A Formalization of Assumptions and Guarantees for Compositional Noninterference

Sylvia Grewe, Heiko Mantel, Daniel Schoepe

February 6, 2026

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** ‹⊑› *50*) **and**
less (**infix** ‹⊏› *50*)

Sec forms a (complete) lattice:

instantiation *Sec* :: *complete-lattice*
begin

definition *top-Sec-def*: ⊤ = *High*
definition *sup-Sec-def*: *d1* ⊔ *d2* = (*if* (*d1* = *High* ∨ *d2* = *High*) *then High else Low*)
definition *inf-Sec-def*: *d1* ⊓ *d2* = (*if* (*d1* = *Low* ∨ *d2* = *Low*) *then Low else High*)

definition *bot-Sec-def*: $\bot = Low$
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*: $\bigsqcup S = (if\ (High \in S)\ then\ High\ else\ Low)$
definition *Inf-Sec-def*: $\bigsqcap S = (if\ (Low \in S)\ then\ Low\ else\ High)$

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* ⇒ '*val*

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

type-synonym '*var Mds* = *Mode* ⇒ '*var set*

Local configurations:

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

Global configurations:

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

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

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

fixes *some-val'* :: *'Val*
assumes *stop-no-eval*: ¬ ((((*stop*, *mds*), *mem*), ((*c'*, *mds'*), *mem'*)) ∈ *eval*)
assumes *deterministic*: ⟦ (*lc*, *lc'*) ∈ *eval*; (*lc*, *lc''*) ∈ *eval* ⟧ ⟹ *lc'* = *lc''*
assumes *finite-memory*: *finite* {(*x*::*'Var*). *True*}
assumes *different-values*: *some-val* ≠ *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* ⇒ (-, -, -) *LocalConf* ⇒ *bool*
(**infixl** ‹⇝› *70*)
where
x ⇝ *y* ≡ (*x*, *y*) ∈ *eval*

abbreviation *conf-abv* :: *'Com* ⇒ *'Var Mds* ⇒ (*'Var*, *'Val*) *Mem* ⇒ (-,-,-) *LocalConf*
(‹⟨-, -, -⟩› [*0*, *0*, *0*] *1000*)
where
⟨ *c*, *mds*, *mem* ⟩ ≡ ((*c*, *mds*), *mem*)

inductive-set *meval* :: (-,-,-) *GlobalConf rel*
and *meval-abv* :: - ⇒ - ⇒ *bool* (**infixl** ‹→› *70*)
where
conf → *conf'* ≡ (*conf*, *conf'*) ∈ *meval* |
meval-intro [*iff*]: ⟦ (*cms* ! *n*, *mem*) ⇝ (*cm'*, *mem'*); *n* < *length cms* ⟧ ⟹
((*cms*, *mem*), (*cms* [*n* := *cm'*], *mem'*)) ∈ *meval*

inductive-cases *meval-elim* [*elim*!]: ((*cms*, *mem*), (*cms'*, *mem'*)) ∈ *meval*

abbreviation *meval-clos* :: - ⇒ - ⇒ *bool* (**infixl** ‹→*›› *70*)
where
conf →* *conf'* ≡ (*conf*, *conf'*) ∈ *meval**

fun *lc-set-var* :: (-, -, -) *LocalConf* ⇒ *'Var* ⇒ *'Val* ⇒ (-, -, -) *LocalConf*
where
lc-set-var (*c*, *mem*) *x v* = (*c*, *mem* (*x* := *v*))

fun *meval-k* :: *nat* ⇒ (*'Com*, *'Var*, *'Val*) *GlobalConf* ⇒ (-, -, -) *GlobalConf* ⇒ *bool*
where
meval-k 0 c c' = (*c* = *c'*) |
meval-k (*Suc n*) *c c'* = (∃ *c''*. *meval-k n c c''* ∧ *c''* → *c'*)

abbreviation *meval-k-abv* :: *nat* ⇒ (-, -, -) *GlobalConf* ⇒ (-, -, -) *GlobalConf* ⇒ *bool*
(‹- →1 -› [*100*, *100*] *80*)
where
gc →$_k$ *gc'* ≡ *meval-k k gc gc'*

2.2 Low-equivalence and Strong Low Bisimulations

definition *low-eq* :: (*'Var*, *'Val*) *Mem* ⇒ (-, -) *Mem* ⇒ *bool* (**infixl** ‹=^{l}› *80*)
where
mem$_1$ =^{l} *mem*$_2$ ≡ (∀ *x*. *dma x* = *Low* ⟶ *mem*$_1$ *x* = *mem*$_2$ *x*)

definition *low-mds-eq* :: *'Var Mds* ⇒ (*'Var*, *'Val*) *Mem* ⇒ (-, -) *Mem* ⇒ *bool*
(‹- =1^l -› [*100*, *100*] *80*)
where
(*mem*$_1$ =$_{mds}{}^l$ *mem*$_2$) ≡ (∀ *x*. *dma x* = *Low* ∧ *x* ∉ *mds AsmNoRead* ⟶ *mem*$_1$ *x* = *mem*$_2$ *x*)

definition *mds*$_s$ :: *'Var Mds* **where**
mds$_s$ *x* = {}

lemma [*simp*]: *mem* =^{l} *mem'* ⟹ *mem* =$_{mds}{}^l$ *mem'*
by (*simp add*: *low-mds-eq-def low-eq-def*)

lemma [*simp*]: (∀ *mds*. *mem* =$_{mds}{}^l$ *mem'*) ⟹ *mem* =^{l} *mem'*
by (*auto simp*: *low-mds-eq-def low-eq-def*)

definition *closed-glob-consistent* :: ((*'Com*, *'Var*, *'Val*) *LocalConf*) *rel* ⇒ *bool*
where
closed-glob-consistent $\mathcal{R}$ =
(∀ *c*$_1$ *mds mem*$_1$ *c*$_2$ *mem*$_2$. (⟨ *c*$_1$, *mds*, *mem*$_1$ ⟩, ⟨ *c*$_2$, *mds*, *mem*$_2$ ⟩) ∈ $\mathcal{R}$ ⟶
(∀ *x*. ((*dma x* = *High* ∧ *x* ∉ *mds AsmNoWrite*) ⟶
(∀ *v*$_1$ *v*$_2$. (⟨ *c*$_1$, *mds*, *mem*$_1$ (*x* := *v*$_1$) ⟩, ⟨ *c*$_2$, *mds*, *mem*$_2$ (*x* := *v*$_2$) ⟩) ∈ $\mathcal{R}$)) ∧
((*dma x* = *Low* ∧ *x* ∉ *mds AsmNoWrite*) ⟶
(∀ *v*. (⟨ *c*$_1$, *mds*, *mem*$_1$ (*x* := *v*) ⟩, ⟨ *c*$_2$, *mds*, *mem*$_2$ (*x* := *v*) ⟩) ∈ $\mathcal{R}$))))

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

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

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** $\langle\approx\rangle$ *60*)
where
mm-equiv-abv x y $\equiv (x,\ y) \in$ *mm-equiv* |
mm-equiv-intro [*iff*]: $[\![$ *strong-low-bisim-mm* $\mathcal{R}$; $(lc_1,\ lc_2) \in \mathcal{R}\ ]\!] \Longrightarrow (lc_1,\ lc_2) \in$ *mm-equiv*

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

definition *low-indistinguishable* :: $'Var\ Mds \Rightarrow 'Com \Rightarrow 'Com \Rightarrow bool$
$(\langle\text{- }\sim_1\text{ -}\rangle\ [100,\ 100]\ 80)$
where $c_1 \sim_{mds}\ c_2 = (\forall\ mem_1\ mem_2.\ mem_1 =_{mds}{}^{l}\ mem_2 \longrightarrow$
$\langle\ c_1,\ mds,\ mem_1\ \rangle \approx \langle\ c_2,\ mds,\ mem_2\ \rangle)$

2.3 SIFUM-Security

definition *com-sifum-secure* :: $'Com \Rightarrow bool$
where *com-sifum-secure* $c = c \sim_{mds_s}\ c$

definition *add-initial-modes* :: $'Com\ list \Rightarrow ('Com \times 'Var\ Mds)\ list$
where *add-initial-modes cmds* $=$ *zip cmds* (*replicate* (*length cmds*) mds_s)

definition *no-assumptions-on-termination* :: $'Com\ list \Rightarrow bool$
where *no-assumptions-on-termination cmds* $=$
$(\forall\ mem\ mem'\ cms'.$
$(add\text{-}initial\text{-}modes\ cmds,\ mem) \rightarrow^*\ (cms',\ mem') \wedge$
list-all $(\lambda\ c.\ c = stop)$ (*map fst cms'*) $\longrightarrow$
$(\forall\ mds' \in set\ (map\ snd\ cms').\ mds'\ AsmNoRead = \{\} \wedge mds'\ AsmNoWrite = \{\}))$

definition *prog-sifum-secure* :: $'Com\ list \Rightarrow bool$
where *prog-sifum-secure cmds* $=$
$(no\text{-}assumptions\text{-}on\text{-}termination\ cmds \wedge$
$(\forall\ mem_1\ mem_2.\ mem_1 =^{l}\ mem_2 \longrightarrow$
$(\forall\ k\ cms_1'\ mem_1'.$
$(add\text{-}initial\text{-}modes\ cmds,\ mem_1) \rightarrow_k\ (cms_1',\ mem_1') \longrightarrow$
$(\exists\ cms_2'\ mem_2'.\ (add\text{-}initial\text{-}modes\ cmds,\ mem_2) \rightarrow_k\ (cms_2',\ mem_2') \wedge$

$map\ snd\ cms_1' = map\ snd\ cms_2' \wedge$
$length\ cms_2' = length\ cms_1' \wedge$
$(\forall\ x.\ dma\ x = Low \wedge (\forall\ i < length\ cms_1'.$
$x \notin snd\ (cms_1'\ !\ i)\ AsmNoRead) \longrightarrow mem_1'\ x = mem_2'\ x)))))$

2.4 Sound Mode Use

definition *doesnt-read* :: *'Com* ⇒ *'Var* ⇒ *bool*
where
doesnt-read c x = (∀ *mds mem c' mds' mem'*.
⟨ *c*, *mds*, *mem* ⟩ ⇝ ⟨ *c'*, *mds'*, *mem'* ⟩ ⟶
((∀ *v*. ⟨ *c*, *mds*, *mem* (*x* := *v*) ⟩ ⇝ ⟨ *c'*, *mds'*, *mem'* (*x* := *v*) ⟩) ∨
(∀ *v*. ⟨ *c*, *mds*, *mem* (*x* := *v*) ⟩ ⇝ ⟨ *c'*, *mds'*, *mem'* ⟩))))

definition *doesnt-modify* :: *'Com* ⇒ *'Var* ⇒ *bool*
where
doesnt-modify c x = (∀ *mds mem c' mds' mem'*. (⟨ *c*, *mds*, *mem* ⟩ ⇝ ⟨ *c'*, *mds'*, *mem'* ⟩) ⟶
mem x = *mem' x*)

inductive-set *loc-reach* :: (*'Com*, *'Var*, *'Val*) *LocalConf* ⇒ (*'Com*, *'Var*, *'Val*) *LocalConf set*
for *lc* :: (-, -, -) *LocalConf*
where
refl : ⟨*fst* (*fst lc*), *snd* (*fst lc*), *snd lc*⟩ ∈ *loc-reach lc* |
step : ⟦ ⟨*c'*, *mds'*, *mem'*⟩ ∈ *loc-reach lc*;
⟨*c'*, *mds'*, *mem'*⟩ ⇝ ⟨*c''*, *mds''*, *mem''*⟩ ⟧ ⟹
⟨*c''*, *mds''*, *mem''*⟩ ∈ *loc-reach lc* |
mem-diff : ⟦ ⟨ *c'*, *mds'*, *mem'* ⟩ ∈ *loc-reach lc*;
(∀ *x* ∈ *mds' AsmNoWrite*. *mem' x* = *mem'' x*) ⟧ ⟹
⟨ *c'*, *mds'*, *mem''* ⟩ ∈ *loc-reach lc*

definition *locally-sound-mode-use* :: (-, -, -) *LocalConf* ⇒ *bool*
where
locally-sound-mode-use lc =
(∀ *c' mds' mem'*. ⟨ *c'*, *mds'*, *mem'* ⟩ ∈ *loc-reach lc* ⟶
(∀ *x*. (*x* ∈ *mds' GuarNoRead* ⟶ *doesnt-read c' x*) ∧
(*x* ∈ *mds' GuarNoWrite* ⟶ *doesnt-modify c' x*)))

definition *compatible-modes* :: (*'Var Mds*) *list* ⇒ *bool*
where
compatible-modes mdss = (∀ (*i* :: *nat*) *x*. *i* < *length mdss* ⟶
(*x* ∈ (*mdss* ! *i*) *AsmNoRead* ⟶
(∀ *j* < *length mdss*. *j* ≠ *i* ⟶ *x* ∈ (*mdss* ! *j*) *GuarNoRead*)) ∧
(*x* ∈ (*mdss* ! *i*) *AsmNoWrite* ⟶
(∀ *j* < *length mdss*. *j* ≠ *i* ⟶ *x* ∈ (*mdss* ! *j*) *GuarNoWrite*)))

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

list) *set*
where *reachable-mode-states gc* =
{*mdss*. (∃ *cms′ mem′*. *gc* →* (*cms′*, *mem′*) ∧ *map snd cms′* = *mdss*)}

definition *globally-sound-mode-use* :: (*'Com*, *'Var*, *'Val*) *GlobalConf* ⇒ *bool*
where *globally-sound-mode-use gc* =
(∀ *mdss*. *mdss* ∈ *reachable-mode-states gc* ⟶ *compatible-modes mdss*)

primrec *sound-mode-use* :: (-, -, -) *GlobalConf* ⇒ *bool*
where
sound-mode-use (*cms*, *mem*) =
(*list-all* (λ *cm*. *locally-sound-mode-use* (*cm*, *mem*)) *cms* ∧
globally-sound-mode-use (*cms*, *mem*))

lemma *mm-equiv-sym*:
assumes *equivalent*: ⟨c_1, mds_1, mem_1⟩ ≈ ⟨c_2, mds_2, mem_2⟩
shows ⟨c_2, mds_2, mem_2⟩ ≈ ⟨c_1, mds_1, mem_1⟩
proof −
from *equivalent* **obtain** $\mathcal{R}$
where $\mathcal{R}$-*bisim*: *strong-low-bisim-mm* $\mathcal{R}$ ∧ (⟨c_1, mds_1, mem_1⟩, ⟨c_2, mds_2, mem_2⟩) ∈ $\mathcal{R}$
by (*metis mm-equiv.simps*)
hence *sym* $\mathcal{R}$
by (*auto simp*: *strong-low-bisim-mm-def*)
hence (⟨c_2, mds_2, mem_2⟩, ⟨c_1, mds_1, mem_1⟩) ∈ $\mathcal{R}$
by (*metis* $\mathcal{R}$-*bisim symE*)
thus *?thesis*
by (*metis* $\mathcal{R}$-*bisim mm-equiv.intros*)
qed

lemma *low-indistinguishable-sym*: *lc* $\sim_{mds}$ *lc′* ⟹ *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 ⟨ c_1, *mds*, mem_1 ⟩ ≈ ⟨ c_2, *mds*, mem_2 ⟩
then obtain $\mathcal{R}$ **where**
strong-low-bisim-mm $\mathcal{R}$ ∧ (⟨ c_1, *mds*, mem_1 ⟩, ⟨ c_2, *mds*, mem_2 ⟩) ∈ $\mathcal{R}$

by *blast*
thus $mem_1 =_{mds}{}^{l} mem_2$ **by** (*auto simp*: *strong-low-bisim-mm-def*)
next
fix c_1 :: *'Com*
fix *mds* mem_1 c_2 mem_2 $c_1{}'$ *mds'* $mem_1{}'$
let *?lc*$_1$ = ⟨ c_1, *mds*, mem_1 ⟩ **and**
?lc$_1{}'$ = ⟨ $c_1{}'$, *mds'*, $mem_1{}'$ ⟩ **and**
?lc$_2$ = ⟨ c_2, *mds*, mem_2 ⟩
assume *?lc*$_1$ ≈ *?lc*$_2$
then obtain $\mathcal{R}$ **where** *strong-low-bisim-mm* $\mathcal{R}$ ∧ (*?lc*$_1$, *?lc*$_2$) ∈ $\mathcal{R}$
by (*rule mm-equiv-elim*, *blast*)
moreover assume *?lc*$_1$ ⇝ *?lc*$_1{}'$
ultimately show ∃ $c_2{}'$ $mem_2{}'$. *?lc*$_2$ ⇝ ⟨ $c_2{}'$, *mds'*, $mem_2{}'$ ⟩ ∧ *?lc*$_1{}'$ ≈ ⟨ $c_2{}'$, *mds'*, $mem_2{}'$ ⟩
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* ⇒ (-, -) *Mem* ⇒ *'Var set*
where
differing-vars mem_1 mem_2 = {x. mem_1 x ≠ mem_2 x}

definition *differing-vars-lists* :: (*'Var*, *'Val*) *Mem* ⇒ (-, -) *Mem* ⇒
((-, -) *Mem* × (-, -) *Mem*) *list* ⇒ *nat* ⇒ *'Var set*
where
differing-vars-lists mem_1 mem_2 *mems* i =
(*differing-vars* mem_1 (*fst* (*mems* ! i)) ∪ *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* mem_1 mem_2 *mems i*)
by (*simp add*: *differing-finite differing-vars-lists-def*)

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

abbreviation *subst-abv* :: $('a \Rightarrow 'b) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \Rightarrow 'b)$ ($\langle$- [$\mapsto$-]$\rangle$ [900, 0] 1000)
where
$f\ [\mapsto \sigma] \equiv subst\ \sigma\ f$

lemma *subst-not-in-dom* : $[\![\ x \notin dom\ \sigma\ ]\!] \Longrightarrow mem\ [\mapsto \sigma]\ x = mem\ x$
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 (cms_1, mem_1) (cms_2, mem_2) *mems* =
(*length* cms_1 = *length* cms_2 $\wedge$ *length* cms_1 = *length mems* $\wedge$
($\forall$ *i*. *i* < *length* cms_1 $\longrightarrow$
($\forall$ σ. *dom* σ = *differing-vars-lists* mem_1 mem_2 *mems i* $\longrightarrow$
(cms_1 ! *i*, (*fst* (*mems* ! *i*)) [$\mapsto$ σ]) $\approx$ (cms_2 ! *i*, (*snd* (*mems* ! *i*)) [$\mapsto$ σ])) $\wedge$
($\forall$ *x*. (mem_1 *x* = mem_2 *x* $\vee$ *dma x* = *High*) $\longrightarrow$
x $\notin$ *differing-vars-lists* mem_1 mem_2 *mems i*)) $\wedge$
((*length* cms_1 = 0 $\wedge$ mem_1 =^{l} mem_2) $\vee$ ($\forall$ *x*. $\exists$ *i*. *i* < *length* cms_1 $\wedge$
x $\notin$ *differing-vars-lists* mem_1 mem_2 *mems i*)))

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

lemma *compat-low*:
⟦ *makes-compatible* (*cms*$_1$, *mem*$_1$) (*cms*$_2$, *mem*$_2$) *mems*;
i < *length cms*$_1$;
x ∈ *differing-vars-lists mem*$_1$ *mem*$_2$ *mems i* ⟧ ⟹ *dma x* = *Low*
proof −
assume *i* < *length cms*$_1$ **and** ∗: *x* ∈ *differing-vars-lists mem*$_1$ *mem*$_2$ *mems i* **and**
makes-compatible (*cms*$_1$, *mem*$_1$) (*cms*$_2$, *mem*$_2$) *mems*
then have
(*mem*$_1$ *x* = *mem*$_2$ *x* ∨ *dma x* = *High*) ⟶ *x* ∉ *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*:
⟦ *makes-compatible* (*cms*$_1$, *mem*$_1$) (*cms*$_2$, *mem*$_2$) *mems*;
i < *length cms*$_1$;
x ∈ *differing-vars-lists mem*$_1$ *mem*$_2$ *mems i* ⟧ ⟹ *mem*$_1$ *x* ≠ *mem*$_2$ *x* ∧ *dma x* = *Low*
by (*cases dma x*, *auto*)

lemma *sound-modes-no-read* :
⟦ *sound-mode-use* (*cms*, *mem*); *x* ∈ (*map snd cms* ! *i*) *GuarNoRead*; *i* < *length cms* ⟧ ⟹
doesnt-read (*fst* (*cms* ! *i*)) *x*
proof −
fix *cms mem x i*
assume *sound-modes*: *sound-mode-use* (*cms*, *mem*) **and** *i* < *length cms*
hence *locally-sound-mode-use* (*cms* ! *i*, *mem*)
by (*auto simp*: *sound-mode-use-def list-all-length*)
moreover
assume *x* ∈ (*map snd cms* ! *i*) *GuarNoRead*
ultimately show *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*:
⟦ *fst* (*mems* ! *i*) *x* = *snd* (*mems* ! *i*) *x*;
x ∉ *differing-vars-lists mem*$_1$ *mem*$_2$ *mems i* ⟧ ⟹
mem$_1$ *x* = *mem*$_2$ *x*
proof −
assume *x* ∉ *differing-vars-lists mem*$_1$ *mem*$_2$ *mems i*
hence *fst* (*mems* ! *i*) *x* = *mem*$_1$ *x* ∧ *snd* (*mems* ! *i*) *x* = *mem*$_2$ *x*
by (*simp add*: *differing-vars-lists-def differing-vars-def*)
moreover assume *fst* (*mems* ! *i*) *x* = *snd* (*mems* ! *i*) *x*
ultimately show *mem*$_1$ *x* = *mem*$_2$ *x* **by** *auto*
qed

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

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

lemma *globally-sound-modes-compatible*:
$[\![$ *globally-sound-mode-use* (*cms*, *mem*) $]\!] \Longrightarrow$ *compatible-modes* (*map 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* $cms_1 =$ *map snd* cms_2
assumes *ile*: $i <$ *length* cms_1
assumes x: $x \in$ *differing-vars-lists* mem_1 mem_2 *mems* i
shows *doesnt-read* (*fst* (cms_1 ! i)) $x \wedge$ *doesnt-read* (*fst* (cms_2 ! i)) x
proof −
from *compat* **have** *len*: *length* $cms_1 =$ *length* cms_2
by *simp*

let $?X_i =$ *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: $j <$ *length* $cms_1 \wedge x \notin$ *differing-vars-lists* mem_1 mem_2 *mems* j
by *fastforce*

