq:  $mem_1' \ x = mem_2' \ x \vee dma \ x = High$ 
show  $x \notin ?X' \ i$ 
proof (cases  $i = k$ )
  assume  $i = k$ 
  thus  $x \notin ?X' \ i$ 
  apply (cases  $x \notin ?X \ k \vee x \notin \text{dom-g1} \vee x \notin \text{dom-g2}$ )
  apply (metis differing-vars-neg-intro mems'-k-1 mems'-k-2)
  by (metis Sec.simps(2) b compat compat-different mem-eq x-unchanged)

```

```

next
  assume  $i \neq k$ 
  thus  $x \notin ?X' i$ 
  proof (rule mems'-i-cases)
    from  $b$  i-le show  $i < \text{length } cms_1$ 
    by (metis length-list-update)
  next
  assume  $\text{fst } (mems' ! i) x = mem_1' x$ 
     $\text{snd } (mems' ! i) x = mem_2' x$ 
  thus  $x \notin ?X' i$ 
    by (metis differing-vars-neg-intro)
  next
  assume  $mem_1 x \neq mem_1' x \vee mem_2 x \neq mem_2' x$ 
     $mem_1' x \neq mem_2' x$  and  $\text{dma } x = Low$ 
  — In this case, for example, the values of  $(mems' ! i)$  are not needed.
  thus  $x \notin ?X' i$ 
    by (metis Sec.simps(2) mem-eq)
  next
  assume case3:  $mem_1 x = mem_1' x \wedge mem_2 x = mem_2' x$ 
     $\text{fst } (mems' ! i) x = \text{fst } (mems ! i) x$ 
     $\text{snd } (mems' ! i) x = \text{snd } (mems ! i) x$ 
  have  $x \in ?X' i \implies mem_1' x \neq mem_2' x \wedge \text{dma } x = Low$ 
  proof —
    assume  $x \in ?X' i$ 
    from case3 and  $\langle x \in ?X' i \rangle$  have  $x \in ?X i$ 
      by (metis differing-vars-neg differing-vars-elim)
    with case3 show ?thesis
      by (metis b compat compat-different i-le length-list-update)
  qed
  with  $\langle mem_1' x = mem_2' x \vee \text{dma } x = High \rangle$  show  $x \notin ?X' i$ 
    by auto
  qed
next
next
{ fix  $x$ 
  have  $\exists i < \text{length } cms_1. x \notin ?X' i$ 
  proof (cases  $mem_1 x \neq mem_1' x \vee mem_2 x \neq mem_2' x$ )
    assume var-changed:  $mem_1 x \neq mem_1' x \vee mem_2 x \neq mem_2' x$ 
    have  $x \notin ?X' k$ 
      apply (rule mems'-k-cases)
      apply (metis differing-vars-neg-intro)
      by (metis var-changed x-unchanged)
    thus ?thesis by (metis b)
  next
  assume  $\neg (mem_1 x \neq mem_1' x \vee mem_2 x \neq mem_2' x)$ 
  hence assms:  $mem_1 x = mem_1' x \wedge mem_2 x = mem_2' x$  by auto

  have  $\text{length } cms_1 \neq 0$ 
    using  $b$ 

```

```

    by (metis less-zeroE)
  then obtain i where i-prop: i < length cms1 ∧ x ∉ ?X i
    using compat
    by (auto, blast)
  show ?thesis
  proof (cases i = k)
    assume i = k
    have x ∉ ?X' k
      apply (rule mems'-k-cases)
      apply (metis differing-vars-neg-intro)
      by (metis i-prop ⟨i = k⟩)
    thus ?thesis
      by (metis b)
  next
    assume i ≠ k
    hence fst (mems' ! i) x = fst (mems ! i) x
      snd (mems' ! i) x = snd (mems ! i) x
      using i-prop assms mems'-i-3
      by auto
    with i-prop have x ∉ ?X' i
      by (metis assms differing-vars-neg differing-vars-neg-intro)
    with i-prop show ?thesis
      by auto
  qed
}
}
thus (length cms1' = 0 ∧ mem1' =l mem2') ∨ (∀ x. ∃ i < length cms1'. x ∉
?X' i)
  by (metis cms2'-def equal-size length-list-update new-length)
qed
qed

```

ultimately show ?thesis using that by blast
qed

The Isar proof language provides a readable way of specifying assumptions while also giving them names for subsequent usage.

lemma *compat-low-eq*:

```

  assumes compat: makes-compatible (cms1, mem1) (cms2, mem2) mems
  assumes modes-eq: map snd cms1 = map snd cms2
  assumes x-low: dma x = Low
  assumes x-readable: ∀ i < length cms1. x ∉ snd (cms1 ! i) AsmNoRead
  shows mem1 x = mem2 x
  proof –
    let ?X = λ i. differing-vars-lists mem1 mem2 mems i
    from compat have (length cms1 = 0 ∧ mem1 =l mem2) ∨
      (∀ x. ∃ j. j < length cms1 ∧ x ∉ ?X j)
      by auto
    thus mem1 x = mem2 x
  
```

proof
 assume $\text{length } cms_1 = 0 \wedge mem_1 =^l mem_2$
 with $x\text{-low}$ show $?thesis$
 by (*simp add: low-eq-def*)
next
 assume $\forall x. \exists j. j < \text{length } cms_1 \wedge x \notin ?X j$
 then obtain j where $j\text{-prop}: j < \text{length } cms_1 \wedge x \notin ?X j$
 by *auto*
 let $?mems_1j = \text{fst } (mems ! j)$ and
 $?mems_2j = \text{snd } (mems ! j)$

 obtain $\sigma :: 'Var \rightarrow 'Val$ where $\text{dom } \sigma = ?X j$
 by (*metis dom-restrict-total*)

 with *compat* and $j\text{-prop}$ have $(cms_1 ! j, ?mems_1j [\mapsto \sigma]) \approx (cms_2 ! j, ?mems_2j [\mapsto \sigma])$
 by *auto*

moreover
 have $\text{snd } (cms_1 ! j) = \text{snd } (cms_2 ! j)$
 using *modes-eq*
 by (*metis j-prop length-map nth-map*)

 ultimately have $?mems_1j [\mapsto \sigma] = \text{snd } (cms_1 ! j)^l ?mems_2j [\mapsto \sigma]$
 using *modes-eq j-prop*
 by (*metis prod.collapse mm-equiv-low-eq*)
 hence $?mems_1j x = ?mems_2j x$
 using $x\text{-low } x\text{-readable } j\text{-prop } \langle \text{dom } \sigma = ?X j \rangle$
 unfolding *low-mds-eq-def*
 by (*metis subst-not-in-dom*)

 thus $?thesis$
 using $j\text{-prop}$
 by (*metis compat-different-vars*)
qed
qed

lemma *loc-reach-subset*:

assumes $\text{eval}: \langle c, mds, mem \rangle \rightsquigarrow \langle c', mds', mem' \rangle$
 shows $\text{loc-reach } \langle c', mds', mem' \rangle \subseteq \text{loc-reach } \langle c, mds, mem \rangle$
proof (*clarify*)
 fix $c'' mds'' mem''$
 from *eval* have $\langle c', mds', mem' \rangle \in \text{loc-reach } \langle c, mds, mem \rangle$
 by (*metis loc-reach.refl loc-reach.step surjective-pairing*)
 assume $\langle c'', mds'', mem'' \rangle \in \text{loc-reach } \langle c', mds', mem' \rangle$
 thus $\langle c'', mds'', mem'' \rangle \in \text{loc-reach } \langle c, mds, mem \rangle$
 apply *induct*
 apply (*metis* $\langle c', mds', mem' \rangle \in \text{loc-reach } \langle c, mds, mem \rangle$) *surjective-pairing*)
 apply (*metis loc-reach.step*)

by (*metis loc-reach.mem-diff*)
 qed

lemma *locally-sound-modes-invariant*:

assumes *sound-modes: locally-sound-mode-use* $\langle c, mds, mem \rangle$
 assumes *eval*: $\langle c, mds, mem \rangle \rightsquigarrow \langle c', mds', mem' \rangle$
 shows *locally-sound-mode-use* $\langle c', mds', mem' \rangle$

proof –

from *eval* have $\langle c', mds', mem' \rangle \in \text{loc-reach } \langle c, mds, mem \rangle$
 by (*metis fst-conv loc-reach.refl loc-reach.step snd-conv*)
 thus ?thesis
 using *sound-modes*
 unfolding *locally-sound-mode-use-def*
 by (*metis (no-types) Collect-empty-eq eval loc-reach-subset subsetD*)

qed

lemma *reachable-modes-subset*:

assumes *eval*: $(cms, mem) \rightarrow (cms', mem')$
 shows *reachable-mode-states* $(cms', mem') \subseteq \text{reachable-mode-states } (cms, mem)$

proof

fix *mdss*
 assume *mdss* $\in \text{reachable-mode-states } (cms', mem')$
 thus *mdss* $\in \text{reachable-mode-states } (cms, mem)$
 using *reachable-mode-states-def*
 apply *auto*
 by (*metis (opaque-lifting, no-types) assms converse-rtrancl-into-rtrancl*)

qed

lemma *globally-sound-modes-invariant*:

assumes *globally-sound: globally-sound-mode-use* (cms, mem)
 assumes *eval*: $(cms, mem) \rightarrow (cms', mem')$
 shows *globally-sound-mode-use* (cms', mem')
 using *assms reachable-modes-subset*
 unfolding *globally-sound-mode-use-def*
 by (*metis (no-types) subsetD*)

lemma *loc-reach-mem-diff-subset*:

assumes *mem-diff*: $\forall x. x \in mds \text{ AsmNoWrite} \longrightarrow mem_1 x = mem_2 x$
 shows $\langle c', mds', mem' \rangle \in \text{loc-reach } \langle c, mds, mem_1 \rangle \implies \langle c', mds', mem' \rangle \in \text{loc-reach } \langle c, mds, mem_2 \rangle$

proof –

let ?lc = $\langle c', mds', mem' \rangle$
 assume ?lc $\in \text{loc-reach } \langle c, mds, mem_1 \rangle$
 thus ?thesis
proof (*induct*)
 case *refl*
 thus ?case
 by (*metis fst-conv loc-reach.mem-diff loc-reach.refl mem-diff snd-conv*)

next

```

    case step
  thus ?case
    by (metis loc-reach.step)
next
  case mem-diff
  thus ?case
    by (metis loc-reach.mem-diff)
qed
qed

```

lemma *loc-reach-mem-diff-eq*:

```

  assumes mem-diff:  $\forall x. x \in mds \text{ AsmNoWrite} \longrightarrow mem' x = mem x$ 
  shows loc-reach  $\langle c, mds, mem \rangle = loc-reach \langle c, mds, mem' \rangle$ 
  using assms loc-reach-mem-diff-subset
  by (auto, metis)

```

lemma *sound-modes-invariant*:

```

  assumes sound-modes: sound-mode-use (cms, mem)
  assumes eval: (cms, mem)  $\rightarrow$  (cms', mem')
  shows sound-mode-use (cms', mem')

```

proof –

```

  from sound-modes and eval have globally-sound-mode-use (cms', mem')
  by (metis globally-sound-modes-invariant sound-mode-use.simps)

```

moreover

```

  from sound-modes have loc-sound:  $\forall i < length cms. locally-sound-mode-use$ 
  (cms ! i, mem)

```

```

  unfolding sound-mode-use-def

```

```

  by simp (metis list-all-length)

```

from eval **obtain** k cms_k' **where**

```

  ev: (cms ! k, mem)  $\rightsquigarrow$  (cms_k', mem')  $\wedge k < length cms \wedge cms' = cms [k :=$ 
  cms_k']

```

```

  by (metis meval-elim)

```

```

  hence length cms = length cms'

```

```

  by auto

```

```

  have  $\bigwedge i. i < length cms' \implies locally-sound-mode-use (cms' ! i, mem')$ 

```

proof –

```

  fix i

```

```

  assume i-le:  $i < length cms'$ 

```

```

  thus ?thesis i

```

proof (*cases* $i = k$)

```

  assume  $i = k$ 

```

```

  thus ?thesis

```

```

  using i-le ev loc-sound

```

```

  by (metis (opaque-lifting, no-types) locally-sound-modes-invariant nth-list-update
  surj-pair)

```

next

```

  assume  $i \neq k$ 

```

```

  hence cms' ! i = cms ! i

```

```

  by (metis ev nth-list-update-neq)

```


from *sound-modes* **have** *compatible-modes* (map snd cms)
unfolding *sound-mode-use.simps*
by (metis *globally-sound-modes-compatible*)
hence $\bigwedge x. x \in \text{snd } (cms ! i) \text{ AsmNoWrite} \implies x \in \text{snd } (cms ! k) \text{ GuarNoWrite}$
unfolding *compatible-modes-def*
by (metis (no-types) $\langle i \neq k \rangle \langle \text{length } cms = \text{length } cms' \rangle \text{ ev } i\text{-le } \text{length-map}$
nth-map)
hence $\bigwedge x. x \in \text{snd } (cms ! i) \text{ AsmNoWrite} \longrightarrow \text{doesn't-modify } (\text{fst } (cms ! k))$
x
using *ev loc-sound*
unfolding *locally-sound-mode-use-def*
by (metis *loc-reach.refl surjective-pairing*)
with *eval* **have** $\bigwedge x. x \in \text{snd } (cms ! i) \text{ AsmNoWrite} \longrightarrow \text{mem } x = \text{mem}' x$
by (metis (no-types) *doesn't-modify-def ev prod.collapse*)
then **have** *loc-reach* (cms ! i, mem') = *loc-reach* (cms ! i, mem)
by (metis *loc-reach-mem-diff-eq prod.collapse*)
thus ?thesis
using *loc-sound i-le* $\langle \text{length } cms = \text{length } cms' \rangle$
unfolding *locally-sound-mode-use-def*
by (metis $\langle cms' ! i = cms ! i \rangle$)
qed
qed
ultimately show ?thesis
unfolding *sound-mode-use.simps*
by (metis (no-types) *list-all-length*)
qed

lemma *makes-compatible-eval-k*:

assumes *compat*: *makes-compatible* (cms₁, mem₁) (cms₂, mem₂) *mems*
assumes *modes-eq*: map snd cms₁ = map snd cms₂
assumes *sound-modes*: *sound-mode-use* (cms₁, mem₁) *sound-mode-use* (cms₂,
*mem*₂)
assumes *eval*: (cms₁, mem₁) \rightarrow_k (cms₁', mem₁')
shows $\exists cms_2' mem_2' mems'. \text{sound-mode-use } (cms_1', mem_1') \wedge$
 $\text{sound-mode-use } (cms_2', mem_2') \wedge$
 $\text{map snd } cms_1' = \text{map snd } cms_2' \wedge$
 $(cms_2, mem_2) \rightarrow_k (cms_2', mem_2') \wedge$
 $\text{makes-compatible } (cms_1', mem_1') (cms_2', mem_2') \text{ mems}'$

proof –

from *eval* **show** ?thesis

proof (*induct k arbitrary*: cms₁' mem₁')
case 0

hence cms₁' = cms₁ \wedge mem₁' = mem₁

by (metis *prod.inject meval-k.simps(1)*)

thus ?case

by (metis *compat meval-k.simps(1) modes-eq sound-modes*)

next

case (*Suc k*)

then obtain $cms_1'' mem_1''$ **where** $eval''$:
 $(cms_1, mem_1) \rightarrow_k (cms_1'', mem_1'') \wedge (cms_1'', mem_1'') \rightarrow (cms_1', mem_1')$
by $(metis meval-k.simps(2) prod-cases3 snd-conv)$
hence $(cms_1'', mem_1'') \rightarrow (cms_1', mem_1')$..
moreover
from $eval''$ **obtain** $cms_2'' mem_2'' mems''$ **where** IH :
 $sound-mode-use (cms_1'', mem_1'') \wedge$
 $sound-mode-use (cms_2'', mem_2'') \wedge$
 $map\ snd\ cms_1'' = map\ snd\ cms_2'' \wedge$
 $(cms_2, mem_2) \rightarrow_k (cms_2'', mem_2'') \wedge$
 $makes-compatible (cms_1'', mem_1'') (cms_2'', mem_2'') mems''$
using Suc
by $metis$
ultimately obtain $cms_2' mem_2' mems'$ **where**
 $map\ snd\ cms_1' = map\ snd\ cms_2' \wedge$
 $(cms_2'', mem_2'') \rightarrow (cms_2', mem_2') \wedge$
 $makes-compatible (cms_1', mem_1') (cms_2', mem_2') mems'$
using $makes-compatible-invariant$
by $blast$
thus $?case$
by $(metis IH eval'' meval-k.simps(2) sound-modes-invariant)$

qed

qed

lemma $differing-vars-initially-empty$:

$i < n \implies x \notin differing-vars-lists\ mem_1\ mem_2\ (zip\ (replicate\ n\ mem_1)\ (replicate\ n\ mem_2))\ i$
unfolding $differing-vars-lists-def\ differing-vars-def$
by $auto$

lemma $compatible-refl$:

assumes $coms-secure$: $list-all\ com-sifum-secure\ cmds$
assumes $low-eq$: $mem_1 =^l mem_2$
shows $makes-compatible (add-initial-modes\ cmds, mem_1)$
 $(add-initial-modes\ cmds, mem_2)$
 $(replicate\ (length\ cmds)\ (mem_1, mem_2))$

proof –

let $?n = length\ cmds$
let $?mems = replicate\ ?n\ (mem_1, mem_2)$ **and**
 $?mdss = replicate\ ?n\ mds_s$
let $?X = differing-vars-lists\ mem_1\ mem_2\ ?mems$
have $diff-empty$: $\forall\ i < ?n. ?X\ i = \{\}$
by $(metis\ differing-vars-initially-empty\ ex-in-conv\ min.idem\ zip-replicate)$

show $?thesis$

unfolding $add-initial-modes-def$

proof

show $length\ (zip\ cmds\ ?mdss) = length\ (zip\ cmds\ ?mdss) \wedge length\ (zip\ cmds\ ?mdss) = length\ ?mems$

```

    by auto
  next
  fix i σ
  let ?mems1i = fst (?mems ! i) and ?mems2i = snd (?mems ! i)
  assume i: i < length (zip cmds ?mdss)
  with coms-secure have com-sifum-secure (cmds ! i)
    using coms-secure
    by (metis length-map length-replicate list-all-length map-snd-zip)
  with i have  $\bigwedge mem_1 mem_2. mem_1 =_{mds_s^l} mem_2 \implies$ 
    (zip cmds (replicate ?n mdss) ! i, mem1)  $\approx$  (zip cmds (replicate ?n mdss) ! i,
  mem2)
    using com-sifum-secure-def low-indistinguishable-def
    by auto

  moreover
  from i have ?mems1i = mem1 ?mems2i = mem2
    by auto
  with low-eq have ?mems1i [↦ σ] =mdssl ?mems2i [↦ σ]
    by (auto simp: subst-def mdss-def low-mds-eq-def low-eq-def, case-tac σ x,
  auto)

  ultimately show (zip cmds ?mdss ! i, ?mems1i [↦ σ])  $\approx$  (zip cmds ?mdss !
  i, ?mems2i [↦ σ])
    by simp
  next
  fix i x
  assume i < length (zip cmds ?mdss)
  with diff-empty show x  $\notin$  ?X i by auto
  next
  show (length (zip cmds ?mdss) = 0  $\wedge$  mem1 =l mem2)  $\vee$  ( $\forall x. \exists i < length$ 
  (zip cmds ?mdss). x  $\notin$  ?X i)
    using diff-empty
    by (metis bot-less bot-nat-def empty-iff length-zip low-eq min-0L)
  qed
qed

```

theorem *sifum-compositionality*:

