A Dependent Security Type System for Concurrent Imperative Programs

Toby Murray, Robert Sison, Edward Pierzchalski and Christine Rizkallah

December 14, 2021

Abstract

The paper "Compositional Verification and Refinement of Concurrent Value-Dependent Noninterference" by Murray et. al. [MSPR16] presents a dependent security type system for compositionally verifying a value-dependent noninterference property, defined in [Mur15], for concurrent programs. This development formalises that security definition, the type system and its soundness proof, and demonstrates its application on some small examples. It was derived from the `SIFUM_Type_Systems` AFP entry [GMS14], by Sylvia Grewe, Heiko Mantel and Daniel Schoepe and which itself formalises the work in [MSS11], and whose structure it inherits.

The formalization includes the following parts:

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

Examples are also present in the formalisation in the `Examples/` directory.

Contents

1 Preliminaries

theory *Preliminaries*
imports *Main*
begin

Possible modes for variables:

datatype *Mode* = *AsmNoReadOrWrite* | *AsmNoWrite* | *GuarNoReadOrWrite* | *GuarNoWrite*

We consider a two-element security lattice:

datatype *Sec* = *High* | *Low*

Sec forms a (complete) lattice:

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

definition *top-Sec-def*: *top* = *High*
definition *sup-Sec-def*: *sup d1 d2* = (*if* (*d1* = *High* ∨ *d2* = *High*) *then High else Low*)
definition *inf-Sec-def*: *inf d1 d2* = (*if* (*d1* = *Low* ∨ *d2* = *Low*) *then Low else High*)
definition *bot-Sec-def*: *bot* = *Low*
definition *less-eq-Sec-def*: *d1* ≤ *d2* = (*d1* = *d2* ∨ *d1* = *Low*)
definition *less-Sec-def*: *d1* < *d2* = (*d1* = *Low* ∧ *d2* = *High*)
definition *Sup-Sec-def*: *Sup S* = (*if* (*High* ∈ *S*) *then High else Low*)
definition *Inf-Sec-def*: *Inf S* = (*if* (*Low* ∈ *S*) *then Low else High*)

instance
apply (*intro-classes*)

using *Sec.exhaust less-Sec-def less-eq-Sec-def inf-Sec-def sup-Sec-def*
apply *auto*[*10*]
apply (*metis Inf-Sec-def Sec.exhaust less-eq-Sec-def*)
apply (*metis Inf-Sec-def Sec.exhaust less-eq-Sec-def*)
using *Sec.exhaust less-Sec-def less-eq-Sec-def inf-Sec-def sup-Sec-def Inf-Sec-def Sup-Sec-def top-Sec-def bot-Sec-def*
by *auto*
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-init* =
fixes *dma* :: (*'Var*,*'Val*) *Mem* ⇒ *'Var* ⇒ *Sec*
fixes $\mathcal{C}$*-vars* :: *'Var* ⇒ *'Var set*
fixes $\mathcal{C}$ :: *'Var set*
fixes *eval* :: (*'Com*, *'Var*, *'Val*) *LocalConf rel*
fixes *some-val* :: *'Val*
fixes *INIT* :: (*'Var*,*'Val*) *Mem* ⇒ *bool*
assumes *deterministic*: ⟦ (*lc*, *lc'*) ∈ *eval*; (*lc*, *lc''*) ∈ *eval* ⟧ ⟹ *lc'* = *lc''*
assumes *finite-memory*: *finite* {(*x*::*'Var*). *True*}
defines $\mathcal{C} \equiv \bigcup x.\ \mathcal{C}$*-vars x*
assumes $\mathcal{C}$*-vars-*$\mathcal{C}$: $x \in \mathcal{C} \Longrightarrow \mathcal{C}$*-vars x* = {}
assumes *dma-*$\mathcal{C}$*-vars*: ∀ *x*∈$\mathcal{C}$*-vars y*. *mem*$_1$ *x* = *mem*$_2$ *x* ⟹ *dma mem*$_1$ *y* = *dma mem*$_2$ *y*
assumes $\mathcal{C}$*-Low*: ∀ *x*∈$\mathcal{C}$. *dma mem x* = *Low*

locale *sifum-security* = *sifum-security-init dma* $\mathcal{C}$*-vars* $\mathcal{C}$ *eval some-val* λ*-.True*
for *dma* :: (*'Var*,*'Val*) *Mem* ⇒ *'Var* ⇒ *Sec*
and $\mathcal{C}$*-vars* :: *'Var* ⇒ *'Var set*
and $\mathcal{C}$ :: *'Var set*
and *eval* :: (*'Com*, *'Var*, *'Val*) *LocalConf rel*
and *some-val* :: *'Val*

context *sifum-security-init* **begin**

lemma *$\mathcal{C}$-vars-subset-$\mathcal{C}$*:
$\mathcal{C}$-vars x $\subseteq$ $\mathcal{C}$
by(*force simp*: *$\mathcal{C}$-def*)

lemma *dma-$\mathcal{C}$*:
$\forall\, x{\in}\mathcal{C}.\ mem_1\ x = mem_2\ x \Longrightarrow dma\ mem_1 = dma\ mem_2$
proof
fix *y*
assume $\forall\, x{\in}\mathcal{C}.\ mem_1\ x = mem_2\ x$
hence $\forall\, x{\in}\mathcal{C}$*-vars y*. $mem_1\ x = mem_2\ x$
using *$\mathcal{C}$-vars-subset-$\mathcal{C}$* **by** *blast*
thus $dma\ mem_1\ y = dma\ mem_2\ y$
by(*rule dma-$\mathcal{C}$-vars*)
qed

end

lemma *my-trancl-induct* [*consumes 1*, *case-names base step*]:
$[\![(a,\ b) \in r^+;$
$P\ a\ ;$
$\bigwedge x\ y.\ [\![(x,\ y) \in r;\ P\ x]\!] \Longrightarrow P\ y]\!] \Longrightarrow P\ b$
by (*induct rule*: *trancl.induct*, *blast+*)

lemma *my-trancl-step-induct* [*consumes 1*, *case-names base step*]:
$[\![(a,\ b) \in r^+;$
$\bigwedge x\ y.\ (x,\ y) \in r \Longrightarrow P\ x\ y;$
$\bigwedge x\ y\ z.\ P\ x\ y \Longrightarrow (y,\ z) \in r \Longrightarrow P\ x\ z]\!] \Longrightarrow P\ a\ b$
by (*induct rule*: *trancl-induct*, *blast+*)

lemma *my-trancl-big-step-induct* [*consumes 1*, *case-names base step*]:
$[\![(a,\ b) \in r^+;$
$\bigwedge x\ y.\ (x,\ y) \in r \Longrightarrow P\ x\ y;$
$\bigwedge x\ y\ z.\ (x,\ y) \in r^+ \Longrightarrow P\ x\ y \Longrightarrow (y,\ z) \in r \Longrightarrow P\ y\ z \Longrightarrow P\ x\ z]\!] \Longrightarrow P\ a\ b$
by (*induct rule*: *trancl.induct*, *blast+*)

lemmas *my-trancl-step-induct3* =
my-trancl-step-induct[*of* ((*ax*,*ay*), *az*) ((*bx*,*by*), *bz*), *split-format* (*complete*),
consumes 1, *case-names step*]

lemmas *my-trancl-big-step-induct3* =
my-trancl-big-step-induct[*of* ((*ax*,*ay*), *az*) ((*bx*,*by*), *bz*), *split-format* (*complete*),
consumes 1, *case-names base step*]

end

2 Definition of the SIFUM-Security Property

theory *Security*
imports *Preliminaries*
begin

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 (*v*$_1$, *v*$_2$) ⇒ *if first then v*$_1$ *else v*$_2$
        | *None* ⇒ *mem x*)

abbreviation *apply-adaptation*$_1$ ::
  (*'Var*, *'Val*) *Mem* ⇒ (*'Var*, *'Val*) *adaptation* ⇒ (*'Var*, *'Val*) *Mem*
  (- [||$_1$ -] [*900*, *0*] *1000*)
  where *mem* [||$_1$ *A*] ≡ *apply-adaptation True mem A*

abbreviation *apply-adaptation*$_2$ ::
  (*'Var*, *'Val*) *Mem* ⇒ (*'Var*, *'Val*) *adaptation* ⇒ (*'Var*, *'Val*) *Mem*
  (- [||$_2$ -] [*900*, *0*] *1000*)
  where *mem* [||$_2$ *A*] ≡ *apply-adaptation False mem A*

definition
  var-asm-not-written :: *'Var Mds* ⇒ *'Var* ⇒ *bool*
where
  var-asm-not-written mds x ≡ *x* ∈ *mds AsmNoWrite* ∨ *x* ∈ *mds AsmNoReadOrWrite*

context *sifum-security-init* **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* × *nat* × (-,-,-) *GlobalConf*) *set*
  and *meval-abv* :: - ⇒ - ⇒ - ⇒ *bool* (- ⇝$_-$ - *70*)

where
conf $\rightsquigarrow_k$ *conf'* $\equiv$ (*conf*, *k*, *conf'*) $\in$ *meval* |
meval-intro [*iff*]: $[\![$ (*cms* ! *n*, *mem*) $\rightsquigarrow$ (*cm'*, *mem'*); *n* < *length cms* $]\!] \Longrightarrow$
((*cms*, *mem*), *n*, (*cms* [*n* := *cm'*], *mem'*)) $\in$ *meval*

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

inductive *neval* :: ('*Com*, '*Var*, '*Val*) *LocalConf* $\Rightarrow$ *nat* $\Rightarrow$ (-, -, -) *LocalConf* $\Rightarrow$ *bool*
(**infixl** $\rightsquigarrow^-$ *70*)
where
neval-0: *x* = *y* $\Longrightarrow$ *x* $\rightsquigarrow^0$ *y* |
neval-S-n: *x* $\rightsquigarrow$ *y* $\Longrightarrow$ *y* $\rightsquigarrow^n$ *z* $\Longrightarrow$ *x* $\rightsquigarrow^{Suc\ n}$ *z*

inductive-cases *neval-ZeroE*: *neval x 0 y*
inductive-cases *neval-SucE*: *neval x* (*Suc n*) *y*

lemma *neval-det*:
x $\rightsquigarrow^n$ *y* $\Longrightarrow$ *x* $\rightsquigarrow^n$ *y'* $\Longrightarrow$ *y* = *y'*
apply(*induct arbitrary*: *y'* *rule*: *neval.induct*)
apply(*blast elim*: *neval-ZeroE*)
apply(*blast elim*: *neval-SucE dest*: *deterministic*)
done

lemma *neval-Suc-simp* [*simp*]:
neval x (*Suc 0*) *y* = *x* $\rightsquigarrow$ *y*
proof
assume *a*: *neval x* (*Suc 0*) *y*
have $\bigwedge$*n*. *neval x n y* $\Longrightarrow$ *n* = *Suc 0* $\Longrightarrow$ *x* $\rightsquigarrow$ *y*
proof −
fix *n*
assume *neval x n y*
and *n* = *Suc 0*
thus *x* $\rightsquigarrow$ *y*
by(*induct rule*: *neval.induct*, *auto elim*: *neval-ZeroE*)
qed
with *a* **show** *x* $\rightsquigarrow$ *y* **by** *simp*
next
assume *x* $\rightsquigarrow$ *y*
thus *neval x* (*Suc 0*) *y*
by(*force intro*: *neval.intros*)
qed

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

fun

meval-sched :: *nat list* ⇒ (*'Com*, *'Var*, *'Val*) *GlobalConf* ⇒ (-, -, -) *GlobalConf* ⇒ *bool*
where
meval-sched [] *c c'* = (*c* = *c'*) |
meval-sched (*n*#*ns*) *c c'* = (∃ *c''*. *c* ⇝$_n$ *c''* ∧ *meval-sched ns c'' c'*)

abbreviation
meval-sched-abv :: (-,-,-) *GlobalConf* ⇒ *nat list* ⇒ (-,-,-) *GlobalConf* ⇒ *bool* (- →$_-$ - *70*)
where
c →$_{ns}$ *c'* ≡ *meval-sched ns c c'*

lemma *meval-sched-det*:
meval-sched ns c c' ⟹ *meval-sched ns c c''* ⟹ *c'* = *c''*
apply(*induct ns arbitrary*: *c*)
apply(*auto dest*: *deterministic*)
done

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 mem*$_1$ *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 mem*$_1$ *x* = *Low* ∧ (*x* ∈ *C* ∨ *x* ∉ *mds Asm-NoReadOrWrite*) ⟶ *mem*$_1$ *x* = *mem*$_2$ *x*)

lemma *low-eq-low-mds-eq*:
(*mem*$_1$ =^{l} *mem*$_2$) = (*mem*$_1$ =$_{(\lambda m.\ \{\})}{}^l$ *mem*$_2$)
by(*simp add*: *low-eq-def low-mds-eq-def*)

lemma *low-mds-eq-dma*:
(*mem*$_1$ =$_{mds}{}^l$ *mem*$_2$) ⟹ *dma mem*$_1$ = *dma mem*$_2$
apply(*rule dma-C*)
apply(*simp add*: *low-mds-eq-def C-Low*)
done

lemma *low-mds-eq-sym*:
(*mem*$_1$ =$_{mds}{}^l$ *mem*$_2$) ⟹ (*mem*$_2$ =$_{mds}{}^l$ *mem*$_1$)
apply(*frule low-mds-eq-dma*)
apply(*simp add*: *low-mds-eq-def*)
done

lemma *low-eq-sym*:
(*mem*$_1$ =^{l} *mem*$_2$) ⟹ (*mem*$_2$ =^{l} *mem*$_1$)
apply(*simp add*: *low-eq-low-mds-eq low-mds-eq-sym*)
done

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*)

lemma *High-not-in-*$\mathcal{C}$ [*simp*]:
dma mem$_1$ *x* = *High* ⟹ *x* ∉ $\mathcal{C}$
apply(*case-tac x* ∈ $\mathcal{C}$)
by(*simp add*: $\mathcal{C}$*-Low*)

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*$_2$ ⟩) ∈ $\mathcal{R}$ ⟶
(∀ *A*. ((∀ *x*. *case A x of Some* (*v*,*v*′) ⇒ (*mem*$_1$ *x* ≠ *v* ∨ *mem*$_2$ *x* ≠ *v*′) ⟶ ¬ *var-asm-not-written mds x* | - ⇒ *True*) ∧
(∀ *x*. *dma* (*mem*$_1$ [‖$_1$ *A*]) *x* ≠ *dma mem*$_1$ *x* ⟶ ¬ *var-asm-not-written mds x*) ∧
(∀ *x*. *dma* (*mem*$_1$ [‖$_1$ *A*]) *x* = *Low* ∧ (*x* ∉ *mds AsmNoReadOrWrite* ∨ *x* ∈ $\mathcal{C}$) ⟶ (*mem*$_1$ [‖$_1$ *A*]) *x* = (*mem*$_2$ [‖$_2$ *A*]) *x*)) ⟶
(⟨ *c*$_1$, *mds*, *mem*$_1$[‖$_1$ *A*] ⟩, ⟨ *c*$_2$, *mds*, *mem*$_2$[‖$_2$ *A*] ⟩) ∈ $\mathcal{R}$))

definition
strong-low-bisim-mm :: (('*Com*, '*Var*, '*Val*) *LocalConf*) *rel* ⇒ *bool*
where
strong-low-bisim-mm $\mathcal{R}$ ≡
sym $\mathcal{R}$ ∧
closed-glob-consistent $\mathcal{R}$ ∧
(∀ *c*$_1$ *mds mem*$_1$ *c*$_2$ *mem*$_2$. (⟨ *c*$_1$, *mds*, *mem*$_1$ ⟩, ⟨ *c*$_2$, *mds*, *mem*$_2$ ⟩) ∈ $\mathcal{R}$ ⟶
(*mem*$_1$ =$_{mds}{}^l$ *mem*$_2$) ∧
(∀ *c*$_1$′ *mds*′ *mem*$_1$′. ⟨ *c*$_1$, *mds*, *mem*$_1$ ⟩ ⇝ ⟨ *c*$_1$′, *mds*′, *mem*$_1$′ ⟩ ⟶
(∃ *c*$_2$′ *mem*$_2$′. ⟨ *c*$_2$, *mds*, *mem*$_2$ ⟩ ⇝ ⟨ *c*$_2$′, *mds*′, *mem*$_2$′ ⟩ ∧
(⟨ *c*$_1$′, *mds*′, *mem*$_1$′ ⟩, ⟨ *c*$_2$′, *mds*′, *mem*$_2$′ ⟩) ∈ $\mathcal{R}$)))

inductive-set *mm-equiv* :: (('*Com*, '*Var*, '*Val*) *LocalConf*) *rel*
and *mm-equiv-abv* :: ('*Com*, '*Var*, '*Val*) *LocalConf* ⇒
('*Com*, '*Var*, '*Val*) *LocalConf* ⇒ *bool* (**infix** ≈ *60*)
where
mm-equiv-abv x y ≡ (*x*, *y*) ∈ *mm-equiv* |

mm-equiv-intro [*iff*]: ⟦ *strong-low-bisim-mm* $\mathcal{R}$; $(lc_1, lc_2) \in \mathcal{R}$ ⟧ ⟹ $(lc_1, lc_2) \in$ *mm-equiv*

inductive-cases *mm-equiv-elim* [*elim*]: ⟨ c_1, *mds*, mem_1 ⟩ ≈ ⟨ c_2, *mds*, mem_2 ⟩

definition *low-indistinguishable* :: *'Var Mds* ⇒ *'Com* ⇒ *'Com* ⇒ *bool*
(- ∼₁ - [*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* × *'Var Mds* ⇒ *bool*
where
com-sifum-secure cmd ≡ *case cmd of* (c, mds_s) ⇒ $c \sim_{mds_s} c$

definition
prog-sifum-secure-cont :: (*'Com* × *'Var Mds*) *list* ⇒ *bool*
where *prog-sifum-secure-cont cmds* =
(∀ $mem_1\ mem_2$. *INIT* mem_1 ∧ *INIT* mem_2 ∧ $mem_1 =^{l} mem_2$ ⟶
(∀ *sched* $cms_1'\ mem_1'$.
(*cmds*, mem_1) $\rightarrow_{sched}$ (cms_1', mem_1') ⟶
(∃ $cms_2'\ mem_2'$. (*cmds*, mem_2) $\rightarrow_{sched}$ (cms_2', mem_2') ∧
map snd $cms_1' =$ *map snd* cms_2' ∧
length $cms_2' =$ *length* cms_1' ∧
(∀ *x*. *dma* $mem_1'\ x = Low \wedge (x \in \mathcal{C} \vee (\forall\ i < length\ cms_1'.$
$x \notin snd\ (cms_1'\ !\ i)\ AsmNoReadOrWrite)) \longrightarrow mem_1'\ x = mem_2'\ x$))))

lemma *prog-sifum-secure-cont-def2*:
prog-sifum-secure-cont cmds ≡
(∀ $mem_1\ mem_2$. *INIT* mem_1 ∧ *INIT* mem_2 ∧ $mem_1 =^{l} mem_2$ ⟶
(∀ *sched* $cms_1'\ mem_1'$.
(*cmds*, mem_1) $\rightarrow_{sched}$ (cms_1', mem_1') ⟶
(∃ $cms_2'\ mem_2'$. (*cmds*, mem_2) $\rightarrow_{sched}$ (cms_2', mem_2')) ∧
(∀ $cms_2'\ mem_2'$. (*cmds*, mem_2) $\rightarrow_{sched}$ (cms_2', mem_2') ⟶
map snd $cms_1' =$ *map snd* cms_2' ∧
length $cms_2' =$ *length* cms_1' ∧
(∀ *x*. *dma* $mem_1'\ x = Low \wedge (x \in \mathcal{C} \vee (\forall\ i < length\ cms_1'.$
$x \notin snd\ (cms_1'\ !\ i)\ AsmNoReadOrWrite)) \longrightarrow mem_1'\ x = mem_2'\ x$))))
apply(*rule eq-reflection*)
unfolding *prog-sifum-secure-cont-def*
apply(*rule iffI*)
apply(*blast dest*: *meval-sched-det*)