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

```
  show dom ?σ = ?X_j unfolding dom-def by auto
qed
let ?mdss = map snd cms_1 and
    ?mems_1j = fst (mems ! j) and
    ?mems_2j = snd (mems ! j)

from jprop domσ have subst-eq:
?mems_1j [↦ σ] x = ?mems_1j x ∧ ?mems_2j [↦ σ] x = ?mems_2j x
by (metis subst-not-in-dom)

from compat jprop domσ
have (cms_1 ! j, ?mems_1j [↦ σ]) ≈ (cms_2 ! j, ?mems_2j [↦ σ])
  by (auto simp: Let-def)

hence low-eq: ?mems_1j [↦ σ] =_{?mdss ! j}^{l} ?mems_2j [↦ 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: ?mems_1j x = ?mems_2j x
      using ‹dma x = Low› low-eq subst-eq
      by (metis (full-types) low-mds-eq-def subst-eq)

    hence mem_1 x = mem_2 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 (cms_1 ! i)) x ∧ doesnt-read (fst (cms_2 ! 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 ⇒
```

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

lemma *change-respecting-dom-unique*:
$[\![$ *change-respecting* $\langle c, \mathit{mds}, \mathit{mem}\rangle$ $\langle c', \mathit{mds'}, \mathit{mem'}\rangle$ X g $]\!] \Longrightarrow$
$\exists\ d.\ \forall\ f.\ \mathit{dom}\ f = X \longrightarrow d = \mathit{dom}\ (g\ f)$
by (*metis change-respecting.simps*)

lemma *func-le-restrict*: $[\![\ f \preceq g;\ X \subseteq \mathit{dom}\ f\ ]\!] \Longrightarrow f\ |`\ X \preceq g$
by (*auto simp*: *func-le-def*)

definition *to-partial* :: $('a \Rightarrow 'b) \Rightarrow ('a \rightharpoonup 'b)$
where *to-partial* $f = (\lambda\ x.\ \mathit{Some}\ (f\ x))$

lemma *func-le-dom*: $f \preceq g \Longrightarrow \mathit{dom}\ f \subseteq \mathit{dom}\ g$
by (*auto simp add*: *func-le-def*, *metis domIff option.simps*(*2*))

lemma *doesnt-read-mutually-exclusive*:
assumes *noread*: *doesnt-read c x*
assumes *eval*: $\langle c, \mathit{mds}, \mathit{mem}\rangle \rightsquigarrow \langle c', \mathit{mds'}, \mathit{mem'}\rangle$
assumes *unchanged*: $\forall\ v.\ \langle c, \mathit{mds}, \mathit{mem}\ (x := v)\rangle \rightsquigarrow \langle c', \mathit{mds'}, \mathit{mem'}\ (x := v)\rangle$
shows $\neg\ (\forall\ v.\ \langle c, \mathit{mds}, \mathit{mem}\ (x := v)\rangle \rightsquigarrow \langle c', \mathit{mds'}, \mathit{mem'}\rangle)$
using *assms*
apply (*case-tac mem' x = some-val*)
apply (*metis* (*full-types*) *prod.inject deterministic different-values fun-upd-same*)
by (*metis* (*full-types*) *prod.inject deterministic fun-upd-same*)

lemma *doesnt-read-mutually-exclusive'*:
assumes *noread*: *doesnt-read c x*
assumes *eval*: $\langle c, \mathit{mds}, \mathit{mem}\rangle \rightsquigarrow \langle c', \mathit{mds'}, \mathit{mem'}\rangle$
assumes *overwrite*: $\forall\ v.\ \langle c, \mathit{mds}, \mathit{mem}\ (x := v)\rangle \rightsquigarrow \langle c', \mathit{mds'}, \mathit{mem'}\rangle$
shows $\neg\ (\forall\ v.\ \langle c, \mathit{mds}, \mathit{mem}\ (x := v)\rangle \rightsquigarrow \langle c', \mathit{mds'}, \mathit{mem'}\ (x := v)\rangle)$
by (*metis assms doesnt-read-mutually-exclusive*)

lemma *change-respecting-dom*:
assumes *cr*: *change-respecting* $(\mathit{cms}, \mathit{mem})$ $(\mathit{cms'}, \mathit{mem'})$ X g
assumes *domσ*: $\mathit{dom}\ \sigma = X$
shows $\mathit{dom}\ (g\ \sigma) \subseteq X$
by (*metis assms change-respecting.simps func-le-dom*)

lemma *change-respecting-intro* [*iff*]:
$[\![\ \langle\ c, \mathit{mds}, \mathit{mem}\ \rangle \rightsquigarrow \langle\ c', \mathit{mds'}, \mathit{mem'}\ \rangle;$
$\bigwedge f.\ \mathit{dom}\ f = X \Longrightarrow$
$g\ f \preceq f \wedge$

```
        (∀ f'. dom f' = X ⟶ dom (g f) = dom (g f')) ∧
        (⟨ c, mds, mem [↦] f ] ⟩ ⇝ ⟨ c', mds', mem' [↦] g f ] ⟩) ⟧
  ⟹ change-respecting ⟨c, mds, mem⟩ ⟨c', mds', mem'⟩ X g
  unfolding change-respecting.simps
  by blast


lemma conjI3: ⟦ A; B; C ⟧ ⟹ A ∧ B ∧ C
  by simp

lemma noread-exists-change-respecting:
  assumes fin: finite (X :: 'Var set)
  assumes eval: ⟨ c, mds, mem ⟩ ⇝ ⟨ c', mds', mem' ⟩
  assumes noread: ∀ x ∈ X. doesnt-read c x
  shows ∃ (g :: ('Var ⇀ 'Val) ⇒ ('Var ⇀ 'Val)). change-respecting ⟨c, mds,
mem⟩ ⟨c', mds', mem'⟩ X g
proof −
  let ?lc = ⟨c, mds, mem⟩ and ?lc' = ⟨c', mds', mem'⟩
  from fin eval noread show ∃ g. change-respecting ⟨c, mds, mem⟩ ⟨c', mds',
mem'⟩ X g
  proof (induct X arbitrary: mem mem' rule: finite-induct)
    case empty
    let ?g = λ σ. Map.empty
    have mem [↦] Map.empty] = mem mem' [↦] ?g Map.empty] = mem'
      unfolding subst-def
      by auto
    hence change-respecting ⟨c, mds, mem⟩ ⟨c', mds', mem'⟩ {} ?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 ⟨c, mds, mem⟩ ⟨c', mds', mem'⟩
X g_X
      by (metis insert-iff)

    define g where g σ =
      (let σ' = σ |` X in
        (if (∀ v. ⟨c, mds, mem [↦] σ'] (x := v)⟩ ⇝ ⟨c', mds', mem' [↦] g_X σ'] (x
:= v))
        then (λ y :: 'Var.
                if x = y
                then σ y
                else g_X σ' y)
        else (λ y. g_X σ' y)))
      for σ :: 'Var ⇀ 'Val
    have change-respecting ⟨c, mds, mem⟩ ⟨c', mds', mem'⟩ (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 $\sigma :: 'Var \rightharpoonup 'Val$
let $?\sigma_X = \sigma\ |`\ X$
assume $dom\ \sigma = insert\ x\ X$
hence $dom\ ?\sigma_X = X$
by (*metis dom-restrict inf-absorb2 subset-insertI*)
from *insert* **have** *doesnt-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 ‹$dom\ ?\sigma_X = X$›
unfolding *change-respecting.simps*
by *auto*
ultimately have
$noread_x$:
$(\forall\ v.\ \langle c,\ mds,\ mem\ [\mapsto\ ?\sigma_X]\ (x := v)\rangle \rightsquigarrow \langle c',\ mds',\ mem'\ [\mapsto\ g_X\ ?\sigma_X]\ (x := v)\ \rangle) \vee$
$(\forall\ v.\ \langle c,\ mds,\ mem\ [\mapsto\ ?\sigma_X]\ (x := v)\rangle \rightsquigarrow \langle c',\ mds',\ mem'\ [\mapsto\ g_X\ ?\sigma_X]\rangle)$
unfolding *doesnt-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 $noread_x$ **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* ‹$dom\ (\sigma\ |`\ X) = X$› *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* ‹*doesnt-read c x*› *doesnt-read-mutually-exclusive* $eval_X$)
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* ‹$dom\ (\sigma\ |`\ X) = X$› *change-respecting.simps*)

ultimately show $g\ \sigma \preceq \sigma$
unfolding *func-le-def*
by (*auto*, *metis* ‹*dom* $(\sigma\ |`\ X) = X$› *domI restrict-in*)
qed
next — This part proves that the domain of the family is unique
{
fix $\sigma' :: 'Var \rightharpoonup 'Val$
assume *dom* $\sigma' =$ *insert* $x\ X$
let $?\sigma'_X = \sigma'\ |`\ X$
have *dom* $?\sigma'_X = X$
by (*metis* ‹*dom* $(\sigma\ |`\ X) = X$› ‹*dom* $\sigma =$ *insert* $x\ X$› ‹*dom* $\sigma' =$ *insert* $x\ X$› *dom-restrict*)
— We first show, that we are always in the same case of the no read assumption:
have *same-case*:
$((\forall\ v.\ \langle c,\ mds,\ mem\ [\mapsto\ ?\sigma_X]\ (x := v)\rangle \rightsquigarrow \langle c',\ mds',\ mem'\ [\mapsto\ g_X\ \ ?\sigma_X]\ (x := v)\rangle) \wedge$
$(\forall\ v.\ \langle c,\ mds,\ mem\ [\mapsto\ ?\sigma'_X]\ (x := v)\rangle \rightsquigarrow \langle c',\ mds',\ mem'\ [\mapsto\ g_X\ \ ?\sigma'_X]\ (x := v)\rangle))$
$\vee$
$((\forall\ v.\ \langle c,\ mds,\ mem\ [\mapsto\ ?\sigma_X]\ (x := v)\rangle \rightsquigarrow \langle c',\ mds',\ mem'\ [\mapsto\ g_X\ \ ?\sigma_X]\rangle) \wedge$
$(\forall\ v.\ \langle c,\ mds,\ mem\ [\mapsto\ ?\sigma'_X]\ (x := v)\rangle \rightsquigarrow \langle c',\ mds',\ mem'\ [\mapsto\ g_X\ \ ?\sigma'_X]\rangle))$
(**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*:
$\bigwedge h\ h'.\ [\![\ dom\ h = insert\ x\ X;\ dom\ h' = insert\ x\ X;$
$\forall\ v.\ \langle c,\ mds,\ mem\ [\mapsto\ h\ |`\ X]\ (x := v)\rangle \rightsquigarrow \langle c',\ mds',\ mem'\ [\mapsto\ g_X\ (h\ |`\ X)]\ (x := v)\rangle;$
$\forall\ v.\ \langle c,\ mds,\ mem\ [\mapsto\ h'\ |`\ X]\ (x := v)\rangle \rightsquigarrow \langle c',\ mds',\ mem'\ [\mapsto\ g_X\ (h'\ |`\ X)]\rangle\ ]\!]$
$\Longrightarrow$ *False*
proof −
— Introduce new names to avoid clashes with functions in the outer scope.
fix $h\ h' :: 'Var \rightharpoonup 'Val$
assume *doms*: *dom* $h =$ *insert* $x\ X$ *dom* $h' =$ *insert* $x\ X$
assume *nowrite*: $\forall\ v.\ \langle c,\ mds,\ mem\ [\mapsto\ h\ |`\ X]\ (x := v)\rangle \rightsquigarrow \langle c',\ mds',\ mem'\ [\mapsto\ g_X\ (h\ |`\ X)]\ (x := v)\rangle$
assume *overwrite*: $\forall\ v.\ \langle c,\ mds,\ mem\ [\mapsto\ h'\ |`\ X]\ (x := v)\rangle \rightsquigarrow \langle c',\ mds',\ mem'\ [\mapsto\ g_X\ (h'\ |`\ X)]\rangle$

let $?h_X = h\ |`\ X$
let $?h'_X = h'\ |`\ X$

have *dom* $?h_X = X$
by (*metis* ‹*dom* $(\sigma\ |`\ X) = X$› ‹*dom* $\sigma =$ *insert* $x\ X$› *dom-restrict*

doms(1))
have *dom* $?h'_X = X$
by (*metis* ‹*dom* $(\sigma \mid` X) = X$› ‹*dom* σ = *insert* x X› *dom-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-respecting.simps*
by *auto*
with ‹*doesnt-read c x*› **have** $noread_x'$:
$(\forall\ v.\ \langle c, mds, mem\ [\mapsto\ ?h'_X]\ (x := v)\rangle \rightsquigarrow \langle c', mds', mem'\ [\mapsto\ g_X\ \ ?h'_X]\ (x := v)\rangle) \vee$
$(\forall\ v.\ \langle c, mds, mem\ [\mapsto\ ?h'_X]\ (x := v)\rangle \rightsquigarrow \langle c', mds', mem'\ [\mapsto\ g_X\ \ ?h'_X]\rangle)$
unfolding *doesnt-read-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* ‹*doesnt-read c x*› *doesnt-read-mutually-exclusive fun-upd-triv*)
moreover

have $x \notin dom\ (?h'_X)$
by (*metis* ‹*dom* $(h' \mid` X) = X$› *insert(2)*)
with *IH* **have** $x \notin dom\ (g_X\ \ ?h'_X)$
by (*metis* ‹*dom* $(h' \mid` X) = X$› *change-respecting.simps func-le-dom rev-subsetD*)

ultimately have $mem'\ x \neq v$
by (*metis fun-upd-triv overwrite subst-not-in-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* ‹*doesnt-read c x*›
unfolding *doesnt-read-def*
by (*auto*, *metis*)
also have $\forall\ x \in X.\ doesnt\text{-}read\ c\ x$
by (*metis insert(5) insert-iff*)

ultimately obtain g_v **where**
IH_v: *change-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* ‹*dom* $(h \mid` X) = X$› *change-respecting.simps*)
by (*metis* IH_v ‹*dom* $(h' \mid` X) = X$› *change-respecting.simps*)

```
    from eval_v(1) have mem_v' x = v
    proof −
      assume ⟨c, mds, mem (x := v) [↦ ?h_X]⟩ ⇝ ⟨c', mds', mem_v' [↦ g_v
?h_X]⟩
      have ?mem_v [↦ ?h_X] = mem [↦ ?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' [↦ g_v ?h_X] = mem' [↦ g_X ?h_X] (x := v)
        using deterministic
        by (erule-tac x = v in allE, auto, metis eval_v(1))

      hence mem_v' [↦ g_v ?h_X] x = v
        by simp
      also have x ∉ dom (g_v ?h_X)
        using IH_v ‹dom ?h_X = X› 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 ⟨c, mds, ?mem_v [↦ ?h'_X]⟩ ⇝ ⟨c', mds', mem_v' [↦ g_v ?h'_X]⟩
      moreover
      from overwrite have
        ⟨c, mds, mem [↦ ?h'_X] (x := v)⟩ ⇝ ⟨c', mds', mem' [↦ g_X ?h'_X]⟩
        by auto
      moreover
      have ?mem_v [↦ ?h'_X] = mem [↦ ?h'_X] (x := v)
        apply (rule ext, rename-tac y)
        apply (case-tac y = x)
         apply (metis ‹x ∉ dom (h' |` X)› fun-upd-apply subst-not-in-dom)
        apply (auto simp: subst-def)
        by (metis fun-upd-other)
      ultimately have mem' [↦ g_X ?h'_X] = mem_v' [↦ g_v ?h'_X]
        using deterministic
        by auto
      also have x ∉ dom (g_v ?h'_X)
        using IH_v ‹dom ?h'_X = X› change-respecting-dom
        by (metis func-le-dom insert(2) subsetD)
      ultimately show mem_v' x = mem' x
        using ‹x ∉ dom (g_X ?h'_X)›
        by (metis subst-not-in-dom)
```

```
          qed
          ultimately show False
            using ‹mem' x ≠ v›
            by auto
        qed

        moreover
        have dom ?σ'_X = X
          by (metis ‹dom (σ |` X) = X› ‹dom σ = insert x X› ‹dom σ' = insert
x X› dom-restrict)

        with IH have eval_X': ⟨c, mds, mem [↦ ?σ'_X]⟩ ⇝ ⟨c', mds', mem' [↦ g_X
?σ'_X]⟩
          unfolding change-respecting.simps
          by auto
        with ‹doesnt-read c x› have noread_x':
          (∀ v. ⟨c, mds, mem [↦ ?σ'_X] (x := v)⟩ ⇝ ⟨c', mds', mem' [↦ g_X ?σ'_X]
(x := v)⟩)
            ∨
              (∀ v. ⟨c, mds, mem [↦ ?σ'_X] (x := v)⟩ ⇝ ⟨c', mds', mem' [↦ g_X
?σ'_X]⟩)
          unfolding doesnt-read-def
          by auto

        ultimately show ?thesis
         using noread_x not-different ‹dom σ = insert x X› ‹dom σ' = insert x X›
          by auto
      qed
      hence dom (g σ) = dom (g σ')
      proof
        assume
          (∀ v. ⟨c, mds, mem [↦ ?σ_X] (x := v)⟩ ⇝ ⟨c', mds', mem' [↦ g_X ?σ_X]
(x := v)⟩) ∧
          (∀ v. ⟨c, mds, mem [↦ ?σ'_X] (x := v)⟩ ⇝ ⟨c', mds', mem' [↦ g_X ?σ'_X]
(x := v)⟩)
        hence g-simp [simp]: g σ = (λ y. if y = x then σ y else g_X ?σ_X y) ∧
                             g σ' = (λ y. if y = x then σ' y else g_X ?σ'_X y)
          unfolding g-def
          by auto
        thus ?thesis
          using IH ‹dom σ = insert x X› ‹dom σ' = insert x X›
          unfolding change-respecting.simps
          apply (auto simp: domD)
           apply (metis ‹dom (σ |` X) = X› ‹dom (σ' |` X) = X› domD domI)
          by (metis ‹dom (σ |` X) = X› ‹dom (σ' |` X) = X› domD domI)
      next
        assume
          (∀ v. ⟨c, mds, mem [↦ ?σ_X] (x := v)⟩ ⇝ ⟨c', mds', mem' [↦ g_X ?σ_X]⟩)
∧
```

```
      (∀ v. ⟨c, mds, mem [↦ ?σ'_X] (x := v)⟩ ⇝ ⟨c', mds', mem' [↦ g_X ?σ'_X]⟩)
     hence
      ¬ (∀ v. ⟨c, mds, mem [↦ ?σ_X] (x := v)⟩ ⇝ ⟨c', mds', mem' [↦ g_X ?σ_X]
(x := v)⟩)
       ∧
        ¬ (∀ v. ⟨c, mds, mem [↦ ?σ'_X] (x := v)⟩ ⇝ ⟨c', mds', mem' [↦ g_X
?σ'_X] (x := v)⟩)
       by (metis ‹doesnt-read c x› doesnt-read-mutually-exclusive' fun-upd-triv)

     hence g-simp [simp]: g σ = g_X ?σ_X ∧ g σ' = g_X ?σ'_X
      unfolding g-def
      by (auto simp: Let-def)
     with IH show ?thesis
      unfolding change-respecting.simps
      by (metis ‹dom (σ |` X) = X› ‹dom (σ' |` X) = X›)
    qed
   }
   thus ∀ σ'. dom σ' = insert x X ⟶ dom (g σ) = dom (g σ') by blast
  next

   from noread_x show ⟨c, mds, mem [↦ σ]⟩ ⇝ ⟨c', mds', mem' [↦ g σ]⟩
   proof
    assume nowrite:
     ∀ v. ⟨c, mds, mem[↦ ?σ_X] (x := v)⟩ ⇝ ⟨c', mds', mem' [↦ g_X ?σ_X] (x
:= v)⟩
    then have g-simp [simp]: g σ = (λ y. if y = x then σ y else g_X ?σ_X y)
     unfolding g-def
     by auto
    obtain v where σ x = Some v
     by (metis ‹dom σ = insert x X› domD insertI1)

    from nowrite have
     ⟨c, mds, mem [↦ ?σ_X] (x := v)⟩ ⇝ ⟨c', mds', mem' [↦ g_X ?σ_X] (x := v)⟩
     by auto
    moreover
    have mem [↦ ?σ_X] (x := v) = mem [↦ σ]
     apply (rule ext, rename-tac y)
     apply (case-tac y = x)
      apply (auto simp: subst-def)
      apply (metis ‹σ x = Some v› option.simps(5))
     by (metis ‹dom (σ |` X) = X› ‹dom σ = insert x X› insertE
       restrict-in subst-def subst-not-in-dom)
    moreover
    have mem' [↦ g_X ?σ_X] (x := v) = mem' [↦ g σ]
     apply (rule ext, rename-tac y)
     apply (case-tac y = x)
      by (auto simp: subst-def option.simps ‹σ x = Some v›)
    ultimately show ?thesis
     by auto
```