```

  assumes com-secure: list-all com-sifum-secure cmds
  assumes no-assms: no-assumptions-on-termination cmds
  assumes sound-modes:  $\forall mem. sound-mode-use (add-initial-modes cmds, mem)$ 
  shows prog-sifum-secure cmds
  unfolding prog-sifum-secure-def
  using assms
  proof (clarify)
    fix mem1 mem2 :: 'Var  $\Rightarrow$  'Val
    fix k cms1' mem1'
    let ?n = length cmds
    let ?mems = zip (replicate ?n mem1) (replicate ?n mem2)
    assume low-eq: mem1 =l mem2

```

with *com-secure* **have** *compat*:
makes-compatible (*add-initial-modes* *cmds*, *mem₁*) (*add-initial-modes* *cmds*,
mem₂) ?*mems*
by (*metis compatible-refl fst-conv length-replicate map-replicate snd-conv zip-eq-conv*)

also assume *eval*: (*add-initial-modes* *cmds*, *mem₁*) \rightarrow_k (*cms₁'*, *mem₁'*)

ultimately obtain *cms₂' mem₂' mems'*
where *p*: *map snd cms₁' = map snd cms₂'* \wedge
(*add-initial-modes* *cmds*, *mem₂*) \rightarrow_k (*cms₂'*, *mem₂'*) \wedge
makes-compatible (*cms₁'*, *mem₁'*) (*cms₂'*, *mem₂'*) *mems'*
using *sound-modes makes-compatible-eval-k*
by *blast*

thus \exists *cms₂' mem₂'*. (*add-initial-modes* *cmds*, *mem₂*) \rightarrow_k (*cms₂'*, *mem₂'*) \wedge
map snd cms₁' = map snd cms₂' \wedge
length cms₂' = length cms₁' \wedge
 $(\forall x. \text{dma } x = \text{Low} \wedge (\forall i < \text{length } \text{cms}_1'. x \notin \text{snd } (\text{cms}_1' ! i))$
AsmNoRead)
 $\longrightarrow \text{mem}_1' x = \text{mem}_2' x$
using *p compat-low-eq*
by (*metis length-map*)
qed
end
end

4 Language for Instantiating the SIFUM-Security Property

theory *Language*
imports *Main Preliminaries*
begin

4.1 Syntax

datatype *'var ModeUpd = Acq 'var Mode* (**infix** $+=_m$ 75)
| *Rel 'var Mode* (**infix** $-=_m$ 75)

datatype (*'var, 'aexp, 'bexp*) *Stmt = Assign 'var 'aexp* (**infix** \leftarrow 130)
| *Skip*
| *ModeDecl ('var, 'aexp, 'bexp) Stmt 'var ModeUpd* (**-@[** $-$] $[0, 0]$ 150)
| *Seq ('var, 'aexp, 'bexp) Stmt ('var, 'aexp, 'bexp) Stmt* (**infixr** $::$ 150)
| *If 'bexp ('var, 'aexp, 'bexp) Stmt ('var, 'aexp, 'bexp) Stmt*
| *While 'bexp ('var, 'aexp, 'bexp) Stmt*
| *Stop*

type-synonym (*'var, 'aexp, 'bexp*) *EvalCxt = ('var, 'aexp, 'bexp) Stmt list*

locale *sifum-lang =*

fixes *eval_A :: ('Var, 'Val) Mem ⇒ 'AExp ⇒ 'Val*

fixes *eval_B :: ('Var, 'Val) Mem ⇒ 'BExp ⇒ bool*

fixes *aexp-vars :: 'AExp ⇒ 'Var set*

fixes *bexp-vars :: 'BExp ⇒ 'Var set*

fixes *dma :: 'Var ⇒ Sec*

assumes *Var-finite : finite {(x :: 'Var). True}*

assumes *eval-vars-det_A : [[∀ x ∈ aexp-vars e. mem₁ x = mem₂ x] ⇒ eval_A mem₁ e = eval_A mem₂ e*

assumes *eval-vars-det_B : [[∀ x ∈ bexp-vars b. mem₁ x = mem₂ x] ⇒ eval_B mem₁ b = eval_B mem₂ b*

context *sifum-lang*

begin

notation (*latex output*)

Seq (-; - 60)

notation (*Rule output*)

Seq (- ; - 60)

notation (*Rule output*)

If (if - then - else - fi 50)

notation (*Rule output*)

While (while - do - done)

abbreviation *conf_w-abv :: ('Var, 'AExp, 'BExp) Stmt ⇒*

'Var Mds ⇒ ('Var, 'Val) Mem ⇒ (-,-) LocalConf

(⟨-, -, -⟩_w [0, 120, 120] 100)

where

⟨ c, mds, mem ⟩_w ≡ ((c, mds), mem)

4.2 Semantics

primrec *update-modes :: 'Var ModeUpd ⇒ 'Var Mds ⇒ 'Var Mds*

where

update-modes (Acq x m) mds = mds (m := insert x (mds m)) |

update-modes (Rel x m) mds = mds (m := {y. y ∈ mds m ∧ y ≠ x})

fun *updated-var :: 'Var ModeUpd ⇒ 'Var*

where

updated-var (Acq x -) = x |

updated-var (Rel x -) = x

```

fun updated-mode :: 'Var ModeUpd  $\Rightarrow$  Mode
  where
    updated-mode (Acq - m) = m |
    updated-mode (Rel - m) = m

inductive-set evalw-simple :: (('Var, 'AExp, 'BExp) Stmt  $\times$  ('Var, 'Val) Mem)
rel
and evalw-simple-abv :: (('Var, 'AExp, 'BExp) Stmt  $\times$  ('Var, 'Val) Mem)  $\Rightarrow$ 
('Var, 'AExp, 'BExp) Stmt  $\times$  ('Var, 'Val) Mem  $\Rightarrow$  bool
  (infixr  $\rightsquigarrow_s$  60)
  where
    c  $\rightsquigarrow_s$  c'  $\equiv$  (c, c')  $\in$  evalw-simple |
    assign: ((x  $\leftarrow$  e, mem), (Stop, mem (x := evalA mem e)))  $\in$  evalw-simple |
    skip: ((Skip, mem), (Stop, mem))  $\in$  evalw-simple |
    seq-stop: ((Seq Stop c, mem), (c, mem))  $\in$  evalw-simple |
    if-true:  $\llbracket$  evalB mem b  $\rrbracket \Rightarrow$  ((If b t e, mem), (t, mem))  $\in$  evalw-simple |
    if-false:  $\llbracket$   $\neg$  evalB mem b  $\rrbracket \Rightarrow$  ((If b t e, mem), (e, mem))  $\in$  evalw-simple |
    while: ((While b c, mem), (If b (c ;; While b c) Stop, mem))  $\in$  evalw-simple

primrec cxt-to-stmt :: ('Var, 'AExp, 'BExp) EvalCxt  $\Rightarrow$  ('Var, 'AExp, 'BExp)
Stmt
 $\Rightarrow$  ('Var, 'AExp, 'BExp) Stmt
  where
    cxt-to-stmt  $\llbracket$  c = c  $\rrbracket$ 
    cxt-to-stmt (c # cs) c' = Seq c' (cxt-to-stmt cs c)

```

```

inductive-set evalw :: (('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf rel
and evalw-abv :: (('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf  $\Rightarrow$ 
('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf  $\Rightarrow$  bool
  (infixr  $\rightsquigarrow_w$  60)
  where
    c  $\rightsquigarrow_w$  c'  $\equiv$  (c, c')  $\in$  evalw |
    unannotated:  $\llbracket$  (c, mem)  $\rightsquigarrow_s$  (c', mem')  $\rrbracket$ 
 $\Rightarrow$  ( $\langle$ cxt-to-stmt E c, mds, mem $\rangle_w$ ,  $\langle$ cxt-to-stmt E c', mds, mem' $\rangle_w$ )  $\in$  evalw |
    seq:  $\llbracket$   $\langle$ c1, mds, mem $\rangle_w$   $\rightsquigarrow_w$   $\langle$ c1', mds', mem' $\rangle_w$   $\rrbracket \Rightarrow$  ( $\langle$ (c1 ;; c2), mds, mem $\rangle_w$ ,
 $\langle$ (c1' ;; c2), mds', mem' $\rangle_w$ )  $\in$  evalw |
    decl:  $\llbracket$   $\langle$ c, update-modes mu mds, mem $\rangle_w$   $\rightsquigarrow_w$   $\langle$ c', mds', mem' $\rangle_w$   $\rrbracket \Rightarrow$ 
( $\langle$ cxt-to-stmt E (ModeDecl c mu), mds, mem $\rangle_w$ ,  $\langle$ cxt-to-stmt E c', mds',
mem' $\rangle_w$ )  $\in$  evalw

```

4.3 Semantic Properties

The following lemmas simplify working with evaluation contexts in the soundness proofs for the type system(s).

```

inductive-cases eval-elim: (((c, mds), mem), ((c', mds'), mem'))  $\in$  evalw
inductive-cases stop-no-eval' [elim]: ((Stop, mem), (c', mem'))  $\in$  evalw-simple

```

inductive-cases *assign-elim'* [elim]: $((x \leftarrow e, mem), (c', mem')) \in eval_w\text{-simple}$
inductive-cases *skip-elim'* [elim]: $(Skip, mem) \rightsquigarrow_s (c', mem')$

lemma *cxt-inv*:

$\llbracket cxt\text{-to}\text{-stmt } E \ c = c' ; \bigwedge p \ q. \ c' \neq Seq \ p \ q \rrbracket \implies E = [] \wedge c' = c$
by (*metis cxt-to-stmt.simps(1) cxt-to-stmt.simps(2) neq-Nil-conv*)

lemma *cxt-inv-assign*:

$\llbracket cxt\text{-to}\text{-stmt } E \ c = x \leftarrow e \rrbracket \implies c = x \leftarrow e \wedge E = []$
by (*metis Stmt.simps(11) cxt-inv*)

lemma *cxt-inv-skip*:

$\llbracket cxt\text{-to}\text{-stmt } E \ c = Skip \rrbracket \implies c = Skip \wedge E = []$
by (*metis Stmt.simps(21) cxt-inv*)

lemma *cxt-inv-stop*:

$cxt\text{-to}\text{-stmt } E \ c = Stop \implies c = Stop \wedge E = []$
by (*metis Stmt.simps(40) cxt-inv*)

lemma *cxt-inv-if*:

$cxt\text{-to}\text{-stmt } E \ c = If \ e \ p \ q \implies c = If \ e \ p \ q \wedge E = []$
by (*metis Stmt.simps(37) cxt-inv*)

lemma *cxt-inv-while*:

$cxt\text{-to}\text{-stmt } E \ c = While \ e \ p \implies c = While \ e \ p \wedge E = []$
by (*metis Stmt.simps(39) cxt-inv*)

lemma *skip-elim* [elim]:

$\langle Skip, mds, mem \rangle_w \rightsquigarrow_w \langle c', mds', mem' \rangle_w \implies c' = Stop \wedge mds = mds' \wedge mem = mem'$

apply (*erule eval-elim*)

apply (*metis (lifting) cxt-inv-skip cxt-to-stmt.simps(1) skip-elim'*)

apply (*metis Stmt.simps(20)*)

by (*metis Stmt.simps(18) cxt-inv-skip*)

lemma *assign-elim* [elim]:

$\langle x \leftarrow e, mds, mem \rangle_w \rightsquigarrow_w \langle c', mds', mem' \rangle_w \implies c' = Stop \wedge mds = mds' \wedge mem' = mem \ (x := eval_A \ mem \ e)$

apply (*erule eval-elim*)

apply (*rename-tac c c'a E*)

apply (*subgoal-tac c = x \leftarrow e \wedge E = []*)

apply *auto*

apply (*metis cxt-inv-assign*)

apply (*metis cxt-inv-assign*)

apply (*metis Stmt.simps(8) cxt-inv-assign*)

apply (*metis Stmt.simps(8) cxt-inv-assign*)

by (*metis Stmt.simps(8) cxt-inv-assign*)

inductive-cases *if-elim'* [elim!]: $(If \ b \ p \ q, mem) \rightsquigarrow_s (c', mem')$

lemma *if-elim* [elim]:

$\bigwedge P.$
 $\llbracket \langle \text{If } b \text{ } p \text{ } q, \text{ mds}, \text{ mem} \rangle_w \rightsquigarrow_w \langle c', \text{ mds}', \text{ mem}' \rangle_w ;$
 $\llbracket c' = p; \text{ mem}' = \text{ mem} ; \text{ mds}' = \text{ mds} ; \text{ eval}_B \text{ mem } b \rrbracket \implies P ;$
 $\llbracket c' = q; \text{ mem}' = \text{ mem} ; \text{ mds}' = \text{ mds} ; \neg \text{ eval}_B \text{ mem } b \rrbracket \implies P \rrbracket \implies P$
apply (*erule eval-elim*)
apply (*metis (no-types) cxt-inv-if cxt-to-stmt.simps(1) if-elim'*)
apply (*metis Stmt.simps(36)*)
by (*metis Stmt.simps(30) cxt-inv-if*)

inductive-cases *while-elim'* [elim!]: $(\text{While } e \text{ } c, \text{ mem}) \rightsquigarrow_s (c', \text{ mem}')$

lemma *while-elim* [elim]:

$\llbracket \langle \text{While } e \text{ } c, \text{ mds}, \text{ mem} \rangle_w \rightsquigarrow_w \langle c', \text{ mds}', \text{ mem}' \rangle_w \rrbracket \implies c' = \text{If } e \text{ } (c ;; \text{While } e \text{ } c) \text{ Stop} \wedge \text{ mds}' = \text{ mds} \wedge \text{ mem}' = \text{ mem}$
apply (*erule eval-elim*)
apply (*metis (no-types) cxt-inv-while cxt-to-stmt.simps(1) while-elim'*)
apply (*metis Stmt.simps(38)*)
by (*metis (lifting) Stmt.simps(33) cxt-inv-while*)

inductive-cases *upd-elim'* [elim]: $(c@[upd], \text{ mem}) \rightsquigarrow_s (c', \text{ mem}')$

lemma *upd-elim* [elim]:

$\langle c@[upd], \text{ mds}, \text{ mem} \rangle_w \rightsquigarrow_w \langle c', \text{ mds}', \text{ mem}' \rangle_w \implies \langle c, \text{ update-modes } \text{ upd } \text{ mds}, \text{ mem} \rangle_w \rightsquigarrow_w \langle c', \text{ mds}', \text{ mem}' \rangle_w$
apply (*erule eval-elim*)
apply (*metis (lifting) Stmt.simps(28) cxt-inv upd-elim'*)
apply (*metis Stmt.simps(29)*)
by (*metis (lifting) Stmt.simps(2) Stmt.simps(29) cxt-inv cxt-to-stmt.simps(1)*)

lemma *cxt-seq-elim* [elim]:

$c_1 ;; c_2 = \text{cxt-to-stmt } E \text{ } c \implies (E = [] \wedge c = c_1 ;; c_2) \vee (\exists c' \text{ cs. } E = c' \# \text{ cs} \wedge c = c_1 \wedge c_2 = \text{cxt-to-stmt } \text{ cs } c')$
apply (*cases E*)
apply (*metis cxt-to-stmt.simps(1)*)
by (*metis Stmt.simps(3) cxt-to-stmt.simps(2)*)

inductive-cases *seq-elim'* [elim]: $(c_1 ;; c_2, \text{ mem}) \rightsquigarrow_s (c', \text{ mem}')$

lemma *stop-no-eval*: $\neg (\langle \text{Stop}, \text{ mds}, \text{ mem} \rangle_w \rightsquigarrow_w \langle c', \text{ mds}', \text{ mem}' \rangle_w)$

apply *auto*
apply (*erule eval-elim*)
apply (*metis cxt-inv-stop stop-no-eval'*)
apply (*metis Stmt.simps(41)*)
by (*metis Stmt.simps(35) cxt-inv-stop*)

lemma *seq-stop-elim* [elim]:

$\langle \text{Stop} ;; c, \text{ mds}, \text{ mem} \rangle_w \rightsquigarrow_w \langle c', \text{ mds}', \text{ mem}' \rangle_w \implies c' = c \wedge \text{ mds}' = \text{ mds} \wedge \text{ mem}'$


```

= mem
apply (erule eval-elim)
  apply clarify
  apply (metis (no-types) cxt-seq-elim cxt-to-stmt.simps(1) seq-elim' stop-no-eval')
  apply (metis Stmt.inject(3) stop-no-eval)
  by (metis Stmt.distinct(23) Stmt.distinct(29) cxt-seq-elim)

```

```

lemma cxt-stmt-seq:
  c ;; cxt-to-stmt E c' = cxt-to-stmt (c' # E) c
by (metis cxt-to-stmt.simps(2))

```

```

lemma seq-elim [elim]:
   $\llbracket \langle c_1 ;; c_2, mds, mem \rangle_w \rightsquigarrow_w \langle c', mds', mem' \rangle_w ; c_1 \neq Stop \rrbracket \implies$ 
   $(\exists c_1'. \langle c_1, mds, mem \rangle_w \rightsquigarrow_w \langle c_1', mds', mem' \rangle_w \wedge c' = c_1' ;; c_2)$ 
apply (erule eval-elim)
  apply clarify
  apply (drule cxt-seq-elim)
  apply (erule disjE)
  apply (metis seq-elim')
  apply auto
  apply (metis cxt-to-stmt.simps(1) eval_w.unannotated)
  apply (subgoal-tac c_1 = (c@[mu]))
  apply simp
  apply (drule cxt-seq-elim)
  apply (metis Stmt.distinct(23) cxt-stmt-seq cxt-to-stmt.simps(1) eval_w.decl)
by (metis Stmt.distinct(23) cxt-seq-elim)

```

```

lemma stop-cxt: Stop = cxt-to-stmt E c  $\implies$  c = Stop
by (metis Stmt.simps(41) cxt-to-stmt.simps(1) cxt-to-stmt.simps(2) neg-Nil-conv)

```

end

end

5 Type System for Ensuring SIFUM-Security of Commands

```

theory TypeSystem
imports Main Preliminaries Security Language Compositionality
begin

```

5.1 Typing Rules