by *fastforce*

2.4 Sound Mode Use

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

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

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

definition
doesnt-read-or-modify-vars :: *'Com* $\Rightarrow$ *'Var set* $\Rightarrow$ *bool*
where
doesnt-read-or-modify-vars c X = $(\forall$ *mds mem c' mds' mem'*.
$\langle$ *c*, *mds*, *mem* $\rangle \rightsquigarrow \langle$ *c'*, *mds'*, *mem'* $\rangle \longrightarrow$
$((\forall x{\in}X.\ (\forall\ v.\ \langle$ *c*, *mds*, *mem* $(x := v)\ \rangle \rightsquigarrow \langle$ *c'*, *mds'*, *mem'* $(x := v)\ \rangle)))))$

definition
vars-$\mathcal{C}$ :: *'Var set* $\Rightarrow$ *'Var set*
where
vars-$\mathcal{C}$ *X* $\equiv \bigcup x{\in}X.$ $\mathcal{C}$*-vars x*

lemma *vars-*$\mathcal{C}$*-subset-*$\mathcal{C}$:
vars-$\mathcal{C}$ *X* $\subseteq \mathcal{C}$
by(*auto simp*: $\mathcal{C}$*-def vars-*$\mathcal{C}$*-def*)

definition
doesnt-read-or-modify :: *'Com* $\Rightarrow$ *'Var* $\Rightarrow$ *bool*
where
doesnt-read-or-modify c x $\equiv$ *doesnt-read-or-modify-vars c* $(\{x\} \cup \mathcal{C}$*-vars x*)

definition
doesnt-modify :: *'Com* $\Rightarrow$ *'Var* $\Rightarrow$ *bool*
where
doesnt-modify c x = $(\forall$ *mds mem c' mds' mem'*. $(\langle$ *c*, *mds*, *mem* $\rangle \rightsquigarrow \langle$ *c'*, *mds'*, *mem'* $\rangle) \longrightarrow$
mem x = *mem' x* $\wedge$ *dma mem x* = *dma mem' x*)

lemma *noread-nowrite*:
assumes *step*: $\langle c, mds, mem\rangle \rightsquigarrow \langle c', mds', mem'\rangle$
assumes *noread*: $(\bigwedge v.\ \langle c, mds, mem(x := v)\rangle \rightsquigarrow \langle c', mds', mem'(x := v)\rangle)$
shows $mem\ x = mem'\ x$
proof −
from *noread* **have** $\langle c, mds, mem(x := (mem\ x))\rangle \rightsquigarrow \langle c', mds', mem'(x := (mem\ x))\rangle$
by *blast*
hence $\langle c, mds, mem\rangle \rightsquigarrow \langle c', mds', mem'(x := (mem\ x))\rangle$ **by** *simp*
from *step this* **have** $mem' = mem'(x := (mem\ x))$ **by** (*blast dest*: *deterministic*)
hence $mem'\ x = (mem'(x := (mem\ x)))\ x$ **by**(*rule arg-cong*)
thus *?thesis* **by** *simp*
qed

lemma *doesnt-read-or-modify-doesnt-modify*:
$doesnt\text{-}read\text{-}or\text{-}modify\ c\ x \Longrightarrow doesnt\text{-}modify\ c\ x$
by(*fastforce simp*: *doesnt-modify-def doesnt-read-or-modify-def doesnt-read-or-modify-vars-def*

intro: *noread-nowrite dma-*$\mathcal{C}$*-vars*)

inductive-set
loc-reach :: (*'Com*, *'Var*, *'Val*) *LocalConf* ⇒ (*'Com*, *'Var*, *'Val*) *LocalConf set*
for *lc* :: (-, -, -) *LocalConf*
where
refl : $\langle fst\ (fst\ lc), snd\ (fst\ lc), snd\ lc\rangle \in loc\text{-}reach\ lc$ |
step : $[\![\ \langle c', mds', mem'\rangle \in loc\text{-}reach\ lc;$
$\langle c', mds', mem'\rangle \rightsquigarrow \langle c'', mds'', mem''\rangle\ ]\!] \Longrightarrow$
$\langle c'', mds'', mem''\rangle \in loc\text{-}reach\ lc$ |
mem-diff : $[\![\ \langle\ c', mds', mem'\ \rangle \in loc\text{-}reach\ lc;$
$(\forall\ x.\ var\text{-}asm\text{-}not\text{-}written\ mds'\ x \longrightarrow mem'\ x = mem''\ x \wedge dma\ mem'\ x = dma\ mem''\ x)\ ]\!] \Longrightarrow$
$\langle\ c', mds', mem''\ \rangle \in loc\text{-}reach\ lc$

lemma *neval-loc-reach*:
$neval\ lc'\ n\ lc'' \Longrightarrow lc' \in loc\text{-}reach\ lc \Longrightarrow lc'' \in loc\text{-}reach\ lc$
proof(*induct rule*: *neval.induct*)
case (*neval-0 x y*)
thus *?case* **by** *simp*
next
case (*neval-S-n x y n z*)
from ‹$x \in loc\text{-}reach\ lc$› **and** ‹$x \rightsquigarrow y$› **have** $y \in loc\text{-}reach\ lc$

apply(*case-tac x*, *rename-tac a b*, *case-tac a*, *clarsimp*)
apply(*case-tac y*, *rename-tac c d*, *case-tac c*, *clarsimp*)
by(*blast intro*: *loc-reach.step*)
thus *?case*
using *neval-S-n*(*3*) **by** *blast*

qed

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′ GuarNoReadOrWrite* ⟶ *doesnt-read-or-modify c′ x*) ∧
(*x* ∈ *mds′ GuarNoWrite* ⟶ *doesnt-modify c′ x*)))

definition
respects-own-guarantees :: (*′Com* × *′Var Mds*) ⇒ *bool*
where
respects-own-guarantees cm ≡
(∀ *x*. (*x* ∈ (*snd cm*) *GuarNoReadOrWrite* ⟶ *doesnt-read-or-modify* (*fst cm*) *x*)
∧
(*x* ∈ (*snd cm*) *GuarNoWrite* ⟶ *doesnt-modify* (*fst cm*) *x*))

lemma *locally-sound-mode-use-def2*:
locally-sound-mode-use lc ≡ ∀ *lc′* ∈ *loc-reach lc*. *respects-own-guarantees* (*fst lc′*)
apply(*rule eq-reflection*)
apply(*simp add*: *locally-sound-mode-use-def respects-own-guarantees-def*)
apply *force*
done

lemma *locally-sound-respects-guarantees*:
locally-sound-mode-use (*cm*, *mem*) ⟹ *respects-own-guarantees cm*
unfolding *locally-sound-mode-use-def respects-own-guarantees-def*
by (*metis fst-conv loc-reach.refl*)

definition
compatible-modes :: (*′Var Mds*) *list* ⇒ *bool*
where
compatible-modes mdss = (∀ (*i* :: *nat*) *x*. *i* < *length mdss* ⟶
(*x* ∈ (*mdss* ! *i*) *AsmNoReadOrWrite* ⟶
(∀ *j* < *length mdss*. *j* ≠ *i* ⟶ *x* ∈ (*mdss* ! *j*) *GuarNoReadOrWrite*)) ∧
(*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′ sched*. *gc* →$_{sched}$ (*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*: $lc \sim_{mds} lc' \Longrightarrow lc' \sim_{mds} lc$
apply(*clarsimp simp*: *low-indistinguishable-def*)
apply(*rule mm-equiv-sym*)
apply(*blast dest*: *low-mds-eq-sym*)
done

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

end

end

3 Compositionality Proof for SIFUM-Security Property

theory *Compositionality*
imports *Security*
begin

context *sifum-security-init*
begin

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

definition
differing-vars-lists :: (*'Var*, *'Val*) *Mem* ⇒ (-, -) *Mem* ⇒
((-, -) *Mem* × (-, -) *Mem*) *list* ⇒ *nat* ⇒ *'Var set*
where
differing-vars-lists mem_1 mem_2 *mems* $i \equiv$
(*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*)

fun *makes-compatible* ::
('*Com*, '*Var*, '*Val*) *GlobalConf* ⇒
('*Com*, '*Var*, '*Val*) *GlobalConf* ⇒
((-, -) *Mem* × (-, -) *Mem*) *list* ⇒
bool
where
makes-compatible (cms_1, mem_1) (cms_2, mem_2) *mems* =
(*length* cms_1 = *length* cms_2 ∧ *length* cms_1 = *length mems* ∧
(∀ *i*. *i* < *length* cms_1 ⟶
(∀ σ. *dom* σ = *differing-vars-lists* mem_1 mem_2 *mems i* ⟶
(cms_1 ! *i*, (*fst* (*mems* ! *i*)) [↦ σ]) ≈ (cms_2 ! *i*, (*snd* (*mems* ! *i*)) [↦ σ])) ∧
(∀ *x*. (mem_1 *x* = mem_2 *x* ∨ *dma* mem_1 *x* = *High* ∨ *x* ∈ 𝒞) ⟶
x ∉ *differing-vars-lists* mem_1 mem_2 *mems i*)) ∧
((*length* cms_1 = 0 ∧ mem_1 =^{l} mem_2) ∨ (∀ *x*. ∃ *i*. *i* < *length* cms_1 ∧
x ∉ *differing-vars-lists* mem_1 mem_2 *mems i*)))

lemma *makes-compatible-intro* [*intro*]:
⟦ *length* cms_1 = *length* cms_2 ∧ *length* cms_1 = *length mems*;
(⋀ *i* σ. ⟦ *i* < *length* cms_1; *dom* σ = *differing-vars-lists* mem_1 mem_2 *mems i* ⟧
⟹
(cms_1 ! *i*, (*fst* (*mems* ! *i*)) [↦ σ]) ≈ (cms_2 ! *i*, (*snd* (*mems* ! *i*)) [↦ σ]));
(⋀ *i x*. ⟦ *i* < *length* cms_1; mem_1 *x* = mem_2 *x* ∨ *dma* mem_1 *x* = *High* ∨ *x* ∈
𝒞 ⟧ ⟹
x ∉ *differing-vars-lists* mem_1 mem_2 *mems i*);
(*length* cms_1 = 0 ∧ mem_1 =^{l} mem_2) ∨
(∀ *x*. ∃ *i*. *i* < *length* cms_1 ∧ *x* ∉ *differing-vars-lists* mem_1 mem_2 *mems i*) ⟧
⟹
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* mem_1 *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 mem*$_1$ *x* = *High* ∨ *x* ∈ 𝒞) ⟶ *x* ∉ *differing-vars-lists*
mem$_1$ *mem*$_2$ *mems i*

by (*simp add*: *Let-def*, *blast*)
with ∗ **show** *dma mem*$_1$ *x* = *Low*
by (*cases dma mem*$_1$ *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 mem*$_1$ *x* = *Low* ∧ *x* ∉ $\mathcal{C}$
by (*cases dma mem*$_1$ *x*, *auto*)

lemma *sound-modes-no-read* :
⟦ *sound-mode-use* (*cms*, *mem*); *x* ∈ (*map snd cms* ! *i*) *GuarNoReadOrWrite*; *i* < *length cms* ⟧ ⟹
doesnt-read-or-modify (*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*) *GuarNoReadOrWrite*
ultimately show *doesnt-read-or-modify* (*fst* (*cms* !*i*)) *x*
apply (*simp add*: *locally-sound-mode-use-def*)
using ‹*i* < *length cms*› ‹*locally-sound-mode-use* (*cms* ! *i*, *mem*)› *locally-sound-respects-guarantees respects-own-guarantees-def* **by** *auto*
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 *makes-compatible-dma-eq*:
assumes *compat*: *makes-compatible* (*cms*$_1$, *mem*$_1$) (*cms*$_2$, *mem*$_2$) *mems*
assumes *ile*: *i* < *length cms*$_1$
assumes *domσ*: *dom* σ = *differing-vars-lists mem*$_1$ *mem*$_2$ *mems i*
shows *dma* ((*fst* (*mems* ! *i*)) [↦ σ]) = *dma mem*$_1$
proof(*rule dma-*$\mathcal{C}$, *clarify*)

fix x
assume $x \in \mathcal{C}$
with *compat ile* **have** *notin-diff*: $x \notin$ *differing-vars-lists* mem_1 mem_2 *mems* i
by *simp*
hence $x \notin$ *dom* σ
by(*metis domσ*)
hence (*fst* (*mems* ! i) [$\mapsto \sigma$]) x = (*fst* (*mems* ! i)) x
by(*metis subst-not-in-dom*)
moreover have (*fst* (*mems* ! i)) $x = mem_1\ x$
using *notin-diff differing-vars-neg* **by** *metis*
ultimately show (*fst* (*mems* ! i) [$\mapsto \sigma$]) $x = mem_1\ x$ **by** *simp*
qed

lemma *compat-different-vars*:
⟦ *fst* (*mems* ! i) x = *snd* (*mems* ! i) x;
$x \notin$ *differing-vars-lists* mem_1 mem_2 *mems* i ⟧ ⟹
$mem_1\ x = mem_2\ x$
proof −
assume $x \notin$ *differing-vars-lists* mem_1 mem_2 *mems* i
hence *fst* (*mems* ! i) $x = mem_1\ x \wedge$ *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*:
⟦ ⟨ c_1, *mds*, mem_1 ⟩ ≈ ⟨ c_2, *mds*, mem_2 ⟩ ⟧ ⟹ $mem_1 =_{mds}{}^{l}\ mem_2$
unfolding *mm-equiv.simps strong-low-bisim-mm-def*
by *fast*

lemma *globally-sound-modes-compatible*:
⟦ *globally-sound-mode-use* (*cms*, *mem*) ⟧ ⟹ *compatible-modes* (*map snd cms*)
apply (*simp add*: *globally-sound-mode-use-def reachable-mode-states-def*)
using *meval-sched.simps*(*1*) **by** *blast*

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* ∈ *differing-vars-lists mem*$_1$ *mem*$_2$ *mems i*
shows *doesnt-read-or-modify* (*fst* (*cms*$_1$! *i*)) *x* ∧ *doesnt-read-or-modify* (*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 mem*$_1$ *x* = *Low*
by (*metis compat-low*)

from *compat ile x* **have** *b*: *mem*$_1$ *x* ≠ *mem*$_2$ *x*
by (*metis compat-different*)

from *compat ile x* **have** *not-in-C*: *x* ∉ *C*
by (*metis compat-different*)

with *a* **and** *compat ile x* **obtain** *j* **where**
jprop: *j* < *length cms*$_1$ ∧ *x* ∉ *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* ⇀ *'Val* **where** *domσ*: *dom σ* = *?X*$_j$
proof
let *?σ* = λ *x*. *if* (*x* ∈ *?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$_1$*j* = *fst* (*mems* ! *j*) **and**
?mems$_2$*j* = *snd* (*mems* ! *j*)

from *jprop domσ* **have** *subst-eq*:
?mems$_1$*j* [↦ *σ*] *x* = *?mems*$_1$*j x* ∧ *?mems*$_2$*j* [↦ *σ*] *x* = *?mems*$_2$*j x*
by (*metis subst-not-in-dom*)

from *compat jprop domσ*
have (*cms*$_1$! *j*, *?mems*$_1$*j* [↦ *σ*]) ≈ (*cms*$_2$! *j*, *?mems*$_2$*j* [↦ *σ*])
by (*auto simp*: *Let-def*)