```
      next
        assume overwrites:
          ∀ v. ⟨c, mds, mem [↦ ?σ_X] (x := v)⟩ ⇝ ⟨c', mds', mem' [↦ g_X ?σ_X]⟩
        hence
          ¬ (∀ v. ⟨c, mds, mem [↦ ?σ_X] (x := v)⟩ ⇝ ⟨c', mds', mem' [↦ g_X ?σ_X]
(x := v)⟩)
          by (metis ‹doesnt-read c x› doesnt-read-mutually-exclusive' eval_X)
        hence g-simp [simp]: g σ = g_X ?σ_X
          unfolding g-def
          by (auto simp: Let-def)
        obtain v where σ x = Some v
          by (metis ‹dom σ = insert x X› domD insertI1)
        have mem [↦ ?σ_X] (x := v) = mem [↦ σ]
          apply (rule ext, rename-tac y)
          apply (case-tac y = x)
           apply (auto simp: subst-def)
           apply (metis ‹σ x = Some v› option.simps(5))
          by (metis ‹dom (σ |` X) = X› ‹dom σ = insert x X› insertE
            restrict-in subst-def subst-not-in-dom)
        moreover
        from overwrites have ⟨c, mds, mem [↦ ?σ_X] (x := v)⟩ ⇝ ⟨c', mds', mem'
[↦ g σ]⟩
          by (metis g-simp)
        ultimately show ⟨c, mds, mem [↦ σ]⟩ ⇝ ⟨c', mds', mem' [↦ g σ]⟩
          by auto
        qed
      qed
    qed
    thus ∃ g. change-respecting ⟨c, mds, mem⟩ ⟨c', mds', mem'⟩ (insert x X) g
      by metis
  qed
qed

lemma differing-vars-neg: x ∉ differing-vars-lists mem1 mem2 mems i ⟹
  (fst (mems ! i) x = mem1 x ∧ snd (mems ! i) x = mem2 x)
  by (simp add: differing-vars-lists-def differing-vars-def)

lemma differing-vars-neg-intro:
  ⟦ mem_1 x = fst (mems ! i) x;
  mem_2 x = snd (mems ! i) x ⟧ ⟹ x ∉ 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 ∈ differing-vars-lists mem_1 mem_2 mems i ⟹
  (fst (mems ! i) x ≠ mem_1 x) ∨ (snd (mems ! i) x ≠ mem_2 x)
  by (auto simp: differing-vars-lists-def differing-vars-def)

lemma subst-overrides: dom σ = dom τ ⟹ mem [↦ τ] [↦ σ] = mem [↦ σ]
  unfolding subst-def
```

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

lemma *dom-restrict-total*: *dom* (*to-partial f* |‘ *X*) = *X*
unfolding *to-partial-def*
by (*metis Int-UNIV-left dom-const dom-restrict*)

lemma *update-nth-eq*:
⟦ *xs* = *ys*; *n* < *length xs* ⟧ ⟹ *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*: (*cms*$_1$, *mem*$_1$) ≈ (*cms*$_2$, *mem*$_2$)
assumes *modes-eq*: *snd cms*$_1$ = *snd cms*$_2$
assumes *step*: (*cms*$_1$, *mem*$_1$) ⇝ (*cms*$_1$′, *mem*$_1$′)
shows ∃ *c*$_2$′ *mem*$_2$′. (*cms*$_2$, *mem*$_2$) ⇝ ⟨ *c*$_2$′, *snd cms*$_1$′, *mem*$_2$′ ⟩ ∧
(*cms*$_1$′, *mem*$_1$′) ≈ ⟨ *c*$_2$′, *snd cms*$_1$′, *mem*$_2$′ ⟩
using *assms mm-equiv-strong-low-bisim*
unfolding *strong-low-bisim-mm-def*
apply *auto*
apply (*erule-tac x* = *fst cms*$_1$ **in** *allE*)
apply (*erule-tac x* = *snd cms*$_1$ **in** *allE*)
by (*metis surjective-pairing*)

lemma *change-respecting-doesnt-modify*:
assumes *cr*: *change-respecting* (*cms*, *mem*) (*cms*′, *mem*′) *X g*
assumes *eval*: (*cms*, *mem*) ⇝ (*cms*′, *mem*′)
assumes *domf*: *dom f* = *X*
assumes *x-in-dom*: *x* ∈ *dom* (*g f*)
assumes *noread*: *doesnt-read* (*fst cms*) *x*
shows *mem x* = *mem*′ *x*
proof −
let *?f*′ = *to-partial mem* |‘ *X*
have *domf*′: *dom ?f*′ = *X*
by (*metis dom-restrict-total*)

from *cr* **and** *eval* **have** ∀ *f*. *dom f* = *X* ⟶ (*cms*, *mem* [↦ *f*]) ⇝ (*cms*′, *mem*′ [↦ *g f*])
unfolding *change-respecting.simps*
by *metis*
hence *eval*′: (*cms*, *mem* [↦ *?f*′]) ⇝ (*cms*′, *mem*′ [↦ *g ?f*′])
by (*metis domf*′)

have *mem-eq*: *mem* [↦ *?f*′] = *mem*
proof
fix *x*
show *mem* [↦ *?f*′] *x* = *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* :: $('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'a \rightharpoonup 'b$
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 $\langle c_1,\ mds,\ mem_1\ [\|_1\ A]\rangle \approx \langle c_2,\ mds,\ mem_2\ [\|_2\ A]\rangle$
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 $mem_1\ [\|_1\ A] = mem_1$ **and** $mem_2\ [\|_2\ A] = mem_2$
unfolding *apply-adaptation-def*
by *auto*
with *bisim* **show** *?case*
by *auto*
next
case (*insert x X*)
define A' **where** $A' = A\ |`\ X$
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'*: $\langle c_1,\ mds,\ mem_1\ [\|_1\ A']\rangle \approx \langle c_2,\ mds,\ mem_2\ [\|_2\ A']\rangle$
using *insert*
by *metis*
with *insert* **have** *writable*: $x \notin mds\ AsmNoWrite$
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': $\bigwedge y.\ y \neq x \Longrightarrow A\ y = A'\ y$
unfolding A'*-def*
by (*metis domIff insert*(*4*) *insert-iff restrict-in restrict-out*)

have eq_1: $mem_1\ [\|_1\ A']\ (x := v) = mem_1\ [\|_1\ A]$
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')

```
    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.prems)
      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: $k < \mathit{length}\ \mathit{cms}_1 \wedge (\mathit{cms}_1 ! k, \mathit{mem}_1) \rightsquigarrow (\mathit{cms}_1' ! k, \mathit{mem}_1') \wedge$
$\mathit{cms}_1' = \mathit{cms}_1\ [k := \mathit{cms}_1' ! k]$
by (*metis meval-elim nth-list-update-eq*)

from *modes-eq* **have** *equal-size*: $\mathit{length}\ \mathit{cms}_1 = \mathit{length}\ \mathit{cms}_2$
by (*metis length-map*)

let $?\mathit{mds}_k = \mathit{snd}\ (\mathit{cms}_1 ! k)$ **and**
$?\mathit{mds}_k' = \mathit{snd}\ (\mathit{cms}_1' ! k)$ **and**
$?\mathit{mems}_1 k = \mathit{fst}\ (\mathit{mems} ! k)$ **and**
$?\mathit{mems}_2 k = \mathit{snd}\ (\mathit{mems} ! k)$ **and**
$?n = \mathit{length}\ \mathit{cms}_1$

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

then obtain *g1* **where**
c: *change-respecting* $(\mathit{cms}_1 ! k, \mathit{mem}_1)\ (\mathit{cms}_1' ! k, \mathit{mem}_1')\ (?X\ k)$ *g1*
using *noread-exists-change-respecting b a*
by (*metis surjective-pairing*)

from *compat* **have** $\bigwedge \sigma.\ \mathit{dom}\ \sigma = ?X\ k \Longrightarrow ?\mathit{mems}_1 k\ [\mapsto \sigma] = \mathit{mem}_1\ [\mapsto \sigma]$
by (*metis* (*no-types*) *Un-upper1 differing-vars-lists-def differing-vars-subst*)

with b **and** c **have**
eval_σ: $\bigwedge \sigma.\ \mathit{dom}\ \sigma = ?X\ k \Longrightarrow (\mathit{cms}_1 ! k, ?\mathit{mems}_1 k\ [\mapsto \sigma]) \rightsquigarrow (\mathit{cms}_1' ! k, \mathit{mem}_1'\ [\mapsto g1\ \sigma])$
by *auto*

moreover
with b **and** *compat* **have**
bisim_σ: $\bigwedge \sigma.\ \mathit{dom}\ \sigma = ?X\ k \Longrightarrow (\mathit{cms}_1 ! k, ?\mathit{mems}_1 k\ [\mapsto \sigma]) \approx (\mathit{cms}_2 ! k, ?\mathit{mems}_2 k\ [\mapsto \sigma])$
by *auto*

moreover have $\mathit{snd}\ (\mathit{cms}_1 ! k) = \mathit{snd}\ (\mathit{cms}_2 ! k)$
by (*metis b equal-size modes-eq nth-map*)

ultimately have d: $\bigwedge \sigma.\ \mathit{dom}\ \sigma = ?X\ k \Longrightarrow \exists\ c_f'\ \mathit{mem}_f'.$
$(\mathit{cms}_2 ! k, ?\mathit{mems}_2 k\ [\mapsto \sigma]) \rightsquigarrow \langle\ c_f', ?\mathit{mds}_k', \mathit{mem}_f'\ \rangle \wedge$
$(\mathit{cms}_1' ! k, \mathit{mem}_1'\ [\mapsto g1\ \sigma]) \approx \langle\ c_f', ?\mathit{mds}_k', \mathit{mem}_f'\ \rangle$
by (*metis mm-equiv-step*)

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

then obtain c_h mem_h **where** *h-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 *g2* **where** *e*:
change-respecting $(cms_2\ !\ k,\ ?mems_2k\ [\mapsto\ h])\ \langle\ c_h,\ ?mds_k{}',\ mem_h\ \rangle\ (?X\ k)\ g2$
using *a b noread-exists-change-respecting*
by (*metis differing-lists-finite surjective-pairing*)

— The following statements are universally quantified since they are reused later:
with *h-prop* **have**
$\forall\ \sigma.\ dom\ \sigma\ =\ ?X\ k\ \longrightarrow$
$(cms_2\ !\ k,\ ?mems_2k\ [\mapsto\ h]\ [\mapsto\ \sigma])\ \rightsquigarrow\ \langle\ c_h,\ ?mds_k{}',\ mem_h\ [\mapsto\ g2\ \sigma]\ \rangle$
unfolding *change-respecting.simps*
by *auto*

with *domh* **have** *f*:
$\forall\ \sigma.\ dom\ \sigma\ =\ ?X\ k\ \longrightarrow$
$(cms_2\ !\ k,\ ?mems_2k\ [\mapsto\ \sigma])\ \rightsquigarrow\ \langle\ c_h,\ ?mds_k{}',\ mem_h\ [\mapsto\ g2\ \sigma]\ \rangle$
by (*auto simp*: *subst-overrides*)

from *d* **and** *f* **have** *g*: $\bigwedge\ \sigma.\ dom\ \sigma\ =\ ?X\ k\ \Longrightarrow$
$(cms_2\ !\ k,\ ?mems_2k\ [\mapsto\ \sigma])\ \rightsquigarrow\ \langle\ c_h,\ ?mds_k{}',\ mem_h\ [\mapsto\ g2\ \sigma]\ \rangle\ \wedge$
$(cms_1{}'\ !\ k,\ mem_1{}'\ [\mapsto\ g1\ \sigma])\ \approx\ \langle\ c_h,\ ?mds_k{}',\ mem_h\ [\mapsto\ g2\ \sigma]\ \rangle$
using *h-prop*
by (*metis deterministic*)
let *?σ-mem*$_2$ = *to-partial mem*$_2$ |‘ $?X\ k$
define $mem_2{}'$ **where** $mem_2{}' = mem_h\ [\mapsto\ g2$ *?σ-mem*$_2]$
define $c_2{}'$ **where** $c_2{}' = c_h$

have *domσ-mem*$_2$: *dom ?σ-mem*$_2$ = $?X\ k$
by (*metis dom-restrict-total*)

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

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

define cms_2' **where** $cms_2' = cms_2\ [k := (c_2',\ ?mds_k')]$

with *i b equal-size* **have** $(cms_2,\ mem_2) \to (cms_2',\ mem_2')$
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: $\bigwedge \sigma.\ dom\ \sigma = ?X\ k \Longrightarrow dom\text{-}g1 = dom\ (g1\ \sigma)$
$\bigwedge \sigma.\ dom\ \sigma = ?X\ k \Longrightarrow dom\text{-}g2 = dom\ (g2\ \sigma)$
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 $(x \notin dom\text{-}g1) \vee (x \notin dom\text{-}g2)$
then $(mem_1'\ x,\ mem_2'\ x)$
else $(?mems_1 k\ x,\ ?mems_2 k\ x)$) **for** *x*
— This is used in two of the following cases, so we prove it beforehand:
have *x-unchanged*: $\bigwedge x.\ [\![\ x \in ?X\ k;\ x \in dom\text{-}g1;\ x \in dom\text{-}g2\ ]\!] \Longrightarrow$
$mem_1\ x = mem_1'\ x \wedge mem_2\ x = mem_2'\ x$
proof
fix *x*
assume $x \in ?X\ k$ **and** $x \in dom\text{-}g1$
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** $x \in dom\text{-}g2$

hence *eq-mem*$_2$: $?\sigma\text{-}mem_2\ x = Some\ (mem_2\ x)$
by (*metis restrict-in to-partial-def*)
hence $?mems_2 k\ [\mapsto h]\ [\mapsto ?\sigma\text{-}mem_2]\ x = mem_2\ x$
by (*auto simp*: *subst-def*)

moreover
from ‹$x \in$ *dom-g2*› *dom-uniq e* **have** *g-eq*: *g2 ?σ-mem*$_2$ x = *Some* (*mem*$_2$ x)
unfolding *change-respecting.simps func-le-def*
by (*metis dom-restrict-total eq-mem*$_2$)
hence *mem*$_h$ [↦ *g2 ?σ-mem*$_2$] x = *mem*$_2$ x
by (*auto simp*: *subst-def*)

ultimately have *?mems*$_2$*k* [↦ *h*] [↦ *?σ-mem*$_2$] x = *mem*$_h$ [↦ *g2 ?σ-mem*$_2$] x
by *auto*
thus *mem*$_2$ x = *mem*$_2$′ x
by (*metis* ‹*mem*$_2$ = *?mems*$_2$*k* [↦ *?σ-mem*$_2$]› *domσ-mem*$_2$ *domh mem*$_2$′*-def subst-overrides*)
qed

define *mems′-i*
where *mems′-i i x* =
(*if* ((*mem*$_1$ x ≠ *mem*$_1$′ x ∨ *mem*$_2$ x ≠ *mem*$_2$′ x) ∧
(*mem*$_1$′ x = *mem*$_2$′ x ∨ *dma* x = *High*))
then (*mem*$_1$′ x, *mem*$_2$′ x)
else if ((*mem*$_1$ x ≠ *mem*$_1$′ x ∨ *mem*$_2$ x ≠ *mem*$_2$′ x) ∧
(*mem*$_1$′ x ≠ *mem*$_2$′ 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 (λ *i*.
if i = *k*
then (*fst* ∘ *mems′-k*, *snd* ∘ *mems′-k*)
else (*fst* ∘ *mems′-i i*, *snd* ∘ *mems′-i i*))
[*0*..< *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*: ⟦ *i* ≠ *k*; *i* < *length cms*$_1$ ⟧ ⟹
mems′ ! *i* = (*fst* ∘ *mems′-i i*, *snd* ∘ *mems′-i i*)
unfolding *mems′-def*
by *auto*

have *mems′-k-1* [*simp*]: ⋀ x. ⟦ x ∉ *?X k* ⟧ ⟹
fst (*mems′* ! *k*) x = *mem*$_1$′ x ∧ *snd* (*mems′* ! *k*) x = *mem*$_2$′ x
unfolding *mems′-k-simp mems′-k-def*
by *auto*
have *mems′-k-2* [*simp*]: ⋀ x. ⟦ x ∈ *?X k*; x ∉ *dom-g1* ∨ x ∉ *dom-g2* ⟧ ⟹
fst (*mems′* ! *k*) x = *mem*$_1$′ x ∧ *snd* (*mems′* ! *k*) x = *mem*$_2$′ x
unfolding *mems′-k-simp mems′-k-def*
by *auto*

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

have *mems′-k-cases*:
$\bigwedge P\ x.$
$[\![$
$[\![\ x \notin ?X\ k \vee x \notin dom\text{-}g1 \vee x \notin dom\text{-}g2;$
fst (*mems′* ! *k*) *x* = *mem*$_1$′ *x*;
snd (*mems′* ! *k*) *x* = *mem*$_2$′ *x* $]\!] \Longrightarrow P\ x;$
$[\![\ x \in ?X\ k;\ x \in dom\text{-}g1;\ x \in dom\text{-}g2;$
fst (*mems′* ! *k*) *x* = *fst* (*mems* ! *k*) *x*;
snd (*mems′* ! *k*) *x* = *snd* (*mems* ! *k*) *x* $]\!] \Longrightarrow P\ x\ ]\!] \Longrightarrow P\ x$
using *mems′-k-1 mems′-k-2 mems′-k-3*
by *blast*

have *mems′-i-simp*:
$\bigwedge i.\ [\![\ i < length\ cms_1;\ i \neq k\ ]\!] \Longrightarrow mems'\ !\ i = (fst \circ mems'\text{-}i\ i,\ snd \circ mems'\text{-}i\ i)$
unfolding *mems′-def*
by *auto*

have *mems′-i-1* [*simp*]:
$\bigwedge i\ x.\ [\![\ i \neq k;\ i < length\ cms_1;$
mem$_1$ *x* ≠ *mem*$_1$′ *x* ∨ *mem*$_2$ *x* ≠ *mem*$_2$′ *x*;
mem$_1$′ *x* = *mem*$_2$′ *x* ∨ *dma x* = *High* $]\!] \Longrightarrow$
fst (*mems′* ! *i*) *x* = *mem*$_1$′ *x* ∧ *snd* (*mems′* ! *i*) *x* = *mem*$_2$′ *x*
unfolding *mems′-i-def mems′-i-simp*
by *auto*

have *mems′-i-2* [*simp*]:
$\bigwedge i\ x.\ [\![\ i \neq k;\ i < length\ cms_1;$
mem$_1$ *x* ≠ *mem*$_1$′ *x* ∨ *mem*$_2$ *x* ≠ *mem*$_2$′ *x*;
mem$_1$′ *x* ≠ *mem*$_2$′ *x*; *dma x* = *Low* $]\!] \Longrightarrow$
fst (*mems′* ! *i*) *x* = *some-val* ∧ *snd* (*mems′* ! *i*) *x* = *some-val*
unfolding *mems′-i-def mems′-i-simp*
by *auto*
have *mems′-i-3* [*simp*]:
$\bigwedge i\ x.\ [\![\ i \neq k;\ i < length\ cms_1;$
mem$_1$ *x* = *mem*$_1$′ *x*; *mem*$_2$ *x* = *mem*$_2$′ *x* $]\!] \Longrightarrow$
fst (*mems′* ! *i*) *x* = *fst* (*mems* ! *i*) *x* ∧ *snd* (*mems′* ! *i*) *x* = *snd* (*mems* ! *i*) *x*
unfolding *mems′-i-def mems′-i-simp*
by *auto*

have *mems′-i-cases*:
$\bigwedge P\ i\ x.$
$[\![\ i \neq k;\ i < length\ cms_1;$

```
        ⟦ 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*]: $\bigwedge x.\ x \in ?X'\ k \Longrightarrow x \in ?X\ k$
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 ‹$x \notin ?X\ k$› **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* ‹$i = k$› *differing-vars-elim*)
ultimately show *False*
by *auto*
qed

have *differing-in-dom*: $\bigwedge x.\ [\![\ x \in ?X\ k;\ x \in ?X'\ k\ ]\!] \Longrightarrow x \in dom\text{-}g1 \wedge x \in dom\text{-}g2$
proof (*rule ccontr*)
fix x
assume $x \in ?X\ k$
assume $\neg\ (x \in dom\text{-}g1 \wedge x \in dom\text{-}g2)$
hence *not-in-dom*: $x \notin dom\text{-}g1 \vee x \notin dom\text{-}g2$ **by** *auto*
hence $?mems_1'i\ x = mem_1'\ x\ ?mems_2'i\ x = mem_2'\ x$
using ‹$i = k$› ‹$x \in ?X\ k$› *mems'-k-2*
by *auto*

moreover assume $x \in ?X'\ k$
ultimately show *False*
by (*metis* ‹$i = k$› *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* ‹$i = k$›)
hence $?mems_1'i\ [\mapsto \sigma]\ x = v$
using ‹$x \in ?X'\ k$› *dom*σ
by (*auto simp*: *subst-def*)
moreover
from *c* **have** *le*: $g1\ \sigma' \preceq \sigma'$
using *dom*σ'

```
      by auto
    from domσ' and ‹x ∈ ?X' k› have x ∈ dom (g1 σ')
      by (metis diff-vars-impl differing-in-dom dom-uniq(1))

    hence mem₁' [↦ g1 σ'] x = v
      using domσ' c le
      unfolding func-le-def subst-def
    by (metis σ'-def ‹σ x = Some v› diff-vars-impl option.simps(5) x-in-X'k)

    ultimately show ?mems₁'i [↦ σ] x = mem₁' [↦ g1 σ'] x ..
  next
    assume x ∉ ?X' k

    hence ?mems₁'i [↦ σ] x = ?mems₁'i x
      using domσ
      by (metis ‹i = k› subst-not-in-dom)
    show ?thesis
    proof (cases x ∈ dom-g1)
      assume x ∈ dom-g1
      hence x ∈ dom (g1 σ')
        using domσ' dom-uniq
        by auto
      hence g1 σ' x = σ' x
        using c domσ'
        by (metis change-respecting.simps func-le-def)
      then have σ' x = Some (?mems₁'i x)
        unfolding σ'-def
        using domσ' domh
        by (metis ‹g1 σ' x = σ' x› ‹x ∈ dom (g1 σ')› ‹x ∉ ?X' k› domIff
dom-uniq(1))

      hence mem₁' [↦ g1 σ'] x = ?mems₁'i x
        unfolding subst-def
        by (metis ‹g1 σ' x = σ' x› option.simps(5))
      thus ?thesis
        by (metis ‹?mems₁'i [↦σ] x = ?mems₁'i x›)
    next
      assume x ∉ dom-g1
      then have mem₁' [↦ g1 σ'] x = mem₁' x
        by (metis domσ' dom-uniq(1) subst-not-in-dom)
      moreover
      have ?mems₁'i x = mem₁' x
        by (metis ‹i = k› ‹x ∉ ?X' k› differing-vars-neg)
      ultimately show ?thesis
        by (metis ‹?mems₁'i [↦σ] x = ?mems₁'i x›)
    qed
  qed
qed
```