```

type-synonym Type = Sec

```

```

type-synonym 'Var TyEnv = 'Var  $\rightarrow$  Type

```

locale *sifum-types* =
sifum-lang ev_A ev_B + *sifum-security* dma *Stop* $eval_w$
for $ev_A :: ('Var, 'Val) Mem \Rightarrow 'AExp \Rightarrow 'Val$
and $ev_B :: ('Var, 'Val) Mem \Rightarrow 'BExp \Rightarrow bool$

context *sifum-types*
begin

abbreviation $mm-equiv-abv2 :: (-, -, -) LocalConf \Rightarrow (-, -, -) LocalConf \Rightarrow bool$
(infix ≈ 60)
where $mm-equiv-abv2\ c\ c' \equiv mm-equiv-abv\ c\ c'$

abbreviation $eval-abv2 :: (-, 'Var, 'Val) LocalConf \Rightarrow (-, -, -) LocalConf \Rightarrow bool$
(infixl $\rightsquigarrow 70$)
where
 $x \rightsquigarrow y \equiv (x, y) \in eval_w$

abbreviation $low-indistinguishable-abv :: 'Var\ Mds \Rightarrow ('Var, 'AExp, 'BExp)\ Stmt$
 $\Rightarrow (-, -, -)\ Stmt \Rightarrow bool$
(- \sim_1 - [100, 100] 80)
where
 $c \sim_{mds}\ c' \equiv low-indistinguishable\ mds\ c\ c'$

definition $to-total :: 'Var\ TyEnv \Rightarrow 'Var \Rightarrow Sec$
where $to-total\ \Gamma\ v \equiv if\ v \in dom\ \Gamma\ then\ the\ (\Gamma\ v)\ else\ dma\ v$

definition $max-dom :: Sec\ set \Rightarrow Sec$
where $max-dom\ xs \equiv if\ High \in xs\ then\ High\ else\ Low$

inductive $type-aexpr :: 'Var\ TyEnv \Rightarrow 'AExp \Rightarrow Type \Rightarrow bool$ (- \vdash_a - \in - [120, 120, 120] 1000)
where
 $type-aexpr\ [intro!]: \Gamma\ \vdash_a\ e \in max-dom\ (image\ (\lambda\ x.\ to-total\ \Gamma\ x)\ (aexpr-vars\ e))$

inductive-cases $type-aexpr-elim\ [elim]: \Gamma\ \vdash_a\ e \in t$

inductive $type-bexpr :: 'Var\ TyEnv \Rightarrow 'BExp \Rightarrow Type \Rightarrow bool$ (- \vdash_b - \in - [120, 120, 120] 1000)
where
 $type-bexpr\ [intro!]: \Gamma\ \vdash_b\ e \in max-dom\ (image\ (\lambda\ x.\ to-total\ \Gamma\ x)\ (bexpr-vars\ e))$

inductive-cases $type-bexpr-elim\ [elim]: \Gamma\ \vdash_b\ e \in t$

definition $mds-consistent :: 'Var\ Mds \Rightarrow 'Var\ TyEnv \Rightarrow bool$
where $mds-consistent\ mds\ \Gamma \equiv$
 $dom\ \Gamma = \{(x :: 'Var).\ (dma\ x = Low \wedge x \in mds\ AsmNoRead)\} \vee$

$$(dma\ x = High \wedge x \in mds\ AsmNoWrite)\}$$

fun *add-anno-dom* :: 'Var TyEnv \Rightarrow 'Var ModeUpd \Rightarrow 'Var set

where

add-anno-dom Γ (*Acq* *v* *AsmNoRead*) = (if *dma v* = *Low* then *dom* Γ \cup {*v*} else *dom* Γ) |

add-anno-dom Γ (*Acq* *v* *AsmNoWrite*) = (if *dma v* = *High* then *dom* Γ \cup {*v*} else *dom* Γ) |

add-anno-dom Γ (*Acq* *v* -) = *dom* Γ |

add-anno-dom Γ (*Rel* *v* *AsmNoRead*) = (if *dma v* = *Low* then *dom* Γ - {*v*} else *dom* Γ) |

add-anno-dom Γ (*Rel* *v* *AsmNoWrite*) = (if *dma v* = *High* then *dom* Γ - {*v*} else *dom* Γ) |

add-anno-dom Γ (*Rel* *v* -) = *dom* Γ

definition *add-anno* :: 'Var TyEnv \Rightarrow 'Var ModeUpd \Rightarrow 'Var TyEnv (**infix** \oplus 60)

where

$\Gamma \oplus upd = ((\lambda x. Some (to-total\ \Gamma\ x)) | 'add-anno-dom\ \Gamma\ upd)$

definition *context-le* :: 'Var TyEnv \Rightarrow 'Var TyEnv \Rightarrow bool (**infixr** \sqsubseteq_c 100)

where

$\Gamma \sqsubseteq_c \Gamma' \equiv (dom\ \Gamma = dom\ \Gamma') \wedge (\forall x \in dom\ \Gamma. the\ (\Gamma\ x) \sqsubseteq the\ (\Gamma'\ x))$

inductive *has-type* :: 'Var TyEnv \Rightarrow ('Var, 'AExp, 'BExp) Stmt \Rightarrow 'Var TyEnv \Rightarrow bool

(\vdash - {-} - [120, 120, 120] 1000)

where

stop-type [*intro*]: $\vdash \Gamma \{Stop\} \Gamma$ |

skip-type [*intro*]: $\vdash \Gamma \{Skip\} \Gamma$ |

*assign*₁: $\llbracket x \notin dom\ \Gamma ; \Gamma \vdash_a e \in t ; t \sqsubseteq dma\ x \rrbracket \Longrightarrow \vdash \Gamma \{x \leftarrow e\} \Gamma$ |

*assign*₂: $\llbracket x \in dom\ \Gamma ; \Gamma \vdash_a e \in t \rrbracket \Longrightarrow has-type\ \Gamma (x \leftarrow e) (\Gamma (x := Some\ t))$ |

if-type [*intro*]: $\llbracket \Gamma \vdash_b e \in High \longrightarrow$

$((\forall mds. mds-consistent\ mds\ \Gamma \longrightarrow (low-indistinguishable\ mds\ c_1\ c_2)) \wedge$

$(\forall x \in dom\ \Gamma'. \Gamma' x = Some\ High))$

$;$ $\vdash \Gamma \{c_1\} \Gamma'$

$;$ $\vdash \Gamma \{c_2\} \Gamma'$ $\rrbracket \Longrightarrow$

$\vdash \Gamma \{If\ e\ c_1\ c_2\} \Gamma'$ |

while-type [*intro*]: $\llbracket \Gamma \vdash_b e \in Low ; \vdash \Gamma \{c\} \Gamma \rrbracket \Longrightarrow \vdash \Gamma \{While\ e\ c\} \Gamma$ |

anno-type [*intro*]: $\llbracket \Gamma' = \Gamma \oplus upd ; \vdash \Gamma' \{c\} \Gamma'' ; c \neq Stop ;$

$\forall x. to-total\ \Gamma\ x \sqsubseteq to-total\ \Gamma'\ x \rrbracket \Longrightarrow \vdash \Gamma \{c@[upd]\} \Gamma''$ |

seq-type [*intro*]: $\llbracket \vdash \Gamma \{c_1\} \Gamma' ; \vdash \Gamma' \{c_2\} \Gamma'' \rrbracket \Longrightarrow \vdash \Gamma \{c_1 ;; c_2\} \Gamma''$ |

sub: $\llbracket \vdash \Gamma_1 \{c\} \Gamma_1' ; \Gamma_2 \sqsubseteq_c \Gamma_1 ; \Gamma_1' \sqsubseteq_c \Gamma_2' \rrbracket \Longrightarrow \vdash \Gamma_2 \{c\} \Gamma_2'$

5.2 Typing Soundness

The following predicate is needed to exclude some pathological cases, that abuse the *Stop* command which is not allowed to occur in actual programs.

fun *has-annotated-stop* :: ('Var, 'AExp, 'BExp) Stmt \Rightarrow bool

where

has-annotated-stop ($c@[-]$) = (if $c = \text{Stop}$ then True else *has-annotated-stop* c) |
has-annotated-stop ($\text{Seq } p \ q$) = (*has-annotated-stop* $p \vee$ *has-annotated-stop* q) |
has-annotated-stop ($\text{If } - \ p \ q$) = (*has-annotated-stop* $p \vee$ *has-annotated-stop* q) |
has-annotated-stop ($\text{While } - \ p$) = *has-annotated-stop* p |
has-annotated-stop $- = \text{False}$

inductive-cases *has-type-elim*: $\vdash \Gamma \{ c \} \Gamma'$
inductive-cases *has-type-stop-elim*: $\vdash \Gamma \{ \text{Stop} \} \Gamma'$

definition *tyenv-eq* :: $'Var \ TyEnv \Rightarrow ('Var, 'Val) Mem \Rightarrow ('Var, 'Val) Mem \Rightarrow$
 $bool$
 (infix =₁ 60)
 where $mem_1 =_{\Gamma} mem_2 \equiv \forall x. (\text{to-total } \Gamma \ x = \text{Low} \longrightarrow mem_1 \ x = mem_2 \ x)$

lemma *tyenv-eq-sym*: $mem_1 =_{\Gamma} mem_2 \Longrightarrow mem_2 =_{\Gamma} mem_1$
 by (*auto simp: tyenv-eq-def*)

inductive-set $\mathcal{R}_1 :: 'Var \ TyEnv \Rightarrow (('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf \ rel$
and $\mathcal{R}_1\text{-abv} :: 'Var \ TyEnv \Rightarrow$
 $(('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf \Rightarrow$
 $(('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf \Rightarrow$
 $bool \ (- \ \mathcal{R}_1^1 - [120, 120] \ 1000)$
for $\Gamma' :: 'Var \ TyEnv$
where
 $x \ \mathcal{R}_1^1_{\Gamma} \ y \equiv (x, y) \in \mathcal{R}_1 \ \Gamma \mid$
 $intro \ [intro!] : \llbracket \vdash \Gamma \{ c \} \Gamma' ; mds\text{-consistent } mds \ \Gamma ; mem_1 =_{\Gamma} mem_2 \rrbracket \Longrightarrow \langle c,$
 $mds, mem_1 \rangle \ \mathcal{R}_1^1_{\Gamma'} \langle c, mds, mem_2 \rangle$

inductive-set $\mathcal{R}_2 :: 'Var \ TyEnv \Rightarrow (('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf \ rel$
and $\mathcal{R}_2\text{-abv} :: 'Var \ TyEnv \Rightarrow$
 $(('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf \Rightarrow$
 $(('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf \Rightarrow$
 $bool \ (- \ \mathcal{R}_2^1 - [120, 120] \ 1000)$
for $\Gamma' :: 'Var \ TyEnv$
where
 $x \ \mathcal{R}_2^2_{\Gamma} \ y \equiv (x, y) \in \mathcal{R}_2 \ \Gamma \mid$
 $intro \ [intro!] : \llbracket \langle c_1, mds, mem_1 \rangle \approx \langle c_2, mds, mem_2 \rangle ;$
 $\forall x \in dom \ \Gamma'. \ \Gamma' \ x = \text{Some } High ;$
 $\vdash \Gamma_1 \{ c_1 \} \Gamma' ; \vdash \Gamma_2 \{ c_2 \} \Gamma' ;$
 $mds\text{-consistent } mds \ \Gamma_1 ; mds\text{-consistent } mds \ \Gamma_2 \rrbracket \Longrightarrow$
 $\langle c_1, mds, mem_1 \rangle \ \mathcal{R}_2^2_{\Gamma'} \langle c_2, mds, mem_2 \rangle$

inductive $\mathcal{R}_3\text{-aux} :: 'Var \ TyEnv \Rightarrow (('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf \Rightarrow$
 $(('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf \Rightarrow$

bool (- \mathcal{R}^3_1 - [120, 120] 1000)

and $\mathcal{R}_3 :: 'Var\ TyEnv \Rightarrow (('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf\ rel$
where
 $\mathcal{R}_3\ \Gamma' \equiv \{(lc_1, lc_2). \mathcal{R}_3\text{-aux}\ \Gamma'\ lc_1\ lc_2\} \mid$
 $intro_1\ [intro] : \llbracket \langle c_1, mds, mem_1 \rangle \mathcal{R}^1_\Gamma \langle c_2, mds, mem_2 \rangle; \vdash \Gamma \{c\} \Gamma' \rrbracket \Longrightarrow$
 $\langle Seq\ c_1\ c, mds, mem_1 \rangle \mathcal{R}^3_{\Gamma'} \langle Seq\ c_2\ c, mds, mem_2 \rangle \mid$
 $intro_2\ [intro] : \llbracket \langle c_1, mds, mem_1 \rangle \mathcal{R}^2_\Gamma \langle c_2, mds, mem_2 \rangle; \vdash \Gamma \{c\} \Gamma' \rrbracket \Longrightarrow$
 $\langle Seq\ c_1\ c, mds, mem_1 \rangle \mathcal{R}^3_{\Gamma'} \langle Seq\ c_2\ c, mds, mem_2 \rangle \mid$
 $intro_3\ [intro] : \llbracket \langle c_1, mds, mem_1 \rangle \mathcal{R}^3_\Gamma \langle c_2, mds, mem_2 \rangle; \vdash \Gamma \{c\} \Gamma' \rrbracket \Longrightarrow$
 $\langle Seq\ c_1\ c, mds, mem_1 \rangle \mathcal{R}^3_{\Gamma'} \langle Seq\ c_2\ c, mds, mem_2 \rangle$

definition *weak-bisim* $:: (('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf\ rel \Rightarrow$
 $(('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf\ rel \Rightarrow \text{bool}$
where *weak-bisim* $\mathcal{T}_1\ \mathcal{T} \equiv \forall\ c_1\ c_2\ mds\ mem_1\ mem_2\ c_1'\ mds'\ mem_1'.$
 $((\langle c_1, mds, mem_1 \rangle, \langle c_2, mds, mem_2 \rangle) \in \mathcal{T}_1 \wedge$
 $(\langle c_1, mds, mem_1 \rangle \rightsquigarrow \langle c_1', mds', mem_1' \rangle)) \longrightarrow$
 $(\exists\ c_2'\ mem_2'. \langle c_2, mds, mem_2 \rangle \rightsquigarrow \langle c_2', mds', mem_2' \rangle \wedge$
 $(\langle c_1', mds', mem_1' \rangle, \langle c_2', mds', mem_2' \rangle) \in \mathcal{T})$

inductive-set $\mathcal{R} :: 'Var\ TyEnv \Rightarrow$
 $(('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf\ rel$
and $\mathcal{R}\text{-abv} :: 'Var\ TyEnv \Rightarrow$
 $(('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf \Rightarrow$
 $(('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf \Rightarrow$
 bool (- \mathcal{R}^u_1 - [120, 120] 1000)
for $\Gamma :: 'Var\ TyEnv$
where
 $x\ \mathcal{R}^u_\Gamma\ y \equiv (x, y) \in \mathcal{R}\ \Gamma \mid$
 $intro_1: lc\ \mathcal{R}^1_\Gamma\ lc' \Longrightarrow (lc, lc') \in \mathcal{R}\ \Gamma \mid$
 $intro_2: lc\ \mathcal{R}^2_\Gamma\ lc' \Longrightarrow (lc, lc') \in \mathcal{R}\ \Gamma \mid$
 $intro_3: lc\ \mathcal{R}^3_\Gamma\ lc' \Longrightarrow (lc, lc') \in \mathcal{R}\ \Gamma$

inductive-cases $\mathcal{R}_1\text{-elim}\ [elim]: \langle c_1, mds, mem_1 \rangle \mathcal{R}^1_\Gamma \langle c_2, mds, mem_2 \rangle$
inductive-cases $\mathcal{R}_2\text{-elim}\ [elim]: \langle c_1, mds, mem_1 \rangle \mathcal{R}^2_\Gamma \langle c_2, mds, mem_2 \rangle$
inductive-cases $\mathcal{R}_3\text{-elim}\ [elim]: \langle c_1, mds, mem_1 \rangle \mathcal{R}^3_\Gamma \langle c_2, mds, mem_2 \rangle$

inductive-cases $\mathcal{R}\text{-elim}\ [elim]: (\langle c_1, mds, mem_1 \rangle, \langle c_2, mds, mem_2 \rangle) \in \mathcal{R}\ \Gamma$
inductive-cases $\mathcal{R}\text{-elim}' : (\langle c_1, mds, mem_1 \rangle, \langle c_2, mds_2, mem_2 \rangle) \in \mathcal{R}\ \Gamma$
inductive-cases $\mathcal{R}_1\text{-elim}' : \langle c_1, mds, mem_1 \rangle \mathcal{R}^1_\Gamma \langle c_2, mds_2, mem_2 \rangle$
inductive-cases $\mathcal{R}_2\text{-elim}' : \langle c_1, mds, mem_1 \rangle \mathcal{R}^2_\Gamma \langle c_2, mds_2, mem_2 \rangle$
inductive-cases $\mathcal{R}_3\text{-elim}' : \langle c_1, mds, mem_1 \rangle \mathcal{R}^3_\Gamma \langle c_2, mds_2, mem_2 \rangle$

lemma $\mathcal{R}_1\text{-sym}: \text{sym}\ (\mathcal{R}_1\ \Gamma)$
unfolding *sym-def*
apply *auto*

by (*metis* (*no-types*) $\mathcal{R}_1.intro$ \mathcal{R}_1-elim' *tyenv-eq-sym*)

lemma $\mathcal{R}_2-sym: sym (\mathcal{R}_2 \Gamma)$
unfolding *sym-def*
apply *clarify*
by (*metis* (*no-types*) $\mathcal{R}_2.intro$ \mathcal{R}_2-elim' *mm-equiv-sym*)

lemma $\mathcal{R}_3-sym: sym (\mathcal{R}_3 \Gamma)$
unfolding *sym-def*
proof (*clarify*)
fix $c_1 mds mem_1 c_2 mds' mem_2$
assume *asm*: $\langle c_1, mds, mem_1 \rangle \mathcal{R}^3_\Gamma \langle c_2, mds', mem_2 \rangle$
hence [*simp*]: $mds' = mds$
using \mathcal{R}_3-elim' **by** *blast*
from *asm* **show** $\langle c_2, mds', mem_2 \rangle \mathcal{R}^3_\Gamma \langle c_1, mds, mem_1 \rangle$
apply *auto*
apply (*induct rule*: $\mathcal{R}_3-aux.induct$)
apply (*metis* (*lifting*) \mathcal{R}_1-sym $\mathcal{R}_3-aux.intro_1$ *symD*)
apply (*metis* (*lifting*) \mathcal{R}_2-sym $\mathcal{R}_3-aux.intro_2$ *symD*)
by (*metis* (*lifting*) $\mathcal{R}_3-aux.intro_3$)

qed

lemma $\mathcal{R}-mds$ [*simp*]: $\langle c_1, mds, mem_1 \rangle \mathcal{R}^u_\Gamma \langle c_2, mds', mem_2 \rangle \implies mds = mds'$
apply (*rule* $\mathcal{R}-elim'$)
apply (*auto*)
apply (*metis* \mathcal{R}_1-elim')
apply (*metis* \mathcal{R}_2-elim')
apply (*insert* \mathcal{R}_3-elim')
by *blast*

lemma $\mathcal{R}-sym: sym (\mathcal{R} \Gamma)$
unfolding *sym-def*
proof (*clarify*)
fix $c_1 mds mem_1 c_2 mds_2 mem_2$
assume *asm*: $(\langle c_1, mds, mem_1 \rangle, \langle c_2, mds_2, mem_2 \rangle) \in \mathcal{R} \Gamma$
with $\mathcal{R}-mds$ **have** [*simp*]: $mds_2 = mds$
by *blast*
from *asm* **show** $(\langle c_2, mds_2, mem_2 \rangle, \langle c_1, mds, mem_1 \rangle) \in \mathcal{R} \Gamma$
using $\mathcal{R}.intro_1$ [*of* Γ] **and** $\mathcal{R}.intro_2$ [*of* Γ] **and** $\mathcal{R}.intro_3$ [*of* Γ]
using \mathcal{R}_1-sym [*of* Γ] **and** \mathcal{R}_2-sym [*of* Γ] **and** \mathcal{R}_3-sym [*of* Γ]
apply *simp*
apply (*erule* $\mathcal{R}-elim$)
by (*auto simp*: *sym-def*)

qed

lemma $\mathcal{R}_1-closed-glob-consistent: closed-glob-consistent (\mathcal{R}_1 \Gamma')$
unfolding *closed-glob-consistent-def*

proof (*clarify*)
fix c_1 *mds* mem_1 c_2 mem_2 x Γ'
assume $R1: \langle c_1, mds, mem_1 \rangle \mathcal{R}^1_{\Gamma'} \langle c_2, mds, mem_2 \rangle$
hence [*simp*]: $c_2 = c_1$ **by** *blast*
from $R1$ **obtain** Γ **where** Γ -*props*: $\vdash \Gamma \{ c_1 \} \Gamma' mem_1 =_{\Gamma} mem_2$ *mds-consistent*
mds Γ
by *blast*
hence $\bigwedge v. \langle c_1, mds, mem_1(x := v) \rangle \mathcal{R}^1_{\Gamma'} \langle c_2, mds, mem_2(x := v) \rangle$
by (*auto simp: tyenv-eq-def mds-consistent-def*)
moreover
from Γ -*props* **have** $\bigwedge v_1 v_2. \llbracket dma\ x = High ; x \notin mds\ AsmNoWrite \rrbracket \implies$
 $\langle c_1, mds, mem_1(x := v_1) \rangle \mathcal{R}^1_{\Gamma'} \langle c_2, mds, mem_2(x := v_2) \rangle$
apply (*auto simp: mds-consistent-def tyenv-eq-def*)
by (*metis (lifting, full-types) Sec.simps(2) mem-Collect-eq to-total-def*)
ultimately show
 $(dma\ x = High \wedge x \notin mds\ AsmNoWrite \longrightarrow$
 $(\forall v_1 v_2. \langle c_1, mds, mem_1(x := v_1) \rangle \mathcal{R}^1_{\Gamma'} \langle c_2, mds, mem_2(x := v_2) \rangle))$
 \wedge
 $(dma\ x = Low \wedge x \notin mds\ AsmNoWrite \longrightarrow$
 $(\forall v. \langle c_1, mds, mem_1(x := v) \rangle \mathcal{R}^1_{\Gamma'} \langle c_2, mds, mem_2(x := v) \rangle))$
using *intro1*
by *auto*
qed

lemma \mathcal{R}_2 -*closed-glob-consistent*: *closed-glob-consistent* (\mathcal{R}_2 Γ')

unfolding *closed-glob-consistent-def*

proof (*clarify*)

fix c_1 *mds* mem_1 c_2 mem_2 x Γ'

assume $R2: \langle c_1, mds, mem_1 \rangle \mathcal{R}^2_{\Gamma'} \langle c_2, mds, mem_2 \rangle$

then obtain Γ_1 Γ_2 **where** Γ -*prop*: $\vdash \Gamma_1 \{ c_1 \} \Gamma' \vdash \Gamma_2 \{ c_2 \} \Gamma'$

mds-consistent *mds* Γ_1 *mds-consistent* *mds* Γ_2

by *blast*

from $R2$ **have** *bisim*: $\langle c_1, mds, mem_1 \rangle \approx \langle c_2, mds, mem_2 \rangle$

by *blast*

then obtain \mathcal{R}' **where** \mathcal{R}' -*prop*: $(\langle c_1, mds, mem_1 \rangle, \langle c_2, mds, mem_2 \rangle) \in \mathcal{R}' \wedge$
strong-low-bisim-mm \mathcal{R}'

apply (*rule mm-equiv-elim*)

by (*auto simp: strong-low-bisim-mm-def*)

from \mathcal{R}' -*prop* **have** \mathcal{R}' -*cons*: *closed-glob-consistent* \mathcal{R}'

by (*auto simp: strong-low-bisim-mm-def*)

moreover

from Γ -*prop* **and** \mathcal{R}' -*prop*

have $\bigwedge mem_1 mem_2. (\langle c_1, mds, mem_1 \rangle, \langle c_2, mds, mem_2 \rangle) \in \mathcal{R}' \implies \langle c_1, mds,$
 $mem_1 \rangle \mathcal{R}^2_{\Gamma'} \langle c_2, mds, mem_2 \rangle$

using \mathcal{R}_2 .*intro* [**where** $\Gamma' = \Gamma'$ **and** $\Gamma_1 = \Gamma_1$ **and** $\Gamma_2 = \Gamma_2$]

using *mm-equiv-intro* **and** $R2$

by *blast*

ultimately show

$(dma\ x = High \wedge x \notin mds\ AsmNoWrite \longrightarrow$

$(\forall v_1 v_2. \langle c_1, mds, mem_1(x := v_1) \rangle \mathcal{R}^2_{\Gamma'} \langle c_2, mds, mem_2(x := v_2) \rangle))$
 \wedge
 $(dma\ x = Low \wedge x \notin mds\ AsmNoWrite \longrightarrow$
 $(\forall v. \langle c_1, mds, mem_1(x := v) \rangle \mathcal{R}^2_{\Gamma'} \langle c_2, mds, mem_2(x := v) \rangle))$
using $\mathcal{R}'\text{-prop}$
unfolding $closed\text{-glob}\text{-consistent}\text{-def}$
by $simp$
qed

fun $closed\text{-glob}\text{-helper} :: 'Var\ TyEnv \Rightarrow (('Var, 'AExp, 'BExp)\ Stmt, 'Var, 'Val)$
 $LocalConf \Rightarrow (('Var, 'AExp, 'BExp)\ Stmt, 'Var, 'Val)\ LocalConf \Rightarrow bool$
where
 $closed\text{-glob}\text{-helper}\ \Gamma' \langle c_1, mds, mem_1 \rangle \langle c_2, mds_2, mem_2 \rangle =$
 $(\forall x. ((dma\ x = High \wedge x \notin mds\ AsmNoWrite) \longrightarrow$
 $(\forall v_1 v_2. (\langle c_1, mds, mem_1(x := v_1) \rangle, \langle c_2, mds, mem_2(x := v_2) \rangle) \in$
 $\mathcal{R}_3\ \Gamma')) \wedge$
 $((dma\ x = Low \wedge x \notin mds\ AsmNoWrite) \longrightarrow$
 $(\forall v. (\langle c_1, mds, mem_1(x := v) \rangle, \langle c_2, mds, mem_2(x := v) \rangle) \in \mathcal{R}_3\ \Gamma')))$

lemma $\mathcal{R}_3\text{-closed}\text{-glob}\text{-consistent}$:

assumes $R3: \langle c_1, mds, mem_1 \rangle \mathcal{R}^3_{\Gamma'} \langle c_2, mds, mem_2 \rangle$
shows $\forall x.$
 $(dma\ x = High \wedge x \notin mds\ AsmNoWrite \longrightarrow$
 $(\forall v_1 v_2. (\langle c_1, mds, mem_1(x := v_1) \rangle, \langle c_2, mds, mem_2(x := v_2) \rangle) \in \mathcal{R}_3\ \Gamma'))$
 \wedge
 $(dma\ x = Low \wedge x \notin mds\ AsmNoWrite \longrightarrow (\forall v. (\langle c_1, mds, mem_1(x := v) \rangle,$
 $\langle c_2, mds, mem_2(x := v) \rangle) \in \mathcal{R}_3\ \Gamma'))$
proof –
from $R3$ **have** $closed\text{-glob}\text{-helper}\ \Gamma' \langle c_1, mds, mem_1 \rangle \langle c_2, mds, mem_2 \rangle$
proof ($induct$ rule: $\mathcal{R}_3\text{-aux.}induct$)
case ($intro_1\ \Gamma\ c_1\ mds\ mem_1\ c_2\ mem_2\ c\ \Gamma'$)
thus $?case$
using $\mathcal{R}_1\text{-closed}\text{-glob}\text{-consistent}$ [$of\ \Gamma$] **and** $\mathcal{R}_3\text{-aux.}intro_1$
unfolding $closed\text{-glob}\text{-consistent}\text{-def}$
by ($simp, blast$)
next
case ($intro_2\ \Gamma\ c_1\ mds\ mem_1\ c_2\ mem_2\ c\ \Gamma'$)
thus $?case$
using $\mathcal{R}_2\text{-closed}\text{-glob}\text{-consistent}$ [$of\ \Gamma$] **and** $\mathcal{R}_3\text{-aux.}intro_2$
unfolding $closed\text{-glob}\text{-consistent}\text{-def}$
by ($simp, blast$)
next
case $intro_3$
thus $?case$
using $\mathcal{R}_3\text{-aux.}intro_3$
by ($simp, blast$)
qed
thus $?thesis$ **by** $simp$

qed

lemma \mathcal{R} -closed-glob-consistent: closed-glob-consistent ($\mathcal{R} \Gamma'$)

unfolding closed-glob-consistent-def

proof (clarify, erule \mathcal{R} -elim, simp-all)

fix c_1 mds mem_1 c_2 mem_2 x

assume $R1$: $\langle c_1, mds, mem_1 \rangle \mathcal{R}^1_{\Gamma'} \langle c_2, mds, mem_2 \rangle$

with \mathcal{R}_1 -closed-glob-consistent **show**

($dma \ x = High \wedge x \notin mds \ AsmNoWrite \longrightarrow$

$(\forall v_1 v_2. (\langle c_1, mds, mem_1(x := v_1) \rangle, \langle c_2, mds, mem_2(x := v_2) \rangle) \in \mathcal{R} \Gamma')) \wedge$

($dma \ x = Low \wedge x \notin mds \ AsmNoWrite \longrightarrow$

$(\forall v. (\langle c_1, mds, mem_1(x := v) \rangle, \langle c_2, mds, mem_2(x := v) \rangle) \in \mathcal{R} \Gamma'))$)

unfolding closed-glob-consistent-def

using $intro_1$

apply clarify

by $metis$

next

fix c_1 mds mem_1 c_2 mem_2 x

assume $R2$: $\langle c_1, mds, mem_1 \rangle \mathcal{R}^2_{\Gamma'} \langle c_2, mds, mem_2 \rangle$

with \mathcal{R}_2 -closed-glob-consistent **show**

($dma \ x = High \wedge x \notin mds \ AsmNoWrite \longrightarrow$

$(\forall v_1 v_2. (\langle c_1, mds, mem_1(x := v_1) \rangle, \langle c_2, mds, mem_2(x := v_2) \rangle) \in \mathcal{R} \Gamma'))$

\wedge

($dma \ x = Low \wedge x \notin mds \ AsmNoWrite \longrightarrow$

$(\forall v. (\langle c_1, mds, mem_1(x := v) \rangle, \langle c_2, mds, mem_2(x := v) \rangle) \in \mathcal{R} \Gamma'))$)

unfolding closed-glob-consistent-def

using $intro_2$

apply clarify

by $metis$

next

fix c_1 mds mem_1 c_2 mem_2 x Γ'

assume $R3$: $\langle c_1, mds, mem_1 \rangle \mathcal{R}^3_{\Gamma'} \langle c_2, mds, mem_2 \rangle$

thus

($dma \ x = High \wedge x \notin mds \ AsmNoWrite \longrightarrow$

$(\forall v_1 v_2. (\langle c_1, mds, mem_1(x := v_1) \rangle, \langle c_2, mds, mem_2(x := v_2) \rangle) \in \mathcal{R} \Gamma'))$

\wedge

($dma \ x = Low \wedge x \notin mds \ AsmNoWrite \longrightarrow$

$(\forall v. (\langle c_1, mds, mem_1(x := v) \rangle, \langle c_2, mds, mem_2(x := v) \rangle) \in \mathcal{R} \Gamma'))$)

using \mathcal{R}_3 -closed-glob-consistent

apply auto

apply ($metis \ \mathcal{R}.intro_3$)

by ($metis \ (lifting) \ \mathcal{R}.intro_3$)

qed

lemma type-low-vars-low:

assumes $typed$: $\Gamma \vdash_a e \in Low$

assumes *mds-cons*: *mds-consistent mds* Γ
assumes *x-in-vars*: $x \in \text{aexp-vars } e$
shows *to-total* $\Gamma \ x = \text{Low}$
using *assms*
by (*metis (full-types) Sec.exhaust imageI max-dom-def type-aexpr-elim*)

lemma *type-low-vars-low-b*:
assumes *typed* : $\Gamma \vdash_b e \in \text{Low}$
assumes *mds-cons*: *mds-consistent mds* Γ
assumes *x-in-vars*: $x \in \text{bexp-vars } e$
shows *to-total* $\Gamma \ x = \text{Low}$
using *assms*
by (*metis (full-types) Sec.exhaust imageI max-dom-def type-bexpr.simps*)

lemma *mode-update-add-anno*:
mds-consistent mds $\Gamma \implies \text{mds-consistent } (\text{update-modes upd mds}) (\Gamma \oplus \text{upd})$
apply (*induct arbitrary*: Γ *rule*: *add-anno-dom.induct*)
by (*auto simp*: *add-anno-def mds-consistent-def*)

lemma *context-le-trans*: $\llbracket \Gamma \sqsubseteq_c \Gamma' ; \Gamma' \sqsubseteq_c \Gamma'' \rrbracket \implies \Gamma \sqsubseteq_c \Gamma''$
apply (*auto simp*: *context-le-def*)
by (*metis domI order-trans option.sel*)

lemma *context-le-refl* [*simp*]: $\Gamma \sqsubseteq_c \Gamma$
by (*metis context-le-def order-refl*)

lemma *stop-cxt* :
 $\llbracket \vdash \Gamma \{ c \} \Gamma' ; c = \text{Stop} \rrbracket \implies \Gamma \sqsubseteq_c \Gamma'$
apply (*induct rule*: *has-type.induct*)
apply *auto*
by (*metis context-le-trans*)

lemma *preservation*:
assumes *typed*: $\vdash \Gamma \{ c \} \Gamma'$
assumes *eval*: $\langle c, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \rangle$
shows $\exists \Gamma'' . (\vdash \Gamma'' \{ c' \} \Gamma') \wedge (\text{mds-consistent mds } \Gamma \longrightarrow \text{mds-consistent mds}' \Gamma'')$
using *typed eval*
proof (*induct arbitrary*: c' *mds rule*: *has-type.induct*)
case (*anno-type* $\Gamma'' \Gamma \text{ upd } c_1 \Gamma'$)
hence $\langle c_1, \text{update-modes upd mds}, \text{mem} \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \rangle$
by (*metis upd-elim*)
with *anno-type*(\exists) **obtain** Γ''' **where**
 $\vdash \Gamma''' \{ c' \} \Gamma' \wedge (\text{mds-consistent } (\text{update-modes upd mds}) \Gamma'' \longrightarrow \text{mds-consistent mds}' \Gamma''')$
by *auto*
moreover