hence *low-eq*: *?mems*$_1$*j* [↦ *σ*] =$_{?mdss\ !\ j}^{l}$ *?mems*$_2$*j* [↦ *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*) *AsmNoReadOrWrite*
proof −
{ **assume** *x* ∉ (*?mdss* ! *j*) *AsmNoReadOrWrite*

```
      then have mems-eq: ?mems₁j x = ?mems₂j x
        using ‹dma mem₁ x = Low› low-eq subst-eq
        makes-compatible-dma-eq[OF compat jprop[THEN conjunct1] domσ]
        low-mds-eq-def
        by (metis (poly-guards-query))

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

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

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

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

definition
  vars-and-𝒞 :: 'Var set ⇒ 'Var set
where
  vars-and-𝒞 X ≡ X ∪ vars-𝒞 X


fun change-respecting ::
  ('Com, 'Var, 'Val) LocalConf ⇒
   ('Com, 'Var, 'Val) LocalConf ⇒
   'Var set ⇒ bool
  where change-respecting (cms, mem) (cms', mem') X =
      ((cms, mem) ⇝ (cms', mem') ∧
        (∀ σ. dom σ = vars-and-𝒞 X ⟶ (cms, mem [↦ σ]) ⇝ (cms', mem' [↦
σ])))

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))

definition to-partial :: ('a ⇒ 'b) ⇒ ('a ⇀ 'b)
  where to-partial f = (λ x. Some (f x))

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

lemma *change-respecting-doesnt-modify'*:
assumes *eval*: $(cms, mem) \rightsquigarrow (cms', mem')$
assumes *cr*: $\forall\ f.\ dom\ f = Y \longrightarrow (cms, mem\ [\mapsto f]) \rightsquigarrow (cms', mem'\ [\mapsto f])$
assumes *x-in-dom*: $x \in Y$
shows $mem\ x = mem'\ x$
proof −
let $?f' = to\text{-}partial\ mem\ |`\ Y$
have *domf'*: $dom\ ?f' = Y$
by (*metis dom-restrict-total*)

from *this cr* **have** *eval'*: $(cms, mem\ [\mapsto ?f']) \rightsquigarrow (cms', mem'\ [\mapsto ?f'])$
by (*metis*)

have *mem-eq*: $mem\ [\mapsto ?f'] = mem$
proof
fix x
show $mem\ [\mapsto ?f']\ x = mem\ x$
unfolding *subst-def*
apply (*cases* $x \in Y$)
apply (*metis option.simps*(5) *restrict-in to-partial-def*)
by (*metis domf' subst-def subst-not-in-dom*)
qed

then have *mem'-eq*: $mem'\ [\mapsto ?f'] = mem'$
using *eval eval' deterministic*
by (*metis Pair-inject*)

moreover
have *x-in-dom'*: $x \in dom\ ?f'$
by (*metis x-in-dom dom-restrict-total*)
hence $?f'\ x = Some\ (mem\ x)$
by (*metis restrict-in to-partial-def x-in-dom*)
hence $mem'\ [\mapsto ?f']\ x = mem\ x$
using *subst-def x-in-dom'*
by (*metis option.simps*(5))
thus $mem\ x = mem'\ x$
by (*metis mem'-eq*)
qed

lemma *change-respecting-subset'*:
assumes *step*: $(cms, mem) \rightsquigarrow (cms', mem')$
assumes *noread*: $(\forall\ \sigma.\ dom\ \sigma = X \longrightarrow (cms, mem\ [\mapsto \sigma]) \rightsquigarrow (cms', mem'\ [\mapsto \sigma]))$
assumes *dom-subset*: $dom\ \sigma \subseteq X$
shows $(cms, mem\ [\mapsto \sigma]) \rightsquigarrow (cms', mem'\ [\mapsto \sigma])$
proof −
define σ_X **where** $\sigma_X\ x = (if\ x \in X\ then\ if\ x \in dom\ \sigma\ then\ \sigma\ x\ else\ Some\ (mem\ x)\ else\ None)$ **for** x

have *dom* $\sigma_X = X$ **using** *dom-subset* **by**(*auto simp*: σ_X*-def*)

have *mem* $[\mapsto \sigma] = mem\ [\mapsto \sigma_X]$
apply(*rule ext*)
using *dom-subset* **apply**(*auto simp*: *subst-def* σ_X*-def split*: *option.splits*)
done

moreover have *mem'* $[\mapsto \sigma] = mem'\ [\mapsto \sigma_X]$
apply(*rule ext*)
using *dom-subset* **apply**(*auto simp*: *subst-def* σ_X*-def split*: *option.splits simp*: *change-respecting-doesnt-modify'*[*OF step noread*])
done

moreover from *noread* ‹*dom* $\sigma_X = X$› **have** (*cms*, *mem* $[\mapsto \sigma_X]$) $\rightsquigarrow$ (*cms'*, *mem'* $[\mapsto \sigma_X]$) **by** *metis*
ultimately show *?thesis* **by** *simp*
qed

lemma *change-respecting-subst*:
change-respecting (*cms*, *mem*) (*cms'*, *mem'*) *X* $\Longrightarrow$
$(\forall\ \sigma.\ dom\ \sigma = X \longrightarrow (cms, mem\ [\mapsto \sigma]) \rightsquigarrow (cms', mem'\ [\mapsto \sigma]))$
unfolding *change-respecting.simps vars-and-*$\mathcal{C}$*-def*
using *change-respecting-subset'* **by** *blast*

lemma *change-respecting-intro* [*iff*]:
$[\![\langle\ c, mds, mem\ \rangle \rightsquigarrow \langle\ c', mds', mem'\ \rangle;$
$\bigwedge f.\ dom\ f = vars\text{-}and\text{-}\mathcal{C}\ X \Longrightarrow$
$(\langle\ c, mds, mem\ [\mapsto f]\ \rangle \rightsquigarrow \langle\ c', mds', mem'\ [\mapsto f]\ \rangle)\ ]\!]$
$\Longrightarrow$ *change-respecting* $\langle c, mds, mem\rangle$ $\langle c', mds', mem'\rangle$ *X*
unfolding *change-respecting.simps*
by *blast*

lemma *vars-*$\mathcal{C}$*-mono*:
$X \subseteq Y \Longrightarrow vars\text{-}\mathcal{C}\ X \subseteq vars\text{-}\mathcal{C}\ Y$
by(*auto simp*: *vars-*$\mathcal{C}$*-def*)

lemma *vars-*$\mathcal{C}$*-Un*:
$vars\text{-}\mathcal{C}\ (X \cup Y) = (vars\text{-}\mathcal{C}\ X \cup vars\text{-}\mathcal{C}\ Y)$
by(*simp add*: *vars-*$\mathcal{C}$*-def*)

lemma *vars-*$\mathcal{C}$*-insert*:
$vars\text{-}\mathcal{C}\ (insert\ x\ Y) = (vars\text{-}\mathcal{C}\ \{x\}) \cup (vars\text{-}\mathcal{C}\ Y)$
apply(*subst insert-is-Un*)
apply(*rule vars-*$\mathcal{C}$*-Un*)
done

lemma *vars-*$\mathcal{C}$*-empty*[*simp*]:
$vars\text{-}\mathcal{C}\ \{\} = \{\}$
by(*simp add*: *vars-*$\mathcal{C}$*-def*)

lemma *$\mathcal{C}$-vars-of-$\mathcal{C}$-vars-empty*:
$x \in \mathcal{C}$-vars $y \Longrightarrow \mathcal{C}$-vars $x = \{\}$
apply(*drule subsetD*[*OF $\mathcal{C}$-vars-subset-$\mathcal{C}$*])
apply(*erule $\mathcal{C}$-vars-$\mathcal{C}$*)
done

lemma *vars-and-$\mathcal{C}$-mono*:
$X \subseteq X' \Longrightarrow$ vars-and-$\mathcal{C}$ $X \subseteq$ vars-and-$\mathcal{C}$ X'
apply(*unfold vars-and-$\mathcal{C}$-def*)
apply(*metis Un-mono vars-$\mathcal{C}$-mono*)
done

lemma *$\mathcal{C}$-vars-finite*[*simp*]:
finite ($\mathcal{C}$-vars x)
apply(*rule finite-subset*[*OF - finite-memory*])
by *blast*

lemma *finite-dom*:
finite (dom (σ::'Var $\Rightarrow$ 'Val option))
by(*blast intro*: *finite-subset*[*OF - finite-memory*])

lemma *doesnt-read-or-modify-subst*:
assumes *noread*: *doesnt-read-or-modify c x*
assumes *step*: *$\langle c, mds, mem\rangle \rightsquigarrow \langle c', mds', mem'\rangle$*
assumes *subset*: *$X \subseteq \{x\} \cup \mathcal{C}$-vars x*
shows *$\bigwedge \sigma$. dom σ = X $\Longrightarrow \langle c, mds, mem[\mapsto \sigma]\rangle \rightsquigarrow \langle c', mds', mem'[\mapsto \sigma]\rangle$*
proof −
have *finite X*
using *subset* **apply**(*rule finite-subset*)
by *simp*
show *$\bigwedge \sigma$. dom σ = X $\Longrightarrow \langle c, mds, mem[\mapsto \sigma]\rangle \rightsquigarrow \langle c', mds', mem'[\mapsto \sigma]\rangle$*
using *‹finite X› subset*
proof(*induct X rule*: *finite-subset-induct*[**where** *A*={*x*} ∪ *$\mathcal{C}$-vars x*])
case *empty*
thus *$\langle c, mds, subst\ \sigma\ mem\rangle \rightsquigarrow \langle c', mds', subst\ \sigma\ mem'\rangle$*
using *step* **by**(*simp add*: *subst-def*)
next
case (*insert a X*)
show *$\langle c, mds, subst\ \sigma\ mem\rangle \rightsquigarrow \langle c', mds', subst\ \sigma\ mem'\rangle$*
proof −
let *?σ_X = (σ |‘ X)*
have *IH_X*: *$\langle c, mds, subst\ ?\sigma_X\ mem\rangle \rightsquigarrow \langle c', mds', subst\ ?\sigma_X\ mem'\rangle$*
apply(*rule insert*(*4*))
using *insert* **by** (*metis dom-restrict inf.absorb2 subset-insertI*)
from *insert* **obtain** *v* **where** *σa*: *σ a = Some v* **by** *auto*
have *r*: *$\bigwedge$mem. (subst ?σ_X mem)(a := v) = subst σ mem*
apply(*rule ext*, *rename-tac y*)
apply(*simp*, *safe*)

apply(*simp add*: *subst-def* σa)
using ‹$a \notin X$› *insert* **apply**(*auto simp*: *subst-def split*: *option.splits simp*: *restrict-map-def*)
done
have $\langle c, mds, (subst\ ?\sigma_X\ mem)(a := v)\rangle \rightsquigarrow \langle c', mds', (subst\ ?\sigma_X\ mem')(a := v)\rangle$
using *noread* ‹$a \in \{x\} \cup \mathcal{C}$-*vars x*› IH_X
unfolding *doesnt-read-or-modify-def doesnt-read-or-modify-vars-def* **by** *metis*
thus *?thesis* **by**(*simp add*: *r*)
qed
qed
qed

lemma *subst-restrict-twice*:
$dom\ \sigma = A \cup B \Longrightarrow$
$mem\ [\mapsto (\sigma\ |`\ A)]\ [\mapsto (\sigma\ |`\ B)] = mem\ [\mapsto \sigma]$
by(*fastforce simp*: *subst-def split*: *option.splits intro*!: *ext simp*: *restrict-map-def*)

lemma *noread-exists-change-respecting*:
assumes *fin*: *finite* (X :: *'Var set*)
assumes *eval*: $\langle\ c, mds, mem\ \rangle \rightsquigarrow \langle\ c', mds', mem'\ \rangle$
assumes *noread*: $\forall\ x \in X.$ *doesnt-read-or-modify c x*
shows *change-respecting* $\langle c, mds, mem\rangle\ \langle c', mds', mem'\rangle\ X$
proof −
let *?lc* = $\langle c, mds, mem\rangle$ **and** *?lc'* = $\langle c', mds', mem'\rangle$
from *fin eval noread* **show** *change-respecting* $\langle c, mds, mem\rangle\ \langle c', mds', mem'\rangle\ X$
proof (*induct X arbitrary*: *mem mem' rule*: *finite-induct*)
case *empty*
have $mem\ [\mapsto Map.empty] = mem\ mem'\ [\mapsto Map.empty] = mem'$
unfolding *subst-def*
by *auto*
hence *change-respecting* $\langle c, mds, mem\rangle\ \langle c', mds', mem'\rangle\ \{\}$
using *empty*
unfolding *change-respecting.simps subst-def vars-*$\mathcal{C}$*-def vars-and-*$\mathcal{C}$*-def*
by *auto*
thus *?case* **by** *blast*
next
case (*insert x X*)
then have *IH*: *change-respecting* $\langle c, mds, mem\rangle\ \langle c', mds', mem'\rangle\ X$
by (*metis* (*poly-guards-query*) *insertCI insert-disjoint*(*1*))
show *change-respecting* $\langle c, mds, mem\rangle\ \langle c', mds', mem'\rangle$ (*insert x X*)
proof
show $\langle c, mds, mem\rangle \rightsquigarrow \langle c', mds', mem'\rangle$ **using** *insert* **by** *auto*
next
fix σ :: *'Var* ⇀ *'Val*
let $?\sigma_X = \sigma\ |`\ vars\text{-}and\text{-}\mathcal{C}\ X$
let $?\sigma_x = \sigma\ |`\ (\{x\} \cup \mathcal{C}\text{-}vars\ x)$
assume *domσ*: *dom* σ = *vars-and-*$\mathcal{C}$ (*insert x X*)

hence *dom* $?\sigma_X$ = *vars-and-*$\mathcal{C}$ *X*
by (*metis dom-restrict inf-absorb2 subset-insertI vars-and-*$\mathcal{C}$*-mono*)
from *dom*σ **have** *dom*σ_x: *dom* $?\sigma_x = \{x\} \cup \mathcal{C}$*-vars x*
by(*simp add*: *dom*σ *vars-and-*$\mathcal{C}$*-def vars-*$\mathcal{C}$*-def*, *blast*)
have *dom* σ = *vars-and-*$\mathcal{C}$ *X* $\cup$ ($\{x\} \cup \mathcal{C}$*-vars x*)
by(*simp add*: *dom*σ *vars-and-*$\mathcal{C}$*-def vars-*$\mathcal{C}$*-def*, *blast*)
hence *subst*σ: $\bigwedge$ *mem*. *mem* [$\mapsto$ $?\sigma_X$] [$\mapsto$ $?\sigma_x$] = *mem* [$\mapsto$ σ]
by(*rule subst-restrict-twice*)
from *insert* **have** *doesnt-read-or-modify c x* **by** *auto*
moreover from *IH* **have** $eval_X$: $\langle c, mds, mem\ [\mapsto\ ?\sigma_X]\rangle \rightsquigarrow \langle c', mds', mem'\ [\mapsto\ ?\sigma_X]\rangle$
using ‹*dom* $?\sigma_X$ = *vars-and-*$\mathcal{C}$ *X*›
unfolding *change-respecting.simps*
by *auto*
ultimately have $\langle c, mds, mem\ [\mapsto\ ?\sigma_X]\ [\mapsto\ ?\sigma_x]\rangle \rightsquigarrow \langle c', mds', mem'\ [\mapsto\ ?\sigma_X]\ [\mapsto\ ?\sigma_x]\rangle$
using *subset-refl dom*σ_x *doesnt-read-or-modify-subst* **by** *metis*
thus $\langle c, mds, mem\ [\mapsto \sigma]\rangle \rightsquigarrow \langle c', mds', mem'\ [\mapsto \sigma]\rangle$
using *subst*σ **by** *metis*
qed
qed
qed

lemma *update-nth-eq*:
$[\![\ xs = ys;\ n < length\ xs\ ]\!] \Longrightarrow xs = ys\ [n := xs\ !\ n]$
by (*metis list-update-id*)

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

lemma *mm-equiv-step*:
assumes *bisim*: $(cms_1, mem_1) \approx (cms_2, mem_2)$
assumes *modes-eq*: *snd* cms_1 = *snd* cms_2
assumes *step*: $(cms_1, mem_1) \rightsquigarrow (cms_1', mem_1')$
shows $\exists\ c_2'\ mem_2'.\ (cms_2, mem_2) \rightsquigarrow \langle\ c_2', snd\ cms_1', mem_2'\ \rangle \wedge$
$(cms_1', mem_1') \approx \langle\ c_2', snd\ cms_1', mem_2'\ \rangle$
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*
assumes *eval*: (*cms*, *mem*) $\rightsquigarrow$ (*cms'*, *mem'*)
assumes *x-in-dom*: *x* $\in$ *X* $\cup$ *vars-*$\mathcal{C}$ *X*
shows *mem x* = *mem' x*
using *change-respecting-doesnt-modify'*[**where** *Y*=*X* $\cup$ *vars-*$\mathcal{C}$ *X*, *OF eval*] *cr*

change-respecting.simps vars-and-$\mathcal{C}$-def x-in-dom
by *metis*

lemma *change-respecting-doesnt-modify-dma*:
assumes *cr*: *change-respecting* (*cms*, *mem*) (*cms'*, *mem'*) *X*
assumes *eval*: (*cms*, *mem*) $\rightsquigarrow$ (*cms'*, *mem'*)
assumes *x-in-dom*: $x \in X$
shows *dma mem x* = *dma mem' x*
proof −
have $\bigwedge y.\ y \in \mathcal{C}$-*vars x* $\Longrightarrow$ *mem y* = *mem' y*
proof −
fix *y*
assume $y \in \mathcal{C}$-*vars x*
hence $y \in$ *vars-*$\mathcal{C}$ *X*
using *x-in-dom* **by**(*auto simp*: *vars-*$\mathcal{C}$*-def*)
thus *mem y* = *mem' y*
using *cr eval change-respecting-doesnt-modify* **by** *blast*
qed
thus *?thesis* **by**(*metis dma-*$\mathcal{C}$*-vars*)
qed

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

lemma *differing-empty-eq*:
⟦ *differing-vars mem mem'* = {} ⟧ $\Longrightarrow$ *mem* = *mem'*
unfolding *differing-vars-def*
by *auto*

lemma *adaptation-finite*:
finite (*dom* (*A*::('*Var*,'*Val*) *adaptation*))
apply(*rule finite-subset*[*OF - finite-memory*])
by *blast*

definition
globally-consistent :: ('*Var*, '*Val*) *adaptation* ⇒ '*Var Mds* ⇒ ('*Var*,'*Val*) *Mem*
⇒ ('*Var*,'*Val*) *Mem* ⇒ *bool*
where *globally-consistent A mds mem*$_1$ *mem*$_2$ ≡
(∀ *x*. *case A x of Some* (*v*,*v'*) ⇒ (*mem*$_1$ *x* ≠ *v* ∨ *mem*$_2$ *x* ≠ *v'*) ⟶ ¬ *var-asm-not-written*
mds x | - ⇒ *True*) ∧
(∀ *x*. *dma mem*$_1$ [‖$_1$ *A*] *x* ≠ *dma mem*$_1$ *x* ⟶ ¬ *var-asm-not-written mds x*)
∧
(∀ *x*. *dma* (*mem*$_1$ [‖$_1$ *A*]) *x* = *Low* ∧ (*x* ∉ *mds AsmNoReadOrWrite* ∨ *x* ∈
$\mathcal{C}$) ⟶ (*mem*$_1$ [‖$_1$ *A*]) *x* = (*mem*$_2$ [‖$_2$ *A*]) *x*)

lemma *globally-consistent-adapt-bisim*:
assumes *bisim*: ⟨*c*$_1$, *mds*, *mem*$_1$⟩ ≈ ⟨*c*$_2$, *mds*, *mem*$_2$⟩
assumes *globally-consistent*: *globally-consistent A mds mem*$_1$ *mem*$_2$

shows $\langle c_1,\ mds,\ mem_1\ [\|_1\ A]\rangle \approx \langle c_2,\ mds,\ mem_2\ [\|_2\ A]\rangle$
apply(*rule mm-equiv-glob-consistent*[*simplified closed-glob-consistent-def*, *rule-format*])
apply(*rule bisim*)
apply(*fold globally-consistent-def*)
by(*rule globally-consistent*)

lemma *mm-equiv-$\mathcal{C}$-eq*:
$(a,b) \approx (a',b') \Longrightarrow snd\ a = snd\ a' \Longrightarrow$
$\forall x{\in}\mathcal{C}.\ b\ x = b'\ x$
apply(*case-tac a*, *case-tac a'*)
using *mm-equiv-strong-low-bisim*[*simplified strong-low-bisim-mm-def*, *rule-format*]
by(*auto simp*: *low-mds-eq-def $\mathcal{C}$-Low*)

lemma *apply-adaptation-not-in-dom*:
$x \notin dom\ A \Longrightarrow apply\text{-}adaptation\ b\ blah\ A\ x = blah\ x$
apply(*simp add*: *apply-adaptation-def domIff split*: *option.splits*)
done

lemma *makes-compatible-invariant*:
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 *eval*: $(cms_1,\ mem_1) \rightsquigarrow_k (cms_1',\ mem_1')$
obtains $cms_2'\ mem_2'\ mems'$ **where**
$map\ snd\ cms_1' = map\ snd\ cms_2' \wedge$
$(cms_2,\ mem_2) \rightsquigarrow_k (cms_2',\ mem_2') \wedge$
$makes\text{-}compatible\ (cms_1',\ mem_1')\ (cms_2',\ mem_2')\ mems'$
proof −
let $?X = \lambda\ i.\ differing\text{-}vars\text{-}lists\ mem_1\ mem_2\ mems\ i$
from *sound-modes compat modes-eq* **have**
a: $\forall\ i < length\ cms_1.\ \forall\ x \in (?X\ i).\ doesnt\text{-}read\text{-}or\text{-}modify\ (fst\ (cms_1\ !\ i))\ x \wedge$
$doesnt\text{-}read\text{-}or\text{-}modify\ (fst\ (cms_2\ !\ i))\ x$
by (*metis compatible-different-no-read*)
from *eval* **have**
b: $k < length\ cms_1 \wedge (cms_1\ !\ k,\ mem_1) \rightsquigarrow (cms_1'\ !\ k,\ mem_1') \wedge$
$cms_1' = cms_1\ [k := cms_1'\ !\ k]$
by (*metis meval-elim nth-list-update-eq*)

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

let $?mds_k = snd\ (cms_1\ !\ k)$ **and**
$?mds_k' = snd\ (cms_1'\ !\ k)$ **and**
$?mems_1k = fst\ (mems\ !\ k)$ **and**
$?mems_2k = snd\ (mems\ !\ k)$ **and**
$?n = length\ cms_1$

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

then have
c: *change-respecting* (*cms*$_1$! *k*, *mem*$_1$) (*cms*$_1$' ! *k*, *mem*$_1$') (*?X k*)
using *noread-exists-change-respecting b a*
by (*metis surjective-pairing*)

from *compat* **have** $\bigwedge \sigma$. *dom* σ = *?X k* $\Longrightarrow$ *?mems*$_1$*k* [$\mapsto$ σ] = *mem*$_1$ [$\mapsto$ σ]
using *differing-vars-subst differing-vars-lists-def*
by (*metis Un-upper1 Un-subset-iff*)

hence
eval$_\sigma$: $\bigwedge \sigma$. *dom* σ = *?X k* $\Longrightarrow$ (*cms*$_1$! *k*, *?mems*$_1$*k* [$\mapsto$ σ]) $\rightsquigarrow$ (*cms*$_1$' ! *k*, *mem*$_1$' [$\mapsto$ σ])
by(*metis change-respecting-subst*[*rule-format*, **where** *X*=*?X k*] *c*)

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

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

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

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

then obtain *c*$_h$ *mem*$_h$ **where** *h-prop*:
(*cms*$_2$! *k*, *?mems*$_2$*k* [$\mapsto$ *h*]) $\rightsquigarrow$ $\langle$ *c*$_h$, *?mds*$_k$', *mem*$_h$ $\rangle$ $\wedge$
(*cms*$_1$' ! *k*, *mem*$_1$' [$\mapsto$ *h*]) $\approx$ $\langle$ *c*$_h$, *?mds*$_k$', *mem*$_h$ $\rangle$
using *d*
by *metis*

then have
change-respecting (*cms*$_2$! *k*, *?mems*$_2$*k* [$\mapsto$ *h*]) $\langle$ *c*$_h$, *?mds*$_k$', *mem*$_h$ $\rangle$ (*?X k*)
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**
$\forall$ σ. *dom* σ = *?X k* $\longrightarrow$

$(cms_2\ !\ k,\ ?mems_2k\ [\mapsto h]\ [\mapsto \sigma]) \rightsquigarrow \langle\ c_h,\ ?mds_k{}',\ mem_h\ [\mapsto \sigma]\ \rangle$
by (*metis change-respecting-subst*)

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

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

have $dom\sigma\text{-}mem_2$: $dom\ ?\sigma\text{-}mem_2 = ?X\ k$
by (*metis 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 $dom\sigma\text{-}mem_2$
unfolding *to-partial-def subst-def*
apply (*cases* $x \in ?X\ k$)
apply *auto*
by (*metis differing-vars-neg*)
qed

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

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

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

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

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

moreover

— This is the complicated part of the proof.
obtain *mems′* **where** *makes-compatible* (*cms*$_1$′, *mem*$_1$′) (*cms*$_2$′, *mem*$_2$′) *mems′*
proof
— This is used in two of the following cases, so we prove it beforehand:
have *x-unchanged*: $\bigwedge x.\ [\![\ x \in ?X\ k\ ]\!] \Longrightarrow$
mem$_1$ *x* = *mem*$_1$′ *x* ∧ *mem*$_2$ *x* = *mem*$_2$′ *x* ∧ *dma mem*$_1$ *x* = *dma mem*$_1$′ *x*
proof(*intro conjI*)
fix *x*
assume *x* ∈ *?X k*
thus *mem*$_1$ *x* = *mem*$_1$′ *x*
using *a b c change-respecting-doesnt-modify domh*
by (*metis* (*erased*, *opaque-lifting*) *Un-upper1 contra-subsetD*)
next
fix *x*
assume *x* ∈ *?X k*

hence *eq-mem*$_2$: *?σ-mem*$_2$ *x* = *Some* (*mem*$_2$ *x*)
by (*metis restrict-in to-partial-def*)
hence *?mems*$_2$*k* [↦ *h*] [↦ *?σ-mem*$_2$] *x* = *mem*$_2$ *x*
by (*auto simp*: *subst-def*)

moreover have *mem*$_h$ [↦ *?σ-mem*$_2$] *x* = *mem*$_2$ *x*
by (*auto simp*: *subst-def* ‹*x* ∈ *?X k*› *eq-mem*$_2$)

ultimately have *?mems*$_2$*k* [↦ *h*] [↦ *?σ-mem*$_2$] *x* = *mem*$_h$ [↦ *?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*)
next
fix *x*
assume *x* ∈ *?X k*
thus *dma mem*$_1$ *x* = *dma mem*$_1$′ *x*
using *a b c change-respecting-doesnt-modify-dma domh*
by (*metis* (*erased*, *opaque-lifting*))
qed

define *mems′-k* **where** *mems′-k x* =
(*if x* ∉ *?X k*
then (*mem*$_1$′ *x*, *mem*$_2$′ *x*)
else (*?mems*$_1$*k x*, *?mems*$_2$*k x*)) **for** *x*

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 mem*$_1$′ *x* = *High*))
then (*mem*$_1$′ *x*, *mem*$_2$′ *x*)