```
      moreover have ?mems₂'i [↦ σ] = mem_h [↦ g2 σ']
      proof (rule ext)
        fix x

        show ?mems₂'i [↦ σ] x = mem_h [↦ g2 σ'] x
        proof (cases x ∈ ?X' k)
          assume x ∈ ?X' k

          then obtain v where σ x = Some v
            using domσ
            by (metis domD ‹i = k›)
          hence ?mems₂'i [↦ σ] x = v
            using ‹x ∈ ?X' k› domσ
            unfolding subst-def
            by (metis option.simps(5))
          moreover
          from e have le: g2 σ' ⪯ σ'
            using domσ'
            by auto
          from ‹x ∈ ?X' k› have x ∈ ?X k
            by auto
          hence x ∈ dom (g2 σ')
            by (metis differing-in-dom domσ' dom-uniq(2) ‹x ∈ ?X' k›)
          hence mem₂' [↦ g2 σ'] x = v
            using domσ' c le
            unfolding func-le-def subst-def
            by (metis σ'-def ‹σ x = Some v› diff-vars-impl option.simps(5) ‹x ∈
?X' k›)

          ultimately show ?thesis
        by (metis domσ' dom-restrict-total dom-uniq(2) mem₂'-def subst-overrides)
        next
          assume x ∉ ?X' k

          hence ?mems₂'i [↦ σ] x = ?mems₂'i x
            using domσ
            by (metis ‹i = k› subst-not-in-dom)
          show ?thesis
          proof (cases x ∈ dom-g2)
            assume x ∈ dom-g2
            hence x ∈ dom (g2 σ')
              using domσ'
              by (metis dom-uniq)
            hence g2 σ' x = σ' x
              using e domσ'
              by (metis change-respecting.simps func-le-def)
            then have σ' x = Some (?mems₂'i x)
            proof (cases x ∈ dom-g1)
              — This can't happen, so derive a contradiction.
```

assume $x \in$ *dom-g1*

have $x \notin$ *?X k*
proof (*rule ccontr*)
assume $\neg$ ($x \notin$ *?X k*)
hence $x \in$ *?X k* **by** *auto*
have $mem_1\ x = {mem_1}'\ x \wedge mem_2\ x = {mem_2}'\ x$
by (*metis* σ'*-def* ‹*g2* $\sigma'\ x = \sigma'\ x$› ‹$x \in$ *dom* (*g2* σ')›
‹$x \in$ *dom-g1*› ‹$x \in$ *dom-g2*› *domIff x-unchanged*)
moreover
from ‹$x \notin$ *?X′ k*› **have**
?mems$_1$′*i x* = *?mems*$_1$*k x* $\wedge$ *?mems*$_2$′*i x* = *?mems*$_2$*k x*
using ‹$x \in$ *?X k*› ‹$x \in$ *dom-g1*› ‹$x \in$ *dom-g2*›
by *auto*
ultimately show *False*
using ‹$x \in$ *?X k*› ‹$x \notin$ *?X′ k*›
by (*metis* ‹*i* = *k*› *differing-vars-elim differing-vars-neg*)
qed
hence *False*
by (*metis* σ'*-def* ‹*g2* $\sigma'\ x = \sigma'\ x$› ‹$x \in$ *dom* (*g2* σ')› *domIff*)
thus *?thesis*
by *blast*
next
assume $x \notin$ *dom-g1*
thus *?thesis*
unfolding σ'*-def*
by (*metis* ‹*g2* $\sigma'\ x = \sigma'\ x$› ‹$x \in$ *dom* (*g2* σ')› ‹$x \notin$ *?X′ k*›
domIff domσ′ dom-uniq domh)
qed
hence ${mem_2}'\ [\mapsto g2\ \sigma']\ x =$ *?mems*$_2$′*i x*
unfolding *subst-def*
by (*metis* ‹*g2* $\sigma'\ x = \sigma'\ x$› *option.simps*(*5*))
thus *?thesis*
using ‹$x \notin$ *?X′ k*› *domσ domσ′*
by (*metis* ‹*i* = *k*› *dom-restrict-total dom-uniq*(*2*)
${mem_2}'$*-def subst-not-in-dom subst-overrides*)
next
assume $x \notin$ *dom-g2*
then have $mem_h\ [\mapsto g2\ \sigma']\ x = mem_h\ x$
by (*metis domσ′ dom-uniq*(*2*) *subst-not-in-dom*)
moreover
have *?mems*$_2$′*i x* = ${mem_2}'\ x$
by (*metis* ‹*i* = *k*› ‹$x \notin$ *dom-g2*› *mems′-k-1 mems′-k-2*)

hence *?mems*$_2$′*i x* = $mem_h\ x$
unfolding ${mem_2}'$*-def*
by (*metis* ‹$x \notin$ *dom-g2*› *domσ-mem*$_2$ *dom-uniq*(*2*) *subst-not-in-dom*)
ultimately show *?thesis*
by (*metis* ‹*?mems*$_2$′*i* $[\mapsto\sigma]$ *x* = *?mems*$_2$′*i x*›)

```
      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 ∉ ?X' i
           hence ?mems₁'i x = mem₁' x ∧ ?mems₂'i x = mem₂' x
            by (metis differing-vars-neg)
           hence x ∉ ?X i
            using eq-mem mems'-simp
            by (auto simp: differing-vars-neg-intro)
           thus ?thesis
                  by (metis ‹dom σ' = ?X i› ‹x ∉ ?X' i› domσ mems'-simp
subst-not-in-dom)
          qed
         }
         thus ?thesis x by blast
        qed

        from o have
         p: ⋀ x. ⟦ ?mems₁'i [↦ σ] x ≠ ?mems₁i [↦ σ'] x ∨
                   ?mems₂'i [↦ σ] x ≠ ?mems₂i [↦ σ'] x ⟧ ⟹
         x ∉ snd (cms₁ ! i) AsmNoWrite
        proof
         fix x
         assume mems-neq:
          ?mems₁'i [↦ σ] x ≠ ?mems₁i [↦ σ'] x ∨ ?mems₂'i [↦ σ] x ≠ ?mems₂i
[↦ σ'] x
         hence modified:
          ¬ (doesnt-modify (fst (cms₁ ! k)) x) ∨ ¬ (doesnt-modify (fst (cms₂ ! 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 (cms₁ ! k, mem₁) ∧
           locally-sound-mode-use (cms₂ ! k, mem₂)
          unfolding sound-mode-use.simps
          by (metis b equal-size list-all-length)
         moreover
         have snd (cms₁ ! k) = snd (cms₂ ! k)
          by (metis b equal-size modes-eq nth-map)
         have (cms₁ ! k, mem₁) ∈ loc-reach (cms₁ ! k, mem₁)
          by (metis loc-reach.refl prod.collapse)
         hence guars:
              x ∈ snd (cms₁ ! k) GuarNoWrite ⟶ doesnt-modify (fst (cms₁ ! k))
x ∧
             x ∈ snd (cms₂ ! k) GuarNoWrite ⟶ doesnt-modify (fst (cms₁ ! k)) x
          using loc-modes
```

```
    unfolding locally-sound-mode-use-def ‹snd (cms₁ ! k) = snd (cms₂ ! k)›
    by (metis loc-reach.refl surjective-pairing)

  hence x ∉ snd (cms₁ ! k) GuarNoWrite
    using modified loc-modes locally-sound-mode-use-def
    by (metis ‹snd (cms₁ ! k) = snd (cms₂ ! k)› 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 ∉ snd (cms₁ ! i) AsmNoWrite
    unfolding compatible-modes-def
    using ‹i ≠ k› i-le
    by (metis (no-types) b length-list-update length-map nth-map)
qed

have q:
  ⋀ x. ⟦ dma x = Low;
          ?mems₁'i [↦ σ] x ≠ ?mems₁i [↦ σ'] x ∨
          ?mems₂'i [↦ σ] x ≠ ?mems₂i [↦ σ'] x;
          x ∉ ?X' i ⟧ ⟹
        mem₁' x = mem₂' x
  by (metis ‹i ≠ k› b compat-different-vars i-le length-list-update mems'-i-2
o)
have i < length cms₁
  by (metis cms₂'-def equal-size i-le length-list-update new-length)
with compat and ‹dom σ' = ?X i› have
  bisim: (cms₁ ! i, ?mems₁i [↦ σ']) ≈ (cms₂ ! i, ?mems₂i [↦ σ'])
  by auto

let ?Δ = differing-vars (?mems₁i [↦ σ']) (?mems₁'i [↦ σ]) ∪
         differing-vars (?mems₂i [↦ σ']) (?mems₂'i [↦ σ])

have Δ-finite: finite ?Δ
  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 =
           (if x ∈ ?Δ
            then if dma x = High
                 then Some (?mems₁'i [↦ σ] x, ?mems₂'i [↦ σ] x)
                 else if x ∈ ?X' i
                      then (case σ x of
                              Some v ⇒ Some (v, v)
                            | None ⇒ None)
                      else Some (mem₁' x, mem₁' x)
            else None) for x
have domA: dom A = ?Δ
proof
  show dom A ⊆ ?Δ
```

```
      using A-def
      apply (auto simp: domD)
      by (metis option.simps(2))
    next
     show ?Δ ⊆ dom A
      unfolding A-def
      apply auto
       apply (metis (no-types) domIff domσ option.exhaust option.simps(5))
      by (metis (no-types) domIff domσ option.exhaust option.simps(5))
    qed

    have A-correct:
        ⋀ x.
         globally-consistent-var A (snd (cms1 ! i)) x ∧
         ?mems1 i [↦ σ'] [||1 A] x = ?mems1' i [↦ σ] x ∧
         ?mems2 i [↦ σ'] [||2 A] x = ?mems2' i [↦ σ] x
    proof −
     fix x
     show ?thesis x
      (is ?A ∧ ?Eq1 ∧ ?Eq2)
     proof (cases x ∈ ?Δ)
      assume x ∈ ?Δ
      hence diff:
       ?mems1' i [↦ σ] x ≠ ?mems1 i [↦ σ'] x ∨ ?mems2' i [↦ σ] x ≠ ?mems2 i
[↦ σ'] x
        by (auto simp: differing-vars-def)
      from p and diff have writable: x ∉ snd (cms1 ! i) AsmNoWrite
       by auto
      show ?thesis
      proof (cases dma x)
       assume dma x = High
       from ‹dma x = High› have A-simp [simp]:
         A x = Some (?mems1' i [↦ σ] x, ?mems2' i [↦ σ] x)
         unfolding A-def
         by (metis ‹x ∈ ?Δ›)
       from writable have ?A
         unfolding globally-consistent-var-def A-simp
         using ‹dma x = High›
         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 ∈ ?X' i)
```

```
      assume x ∈ ?X' i
      then obtain v where σ x = Some v
        by (metis domD domσ)
      hence eq: ?mems₁'i [↦ σ] x = v ∧ ?mems₂'i [↦ σ] x = v
        unfolding subst-def
        by auto
      moreover
      from ‹x ∈ ?X' i› and ‹dma x = Low› have A-simp [simp]:
        A x = (case σ x of
                 Some v ⇒ Some (v, v)
               | None ⇒ None)
        unfolding A-def
        by (metis Sec.simps(1) ‹x ∈ ?Δ›)
      with writable eq ‹σ x = Some v› have ?A
        unfolding globally-consistent-var-def
        by auto
      ultimately show ?thesis
        using domA ‹x ∈ ?Δ› ‹σ x = Some v›
        by (auto simp: apply-adaptation-def)

    next
      assume x ∉ ?X' i
      hence A-simp [simp]: A x = Some (mem₁' x, mem₁' x)
        unfolding A-def
        using ‹x ∈ ?Δ› ‹dma x = Low›
        by auto
      from q have mem₁' x = mem₂' x
        by (metis ‹dma x = Low› diff ‹x ∉ ?X' i›)
      with writable have ?A
        unfolding globally-consistent-var-def
        by auto

      moreover
      from ‹x ∉ ?X' i› have
       ?mems₁'i [↦ σ] x = ?mems₁'i x ∧ ?mems₂'i [↦ σ] x = ?mems₂'i x
        by (metis domσ subst-not-in-dom)
      moreover
        from ‹x ∉ ?X' i› have ?mems₁'i x = mem₁' x ∧ ?mems₂'i x =
mem₂' x
        by (metis differing-vars-neg)
      ultimately show ?thesis
        using ‹mem₁' x = mem₂' x›
        by (auto simp: apply-adaptation-def)
    qed
  qed
next
  assume x ∉ ?Δ
  hence A x = None
    by (metis domA domIff)
```

```
          hence globally-consistent-var A (snd (cms₁ ! i)) x
            by (auto simp: globally-consistent-var-def)
          moreover
          from ‹A x = None› have x ∉ dom A
            by (metis domIff)
          from ‹x ∉ ?Δ› have ?mems₁ i [↦ σ'] [||₁ A] x = ?mems₁' i [↦ σ] x ∧
                             ?mems₂ i [↦ σ'] [||₂ A] x = ?mems₂' i [↦ σ] x
            using ‹A x = None›
            unfolding differing-vars-def apply-adaptation-def
            by auto

          ultimately show ?thesis
            by auto
        qed
      qed
      hence ?mems₁ i [↦ σ'] [||₁ A] = ?mems₁' i [↦ σ] ∧
            ?mems₂ i [↦ σ'] [||₂ A] = ?mems₂' i [↦ σ]
        by auto
      moreover
      from A-correct have globally-consistent A (snd (cms₁ ! i))
        by (metis Δ-finite globally-consistent-def domA)

      have snd (cms₁ ! i) = snd (cms₂ ! i)
        by (metis ‹i < length cms₁› equal-size modes-eq nth-map)

      with bisim have (cms₁ ! i, ?mems₁ i [↦ σ'] [||₁ A]) ≈ (cms₂ ! i, ?mems₂ i
[↦ σ'] [||₂ A])
        using ‹globally-consistent A (snd (cms₁ ! i))›
        apply (subst surjective-pairing[of cms₁ ! i])
        apply (subst surjective-pairing[of cms₂ ! i])
        by (metis surjective-pairing globally-consistent-adapt-bisim)

      ultimately show ?thesis
        by (metis ‹i ≠ k› b cms₂'-def nth-list-update-neq)
    qed
  next
    fix i x

    let ?mems₁'i = fst (mems' ! i)
    let ?mems₂'i = snd (mems' ! i)
    assume i-le: i < length cms₁'
    assume mem-eq: mem₁' x = mem₂' x ∨ dma x = High
    show x ∉ ?X' i
    proof (cases i = k)
      assume i = k
      thus x ∉ ?X' i
        apply (cases x ∉ ?X k ∨ x ∉ dom-g1 ∨ x ∉ 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 ≠ k
    thus x ∉ ?X' i
    proof (rule mems'-i-cases)
      from b i-le show i < length cms₁
        by (metis length-list-update)
    next
      assume fst (mems' ! i) x = mem₁' x
        snd (mems' ! i) x = mem₂' x
      thus x ∉ ?X' i
        by (metis differing-vars-neg-intro)
    next
      assume mem₁ x ≠ mem₁' x ∨ mem₂ x ≠ mem₂' x
        mem₁' x ≠ mem₂' x and dma x = Low
      — In this case, for example, the values of (mems' ! i) are not needed.
      thus x ∉ ?X' i
        by (metis Sec.simps(2) mem-eq)
    next
      assume case3: mem₁ x = mem₁' x mem₂ x = mem₂' x
        fst (mems' ! i) x = fst (mems ! i) x
        snd (mems' ! i) x = snd (mems ! i) x
      have x ∈ ?X' i ⟹ mem₁' x ≠ mem₂' x ∧ dma x = Low
      proof −
        assume x ∈ ?X' i
        from case3 and ‹x ∈ ?X' i› have x ∈ ?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 ‹mem₁' x = mem₂' x ∨ dma x = High› show x ∉ ?X' i
        by auto
    qed
  qed
next
  { fix x
    have ∃ i < length cms₁. x ∉ ?X' i
    proof (cases mem₁ x ≠ mem₁' x ∨ mem₂ x ≠ mem₂' x)
      assume var-changed: mem₁ x ≠ mem₁' x ∨ mem₂ x ≠ mem₂' x
      have x ∉ ?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 ¬ (mem₁ x ≠ mem₁' x ∨ mem₂ x ≠ mem₂' x)
      hence assms: mem₁ x = mem₁' x mem₂ x = mem₂' x by auto

      have length cms₁ ≠ 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
  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 length cms₁ = 0 ∧ mem₁ =ˡ mem₂
    with x-low show ?thesis
      by (simp add: low-eq-def)
  next
    assume ∀ x. ∃ j. j < length cms₁ ∧ x ∉ ?X j
    then obtain j where j-prop: j < length cms₁ ∧ x ∉ ?X j
      by auto
    let ?mems₁j = fst (mems ! j) and
        ?mems₂j = snd (mems ! j)

    obtain σ :: 'Var ⇀ 'Val where dom σ = ?X j
      by (metis dom-restrict-total)

   with compat and j-prop have (cms₁ ! j, ?mems₁j [↦ σ]) ≈ (cms₂ ! j, ?mems₂j
[↦ σ])
      by auto

    moreover
    have snd (cms₁ ! j) = snd (cms₂ ! j)
      using modes-eq
      by (metis j-prop length-map nth-map)

    ultimately have ?mems₁j [↦ σ] =snd (cms₁ ! j)ˡ ?mems₂j [↦ σ]
      using modes-eq j-prop
      by (metis prod.collapse mm-equiv-low-eq)
    hence ?mems₁j x = ?mems₂j x
      using x-low x-readable j-prop ‹dom σ = ?X j›
      unfolding low-mds-eq-def
      by (metis subst-not-in-dom)

    thus ?thesis
      using j-prop
      by (metis compat-different-vars)
  qed
qed

lemma loc-reach-subset:
  assumes eval: ⟨c, mds, mem⟩ ⇝ ⟨c', mds', mem'⟩
  shows loc-reach ⟨c', mds', mem'⟩ ⊆ loc-reach ⟨c, mds, mem⟩
proof (clarify)
  fix c'' mds'' mem''
  from eval have ⟨c', mds', mem'⟩ ∈ loc-reach ⟨c, mds, mem⟩
    by (metis loc-reach.refl loc-reach.step surjective-pairing)
  assume ⟨c'', mds'', mem''⟩ ∈ loc-reach ⟨c', mds', mem'⟩
  thus ⟨c'', mds'', mem''⟩ ∈ loc-reach ⟨c, mds, mem⟩
    apply induct
     apply (metis ‹⟨c', mds', mem'⟩ ∈ loc-reach ⟨c, mds, mem⟩› 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** $\langle c', mds', mem'\rangle \in$ *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* $(cms', mem') \subseteq$ *reachable-mode-states* (cms, mem)
proof
fix *mdss*
assume *mdss* $\in$ *reachable-mode-states* (cms', mem')
thus *mdss* $\in$ *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*: $\forall\ x.\ x \in mds\ AsmNoWrite \longrightarrow mem_1\ x = mem_2\ x$
shows $\langle c', mds', mem'\rangle \in$ *loc-reach* $\langle c, mds, mem_1\rangle \Longrightarrow \langle c', mds', mem'\rangle \in$ *loc-reach* $\langle c, mds, mem_2\rangle$
proof −
let *?lc* $= \langle c', mds', mem'\rangle$
assume *?lc* $\in$ *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: ∀ x. x ∈ mds AsmNoWrite ⟶ mem' x = mem x
  shows loc-reach ⟨c, mds, mem⟩ = loc-reach ⟨c, mds, mem'⟩
  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) → (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: ∀ 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) ⇝ (cms_k', mem') ∧ k < length cms ∧ cms' = cms [k :=
cms_k']
    by (metis meval-elim)
  hence length cms = length cms'
    by auto
  have ⋀ i. i < length cms' ⟹ 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 ≠ 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 $\bigwedge x.\ x \in snd\ (cms\ !\ i)\ AsmNoWrite \Longrightarrow x \in snd\ (cms\ !\ k)\ GuarNoWrite$
unfolding *compatible-modes-def*
by (*metis* (*no-types*) ‹$i \neq k$› ‹*length cms = length cms'*› *ev i-le length-map nth-map*)
hence $\bigwedge x.\ x \in snd\ (cms\ !\ i)\ AsmNoWrite \longrightarrow doesnt\text{-}modify\ (fst\ (cms\ !\ k))\ x$
using *ev loc-sound*
unfolding *locally-sound-mode-use-def*
by (*metis loc-reach.refl surjective-pairing*)
with *eval* **have** $\bigwedge x.\ x \in snd\ (cms\ !\ i)\ AsmNoWrite \longrightarrow mem\ x = mem'\ x$
by (*metis* (*no-types*) *doesnt-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* ‹*length cms = length cms'*›
unfolding *locally-sound-mode-use-def*
by (*metis* ‹*cms'* ! *i* = *cms* ! *i*›)
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* (*cms*$_1$, *mem*$_1$) (*cms*$_2$, *mem*$_2$) *mems*
assumes *modes-eq*: *map snd cms*$_1$ = *map snd cms*$_2$
assumes *sound-modes*: *sound-mode-use* (*cms*$_1$, *mem*$_1$) *sound-mode-use* (*cms*$_2$, *mem*$_2$)
assumes *eval*: (*cms*$_1$, *mem*$_1$) $\rightarrow_k$ (*cms*$_1$', *mem*$_1$')
shows $\exists$ *cms*$_2$' *mem*$_2$' *mems'*. *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'*
proof −

from *eval* **show** *?thesis*
proof (*induct k arbitrary*: *cms*$_1$' *mem*$_1$')
case *0*
hence *cms*$_1$' = *cms*$_1$ $\wedge$ *mem*$_1$' = *mem*$_1$
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*:
$i < n \Longrightarrow x \notin$ *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*) ∧ *length* (*zip cmds ?mdss*) = *length ?mems*

by *auto*
next
fix i σ
let $?mems_1 i = fst\ (?mems\ !\ i)$ **and** $?mems_2 i = snd\ (?mems\ !\ i)$
assume i: $i < length\ (zip\ cmds\ ?mdss)$
with *coms-secure* **have** $com\text{-}sifum\text{-}secure\ (cmds\ !\ i)$
using *coms-secure*
by (*metis length-map length-replicate list-all-length map-snd-zip*)
with i **have** $\bigwedge mem_1\ mem_2.\ mem_1 =_{mds_s}{}^{l} mem_2 \Longrightarrow$
$(zip\ cmds\ (replicate\ ?n\ mds_s)\ !\ i, mem_1) \approx (zip\ cmds\ (replicate\ ?n\ mds_s)\ !\ i, mem_2)$
using *com-sifum-secure-def low-indistinguishable-def*
by *auto*

moreover
from i **have** $?mems_1 i = mem_1\ ?mems_2 i = mem_2$
by *auto*
with *low-eq* **have** $?mems_1 i\ [\mapsto \sigma] =_{mds_s}{}^{l} ?mems_2 i\ [\mapsto \sigma]$
by (*auto simp*: *subst-def* mds_s*-def low-mds-eq-def low-eq-def*, *case-tac* $\sigma\ x$, *auto*)

ultimately show $(zip\ cmds\ ?mdss\ !\ i, ?mems_1 i\ [\mapsto \sigma]) \approx (zip\ cmds\ ?mdss\ !\ i, ?mems_2 i\ [\mapsto \sigma])$
by *simp*
next
fix $i\ x$
assume $i < length\ (zip\ cmds\ ?mdss)$
with *diff-empty* **show** $x \notin ?X\ i$ **by** *auto*
next
show $(length\ (zip\ cmds\ ?mdss) = 0 \wedge mem_1 =^{l} mem_2) \vee (\forall\ x.\ \exists\ i < length\ (zip\ cmds\ ?mdss).\ x \notin ?X\ i)$
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 $mem_1\ mem_2 :: 'Var \Rightarrow 'Val$
fix $k\ cms_1'\ mem_1'$
let $?n = length\ cmds$
let $?mems = zip\ (replicate\ ?n\ mem_1)\ (replicate\ ?n\ mem_2)$
assume *low-eq*: $mem_1 =^{l} mem_2$

with *com-secure* **have** *compat*:
makes-compatible (*add-initial-modes cmds*, mem_1) (*add-initial-modes cmds*, mem_2) *?mems*
by (*metis compatible-refl fst-conv length-replicate map-replicate snd-conv zip-eq-conv*)

also assume *eval*: (*add-initial-modes cmds*, mem_1) $\rightarrow_k$ (cms_1', mem_1')

ultimately obtain cms_2' mem_2' *mems'*
where *p*: *map snd* $cms_1' = map\ snd\ cms_2' \wedge$
(*add-initial-modes cmds*, mem_2) $\rightarrow_k$ (cms_2', mem_2') $\wedge$
makes-compatible (cms_1', mem_1') (cms_2', mem_2') *mems'*
using *sound-modes makes-compatible-eval-k*
by *blast*