```

have mds-consistent mds  $\Gamma \longrightarrow \text{mds-consistent (update-modes upd mds)} \Gamma''$ 
  using anno-type
  apply auto
  by (metis mode-update-add-anno)
ultimately show ?case
  by blast
next
  case stop-type
  with stop-no-eval show ?case ..
next
  case skip-type
  hence  $c' = \text{Stop} \wedge \text{mds}' = \text{mds}$ 
    by (metis skip-elim)
  thus ?case
    by (metis stop-type)
next
  case (assign1 x  $\Gamma$  e t c' mds)
  hence  $c' = \text{Stop} \wedge \text{mds}' = \text{mds}$ 
    by (metis assign-elim)
  thus ?case
    by (metis stop-type)
next
  case (assign2 x  $\Gamma$  e t c' mds)
  hence  $c' = \text{Stop} \wedge \text{mds}' = \text{mds}$ 
    by (metis assign-elim)
  thus ?case
    apply (rule-tac  $x = \Gamma (x \mapsto t)$  in exI)
    apply (auto simp: mds-consistent-def)
      apply (metis Sec.exhaust)
      apply (metis (lifting, full-types) CollectD domI assign2(1))
      apply (metis (lifting, full-types) CollectD domI assign2(1))
      apply (metis (lifting) CollectE domI assign2(1))
      apply (metis (lifting, full-types) domD mem-Collect-eq)
    by (metis (lifting, full-types) domD mem-Collect-eq)
next
  case (if-type  $\Gamma$  e th el  $\Gamma'$  c' mds)
  thus ?case
    apply (rule-tac  $x = \Gamma$  in exI)
    by force
next
  case (while-type  $\Gamma$  e c c' mds)
  hence [simp]:  $\text{mds}' = \text{mds} \wedge c' = \text{If } e (c ;; \text{While } e c) \text{ Stop}$ 
    by (metis while-elim)
  thus ?case
    apply (rule-tac  $x = \Gamma$  in exI)
    apply auto
    by (metis Sec.simps(2) has-type.while-type if-type while-type(1) while-type(2))
seq-type stop-type type-bexpr-elim
next

```

case (*seq-type* Γ c_1 Γ_1 c_2 Γ_2 c' *mds*)
thus *?case*
proof (*cases* $c_1 = \text{Stop}$)
 assume [*simp*]: $c_1 = \text{Stop}$
 with *seq-type* **have** [*simp*]: $\text{mds}' = \text{mds} \wedge c' = c_2$
 by (*metis seq-stop-elim*)
 thus *?case*
 apply *auto*
 by (*metis (lifting) <c₁ = Stop> context-le-refl seq-type(1) seq-type(3) stop-cxt*
sub)
next
 assume $c_1 \neq \text{Stop}$
 then obtain c_1' **where** $\langle c_1, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c_1', \text{mds}', \text{mem}' \rangle \wedge c' = (c_1' ;; c_2)$
 by (*metis seq-elim seq-type.prem*)
 then obtain Γ''' **where** $\vdash \Gamma''' \{c_1'\} \Gamma_1 \wedge$
 $(\text{mds-consistent } \text{mds } \Gamma \longrightarrow \text{mds-consistent } \text{mds}' \Gamma''')$
 using *seq-type(2)*
 by *auto*
 moreover
 from *seq-type* **have** $\vdash \Gamma_1 \{c_2\} \Gamma_2$ **by** *auto*
 moreover
 ultimately show *?case*
 apply (*rule-tac* $x = \Gamma'''$ **in** *exI*)
 by (*metis (lifting) <c₁, mds, mem> <c₁', mds', mem'> <c' = c₁' ;; c₂>*
has-type.seq-type)
 qed
next
 case (*sub* Γ_1 c Γ_1' Γ_2 Γ_2' c' *mds*)
 then obtain Γ'' **where** $\vdash \Gamma'' \{c'\} \Gamma_1' \wedge$
 $(\text{mds-consistent } \text{mds } \Gamma_1 \longrightarrow \text{mds-consistent } \text{mds}' \Gamma'')$
 by *auto*
 thus *?case*
 apply (*rule-tac* $x = \Gamma''$ **in** *exI*)
 apply (*rule conjI*)
 apply (*metis (lifting) has-type.sub sub(4) stop-cxt stop-type*)
 apply (*simp add: mds-consistent-def*)
 by (*metis context-le-def sub.hyps(3)*)
qed

lemma $\mathcal{R}_1\text{-mem-eq}$: $\langle c_1, \text{mds}, \text{mem}_1 \rangle \mathcal{R}_{\Gamma'}^1 \langle c_2, \text{mds}, \text{mem}_2 \rangle \implies \text{mem}_1 =_{\text{mds}}^l \text{mem}_2$
 apply (*rule* $\mathcal{R}_1\text{-elim}$)
 apply (*auto simp: tyenv-eq-def mds-consistent-def to-total-def*)
 by (*metis (lifting) Sec.simps(1) low-mds-eq-def*)

lemma $\mathcal{R}_2\text{-mem-eq}$: $\langle c_1, \text{mds}, \text{mem}_1 \rangle \mathcal{R}_{\Gamma'}^2 \langle c_2, \text{mds}, \text{mem}_2 \rangle \implies \text{mem}_1 =_{\text{mds}}^l \text{mem}_2$
 apply (*rule* $\mathcal{R}_2\text{-elim}$)
 by (*auto simp: mm-equiv-elim strong-low-bisim-mm-def*)

fun *bisim-helper* :: (('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf \Rightarrow
 (('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf \Rightarrow bool
where
bisim-helper $\langle c_1, mds, mem_1 \rangle \langle c_2, mds_2, mem_2 \rangle = mem_1 =_{mds} mem_2$

lemma \mathcal{R}_3 -mem-eq: $\langle c_1, mds, mem_1 \rangle \mathcal{R}_{\Gamma'}^3 \langle c_2, mds, mem_2 \rangle \implies mem_1 =_{mds} mem_2$
apply (*subgoal-tac bisim-helper* $\langle c_1, mds, mem_1 \rangle \langle c_2, mds, mem_2 \rangle$)
apply *simp*
apply (*induct rule: \mathcal{R}_3-aux.induct*)
by (*auto simp: \mathcal{R}_1-mem-eq \mathcal{R}_2-mem-eq*)

lemma \mathcal{R}_2 -bisim-step:

assumes *case2*: $\langle c_1, mds, mem_1 \rangle \mathcal{R}_{\Gamma'}^2 \langle c_2, mds, mem_2 \rangle$
assumes *eval*: $\langle c_1, mds, mem_1 \rangle \rightsquigarrow \langle c_1', mds', mem_1 \wedge \rangle$
shows $\exists c_2' mem_2'. \langle c_2, mds, mem_2 \rangle \rightsquigarrow \langle c_2', mds', mem_2 \wedge \rangle \wedge \langle c_1', mds', mem_1 \wedge \rangle$
 $\mathcal{R}_{\Gamma'}^2 \langle c_2', mds', mem_2 \wedge \rangle$
proof –

from *case2* **have** *aux*: $\langle c_1, mds, mem_1 \rangle \approx \langle c_2, mds, mem_2 \rangle \forall x \in dom \Gamma'. \Gamma' x = Some High$

by (*rule \mathcal{R}_2-elim, auto*)

with *eval* **obtain** $c_2' mem_2'$ **where** c_2' -props:

$\langle c_2, mds, mem_2 \rangle \rightsquigarrow \langle c_2', mds', mem_2 \wedge \rangle$

$\langle c_1', mds', mem_1 \wedge \rangle \approx \langle c_2', mds', mem_2 \wedge \rangle$

using *mm-equiv-strong-low-bisim strong-low-bisim-mm-def*

by *metis*

from *case2* **obtain** $\Gamma_1 \Gamma_2$ **where** $\vdash \Gamma_1 \{ c_1 \} \Gamma' \vdash \Gamma_2 \{ c_2 \} \Gamma'$

mds-consistent mds \Gamma_1 mds-consistent mds \Gamma_2

by (*metis \mathcal{R}_2-elim'*)

with *preservation* **and** c_2' -props **obtain** $\Gamma_1' \Gamma_2'$ **where**

$\vdash \Gamma_1' \{ c_1 \wedge \} \Gamma' mds-consistent mds' \Gamma_1'$

$\vdash \Gamma_2' \{ c_2 \wedge \} \Gamma' mds-consistent mds' \Gamma_2'$

using *eval*

by *metis*

with c_2' -props **show** *?thesis*

using *\mathcal{R}_2.intro aux(2) c_2'-props*

by *blast*

qed

lemma \mathcal{R}_2 -weak-bisim:

weak-bisim (\mathcal{R}_2 \Gamma') (\mathcal{R} \Gamma')

```

unfolding weak-bisim-def
using  $\mathcal{R}.intro_2$ 
apply auto
by (metis  $\mathcal{R}_2$ -bisim-step)

lemma  $\mathcal{R}_2$ -bisim: strong-low-bisim-mm ( $\mathcal{R}_2 \Gamma'$ )
unfolding strong-low-bisim-mm-def
by (auto simp:  $\mathcal{R}_2$ -sym  $\mathcal{R}_2$ -closed-glob-consistent  $\mathcal{R}_2$ -mem-eq  $\mathcal{R}_2$ -bisim-step)

lemma annotated-no-stop:  $\llbracket \neg \text{has-annotated-stop } (c@[upd]) \rrbracket \implies \neg \text{has-annotated-stop } c$ 
apply (cases  $c$ )
by auto

lemma typed-no-annotated-stop:
 $\llbracket \vdash \Gamma \{ c \} \Gamma' \rrbracket \implies \neg \text{has-annotated-stop } c$ 
by (induct rule: has-type.induct, auto)

lemma not-stop-eval:
 $\llbracket c \neq \text{Stop} ; \neg \text{has-annotated-stop } c \rrbracket \implies$ 
 $\forall \text{ mds mem. } \exists c' \text{ mds}' \text{ mem}'. \langle c, \text{ mds}, \text{ mem} \rangle \rightsquigarrow \langle c', \text{ mds}', \text{ mem}' \rangle$ 
proof (induct)
case (Assign  $x \text{ exp}$ )
thus ?case
by (metis cxt-to-stmt.simps(1) evalw-simplep.assign evalwp.unannotated evalwp-evalw-eq)
next
case Skip
thus ?case
by (metis cxt-to-stmt.simps(1) evalw.unannotated evalw-simple.skip)
next
case (ModeDecl  $c \text{ mu}$ )
hence  $\neg \text{has-annotated-stop } c \wedge c \neq \text{Stop}$ 
by (metis has-annotated-stop.simps(1))
with ModeDecl show ?case
apply (clarify, rename-tac  $\text{ mds mem}$ )
apply simp
apply (erule-tac  $x = \text{update-modes } \text{ mu mds}$  in allE)
apply (erule-tac  $x = \text{ mem}$  in allE)
apply (erule exE)+
by (metis cxt-to-stmt.simps(1) evalw.decl)
next
case (Seq  $c_1 \text{ } c_2$ )
thus ?case
proof (cases  $c_1 = \text{Stop}$ )
assume  $c_1 = \text{Stop}$ 
thus ?case
by (metis cxt-to-stmt.simps(1) evalw-simplep.seq-stop evalwp.unannotated)

```