```
        else if ((mem1 x ≠ mem1' x ∨ mem2 x ≠ mem2' x) ∧
                 (mem1' x ≠ mem2' x ∧ dma mem1' x = Low))
             then (some-val, some-val)
               else if dma mem1 x = High ∧ dma mem1' x = Low then (mem1 x,
mem1 x)
            else if dma mem1' x = dma mem1 x then (fst (mems ! i) x, snd (mems
! i) x)
             else (mem1' x, mem2' 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 cms1]
    from b have mems'-k-simp: mems' ! k = (fst ∘ mems'-k, snd ∘ mems'-k)
      unfolding mems'-def
      by auto

    have mems'-simp2: ⋀i. ⟦ i ≠ k; i < length cms1 ⟧ ⟹
      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 = mem1' x ∧ snd (mems' ! k) x = mem2' x
      unfolding mems'-k-simp mems'-k-def
      by auto
    have mems'-k-2 [simp]: ⋀ x. ⟦ x ∈ ?X k ⟧ ⟹
     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:
     ⋀ P x.
       ⟦
        ⟦ x ∉ ?X k;
          fst (mems' ! k) x = mem1' x;
          snd (mems' ! k) x = mem2' x ⟧ ⟹ P x;
        ⟦ x ∈ ?X k;
          fst (mems' ! k) x = fst (mems ! k) x;
          snd (mems' ! k) x = snd (mems ! k) x ⟧ ⟹ P x ⟧ ⟹ P x
      apply(case-tac x ∉ ?X k)
       apply simp
      apply simp
      done
```

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*]:
$\bigwedge i\ x.\ [\![\ i \neq k;\ i < length\ cms_1;$
$mem_1\ x \neq mem_1'\ x \vee mem_2\ x \neq mem_2'\ x;$
$mem_1'\ x = mem_2'\ x \vee dma\ mem_1'\ x = High\ ]\!] \Longrightarrow$
$fst\ (mems'\ !\ i)\ x = mem_1'\ x \wedge snd\ (mems'\ !\ i)\ x = mem_2'\ x$
unfolding *mems′-i-def mems′-i-simp*
by *auto*

have *mems′-i-2* [*simp*]:
$\bigwedge i\ x.\ [\![\ i \neq k;\ i < length\ cms_1;$
$mem_1\ x \neq mem_1'\ x \vee mem_2\ x \neq mem_2'\ x;$
$mem_1'\ x \neq mem_2'\ x;\ dma\ mem_1'\ x = Low\ ]\!] \Longrightarrow$
$fst\ (mems'\ !\ i)\ x = some\text{-}val \wedge snd\ (mems'\ !\ i)\ x = some\text{-}val$
unfolding *mems′-i-def mems′-i-simp*
by *auto*
have *mems′-i-3* [*simp*]:
$\bigwedge i\ x.\ [\![\ i \neq k;\ i < length\ cms_1;$
$mem_1\ x = mem_1'\ x;\ mem_2\ x = mem_2'\ x;$
$dma\ mem_1\ x = High \wedge dma\ mem_1'\ x = Low\ ]\!] \Longrightarrow$
$fst\ (mems'\ !\ i)\ x = mem_1\ x \wedge snd\ (mems'\ !\ i)\ x = mem_1\ x$
unfolding *mems′-i-def mems′-i-simp*
by *auto*

have *mems′-i-4* [*simp*]:
$\bigwedge i\ x.\ [\![\ i \neq k;\ i < length\ cms_1;$
$mem_1\ x = mem_1'\ x;\ mem_2\ x = mem_2'\ x;$
$dma\ mem_1\ x = Low \vee dma\ mem_1'\ x = High;$
$dma\ mem_1'\ x = dma\ mem_1\ x\ ]\!] \Longrightarrow$
$fst\ (mems'\ !\ i)\ x = fst\ (mems\ !\ i)\ x \wedge snd\ (mems'\ !\ i)\ x = snd\ (mems\ !\ i)\ x$
unfolding *mems′-i-def mems′-i-simp*
by *auto*

have *mems′-i-5* [*simp*]:
$\bigwedge i\ x.\ [\![\ i \neq k;\ i < length\ cms_1;$
$mem_1\ x = mem_1'\ x;\ mem_2\ x = mem_2'\ x;$
$dma\ mem_1\ x = Low \wedge dma\ mem_1'\ x = High;$
$dma\ mem_1'\ x \neq dma\ mem_1\ x\ ]\!] \Longrightarrow$
$fst\ (mems'\ !\ i)\ x = mem_1'\ x \wedge snd\ (mems'\ !\ i)\ x = mem_2'\ x$
unfolding *mems′-i-def mems′-i-simp*
by *auto*

have *mems′-i-cases*:
$\bigwedge P\ i\ x.$
$[\![\ i \neq k;\ i < length\ cms_1;$
$[\![\ mem_1\ x \neq mem_1'\ x \vee mem_2\ x \neq mem_2'\ x;$
$mem_1'\ x = mem_2'\ x \vee dma\ mem_1'\ x = High;$
$fst\ (mems'\ !\ i)\ x = mem_1'\ x;\ snd\ (mems'\ !\ i)\ x = mem_2'\ x\ ]\!] \Longrightarrow P\ x;$
$[\![\ mem_1\ x \neq mem_1'\ x \vee mem_2\ x \neq mem_2'\ x;$
$mem_1'\ x \neq\ mem_2'\ x;\ dma\ mem_1'\ x = Low;$
$fst\ (mems'\ !\ i)\ x = some\text{-}val;\ snd\ (mems'\ !\ i)\ x = some\text{-}val\ ]\!] \Longrightarrow P\ x;$
$[\![\ mem_1\ x = mem_1'\ x;\ mem_2\ x = mem_2'\ x;\ dma\ mem_1\ x = High;$
$dma\ mem_1'\ x = Low;$
$fst\ (mems'\ !\ i)\ x = mem_1\ x;\ snd\ (mems'\ !\ i)\ x = mem_1\ x\ ]\!] \Longrightarrow P\ x;$
$[\![\ mem_1\ x = mem_1'\ x;\ mem_2\ x = mem_2'\ x;\ dma\ mem_1\ x = Low \vee dma\ mem_1'\ x = High;$
$dma\ mem_1'\ x = dma\ mem_1\ x;$
$fst\ (mems'\ !\ i)\ x = fst\ (mems\ !\ i)\ x;\ snd\ (mems'\ !\ i)\ x = snd\ (mems\ !\ i)\ x\ ]\!] \Longrightarrow P\ x;$
$[\![\ mem_1\ x = mem_1'\ x;\ mem_2\ x = mem_2'\ x;\ dma\ mem_1\ x = Low;\ dma\ mem_1'\ x = High;$
$fst\ (mems'\ !\ i)\ x = mem_1'\ x;\ snd\ (mems'\ !\ i)\ x = mem_2'\ x\ ]\!] \Longrightarrow P\ x\ \ ]\!]$
$\Longrightarrow P\ x$
using *mems′-i-1 mems′-i-2 mems′-i-3 mems′-i-4 mems′-i-5*
by (*metis* (*full-types*) *Sec.exhaust*)

let *?X′* = λ *i. differing-vars-lists mem₁′ mem₂′ mems′ i*

have *len-unchanged*: *length cms₁′* = *length cms₁*
by (*metis cms₂′-def equal-size length-list-update new-length*)

have *mm-equiv′*: (*cms₁′* ! *k*, *subst* (*?σ-mem₂*) *mem₁′*) $\approx$ $\langle c_h,$ *snd* (*cms₁′* ! *k*), *mem₂′*$\rangle$
apply(*simp add*: *mem₂′-def*)
apply(*rule g*[*THEN conjunct2*])
apply(*rule dom-restrict-total*)
done

hence $\mathcal{C}$*-subst-eq*: $\forall\, x{\in}\mathcal{C}.$ (*subst* (*?σ-mem₂*) *mem₁′*) *x* = *mem₂′ x*
apply(*rule mm-equiv-*$\mathcal{C}$*-eq*)
by *simp*

have *low-mds-eq′*: (*subst* (*?σ-mem₂*) *mem₁′*) $=_{snd\ (cms_1'\ !\ k)}^{l}$ *mem₂′*
apply(*rule mm-equiv-low-eq*[**where** *c₁*=*fst* (*cms₁′* ! *k*)])
apply(*force intro*: *mm-equiv′*)
done

have $\mathcal{C}$*-subst-eq-idemp*: $\bigwedge x.\ x \in \mathcal{C} \Longrightarrow$ (*subst* (*?σ-mem₂*) *mem₁′*) *x* = *mem₁′ x*
apply(*rule subst-not-in-dom*)
apply(*rule notI*)
apply(*simp add*: *dom-restrict-total*)

```
 using compat b by force

from C-subst-eq C-subst-eq-idemp
have C-eq: ⋀x. x ∈ C ⟹ mem₁' x = mem₂' x
 by simp

have not-control: ⋀x i. i < length cms₁' ⟹ x ∈ ?X' i ⟹ x ∉ C
proof(rule ccontr, clarsimp)
  fix x i
  let ?mems₁i = fst (mems ! i)
  let ?mems₂i = snd (mems ! i)
  let ?mems₁'i = fst (mems' ! i)
  let ?mems₂'i = snd (mems' ! i)
  assume i < length cms₁'
  have i < length cms₁ by (metis len-unchanged ‹i < length cms₁'›)
  assume x ∈ ?X' i
  assume x ∈ C
  have x ∉ ?X i
    using compat ‹i < length cms₁'› len-unchanged new-length
    by (metis ‹x ∈ C› compat-different)
  from ‹x ∈ C› have  mem₁' x = mem₂' x by(rule C-eq)
  from ‹x ∈ C› have dma mem₁' x = Low by(simp add: C-Low)
  show False
  proof(cases i = k)
    assume eq[simp]: i = k
    show ?thesis
    using ‹x ∉ ?X i› ‹x ∈ ?X' i›
    by(force simp: differing-vars-lists-def differing-vars-def)
  next
    assume neq: i ≠ k
    thus ?thesis
    using ‹x ∈ ?X' i› ‹x ∉ ?X i› ‹x ∈ C› C-Low ‹mem₁' x = mem₂' x›
    by(force elim: mems'-i-cases[of i x λx. False, OF - ‹i < length cms₁›]
            simp: differing-vars-lists-def differing-vars-def)
  qed
qed

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 Some (*mem₁′ x*)
else None) **for** *x*
have *domσ′*: *dom σ′* = *?X k*
using *σ′-def* [*abs-def*]
apply (*clarsimp*, *safe*)
by(*metis domI domIff*, *metis* ‹*i* = *k*› *domD domσ*)
have *diff-vars-impl* [*simp*]: ⋀*x*. *x* ∈ *?X′ k* ⟹ *x* ∈ *?X k*
proof (*rule ccontr*)
fix *x*
assume *x* ∉ *?X k*
hence *mem₁ x* = *?mems₁k x* ∧ *mem₂ x* = *?mems₂k x*
by (*metis differing-vars-neg*)
from ‹*x* ∉ *?X k*› **have** *?mems₁′i x* = *mem₁′ x* ∧ *?mems₂′i x* = *mem₂′ x*
by *auto*
moreover
assume *x* ∈ *?X′ k*
hence *mem₁′ x* ≠ *?mems₁′i x* ∨ *mem₂′ x* ≠ *?mems₂′i x*
by (*metis* ‹*i* = *k*› *differing-vars-elim*)
ultimately show *False*
by *auto*
qed

have *?mems₁′i* [↦ *σ*] = *mem₁′* [↦ *σ′*]
proof (*rule ext*)
fix *x*

show *?mems₁′i* [↦ *σ*] *x* = *mem₁′* [↦ *σ′*] *x*
proof (*cases x* ∈ *?X′ k*)
assume *x-in-X′k*: *x* ∈ *?X′ k*

then obtain *v* **where** *σ x* = *Some v*
by (*metis domσ domD* ‹*i* = *k*›)
hence *?mems₁′i* [↦ *σ*] *x* = *v*
using ‹*x* ∈ *?X′ k*› *domσ*
by (*auto simp*: *subst-def*)

```
    moreover
    from domσ' and ‹x ∈ ?X' k› have x ∈ dom σ' by simp

    hence mem₁' [↦ σ'] x = v
      using domσ'
      unfolding 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' [↦ 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(case-tac x ∈ ?X k)
      assume x ∈ ?X k
      from ‹x ∉ ?X' k› have mem₁' x = ?mems₁'i x
        by(metis differing-vars-neg ‹i = k›)
      then have σ' x = Some (?mems₁'i x)
        unfolding σ'-def
        using domσ' domh
        by(simp add: ‹x ∈ ?X k› ‹x ∉ ?X' k›)
      hence mem₁' [↦ σ'] x = ?mems₁'i x
        unfolding subst-def
        by (metis option.simps(5))
      thus ?thesis
        by (metis ‹?mems₁'i [↦σ] x = ?mems₁'i x›)
    next
      assume x ∉ ?X k
      then have mem₁' [↦ σ'] x = mem₁' x
        by (metis domσ' 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] = memₕ [↦ σ']
proof (rule ext)
  fix x

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

then obtain v **where** $\sigma\ x = Some\ v$
using $dom\sigma$
by (*metis domD* ‹$i = k$›)
hence $?mems_2'i\ [\mapsto \sigma]\ x = v$
using ‹$x \in ?X'\ k$› $dom\sigma$
unfolding *subst-def*
by (*metis option.simps*(5))
moreover
from ‹$x \in ?X'\ k$› **have** $x \in ?X\ k$
by *auto*
hence $x \in dom\ (\sigma')$
by (*metis dom*σ' ‹$x \in ?X'\ k$›)
hence $mem_2'\ [\mapsto \sigma']\ x = v$
using $dom\sigma'\ c$
unfolding *subst-def*
by (*metis* σ'*-def* ‹$\sigma\ x = Some\ v$› *diff-vars-impl option.simps*(5) ‹$x \in ?X'\ k$›)

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

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

proof(*case-tac* $x \in ?X\ k$)
assume $x \in ?X\ k$

hence $mem_1\ x = mem_1'\ x \wedge mem_2\ x = mem_2'\ x$ **by** (*metis x-unchanged*)

moreover from ‹$x \notin ?X'\ k$› ‹$i = k$› **have** $?mems_1'i\ x = mem_1'\ x \wedge ?mems_2'i\ x = mem_2'\ x$
by(*metis differing-vars-neg*)

moreover from ‹$x \in ?X\ k$› **have** $fst\ (mems\ !\ i)\ x \neq mem_1\ x \vee snd\ (mems\ !\ i)\ x \neq mem_2\ x$
by(*metis differing-vars-elim* ‹$i = k$›)

moreover from ‹$x \in ?X\ k$› **have** $fst\ (mems'\ !\ i)\ x = fst\ (mems\ !\ i)\ x \wedge$
$snd\ (mems'\ !\ i)\ x = snd\ (mems\ !\ i)\ x$
by(*metis mems'-k-2* ‹$i = k$›)

ultimately have *False* **by** *auto*

```
      thus ?thesis by blast
    next
      assume x ∉ ?X k
      hence x ∉ dom σ' by (simp add: domσ')
      then have mem_h [↦ σ''] x = mem_h x
        by (metis subst-not-in-dom)
      moreover
      have ?mems_2'i x = mem_2' x
        by (metis ‹i = k›  mems'-k-1 ‹x ∉ ?X k›)

      hence ?mems_2'i x = mem_h x
        unfolding mem_2'-def
        by (metis domσ-mem_2 subst-not-in-dom ‹x ∉ ?X k›)
      ultimately show ?thesis
        by (metis ‹?mems_2'i [↦σ] x = ?mems_2'i x›)
    qed
  qed
qed

ultimately show
  (cms_1' ! i, (fst (mems' ! i)) [↦ σ]) ≈ (cms_2' ! i, (snd (mems' ! i)) [↦ σ])
  using domσ domσ' g b ‹i = k›
  by (metis c_2'-def cms_2'-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 (mem_1' x)
     else None) for x
let ?mems_1i = fst (mems ! i) and
    ?mems_2i = 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.
        ((?mems_1'i [↦ σ] x ≠ ?mems_1i [↦ σ''] x ∨
          ?mems_2'i [↦ σ] x ≠ ?mems_2i [↦ σ''] x) ∧
        (dma mem_1 x = Low ∨ dma mem_1' x = High) ∧
        (dma mem_1' x = dma mem_1 x))
        ⟶ (mem_1' x ≠ mem_1 x ∨ mem_2' x ≠ mem_2 x)
proof −
  fix x
  {
```