thus $\exists\ cms_2'\ mem_2'.$ (*add-initial-modes cmds*, mem_2) $\rightarrow_k$ (cms_2', mem_2') $\wedge$
map snd $cms_1' = map\ snd\ cms_2' \wedge$
length $cms_2' = length\ cms_1' \wedge$
$(\forall\ x.\ dma\ x = Low \wedge (\forall\ i < length\ cms_1'.\ x \notin snd\ (cms_1'\ !\ i)$ *AsmNoRead*)
$\longrightarrow mem_1'\ x = mem_2'\ x$)
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** ‹←› *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 $eval\text{-}vars\text{-}det_A$: ⟦ ∀ *x* ∈ *aexp-vars e*. mem_1 *x* = mem_2 *x* ⟧ ⟹ $eval_A$ mem_1 *e* = $eval_A$ mem_2 *e*
assumes $eval\text{-}vars\text{-}det_B$: ⟦ ∀ *x* ∈ *bexp-vars b*. mem_1 *x* = mem_2 *x* ⟧ ⟹ $eval_B$ mem_1 *b* = $eval_B$ mem_2 *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 $conf_w\text{-}abv$ :: (*'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* ⇒ *Mode*
where
updated-mode (*Acq* - *m*) = *m* |
updated-mode (*Rel* - *m*) = *m*

inductive-set $eval_w$-*simple* :: ((*'Var*, *'AExp*, *'BExp*) *Stmt* × (*'Var*, *'Val*) *Mem*) *rel*
and $eval_w$-*simple-abv* :: ((*'Var*, *'AExp*, *'BExp*) *Stmt* × (*'Var*, *'Val*) *Mem*) ⇒
(*'Var*, *'AExp*, *'BExp*) *Stmt* × (*'Var*, *'Val*) *Mem* ⇒ *bool*
(**infixr** ‹$\rightsquigarrow_s$› *60*)
where
c $\rightsquigarrow_s$ *c'* ≡ (*c*, *c'*) ∈ $eval_w$-*simple* |
assign: ((*x* ← *e*, *mem*), (*Stop*, *mem* (*x* := $eval_A$ *mem e*))) ∈ $eval_w$-*simple* |
skip: ((*Skip*, *mem*), (*Stop*, *mem*)) ∈ $eval_w$-*simple* |
seq-stop: ((*Seq Stop c*, *mem*), (*c*, *mem*)) ∈ $eval_w$-*simple* |
if-true: ⟦ $eval_B$ *mem b* ⟧ ⟹ ((*If b t e*, *mem*), (*t*, *mem*)) ∈ $eval_w$-*simple* |
if-false: ⟦ ¬ $eval_B$ *mem b* ⟧ ⟹ ((*If b t e*, *mem*), (*e*, *mem*)) ∈ $eval_w$-*simple* |
while: ((*While b c*, *mem*), (*If b* (*c* ;; *While b c*) *Stop*, *mem*)) ∈ $eval_w$-*simple*

primrec *cxt-to-stmt* :: (*'Var*, *'AExp*, *'BExp*) *EvalCxt* ⇒ (*'Var*, *'AExp*, *'BExp*) *Stmt*
⇒ (*'Var*, *'AExp*, *'BExp*) *Stmt*
where
cxt-to-stmt [] *c* = *c* |
cxt-to-stmt (*c* # *cs*) *c'* = *Seq c'* (*cxt-to-stmt cs c*)

inductive-set $eval_w$ :: ((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf rel*
and $eval_w$-*abv* :: ((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf* ⇒
((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf* ⇒ *bool*
(**infixr** ‹$\rightsquigarrow_w$› *60*)
where
c $\rightsquigarrow_w$ *c'* ≡ (*c*, *c'*) ∈ $eval_w$ |
unannotated: ⟦ (*c*, *mem*) $\rightsquigarrow_s$ (*c'*, *mem'*) ⟧
⟹ (⟨*cxt-to-stmt E c*, *mds*, *mem*⟩$_w$, ⟨*cxt-to-stmt E c'*, *mds*, *mem'*⟩$_w$) ∈ $eval_w$ |
seq: ⟦ ⟨c_1, *mds*, *mem*⟩$_w$ $\rightsquigarrow_w$ ⟨c_1', *mds'*, *mem'*⟩$_w$ ⟧ ⟹ (⟨(c_1 ;; c_2), *mds*, *mem*⟩$_w$, ⟨(c_1' ;; c_2), *mds'*, *mem'*⟩$_w$) ∈ $eval_w$ |
decl: ⟦ ⟨*c*, *update-modes mu mds*, *mem*⟩$_w$ $\rightsquigarrow_w$ ⟨*c'*, *mds'*, *mem'*⟩$_w$ ⟧ ⟹
(⟨*cxt-to-stmt E* (*ModeDecl c mu*), *mds*, *mem*⟩$_w$, ⟨*cxt-to-stmt E c'*, *mds'*, *mem'*⟩$_w$) ∈ $eval_w$

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'*)) ∈ $eval_w$
inductive-cases *stop-no-eval'* [*elim*]: ((*Stop*, *mem*), (*c'*, *mem'*)) ∈ $eval_w$-*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*:
$[\![\ cxt\text{-}to\text{-}stmt\ E\ c = c'\ ;\ \bigwedge p\ q.\ c' \neq Seq\ p\ q\ ]\!] \Longrightarrow E = [] \wedge c' = c$
by (*metis cxt-to-stmt.simps*(*1*) *cxt-to-stmt.simps*(*2*) *neq-Nil-conv*)

lemma *cxt-inv-assign*:
$[\![\ cxt\text{-}to\text{-}stmt\ E\ c = x \leftarrow e\ ]\!] \Longrightarrow c = x \leftarrow e \wedge E = []$
by (*metis Stmt.simps*(*11*) *cxt-inv*)

lemma *cxt-inv-skip*:
$[\![\ cxt\text{-}to\text{-}stmt\ E\ c = Skip\ ]\!] \Longrightarrow c = Skip \wedge E = []$
by (*metis Stmt.simps*(*21*) *cxt-inv*)

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

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

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

lemma *skip-elim* [*elim*]:
$\langle Skip,\ mds,\ mem\rangle_w \rightsquigarrow_w \langle c',\ mds',\ mem'\rangle_w \Longrightarrow c' = Stop \wedge mds = mds' \wedge mem = mem'$
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*]:
$\langle x \leftarrow e,\ mds,\ mem\rangle_w \rightsquigarrow_w \langle c',\ mds',\ mem'\rangle_w \Longrightarrow c' = Stop \wedge mds = mds' \wedge mem' = mem\ (x := eval_A\ mem\ e)$
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*]:
⋀ *P*.
⟦ ⟨*If b p q*, *mds*, *mem*⟩$_w$ ⇝$_w$ ⟨*c′*, *mds′*, *mem′*⟩$_w$;
⟦ *c′* = *p*; *mem′* = *mem* ; *mds′* = *mds* ; *eval*$_B$ *mem b* ⟧ ⟹ *P* ;
⟦ *c′* = *q*; *mem′* = *mem* ; *mds′* = *mds* ; ¬ *eval*$_B$ *mem b* ⟧ ⟹ *P* ⟧ ⟹ *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*!]: (*While e c*, *mem*) ⇝$_s$ (*c′*, *mem′*)

lemma *while-elim* [*elim*]:
⟦ ⟨*While e c*, *mds*, *mem*⟩$_w$ ⇝$_w$ ⟨*c′*, *mds′*, *mem′*⟩$_w$ ⟧ ⟹ *c′* = *If e* (*c* ;; *While e c*) *Stop* ∧ *mds′* = *mds* ∧ *mem′* = *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*], *mem*) ⇝$_s$ (*c′*, *mem′*)

lemma *upd-elim* [*elim*]:
⟨*c*@[*upd*], *mds*, *mem*⟩$_w$ ⇝$_w$ ⟨*c′*, *mds′*, *mem′*⟩$_w$ ⟹ ⟨*c*, *update-modes upd mds*, *mem*⟩$_w$ ⇝$_w$ ⟨*c′*, *mds′*, *mem′*⟩$_w$
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*$_2$ = *cxt-to-stmt E c* ⟹ (*E* = [] ∧ *c* = *c*$_1$;; *c*$_2$) ∨ (∃ *c′ cs*. *E* = *c′* # *cs* ∧ *c* = *c*$_1$ ∧ *c*$_2$ = *cxt-to-stmt 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*$_2$, *mem*) ⇝$_s$ (*c′*, *mem′*)

lemma *stop-no-eval*: ¬ (⟨*Stop*, *mds*, *mem*⟩$_w$ ⇝$_w$ ⟨*c′*, *mds′*, *mem′*⟩$_w$)
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*]:
⟨*Stop* ;; *c*, *mds*, *mem*⟩$_w$ ⇝$_w$ ⟨*c′*, *mds′*, *mem′*⟩$_w$ ⟹ *c′* = *c* ∧ *mds′* = *mds* ∧ *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*]:
⟦ ⟨c_1 ;; c_2, *mds*, *mem*⟩$_w$ ⇝$_w$ ⟨c', *mds'*, *mem'*⟩$_w$; $c_1 \neq$ *Stop* ⟧ ⟹
(∃ c_1'. ⟨c_1, *mds*, *mem*⟩$_w$ ⇝$_w$ ⟨c_1', *mds'*, *mem'*⟩$_w$ ∧ $c' = c_1'$;; 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 ⟹ c = *Stop*
by (*metis Stmt.simps*(*41*) *cxt-to-stmt.simps*(*1*) *cxt-to-stmt.simps*(*2*) *neq-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* ⇀ *Type*

locale *sifum-types* =
  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-types*
begin

abbreviation *mm-equiv-abv2* :: (-, -, -) *LocalConf* ⇒ (-, -, -) *LocalConf* ⇒ *bool* (**infix** ‹≈› *60*)
  where *mm-equiv-abv2 c c'* ≡ *mm-equiv-abv c c'*

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

abbreviation *low-indistinguishable-abv* :: *'Var Mds* ⇒ (*'Var*, *'AExp*, *'BExp*) *Stmt* ⇒ (-, -, -) *Stmt* ⇒ *bool*
  (‹- ∼1 -› [*100*, *100*] *80*)
  where
  $c \sim_{mds} c' \equiv$ *low-indistinguishable mds c c'*

definition *to-total* :: *'Var TyEnv* ⇒ *'Var* ⇒ *Sec*
where *to-total* Γ *v* ≡ *if v* ∈ *dom* Γ *then the* (Γ *v*) *else dma v*

definition *max-dom* :: *Sec set* ⇒ *Sec*
  where *max-dom xs* ≡ *if High* ∈ *xs then High else Low*

inductive *type-aexpr* :: *'Var TyEnv* ⇒ *'AExp* ⇒ *Type* ⇒ *bool* (‹- ⊢$_a$ - ∈ -› [*120*, *120*, *120*] *1000*)
  where
  type-aexpr [*intro*!]: Γ ⊢$_a$ *e* ∈ *max-dom* (*image* (λ *x*. *to-total* Γ *x*) (*aexp-vars e*))

inductive-cases *type-aexpr-elim* [*elim*]: Γ ⊢$_a$ *e* ∈ *t*

inductive *type-bexpr* :: *'Var TyEnv* ⇒ *'BExp* ⇒ *Type* ⇒ *bool* (‹- ⊢$_b$ - ∈ - › [*120*, *120*, *120*] *1000*)
  where
  type-bexpr [*intro*!]: Γ ⊢$_b$ *e* ∈ *max-dom* (*image* (λ *x*. *to-total* Γ *x*) (*bexp-vars e*))

inductive-cases *type-bexpr-elim* [*elim*]: Γ ⊢$_b$ *e* ∈ *t*

definition *mds-consistent* :: *'Var Mds* ⇒ *'Var TyEnv* ⇒ *bool*
  where *mds-consistent mds* Γ ≡
    dom Γ = {(*x* :: *'Var*). (*dma x* = *Low* ∧ *x* ∈ *mds AsmNoRead*) ∨

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

fun *add-anno-dom* :: *'Var TyEnv* ⇒ *'Var ModeUpd* ⇒ *'Var set*
where
add-anno-dom Γ (*Acq v AsmNoRead*) = (*if dma v = Low then dom* Γ ∪ {*v*} *else dom* Γ) |
add-anno-dom Γ (*Acq v AsmNoWrite*) = (*if dma v = High then dom* Γ ∪ {*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* ⇒ *'Var ModeUpd* ⇒ *'Var TyEnv* (**infix** ‹⊕› *60*)
where
Γ ⊕ *upd* = ((λ*x*. *Some* (*to-total* Γ *x*)) |‘ *add-anno-dom* Γ *upd*)

definition *context-le* :: *'Var TyEnv* ⇒ *'Var TyEnv* ⇒ *bool* (**infixr** ‹⊑$_c$› *100*)
where
Γ ⊑$_c$ Γ′ ≡ (*dom* Γ = *dom* Γ′) ∧ (∀ *x* ∈ *dom* Γ. *the* (Γ *x*) ⊑ *the* (Γ′ *x*))

inductive *has-type* :: *'Var TyEnv* ⇒ (*'Var*, *'AExp*, *'BExp*) *Stmt* ⇒ *'Var TyEnv* ⇒ *bool*
(‹⊢ - {-} ->› [*120*, *120*, *120*] *1000*)
where
stop-type [*intro*]: ⊢ Γ {*Stop*} Γ |
skip-type [*intro*] : ⊢ Γ {*Skip*} Γ |
assign$_1$: ⟦ *x* ∉ *dom* Γ ; Γ ⊢$_a$ *e* ∈ *t*; *t* ⊑ *dma x* ⟧ ⟹ ⊢ Γ {*x* ← *e*} Γ |
assign$_2$: ⟦ *x* ∈ *dom* Γ ; Γ ⊢$_a$ *e* ∈ *t* ⟧ ⟹ *has-type* Γ (*x* ← *e*) (Γ (*x* := *Some t*)) |
if-type [*intro*]: ⟦ Γ ⊢$_b$ *e* ∈ *High* ⟶
((∀ *mds*. *mds-consistent mds* Γ ⟶ (*low-indistinguishable mds c*$_1$ *c*$_2$)) ∧
(∀ *x* ∈ *dom* Γ′. Γ′ *x* = *Some High*))
; ⊢ Γ { *c*$_1$ } Γ′
; ⊢ Γ { *c*$_2$ } Γ′ ⟧ ⟹
⊢ Γ { *If e c*$_1$ *c*$_2$ } Γ′ |
while-type [*intro*]: ⟦ Γ ⊢$_b$ *e* ∈ *Low* ; ⊢ Γ { *c* } Γ ⟧ ⟹ ⊢ Γ { *While e c* } Γ |
anno-type [*intro*]: ⟦ Γ′ = Γ ⊕ *upd* ; ⊢ Γ′ { *c* } Γ″ ; *c* ≠ *Stop* ;
∀ *x*. *to-total* Γ *x* ⊑ *to-total* Γ′ *x* ⟧ ⟹ ⊢ Γ { *c*@[*upd*] } Γ″ |
seq-type [*intro*]: ⟦ ⊢ Γ { *c*$_1$ } Γ′ ; ⊢ Γ′ { *c*$_2$ } Γ″ ⟧ ⟹ ⊢ Γ { *c*$_1$;; *c*$_2$ } Γ″ |
sub : ⟦ ⊢ Γ$_1$ { *c* } Γ$_1$′ ; Γ$_2$ ⊑$_c$ Γ$_1$; Γ$_1$′ ⊑$_c$ Γ$_2$′ ⟧ ⟹ ⊢ Γ$_2$ { *c* } Γ$_2$′

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* ⇒ *bool*

where
has-annotated-stop (*c*@[-]) = (*if c = Stop then True else has-annotated-stop c*) |
has-annotated-stop (*Seq p q*) = (*has-annotated-stop p* ∨ *has-annotated-stop q*) |
has-annotated-stop (*If - p q*) = (*has-annotated-stop p* ∨ *has-annotated-stop q*) |
has-annotated-stop (*While - p*) = *has-annotated-stop p* |
has-annotated-stop - = *False*

inductive-cases *has-type-elim*: ⊢ Γ { *c* } Γ′
inductive-cases *has-type-stop-elim*: ⊢ Γ { *Stop* } Γ′

definition *tyenv-eq* :: *'Var TyEnv* ⇒ (*'Var*, *'Val*) *Mem* ⇒ (*'Var*, *'Val*) *Mem* ⇒ *bool*
(**infix** ‹=₁› *60*)
where $mem_1 =_\Gamma mem_2 \equiv \forall\ x.$ (*to-total* Γ *x* = *Low* ⟶ $mem_1\ x = mem_2\ x$)

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* ⇒ ((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf rel*
and $\mathcal{R}_1$-*abv* :: *'Var TyEnv* ⇒
((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf* ⇒
((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf* ⇒
bool (‹- $\mathcal{R}^1$₁ -› [*120*, *120*] *1000*)
for Γ′ :: *'Var TyEnv*
where
x $\mathcal{R}^1_\Gamma$ *y* ≡ (*x*, *y*) ∈ $\mathcal{R}_1$ Γ |
intro [*intro*!] : ⟦ ⊢ Γ { *c* } Γ′ ; *mds-consistent mds* Γ ; $mem_1 =_\Gamma mem_2$ ⟧ ⟹ ⟨*c*, *mds*, mem_1⟩ $\mathcal{R}^1{}_{\Gamma'}$ ⟨*c*, *mds*, mem_2⟩

inductive-set $\mathcal{R}_2$ :: *'Var TyEnv* ⇒ ((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf rel*
and $\mathcal{R}_2$-*abv* :: *'Var TyEnv* ⇒
((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf* ⇒
((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf* ⇒
bool (‹- $\mathcal{R}^2$₁ -› [*120*, *120*] *1000*)
for Γ′ :: *'Var TyEnv*
where
x $\mathcal{R}^2_\Gamma$ *y* ≡ (*x*, *y*) ∈ $\mathcal{R}_2$ Γ |
intro [*intro*!] : ⟦ ⟨c_1, *mds*, mem_1⟩ ≈ ⟨c_2, *mds*, mem_2⟩ ;
∀ *x* ∈ *dom* Γ′. Γ′ *x* = *Some High* ;
⊢ Γ_1 { c_1 } Γ′ ; ⊢ Γ_2 { c_2 } Γ′ ;
mds-consistent mds Γ_1 ; *mds-consistent mds* Γ_2 ⟧ ⟹
⟨c_1, *mds*, mem_1⟩ $\mathcal{R}^2{}_{\Gamma'}$ ⟨c_2, *mds*, mem_2⟩

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

$(('Var, 'AExp, 'BExp)\ Stmt, 'Var, 'Val)\ LocalConf \Rightarrow$
$bool\ (\langle\text{-}\ \mathcal{R}^3{}_1\ \text{-}\rangle\ [120, 120]\ 1000)$
and $\mathcal{R}_3 :: 'Var\ TyEnv \Rightarrow (('Var, 'AExp, 'BExp)\ Stmt, 'Var, 'Val)\ LocalConf\ rel$
where
$\mathcal{R}_3\ \Gamma' \equiv \{(lc_1, lc_2).\ \mathcal{R}_3\text{-}aux\ \Gamma'\ lc_1\ lc_2\}$ |
$intro_1\ [intro] : [\![\langle c_1, mds, mem_1\rangle\ \mathcal{R}^1{}_\Gamma\ \langle c_2, mds, mem_2\rangle; \vdash \Gamma\ \{\ c\ \}\ \Gamma']\!] \Longrightarrow$
$\langle Seq\ c_1\ c, mds, mem_1\rangle\ \mathcal{R}^3{}_{\Gamma'}\ \langle Seq\ c_2\ c, mds, mem_2\rangle$ |
$intro_2\ [intro] : [\![\langle c_1, mds, mem_1\rangle\ \mathcal{R}^2{}_\Gamma\ \langle c_2, mds, mem_2\rangle; \vdash \Gamma\ \{\ c\ \}\ \Gamma']\!] \Longrightarrow$
$\langle Seq\ c_1\ c, mds, mem_1\rangle\ \mathcal{R}^3{}_{\Gamma'}\ \langle Seq\ c_2\ c, mds, mem_2\rangle$ |
$intro_3\ [intro] : [\![\langle c_1, mds, mem_1\rangle\ \mathcal{R}^3{}_\Gamma\ \langle c_2, mds, mem_2\rangle; \vdash \Gamma\ \{\ c\ \}\ \Gamma']\!] \Longrightarrow$
$\langle Seq\ c_1\ c, mds, mem_1\rangle\ \mathcal{R}^3{}_{\Gamma'}\ \langle Seq\ c_2\ c, mds, mem_2\rangle$

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

inductive-set $\mathcal{R} :: 'Var\ TyEnv \Rightarrow$
$(('Var, 'AExp, 'BExp)\ Stmt, 'Var, 'Val)\ LocalConf\ rel$
and $\mathcal{R}\text{-}abv :: 'Var\ TyEnv \Rightarrow$
$(('Var, 'AExp, 'BExp)\ Stmt, 'Var, 'Val)\ LocalConf \Rightarrow$
$(('Var, 'AExp, 'BExp)\ Stmt, 'Var, 'Val)\ LocalConf \Rightarrow$
$bool\ (\langle\text{-}\ \mathcal{R}^u{}_1\ \text{-}\rangle\ [120, 120]\ 1000)$
for $\Gamma :: 'Var\ TyEnv$
where
$x\ \mathcal{R}^u{}_\Gamma\ y \equiv (x, y) \in \mathcal{R}\ \Gamma$ |
$intro_1$: $lc\ \mathcal{R}^1{}_\Gamma\ lc' \Longrightarrow (lc, lc') \in \mathcal{R}\ \Gamma$ |
$intro_2$: $lc\ \mathcal{R}^2{}_\Gamma\ lc' \Longrightarrow (lc, lc') \in \mathcal{R}\ \Gamma$ |
$intro_3$: $lc\ \mathcal{R}^3{}_\Gamma\ lc' \Longrightarrow (lc, lc') \in \mathcal{R}\ \Gamma$

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

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

lemma $\mathcal{R}_2$-*sym*: *sym* ($\mathcal{R}_2$ Γ)
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* ($\mathcal{R}_3$ Γ)
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*]: $\langle c_1, mds, mem_1\rangle$ ${\mathcal{R}^u}_\Gamma$ $\langle c_2, mds', mem_2\rangle \Longrightarrow mds = mds'$
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* ($\mathcal{R}$ Γ)
unfolding *sym-def*
proof (*clarify*)
fix c_1 *mds* mem_1 c_2 mds_2 mem_2
assume *asm*: ($\langle c_1, mds, mem_1\rangle$, $\langle c_2, mds_2, mem_2\rangle$) $\in$ $\mathcal{R}$ Γ
with $\mathcal{R}$-*mds* **have** [*simp*]: mds_2 = *mds*
by *blast*
from *asm* **show** ($\langle c_2, mds_2, mem_2\rangle$, $\langle c_1, mds, mem_1\rangle$) $\in$ $\mathcal{R}$ Γ
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* ($\mathcal{R}_1$ Γ')