```

evalwp-evalw-eq)
  next
  assume  $c_1 \neq \text{Stop}$ 
  with Seq show ?case
  by (metis evalw.seq has-annotated-stop.simps(2))
qed
next
case (If bexp  $c_1$   $c_2$ )
thus ?case
  apply (clarify, rename-tac mds mem)
  apply (case-tac evB mem bexp)
  apply (metis cxt-to-stmt.simps(1) evalw.unannotated evalw-simple.if-true)
  by (metis cxt-to-stmt.simps(1) evalw.unannotated evalw-simple.if-false)
next
case (While bexp  $c$ )
thus ?case
  by (metis cxt-to-stmt.simps(1) evalw-simplep.while evalwp.unannotated evalwp-evalw-eq)
next
case Stop
thus ?case by blast
qed

```

lemma stop-bisim:

```

assumes bisim:  $\langle \text{Stop}, \text{mds}, \text{mem}_1 \rangle \approx \langle c, \text{mds}, \text{mem}_2 \rangle$ 
assumes typeable:  $\vdash \Gamma \{ c \} \Gamma'$ 
shows  $c = \text{Stop}$ 
proof (rule ccontr)
  let ?lc1 =  $\langle \text{Stop}, \text{mds}, \text{mem}_1 \rangle$  and
  ?lc2 =  $\langle c, \text{mds}, \text{mem}_2 \rangle$ 
  assume  $c \neq \text{Stop}$ 
  from typeable have  $\neg$  has-annotated-stop  $c$ 
  by (metis typed-no-annotated-stop)
  with  $\langle c \neq \text{Stop} \rangle$  obtain  $c' \text{ mds}' \text{ mem}_2'$  where ?lc2  $\rightsquigarrow \langle c', \text{mds}', \text{mem}_2' \rangle$ 
  using not-stop-eval
  by blast
  moreover
  from bisim have ?lc2  $\approx$  ?lc1
  by (metis mm-equiv-sym)
  ultimately obtain  $c_1' \text{ mds}_1' \text{ mem}_1'$ 
  where  $\langle \text{Stop}, \text{mds}, \text{mem}_1 \rangle \rightsquigarrow \langle c_1', \text{mds}_1', \text{mem}_1' \rangle$ 
  using mm-equiv-strong-low-bisim
  unfolding strong-low-bisim-mm-def
  by blast
  thus False
  by (metis (lifting) stop-no-eval)
qed

```

lemma \mathcal{R} -typed-step:

$\llbracket \vdash \Gamma \{ c_1 \} \Gamma' ;$
 $mds\text{-consistent } mds \Gamma ;$
 $mem_1 =_{\Gamma} mem_2 ;$
 $\langle c_1, mds, mem_1 \rangle \rightsquigarrow \langle c_1', mds', mem_1 \rangle \rrbracket \implies$
 $(\exists c_2' mem_2'. \langle c_1, mds, mem_2 \rangle \rightsquigarrow \langle c_2', mds', mem_2 \rangle \wedge$
 $\langle c_1', mds', mem_1 \rangle \mathcal{R}_{\Gamma'}^u \langle c_2', mds', mem_2 \rangle)$

proof (*induct arbitrary: mds* c_1' *rule: has-type.induct*)

case (*seq-type* Γ c_1 Γ'' c_2 Γ' mds)

show $?case$

proof (*cases* $c_1 = Stop$)

assume $c_1 = Stop$

hence [*simp*]: $c_1' = c_2$ $mds' = mds$ $mem_1' = mem_1$

using *seq-type*

by (*auto simp: seq-stop-elim*)

from *seq-type* $\langle c_1 = Stop \rangle$ **have** $\Gamma \sqsubseteq_c \Gamma''$

by (*metis stop-cxt*)

hence $\vdash \Gamma \{ c_2 \} \Gamma'$

by (*metis context-le-refl seq-type(3) sub*)

have $\langle c_2, mds, mem_1 \rangle \mathcal{R}_{\Gamma'}^1 \langle c_2, mds, mem_2 \rangle$

apply (*rule* $\mathcal{R}_1.intro$ [*of* Γ])

by (*auto simp: seq-type* $\langle \vdash \Gamma \{ c_2 \} \Gamma' \rangle$)

thus $?case$

using $\mathcal{R}.intro_1$

apply *clarify*

apply (*rule-tac* $x = c_2$ **in** *exI*)

apply (*rule-tac* $x = mem_2$ **in** *exI*)

apply (*auto simp:* $\langle c_1 = Stop \rangle$)

by (*metis cxt-to-stmt.simps(1) eval_w-simplep.seq-stop eval_wp.unannotated eval_wp-eval_w-eq*)

next

assume $c_1 \neq Stop$

with $\langle c_1 ;; c_2, mds, mem_1 \rangle \rightsquigarrow \langle c_1', mds', mem_1 \rangle$ **obtain** c_1'' **where**
 c_1'' -props:

$\langle c_1, mds, mem_1 \rangle \rightsquigarrow \langle c_1'', mds', mem_1 \rangle \wedge c_1' = c_1'' ;; c_2$

by (*metis seq-elim*)

with *seq-type(2)* **obtain** $c_2'' mem_2'$ **where** c_2'' -props:

$\langle c_1, mds, mem_2 \rangle \rightsquigarrow \langle c_2'', mds', mem_2 \rangle \wedge \langle c_1'', mds', mem_1 \rangle \mathcal{R}_{\Gamma''}^u \langle c_2'',$
 $mds', mem_2 \rangle$

by (*metis seq-type(5) seq-type(6)*)

hence $\langle c_1'' ;; c_2, mds', mem_1 \rangle \mathcal{R}_{\Gamma'}^u \langle c_2'' ;; c_2, mds', mem_2 \rangle$

apply (*rule conjE*)

apply (*erule* $\mathcal{R}.elim$, *auto*)

apply (*metis* $\mathcal{R}.intro_3$ $\mathcal{R}_3\text{-aux}.intro_1$ *seq-type(3)*)

apply (*metis* $\mathcal{R}.intro_3$ $\mathcal{R}_3\text{-aux}.intro_2$ *seq-type(3)*)

by (*metis* $\mathcal{R}.intro_3$ $\mathcal{R}_3\text{-aux}.intro_3$ *seq-type(3)*)

moreover

from c_2'' -props **have** $\langle c_1 ;; c_2, mds, mem_2 \rangle \rightsquigarrow \langle c_2'' ;; c_2, mds', mem_2 \rangle$

by (*metis eval_w.seq*)


```

    ultimately show ?case
      by (metis c1''-props)
  qed
next
case (anno-type Γ' Γ upd c Γ'' mds)
have mem1 =Γ' mem2
  by (metis less-eq-Sec-def anno-type(5) anno-type(7) tyenv-eq-def)
have mds-consistent (update-modes upd mds) Γ'
  by (metis (lifting) anno-type(1) anno-type(6) mode-update-add-anno)
then obtain c2' mem2' where (⟨c, update-modes upd mds, mem2⟩ ∼2 ⟨c2', mds',
mem2'⟩ ∧
  ⟨c1', mds', mem1'⟩  $\mathcal{R}^u_{\Gamma''}$  ⟨c2', mds', mem2'⟩)
  using anno-type
  apply auto
  by (metis ⟨mem1 =Γ' mem2⟩ anno-type(1) upd-elim)
thus ?case
  apply (rule-tac x = c2' in exI)
  apply (rule-tac x = mem2' in exI)
  apply auto
  by (metis cxt-to-stmt.simps(1) evalw.decl)
next
case stop-type
with stop-no-eval show ?case by auto
next
case (skip-type Γ mds)
moreover
with skip-type have [simp]: mds' = mds c1' = Stop mem1' = mem1
  using skip-elim
  by (metis, metis, metis)
with skip-type have ⟨Stop, mds, mem1⟩  $\mathcal{R}^1_{\Gamma}$  ⟨Stop, mds, mem2⟩
  by auto
thus ?case
  using  $\mathcal{R}$ .intro1 and unannotated [where c = Skip and E = []]
  apply auto
  by (metis evalw-simple.skip)
next
case (assign1 x Γ e t mds)
hence [simp]: c1' = Stop mds' = mds mem1' = mem1 (x := evA mem1 e)
  using assign-elim
  by (auto, metis)
have mem1 (x := evA mem1 e) =Γ mem2 (x := evA mem2 e)
proof (cases to-total Γ x)
  assume to-total Γ x = High
  thus ?thesis
    using assign1 tyenv-eq-def
    by auto
next
assume to-total Γ x = Low
with assign1 have [simp]: t = Low

```

by (*metis less-eq-Sec-def to-total-def*)
 hence $\text{dma } x = \text{Low}$
 using *assign₁ <to-total $\Gamma x = \text{Low}$ >*
 by (*metis to-total-def*)
 with *assign₁ have $\forall v \in \text{aexp-vars } e. \text{to-total } \Gamma v = \text{Low}$*
 using *type-low-vars-low*
 by *auto*
 thus *?thesis*
 using *eval-vars-det_A*
 apply (*auto simp: tyenv-eq-def*)
 apply (*metis (no-types) assign₁(5) tyenv-eq-def*)
 by (*metis assign₁(5) tyenv-eq-def*)
 qed
 hence $\langle x \leftarrow e, \text{mds}, \text{mem}_2 \rangle \rightsquigarrow \langle \text{Stop}, \text{mds}', \text{mem}_2 (x := \text{ev}_A \text{mem}_2 e) \rangle$
 $\langle \text{Stop}, \text{mds}', \text{mem}_1 (x := \text{ev}_A \text{mem}_1 e) \rangle \mathcal{R}^u_{\Gamma} \langle \text{Stop}, \text{mds}', \text{mem}_2 (x := \text{ev}_A$
 $\text{mem}_2 e) \rangle$
 apply *auto*
 apply (*metis cxt-to-stmt.simps(1) eval_w.unannotated eval_w-simple.assign*)
 by (*rule \mathcal{R} .intro₁, auto simp: assign₁*)
 thus *?case*
 using $\langle c_1' = \text{Stop} \rangle$ and $\langle \text{mem}_1' = \text{mem}_1 (x := \text{ev}_A \text{mem}_1 e) \rangle$
 by *blast*
 next
 case (*assign₂ x Γ e t mds*)
 hence [*simp*]: $c_1' = \text{Stop}$ $\text{mds}' = \text{mds}$ $\text{mem}_1' = \text{mem}_1 (x := \text{ev}_A \text{mem}_1 e)$
 using *assign-elim assign₂*
 by (*auto, metis*)
 let $? \Gamma' = \Gamma (x \mapsto t)$
 have $\langle x \leftarrow e, \text{mds}, \text{mem}_2 \rangle \rightsquigarrow \langle \text{Stop}, \text{mds}, \text{mem}_2 (x := \text{ev}_A \text{mem}_2 e) \rangle$
 using *assign₂*
 by (*metis cxt-to-stmt.simps(1) eval_w-simplep.assign eval_wp.unannotated eval_wp-eval_w-eq*)
 moreover
 have $\langle \text{Stop}, \text{mds}, \text{mem}_1 (x := \text{ev}_A \text{mem}_1 e) \rangle \mathcal{R}^1_{\Gamma'} \langle \text{Stop}, \text{mds}, \text{mem}_2 (x := \text{ev}_A$
 $\text{mem}_2 e) \rangle$
 proof (*auto*)
 from *assign₂ show mds-consistent mds ? Γ'*
 apply (*simp add: mds-consistent-def*)
 by (*metis (lifting) insert-absorb assign₂(1)*)
 next
 show $\text{mem}_1 (x := \text{ev}_A \text{mem}_1 e) =_{\Gamma'} \text{mem}_2 (x := \text{ev}_A \text{mem}_2 e)$
 unfolding *tyenv-eq-def*
 proof (*auto*)
 assume *to-total ($\Gamma(x \mapsto t)$) x = Low*
 with $\langle \Gamma \vdash_a e \in t \rangle$ have $\bigwedge x. x \in \text{aexp-vars } e \implies \text{to-total } \Gamma x = \text{Low}$
 by (*metis assign₂.prems(1) domI fun-upd-same option.sel to-total-def*
type-low-vars-low)
 thus $\text{ev}_A \text{mem}_1 e = \text{ev}_A \text{mem}_2 e$
 by (*metis assign₂.prems(2) eval-vars-det_A tyenv-eq-def*)
 next

```

fix y
assume  $y \neq x$  and to-total ( $\Gamma (x \mapsto t)$ )  $y = \text{Low}$ 
thus  $\text{mem}_1 y = \text{mem}_2 y$ 
by (metis (full-types) assign2.prems(2)) domD domI fun-upd-other to-total-def
tyenv-eq-def)
qed
qed
ultimately have  $\langle x \leftarrow e, \text{mds}, \text{mem}_2 \rangle \rightsquigarrow \langle \text{Stop}, \text{mds}', \text{mem}_2 (x := \text{ev}_A \text{mem}_2 e) \rangle$ 
 $\langle \text{Stop}, \text{mds}', \text{mem}_1 (x := \text{ev}_A \text{mem}_1 e) \rangle \mathcal{R}^u_{\Gamma(x \mapsto t)} \langle \text{Stop}, \text{mds}', \text{mem}_2 (x := \text{ev}_A \text{mem}_2 e) \rangle$ 
using  $\mathcal{R}.\text{intro}_1$ 
by auto
thus ?case
using  $\langle \text{mds}' = \text{mds} \rangle \langle c_1' = \text{Stop} \rangle \langle \text{mem}_1' = \text{mem}_1(x := \text{ev}_A \text{mem}_1 e) \rangle$ 
by blast
next
case (if-type  $\Gamma e \text{th el } \Gamma'$ )
have  $(\langle c_1', \text{mds}, \text{mem}_1 \rangle, \langle c_1', \text{mds}, \text{mem}_2 \rangle) \in \mathcal{R} \Gamma'$ 
apply (rule intro1)
apply clarify
apply (rule  $\mathcal{R}_1.\text{intro}$  [where  $\Gamma = \Gamma$  and  $\Gamma' = \Gamma'$ ])
apply (auto simp: if-type)
by (metis (lifting) if-elim if-type(2) if-type(4) if-type(8))
have eq-condition:  $\text{ev}_B \text{mem}_1 e = \text{ev}_B \text{mem}_2 e \implies ?\text{case}$ 
proof –
assume  $\text{ev}_B \text{mem}_1 e = \text{ev}_B \text{mem}_2 e$ 
with if-type(8) have  $(\langle \text{If } e \text{th el}, \text{mds}, \text{mem}_2 \rangle \rightsquigarrow \langle c_1', \text{mds}, \text{mem}_2 \rangle)$ 
apply (cases  $\text{ev}_B \text{mem}_1 e$ )
apply (subgoal-tac  $c_1' = \text{th}$ )
apply clarify
apply (metis cxt-to-stmt.simps(1) evalw-simplep.if-true evalwp.unannotated
evalwp-evalw-eq if-type(8))
apply (metis if-elim if-type(8))
apply (subgoal-tac  $c_1' = \text{el}$ )
apply (metis (opaque-lifting, mono-tags) cxt-to-stmt.simps(1) evalw.unannotated
evalw-simple.if-false if-type(8))
by (metis if-elim if-type(8))
thus ?thesis
by (metis  $\langle c_1', \text{mds}, \text{mem}_1 \rangle \mathcal{R}^u_{\Gamma'} \langle c_1', \text{mds}, \text{mem}_2 \rangle$ ) if-elim if-type(8))
qed
have  $\text{mem}_1 =_{\text{mds}'} \text{mem}_2$ 
apply (auto simp: low-mds-eq-def mds-consistent-def)
apply (subgoal-tac  $x \notin \text{dom } \Gamma$ )
apply (metis if-type(7) to-total-def tyenv-eq-def)
by (metis (lifting, mono-tags) CollectD Sec.simps(2) if-type(6) mds-consistent-def)
obtain  $t$  where  $\Gamma \vdash_b e \in t$ 
by (metis type-bexpr.intros)
from if-type show ?case

```

proof (cases t)
assume $t = \text{High}$
with if-type show ?thesis
proof (cases $ev_B \text{ mem}_1 e = ev_B \text{ mem}_2 e$)
assume $ev_B \text{ mem}_1 e = ev_B \text{ mem}_2 e$
with eq-condition show ?thesis **by auto**
next
assume $neq: ev_B \text{ mem}_1 e \neq ev_B \text{ mem}_2 e$
from if-type $\langle t = \text{High} \rangle$ **have** $th \sim_{\text{mds}} el$
by (metis $\langle \Gamma \vdash_b e \in t \rangle$)
from neq show ?thesis
proof (cases $ev_B \text{ mem}_1 e$)
assume $ev_B \text{ mem}_1 e$
hence $c_1' = th$
by (metis (lifting) if-elim if-type(8))
hence $\langle \text{If } e \text{ th } el, \text{ mds}, \text{ mem}_2 \rangle \rightsquigarrow \langle el, \text{ mds}, \text{ mem}_2 \rangle$
by (metis $\langle ev_B \text{ mem}_1 e \rangle$ cxt-to-stmt.simps(1) eval_w.unannotated eval_w-simple.if-false
if-type(8) neq)
moreover
with $\langle \text{mem}_1 =_{\text{mds}}^l \text{ mem}_2 \rangle$ **have** $\langle th, \text{ mds}, \text{ mem}_1 \rangle \approx \langle el, \text{ mds}, \text{ mem}_2 \rangle$
by (metis low-indistinguishable-def $\langle th \sim_{\text{mds}} el \rangle$)
have $\forall x \in \text{dom } \Gamma'. \Gamma' x = \text{Some High}$
using if-type $\langle t = \text{High} \rangle$
by (metis $\langle \Gamma \vdash_b e \in t \rangle$)
have $\langle th, \text{ mds}, \text{ mem}_1 \rangle \mathcal{R}_{\Gamma'}^2 \langle el, \text{ mds}, \text{ mem}_2 \rangle$
by (metis (lifting) \mathcal{R}_2 .intro $\langle \forall x \in \text{dom } \Gamma'. \Gamma' x = \text{Some High} \rangle$ $\langle \langle th, \text{ mds}, \text{ mem}_1 \rangle \approx \langle el, \text{ mds}, \text{ mem}_2 \rangle$ if-type(2) if-type(4) if-type(6))
ultimately show ?thesis
using \mathcal{R} .intro₂
apply clarify
by (metis $\langle c_1' = th \rangle$ if-elim if-type(8))
next
assume $\neg ev_B \text{ mem}_1 e$
hence [simp]: $c_1' = el$
by (metis (lifting) if-type(8) if-elim)
hence $\langle \text{If } e \text{ th } el, \text{ mds}, \text{ mem}_2 \rangle \rightsquigarrow \langle th, \text{ mds}, \text{ mem}_2 \rangle$
by (metis (opaque-lifting, mono-tags) $\langle \neg ev_B \text{ mem}_1 e \rangle$ cxt-to-stmt.simps(1)
eval_w.unannotated eval_w-simple.if-true if-type(8) neq)
moreover
from $\langle th \sim_{\text{mds}} el \rangle$ **have** $el \sim_{\text{mds}} th$
by (metis low-indistinguishable-sym)
with $\langle \text{mem}_1 =_{\text{mds}}^l \text{ mem}_2 \rangle$ **have** $\langle el, \text{ mds}, \text{ mem}_1 \rangle \approx \langle th, \text{ mds}, \text{ mem}_2 \rangle$
by (metis low-indistinguishable-def)
have $\forall x \in \text{dom } \Gamma'. \Gamma' x = \text{Some High}$
using if-type $\langle t = \text{High} \rangle$
by (metis $\langle \Gamma \vdash_b e \in t \rangle$)
have $\langle el, \text{ mds}, \text{ mem}_1 \rangle \mathcal{R}_{\Gamma'}^2 \langle th, \text{ mds}, \text{ mem}_2 \rangle$
apply (rule \mathcal{R}_2 .intro [where $\Gamma_1 = \Gamma$ and $\Gamma_2 = \Gamma$])
by (auto simp: if-type $\langle \langle el, \text{ mds}, \text{ mem}_1 \rangle \approx \langle th, \text{ mds}, \text{ mem}_2 \rangle \rangle$ $\langle \forall x \in \text{dom}$

$\Gamma'. \Gamma' x = \text{Some High}$
ultimately show *?thesis*
using $\mathcal{R}.intro_2$
apply *clarify*
by (*metis* $\langle c_1' = el \rangle$ *if-elim if-type(8)*)
qed
qed
next
assume $t = Low$
with *if-type* **have** $ev_B mem_1 e = ev_B mem_2 e$
using *eval-vars-det_B*
apply (*simp add: tyenv-eq-def* $\langle \Gamma \vdash_b e \in t \rangle$ *type-low-vars-low-b*)
by (*metis (lifting)* $\langle \Gamma \vdash_b e \in t \rangle$ *type-low-vars-low-b*)
with *eq-condition* **show** *?thesis* **by** *auto*
qed
next
case (*while-type* $\Gamma e c$)
hence [*simp*]: $c_1' = (If e (c ;; While e c) Stop)$ **and**
[*simp*]: $mds' = mds$ **and**
[*simp*]: $mem_1' = mem_1$
by (*auto simp: while-elim*)
with *while-type* **have** $\langle While e c, mds, mem_2 \rangle \rightsquigarrow \langle c_1', mds, mem_2 \rangle$
by (*metis cxt-to-stmt.simps(1) eval_w-simplep.while eval_wp.unannotated eval_wp-eval_w-eq*)
moreover
have $\langle c_1', mds, mem_1 \rangle \mathcal{R}^1_{\Gamma} \langle c_1', mds, mem_2 \rangle$
apply (*rule* $\mathcal{R}_1.intro$ [**where** $\Gamma = \Gamma$])
apply (*auto simp: while-type*)
apply (*rule if-type*)
apply (*metis (lifting) Sec.simps(1) while-type(1) type-bexpr-elim*)
apply (*rule seq-type* [**where** $\Gamma' = \Gamma$])
by (*auto simp: while-type*)
ultimately show *?case*
using $\mathcal{R}.intro_1$ [*of* Γ]
by (*auto, blast*)
next
case (*sub* $\Gamma_1 c \Gamma_1' \Gamma \Gamma' mds c_1'$)
hence $dom \Gamma_1 \subseteq dom \Gamma \ dom \Gamma' \subseteq dom \Gamma_1'$
apply (*metis (lifting) context-le-def equalityE*)
by (*metis context-le-def sub(4) order-refl*)
hence *mds-consistent* $mds \Gamma_1$
using *sub*
apply (*auto simp: mds-consistent-def*)
apply (*metis (lifting, full-types) context-le-def domD mem-Collect-eq*)
by (*metis (lifting, full-types) context-le-def domD mem-Collect-eq*)
moreover **have** $mem_1 =_{\Gamma_1} mem_2$
unfolding *tyenv-eq-def*
by (*metis (lifting, no-types) context-le-def less-eq-Sec-def sub.hyps(3) sub.prem(2)*)
to-total-def tyenv-eq-def
ultimately obtain $c_2' mem_2'$ **where** c_2' -*props*: $\langle c, mds, mem_2 \rangle \rightsquigarrow \langle c_2', mds',$

$\langle c_1', mds', mem_1 \rangle \mathcal{R}_{\Gamma_1}^u \langle c_2', mds', mem_2 \rangle$
using *sub*
by *blast*
moreover
have $\bigwedge c_1 mds mem_1 c_2 mem_2. \langle c_1, mds, mem_1 \rangle \mathcal{R}_{\Gamma_1}^u \langle c_2, mds, mem_2 \rangle \implies$
 $\langle c_1, mds, mem_1 \rangle \mathcal{R}_{\Gamma'}^u \langle c_2, mds, mem_2 \rangle$
proof –
fix $c_1 mds mem_1 c_2 mem_2$
let $?lc_1 = \langle c_1, mds, mem_1 \rangle$ **and** $?lc_2 = \langle c_2, mds, mem_2 \rangle$
assume *asm*: $?lc_1 \mathcal{R}_{\Gamma_1}^u ?lc_2$
moreover
have $?lc_1 \mathcal{R}_{\Gamma_1}^1 ?lc_2 \implies ?lc_1 \mathcal{R}_{\Gamma'}^1 ?lc_2$
apply (*erule* \mathcal{R}_1 -*elim*)
apply *auto*
by (*metis* (*lifting*) *has-type.sub sub(4)* *stop-ctx stop-type*)
moreover
have $?lc_1 \mathcal{R}_{\Gamma_1}^2 ?lc_2 \implies ?lc_1 \mathcal{R}_{\Gamma'}^2 ?lc_2$
proof –
assume *r2*: $?lc_1 \mathcal{R}_{\Gamma_1}^2 ?lc_2$
then obtain $\Lambda_1 \Lambda_2$ **where** $\vdash \Lambda_1 \{ c_1 \} \Gamma_1' \vdash \Lambda_2 \{ c_2 \} \Gamma_1'$ *mds-consistent*
 $mds \Lambda_1$
mds-consistent $mds \Lambda_2$
by (*metis* \mathcal{R}_2 -*elim*)
hence $\vdash \Lambda_1 \{ c_1 \} \Gamma'$
using *sub(4)*
by (*metis* *context-le-refl has-type.sub sub(4)*)
moreover
have $\vdash \Lambda_2 \{ c_2 \} \Gamma'$
by (*metis* $\langle \vdash \Lambda_2 \{ c_2 \} \Gamma_1' \rangle$ *context-le-refl has-type.sub sub(4)*)
moreover
from *r2* **have** $\forall x \in dom \Gamma_1'. \Gamma_1' x = Some High$
apply (*rule* \mathcal{R}_2 -*elim*)
by *auto*
hence $\forall x \in dom \Gamma'. \Gamma' x = Some High$
by (*metis* *Sec.simps(2)* $\langle dom \Gamma' \subseteq dom \Gamma_1' \rangle$ *context-le-def domD less-eq-Sec-def*
sub(4) *rev-subsetD option.sel*)
ultimately show *thesis*
by (*metis* (*no-types*) \mathcal{R}_2 .*intro* \mathcal{R}_2 -*elim'* $\langle mds-consistent mds \Lambda_1 \rangle$ $\langle mds-consistent$
 $mds \Lambda_2 \rangle$ *r2*)
qed
moreover
have $?lc_1 \mathcal{R}_{\Gamma_1}^3 ?lc_2 \implies ?lc_1 \mathcal{R}_{\Gamma'}^3 ?lc_2$
apply (*erule* \mathcal{R}_3 -*elim*)
proof –
fix $\Gamma c_1'' c_2''' c$
assume [*simp*]: $c_1 = c_1''$;; $c c_2 = c_2'''$;; c
assume *case1*: $\vdash \Gamma \{ c \} \Gamma_1' \langle c_1'', mds, mem_1 \rangle \mathcal{R}_{\Gamma}^1 \langle c_2''', mds, mem_2 \rangle$
hence $\vdash \Gamma \{ c \} \Gamma'$