```
        assume eq-mem: mem₁' x = mem₁ x ∧ mem₂' x = mem₂ x
          and clas: dma mem₁ x = Low ∨ dma mem₁' x = High
          and clas-eq: dma mem₁' x = dma mem₁ x
        hence mems'-simp: ?mems₁'i x = ?mems₁i x ∧ ?mems₂'i x = ?mems₂i
x
          using mems'-i-4
          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
            by (clarsimp simp: subst-def mems'-simp split: option.splits)
        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


    have o-downgrade: ⋀x. x ∉ ?X' i ∧ (subst σ (fst (mems' ! i)) x ≠ subst σ'
(fst (mems ! i)) x ∨
                subst σ (snd (mems' ! i)) x ≠ subst σ' (snd (mems ! i)) x) ∧
               (dma mem₁ x = High ∧ dma mem₁' x = Low) ⟶
                mem₁' x ≠ mem₁ x ∨ mem₂' x ≠ mem₂ x
      proof −
      fix x {
        assume mem-eq: mem₁' x = mem₁ x ∧ mem₂' x = mem₂ x
          and clas: (dma mem₁ x = High ∧ dma mem₁' x = Low)
          and notin: x ∉ ?X' i
       hence mems'-simp [simp]: ?mems₁'i x = mem₁ x ∧ ?mems₂'i x = mem₁
```

x
using *mems′-i-3*
by (*metis* ‹*i* ≠ *k*› *b i-le length-list-update*)
have
?mems$_1$′ *i* [↦ *σ*] *x* = *?mems*$_1$ *i* [↦ *σ′*] *x* ∧ *?mems*$_2$′ *i* [↦ *σ*] *x* = *?mems*$_2$ *i* [↦ *σ′*] *x*
proof (*cases x* ∈ *?X′ i*)
assume *x* ∈ *?X′ i*
thus *?thesis* **using** *notin* **by** *blast*
next
assume *x* ∉ *?X′ i*
hence *?mems*$_1$′ *i x* = *mem*$_1$′ *x* ∧ *?mems*$_2$′ *i x* = *mem*$_2$′ *x*
by (*metis differing-vars-neg*)
moreover have *x* ∉ *?X i*
using *clas compat i-le len-unchanged*
by (*force*)
ultimately show *?thesis*
using *domσ* ‹*dom σ′* = *?X i*› ‹*x* ∉ *?X′ i*› **apply**(*simp add*: *subst-not-in-dom*)
apply(*simp add*: *mem-eq*)
apply(*force simp*: *differing-vars-def differing-vars-lists-def*)
done
qed

} **thus** *?thesis x* **by** *blast*
qed

have *modifies-no-var-asm-not-written*:
⋀*x*. *mem*$_1$′ *x* ≠ *mem*$_1$ *x* ∨ *mem*$_2$′ *x* ≠ *mem*$_2$ *x* ∨
dma mem$_1$′ *x* ≠ *dma mem*$_1$ *x* ∨ *dma mem*$_2$′ *x* ≠ *dma mem*$_2$ *x* ⟹
¬ *var-asm-not-written* (*snd* (*cms*$_1$! *i*)) *x*
proof −
fix *x*
assume *mem*$_1$′ *x* ≠ *mem*$_1$ *x* ∨ *mem*$_2$′ *x* ≠ *mem*$_2$ *x* ∨ *dma mem*$_1$′ *x* ≠ *dma mem*$_1$ *x* ∨ *dma mem*$_2$′ *x* ≠ *dma mem*$_2$ *x*
hence *modified*: ¬ (*doesnt-modify* (*fst* (*cms*$_1$! *k*)) *x*) ∨ ¬ (*doesnt-modify* (*fst* (*cms*$_2$! *k*)) *x*)
using *b i*
unfolding *doesnt-modify-def*
by (*metis surjective-pairing*)
hence *modified-r*: ¬ (*doesnt-read-or-modify* (*fst* (*cms*$_1$! *k*)) *x*) ∨ ¬ (*doesnt-read-or-modify* (*fst* (*cms*$_2$! *k*)) *x*) **using** *doesnt-read-or-modify-doesnt-modify* **by** *fastforce*

from *sound-modes* **have** *loc-modes*:
locally-sound-mode-use (*cms*$_1$! *k*, *mem*$_1$) ∧
locally-sound-mode-use (*cms*$_2$! *k*, *mem*$_2$)
unfolding *sound-mode-use.simps*
by (*metis b equal-size list-all-length*)

moreover
have *snd* (*cms*$_1$! *k*) = *snd* (*cms*$_2$! *k*)
by (*metis b equal-size modes-eq nth-map*)
have (*cms*$_1$! *k*, *mem*$_1$) ∈ *loc-reach* (*cms*$_1$! *k*, *mem*$_1$)
using *loc-reach.refl* **by** *auto*
hence *guars*:
x ∈ *snd* (*cms*$_1$! *k*) *GuarNoWrite* ⟶ *doesnt-modify* (*fst* (*cms*$_1$! *k*)) *x* ∧
x ∈ *snd* (*cms*$_2$! *k*) *GuarNoWrite* ⟶ *doesnt-modify* (*fst* (*cms*$_1$! *k*)) *x* ∧
x ∈ *snd* (*cms*$_1$! *k*) *GuarNoReadOrWrite* ⟶ *doesnt-read-or-modify* (*fst* (*cms*$_1$! *k*)) *x* ∧
x ∈ *snd* (*cms*$_2$! *k*) *GuarNoReadOrWrite* ⟶ *doesnt-read-or-modify* (*fst* (*cms*$_1$! *k*)) *x*
using *loc-modes*
unfolding *locally-sound-mode-use-def* ‹*snd* (*cms*$_1$! *k*) = *snd* (*cms*$_2$! *k*)›
by (*metis loc-reach.refl surjective-pairing*)

hence *x* ∉ *snd* (*cms*$_1$! *k*) *GuarNoWrite* ∧ *x* ∉ *snd* (*cms*$_1$! *k*) *GuarNoReadOrWrite*
using *modified modified-r loc-modes locally-sound-mode-use-def*
by (*metis* (*no-types*, *lifting*) ‹(*cms*$_2$, *mem*$_2$) ⇝$_k$ (*cms*$_2$′, *mem*$_2$′)› *b locally-sound-respects-guarantees modes-eq nth-map meval-elim respects-own-guarantees-def sifum-security-init-axioms*)
moreover
from *sound-modes* **have** *compatible-modes* (*map snd cms*$_1$)
by (*metis globally-sound-modes-compatible sound-mode-use.simps*)

ultimately show (*?thesis x*)
unfolding *compatible-modes-def var-asm-not-written-def*
using ‹*i* ≠ *k*› *i-le*
by (*metis* (*no-types*) *b length-list-update length-map nth-map*)
qed

from *o o-downgrade* **have**
p: ⋀ *x*. ⟦ *?mems*$_1$′*i* [↦ *σ*] *x* ≠ *?mems*$_1$*i* [↦ *σ*′] *x* ∨
?mems$_2$′*i* [↦ *σ*] *x* ≠ *?mems*$_2$*i* [↦ *σ*′] *x*;
x ∉ *?X*′ *i* ∨ ((*dma mem*$_1$ *x* = *Low* ∨ *dma mem*$_1$′ *x* = *High*) ∧
(*dma mem*$_1$′ *x* = *dma mem*$_1$ *x*)) ⟧ ⟹
¬ *var-asm-not-written* (*snd* (*cms*$_1$! *i*)) *x*
proof −
fix *x*
assume *mems-neq*:
?mems$_1$′*i* [↦ *σ*] *x* ≠ *?mems*$_1$*i* [↦ *σ*′] *x* ∨ *?mems*$_2$′*i* [↦ *σ*] *x* ≠ *?mems*$_2$*i* [↦ *σ*′] *x*
and *nin*:
x ∉ *?X*′ *i* ∨ ((*dma mem*$_1$ *x* = *Low* ∨ *dma mem*$_1$′ *x* = *High*) ∧
(*dma mem*$_1$′ *x* = *dma mem*$_1$ *x*))

hence $mem_1\,'\,x \neq mem_1\,x \vee mem_2\,'\,x \neq mem_2\,x \vee dma\;mem_1\,'\,x \neq dma\;mem_1\,x$

apply $-$
apply(*erule disjE*[**where** $P{=}x \notin ?X'\,i$])
apply(*case-tac* $(dma\;mem_1\;x = High \wedge dma\;mem_1\,'\,x = Low)$)
apply(*metis o-downgrade*[*rule-format*])
apply(*case-tac* $dma\;mem_1\,'\,x = dma\;mem_1\;x$)

apply(*metis* (*poly-guards-query*) *o Sec.exhaust*)

apply *fastforce*
apply(*metis* (*poly-guards-query*) *o Sec.exhaust*)
done
thus *?thesis x*
by(*force simp*: *modifies-no-var-asm-not-written*)
qed

have q':
$\bigwedge x.\ [\![\ dma\;mem_1\;x = Low;\ dma\;mem_1\,'\,x = Low;$
$?mems_1\,'i\;[\mapsto \sigma]\;x \neq ?mems_1 i\;[\mapsto \sigma']\;x \vee$
$?mems_2\,'i\;[\mapsto \sigma]\;x \neq ?mems_2 i\;[\mapsto \sigma']\;x;$
$x \notin ?X'\,i\ ]\!] \Longrightarrow$
$mem_1\,'\,x = mem_2\,'\,x$
by (*metis* ‹$i \neq k$› *b compat-different-vars i-le length-list-update mems'-i-2 o*)
have $i < length\;cms_1$
by (*metis* cms_2*'-def equal-size i-le length-list-update new-length*)
with *compat* **and** ‹$dom\;\sigma' = ?X\;i$› **have**
bisim: $(cms_1\;!\;i,\ ?mems_1 i\;[\mapsto \sigma'])$ ≈ $(cms_2\;!\;i,\ ?mems_2 i\;[\mapsto \sigma'])$
by *auto*

define σ'_k **where** $\sigma'_k\;x = (if\;x \in ?X\;k\;then\;Some\;(undefined::'Val)\;else\;None)$ **for** x
have $dom\;\sigma'_k = ?X\;k$ **unfolding** σ'_k*-def* **by** (*simp add*: *dom-def*)
with *compat* **and** ‹$dom\;\sigma'_k = ?X\;k$› **and** *b* **have**
$bisim_k$: $(cms_1\;!\;k,\ ?mems_1 k\;[\mapsto \sigma'_k])$ ≈ $(cms_2\;!\;k,\ ?mems_2 k\;[\mapsto \sigma'_k])$
by *auto*

have *q-downgrade*:
$\bigwedge x.\ [\![\ dma\;mem_1\;x = High;\ dma\;mem_1\,'\,x = Low;$
$?mems_1\,'i\;[\mapsto \sigma]\;x \neq ?mems_1 i\;[\mapsto \sigma']\;x \vee$
$?mems_2\,'i\;[\mapsto \sigma]\;x \neq ?mems_2 i\;[\mapsto \sigma']\;x;$
$x \notin ?X'\,i\ ]\!] \Longrightarrow$
$mem_1\,'\,x = mem_2\,'\,x$
by (*metis* (*erased*, *opaque-lifting*) ‹$i \neq k$› *compat-different-vars i-le len-unchanged mems'-i-2 o-downgrade*)

have q: $\bigwedge x.\ [\![\ dma\;mem_1\,'\,x = Low;$

```
        ?mems₁'i [↦ σ] x ≠ ?mems₁i [↦ σ'] x ∨
        ?mems₂'i [↦ σ] x ≠ ?mems₂i [↦ σ'] x;
        x ∉ ?X' i ⟧ ⟹
       mem₁' x = mem₂' x
 apply(case-tac dma mem₁ x)
  apply(blast intro: q-downgrade)
 by(blast intro: q')

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 (?mems₁'i [↦ σ]) 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 dma-eq: dma (?mems₁'i [↦ σ]) = dma mem₁'
 apply(rule dma-C)
 apply(rule ballI)
 apply(case-tac x ∈ ?X' i)
  apply(drule not-control[rotated])
   apply (metis i-le)
  apply blast
 apply(subst subst-not-in-dom)
  apply(simp add: domσ)
 apply(simp add: differing-vars-lists-def differing-vars-def)
```

done

have *dma-eq''*: *dma* (*?mems*$_1$ *i* [$\mapsto$ σ']) = *dma mem*$_1$
apply(*rule dma-*$\mathcal{C}$)
apply(*rule ballI*)
apply(*case-tac* $x \in$ *?X i*)
using *compat compat i-le len-unchanged* **apply** *fastforce*
apply(*subst subst-not-in-dom*)
apply(*simp add*: ‹*dom* σ' = *?X i*›)
apply(*simp add*: *differing-vars-lists-def differing-vars-def*)
done

have *dma-eq'*: *dma* (*subst* ((*to-partial mem*$_2$ |‘ *differing-vars-lists mem*$_1$ *mem*$_2$ *mems k*)) *mem*$_1'$) = *dma mem*$_1'$
using *compat b*
by(*force intro*!: *dma-*$\mathcal{C}$ *subst-not-in-dom*)

have *A-correct*:
$\bigwedge$ *x*.
?mems$_1$ *i* [$\mapsto$ σ'] [$\|_1$ *A*] *x* = *?mems*$_1'$ *i* [$\mapsto$ σ] *x* $\wedge$
?mems$_2$ *i* [$\mapsto$ σ'] [$\|_2$ *A*] *x* = *?mems*$_2'$ *i* [$\mapsto$ σ] *x*
proof −
fix *x*
show *?thesis x*
(**is** *?Eq*$_1$ $\wedge$ *?Eq*$_2$)
proof (*cases* $x \in$ *?*Δ)
assume $x \in$ *?*Δ
hence *diff*:
?mems$_1'$ *i* [$\mapsto$ σ] *x* $\neq$ *?mems*$_1$ *i* [$\mapsto$ σ'] *x* $\vee$ *?mems*$_2'$ *i* [$\mapsto$ σ] *x* $\neq$ *?mems*$_2$ *i* [$\mapsto$ σ'] *x*
by (*auto simp*: *differing-vars-def*)
show *?thesis*
proof (*cases dma* (*?mems*$_1'$ *i* [$\mapsto$ σ]) *x*)
assume *dma* (*?mems*$_1'$ *i* [$\mapsto$ σ]) *x* = *High*
from ‹*dma* (*?mems*$_1'$ *i* [$\mapsto$ σ]) *x* = *High*› **have** *A-simp* [*simp*]:
A x = *Some* (*?mems*$_1'$ *i* [$\mapsto$ σ] *x*, *?mems*$_2'$ *i* [$\mapsto$ σ] *x*)
unfolding *A-def*
by (*metis* ‹$x \in$ *?*Δ›)
from *A-simp* **have** *?Eq*$_1$ *?Eq*$_2$
unfolding *A-def apply-adaptation-def*
by *auto*
thus *?thesis*
by *auto*
next
assume *dma* (*?mems*$_1'$ *i* [$\mapsto$ σ]) *x* = *Low*
show *?thesis*
proof (*cases* $x \in$ *?X'* *i*)
assume $x \in$ *?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 (?mems₁'i [↦ σ]) 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›)
      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› ‹x ∉ ?X' i› ‹dma (?mems₁'i [↦ σ]) x = Low›
        by auto

      from q have mem₁' x = mem₂' x
      by (metis ‹dma (?mems₁'i [↦ σ]) x = Low› diff ‹x ∉ ?X' i› dma-eq
dma-eq'')

      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)
  from ‹A x = None› have x ∉ dom A
    by (metis domIff)
  from ‹x ∉ ?Δ› have ?mems₁i [↦ 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

        thus ?thesis
          by auto
      qed
    qed
    hence adapt-eq:
        ?mems₁ i [↦ σ'] [||₁ A] = ?mems₁' i [↦ σ] ∧
         ?mems₂ i [↦ σ'] [||₂ A] = ?mems₂' i [↦ σ]
      by auto

    have cms₁' ! i = cms₁ ! i
      by (metis ‹i ≠ k› b nth-list-update-neq)

     have A-correct': globally-consistent A (snd (cms₁ ! i)) (?mems₁ i [↦ σ'])
(?mems₂ i [↦ σ'])

      apply(clarsimp simp: globally-consistent-def)
      apply(rule conjI)
       apply(split option.split)
       apply(intro allI conjI)
        apply simp
       apply(intro allI impI)
       apply(split prod.split)
       apply(intro allI impI)
       apply(simp only:)
       proof −
        fix x v v'
        assume A-updates-x₁: A x = Some (v, v')
            and A-updates-x₂:subst σ' (fst (mems ! i)) x ≠ v ∨ subst σ' (snd
(mems ! i)) x ≠ v'
        hence x ∈ dom A by(blast)
        hence diff:
        ?mems₁' i [↦ σ] x ≠ ?mems₁ i [↦ σ'] x ∨ ?mems₂' i [↦ σ] x ≠ ?mems₂ i
[↦ σ'] x
          by (auto simp: differing-vars-def domA)
        show ¬ var-asm-not-written (snd (cms₁ ! i)) x
         proof (cases x ∉ ?X' i ∨ ((dma mem₁ x = Low ∨ dma mem₁' x =
High) ∧ dma mem₁' x = dma mem₁ x))
          assume x ∉ ?X' i ∨ ((dma mem₁ x = Low ∨ dma mem₁' x = High)
∧ (dma mem₁' x = dma mem₁ x))
          from this p and diff show writable: ¬ var-asm-not-written (snd (cms₁
! i)) x
            by auto
          next
            assume ¬ (x ∉ ?X' i ∨ ((dma mem₁ x = Low ∨ dma mem₁' x =
```

High) ∧ (*dma mem*$_1$′ *x* = *dma mem*$_1$ *x*)))
hence *x* ∈ *?X*′ *i* ((*dma mem*$_1$ *x* = *High* ∧ *dma mem*$_1$′ *x* = *Low*) ∨ (*dma mem*$_1$′ *x* ≠ *dma mem*$_1$ *x*))
by (*metis Sec.exhaust*)+

thus *?thesis* **by**(*fastforce simp add*: *modifies-no-var-asm-not-written*)
qed

next

have *reclas*: (∀ *x*. *dma* ((*subst* σ′ (*fst* (*mems* ! *i*))) [‖$_1$ *A*]) *x* ≠ *dma* (*subst* σ′ (*fst* (*mems* ! *i*))) *x* ⟶
¬ *var-asm-not-written* (*snd* (*cms*$_1$! *i*)) *x*)
apply(*simp add*: *adapt-eq dma-eq dma-eq*′′)
apply(*simp add*: *modifies-no-var-asm-not-written*)
done

have *snd* (*cms*$_2$! *i*) = *snd* (*cms*$_1$! *i*)
by (*metis* ‹*i* < *length cms*$_1$› *equal-size modes-eq nth-map*)

hence *low-mds-eq*: (*subst* σ′ (*fst* (*mems* ! *i*))) =$_{snd\ (cms_1\ !\ i)}{}^{l}$ (*subst* σ′ (*snd* (*mems* ! *i*)))
apply −
apply(*rule mm-equiv-low-eq*[**where** *c*$_1$=*fst* (*cms*$_1$! *i*) **and** *c*$_2$=*fst* (*cms*$_2$! *i*)])
using *bisim*
by (*metis prod.collapse*)

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

hence *low-mds-eq*$_k$: (*subst* σ′$_k$ (*fst* (*mems* ! *k*))) =$_{snd\ (cms_1\ !\ k)}{}^{l}$ (*subst* σ′$_k$ (*snd* (*mems* ! *k*)))
apply −
apply(*rule mm-equiv-low-eq*[**where** *c*$_1$=*fst* (*cms*$_1$! *k*) **and** *c*$_2$=*fst* (*cms*$_2$! *k*)])
using *bisim*$_k$
by (*metis prod.collapse*)

have *eq*: ∀ *x*. *dma* ((*subst* σ′ (*fst* (*mems* ! *i*))) [‖$_1$ *A*]) *x* = *Low* ∧ (*x* ∈ (*snd* (*cms*$_1$! *i*)) *AsmNoReadOrWrite* ⟶ *x* ∈ *C*) ⟶
(*subst* σ′ (*fst* (*mems* ! *i*))) [‖$_1$ *A*] *x* = (*subst* σ′ (*snd* (*mems* ! *i*))) [‖$_2$ *A*] *x*

apply(*clarsimp simp*: *adapt-eq dma-eq*)
apply(*case-tac x* ∈ *dom* σ)
apply(*force simp*: *subst-def*)
apply(*simp add*: *subst-not-in-dom*)
apply(*simp add*: *domσ*)
apply(*clarsimp simp*: *differing-vars-lists-def differing-vars-def*)

apply(*case-tac* $i = k$)
apply(*simp add*: ‹$i \neq k$›)
apply(*erule mems'-i-cases*)
apply(*rule* ‹$i <$ *length* cms_1'›[*simplified len-unchanged*])
apply *force*
apply *fastforce*
apply *clarsimp*
apply *clarsimp*

apply(*case-tac* $x \in$ *differing-vars-lists* mem_1 mem_2 *mems* i)
apply(*force simp*: *differing-vars-lists-def differing-vars-def*)

apply(*insert low-mds-eq*)[*1*]
apply(*simp add*: *low-mds-eq-def*)
apply(*drule-tac* x=x **in** *spec*)

apply(*subst* (*asm*) *makes-compatible-dma-eq*)
apply(*rule compat*)
apply(*rule* ‹$i <$ *length* cms_1›)
apply(*rule* ‹*dom* σ' = *differing-vars-lists* mem_1 mem_2 *mems* i›)
apply *simp*
apply(*subgoal-tac* $x \notin$ *dom* σ')
apply(*simp add*: *subst-not-in-dom*)
apply *force*
apply(*simp add*: ‹*dom* σ' = *differing-vars-lists* mem_1 mem_2 *mems* i›)+
done
from *reclas eq*
show $(\forall x.\ dma\ ((subst\ \sigma'\ (fst\ (mems\ !\ i)))\ [\|_1\ A])\ x \neq dma\ (subst\ \sigma'\ (fst\ (mems\ !\ i)))\ x \longrightarrow$
$\neg$ *var-asm-not-written* $(snd\ (cms_1\ !\ i))\ x) \wedge$
$(\forall x.\ dma\ ((subst\ \sigma'\ (fst\ (mems\ !\ i)))\ [\|_1\ A])\ x = Low \wedge (x \in snd\ (cms_1\ !\ i)\ AsmNoReadOrWrite \longrightarrow x \in \mathcal{C}) \longrightarrow$
$(subst\ \sigma'\ (fst\ (mems\ !\ i)))\ [\|_1\ A]\ x = (subst\ \sigma'\ (snd\ (mems\ !\ i)))\ [\|_2\ A]\ x)$
by *blast*
qed

have $snd\ (cms_1\ !\ i) = snd\ (cms_2\ !\ i)$
by (*metis* ‹$i <$ *length* cms_1› *equal-size modes-eq nth-map*)