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** $\bigwedge v_1\ v_2.\ [\![\ dma\ x = High\ ;\ x \notin mds\ AsmNoWrite\ ]\!] \Longrightarrow$
$\langle c_1, mds, mem_1(x := v_1)\rangle\ \mathcal{R}^1{}_{\Gamma'}\ \langle c_2, mds, mem_2(x := v_2)\rangle$
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 $intro_1$
by *auto*
qed

lemma $\mathcal{R}_2$-*closed-glob-consistent*: *closed-glob-consistent* ($\mathcal{R}_2\ \Gamma'$)
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 $\Gamma_1\ \Gamma_2$ **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 $\bigwedge mem_1\ mem_2.\ (\langle c_1, mds, mem_1\rangle, \langle c_2, mds, mem_2\rangle) \in \mathcal{R}' \Longrightarrow \langle c_1, mds, mem_1\rangle\ \mathcal{R}^2{}_{\Gamma'}\ \langle c_2, mds, mem_2\rangle$
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 $\mathcal{R}'$*-prop*
unfolding *closed-glob-consistent-def*
by *simp*
qed

fun *closed-glob-helper* :: *'Var TyEnv* ⇒ ((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf* ⇒ ((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf* ⇒ *bool*
where
closed-glob-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 $\mathcal{R}_3$*-closed-glob-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-glob-helper* $\Gamma'\ \langle c_1,\ mds,\ mem_1\rangle\ \langle c_2,\ mds,\ mem_2\rangle$
proof (*induct rule*: $\mathcal{R}_3$*-aux.induct*)
case ($intro_1\ \Gamma\ c_1\ mds\ mem_1\ c_2\ mem_2\ c\ \Gamma'$)
thus *?case*
using $\mathcal{R}_1$*-closed-glob-consistent* [*of* Γ] **and** $\mathcal{R}_3$*-aux.intro*$_1$
unfolding *closed-glob-consistent-def*
by (*simp*, *blast*)
next
case ($intro_2\ \Gamma\ c_1\ mds\ mem_1\ c_2\ mem_2\ c\ \Gamma'$)
thus *?case*
using $\mathcal{R}_2$*-closed-glob-consistent* [*of* Γ] **and** $\mathcal{R}_3$*-aux.intro*$_2$
unfolding *closed-glob-consistent-def*
by (*simp*, *blast*)
next
case $intro_3$
thus *?case*
using $\mathcal{R}_3$*-aux.intro*$_3$
by (*simp*, *blast*)
qed

thus *?thesis* **by** *simp*
qed

lemma $\mathcal{R}$*-closed-glob-consistent*: *closed-glob-consistent* ($\mathcal{R}$ Γ')
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*: $x \in$ *aexp-vars e*
shows *to-total* Γ x = *Low*
using *assms*
by (*metis* (*full-types*) *Sec.exhaust imageI max-dom-def type-aexpr-elim*)

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

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

lemma *context-le-trans*: $[\![\Gamma \sqsubseteq_c \Gamma' ; \Gamma' \sqsubseteq_c \Gamma'']\!] \Longrightarrow \Gamma \sqsubseteq_c \Gamma''$
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* :
$[\![\vdash \Gamma \{ c \} \Gamma' ; c = Stop]\!] \Longrightarrow \Gamma \sqsubseteq_c \Gamma'$
apply (*induct rule*: *has-type.induct*)
apply *auto*
by (*metis context-le-trans*)

lemma *preservation*:
assumes *typed*: $\vdash \Gamma \{ c \} \Gamma'$
assumes *eval*: $\langle c, mds, mem\rangle \rightsquigarrow \langle c', mds', mem'\rangle$
shows $\exists \Gamma''.$ ($\vdash \Gamma'' \{ c' \} \Gamma'$) $\wedge$ (*mds-consistent mds* $\Gamma \longrightarrow$ *mds-consistent mds'* Γ'')
using *typed eval*
proof (*induct arbitrary*: c' *mds rule*: *has-type.induct*)
case (*anno-type* Γ'' Γ *upd* c_1 Γ')
hence $\langle c_1,$ *update-modes upd mds*, *mem*$\rangle \rightsquigarrow \langle c', mds', mem'\rangle$
by (*metis upd-elim*)
with *anno-type*(*3*) **obtain** Γ''' **where**
$\vdash \Gamma''' \{ c' \} \Gamma' \wedge$ (*mds-consistent* (*update-modes upd mds*) $\Gamma'' \longrightarrow$ *mds-consistent mds'* Γ''')
by *auto*

moreover
have *mds-consistent mds* $\Gamma \longrightarrow$ *mds-consistent* (*update-modes upd mds*) Γ''
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 $c' = Stop \wedge mds' = mds$
by (*metis skip-elim*)
thus *?case*
by (*metis stop-type*)
next
case ($assign_1$ x Γ e t c' *mds*)
hence $c' = Stop \wedge mds' = mds$
by (*metis assign-elim*)
thus *?case*
by (*metis stop-type*)
next
case ($assign_2$ x Γ e t c' *mds*)
hence $c' = Stop \wedge mds' = mds$
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* $assign_2(1)$)
apply (*metis* (*lifting*, *full-types*) *CollectD domI* $assign_2(1)$)
apply (*metis* (*lifting*) *CollectE domI* $assign_2(1)$)
apply (*metis* (*lifting*, *full-types*) *domD mem-Collect-eq*)
by (*metis* (*lifting*, *full-types*) *domD mem-Collect-eq*)
next
case (*if-type* Γ e *th el* Γ' c' *mds*)
thus *?case*
apply (*rule-tac* $x = \Gamma$ **in** *exI*)
by *force*
next
case (*while-type* Γ e c c' *mds*)
hence [*simp*]: $mds' = mds \wedge c' = If\ e\ (c\ ;;\ While\ e\ c)\ Stop$
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* $c_1 = Stop$)
assume [*simp*]: $c_1 = Stop$
with *seq-type* **have** [*simp*]: $mds' = mds \wedge c' = c_2$
by (*metis seq-stop-elim*)
thus *?case*
apply *auto*
by (*metis* (*lifting*) ‹$c_1 = Stop$› *context-le-refl seq-type*(*1*) *seq-type*(*3*) *stop-cxt sub*)
next
assume $c_1 \neq Stop$
then obtain c_1' **where** $\langle c_1, mds, mem\rangle \rightsquigarrow \langle c_1', mds', mem'\rangle \wedge c' = (c_1' ;; c_2)$
by (*metis seq-elim seq-type.prems*)
then obtain Γ''' **where** $\vdash \Gamma''' \{c_1'\} \Gamma_1 \wedge$
(mds-*consistent mds* $\Gamma \longrightarrow mds$-*consistent mds'* Γ''')
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*) ‹$\langle c_1, mds, mem\rangle \rightsquigarrow \langle c_1', mds', mem'\rangle \wedge c' = c_1' ;; c_2$› *has-type.seq-type*)
qed
next
case (*sub* Γ_1 c Γ_1' Γ_2 Γ_2' c' *mds*)
then obtain Γ'' **where** $\vdash \Gamma'' \{ c' \} \Gamma_1' \wedge$
(mds-*consistent mds* $\Gamma_1 \longrightarrow mds$-*consistent mds'* Γ'')
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 $\mathcal{R}_1$-*mem-eq*: $\langle c_1, mds, mem_1\rangle\ {\mathcal{R}^1}_{\Gamma'}\ \langle c_2, mds, mem_2\rangle \Longrightarrow mem_1 =_{mds}^{l} mem_2$
apply (*rule* $\mathcal{R}_1$-*elim*)
apply (*auto simp*: *tyenv-eq-def mds-consistent-def to-total-def*)
by (*metis* (*lifting*) *Sec.simps*(*1*) *low-mds-eq-def*)

lemma $\mathcal{R}_2$-*mem-eq*: $\langle c_1, mds, mem_1\rangle\ {\mathcal{R}^2}_{\Gamma'}\ \langle c_2, mds, mem_2\rangle \Longrightarrow mem_1 =_{mds}^{l} mem_2$
apply (*rule* $\mathcal{R}_2$-*elim*)

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

fun *bisim-helper* :: ((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf* ⇒
((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf* ⇒ *bool*
where
bisim-helper ⟨c_1, *mds*, mem_1⟩ ⟨c_2, mds_2, mem_2⟩ = $mem_1 =_{mds}{}^{l} mem_2$

lemma $\mathcal{R}_3$*-mem-eq*: ⟨c_1, *mds*, mem_1⟩ $\mathcal{R}^3{}_{\Gamma'}$ ⟨c_2, *mds*, mem_2⟩ ⟹ $mem_1 =_{mds}{}^{l}$ mem_2
apply (*subgoal-tac bisim-helper* ⟨c_1, *mds*, mem_1⟩ ⟨c_2, *mds*, mem_2⟩)
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*: ⟨c_1, *mds*, mem_1⟩ $\mathcal{R}^2{}_{\Gamma'}$ ⟨c_2, *mds*, mem_2⟩
assumes *eval*: ⟨c_1, *mds*, mem_1⟩ ⇝ ⟨c_1', *mds'*, mem_1'⟩
shows ∃ c_2' mem_2'. ⟨c_2, *mds*, mem_2⟩ ⇝ ⟨c_2', *mds'*, mem_2'⟩ ∧ ⟨c_1', *mds'*, mem_1'⟩ $\mathcal{R}^2{}_{\Gamma'}$ ⟨c_2', *mds'*, mem_2'⟩
proof −
from *case2* **have** *aux*: ⟨c_1, *mds*, mem_1⟩ ≈ ⟨c_2, *mds*, mem_2⟩ ∀ $x \in dom\ \Gamma'$. $\Gamma'\ x$ = *Some High*
by (*rule* $\mathcal{R}_2$*-elim*, *auto*)
with *eval* **obtain** c_2' mem_2' **where** c_2'*-props*:
⟨c_2, *mds*, mem_2⟩ ⇝ ⟨c_2', *mds'*, mem_2'⟩ ∧
⟨c_1', *mds'*, mem_1'⟩ ≈ ⟨c_2', *mds'*, mem_2'⟩
using *mm-equiv-strong-low-bisim strong-low-bisim-mm-def*
by *metis*

from *case2* **obtain** Γ_1 Γ_2 **where** ⊢ Γ_1 { c_1 } Γ' ⊢ Γ_2 { c_2 } Γ'
mds-consistent mds Γ_1 *mds-consistent mds* Γ_2
by (*metis* $\mathcal{R}_2$*-elim'*)
with *preservation* **and** c_2'*-props* **obtain** Γ_1' Γ_2' **where**
⊢ Γ_1' {c_1'} Γ' *mds-consistent mds'* Γ_1'
⊢ Γ_2' {c_2'} Γ' *mds-consistent mds'* Γ_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*: $[\![\neg$ *has-annotated-stop* (*c*@[*upd*]) $]\!] \Longrightarrow \neg$ *has-annotated-stop c*
apply (*cases c*)
by *auto*

lemma *typed-no-annotated-stop*:
$[\![\vdash \Gamma \{\ c\ \} \Gamma']\!] \Longrightarrow \neg$ *has-annotated-stop c*
by (*induct rule*: *has-type.induct*, *auto*)

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

by (*metis cxt-to-stmt.simps*(*1*) *eval*$_w$*-simplep.seq-stop eval*$_w$*p.unannotated eval*$_w$*p-eval*$_w$*-eq*)
next
assume $c_1 \neq Stop$
with *Seq* **show** *?case*
by (*metis eval*$_w$*.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 ev*$_B$ *mem bexp*)
apply (*metis cxt-to-stmt.simps*(*1*) *eval*$_w$*.unannotated eval*$_w$*-simple.if-true*)
by (*metis cxt-to-stmt.simps*(*1*) *eval*$_w$*.unannotated eval*$_w$*-simple.if-false*)
next
case (*While bexp c*)
thus *?case*
by (*metis cxt-to-stmt.simps*(*1*) *eval*$_w$*-simplep.while eval*$_w$*p.unannotated eval*$_w$*p-eval*$_w$*-eq*)
next
case *Stop*
thus *?case* **by** *blast*
qed

lemma *stop-bisim*:
assumes *bisim*: $\langle Stop, mds, mem_1\rangle \approx \langle c, mds, mem_2\rangle$
assumes *typeable*: $\vdash \Gamma \{ c \} \Gamma'$
shows $c = Stop$
proof (*rule ccontr*)
let $?lc_1 = \langle Stop, mds, mem_1\rangle$ **and**
$?lc_2 = \langle c, mds, mem_2\rangle$
assume $c \neq Stop$
from *typeable* **have** $\neg$ *has-annotated-stop c*
by (*metis typed-no-annotated-stop*)
with ‹$c \neq Stop$› **obtain** c' mds' $mem_2{}'$ **where** $?lc_2 \rightsquigarrow \langle c', mds', mem_2{}'\rangle$
using *not-stop-eval*
by *blast*
moreover
from *bisim* **have** $?lc_2 \approx ?lc_1$
by (*metis mm-equiv-sym*)
ultimately obtain $c_1{}'$ $mds_1{}'$ $mem_1{}'$
where $\langle Stop, mds, mem_1\rangle \rightsquigarrow \langle c_1{}', mds_1{}', mem_1{}'\rangle$
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*:
$[\![\vdash \Gamma \{ c_1 \} \Gamma' ;$
mds-consistent mds Γ ;
$mem_1 =_\Gamma mem_2$;
$\langle c_1, mds, mem_1\rangle \rightsquigarrow \langle {c_1}', mds', {mem_1}'\rangle]\!] \Longrightarrow$
$(\exists\ {c_2}'\ {mem_2}'.\ \langle c_1, mds, mem_2\rangle \rightsquigarrow \langle {c_2}', mds', {mem_2}'\rangle \wedge$
$\langle {c_1}', mds', {mem_1}'\rangle\ {\mathcal{R}^u}_{\Gamma'}\ \langle {c_2}', mds', {mem_2}'\rangle)$
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* ‹$c_1 = Stop$› **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 $\langle c_2, mds, mem_1\rangle\ {\mathcal{R}^1}_{\Gamma'}\ \langle c_2, mds, mem_2\rangle$
apply (*rule* $\mathcal{R}_1$*.intro* [*of* Γ])
by (*auto simp*: *seq-type* ‹$\vdash \Gamma \{ c_2 \} \Gamma'$›)
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*: ‹$c_1 = Stop$›)
by (*metis cxt-to-stmt.simps*(*1*) *eval*$_w$*-simplep.seq-stop eval*$_w$*p.unannotated eval*$_w$*p-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*:
$\langle c_1, mds, mem_2\rangle \rightsquigarrow \langle {c_2}'', mds', {mem_2}'\rangle \wedge \langle {c_1}'', mds', {mem_1}'\rangle\ {\mathcal{R}^u}_{\Gamma''}\ \langle {c_2}'', mds', {mem_2}'\rangle$
by (*metis seq-type*(*5*) *seq-type*(*6*))
hence $\langle {c_1}'' ;; c_2, mds', {mem_1}'\rangle\ {\mathcal{R}^u}_{\Gamma'}\ \langle {c_2}'' ;; c_2, mds', {mem_2}'\rangle$
apply (*rule conjE*)
apply (*erule* $\mathcal{R}$*-elim*, *auto*)
apply (*metis* $\mathcal{R}$*.intro*$_3$ $\mathcal{R}_3$*-aux.intro*$_1$ *seq-type*(*3*))
apply (*metis* $\mathcal{R}$*.intro*$_3$ $\mathcal{R}_3$*-aux.intro*$_2$ *seq-type*(*3*))
by (*metis* $\mathcal{R}$*.intro*$_3$ $\mathcal{R}_3$*-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 c_1''-props)
 qed
next
 case (anno-type Γ' Γ upd c Γ'' mds)
 have mem_1 =_Γ' mem_2
  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 c_2' mem_2' where (⟨c, update-modes upd mds, mem_2⟩ ⇝ ⟨c_2', mds',
mem_2'⟩ ∧
  ⟨c_1', mds', mem_1'⟩ R^u_Γ'' ⟨c_2', mds', mem_2'⟩)
  using anno-type
  apply auto
  by (metis ‹mem_1 =_Γ' mem_2› anno-type(1) upd-elim)
 thus ?case
  apply (rule-tac x = c_2' in exI)
  apply (rule-tac x = mem_2' in exI)
  apply auto
  by (metis cxt-to-stmt.simps(1) eval_w.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 c_1' = Stop mem_1' = mem_1
  using skip-elim
  by (metis, metis, metis)
 with skip-type have ⟨Stop, mds, mem_1⟩ R^1_Γ ⟨Stop, mds, mem_2⟩
  by auto
 thus ?case
  using R.intro_1 and unannotated [where c = Skip and E = []]
  apply auto
  by (metis eval_w-simple.skip)
next
 case (assign_1 x Γ e t mds)
 hence [simp]: c_1' = Stop mds' = mds mem_1' = mem_1 (x := ev_A mem_1 e)
  using assign-elim
  by (auto, metis)
 have mem_1 (x := ev_A mem_1 e) =_Γ mem_2 (x := ev_A mem_2 e)
 proof (cases to-total Γ x)
  assume to-total Γ x = High
  thus ?thesis
   using assign_1 tyenv-eq-def
   by auto
 next
  assume to-total Γ x = Low
```

with $assign_1$ **have** [*simp*]: $t = Low$
by (*metis less-eq-Sec-def to-total-def*)
hence $dma\ x = Low$
using $assign_1$ ‹*to-total* Γ *x* = *Low*›
by (*metis to-total-def*)
with $assign_1$ **have** ∀ $v \in aexp\text{-}vars\ e.\ to\text{-}total\ \Gamma\ v = Low$
using *type-low-vars-low*
by *auto*
thus *?thesis*
using $eval\text{-}vars\text{-}det_A$
apply (*auto simp*: *tyenv-eq-def*)
apply (*metis* (*no-types*) $assign_1(5)$ *tyenv-eq-def*)
by (*metis* $assign_1(5)$ *tyenv-eq-def*)
qed
hence $\langle x \leftarrow e,\ mds,\ mem_2\rangle \rightsquigarrow \langle Stop,\ mds',\ mem_2\ (x := ev_A\ mem_2\ e)\rangle$ $\langle Stop,\ mds',\ mem_1\ (x := ev_A\ mem_1\ e)\rangle\ \mathcal{R}^u{}_\Gamma\ \langle Stop,\ mds',\ mem_2\ (x := ev_A\ mem_2\ e)\rangle$
apply *auto*
apply (*metis cxt-to-stmt.simps*(*1*) $eval_w.unannotated\ eval_w\text{-}simple.assign$)
by (*rule* $\mathcal{R}.intro_1$, *auto simp*: $assign_1$)
thus *?case*
using ‹$c_1{}' = Stop$› **and** ‹$mem_1{}' = mem_1\ (x := ev_A\ mem_1\ e)$›
by *blast*
next
case ($assign_2\ x\ \Gamma\ e\ t\ mds$)
hence [*simp*]: $c_1{}' = Stop\ mds' = mds\ mem_1{}' = mem_1\ (x := ev_A\ mem_1\ e)$
using *assign-elim* $assign_2$
by (*auto*, *metis*)
let $?\Gamma' = \Gamma\ (x \mapsto t)$
have $\langle x \leftarrow e,\ mds,\ mem_2\rangle \rightsquigarrow \langle Stop,\ mds,\ mem_2\ (x := ev_A\ mem_2\ e)\rangle$
using $assign_2$
by (*metis cxt-to-stmt.simps*(*1*) $eval_w\text{-}simplep.assign\ eval_wp.unannotated\ eval_wp\text{-}eval_w\text{-}eq$)
moreover
have $\langle Stop,\ mds,\ mem_1\ (x := ev_A\ mem_1\ e)\rangle\ \mathcal{R}^1{}_{?\Gamma'}\ \langle Stop,\ mds,\ mem_2\ (x := ev_A\ mem_2\ e)\rangle$
proof (*auto*)
from $assign_2$ **show** *mds-consistent mds* $?\Gamma'$
apply (*simp add*: *mds-consistent-def*)
by (*metis* (*lifting*) *insert-absorb* $assign_2(1)$)
next
show $mem_1\ (x := ev_A\ mem_1\ e) =_{?\Gamma'} mem_2\ (x := ev_A\ mem_2\ e)$
unfolding *tyenv-eq-def*
proof (*auto*)
assume *to-total* ($\Gamma(x \mapsto t)$) $x = Low$
with ‹$\Gamma \vdash_a e \in t$› **have** $\bigwedge x.\ x \in aexp\text{-}vars\ e \Longrightarrow to\text{-}total\ \Gamma\ x = Low$
by (*metis* $assign_2.prems(1)$ *domI fun-upd-same option.sel to-total-def type-low-vars-low*)
thus $ev_A\ mem_1\ e = ev_A\ mem_2\ e$
by (*metis* $assign_2.prems(2)$ $eval\text{-}vars\text{-}det_A$ *tyenv-eq-def*)

next
fix y
assume $y \neq x$ **and** *to-total* $(\Gamma\ (x \mapsto t))\ y = Low$
thus $mem_1\ y = mem_2\ y$
by (*metis* (*full-types*) $assign_2.prems(2)$ *domD domI fun-upd-other to-total-def tyenv-eq-def*)
qed
qed
ultimately have $\langle x \leftarrow e,\ mds,\ mem_2\rangle \rightsquigarrow \langle Stop,\ mds',\ mem_2\ (x := ev_A\ mem_2\ e)\rangle$
$\langle Stop,\ mds',\ mem_1\ (x := ev_A\ mem_1\ e)\rangle\ \mathcal{R}^u{}_{\Gamma(x \mapsto t)}\ \langle Stop,\ mds',\ mem_2\ (x := ev_A\ mem_2\ e)\rangle$
using $\mathcal{R}.intro_1$
by *auto*
thus *?case*
using $\langle mds' = mds\rangle$ $\langle c_1{}' = Stop\rangle$ $\langle mem_1{}' = mem_1(x := ev_A\ mem_1\ e)\rangle$
by *blast*
next
case (*if-type* $\Gamma\ e\ th\ el\ \Gamma'$)
have $(\langle c_1{}',\ mds,\ mem_1\rangle, \langle c_1{}',\ mds,\ mem_2\rangle) \in \mathcal{R}\ \Gamma'$
apply (*rule* $intro_1$)
apply *clarify*
apply (*rule* $\mathcal{R}_1.intro$ [**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*: $ev_B\ mem_1\ e = ev_B\ mem_2\ e \Longrightarrow$ *?case*
proof −
assume $ev_B\ mem_1\ e = ev_B\ mem_2\ e$
with *if-type*(*8*) **have** $(\langle If\ e\ th\ el,\ mds,\ mem_2\rangle \rightsquigarrow \langle c_1{}',\ mds,\ mem_2\rangle)$
apply (*cases* $ev_B\ mem_1\ e$)
apply (*subgoal-tac* $c_1{}' = th$)
apply *clarify*
apply (*metis cxt-to-stmt.simps*(*1*) $eval_w$*-simplep.if-true* $eval_w p.unannotated$ $eval_w p$-$eval_w$*-eq if-type*(*8*))
apply (*metis if-elim if-type*(*8*))
apply (*subgoal-tac* $c_1{}' = el$)
apply (*metis* (*opaque-lifting*, *mono-tags*) *cxt-to-stmt.simps*(*1*) $eval_w.unannotated$ $eval_w$*-simple.if-false if-type*(*8*))
by (*metis if-elim if-type*(*8*))
thus *?thesis*
by (*metis* $\langle\langle c_1{}',\ mds,\ mem_1\rangle\ \mathcal{R}^u{}_{\Gamma'}\ \langle c_1{}',\ mds,\ mem_2\rangle\rangle$ *if-elim if-type*(*8*))
qed
have $mem_1 =_{mds}{}^{l}\ mem_2$
apply (*auto simp*: *low-mds-eq-def mds-consistent-def*)
apply (*subgoal-tac* $x \notin dom\ \Gamma$)
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 $t = High$
with *if-type* **show** *?thesis*
proof (*cases* $ev_B\ mem_1\ e = ev_B\ mem_2\ e$)
assume $ev_B\ mem_1\ e = ev_B\ mem_2\ e$
with *eq-condition* **show** *?thesis* **by** *auto*
next
assume *neq*: $ev_B\ mem_1\ e \neq ev_B\ mem_2\ e$
from *if-type* ‹$t = High$› **have** $th \sim_{mds} el$
by (*metis* ‹$\Gamma \vdash_b e \in t$›)
from *neq* **show** *?thesis*
proof (*cases* $ev_B\ mem_1\ e$)
assume $ev_B\ mem_1\ e$
hence ${c_1}' = th$
by (*metis* (*lifting*) *if-elim if-type*(*8*))
hence $\langle If\ e\ th\ el,\ mds,\ mem_2\rangle \rightsquigarrow \langle el,\ mds,\ mem_2\rangle$
by (*metis* ‹$ev_B\ mem_1\ e$› *cxt-to-stmt.simps*(*1*) $eval_w$*.unannotated* $eval_w$*-simple.if-false if-type*(*8*) *neq*)
moreover
with ‹$mem_1 =_{mds}{}^{l} mem_2$› **have** $\langle th,\ mds,\ mem_1\rangle \approx \langle el,\ mds,\ mem_2\rangle$
by (*metis low-indistinguishable-def* ‹$th \sim_{mds} el$›)
have $\forall\ x \in dom\ \Gamma'.\ \Gamma'\ x = Some\ High$
using *if-type* ‹$t = High$›
by (*metis* ‹$\Gamma \vdash_b e \in t$›)
have $\langle th,\ mds,\ mem_1\rangle\ {\mathcal{R}^2}_{\Gamma'}\ \langle el,\ mds,\ mem_2\rangle$
by (*metis* (*lifting*) $\mathcal{R}_2$*.intro* ‹$\forall x{\in}dom\ \Gamma'.\ \Gamma'\ x = Some\ High$› ‹$\langle th,\ mds,\ mem_1\rangle \approx \langle el,\ mds,\ mem_2\rangle$› *if-type*(*2*) *if-type*(*4*) *if-type*(*6*))
ultimately show *?thesis*
using $\mathcal{R}$*.intro*$_2$
apply *clarify*
by (*metis* ‹${c_1}' = th$› *if-elim if-type*(*8*))
next
assume ¬ $ev_B\ mem_1\ e$
hence [*simp*]: ${c_1}' = el$
by (*metis* (*lifting*) *if-type*(*8*) *if-elim*)
hence $\langle If\ e\ th\ el,\ mds,\ mem_2\rangle \rightsquigarrow \langle th,\ mds,\ mem_2\rangle$
by (*metis* (*opaque-lifting*, *mono-tags*) ‹¬ $ev_B\ mem_1\ e$› *cxt-to-stmt.simps*(*1*) $eval_w$*.unannotated* $eval_w$*-simple.if-true if-type*(*8*) *neq*)
moreover
from ‹$th \sim_{mds} el$› **have** $el \sim_{mds} th$
by (*metis low-indistinguishable-sym*)
with ‹$mem_1 =_{mds}{}^{l} mem_2$› **have** $\langle el,\ mds,\ mem_1\rangle \approx \langle th,\ mds,\ mem_2\rangle$
by (*metis low-indistinguishable-def*)
have $\forall\ x \in dom\ \Gamma'.\ \Gamma'\ x = Some\ High$
using *if-type* ‹$t = High$›
by (*metis* ‹$\Gamma \vdash_b e \in t$›)
have $\langle el,\ mds,\ mem_1\rangle\ {\mathcal{R}^2}_{\Gamma'}\ \langle th,\ mds,\ mem_2\rangle$
apply (*rule* $\mathcal{R}_2$*.intro* [**where** $\Gamma_1 = \Gamma$ **and** $\Gamma_2 = \Gamma$])