```

    using context-le-refl has-type.sub sub.hyps(4)
    by blast
  with case1 show  $\langle c_1, mds, mem_1 \rangle \mathcal{R}^3_{\Gamma'} \langle c_2, mds, mem_2 \rangle$ 
    using  $\mathcal{R}_3\text{-aux.intro}_1$  by simp
next
  fix  $\Gamma c_1'' c_2''' c''$ 
  assume [simp]:  $c_1 = c_1'' ; ; c'' c_2 = c_2''' ; ; c''$ 
  assume  $\langle c_1'', mds, mem_1 \rangle \mathcal{R}^2_{\Gamma} \langle c_2''', mds, mem_2 \rangle \vdash \Gamma \{c''\} \Gamma_1'$ 
  thus  $\langle c_1, mds, mem_1 \rangle \mathcal{R}^3_{\Gamma'} \langle c_2, mds, mem_2 \rangle$ 
    using  $\mathcal{R}_3\text{-aux.intro}_2$ 
    apply simp
    apply (rule  $\mathcal{R}_3\text{-aux.intro}_2$  [where  $\Gamma = \Gamma$ ])
    apply simp
    by (metis context-le-refl has-type.sub sub.hyps(4))
next
  fix  $\Gamma c_1'' c_2''' c''$ 
  assume [simp]:  $c_1 = c_1'' ; ; c'' c_2 = c_2''' ; ; c''$ 
  assume  $\langle c_1'', mds, mem_1 \rangle \mathcal{R}^3_{\Gamma} \langle c_2''', mds, mem_2 \rangle \vdash \Gamma \{c''\} \Gamma_1'$ 
  thus  $\langle c_1, mds, mem_1 \rangle \mathcal{R}^3_{\Gamma'} \langle c_2, mds, mem_2 \rangle$ 
    using  $\mathcal{R}_3\text{-aux.intro}_3$ 
    apply auto
    by (metis (opaque-lifting, no-types) context-le-refl has-type.sub sub.hyps(4))
qed
ultimately show ?thesis  $c_1 mds mem_1 c_2 mem_2$ 
  by (auto simp: R.intros)
qed
with  $c_2'$ -props show ?case
  by blast
qed

```

lemma \mathcal{R}_1 -weak-bisim:
weak-bisim $(\mathcal{R}_1 \Gamma') (\mathcal{R} \Gamma')$
unfolding *weak-bisim-def*
using $\mathcal{R}_1\text{-elim}$ $\mathcal{R}\text{-typed-step}$
by *auto*

lemma \mathcal{R} -to- \mathcal{R}_3 : $\llbracket \langle c_1, mds, mem_1 \rangle \mathcal{R}^u_{\Gamma} \langle c_2, mds, mem_2 \rangle ; \vdash \Gamma \{c\} \Gamma' \rrbracket \implies$
 $\langle c_1 ; ; c, mds, mem_1 \rangle \mathcal{R}^3_{\Gamma'} \langle c_2 ; ; c, mds, mem_2 \rangle$
apply (*erule* $\mathcal{R}\text{-elim}$)
by *auto*

lemma \mathcal{R}_2 -implies-typeable: $\langle c_1, mds, mem_1 \rangle \mathcal{R}^2_{\Gamma'} \langle c_2, mds, mem_2 \rangle \implies \exists \Gamma_1. \vdash$
 $\Gamma_1 \{c_2\} \Gamma'$
apply (*erule* $\mathcal{R}_2\text{-elim}$)
by *auto*

lemma \mathcal{R}_3 -weak-bisim:
weak-bisim $(\mathcal{R}_3 \Gamma') (\mathcal{R} \Gamma')$

proof –

```

{
  fix c1 mds mem1 c2 mem2 c1' mds' mem1'
  assume case3: ((c1, mds, mem1), (c2, mds, mem2)) ∈ ℛ3 Γ'
  assume eval: ⟨c1, mds, mem1⟩ ∼ ⟨c1', mds', mem1'⟩
  have ∃ c2' mem2'. ⟨c2, mds, mem2⟩ ∼ ⟨c2', mds', mem2'⟩ ∧ ⟨c1', mds', mem1'⟩
  ℛuΓ' ⟨c2', mds', mem2'⟩
  using case3 eval
  apply simp

proof (induct arbitrary: c1' rule: ℛ3-aux.induct)
  case (intro1 Γ c1 mds mem1 c2 mem2 c Γ')
  hence [simp]: c2 = c1
  by (metis (lifting) ℛ1-elim)
  thus ?case
proof (cases c1 = Stop)
  assume [simp]: c1 = Stop
  from intro1(1) show ?thesis
  apply (rule ℛ1-elim)
  apply simp
  apply (rule-tac x = c in exI)
  apply (rule-tac x = mem2 in exI)
  apply (rule conjI)
  apply (metis ⟨c1 = Stop⟩ cxt-to-stmt.simps(1) evalw-simplep.seq-stop
evalwp.unannotated evalwp-evalw-eq intro1.prems seq-stop-elim)
  apply (rule ℛ.intro1, clarify)
  by (metis (no-types) ℛ1.intro ⟨c1 = Stop⟩ context-le-refl intro1.prems
intro1(2) seq-stop-elim stop-cxt sub)
  next
  assume c1 ≠ Stop
  from intro1
  obtain c1'' where ⟨c1, mds, mem1⟩ ∼ ⟨c1'', mds', mem1'⟩ ∧ c1' = (c1'' ;;
c)
  by (metis ⟨c1 ≠ Stop⟩ intro1.prems seq-elim)
  with intro1
  obtain c2'' mem2' where ⟨c2, mds, mem2⟩ ∼ ⟨c2'', mds', mem2'⟩ ⟨c1'',
mds', mem1'⟩ ℛuΓ ⟨c2'', mds', mem2'⟩
  using ℛ1-weak-bisim and weak-bisim-def
  by blast
  thus ?thesis
  using intro1(2) ℛ-to-ℛ3
  apply (rule-tac x = c2'' ;; c in exI)
  apply (rule-tac x = mem2' in exI)
  apply (rule conjI)
  apply (metis evalw.seq)
  apply auto
  apply (rule ℛ.intro3)
  by (metis (opaque-lifting, no-types) ⟨⟨c1, mds, mem1⟩ ∼ ⟨c1'', mds', mem1'⟩
∧ c1' = c1'' ;; c⟩)

```



```

qed
next
case (intro2 Γ c1 mds mem1 c2 mem2 cn Γ')
thus ?case
proof (cases c1 = Stop)
  assume [simp]: c1 = Stop
  hence [simp]: c1' = cn mds' = mds mem1' = mem1
    using eval intro2 seq-stop-elim
    by auto
  from intro2 have bisim: ⟨c1, mds, mem1⟩ ≈ ⟨c2, mds, mem2⟩
    by (metis (lifting) R2-elim')
  from intro2 obtain Γ1 where ⊢ Γ1 { c2 } Γ
    by (metis R2-implies-typeable)
  with bisim have [simp]: c2 = Stop
    apply auto
    apply (rule stop-bisim [of mds mem1 c2 mem2 Γ1 Γ])
    by simp-all
  have ⟨c2 ;; cn, mds, mem2⟩ ∼ ⟨cn, mds', mem2⟩
    apply (auto simp: intro2)
    by (metis cxt-to-stmt.simps(1) evalw-simplep.seq-stop evalwp.unannotated
evalwp-evalw-eq)
  moreover
  from intro2(1) have mds-consistent mds Γ
    apply auto
    apply (erule R2-elim)
    apply (simp add: mds-consistent-def)
    by (metis context-le-def stop-cxt)
  moreover
  from bisim have mem1 =mdsl mem2
    by (auto simp: mm-equiv.simps strong-low-bisim-mm-def)
  have ∀ x ∈ dom Γ. Γ x = Some High
    using intro2(1)
    by (metis R2-elim')
  hence mem1 =Γ mem2
    using ⟨mds-consistent mds Γ⟩ ⟨mem1 =mdsl mem2⟩
    apply (auto simp: tyenv-eq-def low-mds-eq-def mds-consistent-def)
    by (metis Sec.simps(1) ⟨∀ x ∈ dom Γ. Γ x = Some High⟩ ⟨mds' = mds⟩
domI option.sel to-total-def)
  ultimately have ⟨cn, mds, mem1⟩ R1Γ ⟨cn, mds, mem2⟩
    by (metis (lifting) R1.intro intro2(2))
  thus ?thesis
    using R.intro1
    apply auto
    by (metis ⟨c2 ;; cn, mds, mem2⟩ ∼ ⟨cn, mds', mem2⟩ ⟨c2 = Stop⟩ ⟨mds'
= mds⟩)
next
  assume c1 ≠ Stop
  then obtain c1''' where c1' = c1''' ;; cn ⟨c1, mds, mem1⟩ ∼ ⟨c1''', mds',
mem1^⟩

```

```

    by (metis (no-types) intro2.prem1 seq-elim)
  then obtain  $c_2'''$   $mem_2'$  where  $c_2'''$ -props:
     $\langle c_2, mds, mem_2 \rangle \rightsquigarrow \langle c_2''', mds', mem_2' \rangle \wedge$ 
     $\langle c_1''', mds', mem_1' \rangle \mathcal{R}^2_{\Gamma} \langle c_2''', mds', mem_2' \rangle$ 
    using  $\mathcal{R}_2$ -bisim-step intro2
    by blast
  let  $?c_2' = c_2'''$  ;; cn
  from  $c_2'''$ -props have  $\langle c_2 ;; cn, mds, mem_2 \rangle \rightsquigarrow \langle ?c_2', mds', mem_2' \rangle$ 
    by (metis (lifting) intro2 eval_w.seq)
  moreover
  have  $(\langle c_1''' ;; cn, mds', mem_1' \rangle, \langle ?c_2', mds', mem_2' \rangle) \in \mathcal{R}_3 \Gamma'$ 
    by (metis (lifting)  $\mathcal{R}_3$ -aux.intro2  $c_2'''$ -props intro2(2) mem-Collect-eq
case-prodI)
  ultimately show ?thesis
    using  $\mathcal{R}$ .intro3
    by (metis (lifting)  $\mathcal{R}_3$ -aux.intro2  $\langle c_1' = c_1''' ;; cn \rangle$   $c_2'''$ -props intro2(2))
qed
next
case (intro3  $\Gamma$   $c_1$   $mds$   $mem_1$   $c_2$   $mem_2$   $c$   $\Gamma'$ )
thus ?case
  apply (cases  $c_1 = Stop$ )
  apply blast
proof -
  assume  $c_1 \neq Stop$ 
  then obtain  $c_1''$  where  $\langle c_1, mds, mem_1 \rangle \rightsquigarrow \langle c_1'', mds', mem_1' \rangle$   $c_1' = (c_1''$ 
;; c)
    by (metis intro3.prem1 seq-elim)
  then obtain  $c_2''$   $mem_2'$  where  $\langle c_2, mds, mem_2 \rangle \rightsquigarrow \langle c_2'', mds', mem_2' \rangle$ 
 $\langle c_1'', mds', mem_1' \rangle \mathcal{R}^u_{\Gamma} \langle c_2'', mds', mem_2' \rangle$ 
    using intro3(2) mem-Collect-eq case-prodI
    by metis
  thus ?thesis
    apply (rule-tac  $x = c_2''$  ;; c in exI)
    apply (rule-tac  $x = mem_2'$  in exI)
    apply (rule conjI)
    apply (metis eval_w.seq)
    apply (erule  $\mathcal{R}$ -elim)
    apply simp-all
    apply (metis  $\mathcal{R}$ .intro3  $\mathcal{R}$ -to- $\mathcal{R}_3$   $\langle \langle c_1'', mds', mem_1' \rangle \mathcal{R}^u_{\Gamma} \langle c_2'', mds',$ 
 $mem_2' \rangle \rangle \langle c_1' = c_1'' ;; c \rangle$  intro3(3))
    apply (metis (lifting)  $\mathcal{R}$ .intro3  $\mathcal{R}$ -to- $\mathcal{R}_3$   $\langle \langle c_1'', mds', mem_1' \rangle \mathcal{R}^u_{\Gamma} \langle c_2'',$ 
 $mds', mem_2' \rangle \rangle \langle c_1' = c_1'' ;; c \rangle$  intro3(3))
    by (metis (lifting)  $\mathcal{R}$ .intro3  $\mathcal{R}_3$ -aux.intro3  $\langle c_1' = c_1'' ;; c \rangle$  intro3(3))
  qed
qed
}
thus ?thesis
  unfolding weak-bisim-def
  by auto

```

qed

lemma \mathcal{R} -bisim: strong-low-bisim-mm ($\mathcal{R} \Gamma'$)
unfolding strong-low-bisim-mm-def
proof (auto)
from \mathcal{R} -sym **show** sym ($\mathcal{R} \Gamma'$) .
next
from \mathcal{R} -closed-glob-consistent **show** closed-glob-consistent ($\mathcal{R} \Gamma'$) .
next
fix c_1 mds mem_1 c_2 mem_2
assume $\langle c_1, mds, mem_1 \rangle \mathcal{R}^u_{\Gamma'} \langle c_2, mds, mem_2 \rangle$
thus $mem_1 =_{mds} mem_2$
apply (rule \mathcal{R} -elim)
by (auto simp: \mathcal{R}_1 -mem-eq \mathcal{R}_2 -mem-eq \mathcal{R}_3 -mem-eq)
next
fix c_1 mds mem_1 c_2 mem_2 c_1' mds' mem_1'
assume eval: $\langle c_1, mds, mem_1 \rangle \rightsquigarrow \langle c_1', mds', mem_1' \rangle$
assume R : $\langle c_1, mds, mem_1 \rangle \mathcal{R}^u_{\Gamma'} \langle c_2, mds, mem_2 \rangle$
from R **show** $\exists c_2' mem_2'. \langle c_2, mds, mem_2 \rangle \rightsquigarrow \langle c_2', mds', mem_2' \rangle \wedge$
 $\langle c_1', mds', mem_1' \rangle \mathcal{R}^u_{\Gamma'} \langle c_2', mds', mem_2' \rangle$
apply (rule \mathcal{R} -elim)
apply (insert \mathcal{R}_1 -weak-bisim \mathcal{R}_2 -weak-bisim \mathcal{R}_3 -weak-bisim eval weak-bisim-def)
apply (clarify, blast)+
by (metis mem-Collect-eq case-prodI)
qed

lemma Typed-in- \mathcal{R} :
assumes typeable: $\vdash \Gamma \{ c \} \Gamma'$
assumes mds-cons: mds-consistent mds Γ
assumes mem-eq: $\forall x. to-total \Gamma x = Low \longrightarrow mem_1 x = mem_2 x$
shows $\langle c, mds, mem_1 \rangle \mathcal{R}^u_{\Gamma'} \langle c, mds, mem_2 \rangle$
apply (rule \mathcal{R} .intro₁ [of Γ'])
apply clarify
apply (rule \mathcal{R}_1 .intro [of Γ])
by (auto simp: assms tyenv-eq-def)

theorem type-soundness:
assumes well-typed: $\vdash \Gamma \{ c \} \Gamma'$
assumes mds-cons: mds-consistent mds Γ
assumes mem-eq: $\forall x. to-total \Gamma x = Low \longrightarrow mem_1 x = mem_2 x$
shows $\langle c, mds, mem_1 \rangle \approx \langle c, mds, mem_2 \rangle$
using \mathcal{R} -bisim Typed-in- \mathcal{R}
by (metis mds-cons mem-eq mm-equiv.simps well-typed)

definition $\Gamma_0 :: 'Var TyEnv$
where $\Gamma_0 x = None$