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

thus *?thesis* **using** *adapt-eq*
by (*metis* ‹$i \neq k$› *b* cms_2'*-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 mem1' x = High ∨ x ∈ C
    show x ∉ ?X' i
    proof (cases x ∈ C)
      assume x ∈ C
      thus ?thesis by(metis not-control i-le)
    next
      assume x ∉ C
        hence mem-eq: mem1' x = mem2' x ∨ dma mem1' x = High by(metis
mem-eq')
      thus ?thesis
      proof (cases i = k)
        assume i = k
        thus x ∉ ?X' i
          apply (cases x ∉ ?X k)
           apply (metis differing-vars-neg-intro mems'-k-1 mems'-k-2)
       by(metis compat-different[OF compat] b mem-eq Sec.distinct(1) 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 mem1' 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
            dma mem1 x = Low ∨ dma mem1' x = High
            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 mem1' 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 have mem₁' x ≠ mem₂' x ∧ dma mem₁ x = Low
            by (metis b compat compat-different i-le length-list-update)
            with mem-eq have clases: dma mem₁ x = Low ∧ dma mem₁' x =
High by auto
           have fst (mems' ! i) x = mem₁' x ∧ snd (mems' ! i) x = mem₂' x
            apply(rule mems'-i-5)
                apply(rule ‹i ≠ k›)
               using i-le len-unchanged apply(simp)
              apply(simp add: case3)+
             apply(simp add: clases)+
            done
           hence x ∉ ?X' i by (metis differing-vars-neg-intro)
           with ‹x ∈ ?X' i› show ?thesis by blast
          qed
          with ‹mem₁' x = mem₂' x ∨ dma mem₁' x = High› show x ∉ ?X' i
           by auto
         next
          assume case4: mem₁ x = mem₁' x mem₂ x = mem₂' x
                dma mem₁ x = High dma mem₁' x = Low
                fst (mems' ! i) x = mem₁ x snd (mems' ! i) x = mem₁ x
          with mem-eq have mem₁' x = mem₂' x by auto
              with case4 show ?thesis by(auto simp: differing-vars-def differ-
ing-vars-lists-def)
         next
          assume fst (mems' ! i) x = mem₁' x
                 snd (mems' ! i) x = mem₂' x thus ?thesis by(metis differ-
ing-vars-neg-intro)
         qed
        qed
      qed
    next
     { fix x
      have ∃ i < length cms₁. x ∉ ?X' i
       proof (cases mem₁ x ≠ mem₁' x ∨ mem₂ x ≠ mem₂' x ∨ dma mem₁' x ≠
dma mem₁ x)
        assume var-changed: mem₁ x ≠ mem₁' x ∨ mem₂ x ≠ mem₂' x ∨ dma
mem₁' x ≠ dma mem₁ 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 ∨ dma mem₁' x ≠
dma mem₁ x)
          hence assms: mem₁ x = mem₁' x mem₂ x = mem₂' x dma mem₁' x =
dma mem₁ 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
          from i-prop have x ∉ ?X i by simp
          assume i ≠ k
          hence x ∉ ?X' i