```
        by (auto simp: if-type ‹⟨el, mds, mem1⟩ ≈ ⟨th, mds, mem2⟩› ‹∀ x ∈ dom
Γ'. Γ' x = Some High›)
       ultimately show ?thesis
        using R.intro2
        apply clarify
        by (metis ‹c1' = el› if-elim if-type(8))
      qed
    qed
  next
    assume t = Low
    with if-type have ev_B mem1 e = ev_B mem2 e
      using eval-vars-det_B
      apply (simp add: tyenv-eq-def ‹Γ ⊢b e ∈ t› type-low-vars-low-b)
      by (metis (lifting) ‹Γ ⊢b e ∈ t› type-low-vars-low-b)
    with eq-condition show ?thesis by auto
  qed
next
  case (while-type Γ e c)
  hence [simp]: c1' = (If e (c ;; While e c) Stop) and
    [simp]: mds' = mds and
    [simp]: mem1' = mem1
    by (auto simp: while-elim)
  with while-type have ⟨While e c, mds, mem2⟩ ⇝ ⟨c1', mds, mem2⟩
   by (metis cxt-to-stmt.simps(1) eval_w-simplep.while eval_wp.unannotated eval_wp-eval_w-eq)
  moreover
  have ⟨c1', mds, mem1⟩ R^1_Γ ⟨c1', mds, mem2⟩
    apply (rule R1.intro [where Γ = Γ])
      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 Γ' = Γ])
      by (auto simp: while-type)
  ultimately show ?case
    using R.intro1 [of Γ]
    by (auto, blast)
next
  case (sub Γ1 c Γ1' Γ Γ' mds c1')
  hence dom Γ1 ⊆ dom Γ dom Γ' ⊆ dom Γ1'
    apply (metis (lifting) context-le-def equalityE)
    by (metis context-le-def sub(4) order-refl)
  hence mds-consistent mds Γ1
    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 mem1 =Γ1 mem2
    unfolding tyenv-eq-def
   by (metis (lifting, no-types) context-le-def less-eq-Sec-def sub.hyps(3) sub.prems(2)
to-total-def tyenv-eq-def)
```

ultimately obtain c_2' mem_2' **where** c_2'*-props*: $\langle c,\ mds,\ mem_2\rangle \rightsquigarrow \langle c_2',\ mds',\ mem_2'\rangle$
$\langle c_1',\ mds',\ mem_1'\rangle\ \mathcal{R}^u{}_{\Gamma_1'}\ \langle c_2',\ mds',\ mem_2'\rangle$
using *sub*
by *blast*
moreover
have $\bigwedge c_1\ mds\ mem_1\ c_2\ mem_2.\ \langle c_1,\ mds,\ mem_1\rangle\ \mathcal{R}^u{}_{\Gamma_1'}\ \langle c_2,\ mds,\ mem_2\rangle \Longrightarrow$
$\langle c_1,\ mds,\ mem_1\rangle\ \mathcal{R}^u{}_{\Gamma'}\ \langle c_2,\ mds,\ mem_2\rangle$
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*$_1\ \mathcal{R}^u{}_{\Gamma_1'}$ *?lc*$_2$
moreover
have *?lc*$_1\ \mathcal{R}^1{}_{\Gamma_1'}$ *?lc*$_2 \Longrightarrow$ *?lc*$_1\ \mathcal{R}^1{}_{\Gamma'}$ *?lc*$_2$
apply (*erule* $\mathcal{R}_1$*-elim*)
apply *auto*
by (*metis* (*lifting*) *has-type.sub sub*(*4*) *stop-cxt stop-type*)
moreover
have *?lc*$_1\ \mathcal{R}^2{}_{\Gamma_1'}$ *?lc*$_2 \Longrightarrow$ *?lc*$_1\ \mathcal{R}^2{}_{\Gamma'}$ *?lc*$_2$
proof −
assume *r2*: *?lc*$_1\ \mathcal{R}^2{}_{\Gamma_1'}$ *?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* Λ_1
mds-consistent mds Λ_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* ‹$\vdash \Lambda_2\ \{c_2\}\ \Gamma_1'$› *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*) ‹*dom* $\Gamma' \subseteq$ *dom* Γ_1'› *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'* ‹*mds-consistent mds* Λ_1› ‹*mds-consistent mds* Λ_2› *r2*)
qed
moreover
have *?lc*$_1\ \mathcal{R}^3{}_{\Gamma_1'}$ *?lc*$_2 \Longrightarrow$ *?lc*$_1\ \mathcal{R}^3{}_{\Gamma'}$ *?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*: $\vdash \Gamma\ \{c\}\ \Gamma_1'\ \langle c_1'',\ mds,\ mem_1\rangle\ \mathcal{R}^1{}_{\Gamma}\ \langle c_2''',\ mds,\ mem_2\rangle$

```
      hence ⊢ Γ {c} Γ′
        using context-le-refl has-type.sub sub.hyps(4)
        by blast
      with case1 show ⟨c₁, mds, mem₁⟩ 𝓡³_Γ′ ⟨c₂, mds, mem₂⟩
        using 𝓡₃-aux.intro₁ by simp
    next
      fix Γ c₁′′ c₂′′′ c′′
      assume [simp]: c₁ = c₁′′ ;; c′′ c₂ = c₂′′′ ;; c′′
      assume ⟨c₁′′, mds, mem₁⟩ 𝓡²_Γ ⟨c₂′′′, mds, mem₂⟩ ⊢ Γ {c′′} Γ₁′
      thus ⟨c₁, mds, mem₁⟩ 𝓡³_Γ′ ⟨c₂, mds, mem₂⟩
        using 𝓡₃-aux.intro₂
        apply simp
        apply (rule 𝓡₃-aux.intro₂ [where Γ = Γ])
         apply simp
        by (metis context-le-refl has-type.sub sub.hyps(4))
    next
      fix Γ c₁′′ c₂′′′ c′′
      assume [simp]: c₁ = c₁′′ ;; c′′ c₂ = c₂′′′ ;; c′′
      assume ⟨c₁′′, mds, mem₁⟩ 𝓡³_Γ ⟨c₂′′′, mds, mem₂⟩ ⊢ Γ {c′′} Γ₁′
      thus ⟨c₁, mds, mem₁⟩ 𝓡³_Γ′ ⟨c₂, mds, mem₂⟩
        using 𝓡₃-aux.intro₃
        apply auto
       by (metis (opaque-lifting, no-types) context-le-refl has-type.sub sub.hyps(4))
    qed
   ultimately show ?thesis c₁ mds mem₁ c₂ mem₂
    by (auto simp: 𝓡.intros)
  qed
  with c₂′-props show ?case
   by blast
qed


lemma 𝓡₁-weak-bisim:
  weak-bisim (𝓡₁ Γ′) (𝓡 Γ′)
  unfolding weak-bisim-def
  using 𝓡₁-elim 𝓡-typed-step
  by auto

lemma 𝓡-to-𝓡₃: ⟦ ⟨c₁, mds, mem₁⟩ 𝓡ᵘ_Γ ⟨c₂, mds, mem₂⟩ ; ⊢ Γ { c } Γ′ ⟧ ⟹
  ⟨c₁ ;; c, mds, mem₁⟩ 𝓡³_Γ′ ⟨c₂ ;; c, mds, mem₂⟩
  apply (erule 𝓡-elim)
  by auto

lemma 𝓡₂-implies-typeable: ⟨c₁, mds, mem₁⟩ 𝓡²_Γ′ ⟨c₂, mds, mem₂⟩ ⟹ ∃ Γ₁. ⊢
Γ₁ { c₂ } Γ′
  apply (erule 𝓡₂-elim)
  by auto