```

inductive type-global :: ('Var, 'AExp, 'BExp) Stmt list  $\Rightarrow$  bool
  ( $\vdash$  - [120] 1000)
  where
     $\llbracket$  list-all ( $\lambda$  c.  $\vdash$   $\Gamma_0$  { c }  $\Gamma_0$ ) cs ;
       $\forall$  mem. sound-mode-use (add-initial-modes cs, mem)  $\rrbracket \Longrightarrow$ 
      type-global cs

inductive-cases type-global-elim:  $\vdash$  cs

lemma mdss-consistent: mds-consistent mdss  $\Gamma_0$ 
  by (auto simp: mdss-def mds-consistent-def  $\Gamma_0$ -def)

lemma typed-secure:
   $\llbracket \vdash \Gamma_0$  { c }  $\Gamma_0 \rrbracket \Longrightarrow$  com-sifum-secure c
  apply (auto simp: com-sifum-secure-def low-indistinguishable-def mds-consistent-def
    type-soundness)
  apply (auto simp: low-mds-eq-def)
  apply (rule type-soundness [of  $\Gamma_0$  c  $\Gamma_0$ ])
  apply (auto simp: mdss-consistent-to-total-def  $\Gamma_0$ -def)
  by (metis empty-iff mdss-def)

lemma  $\llbracket$  mds-consistent mds  $\Gamma_0$  ; dma x = Low  $\rrbracket \Longrightarrow$  x  $\notin$  mds AsmNoRead
  by (auto simp: mds-consistent-def  $\Gamma_0$ -def)

lemma list-all-set:  $\forall$  x  $\in$  set xs. P x  $\Longrightarrow$  list-all P xs
  by (metis (lifting) list-all-iff)

theorem type-soundness-global:
  assumes typeable:  $\vdash$  cs
  assumes no-assms-term: no-assumptions-on-termination cs
  shows prog-sifum-secure cs
  using typeable
  apply (rule type-global-elim)
  apply (subgoal-tac  $\forall$  c  $\in$  set cs. com-sifum-secure c)
  apply (metis list-all-set no-assms-term sifum-compositionality sound-mode-use.simps)
  by (metis (lifting) list-all-iff typed-secure)

end
end

```

6 Type System for Ensuring Locally Sound Use of Modes

```

theory LocallySoundModeUse
imports Main Security Language
begin

```

6.1 Typing Rules

locale *sifum-modes* = *sifum-lang* *ev_A* *ev_B* +
sifum-security *dma* *Stop* *eval_w*
for *ev_A* :: ('Var, 'Val) Mem ⇒ 'AExp ⇒ 'Val
and *ev_B* :: ('Var, 'Val) Mem ⇒ 'BExp ⇒ bool

context *sifum-modes*
begin

abbreviation *eval-abv-modes* :: (-, 'Var, 'Val) LocalConf ⇒ (-, -, -) LocalConf ⇒ bool

(**infixl** \rightsquigarrow 70)

where

$x \rightsquigarrow y \equiv (x, y) \in \text{eval}_w$

fun *update-annos* :: 'Var Mds ⇒ 'Var ModeUpd list ⇒ 'Var Mds

(**infix** \oplus 140)

where

update-annos *mds* [] = *mds* |

update-annos *mds* (*a* # *as*) = *update-annos* (*update-modes* *a* *mds*) *as*

fun *annotate* :: ('Var, 'AExp, 'BExp) Stmt ⇒ 'Var ModeUpd list ⇒ ('Var, 'AExp, 'BExp) Stmt

(**infix** \otimes 140)

where

annotate *c* [] = *c* |

annotate *c* (*a* # *as*) = (*annotate* *c* *as*)@[*a*]

inductive *mode-type* :: 'Var Mds ⇒

('Var, 'AExp, 'BExp) Stmt ⇒

'Var Mds ⇒ bool (⊢ - { - } -)

where

skip: ⊢ *mds* { *Skip* \otimes *annos* } (*mds* \oplus *annos*) |

assign: [$x \notin \text{mds GuarNoWrite}$; $\text{aexp-vars } e \cap \text{mds GuarNoRead} = \{\}$] \implies

⊢ *mds* { ($x \leftarrow e$) \otimes *annos* } (*mds* \oplus *annos*) |

*if*_: [⊢ (*mds* \oplus *annos*) { *c*₁ } *mds*' ;

⊢ (*mds* \oplus *annos*) { *c*₂ } *mds*' ;

$\text{bexp-vars } e \cap \text{mds GuarNoRead} = \{\}$] \implies

⊢ *mds* { *If* *e* *c*₁ *c*₂ \otimes *annos* } *mds*' |

*while*_: [$\text{mds}' = \text{mds} \oplus \text{annos}$; ⊢ *mds*' { *c* } *mds*' ; $\text{bexp-vars } e \cap \text{mds}' \text{ GuarNoRead} = \{\}$] \implies

⊢ *mds* { *While* *e* *c* \otimes *annos* } *mds*' |

*seq*_: [⊢ *mds* { *c*₁ } *mds*' ; ⊢ *mds*' { *c*₂ } *mds*'] \implies ⊢ *mds* { *c*₁ ;; *c*₂ } *mds*' |

*sub*_: [⊢ *mds*₂ { *c* } *mds*'₂' ; $\text{mds}_1 \leq \text{mds}_2$; $\text{mds}_2' \leq \text{mds}_1'$] \implies

⊢ *mds*₁ { *c* } *mds*'₁'

6.2 Soundness of the Type System

lemma *cxt-eval*:

$\llbracket \langle \text{cxt-to-stmt} \ \square \ c, \text{ mds}, \text{ mem} \rangle \rightsquigarrow \langle \text{cxt-to-stmt} \ \square \ c', \text{ mds}', \text{ mem}' \rangle \rrbracket \implies$
 $\langle c, \text{ mds}, \text{ mem} \rangle \rightsquigarrow \langle c', \text{ mds}', \text{ mem}' \rangle$
by *auto*

lemma *update-preserves-le*:
 $\text{mds}_1 \leq \text{mds}_2 \implies (\text{mds}_1 \oplus \text{annos}) \leq (\text{mds}_2 \oplus \text{annos})$
proof (*induct annos arbitrary: mds₁ mds₂*)
case *Nil*
thus *?case by simp*
next
case (*Cons a annos mds₁ mds₂*)
hence *update-modes a mds₁ ≤ update-modes a mds₂*
by (*case-tac a, auto simp: le-fun-def*)
with *Cons show ?case*
by *auto*
qed

lemma *doesnt-read-annos*:
 $\text{doesnt-read } c \ x \implies \text{doesnt-read } (c \otimes \text{annos}) \ x$
unfolding *doesnt-read-def*
apply *clarify*
apply (*induct annos*)
apply *simp*
apply (*metis (lifting)*)
apply *auto*
apply (*rule cxt-eval*)
apply (*rule eval_w.decl*)
by (*metis cxt-eval eval_w.decl upd-elim*)

lemma *doesnt-modify-annos*:
 $\text{doesnt-modify } c \ x \implies \text{doesnt-modify } (c \otimes \text{annos}) \ x$
unfolding *doesnt-modify-def*
apply *auto*
apply (*induct annos*)
apply *simp*
apply *auto*
by (*metis (lifting) upd-elim*)

lemma *stop-loc-reach*:
 $\llbracket \langle c', \text{ mds}', \text{ mem}' \rangle \in \text{loc-reach } \langle \text{Stop}, \text{ mds}, \text{ mem} \rangle \rrbracket \implies$
 $c' = \text{Stop} \wedge \text{mds}' = \text{mds}$
apply (*induct rule: loc-reach.induct*)
by (*auto simp: stop-no-eval*)

lemma *stop-doesnt-access*:
 $\text{doesnt-modify } \text{Stop} \ x \wedge \text{doesnt-read } \text{Stop} \ x$

unfolding *doesn't-modify-def* **and** *doesn't-read-def*
using *stop-no-eval*
by *auto*

lemma *skip-eval-step*:

$\langle \text{Skip} \otimes \text{annos}, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle \text{Stop}, \text{mds} \oplus \text{annos}, \text{mem} \rangle$
apply (*induct annos arbitrary: mds*)
apply *simp*
apply (*metis cxt-to-stmt.simps(1) eval_w.unannotated eval_w-simple.skip*)
apply *simp*
apply (*insert eval_w.decl*)
apply (*rule cxt-eval*)
apply (*rule eval_w.decl*)
by *auto*

lemma *skip-eval-elim*:

$\llbracket \langle \text{Skip} \otimes \text{annos}, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \rangle \rrbracket \implies c' = \text{Stop} \wedge \text{mds}' = \text{mds}$
 $\oplus \text{annos} \wedge \text{mem}' = \text{mem}$
apply (*rule ccontr*)
apply (*insert skip-eval-step deterministic*)
apply *clarify*
apply *auto*
by *metis+*

lemma *skip-doesnt-read*:

doesn't-read (*Skip* \otimes *annos*) *x*
apply (*rule doesn't-read-annos*)
apply (*auto simp: doesn't-read-def*)
by (*metis annotate.simps(1) skip-elim skip-eval-step*)

lemma *skip-doesnt-write*:

doesn't-modify (*Skip* \otimes *annos*) *x*
apply (*rule doesn't-modify-annos*)
apply (*auto simp: doesn't-modify-def*)
by (*metis skip-elim*)

lemma *skip-loc-reach*:

$\llbracket \langle c', \text{mds}', \text{mem}' \rangle \in \text{loc-reach} \langle \text{Skip} \otimes \text{annos}, \text{mds}, \text{mem} \rangle \rrbracket \implies$
 $(c' = \text{Stop} \wedge \text{mds}' = (\text{mds} \oplus \text{annos})) \vee (c' = \text{Skip} \otimes \text{annos} \wedge \text{mds}' = \text{mds})$
apply (*induct rule: loc-reach.induct*)
apply (*metis fst-conv snd-conv*)
apply (*metis skip-eval-elim stop-no-eval*)
by *metis*

lemma *skip-doesnt-access*:

$\llbracket lc \in \text{loc-reach} \langle \text{Skip} \otimes \text{annos}, \text{mds}, \text{mem} \rangle ; lc = \langle c', \text{mds}', \text{mem}' \rangle \rrbracket \implies$
doesn't-read $c' x \wedge$ *doesn't-modify* $c' x$
apply (*subgoal-tac* ($c' = \text{Stop} \wedge \text{mds}' = (\text{mds} \oplus \text{annos}) \vee (c' = \text{Skip} \otimes \text{annos} \wedge \text{mds}' = \text{mds})$))

apply (*rule conjI, erule disjE*)
apply (*simp add: doesnt-read-def stop-no-eval*)
apply (*metis (lifting) annotate.simps skip-doesnt-read*)
apply (*erule disjE*)
apply (*simp add: doesnt-modify-def stop-no-eval*)
apply (*metis (lifting) annotate.simps skip-doesnt-write*)
by (*metis skip-loc-reach*)

lemma *assign-doesnt-modify*:
 $\llbracket x \neq y \rrbracket \implies \text{doesnt-modify } ((x \leftarrow e) \otimes \text{annos}) y$
apply (*rule doesnt-modify-annos*)
apply (*simp add: doesnt-modify-def*)
by (*metis assign-elim fun-upd-apply*)

lemma *assign-annos-eval*:
 $\langle (x \leftarrow e) \otimes \text{annos}, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle \text{Stop}, \text{mds} \oplus \text{annos}, \text{mem } (x := \text{ev}_A \text{ mem } e) \rangle$
apply (*induct annos arbitrary: mds*)
apply (*simp only: annotate.simps update-annos.simps*)
apply (*rule cxt-eval*)
apply (*rule eval_w.unannotated*)
apply (*rule eval_w-simple.assign*)
apply (*rule cxt-eval*)
apply (*simp del: cxt-to-stmt.simps*)
apply (*rule eval_w.decl*)
by *auto*

lemma *assign-annos-eval-elim*:
 $\llbracket \langle (x \leftarrow e) \otimes \text{annos}, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \rangle \rrbracket \implies$
 $c' = \text{Stop} \wedge \text{mds}' = \text{mds} \oplus \text{annos}$
apply (*rule ccontr*)
apply (*insert deterministic assign-annos-eval*)
apply *auto*
apply (*metis (lifting)*)
by *metis*

lemma *mem-upd-commute*:
 $\llbracket x \neq y \rrbracket \implies \text{mem } (x := v_1, y := v_2) = \text{mem } (y := v_2, x := v_1)$
by (*metis fun-upd-twist*)

lemma *assign-doesnt-read*:
 $\llbracket y \notin \text{aexp-vars } e \rrbracket \implies \text{doesnt-read } ((x \leftarrow e) \otimes \text{annos}) y$
apply (*rule doesnt-read-annos*)
proof (*cases x = y*)
assume $y \notin \text{aexp-vars } e$
and [*simp*]: $x = y$
show *doesnt-read* $(x \leftarrow e) y$
unfolding *doesnt-read-def*
apply (*rule allI*)
apply (*rename-tac mds mem c' mds' mem'*)


```

apply (rule impI)
apply (subgoal-tac  $c' = \text{Stop} \wedge \text{mds}' = \text{mds} \wedge \text{mem}' = \text{mem} (x := \text{ev}_A \text{ mem}$ 
e))
  apply simp
  apply (rule disjI2)
  apply clarify
  apply (rule cxt-eval)
  apply (rule evalw.unannotated)
  apply simp
  apply (metis (opaque-lifting, no-types)  $\langle x = y \rangle \langle y \notin \text{aexp-vars } e \rangle \text{eval}_w\text{-simple.assign}$ 
eval-vars-detA fun-upd-apply fun-upd-upd)
  by (metis assign-elim)
next
assume asms:  $y \notin \text{aexp-vars } e \ x \neq y$ 
show doesn't-read  $(x \leftarrow e) y$ 
  unfolding doesn't-read-def
  apply (rule allI)+
  apply (rename-tac  $\text{mds mem } c' \text{ mds}' \text{ mem}'$ )
  apply (rule impI)
  apply (subgoal-tac  $c' = \text{Stop} \wedge \text{mds}' = \text{mds} \wedge \text{mem}' = \text{mem} (x := \text{ev}_A \text{ mem}$ 
e))
  apply simp
  apply (rule disjI1)
  apply (insert asms)
  apply clarify
  apply (subgoal-tac  $\text{mem} (x := \text{ev}_A \text{ mem } e, y := v) = \text{mem} (y := v, x := \text{ev}_A$ 
mem e))
  apply simp
  apply (rule cxt-eval)
  apply (rule evalw.unannotated)
  apply (metis evalw-simple.assign eval-vars-detA fun-upd-apply)
  apply (metis mem-upd-commute)
  by (metis assign-elim)
qed

```

lemma *assign-loc-reach*:

```

 $\llbracket \langle c', \text{mds}', \text{mem}' \rangle \in \text{loc-reach} ((x \leftarrow e) \otimes \text{annos}, \text{mds}, \text{mem}) \rrbracket \implies$ 
 $(c' = \text{Stop} \wedge \text{mds}' = (\text{mds} \oplus \text{annos})) \vee (c' = (x \leftarrow e) \otimes \text{annos} \wedge \text{mds}' = \text{mds})$ 
apply (induct rule: loc-reach.induct)
apply simp-all
by (metis assign-annos-eval-elim stop-no-eval)

```

lemma *if-doesnt-modify*:

```

doesn't-modify (If  $e \ c_1 \ c_2 \ \otimes \ \text{annos}$ )  $x$ 
apply (rule doesn't-modify-annos)
by (auto simp: doesn't-modify-def)

```

lemma *vars-eval_B*:

```

 $x \notin \text{bexp-vars } e \implies \text{ev}_B \text{ mem } e = \text{ev}_B (\text{mem} (x := v)) e$ 

```

by (metis (lifting) eval-vars-det_B fun-upd-other)

lemma if-doesnt-read:

$x \notin \text{bexp-vars } e \implies \text{doesnt-read } (\text{If } e \ c_1 \ c_2 \ \otimes \ \text{annos}) \ x$
apply (rule doesnt-read-annos)
apply (auto simp: doesnt-read-def)
apply (rename-tac mds mem c' mds' mem' v va)
apply (case-tac ev_B mem e)
apply (subgoal-tac c' = c₁ ∧ mds' = mds ∧ mem' = mem)
apply auto
apply (rule cxt-eval)
apply (rule eval_w.unannotated)
apply (rule eval_w-simple.if-true)
apply (metis (lifting) vars-eval_B)
apply (subgoal-tac c' = c₂ ∧ mds' = mds ∧ mem' = mem)
apply auto
apply (rule cxt-eval)
apply (rule eval_w.unannotated)
apply (rule eval_w-simple.if-false)
by (metis (lifting) vars-eval_B)

lemma if-eval-true:

$\llbracket \text{ev}_B \ \text{mem } e \rrbracket \implies$
 $\langle \text{If } e \ c_1 \ c_2 \ \otimes \ \text{annos}, \ \text{mds}, \ \text{mem} \rangle \rightsquigarrow \langle c_1, \ \text{mds} \oplus \ \text{annos}, \ \text{mem} \rangle$
apply (induct annos arbitrary: mds)
apply simp
apply (metis cxt-eval eval_w.unannotated eval_w-simple.if-true)
by (metis annotate.simps(2) cxt-eval eval_w.decl update-annos.simps(2))

lemma if-eval-false:

$\llbracket \neg \text{ev}_B \ \text{mem } e \rrbracket \implies$
 $\langle \text{If } e \ c_1 \ c_2 \ \otimes \ \text{annos}, \ \text{mds}, \ \text{mem} \rangle \rightsquigarrow \langle c_2, \ \text{mds} \oplus \ \text{annos}, \ \text{mem} \rangle$
apply (induct annos arbitrary: mds)
apply simp
apply (metis cxt-eval eval_w.unannotated eval_w-simple.if-false)
by (metis annotate.simps(2) cxt-eval eval_w.decl update-annos.simps(2))

lemma if-eval-elim:

$\llbracket \langle \text{If } e \ c_1 \ c_2 \ \otimes \ \text{annos}, \ \text{mds}, \ \text{mem} \rangle \rightsquigarrow \langle c', \ \text{mds}', \ \text{mem}' \rangle \rrbracket \implies$
 $((c' = c_1 \wedge \text{ev}_B \ \text{mem } e) \vee (c' = c_2 \wedge \neg \text{ev}_B \ \text{mem } e)) \wedge \text{mds}' = \text{mds} \oplus \ \text{annos} \wedge$
 $\text{mem}' = \text{mem}$
apply (rule ccontr)
apply (cases ev_B mem e)
apply (insert if-eval-true deterministic)
apply (metis prod.inject)
by (metis prod.inject if-eval-false deterministic)

lemma if-eval-elim':

$\llbracket \langle \text{If } e \ c_1 \ c_2, \ \text{mds}, \ \text{mem} \rangle \rightsquigarrow \langle c', \ \text{mds}', \ \text{mem}' \rangle \rrbracket \implies$

$((c' = c_1 \wedge ev_B \text{ mem } e) \vee (c' = c_2 \wedge \neg ev_B \text{ mem } e)) \wedge mds' = mds \wedge mem' = mem$
using *if-eval-elim* [where annos = []]
by *auto*

lemma *loc-reach-refl'*:

$\langle c, mds, mem \rangle \in \text{loc-reach } \langle c, mds, mem \rangle$
apply (*subgoal-tac* $\exists lc. lc \in \text{loc-reach } lc \wedge lc = \langle c, mds, mem \rangle$)
apply *blast*
by (*metis loc-reach.refl fst-conv snd-conv*)

lemma *if-loc-reach*:

$\llbracket \langle c', mds', mem^\wedge \rangle \in \text{loc-reach } \langle \text{If } e \ c_1 \ c_2 \otimes \text{annos}, mds, mem \rangle \rrbracket \implies$
 $(c' = \text{If } e \ c_1 \ c_2 \otimes \text{annos} \wedge mds' = mds) \vee$
 $(\exists mem''. \langle c', mds', mem^\wedge \rangle \in \text{loc-reach } \langle c_1, mds \oplus \text{annos}, mem'' \rangle) \vee$
 $(\exists mem''. \langle c', mds', mem^\wedge \rangle \in \text{loc-reach } \langle c_2, mds \oplus \text{annos}, mem'' \rangle)$
apply (*induct rule: loc-reach.induct*)
apply (*metis fst-conv snd-conv*)
apply (*erule disjE*)
apply (*erule conjE*)
apply *simp*
apply (*drule if-eval-elim*)
apply (*erule conjE*)
apply (*erule disjE*)
apply (*erule conjE*)
apply *simp*
apply (*metis loc-reach-refl'*)
apply (*metis loc-reach-refl'*)
apply (*metis loc-reach.step*)
by (*metis loc-reach.mem-diff*)

lemma *if-loc-reach'*:

$\llbracket \langle c', mds', mem^\wedge \rangle \in \text{loc-reach } \langle \text{If } e \ c_1 \ c_2, mds, mem \rangle \rrbracket \implies$
 $(c' = \text{If } e \ c_1 \ c_2 \wedge mds' = mds) \vee$
 $(\exists mem''. \langle c', mds', mem^\wedge \rangle \in \text{loc-reach } \langle c_1, mds, mem'' \rangle) \vee$
 $(\exists mem''. \langle c', mds', mem^\wedge \rangle \in \text{loc-reach } \langle c_2, mds, mem'' \rangle)$
using *if-loc-reach* [where annos = []]
by *simp*

lemma *seq-loc-reach*:

$\llbracket \langle c', mds', mem^\wedge \rangle \in \text{loc-reach } \langle c_1 ;; c_2, mds, mem \rangle \rrbracket \implies$
 $(\exists c''. c' = c'' ;; c_2 \wedge \langle c'', mds', mem^\wedge \rangle \in \text{loc-reach } \langle c_1, mds, mem \rangle) \vee$
 $(\exists c'' mds'' mem''. \langle \text{Stop}, mds'', mem'' \rangle \in \text{loc-reach } \langle c_1, mds, mem \rangle \wedge$
 $\langle c', mds', mem^\wedge \rangle \in \text{loc-reach } \langle c_2, mds'', mem'' \rangle)$
apply (*induct rule: loc-reach.induct*)
apply *simp*
apply (*metis loc-reach-refl'*)
apply *simp*
apply (*metis (no-types) loc-reach.step loc-reach-refl' seq-elim seq-stop-elim*)

by (metis (lifting) loc-reach.mem-diff)

lemma seq-doesnt-read:

[[doesnt-read c x]] \implies doesnt-read (c ;; c') x
 apply (auto simp: doesnt-read-def)
 apply (rename-tac mds mem c'a mds' mem' v va)
 apply (case-tac c = Stop)
 apply auto
 apply (subgoal-tac c'a = c' \wedge mds' = mds \wedge mem' = mem)
 apply simp
 apply (metis cxt-eval eval_w.unannotated eval_w-simple.seq-stop)
 apply (metis (lifting) seq-stop-elim)
 by (metis (lifting, no-types) eval_w.seq seq-elim)

lemma seq-doesnt-modify:

[[doesnt-modify c x]] \implies doesnt-modify (c ;; c') x
 apply (auto simp: doesnt-modify-def)
 apply (case-tac c = Stop)
 apply auto
 apply (metis (lifting) seq-stop-elim)
 by (metis (no-types) seq-elim)

inductive-cases seq-stop-elim': $\langle \text{Stop} ;; c, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \rangle$

lemma seq-stop-elim: $\langle \text{Stop} ;; c, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \rangle \implies$

$c' = c \wedge \text{mds}' = \text{mds} \wedge \text{mem}' = \text{mem}$
 apply (erule seq-stop-elim')
 apply (metis eval_w.unannotated seq-stop-elim)
 apply (metis eval_w.seq seq-stop-elim)
 by (metis (lifting) Stmt.simps(28) Stmt.simps(34) cxt-seq-elim)

lemma seq-split:

[[$\langle \text{Stop}, \text{mds}', \text{mem}' \rangle \in \text{loc-reach} \langle c_1 ;; c_2, \text{mds}, \text{mem} \rangle$]] \implies
 $\exists \text{mds}'' \text{mem}'' . \langle \text{Stop}, \text{mds}'', \text{mem}'' \rangle \in \text{loc-reach} \langle c_1, \text{mds}, \text{mem} \rangle \wedge$
 $\langle \text{Stop}, \text{mds}', \text{mem}' \rangle \in \text{loc-reach} \langle c_2, \text{mds}'', \text{mem}'' \rangle$
 apply (drule seq-loc-reach)
 by (metis Stmt.simps(41))

lemma while-eval:

$\langle \text{While } e \ c \ \otimes \ \text{annos}, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle (\text{If } e \ (c ;; \text{While } e \ c) \ \text{Stop}), \text{mds} \oplus \ \text{annos}, \text{mem} \rangle$
 apply (induct annos arbitrary: mds)
 apply simp
 apply (rule cxt-eval)
 apply (rule eval_w.unannotated)
 apply (metis (lifting) eval_w-simple.while)
 apply simp
 by (metis cxt-eval eval_w.decl)

lemma *while-eval'*:

$\langle \text{While } e \ c, \text{ mds}, \text{ mem} \rangle \rightsquigarrow \langle \text{If } e \ (c \ ; \ ; \ \text{While } e \ c) \ \text{Stop}, \text{ mds}, \text{ mem} \rangle$
using *while-eval* [**where** *annos* = []]
by *auto*

lemma *while-eval-elim*:

$\llbracket \langle \text{While } e \ c \otimes \text{ annos}, \text{ mds}, \text{ mem} \rangle \rightsquigarrow \langle c', \text{ mds}', \text{ mem}' \rangle \rrbracket \implies$
 $(c' = \text{If } e \ (c \ ; \ ; \ \text{While } e \ c) \ \text{Stop} \wedge \text{ mds}' = \text{mds} \oplus \text{ annos} \wedge \text{ mem}' = \text{ mem})$
apply (*rule ccontr*)
apply (*insert while-eval deterministic*)
by *blast*

lemma *while-eval-elim'*:

$\llbracket \langle \text{While } e \ c, \text{ mds}, \text{ mem} \rangle \rightsquigarrow \langle c', \text{ mds}', \text{ mem}' \rangle \rrbracket \implies$
 $(c' = \text{If } e \ (c \ ; \ ; \ \text{While } e \ c) \ \text{Stop} \wedge \text{ mds}' = \text{mds} \wedge \text{ mem}' = \text{ mem})$
using *while-eval-elim* [**where** *annos* = []]
by *auto*

lemma *while-doesnt-read*:

$\llbracket x \notin \text{bexp-vars } e \rrbracket \implies \text{doesnt-read } (\text{While } e \ c \otimes \text{ annos}) \ x$
unfolding *doesnt-read-def*
using *while-eval while-eval-elim*
by *metis*

lemma *while-doesnt-modify*:

$\text{doesnt-modify } (\text{While } e \ c \otimes \text{ annos}) \ x$
unfolding *doesnt-modify-def*
using *while-eval-elim*
by *metis*

lemma *disjE3*:

$\llbracket A \vee B \vee C ; A \implies P ; B \implies P ; C \implies P \rrbracket \implies P$
by *auto*

lemma *disjE5*:

$\llbracket A \vee B \vee C \vee D \vee E ; A \implies P ; B \implies P ; C \implies P ; D \implies P ; E \implies P \rrbracket$
 $\implies P$
by *auto*

lemma *if-doesnt-read'*:

$x \notin \text{bexp-vars } e \implies \text{doesnt-read } (\text{If } e \ c_1 \ c_2) \ x$
using *if-doesnt-read* [**where** *annos* = []]
by *auto*

theorem *mode-type-sound*:

assumes *typeable*: $\vdash \text{ mds}_1 \ \{ \ c \ } \ \text{ mds}'_1$
assumes *mode-le*: $\text{ mds}_2 \leq \text{ mds}_1$
shows $\forall \text{ mem. } (\langle \text{Stop}, \text{ mds}'_2, \text{ mem}' \rangle \in \text{loc-reach } \langle c, \text{ mds}_2, \text{ mem} \rangle \longrightarrow \text{ mds}'_2 \leq$

$mds_1 \wedge$
 $\text{locally-sound-mode-use } \langle c, mds_2, mem \rangle$
using *typeable mode-le*
proof (*induct arbitrary: mds₂ mds'₂' mem' mem rule: mode-type.induct*)
case (*skip mds annos*)
thus *?case*
apply *auto*
apply (*metis (lifting) skip-eval-step skip-loc-reach stop-no-eval update-preserves-le*)
apply (*simp add: locally-sound-mode-use-def*)
by (*metis annotate.simps skip-doesnt-access*)
next
case (*assign x mds e annos*)
hence $\forall mem. \text{locally-sound-mode-use } \langle (x \leftarrow e) \otimes annos, mds_2, mem \rangle$
unfolding *locally-sound-mode-use-def*
proof (*clarify*)
fix *mem c' mds' mem' y*
assume *asm: $\langle c', mds', mem' \rangle \in \text{loc-reach } \langle (x \leftarrow e) \otimes annos, mds_2, mem \rangle$*
hence $c' = (x \leftarrow e) \otimes annos \wedge mds' = mds_2 \vee c' = \text{Stop} \wedge mds' = mds_2 \oplus$
annos
using *assign-loc-reach by blast*
thus ($y \in mds' \text{ GuarNoRead} \longrightarrow \text{doesnt-read } c' y$) \wedge
 $(y \in mds' \text{ GuarNoWrite} \longrightarrow \text{doesnt-modify } c' y)$
proof
assume $c' = (x \leftarrow e) \otimes annos \wedge mds' = mds_2$
thus *?thesis*
proof (*auto*)
assume $y \in mds_2 \text{ GuarNoRead}$
hence $y \notin \text{aexp-vars } e$
by (*metis IntD2 IntI assign.hyps(2) assign.premis empty-iff inf-apply*
le-iff-inf)
with *assign-doesnt-read show doesnt-read* $((x \leftarrow e) \otimes annos) y$
by *blast*
next
assume $y \in mds_2 \text{ GuarNoWrite}$
hence $x \neq y$
by (*metis assign.hyps(1) assign.premis in-mono le-fun-def*)
with *assign-doesnt-modify show doesnt-modify* $((x \leftarrow e) \otimes annos) y$
by *blast*
qed
next
assume $c' = \text{Stop} \wedge mds' = mds_2 \oplus annos$
with *stop-doesnt-access show ?thesis by blast*
qed
qed
thus *?case*
apply *auto*
by (*metis assign.premis assign-annos-eval assign-loc-reach stop-no-eval up-*
date-preserves-le)
next

```

case (if- mds annos c1 mds'' c2 e)
  let ?c = (If e c1 c2) ⊗ annos
from if- have modes-le': mds2 ⊕ annos ≤ mds ⊕ annos
  by (metis (lifting) update-preserves-le)
from if- show ?case
  apply (simp add: locally-sound-mode-use-def)
  apply clarify
  apply (rule conjI)
  apply clarify
  prefer 2
  apply clarify
proof -
  fix mem
  assume ⟨Stop, mds2', mem'⟩ ∈ loc-reach ⟨If e c1 c2 ⊗ annos, mds2, mem⟩
  with modes-le' and if- show mds2' ≤ mds''
    by (metis if-eval-false if-eval-true if-loc-reach stop-no-eval)
next
  fix mem c' mds' mem' x
  assume ⟨c', mds', mem'⟩ ∈ loc-reach ⟨If e c1 c2 ⊗ annos, mds2, mem⟩
  hence (c' = If e c1 c2 ⊗ annos ∧ mds' = mds2) ∨
    (∃ mem''. ⟨c', mds', mem'⟩ ∈ loc-reach ⟨c1, mds2 ⊕ annos, mem''⟩) ∨
    (∃ mem''. ⟨c', mds', mem'⟩ ∈ loc-reach ⟨c2, mds2 ⊕ annos, mem''⟩)
  using if-loc-reach by blast
  thus (x ∈ mds' GuarNoRead → doesnt-read c' x) ∧
    (x ∈ mds' GuarNoWrite → doesnt-modify c' x)
  proof
    assume c' = If e c1 c2 ⊗ annos ∧ mds' = mds2
    thus ?thesis
    proof (auto)
      assume x ∈ mds2 GuarNoRead
      with ⟨bexp-vars e ∩ mds GuarNoRead = {}⟩ and ⟨mds2 ≤ mds⟩ have x ∉
        bexp-vars e
      by (metis IntD2 disjoint-iff-not-equal inf-fun-def le-iff-inf)
      thus doesnt-read (If e c1 c2 ⊗ annos) x
      using if-doesnt-read by blast
    next
      assume x ∈ mds2 GuarNoWrite
      thus doesnt-modify (If e c1 c2 ⊗ annos) x
      using if-doesnt-modify by blast
    qed
  next
  assume (∃ mem''. ⟨c', mds', mem'⟩ ∈ loc-reach ⟨c1, mds2 ⊕ annos, mem''⟩)
  ∨
    (∃ mem''. ⟨c', mds', mem'⟩ ∈ loc-reach ⟨c2, mds2 ⊕ annos, mem''⟩)
  with if- show ?thesis
  by (metis locally-sound-mode-use-def modes-le')
  qed
qed
next

```

case (*while* $mdsa$ mds *annos* c e)
hence $mds_2 \oplus annos \leq mds \oplus annos$
by (*metis* (*lifting*) *update-preserves-le*)
have *while-loc-reach*: $\bigwedge c' mds' mem' mem.$
 $\langle c', mds', mem \rangle \in loc\text{-}reach \langle While\ e\ c \otimes annos, mds_2, mem \rangle \implies$
 $c' = While\ e\ c \otimes annos \wedge mds' = mds_2 \vee$
 $c' = While\ e\ c \wedge mds' \leq mdsa \vee$
 $c' = Stmt.If\ e\ (c ;; While\ e\ c)\ Stop \wedge mds' \leq mdsa \vee$
 $c' = Stop \wedge mds' \leq mdsa \vee$
 $(\exists c'' mem'' mds_3.$
 $c' = c'' ;; While\ e\ c \wedge$
 $mds_3 \leq mdsa \wedge \langle c'', mds', mem \rangle \in loc\text{-}reach \langle c, mds_3, mem'' \rangle)$
proof –
fix $mem\ c' mds' mem'$
assume $\langle c', mds', mem \rangle \in loc\text{-}reach \langle While\ e\ c \otimes annos, mds_2, mem \rangle$
thus *?thesis* $c' mds' mem' mem$
apply (*induct rule*: *loc-reach.induct*)
apply *simp-all*
apply (*erule disjE*)
apply *simp*
apply (*metis* $\langle mds_2 \oplus annos \leq mds \oplus annos \rangle$ *while.hyps(1)* *while-eval-elim*)
apply (*erule disjE*)
apply (*metis* *while-eval-elim'*)
apply (*erule disjE*)
apply *simp*
apply (*metis* *if-eval-elim' loc-reach-refl'*)
apply (*erule disjE*)
apply (*metis* *stop-no-eval*)
apply (*erule exE*)
apply (*rename-tac* $c' mds' mem' c'' mds'' mem'' c''a$)
apply (*case-tac* $c''a = Stop$)
apply (*insert* *while.hyps(3)*)
apply (*metis* *seq-stop-elim* *while.hyps(3)*)
apply (*metis* *loc-reach.step seq-elim*)
by (*metis* (*full-types*) *loc-reach.mem-diff*)
qed
from *while* **show** *?case*
proof (*auto*)
fix mem
assume $\langle Stop, mds_2', mem \rangle \in loc\text{-}reach \langle While\ e\ c \otimes annos, mds_2, mem \rangle$
thus $mds_2' \leq mds \oplus annos$
by (*metis* *Stmt.distinct(35)* *stop-no-eval* *while.hyps(1)* *while-eval* *while-loc-reach*)
next
fix mem
from *while* **have** *berp-vars* $e \cap (mds_2 \oplus annos)$ *GuarNoRead* = $\{\}$
by (*metis* (*lifting*, *no-types*) *Int-empty-right* *Int-left-commute* $\langle mds_2 \oplus annos \leq mds \oplus annos \rangle$ *inf-fun-def* *le-iff-inf*)
show *locally-sound-mode-use* $\langle While\ e\ c \otimes annos, mds_2, mem \rangle$
unfolding *locally-sound-mode-use-def*


```

apply (rule allI)+
apply (rule impI)
proof –
  fix c' mds' mem'
  define lc where lc = ⟨While e c ⊗ annos, mds2, mem⟩
  assume ⟨c', mds', mem'⟩ ∈ loc-reach lc
  thus ∀ x. (x ∈ mds' GuarNoRead → doesnt-read c' x) ∧
    (x ∈ mds' GuarNoWrite → doesnt-modify c' x)
    apply (simp add: lc-def)
    apply (drule while-loc-reach)
    apply (erule disjE5)
  proof (auto del: conjI)
    fix x
    show (x ∈ mds2 GuarNoRead → doesnt-read (While e c ⊗ annos) x) ∧
      (x ∈ mds2 GuarNoWrite → doesnt-modify (While e c ⊗ annos) x)
      unfolding doesnt-read-def and doesnt-modify-def
      using while-eval and while-eval-elim
      by blast
  next
    fix x
    assume mds' ≤ mdsa
    let ?c' = If e (c ;; While e c) Stop
    have x ∈ mds' GuarNoRead → doesnt-read ?c' x
      apply clarify
      apply (rule if-doesnt-read')
    by (metis IntI ⟨mds' ≤ mdsa⟩ empty-iff le-fun-def rev-subsetD while.hyps(1))
  while.hyps(4))
  moreover
    have x ∈ mds' GuarNoWrite → doesnt-modify ?c' x
      by (metis annotate.simps(1) if-doesnt-modify)
    ultimately show (x ∈ mds' GuarNoRead → doesnt-read ?c' x) ∧
      (x ∈ mds' GuarNoWrite → doesnt-modify ?c' x) ..
  next
    fix x
    assume mds' ≤ mdsa
    show (x ∈ mds' GuarNoRead → doesnt-read Stop x) ∧
      (x ∈ mds' GuarNoWrite → doesnt-modify Stop x)
      by (metis stop-doesnt-access)
  next
    fix c'' x mem'' mds3
    assume mds3 ≤ mdsa
    assume ⟨c'', mds', mem'⟩ ∈ loc-reach ⟨c, mds3, mem''⟩
    thus (x ∈ mds' GuarNoRead → doesnt-read (c'' ;; While e c) x) ∧
      (x ∈ mds' GuarNoWrite → doesnt-modify (c'' ;; While e c) x)
      apply auto
      apply (rule seq-doesnt-read)
      apply (insert while(3))
      apply (metis ⟨mds3 ≤ mdsa⟩ locally-sound-mode-use-def while.hyps(1))
      apply (rule seq-doesnt-modify)

```

```

    by (metis ⟨mds3 ≤ mdsa⟩ locally-sound-mode-use-def while.hyps(1))
next
  fix x
  assume mds' ≤ mdsa
  show (x ∈ mds' GuarNoRead → doesnt-read (While e c) x) ∧
        (x ∈ mds' GuarNoWrite → doesnt-modify (While e c) x)
    unfolding doesnt-read-def and doesnt-modify-def
    using while-eval' and while-eval-elim'
    by blast
  qed
qed
next
case (seq mds c1 mds' c2 mds'')
thus ?case
proof (auto)
  fix mem
  assume ⟨Stop, mds2', mem^⟩ ∈ loc-reach ⟨c1 ;; c2, mds2, mem⟩
  then obtain mds2'' mem'' where ⟨Stop, mds2'', mem''^⟩ ∈ loc-reach ⟨c1, mds2,
mem⟩ and
    ⟨Stop, mds2', mem^⟩ ∈ loc-reach ⟨c2, mds2'', mem''^⟩
  using seq-split by blast
  thus mds2' ≤ mds''
  using seq by blast
next
  fix mem
  from seq show locally-sound-mode-use ⟨c1 ;; c2, mds2, mem⟩
  apply (simp add: locally-sound-mode-use-def)
  apply clarify
  apply (drule seq-loc-reach)
  apply (erule disjE)
  apply (metis seq-doesnt-modify seq-doesnt-read)
  by metis
qed
next
case (sub mds2'' c mds2' mds1 mds1' c1)
thus ?case
  apply auto
  by (metis (opaque-lifting, no-types) inf-absorb2 le-infI1)
qed
end
end

```

References

- [MSS11] Heiko Mantel, David Sands, and Henning Sudbrock. Assumptions and Guarantees for Compositional Noninterference. In *CSF*, pages 218–232. IEEE Computer Society, 2011.