            apply −
            apply(rule mems'-i-cases)
                 apply assumption
                apply(simp add: i-prop)
               apply(simp add: assms)+
             using ‹x ∉ ?X i› differing-vars-neg
             using ‹¬ (mem1 x ≠ mem1' x ∨ mem2 x ≠ mem2' x ∨ dma mem1' x
≠ dma mem1 x)› differing-vars-elim apply auto[1]
            by(metis 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 mem1 x = Low
```

assumes *x-readable*: $x \in \mathcal{C} \vee (\forall\ i < \mathit{length}\ \mathit{cms}_1.\ x \notin \mathit{snd}\ (\mathit{cms}_1\ !\ i)\ \mathit{AsmNoReadOrWrite})$
shows $\mathit{mem}_1\ x = \mathit{mem}_2\ x$
proof −
let $?X = \lambda\ i.\ \mathit{differing\text{-}vars\text{-}lists}\ \mathit{mem}_1\ \mathit{mem}_2\ \mathit{mems}\ i$
from *compat* **have** $(\mathit{length}\ \mathit{cms}_1 = 0 \wedge \mathit{mem}_1 =^l \mathit{mem}_2) \vee (\forall\ x.\ \exists\ j.\ j < \mathit{length}\ \mathit{cms}_1 \wedge x \notin ?X\ j)$
by *auto*
thus $\mathit{mem}_1\ x = \mathit{mem}_2\ x$
proof
assume $\mathit{length}\ \mathit{cms}_1 = 0 \wedge \mathit{mem}_1 =^l \mathit{mem}_2$
with *x-low* **show** *?thesis*
by (*simp add*: *low-eq-def*)
next
assume $\forall\ x.\ \exists\ j.\ j < \mathit{length}\ \mathit{cms}_1 \wedge x \notin ?X\ j$
then obtain j **where** *j-prop*: $j < \mathit{length}\ \mathit{cms}_1 \wedge x \notin ?X\ j$
by *auto*
let $?\mathit{mems}_1 j = \mathit{fst}\ (\mathit{mems}\ !\ j)$ **and**
$?\mathit{mems}_2 j = \mathit{snd}\ (\mathit{mems}\ !\ j)$

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

from *compat x-low makes-compatible-dma-eq j-prop* ‹$\mathit{dom}\ \sigma = ?X\ j$›
have *x-low*: $\mathit{dma}\ (?\mathit{mems}_1 j\ [\mapsto \sigma])\ x = \mathit{Low}$
by *metis*

from *domσ compat* **and** *j-prop* **have** $(\mathit{cms}_1\ !\ j,\ ?\mathit{mems}_1 j\ [\mapsto \sigma]) \approx (\mathit{cms}_2\ !\ j,\ ?\mathit{mems}_2 j\ [\mapsto \sigma])$
by *auto*

moreover
have $\mathit{snd}\ (\mathit{cms}_1\ !\ j) = \mathit{snd}\ (\mathit{cms}_2\ !\ j)$
using *modes-eq*
by (*metis j-prop length-map nth-map*)

ultimately have $?\mathit{mems}_1 j\ [\mapsto \sigma] =_{\mathit{snd}\ (\mathit{cms}_1\ !\ j)}^{l}\ ?\mathit{mems}_2 j\ [\mapsto \sigma]$
using *modes-eq j-prop*
by (*metis mm-equiv-low-eq prod.collapse*)
hence $?\mathit{mems}_1 j\ x = ?\mathit{mems}_2 j\ x$
using *x-low x-readable j-prop* ‹$\mathit{dom}\ \sigma = ?X\ j$›
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*: $\langle c, mds, mem\rangle \rightsquigarrow \langle c', mds', mem'\rangle$
shows *loc-reach* $\langle c', mds', mem'\rangle \subseteq$ *loc-reach* $\langle c, mds, mem\rangle$
proof (*clarify*)
fix c'' mds'' mem''
from *eval* **have** $\langle c', mds', mem'\rangle \in$ *loc-reach* $\langle c, mds, mem\rangle$
by (*metis loc-reach.refl loc-reach.step surjective-pairing*)
assume $\langle c'', mds'', mem''\rangle \in$ *loc-reach* $\langle c', mds', mem'\rangle$
thus $\langle c'', mds'', mem''\rangle \in$ *loc-reach* $\langle c, mds, mem\rangle$
apply *induct*
apply (*metis* ‹$\langle c', mds', mem'\rangle \in$ *loc-reach* $\langle c, mds, mem\rangle$› *surjective-pairing*)
apply (*metis loc-reach.step*)
by (*metis loc-reach.mem-diff*)
qed

lemma *locally-sound-modes-invariant*:
assumes *sound-modes*: *locally-sound-mode-use* $\langle c, mds, mem\rangle$
assumes *eval*: $\langle c, mds, mem\rangle \rightsquigarrow \langle c', mds', mem'\rangle$
shows *locally-sound-mode-use* $\langle c', mds', mem'\rangle$
proof −
from *eval* **have** $\langle c', mds', mem'\rangle \in$ *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 *meval-sched-one*:
$(cms, mem) \rightsquigarrow_k (cms', mem') \Longrightarrow$
$(cms, mem) \rightarrow_{[k]} (cms', mem')$
apply(*simp*)
done

lemma *meval-sched-ConsI*:
$(cms, mem) \rightsquigarrow_k (cms', mem') \Longrightarrow$
$(cms', mem') \rightarrow_{sched} (cms'', mem'') \Longrightarrow$
$(cms, mem) \rightarrow_{(k\#sched)} (cms'', mem'')$
by *fastforce*

lemma *reachable-modes-subset*:
assumes *eval*: $(cms, mem) \rightsquigarrow_k (cms', mem')$
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 *assms*
unfolding *reachable-mode-states-def*
by (*blast intro*: *meval-sched-ConsI*)
qed

lemma *globally-sound-modes-invariant*:
assumes *globally-sound*: *globally-sound-mode-use* (*cms*, *mem*)
assumes *eval*: (*cms*, *mem*) $\rightsquigarrow_k$ (*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*. *var-asm-not-written mds x* $\longrightarrow$ *mem*$_1$ *x* = *mem*$_2$ *x* $\wedge$ *dma mem*$_1$ *x* = *dma 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*: $\forall$ *x*. *var-asm-not-written mds x* $\longrightarrow$ *mem' x* = *mem x* $\wedge$ *dma mem' x* = *dma mem x*
shows *loc-reach* $\langle c, mds, mem \rangle$ = *loc-reach* $\langle c, mds, mem' \rangle$
using *assms loc-reach-mem-diff-subset*
by (*auto*, *metis*)

lemma *sound-modes-invariant*:
assumes *sound-modes*: *sound-mode-use* (*cms*, *mem*)
assumes *eval*: (*cms*, *mem*) $\rightsquigarrow_k$ (*cms'*, *mem'*)
shows *sound-mode-use* (*cms'*, *mem'*)
proof −

from *sound-modes* **and** *eval* **have** *globally-sound-mode-use* (*cms'*, *mem'*)
by (*metis globally-sound-modes-invariant sound-mode-use.simps*)
moreover
from *sound-modes* **have** *loc-sound*: $\forall\ i < length\ cms.$ *locally-sound-mode-use* (*cms* ! *i*, *mem*)
unfolding *sound-mode-use-def*
by *simp* (*metis list-all-length*)
from *eval* **obtain** *k* cms_k' **where**
ev: (*cms* ! *k*, *mem*) $\rightsquigarrow$ (cms_k', *mem'*) $\wedge$ *k* < *length cms* $\wedge$ *cms'* = *cms* [*k* := cms_k']
by (*metis meval-elim*)
hence *length cms* = *length cms'*
by *auto*
have $\bigwedge$ *i*. *i* < *length cms'* $\Longrightarrow$ *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*. *var-asm-not-written* (*snd* (*cms* ! *i*)) *x* $\Longrightarrow$ *x* ∈ *snd* (*cms* ! *k*) *GuarNoWrite* ∨ *x* ∈ *snd* (*cms* ! *k*) *GuarNoReadOrWrite*
unfolding *compatible-modes-def*
by (*metis* (*no-types*) ‹*i* ≠ *k*› ‹*length cms* = *length cms'*› *ev i-le length-map nth-map var-asm-not-written-def*)
hence $\bigwedge$ *x*. *var-asm-not-written* (*snd* (*cms* ! *i*)) *x* $\longrightarrow$ *doesnt-modify* (*fst* (*cms* ! *k*)) *x*
using *ev loc-sound*
unfolding *locally-sound-mode-use-def*
by (*metis loc-reach.refl surjective-pairing doesnt-read-or-modify-doesnt-modify*)
with *ev* **have** $\bigwedge$ *x*. *var-asm-not-written* (*snd* (*cms* ! *i*)) *x* $\Longrightarrow$ *mem x* = *mem' x* ∧ *dma mem x* = *dma mem' x*
unfolding *doesnt-modify-def* **by** (*metis prod.collapse*)
with *loc-reach-mem-diff-eq* **have** *loc-reach* (*cms* ! *i*, *mem'*) = *loc-reach* (*cms* ! *i*, *mem*)
apply −
by(*case-tac cms* ! *i*, *simp*)
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 *app-Cons-rewrite*:
ns @ (*a* # *ms*) = ((*ns* @ [*a*]) @ *ms*)
apply *simp*
done

lemma *meval-sched-app-iff*:
(cms_1, mem_1) $\rightarrow_{ns@ms}$ (cms_1', mem_1') =
($\exists\, cms_1''\ mem_1''$. ($cms_1$, mem_1) $\rightarrow_{ns}$ (cms_1'', mem_1'') $\wedge$ (cms_1'', mem_1'') $\rightarrow_{ms}$ (cms_1', mem_1'))
apply(*induct ns arbitrary*: *ms* cms_1 mem_1)
apply *simp*
apply *force*
done

lemmas *meval-sched-appD* = *meval-sched-app-iff*[*THEN iffD1*]
lemmas *meval-sched-appI* = *meval-sched-app-iff*[*THEN iffD2*, *OF exI*, *OF exI*]

lemma *meval-sched-snocD*:
(cms_1, mem_1) $\rightarrow_{ns@[n]}$ (cms_1', mem_1') $\Longrightarrow$
$\exists\, cms_1''\ mem_1''$. ($cms_1$, mem_1) $\rightarrow_{ns}$ (cms_1'', mem_1'') $\wedge$ (cms_1'', mem_1'') $\rightsquigarrow_{n}$ (cms_1', mem_1')
apply(*fastforce dest*: *meval-sched-appD*)
done

lemma *meval-sched-snocI*:
(cms_1, mem_1) $\rightarrow_{ns}$ (cms_1'', mem_1'') $\wedge$ (cms_1'', mem_1'') $\rightsquigarrow_{n}$ (cms_1', mem_1') $\Longrightarrow$
(cms_1, mem_1) $\rightarrow_{ns@[n]}$ (cms_1', mem_1')
apply(*fastforce intro*: *meval-sched-appI*)
done

lemma *makes-compatible-eval-sched*:
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) $\rightarrow_{sched}$ (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' $\wedge$

$(cms_2, mem_2) \rightarrow_{sched} (cms_2', mem_2') \wedge$
$makes\text{-}compatible\ (cms_1', mem_1')\ (cms_2', mem_2')\ mems'$

proof −

from *eval* **show** *?thesis*
proof (*induct sched arbitrary*: $cms_1'\ mem_1'$ *rule*: *rev-induct*)
case *Nil*
hence $cms_1' = cms_1 \wedge mem_1' = mem_1$
by (*simp add*: *Pair-inject meval-sched.simps*(*1*))
thus *?case*
by (*metis compat meval-sched.simps*(*1*) *modes-eq sound-modes*)
next
case (*snoc n ns*)
then obtain $cms_1''\ mem_1''$ **where** *eval''*:
$(cms_1, mem_1) \rightarrow_{ns} (cms_1'', mem_1'') \wedge (cms_1'', mem_1'') \rightsquigarrow_n (cms_1', mem_1')$
by (*metis meval-sched-snocD prod-cases3 snd-conv*)
hence $(cms_1'', mem_1'') \rightsquigarrow_n (cms_1', mem_1')$ **..**
moreover
from *eval''* **obtain** $cms_2''\ mem_2''\ mems''$ **where** *IH*:
$sound\text{-}mode\text{-}use\ (cms_1'', mem_1'') \wedge$
$sound\text{-}mode\text{-}use\ (cms_2'', mem_2'') \wedge$
$map\ snd\ cms_1'' = map\ snd\ cms_2'' \wedge$
$(cms_2, mem_2) \rightarrow_{ns} (cms_2'', mem_2'') \wedge$
$makes\text{-}compatible\ (cms_1'', mem_1'')\ (cms_2'', mem_2'')\ mems''$
using *snoc*
by *metis*
ultimately obtain $cms_2'\ mem_2'\ mems'$ **where**
$map\ snd\ cms_1' = map\ snd\ cms_2' \wedge$
$(cms_2'', mem_2'') \rightsquigarrow_n (cms_2', mem_2') \wedge$
$makes\text{-}compatible\ (cms_1', mem_1')\ (cms_2', mem_2')\ mems'$
using *makes-compatible-invariant*
by *blast*
thus *?case*
using *IH eval'' meval-sched-snocI sound-modes-invariant*
by *metis*
qed
qed

lemma *differing-vars-initially-empty*:
$i < n \Longrightarrow x \notin differing\text{-}vars\text{-}lists\ mem_1\ mem_2\ (zip\ (replicate\ n\ mem_1)\ (replicate\ n\ mem_2))\ i$
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 $makes\text{-}compatible\ (cmds, mem_1)$
$(cmds, mem_2)$

(*replicate* (*length cmds*) (*mem*$_1$, *mem*$_2$))
proof −
let *?n* = *length cmds*
let *?mems* = *replicate ?n* (*mem*$_1$, *mem*$_2$) **and**
?mdss = *map snd cmds*
let *?X* = *differing-vars-lists mem*$_1$ *mem*$_2$ *?mems*
have *diff-empty*: ∀ *i* < *?n*. *?X i* = {}
by (*metis differing-vars-initially-empty ex-in-conv min.idem zip-replicate*)

show *?thesis*
proof
show *length cmds* = *length cmds* ∧ *length cmds* = *length ?mems*
by *auto*
next
fix *i* **and** σ :: *'Var* ⇒ *'Val option*
let *?mems*$_1$*i* = *fst* (*?mems* ! *i*) **and** *?mems*$_2$*i* = *snd* (*?mems* ! *i*)
let *?mdss*$_i$ = *?mdss* ! *i*
assume *i*: *i* < *length cmds*
assume *domσ*: *dom* σ =
differing-vars-lists mem$_1$ *mem*$_2$
(*replicate* (*length cmds*) (*mem*$_1$, *mem*$_2$)) *i*
from *i* **have** *?mems*$_1$*i* = *mem*$_1$ *?mems*$_2$*i* = *mem*$_2$
by *auto*

with *domσ* **have** [*simp*]: *dom* σ = {} **by**(*auto simp*: *differing-vars-lists-def differing-vars-def i*)

from *i coms-secure* **have** *com-sifum-secure* (*cmds* ! *i*)
using *coms-secure*
by (*metis length-map length-replicate list-all-length map-snd-zip*)
with *i* **have** ⋀ *mem*$_1$ *mem*$_2$. *mem*$_1$ =$_{?mdss_i}{}^{l}$ *mem*$_2$ ⟹
(*cmds* ! *i*, *mem*$_1$) ≈ (*cmds* ! *i*, *mem*$_2$)
using *com-sifum-secure-def low-indistinguishable-def*
by *auto*

moreover
have ⋀*x*. σ *x* = *None* **using** ‹*dom* σ = {}›
by (*metis domIff empty-iff*)
hence [*simp*]: ⋀ *mem*. *mem* [↦ σ] = *mem*
by(*simp add*: *subst-def*)

from *i* **have** *?mems*$_1$*i* = *mem*$_1$ *?mems*$_2$*i* = *mem*$_2$
by *auto*
with *low-eq* **have** *?mems*$_1$*i* [↦ σ] =$_{?mdss_i}{}^{l}$ *?mems*$_2$*i* [↦ σ]
by (*auto simp*: *low-mds-eq-def low-eq-def*)

ultimately show (*cmds* ! *i*, *?mems*$_1$*i* [↦ σ]) ≈ (*cmds* ! *i*, *?mems*$_2$*i* [↦ σ])
by *simp*
next

fix i x
assume $i < length\ cmds$
with *diff-empty* **show** $x \notin ?X\ i$ **by** *auto*
next
show ($length\ cmds = 0 \wedge mem_1 =^l mem_2$) $\vee$ ($\forall\ x.\ \exists\ i < length\ cmds.\ x \notin ?X\ i$)
using *diff-empty*
by (*metis bot-less bot-nat-def empty-iff length-zip low-eq min-0L*)
qed
qed

theorem *sifum-compositionality-cont*:
assumes *com-secure*: *list-all com-sifum-secure cmds*
assumes *sound-modes*: $\forall\ mem.\ INIT\ mem \longrightarrow sound\text{-}mode\text{-}use\ (cmds,\ mem)$
shows *prog-sifum-secure-cont cmds*
unfolding *prog-sifum-secure-cont-def*
using *assms*
proof (*clarify*)
fix $mem_1\ mem_2 :: 'Var \Rightarrow 'Val$
fix $sched\ cms_1'\ mem_1'$
let $?n = length\ cmds$
let $?mems = zip\ (replicate\ ?n\ mem_1)\ (replicate\ ?n\ mem_2)$
assume $INIT_1$: $INIT\ mem_1$ **and** $INIT_2$: $INIT\ mem_2$
assume *low-eq*: $mem_1 =^l mem_2$
with *com-secure* **have** *compat*:
makes-compatible ($cmds$, mem_1) ($cmds$, mem_2) *?mems*
by (*metis compatible-refl fst-conv length-replicate map-replicate snd-conv zip-eq-conv* $INIT_1$ $INIT_2$)

also assume *eval*: $(cmds,\ mem_1) \rightarrow_{sched} (cms_1',\ mem_1')$

ultimately obtain $cms_2'\ mem_2'\ mems'$
where *p*: $map\ snd\ cms_1' = map\ snd\ cms_2' \wedge$
$(cmds,\ mem_2) \rightarrow_{sched} (cms_2',\ mem_2') \wedge$
$makes\text{-}compatible\ (cms_1',\ mem_1')\ (cms_2',\ mem_2')\ mems'$
using *sound-modes makes-compatible-eval-sched* $INIT_1$ $INIT_2$
by *blast*

thus $\exists\ cms_2'\ mem_2'.\ (cms,\ mem_2) \rightarrow_{sched} (cms_2',\ mem_2') \wedge$
$map\ snd\ cms_1' = map\ snd\ cms_2' \wedge$
$length\ cms_2' = length\ cms_1' \wedge$
($\forall\ x.\ dma\ mem_1'\ x = Low \wedge (x \in \mathcal{C} \vee (\forall\ i < length\ cms_1'.\ x \notin snd\ (cms_1'\ !\ i)\ AsmNoReadOrWrite)) \longrightarrow mem_1'\ x = mem_2'\ x$)
using *p compat-low-eq*
by (*metis length-map*)
qed

end

end

4 Language for Instantiating the SIFUM-Security Property

theory *Language*
imports *Preliminaries*
begin

4.1 Syntax

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

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

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

locale *sifum-lang-no-dma* =
  fixes $eval_A$ :: (*'Var*, *'Val*) *Mem* $\Rightarrow$ *'AExp* $\Rightarrow$ *'Val*
  fixes $eval_B$ :: (*'Var*, *'Val*) *Mem* $\Rightarrow$ *'BExp* $\Rightarrow$ *bool*
  fixes *aexp-vars* :: *'AExp* $\Rightarrow$ *'Var set*
  fixes *bexp-vars* :: *'BExp* $\Rightarrow$ *'Var set*
  assumes *Var-finite* : *finite* {(*x* :: *'Var*). *True*}
  assumes *eval-vars-det*$_A$: $[\![\forall\ x \in aexp\text{-}vars\ e.\ mem_1\ x = mem_2\ x\ ]\!] \Longrightarrow eval_A\ mem_1\ e = eval_A\ mem_2\ e$
  assumes *eval-vars-det*$_B$: $[\![\forall\ x \in bexp\text{-}vars\ b.\ mem_1\ x = mem_2\ x\ ]\!] \Longrightarrow eval_B\ mem_1\ b = eval_B\ mem_2\ b$

locale *sifum-lang* = *sifum-lang-no-dma* $eval_A$ $eval_B$ *aexp-vars bexp-vars*
  for $eval_A$ :: (*'Var*, *'Val*) *Mem* $\Rightarrow$ *'AExp* $\Rightarrow$ *'Val*
  and $eval_B$ :: (*'Var*, *'Val*) *Mem* $\Rightarrow$ *'BExp* $\Rightarrow$ *bool*
  and *aexp-vars* :: *'AExp* $\Rightarrow$ *'Var set*
  and *bexp-vars* :: *'BExp* $\Rightarrow$ *'Var set*+
  fixes *dma* :: *'Var* $\Rightarrow$ *Sec*

context *sifum-lang-no-dma*
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*)

notation (*Rule* **output**)
Await (*await* - *do* - *done*)

abbreviation $conf_w$*-abv* :: ('*Var*, '*AExp*, '*BExp*) *Stmt* ⇒
'*Var Mds* ⇒ ('*Var*, '*Val*) *Mem* ⇒ (-,-,-) *LocalConf*
(⟨-, -, -⟩$_w$ [*0*, *120*, *120*] *100*)
where
⟨ *c*, *mds*, *mem* ⟩$_w$ ≡ ((*c*, *mds*), *mem*)

4.2 Semantics

primrec *update-modes* :: '*Var ModeUpd* ⇒ '*Var Mds* ⇒ '*Var Mds*
where
update-modes (*Acq x m*) *mds* = *mds* (*m* := *insert x* (*mds m*)) |
update-modes (*Rel x m*) *mds* = *mds* (*m* := {*y*. *y* ∈ *mds m* ∧ *y* ≠ *x*})

fun *updated-var* :: '*Var ModeUpd* ⇒ '*Var*
where
updated-var (*Acq x* -) = *x* |
updated-var (*Rel x* -) = *x*

fun *updated-mode* :: '*Var ModeUpd* ⇒ *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** ⇝$_s$ *60*)
where
c ⇝$_s$ *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: $[\![eval_B\ mem\ b]\!] \Longrightarrow ((If\ b\ t\ e, mem), (t, mem)) \in eval_w\text{-}simple$ |
if-false: $[\![\neg\ eval_B\ mem\ b]\!] \Longrightarrow ((If\ b\ t\ e, mem), (e, mem)) \in eval_w\text{-}simple$ |
while: $((While\ b\ c, mem), (If\ b\ (c\ ;;\ While\ b\ c)\ Stop, mem)) \in eval_w\text{-}simple$

lemma *cond*:
$((If\ b\ t\ e, mem), (if\ eval_B\ mem\ b\ then\ t\ else\ e, mem)) \in eval_w\text{-}simple$
apply(*case-tac* $eval_B\ mem\ b$)
apply(*auto intro*: $eval_w\text{-}simple.intros$)
done

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*)

lemma *rtrancl-mono-proof*[*mono*]:
$(\bigwedge a\ b.\ x\ a\ b \longrightarrow y\ a\ b) \Longrightarrow rtranclp\ x\ a\ b \longrightarrow rtranclp\ y\ a\ b$
apply (*rule impI*, *rotate-tac*, *induct rule*: *rtranclp.induct*)
apply *simp-all*
apply (*metis rtranclp.intros*)
done

lemma *trancl-mono-proof*[*mono*]:
$(\bigwedge a\ b.\ x\ a\ b \longrightarrow y\ a\ b) \Longrightarrow tranclp\ x\ a\ b \longrightarrow tranclp\ y\ a\ b$
apply (*rule impI*, *rotate-tac*, *induct rule*: *tranclp.induct*)
apply *simp-all*
apply *blast*
by *fastforce*

inductive *no-await* :: ('*Var*, '*AExp*, '*BExp*) *Stmt* ⇒ *bool* **where**
no-await $(x \leftarrow e)$ |
no-await c1 $\Longrightarrow$ *no-await c2* $\Longrightarrow$ *no-await* $(c1\ ;;\ c2)$ |
no-await c1 $\Longrightarrow$ *no-await c2* $\Longrightarrow$ *no-await* $(If\ b\ c1\ c2)$ |
no-await c $\Longrightarrow$ *no-await* $(While\ b\ c)$ |
no-await Skip |
no-await Stop |
no-await c $\Longrightarrow$ *no-await* $(c@[m])$

inductive *is-final* :: ('*Var*, '*AExp*, '*BExp*) *Stmt* ⇒ *bool* **where**
is-final Stop |
is-final c $\Longrightarrow$ *is-final* $(c@[m])$

inductive-set $eval_w$ :: (('*Var*, '*AExp*, '*BExp*) *Stmt*, '*Var*, '*Val*) *LocalConf rel*
and $eval_w\text{-}abv$ :: (('*Var*, '*AExp*, '*BExp*) *Stmt*, '*Var*, '*Val*) *LocalConf* ⇒

((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf* ⇒ *bool*
(**infixr** $\rightsquigarrow_w$ *60*)
where
$c \rightsquigarrow_w c' \equiv (c, c') \in eval_w$ |
unannotated: ⟦ $(c, mem) \rightsquigarrow_s (c', mem')$ ⟧
⟹ $(\langle$*cxt-to-stmt E c*, *mds*, *mem*$\rangle_w, \langle$*cxt-to-stmt E c'*, *mds*, *mem'*$\rangle_w) \in eval_w$ |
seq: ⟦ $\langle c_1$, *mds*, *mem*$\rangle_w \rightsquigarrow_w \langle c_1'$, *mds'*, *mem'*$\rangle_w$ ⟧ ⟹ $(\langle(c_1$;; $c_2)$, *mds*, *mem*$\rangle_w$, $\langle(c_1'$;; $c_2)$, *mds'*, *mem'*$\rangle_w) \in eval_w$ |
decl: ⟦ $\langle c$, *update-modes mu mds*, *mem*$\rangle_w \rightsquigarrow_w \langle c'$, *mds'*, *mem'*$\rangle_w$ ⟧ ⟹
($\langle$*cxt-to-stmt E* (*ModeDecl c mu*), *mds*, *mem*$\rangle_w$, $\langle$*cxt-to-stmt E c'*, *mds'*, *mem'*$\rangle_w) \in eval_w$ |

await: ⟦$eval_B$ *mem b*; *no-await* c_1;
($\langle c_1$, *mds*, *mem*$\rangle_w$, $\langle c_2$, *mds'*, *mem'*$\rangle_w) \in {eval_w}^+$;
is-final c_2⟧ ⟹
($\langle$*Await b* c_1, *mds*, *mem*$\rangle_w$, $\langle c_2$, *mds'*, *mem'*$\rangle_w) \in eval_w$

abbreviation $eval_w$*-plus* ::
((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf* ⇒
((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf* ⇒ *bool* (- ${\rightsquigarrow_w}^+$ -) **where**
ctx ${\rightsquigarrow_w}^+$ *ctx'* ≡ (*ctx*, *ctx'*) ∈ ${eval_w}^+$

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* ← *e*, *mem*), (*c'*, *mem'*)) ∈ $eval_w$*-simple*
inductive-cases *skip-elim'* [*elim*]: (*Skip*, *mem*) $\rightsquigarrow_s$ (*c'*, *mem'*)

lemma *cxt-inv*:
⟦ *cxt-to-stmt E c* = *c'* ; ⋀ *p q*. *c'* ≠ *Seq p q* ⟧ ⟹ *E* = [] ∧ *c'* = *c*
by (*metis cxt-to-stmt.simps*(*1*) *cxt-to-stmt.simps*(*2*) *neq-Nil-conv*)

lemma *cxt-inv-assign*:
⟦ *cxt-to-stmt E c* = *x* ← *e* ⟧ ⟹ *c* = *x* ← *e* ∧ *E* = []
by (*metis Stmt.simps*(*11*) *cxt-inv*)

lemma *cxt-inv-skip*:
⟦ *cxt-to-stmt E c* = *Skip* ⟧ ⟹ *c* = *Skip* ∧ *E* = []
by (*metis Stmt.simps*(*23*) *cxt-inv*)

lemma *cxt-inv-stop*:
cxt-to-stmt E c = *Stop* ⟹ *c* = *Stop* ∧ *E* = []
by (*metis Stmt.simps*(*49*) *cxt-inv*)

lemma *cxt-inv-if*:

cxt-to-stmt E c = If e p q ⟹ c = If e p q ∧ E = []
by (*metis Stmt.simps(43) cxt-inv*)

lemma *ctx-inv-anno*:
cxt-to-stmt E c = c'@[mu] ⟹ c = c'@[mu] ∧ E = []