lemma 𝓡₃-weak-bisim:
```

```
  weak-bisim (R3 Γ') (R Γ')
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'⟩
R^u_Γ' ⟨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) eval_w-simplep.seq-stop
eval_wp.unannotated eval_wp-eval_w-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'⟩ R^u_Γ ⟨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 eval_w.seq)
          apply auto
          apply (rule R.intro3)
        by (metis (opaque-lifting, no-types) ‹⟨c1, mds, mem1⟩ ⇝ ⟨c1'', mds', mem1'⟩
```

$\wedge\ c_1' = c_1''\ ;;\ c\rangle$)
qed
next
case ($intro_2\ \Gamma\ c_1\ mds\ mem_1\ c_2\ mem_2\ cn\ \Gamma'$)
thus *?case*
proof (*cases* $c_1 = Stop$)
assume [*simp*]: $c_1 = Stop$
hence [*simp*]: $c_1' = cn\ mds' = mds\ mem_1' = mem_1$
using *eval intro*$_2$ *seq-stop-elim*
by *auto*
from *intro*$_2$ **have** *bisim*: $\langle c_1, mds, mem_1\rangle \approx \langle c_2, mds, mem_2\rangle$
by (*metis* (*lifting*) $\mathcal{R}_2$*-elim'*)
from *intro*$_2$ **obtain** Γ_1 **where** $\vdash \Gamma_1\ \{\ c_2\ \}\ \Gamma$
by (*metis* $\mathcal{R}_2$*-implies-typeable*)
with *bisim* **have** [*simp*]: $c_2 = Stop$
apply *auto*
apply (*rule stop-bisim* [*of mds mem*$_1$ c_2 *mem*$_2$ Γ_1 Γ])
by *simp-all*
have $\langle c_2\ ;;\ cn, mds, mem_2\rangle \rightsquigarrow \langle cn, mds', mem_2\rangle$
apply (*auto simp*: *intro*$_2$)
by (*metis cxt-to-stmt.simps*(*1*) *eval*$_w$*-simplep.seq-stop eval*$_w$*p.unannotated eval*$_w$*p-eval*$_w$*-eq*)
moreover
from *intro*$_2$(*1*) **have** *mds-consistent mds* Γ
apply *auto*
apply (*erule* $\mathcal{R}_2$*-elim*)
apply (*simp add*: *mds-consistent-def*)
by (*metis context-le-def stop-cxt*)
moreover
from *bisim* **have** $mem_1 =_{mds}^{l} mem_2$
by (*auto simp*: *mm-equiv.simps strong-low-bisim-mm-def*)
have $\forall\ x \in dom\ \Gamma.\ \Gamma\ x = Some\ High$
using *intro*$_2$(*1*)
by (*metis* $\mathcal{R}_2$*-elim'*)
hence $mem_1 =_\Gamma mem_2$
using ‹*mds-consistent mds* Γ› ‹$mem_1 =_{mds}^{l} mem_2$›
apply (*auto simp*: *tyenv-eq-def low-mds-eq-def mds-consistent-def*)
by (*metis Sec.simps*(*1*) ‹$\forall x{\in}dom\ \Gamma.\ \Gamma\ x = Some\ High$› ‹$mds' = mds$› *domI option.sel to-total-def*)
ultimately have $\langle cn, mds, mem_1\rangle\ {\mathcal{R}^1}_{\Gamma'}\ \langle cn, mds, mem_2\rangle$
by (*metis* (*lifting*) $\mathcal{R}_1$*.intro intro*$_2$(*2*))
thus *?thesis*
using $\mathcal{R}$*.intro*$_1$
apply *auto*
by (*metis* ‹$\langle c_2\ ;;\ cn, mds, mem_2\rangle \rightsquigarrow \langle cn, mds', mem_2\rangle$› ‹$c_2 = Stop$› ‹$mds' = mds$›)
next
assume $c_1 \neq Stop$
then obtain c_1''' **where** $c_1' = c_1'''\ ;;\ cn\ \langle c_1, mds, mem_1\rangle \rightsquigarrow \langle c_1''', mds',$

$mem_1{}'\rangle$
by (*metis* (*no-types*) *intro*$_2$*.prems 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 intro*$_2$
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*) *intro*$_2$ *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.intro*$_2$ $c_2{}'''$*-props intro*$_2$(*2*) *mem-Collect-eq case-prodI*)
ultimately show *?thesis*
using $\mathcal{R}$*.intro*$_3$
by (*metis* (*lifting*) $\mathcal{R}_3$*-aux.intro*$_2$ $\langle c_1{}' = c_1{}''' ;; cn\rangle$ $c_2{}'''$*-props intro*$_2$(*2*))
qed
next
case (*intro*$_3$ Γ c_1 *mds* mem_1 c_2 mem_2 *c* Γ')
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 intro*$_3$*.prems 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 *intro*$_3$(*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* $\mathcal{R}$*.intro*$_3$ $\mathcal{R}$*-to-*$\mathcal{R}_3$ $\langle\langle c_1{}'', mds', mem_1{}'\rangle\ \mathcal{R}^u{}_\Gamma\ \langle c_2{}'', mds', mem_2{}'\rangle\rangle$ $\langle c_1{}' = c_1{}'' ;; c\rangle$ *intro*$_3$(*3*))
apply (*metis* (*lifting*) $\mathcal{R}$*.intro*$_3$ $\mathcal{R}$*-to-*$\mathcal{R}_3$ $\langle\langle c_1{}'', mds', mem_1{}'\rangle\ \mathcal{R}^u{}_\Gamma\ \langle c_2{}'', mds', mem_2{}'\rangle\rangle$ $\langle c_1{}' = c_1{}'' ;; c\rangle$ *intro*$_3$(*3*))
by (*metis* (*lifting*) $\mathcal{R}$*.intro*$_3$ $\mathcal{R}_3$*-aux.intro*$_3$ $\langle c_1{}' = c_1{}'' ;; c\rangle$ *intro*$_3$(*3*))
qed
qed
}
thus *?thesis*
unfolding *weak-bisim-def*

by *auto*
qed

lemma $\mathcal{R}$*-bisim*: *strong-low-bisim-mm* ($\mathcal{R}$ Γ')
unfolding *strong-low-bisim-mm-def*
proof (*auto*)
from $\mathcal{R}$*-sym* **show** *sym* ($\mathcal{R}$ Γ') **.**
next
from $\mathcal{R}$*-closed-glob-consistent* **show** *closed-glob-consistent* ($\mathcal{R}$ Γ') **.**
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 ${=_{mds}}^l$ 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* $\mathcal{R}$*.intro*$_1$ [*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 Γ_0 :: *'Var TyEnv*
where $\Gamma_0\ x = None$

inductive *type-global* :: (*'Var*, *'AExp*, *'BExp*) *Stmt list* ⇒ *bool*
(‹⊢ -› [*120*] *1000*)
where
⟦ *list-all* (λ *c*. ⊢ Γ$_0$ { *c* } Γ$_0$) *cs* ;
∀ *mem*. *sound-mode-use* (*add-initial-modes cs*, *mem*) ⟧ ⟹
type-global cs

inductive-cases *type-global-elim*: ⊢ *cs*

lemma *mds$_s$-consistent*: *mds-consistent mds$_s$* Γ$_0$
by (*auto simp*: *mds$_s$-def mds-consistent-def* Γ$_0$*-def*)

lemma *typed-secure*:
⟦ ⊢ Γ$_0$ { *c* } Γ$_0$ ⟧ ⟹ *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* Γ$_0$ *c* Γ$_0$])
apply (*auto simp*: *mds$_s$-consistent to-total-def* Γ$_0$*-def*)
by (*metis empty-iff mds$_s$-def*)

lemma ⟦ *mds-consistent mds* Γ$_0$; *dma x* = *Low* ⟧ ⟹ *x* ∉ *mds AsmNoRead*
by (*auto simp*: *mds-consistent-def* Γ$_0$*-def*)

lemma *list-all-set*: ∀ *x* ∈ *set xs*. *P x* ⟹ *list-all P xs*
by (*metis* (*lifting*) *list-all-iff*)

theorem *type-soundness-global*:
assumes *typeable*: ⊢ *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* ∀ *c* ∈ *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** ‹⇝› *70*)
where
$x \rightsquigarrow y \equiv (x, y) \in eval_w$

fun *update-annos* :: '*Var Mds* ⇒ '*Var ModeUpd list* ⇒ '*Var Mds*
(**infix** ‹⊕› *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** ‹⊗› *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* ⊗ *annos* } (*mds* ⊕ *annos*) |
assign: ⟦ *x* ∉ *mds GuarNoWrite* ; *aexp-vars e* ∩ *mds GuarNoRead* = {} ⟧ ⟹
⊢ *mds* { (*x* ← *e*) ⊗ *annos* } (*mds* ⊕ *annos*) |
if-: ⟦ ⊢ (*mds* ⊕ *annos*) { *c*$_1$ } *mds*'' ;
⊢ (*mds* ⊕ *annos*) { *c*$_2$ } *mds*'' ;
bexp-vars e ∩ *mds GuarNoRead* = {} ⟧ ⟹
⊢ *mds* { *If e c*$_1$ *c*$_2$ ⊗ *annos* } *mds*'' |
while: ⟦ *mds*' = *mds* ⊕ *annos* ; ⊢ *mds*' { *c* } *mds*' ; *bexp-vars e* ∩ *mds*' *GuarNoRead* = {} ⟧ ⟹
⊢ *mds* { *While e c* ⊗ *annos* } *mds*' |
seq: ⟦ ⊢ *mds* { *c*$_1$ } *mds*' ; ⊢ *mds*' { *c*$_2$ } *mds*'' ⟧ ⟹ ⊢ *mds* { *c*$_1$;; *c*$_2$ } *mds*'' |
sub: ⟦ ⊢ *mds*$_2$ { *c* } *mds*$_2$' ; *mds*$_1$ ≤ *mds*$_2$; *mds*$_2$' ≤ *mds*$_1$' ⟧ ⟹
⊢ *mds*$_1$ { *c* } *mds*$_1$'

6.2 Soundness of the Type System

lemma *cxt-eval*:

⟦ ⟨*cxt-to-stmt* [] *c*, *mds*, *mem*⟩ ⇝ ⟨*cxt-to-stmt* [] *c′*, *mds′*, *mem′*⟩ ⟧ ⟹
⟨*c*, *mds*, *mem*⟩ ⇝ ⟨*c′*, *mds′*, *mem′*⟩
by *auto*

lemma *update-preserves-le*:
$mds_1 \leq mds_2 \Longrightarrow (mds_1 \oplus annos) \leq (mds_2 \oplus annos)$
proof (*induct annos arbitrary*: mds_1 mds_2)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons a annos* mds_1 mds_2)
hence *update-modes a* $mds_1 \leq$ *update-modes a* mds_2
by (*case-tac a*, *auto simp*: *le-fun-def*)
with *Cons* **show** *?case*
by *auto*
qed

lemma *doesnt-read-annos*:
doesnt-read c x ⟹ *doesnt-read* (*c* ⊗ *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*:
doesnt-modify c x ⟹ *doesnt-modify* (*c* ⊗ *annos*) *x*
unfolding *doesnt-modify-def*
apply *auto*
apply (*induct annos*)
apply *simp*
apply *auto*
by (*metis* (*lifting*) *upd-elim*)

lemma *stop-loc-reach*:
⟦ ⟨*c′*, *mds′*, *mem′*⟩ ∈ *loc-reach* ⟨*Stop*, *mds*, *mem*⟩ ⟧ ⟹
c′ = *Stop* ∧ *mds′* = *mds*
apply (*induct rule*: *loc-reach.induct*)
by (*auto simp*: *stop-no-eval*)

lemma *stop-doesnt-access*:
doesnt-modify Stop x ∧ *doesnt-read Stop x*

unfolding *doesnt-modify-def* **and** *doesnt-read-def*
using *stop-no-eval*
by *auto*

lemma *skip-eval-step*:
$\langle Skip \otimes annos, mds, mem\rangle \rightsquigarrow \langle Stop, mds \oplus annos, mem\rangle$
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*:
$[\![\langle Skip \otimes annos, mds, mem\rangle \rightsquigarrow \langle c', mds', mem'\rangle]\!] \Longrightarrow c' = Stop \wedge mds' = mds \oplus annos \wedge mem' = mem$
apply (*rule ccontr*)
apply (*insert skip-eval-step deterministic*)
apply *clarify*
apply *auto*
by *metis+*

lemma *skip-doesnt-read*:
doesnt-read (*Skip* $\otimes$ *annos*) *x*
apply (*rule doesnt-read-annos*)
apply (*auto simp*: *doesnt-read-def*)
by (*metis annotate.simps*(*1*) *skip-elim skip-eval-step*)

lemma *skip-doesnt-write*:
doesnt-modify (*Skip* $\otimes$ *annos*) *x*
apply (*rule doesnt-modify-annos*)
apply (*auto simp*: *doesnt-modify-def*)
by (*metis skip-elim*)

lemma *skip-loc-reach*:
$[\![\langle c', mds', mem'\rangle \in$ *loc-reach* $\langle Skip \otimes annos, mds, mem\rangle]\!] \Longrightarrow$
$(c' = Stop \wedge mds' = (mds \oplus annos)) \vee (c' = Skip \otimes annos \wedge mds' = mds)$
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*:
$[\![lc \in$ *loc-reach* $\langle Skip \otimes annos, mds, mem\rangle ; lc = \langle c', mds', mem'\rangle]\!] \Longrightarrow$ *doesnt-read* c' *x* $\wedge$ *doesnt-modify* c' *x*
apply (*subgoal-tac* $(c' = Stop \wedge mds' = (mds \oplus annos)) \vee (c' = Skip \otimes annos \wedge mds' = mds)$)

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*:
⟦ $x \neq y$ ⟧ ⟹ *doesnt-modify* (($x \leftarrow e$) ⊗ *annos*) *y*
apply (*rule doesnt-modify-annos*)
apply (*simp add*: *doesnt-modify-def*)
by (*metis assign-elim fun-upd-apply*)

lemma *assign-annos-eval*:
⟨($x \leftarrow e$) ⊗ *annos*, *mds*, *mem*⟩ ⇝ ⟨*Stop*, *mds* ⊕ *annos*, *mem* ($x := ev_A$ *mem e*)⟩
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*:
⟦ ⟨($x \leftarrow e$) ⊗ *annos*, *mds*, *mem*⟩ ⇝ ⟨c', mds', mem'⟩ ⟧ ⟹
$c' = Stop \wedge mds' = mds \oplus annos$
apply (*rule ccontr*)
apply (*insert deterministic assign-annos-eval*)
apply *auto*
apply (*metis* (*lifting*))
by *metis*

lemma *mem-upd-commute*:
⟦ $x \neq y$ ⟧ ⟹ *mem* ($x := v_1$, $y := v_2$) = *mem* ($y := v_2$, $x := v_1$)
by (*metis fun-upd-twist*)

lemma *assign-doesnt-read*:
⟦ $y \notin$ *aexp-vars e* ⟧ ⟹ *doesnt-read* (($x \leftarrow e$) ⊗ *annos*) *y*
apply (*rule doesnt-read-annos*)
proof (*cases* $x = y$)
assume $y \notin$ *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* $c' = Stop \wedge mds' = mds \wedge mem' = mem\ (x := ev_A\ mem\ e)$)
apply *simp*
apply (*rule disjI2*)
apply *clarify*
apply (*rule cxt-eval*)
apply (*rule* $eval_w$*.unannotated*)
apply *simp*
apply (*metis* (*opaque-lifting*, *no-types*) ‹$x = y$› ‹$y \notin$ *aexp-vars e*› $eval_w$*-simple.assign eval-vars-det*$_A$ *fun-upd-apply fun-upd-upd*)
by (*metis assign-elim*)
next
assume *asms*: $y \notin$ *aexp-vars e* $x \neq y$
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* $c' = Stop \wedge mds' = mds \wedge mem' = mem\ (x := ev_A\ mem\ e)$)
apply *simp*
apply (*rule disjI1*)
apply (*insert asms*)
apply *clarify*
apply (*subgoal-tac mem* ($x := ev_A\ mem\ e$, $y := v$) = *mem* ($y := v$, $x := ev_A\ mem\ e$))
apply *simp*
apply (*rule cxt-eval*)
apply (*rule* $eval_w$*.unannotated*)
apply (*metis* $eval_w$*-simple.assign eval-vars-det*$_A$ *fun-upd-apply*)
apply (*metis mem-upd-commute*)
by (*metis assign-elim*)
qed

lemma *assign-loc-reach*:
$[\![\langle c', mds', mem' \rangle \in$ *loc-reach* $\langle (x \leftarrow e) \otimes annos, mds, mem \rangle\ ]\!] \Longrightarrow$
$(c' = Stop \wedge mds' = (mds \oplus annos)) \vee (c' = (x \leftarrow e) \otimes annos \wedge mds' = mds)$
apply (*induct rule*: *loc-reach.induct*)
apply *simp-all*
by (*metis assign-annos-eval-elim stop-no-eval*)

lemma *if-doesnt-modify*:
doesnt-modify (*If e* c_1 $c_2 \otimes annos$) x
apply (*rule doesnt-modify-annos*)
by (*auto simp*: *doesnt-modify-def*)

lemma $vars\text{-}eval_B$:
$x \notin$ *bexp-vars* $e \Longrightarrow ev_B\ mem\ e = ev_B\ (mem\ (x := v))\ e$

by (*metis* (*lifting*) *eval-vars-det*$_B$ *fun-upd-other*)

lemma *if-doesnt-read*:
x ∉ *bexp-vars e* ⟹ *doesnt-read* (*If e c*$_1$ *c*$_2$ ⊗ *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*′ = *c*$_1$ ∧ *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*′ = *c*$_2$ ∧ *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*:
⟦ *ev*$_B$ *mem e* ⟧ ⟹
⟨*If e c*$_1$ *c*$_2$ ⊗ *annos*, *mds*, *mem*⟩ ⇝ ⟨*c*$_1$, *mds* ⊕ *annos*, *mem*⟩
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*:
⟦ ¬ *ev*$_B$ *mem e* ⟧ ⟹
⟨*If e c*$_1$ *c*$_2$ ⊗ *annos*, *mds*, *mem*⟩ ⇝ ⟨*c*$_2$, *mds* ⊕ *annos*, *mem*⟩
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*:
⟦ ⟨*If e c*$_1$ *c*$_2$ ⊗ *annos*, *mds*, *mem*⟩ ⇝ ⟨*c*′, *mds*′, *mem*′⟩ ⟧ ⟹
((*c*′ = *c*$_1$ ∧ *ev*$_B$ *mem e*) ∨ (*c*′ = *c*$_2$ ∧ ¬ *ev*$_B$ *mem e*)) ∧ *mds*′ = *mds* ⊕ *annos* ∧
mem′ = *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*′:
⟦ ⟨*If e c*$_1$ *c*$_2$, *mds*, *mem*⟩ ⇝ ⟨*c*′, *mds*′, *mem*′⟩ ⟧ ⟹

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

lemma *loc-reach-refl'*:
$\langle c, mds, mem\rangle \in$ *loc-reach* $\langle c, mds, mem\rangle$
apply (*subgoal-tac* $\exists\ lc.\ lc \in$ *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*:
$[\![\langle c', mds', mem'\rangle \in$ *loc-reach* $\langle$*If e* $c_1\ c_2 \otimes annos, mds, mem\rangle\ ]\!] \Longrightarrow$
$(c' =$ *If e* $c_1\ c_2 \otimes annos \wedge mds' = mds) \vee$
$(\exists\ mem''.\ \langle c', mds', mem'\rangle \in$ *loc-reach* $\langle c_1, mds \oplus annos, mem''\rangle) \vee$
$(\exists\ mem''.\ \langle c', mds', mem'\rangle \in$ *loc-reach* $\langle c_2, mds \oplus annos, mem''\rangle)$
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'*:
$[\![\langle c', mds', mem'\rangle \in$ *loc-reach* $\langle$*If e* $c_1\ c_2, mds, mem\rangle\ ]\!] \Longrightarrow$
$(c' =$ *If e* $c_1\ c_2 \wedge mds' = mds) \vee$
$(\exists\ mem''.\ \langle c', mds', mem'\rangle \in$ *loc-reach* $\langle c_1, mds, mem''\rangle) \vee$
$(\exists\ mem''.\ \langle c', mds', mem'\rangle \in$ *loc-reach* $\langle c_2, mds, mem''\rangle)$
using *if-loc-reach* [**where** *annos* = []]
by *simp*

lemma *seq-loc-reach*:
$[\![\langle c', mds', mem'\rangle \in$ *loc-reach* $\langle c_1\ ;;\ c_2, mds, mem\rangle\ ]\!] \Longrightarrow$
$(\exists\ c''.\ c' = c''\ ;;\ c_2 \wedge \langle c'', mds', mem'\rangle \in$ *loc-reach* $\langle c_1, mds, mem\rangle) \vee$
$(\exists\ c''\ mds''\ mem''.\ \langle$*Stop*$, mds'', mem''\rangle \in$ *loc-reach* $\langle c_1, mds, mem\rangle \wedge$
$\langle c', mds', mem'\rangle \in$ *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* ⟧ ⟹ *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'* ∧ *mds'* = *mds* ∧ *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* ⟧ ⟹ *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'*: ⟨*Stop* ;; *c*, *mds*, *mem*⟩ ⇝ ⟨*c'*, *mds'*, *mem'*⟩

lemma *seq-stop-elim*: ⟨*Stop* ;; *c*, *mds*, *mem*⟩ ⇝ ⟨*c'*, *mds'*, *mem'*⟩ ⟹
c' = *c* ∧ *mds'* = *mds* ∧ *mem'* = *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*:
⟦ ⟨*Stop*, *mds'*, *mem'*⟩ ∈ *loc-reach* ⟨*c*$_1$;; *c*$_2$, *mds*, *mem*⟩ ⟧ ⟹
∃ *mds''* *mem''*. ⟨*Stop*, *mds''*, *mem''*⟩ ∈ *loc-reach* ⟨*c*$_1$, *mds*, *mem*⟩ ∧
⟨*Stop*, *mds'*, *mem'*⟩ ∈ *loc-reach* ⟨*c*$_2$, *mds''*, *mem''*⟩
apply (*drule seq-loc-reach*)
by (*metis Stmt.simps*(*41*))

lemma *while-eval*:
⟨*While e c* ⊗ *annos*, *mds*, *mem*⟩ ⇝ ⟨(*If e* (*c* ;; *While e c*) *Stop*), *mds* ⊕ *annos*, *mem*⟩
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'*:
$\langle$*While e c, mds, mem*$\rangle \rightsquigarrow \langle$*If e* (*c* ;; *While e c*) *Stop, mds, mem*$\rangle$
using *while-eval* [**where** *annos* = []]
by *auto*

lemma *while-eval-elim*:
$[\![\langle$*While e c* $\otimes$ *annos, mds, mem*$\rangle \rightsquigarrow \langle c', mds', mem' \rangle]\!] \Longrightarrow$
(*c'* = *If e* (*c* ;; *While e c*) *Stop* $\wedge$ *mds'* = *mds* $\oplus$ *annos* $\wedge$ *mem'* = *mem*)
apply (*rule ccontr*)
apply (*insert while-eval deterministic*)
by *blast*

lemma *while-eval-elim'*:
$[\![\langle$*While e c, mds, mem*$\rangle \rightsquigarrow \langle c', mds', mem' \rangle]\!] \Longrightarrow$
(*c'* = *If e* (*c* ;; *While e c*) *Stop* $\wedge$ *mds'* = *mds* $\wedge$ *mem'* = *mem*)
using *while-eval-elim* [**where** *annos* = []]
by *auto*

lemma *while-doesnt-read*:
$[\![x \notin$ *bexp-vars e* $]\!] \Longrightarrow$ *doesnt-read* (*While e c* $\otimes$ *annos*) *x*
unfolding *doesnt-read-def*
using *while-eval while-eval-elim*
by *metis*

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

lemma *disjE3*:
$[\![A \vee B \vee C ; A \Longrightarrow P ; B \Longrightarrow P ; C \Longrightarrow P]\!] \Longrightarrow P$
by *auto*

lemma *disjE5*:
$[\![A \vee B \vee C \vee D \vee E ; A \Longrightarrow P ; B \Longrightarrow P ; C \Longrightarrow P ; D \Longrightarrow P ; E \Longrightarrow P]\!]$
$\Longrightarrow P$
by *auto*

lemma *if-doesnt-read'*:
x $\notin$ *bexp-vars e* $\Longrightarrow$ *doesnt-read* (*If e* c_1 c_2) *x*
using *if-doesnt-read* [**where** *annos* = []]
by *auto*

theorem *mode-type-sound*:
assumes *typeable*: $\vdash$ mds_1 { *c* } mds_1'
assumes *mode-le*: $mds_2 \leq mds_1$
shows $\forall$ *mem*. ($\langle$*Stop*, mds_2', *mem'*$\rangle \in$ *loc-reach* $\langle c, mds_2, mem \rangle \longrightarrow mds_2' \leq$

$mds_1{}'$) ∧
locally-sound-mode-use ⟨*c*, mds_2, *mem*⟩
using *typeable mode-le*
proof (*induct arbitrary*: mds_2 $mds_2{}'$ *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 ∀ *mem*. *locally-sound-mode-use* ⟨(*x* ← *e*) ⊗ *annos*, mds_2, *mem*⟩
unfolding *locally-sound-mode-use-def*
proof (*clarify*)
fix *mem c′ mds′ mem′ y*
assume *asm*: ⟨*c′*, *mds′*, *mem′*⟩ ∈ *loc-reach* ⟨(*x* ← *e*) ⊗ *annos*, mds_2, *mem*⟩
hence *c′* = (*x* ← *e*) ⊗ *annos* ∧ *mds′* = mds_2 ∨ *c′* = *Stop* ∧ *mds′* = mds_2 ⊕ *annos*
using *assign-loc-reach* **by** *blast*
thus (*y* ∈ *mds′ GuarNoRead* ⟶ *doesnt-read c′ y*) ∧
(*y* ∈ *mds′ GuarNoWrite* ⟶ *doesnt-modify c′ y*)
proof
assume *c′* = (*x* ← *e*) ⊗ *annos* ∧ *mds′* = mds_2
thus *?thesis*
proof (*auto*)
assume *y* ∈ mds_2 *GuarNoRead*
hence *y* ∉ *aexp-vars e*
by (*metis IntD2 IntI assign.hyps*(*2*) *assign.prems empty-iff inf-apply le-iff-inf*)
with *assign-doesnt-read* **show** *doesnt-read* ((*x* ← *e*) ⊗ *annos*) *y*
by *blast*
next
assume *y* ∈ mds_2 *GuarNoWrite*
hence *x* ≠ *y*
by (*metis assign.hyps*(*1*) *assign.prems in-mono le-fun-def*)
with *assign-doesnt-modify* **show** *doesnt-modify* ((*x* ← *e*) ⊗ *annos*) *y*
by *blast*
qed
next
assume *c′* = *Stop* ∧ *mds′* = mds_2 ⊕ *annos*
with *stop-doesnt-access* **show** *?thesis* **by** *blast*
qed
qed
thus *?case*
apply *auto*
by (*metis assign.prems assign-annos-eval assign-loc-reach stop-no-eval update-preserves-le*)
next

case (*if- mds annos* c_1 *mds''* c_2 *e*)
let *?c* = (*If e* c_1 c_2) $\otimes$ *annos*
from *if-* **have** *modes-le'*: $mds_2 \oplus annos \leq mds \oplus annos$
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 $\langle Stop, mds_2', mem' \rangle \in$ *loc-reach* $\langle$*If e* c_1 $c_2 \otimes annos, mds_2, mem\rangle$
with *modes-le'* **and** *if-* **show** $mds_2' \leq mds''$
by (*metis if-eval-false if-eval-true if-loc-reach stop-no-eval*)
next
fix *mem* c' *mds'* *mem'* *x*
assume $\langle c', mds', mem' \rangle \in$ *loc-reach* $\langle$*If e* c_1 $c_2 \otimes annos, mds_2, mem\rangle$
hence $(c' =$ *If e* c_1 $c_2 \otimes annos \wedge mds' = mds_2) \vee$
$(\exists\ mem''.\ \langle c', mds', mem' \rangle \in$ *loc-reach* $\langle c_1, mds_2 \oplus annos, mem''\rangle) \vee$
$(\exists\ mem''.\ \langle c', mds', mem' \rangle \in$ *loc-reach* $\langle c_2, mds_2 \oplus annos, mem''\rangle)$
using *if-loc-reach* **by** *blast*
thus $(x \in mds'$ *GuarNoRead* $\longrightarrow$ *doesnt-read* c' *x*) $\wedge$
$(x \in mds'$ *GuarNoWrite* $\longrightarrow$ *doesnt-modify* c' *x*)
proof
assume $c' =$ *If e* c_1 $c_2 \otimes annos \wedge mds' = mds_2$
thus *?thesis*
proof (*auto*)
assume $x \in mds_2$ *GuarNoRead*
with ‹*bexp-vars e* $\cap$ *mds GuarNoRead* = {}› **and** ‹$mds_2 \leq mds$› **have** $x \notin$ *bexp-vars e*
by (*metis IntD2 disjoint-iff-not-equal inf-fun-def le-iff-inf*)
thus *doesnt-read* (*If e* c_1 $c_2 \otimes annos$) *x*
using *if-doesnt-read* **by** *blast*
next
assume $x \in mds_2$ *GuarNoWrite*
thus *doesnt-modify* (*If e* c_1 $c_2 \otimes annos$) *x*
using *if-doesnt-modify* **by** *blast*
qed
next
assume $(\exists\, mem''.\ \langle c', mds', mem' \rangle \in$ *loc-reach* $\langle c_1, mds_2 \oplus annos, mem''\rangle)$
$\vee$
$(\exists\, mem''.\ \langle c', mds', mem' \rangle \in$ *loc-reach* $\langle c_2, mds_2 \oplus annos, mem''\rangle)$
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-reach* $\langle$*While e c* $\otimes$ *annos*, mds_2, *mem*$\rangle \Longrightarrow$
$c' =$ *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-reach* $\langle c, mds_3, mem'' \rangle)$
proof −
fix *mem c' mds' mem'*
assume $\langle c', mds', mem' \rangle \in$ *loc-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* ‹$mds_2 \oplus annos \leq mds \oplus annos$› *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-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** *bexp-vars e* $\cap$ ($mds_2 \oplus annos$) *GuarNoRead* = {}
by (*metis* (*lifting*, *no-types*) *Int-empty-right Int-left-commute* ‹$mds_2 \oplus annos \leq mds \oplus annos$› *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** $lc = \langle While\ e\ c \otimes annos,\ mds_2,\ mem\rangle$
assume $\langle c',\ mds',\ mem'\rangle \in loc\text{-}reach\ lc$
thus $\forall\ x.\ (x \in mds'\ GuarNoRead \longrightarrow doesnt\text{-}read\ c'\ x) \wedge$
$(x \in mds'\ GuarNoWrite \longrightarrow doesnt\text{-}modify\ c'\ x)$
apply (*simp add*: *lc-def*)
apply (*drule while-loc-reach*)
apply (*erule disjE5*)
proof (*auto del*: *conjI*)
fix x
show $(x \in mds_2\ GuarNoRead \longrightarrow doesnt\text{-}read\ (While\ e\ c \otimes annos)\ x) \wedge$
$(x \in mds_2\ GuarNoWrite \longrightarrow doesnt\text{-}modify\ (While\ e\ c \otimes annos)\ x)$
unfolding *doesnt-read-def* **and** *doesnt-modify-def*
using *while-eval* **and** *while-eval-elim*
by *blast*
next
fix x
assume $mds' \leq mdsa$
let $?c' = If\ e\ (c\ ;;\ While\ e\ c)\ Stop$
have $x \in mds'\ GuarNoRead \longrightarrow doesnt\text{-}read\ ?c'\ x$
apply *clarify*
apply (*rule if-doesnt-read'*)
by (*metis IntI* ‹$mds' \leq mdsa$› *empty-iff le-fun-def rev-subsetD while.hyps*(*1*) *while.hyps*(*4*))
moreover
have $x \in mds'\ GuarNoWrite \longrightarrow doesnt\text{-}modify\ ?c'\ x$
by (*metis annotate.simps*(*1*) *if-doesnt-modify*)
ultimately show $(x \in mds'\ GuarNoRead \longrightarrow doesnt\text{-}read\ ?c'\ x) \wedge$
$(x \in mds'\ GuarNoWrite \longrightarrow doesnt\text{-}modify\ ?c'\ x)$ **..**
next
fix x
assume $mds' \leq mdsa$
show $(x \in mds'\ GuarNoRead \longrightarrow doesnt\text{-}read\ Stop\ x) \wedge$
$(x \in mds'\ GuarNoWrite \longrightarrow doesnt\text{-}modify\ Stop\ x)$
by (*metis stop-doesnt-access*)
next
fix c'' x mem'' mds_3
assume $mds_3 \leq mdsa$
assume $\langle c'',\ mds',\ mem'\rangle \in loc\text{-}reach\ \langle c,\ mds_3,\ mem''\rangle$
thus $(x \in mds'\ GuarNoRead \longrightarrow doesnt\text{-}read\ (c''\ ;;\ While\ e\ c)\ x) \wedge$
$(x \in mds'\ GuarNoWrite \longrightarrow doesnt\text{-}modify\ (c''\ ;;\ While\ e\ c)\ x)$
apply *auto*
apply (*rule seq-doesnt-read*)
apply (*insert while*(*3*))
apply (*metis* ‹$mds_3 \leq mdsa$› *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
  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.