using *cxt-inv* **by** *blast*

lemma *cxt-inv-await*:
cxt-to-stmt E c = Await e p ⟹ c = Await e p ∧ E = []
by (*metis Stmt.simps(47) cxt-inv*)

lemma *cxt-inv-while*:
cxt-to-stmt E c = While e p ⟹ c = While e p ∧ E = []
by (*metis Stmt.simps(45) cxt-inv*)

lemma *skip-elim* [*elim*]:
⟨Skip, mds, mem⟩$_w$ ⇝$_w$ ⟨c', mds', mem'⟩$_w$ ⟹ c' = Stop ∧ mds = mds' ∧ mem = mem'
apply (*erule eval-elim*)
apply (*metis* (*lifting*) *cxt-inv-skip cxt-to-stmt.simps(1) skip-elim'*)
apply (*metis Stmt.simps(24)*)
apply (*metis Stmt.simps(21) cxt-inv-skip*)
by *simp*

lemma *assign-elim* [*elim*]:
⟨x ← e, mds, mem⟩$_w$ ⇝$_w$ ⟨c', mds', mem'⟩$_w$ ⟹ c' = Stop ∧ mds = mds' ∧ mem' = mem (x := eval$_A$ mem e)
apply (*erule eval-elim*)
apply (*rename-tac c c'a E*)
apply (*subgoal-tac c = x ← e ∧ E = []*)
apply *force*
apply *auto*
apply (*metis cxt-inv-assign*)
apply (*metis cxt-inv-assign*)
apply (*metis Stmt.simps(9) cxt-inv-assign*)
apply (*metis Stmt.simps(9) cxt-inv-assign*)
by (*metis Stmt.simps(9) cxt-inv-assign*)

inductive-cases *if-elim'* [*elim*!]: *(If b p q, mem) ⇝$_s$ (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(43)*)
apply (*metis Stmt.simps(35) cxt-inv-if*)

by *simp*

inductive-cases *await-elim'* [*elim*!]: $\langle Await\ b\ p,\ mds,\ mem\rangle_w \rightsquigarrow_w \langle c',mds',\ mem'\rangle_w$

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

lemma *while-elim* [*elim*]:
$[\![\langle While\ e\ c,\ mds,\ mem\rangle_w \rightsquigarrow_w \langle c',\ mds',\ mem'\rangle_w]\!] \Longrightarrow c' = If\ e\ (c\ ;;\ While\ e\ c)\ Stop \wedge mds' = mds \wedge mem' = mem$
apply (*erule eval-elim*)
apply (*metis* (*no-types*) *cxt-inv-while cxt-to-stmt.simps*(*1*) *while-elim'*)
apply (*metis Stmt.simps*(*45*))
apply (*metis* (*lifting*) *Stmt.simps*(*37*) *cxt-inv-while*)
by *simp*

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

lemma *upd-elim* [*elim*]:
$\langle c@[upd],\ mds,\ mem\rangle_w \rightsquigarrow_w \langle c',\ mds',\ mem'\rangle_w \Longrightarrow \langle c,\ update\text{-}modes\ upd\ mds,\ mem\rangle_w \rightsquigarrow_w \langle c',\ mds',\ mem'\rangle_w$
apply (*erule eval-elim*)
apply (*metis* (*lifting*) *Stmt.simps*(*33*) *cxt-inv upd-elim'*)
apply (*metis Stmt.simps*(*34*))
apply (*metis* (*lifting*) *Stmt.simps*(*2*) *Stmt.simps*(*33*) *cxt-inv cxt-to-stmt.simps*(*1*))
by *simp*

lemma *cxt-seq-elim* [*elim*]:
$c_1\ ;;\ c_2 = cxt\text{-}to\text{-}stmt\ E\ c \Longrightarrow (E = [] \wedge c = c_1\ ;;\ c_2) \vee (\exists\ c'\ cs.\ E = c' \#\ cs \wedge c = c_1 \wedge c_2 = cxt\text{-}to\text{-}stmt\ cs\ c')$
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*]: $(c_1\ ;;\ c_2,\ mem) \rightsquigarrow_s (c',\ mem')$

lemma *stop-no-eval*: $\neg\ (\langle Stop,\ mds,\ mem\rangle_w \rightsquigarrow_w \langle c',\ mds',\ mem'\rangle_w)$
apply *auto*
apply (*erule eval-elim*)
apply (*metis cxt-inv-stop stop-no-eval'*)
apply (*metis Stmt.simps*(*49*))
apply (*metis Stmt.simps*(*41*) *cxt-inv-stop*)
by *simp*

lemma *seq-stop-elim* [*elim*]:
$\langle Stop\ ;;\ c,\ mds,\ mem\rangle_w \rightsquigarrow_w \langle c',\ mds',\ mem'\rangle_w \Longrightarrow c' = c \wedge mds' = mds \wedge mem' = mem$
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*)
apply (*metis Stmt.distinct*(*28*) *Stmt.distinct*(*36*) *cxt-seq-elim*)
by *simp*

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 ≠ *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 *blast*
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*(*27*) *cxt-stmt-seq cxt-to-stmt.simps*(*1*) *eval$_w$.decl*)
using *cxt-seq-elim* **by** *blast*

lemma *stop-cxt*: *Stop* = *cxt-to-stmt* E c ⟹ c = *Stop*
by (*metis Stmt.simps*(*50*) *cxt-to-stmt.simps*(*1*) *cxt-to-stmt.simps*(*2*) *neq-Nil-conv*)

lemmas *decl-eval$_w$* = *decl*[*OF unannotated*, *OF skip*, **where** *E*=[], *simplified*, **where** *E1*=[], *simplified*]
lemmas *seq-stop-eval$_w$* = *unannotated*[*OF seq-stop*, **where** *E*=[], *simplified*]
lemmas *assign-eval$_w$* = *unannotated*[*OF assign*, **where** *E*=[], *simplified*]
lemmas *if-eval$_w$* = *unannotated*[*OF cond*, **where** *E*=[], *simplified*]
lemmas *if-false-eval$_w$* = *unannotated*[*OF if-false*, **where** *E*=[], *simplified*]
lemmas *skip-eval$_w$* = *unannotated*[*OF skip*, **where** *E*=[], *simplified*]
lemmas *while-eval$_w$* = *unannotated*[*OF while*, **where** *E*=[], *simplified*]

lemma *decl-eval$_w$'*:
assumes *mem-unchanged*: *mem'* = *mem*
assumes *upd*: *mds'* = *update-modes upd mds*
shows (⟨*Skip*@[*upd*], *mds*, *mem*⟩$_w$, ⟨*Stop*, *mds'*, *mem'*⟩$_w$) ∈ *eval$_w$*
using *assms decl-eval$_w$*
by *auto*

lemma *assign-eval$_w$'*:
⟦*mds* = *mds'*; *mem'* = *mem*(*x* := *eval$_A$ mem e*)⟧ ⟹
⟨*x* ← *e*, *mds*, *mem*⟩$_w$ ⇝$_w$ ⟨*Stop*, *mds'*, *mem'*⟩$_w$

using *assign-eval*$_w$
by *simp*

lemma *seq-decl-elim*:
$\langle (Skip@[upd]) ;; c, mds, mem\rangle_w \rightsquigarrow_w \langle c', mds', mem'\rangle_w \Longrightarrow$
$c' = Stop ;; c \wedge mem' = mem \wedge mds' = update\text{-}modes\ upd\ mds$
apply(*drule seq-elim*, *simp*)
apply(*erule exE*, *clarsimp*)
apply(*drule upd-elim*)
apply(*drule skip-elim*, *clarsimp*)
done

lemma *seq-assign-elim*:
$\langle (x \leftarrow e) ;; c, mds, mem\rangle_w \rightsquigarrow_w \langle c', mds', mem'\rangle_w \Longrightarrow$
$c' = Stop ;; c \wedge mds' = mds \wedge mem' = mem(x := eval_A\ mem\ e)$
apply(*drule seq-elim*, *simp*)
apply(*erule exE*, *clarsimp*)
apply(*drule assign-elim*, *clarsimp*)
done

lemma *no-await-trans*:
$[\![no\text{-}await\ c; \langle c, mds, mem\rangle_w \rightsquigarrow_w \langle c', mds', mem'\rangle_w]\!] \Longrightarrow no\text{-}await\ c'$
apply (*induct arbitrary*: c' *mds rule*: *no-await.induct*)
using *assign-elim no-await.simps* **apply** *blast*
apply (*rename-tac c1 c2 c3 mds*)
apply (*case-tac c1 = Stop*)
apply (*simp*, *frule seq-stop-elim*, *clarsimp*)
using *seq-elim no-await.intros* **apply** *metis*
using *if-elim no-await.intros* **apply** *blast*
apply (*frule while-elim*, *clarsimp*)
apply (*rename-tac c b*)
apply (*subgoal-tac no-await* (*While b c*))
apply (*subgoal-tac no-await* (*c ;; While b c*))
using *no-await.intros* **apply** *blast*
using *no-await.intros* **apply** *blast*
using *no-await.intros* **apply** *blast*
using *no-await.intros skip-elim* **apply** *fast*
using *no-await.intros stop-no-eval* **apply** *fast*
using *no-await.intros upd-elim* **by** *fast*

lemma *no-await-no-await*[*elim*]: $[\![no\text{-}await\ c]\!] \Longrightarrow c \neq Await\ b\ c'$
using *no-await.cases Stmt.distinct* **by** *fast*

lemma *no-await-trancl-impl*:
$[\![ctx \rightsquigarrow_w{}^+\ ctx']\!] \Longrightarrow no\text{-}await\ (fst\ (fst\ ctx)) \longrightarrow no\text{-}await\ (fst\ (fst\ ctx'))$
apply (*erule trancl.induct*, *clarsimp*)
using *no-await-trans* **apply** *blast*

apply *clarsimp*
using *no-await-trans* **by** *blast*

lemma *no-await-trancl*:
⟦*ctx* $\rightsquigarrow_w{}^+$ *ctx'*; *no-await* (*fst* (*fst ctx*))⟧ ⟹ *no-await* (*fst* (*fst ctx'*))
using *no-await-trancl-impl* **by** *blast*

lemma *await-elim*:
⟦⟨*Await b* c_1, *mds*, *mem*⟩$_w$ $\rightsquigarrow_w$ ⟨c_2, *mds'*, *mem'*⟩$_w$⟧ ⟹
eval$_B$ *mem b* ∧ *no-await* c_1 ∧ *is-final* c_2 ∧
⟨c_1, *mds*, *mem*⟩$_w$ $\rightsquigarrow_w{}^+$ ⟨c_2, *mds'*, *mem'*⟩$_w$
apply (*erule eval*$_w$*.cases*; *clarsimp*)
apply (*subgoal-tac cxt-to-stmt E c* = *Await b* c_1)
apply (*drule cxt-inv-await*)
using *eval*$_w$*-simple.cases* **apply** *force*
apply *simp*
by (*metis Stmt.distinct*(*33*) *cxt-inv-await*)

end

end

5 Type System for Ensuring SIFUM-Security of Commands

theory *TypeSystem*
imports *Compositionality Language*
begin

5.1 Typing Rules

Types now depend on memories. To see why, consider an assignment in which some variable x for which we have a *AsmNoReadOrWrite* assumption is assigned the value in variable *input*, but where *input*'s classification depends on some control variable. Then the new type of x depends on memory. If we were to just take the upper bound of *input*'s classification, this would likely give us *High* as x's type, but that would prevent us from treating x as *Low* if we later learn *input*'s original classification.

Instead we need to make x's type explicitly depend on memory so later on, once we learn *input*'s classification, we can resolve x's type to a concrete security level.

We choose to deeply embed types as sets of boolean expressions. If any expression in the set evaluates to *True*, the type is *High*; otherwise it is *Low*.

type-synonym *'BExp Type* = *'BExp set*

We require Γ to track all stable (i.e. *AsmNoWrite* or *AsmNoReadOrWrite*),

non-$\mathcal{C}$ variables.

This differs from Mantel a bit. Mantel would exclude from Γ, variables whose classification (according to *dma*) is *Low* for which we have only an *AsmNoWrite* assumption.

We decouple the requirement for inclusion in Γ from a variable's classification so that we don't need to be updating Γ each time we alter a control variable. Even if we tried to keep Γ up-to-date in that case, we may not be able to precisely compute the new classification of each variable after the modification anyway.

type-synonym $('Var,'BExp)\ TyEnv = 'Var \rightharpoonup 'BExp\ Type$

This records which variables are *stable* in that we have an assumption implying that their value won't change. It duplicates a bit of info in Γ above but I haven't yet thought of a way to remove that duplication cleanly.

The first component of the pair records variables for which we have *AsmNoWrite*; the second component is for *AsmNoReadOrWrite*.

The reason we want to distinguish the different kinds of assumptions is to know whether a variable should remain in Γ when we drop an assumption on it. If we drop e.g. *AsmNoWrite* but also have *AsmNoReadOrWrite* then if we didn't track stability info this way we wouldn't know whether we had to remove the variable from Γ or not.

type-synonym $'Var\ Stable = ('Var\ set \times 'Var\ set)$

We track a set of predicates on memories as we execute. If we evaluate a boolean expression all of whose variables are stable, then we enrich this set predicate with that one. If we assign to a stable variable, then we enrich this predicate also. If we release an assumption making a variable unstable, we need to remove all predicates that pertain to it from this set.

This needs to be deeply embedded (i.e. it cannot be stored as a predicate of type $('Var, 'Val)\ Mem \Rightarrow bool$ or even $('Var, 'Val)\ Mem\ set$), because we need to be able to identify each individual predicate and for each predicate identify all of the variables in it, so we can discard the right predicates each time a variable becomes unstable.

type-synonym $'bexp\ preds = 'bexp\ set$

context *sifum-lang-no-dma* **begin**

definition
$pred :: 'BExp\ preds \Rightarrow ('Var,'Val)\ Mem \Rightarrow bool$
where
$pred\ P \equiv \lambda mem.\ (\forall\, p{\in}P.\ eval_B\ mem\ p)$

end

locale *sifum-types* =
sifum-lang-no-dma ev_A ev_B *aexp-vars bexp-vars* + *sifum-security dma* $\mathcal{C}$*-vars* $\mathcal{C}$ $eval_w$ *undefined*
for ev_A :: ('*Var*, '*Val*) *Mem* ⇒ '*AExp* ⇒ '*Val*
and ev_B :: ('*Var*, '*Val*) *Mem* ⇒ '*BExp* ⇒ *bool*
and *aexp-vars* :: '*AExp* ⇒ '*Var set*
and *bexp-vars* :: '*BExp* ⇒ '*Var set*
and *dma* :: ('*Var*,'*Val*) *Mem* ⇒ '*Var* ⇒ *Sec*
and $\mathcal{C}$*-vars* :: '*Var* ⇒ '*Var set*
and $\mathcal{C}$ :: '*Var set* +

fixes *bexp-neg* :: '*BExp* ⇒ '*BExp*
assumes *bexp-neg-negates*: ⋀*mem e*. (ev_B *mem* (*bexp-neg e*)) = (¬ (ev_B *mem e*))

fixes *assign-post* :: '*BExp preds* ⇒ '*Var* ⇒ '*AExp* ⇒ '*BExp preds*
assumes *assign-post-valid*: ⋀*mem*. *pred P mem* ⟹ *pred* (*assign-post P x e*) (*mem*(x := ev_A *mem e*))
fixes *dma-type* :: '*Var* ⇒ '*BExp set*
assumes *dma-correct*:
dma mem x = (*if* (∀ *e*∈*dma-type x*. ev_B *mem e*) *then Low else High*)
assumes $\mathcal{C}$*-vars-correct*:
$\mathcal{C}$*-vars x* = (⋃(*bexp-vars* ‘ *dma-type x*))
fixes *pred-False* :: '*BExp*
assumes *pred-False-is-False*: ¬ ev_B *mem pred-False*
assumes *bexp-vars-pred-False*: *bexp-vars pred-False* = {}

locale *sifum-types-assign* =
sifum-lang-no-dma ev_A ev_B *aexp-vars bexp-vars* + *sifum-security dma* $\mathcal{C}$*-vars* $\mathcal{C}$ $eval_w$ *undefined*
for ev_A :: ('*Var*, '*Val*) *Mem* ⇒ '*AExp* ⇒ '*Val*
and ev_B :: ('*Var*, '*Val*) *Mem* ⇒ '*BExp* ⇒ *bool*
and *aexp-vars* :: '*AExp* ⇒ '*Var set*
and *bexp-vars* :: '*BExp* ⇒ '*Var set*
and *dma* :: ('*Var*,'*Val*) *Mem* ⇒ '*Var* ⇒ *Sec*
and $\mathcal{C}$*-vars* :: '*Var* ⇒ '*Var set*
and $\mathcal{C}$ :: '*Var set* +

fixes *bexp-neg* :: '*BExp* ⇒ '*BExp*
assumes *bexp-neg-negates*: ⋀*mem e*. (ev_B *mem* (*bexp-neg e*)) = (¬ (ev_B *mem e*))
fixes *dma-type* :: '*Var* ⇒ '*BExp set*
assumes *dma-correct*:
dma mem x = (*if* (∀ *e*∈*dma-type x*. ev_B *mem e*) *then Low else High*)
assumes $\mathcal{C}$*-vars-correct*:
$\mathcal{C}$*-vars x* = (⋃(*bexp-vars* ‘ *dma-type x*))
fixes *pred-False* :: '*BExp*

assumes *pred-False-is-False*: $\neg$ ev_B *mem pred-False*
assumes *bexp-vars-pred-False*: *bexp-vars pred-False* = {}

fixes *bexp-assign* :: *'Var* $\Rightarrow$ *'AExp* $\Rightarrow$ *'BExp*
assumes *bexp-assign-eval*: $\bigwedge$*mem e x*. (ev_B *mem* (*bexp-assign x e*)) = (*mem x* = (ev_A *mem e*))
assumes *bexp-assign-vars*: $\bigwedge$*e x*. (*bexp-vars* (*bexp-assign x e*)) = *aexp-vars e* $\cup$ {*x*}

context *sifum-lang-no-dma* **begin**

definition
stable :: *'Var Stable* $\Rightarrow$ *'Var* $\Rightarrow$ *bool*
where
stable $\mathcal{S}$ *x* $\equiv$ *x* $\in$ (*fst* $\mathcal{S}$ $\cup$ *snd* $\mathcal{S}$)

definition
add-pred :: *'BExp preds* $\Rightarrow$ *'Var Stable* $\Rightarrow$ *'BExp* $\Rightarrow$ *'BExp preds* (- +- - [*120*, *120*, *120*] *1000*)
where
P $+_{\mathcal{S}}$ *e* $\equiv$ (*if* ($\forall$ *x*$\in$*bexp-vars e*. *stable* $\mathcal{S}$ *x*) *then P* $\cup$ {*e*} *else P*)

lemma *add-pred-subset*:
P $\subseteq$ *P* $+_{\mathcal{S}}$ *p*
apply(*clarsimp simp*: *add-pred-def*)
done

definition
restrict-preds-to-vars :: *'BExp preds* $\Rightarrow$ *'Var set* $\Rightarrow$ *'BExp preds* (- |‘ - [*120*, *120*] *1000*)
where
P |‘ *V* $\equiv$ {*e*. *e* $\in$ *P* $\wedge$ *bexp-vars e* $\subseteq$ *V*}

end

context *sifum-types-assign* **begin**

the most simple assignment postcondition transformer

definition
assign-post :: *'BExp preds* $\Rightarrow$ *'Var* $\Rightarrow$ *'AExp* $\Rightarrow$ *'BExp preds*
where
assign-post P x e $\equiv$
(*if x* $\in$ (*aexp-vars e*) *then*
(*restrict-preds-to-vars P* (−{*x*}))
else
(*restrict-preds-to-vars P* (−{*x*})) $\cup$ {*bexp-assign x e*})

end

sublocale *sifum-types-assign* ⊆ *sifum-types* - - - - - - - - *assign-post*
apply(*unfold-locales*)
using *bexp-neg-negates* **apply** *blast*
apply(*clarsimp simp*: *assign-post-def pred-def* | *safe*)+
using *eval-vars-det*$_B$
unfolding *restrict-preds-to-vars-def*
apply (*metis* (*mono-tags*, *lifting*) *ComplD fun-upd-other mem-Collect-eq singletonI subset-eq*)
unfolding *bexp-assign-eval*
using *eval-vars-det*$_A$
apply *fastforce*
using *eval-vars-det*$_B$
apply (*metis* (*mono-tags*, *lifting*) *ComplD fun-upd-other mem-Collect-eq singletonI subset-eq*)
using *dma-correct* **apply** *blast*
using $\mathcal{C}$*-vars-correct pred-False-is-False bexp-vars-pred-False* **apply** *blast*+
done

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 ⇝ *y* ≡ (*x*, *y*) ∈ *eval*$_w$

abbreviation
eval-plus-abv :: (-, *'Var*, *'Val*) *LocalConf* ⇒ (-, -, -) *LocalConf* ⇒ *bool*
(**infixl** ⇝$^+$ *70*)
where
x ⇝$^+$ *y* ≡ (*x*, *y*) ∈ *eval*$_w{}^+$

abbreviation
no-eval-abv :: (-, *'Var*, *'Val*) *LocalConf* ⇒ *bool*
(- ⇝ ⊥)
where
x ⇝ ⊥ ≡ ∀ *y*. (*x*, *y*) ∉ *eval*$_w$

abbreviation
low-indistinguishable-abv :: *'Var Mds* ⇒ (*'Var*, *'AExp*, *'BExp*) *Stmt* ⇒ (-, -, -) *Stmt* ⇒ *bool*
(- ∼1 - [*100*, *100*] *80*)
where
c ∼*mds* *c'* ≡ *low-indistinguishable mds c c'*

abbreviation
vars-of-type :: *'BExp Type* ⇒ *'Var set*
where
vars-of-type t ≡ ⋃(*bexp-vars* ‘ *t*)

definition
type-wellformed :: *'BExp Type* ⇒ *bool*
where
type-wellformed t ≡ *vars-of-type t* ⊆ *C*

lemma *dma-type-wellformed* [*simp*]:
type-wellformed (*dma-type x*)
apply(*clarsimp simp*: *type-wellformed-def C-def* | *safe*)+
using *C-vars-correct* **apply** *blast*
done

definition
to-total :: (*'Var*,*'BExp*) *TyEnv* ⇒ *'Var* ⇒ *'BExp Type*
where
to-total Γ ≡ λ*v*. *if v* ∈ *dom* Γ *then the* (Γ *v*) *else dma-type v*

definition
types-wellformed :: (*'Var*,*'BExp*) *TyEnv* ⇒ *bool*
where
types-wellformed Γ ≡ ∀ *x*∈*dom* Γ. *type-wellformed* (*the* (Γ *x*))

lemma *to-total-type-wellformed*:
types-wellformed Γ ⟹
type-wellformed (*to-total* Γ *x*)
by(*auto simp*: *to-total-def types-wellformed-def*)

lemma *Un-type-wellformed*:
∀ *t*∈*ts*. *type-wellformed t* ⟹ *type-wellformed* (⋃ *ts*)
apply(*clarsimp simp*: *type-wellformed-def* | *safe*)+
by(*fastforce simp*: *C-def elim*!: *subsetCE*)

inductive
type-aexpr :: (*'Var*,*'BExp*) *TyEnv* ⇒ *'AExp* ⇒ *'BExp Type* ⇒ *bool* (- ⊢*a* - ∈ - [*120*, *120*, *120*] *1000*)
where
type-aexpr [*intro*!]: Γ ⊢*a* *e* ∈ ⋃ (*image* (λ *x*. *to-total* Γ *x*) (*aexp-vars e*))

lemma *type-aexprI*:
t = ⋃ (*image* (λ *x*. *to-total* Γ *x*) (*aexp-vars e*)) ⟹ Γ ⊢$_a$ *e* ∈ *t*
apply(*erule ssubst*)
apply(*rule type-aexpr.intros*)
done

lemma *type-aexpr-type-wellformed*:
types-wellformed Γ ⟹ Γ ⊢$_a$ *e* ∈ *t* ⟹ *type-wellformed t*
apply(*erule type-aexpr.cases*)
apply(*erule ssubst*, *rule Un-type-wellformed*)
apply *clarsimp*
apply(*blast intro*: *to-total-type-wellformed*)
done

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

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

lemma *type-bexprI*:
t = ⋃ (*image* (λ *x*. *to-total* Γ *x*) (*bexp-vars e*)) ⟹ Γ ⊢$_b$ *e* ∈ *t*
apply(*erule ssubst*)
apply(*rule type-bexpr.intros*)
done

lemma *type-bexpr-type-wellformed*:
types-wellformed Γ ⟹ Γ ⊢$_b$ *e* ∈ *t* ⟹ *type-wellformed t*
apply(*erule type-bexpr.cases*)
apply(*erule ssubst*, *rule Un-type-wellformed*)
apply *clarsimp*
apply(*blast intro*: *to-total-type-wellformed*)
done

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

Define a sufficient condition for a type to be stable, assuming the type is wellformed.

We need this because there is no point tracking the fact that e.g. variable x's data has a classification that depends on some control variable c (where c might be the control variable for some other variable y whose value we've just assigned to x) if c can then go and be modified, since now the classification of the data in x no longer depends on the value of c, instead it depends on c's *old* value, which has now been lost.

Therefore, if a type depends on c, then c had better be stable.

abbreviation
pred-stable :: *'Var Stable* ⇒ *'BExp* ⇒ *bool*
where
pred-stable $\mathcal{S}$ *p* ≡ ∀ *x*∈*bexp-vars p*. *stable* $\mathcal{S}$ *x*

abbreviation
type-stable :: *'Var Stable* ⇒ *'BExp Type* ⇒ *bool*
where
type-stable $\mathcal{S}$ *t* ≡ (∀ *p*∈*t*. *pred-stable* $\mathcal{S}$ *p*)

lemma *type-stable-is-sufficient*:
⟦*type-stable* $\mathcal{S}$ *t*⟧ ⟹
∀ *mem mem'*. (∀ *x*. *stable* $\mathcal{S}$ *x* ⟶ *mem x* = *mem' x*) ⟶ (ev_B *mem*) ‘ *t* = (ev_B *mem'*) ‘ *t*
apply(*clarsimp simp*: *type-wellformed-def image-def*)
apply *safe*
using *eval-vars-det*$_B$ **apply** *blast*+
done

definition
mds-consistent :: *'Var Mds* ⇒ (*'Var*,*'BExp*) *TyEnv* ⇒ *'Var Stable* ⇒ *'BExp preds* ⇒ *bool*
where
mds-consistent mds Γ $\mathcal{S}$ *P* ≡
($\mathcal{S}$ = (*mds AsmNoWrite*, *mds AsmNoReadOrWrite*)) ∧
(*dom* Γ = {*x*. *x* ∉ $\mathcal{C}$ ∧ *stable* $\mathcal{S}$ *x*}) ∧
(∀ *p* ∈ *P*. *pred-stable* $\mathcal{S}$ *p*)

fun
add-anno-dom :: (*'Var*,*'BExp*) *TyEnv* ⇒ *'Var Stable* ⇒ *'Var ModeUpd* ⇒ *'Var set*
where
add-anno-dom Γ $\mathcal{S}$ (*Acq v AsmNoReadOrWrite*) = (*if v* ∉ $\mathcal{C}$ *then dom* Γ ∪ {*v*} *else dom* Γ) |
add-anno-dom Γ $\mathcal{S}$ (*Acq v AsmNoWrite*) = (*if v* ∉ $\mathcal{C}$ *then dom* Γ ∪ {*v*} *else dom* Γ) |
add-anno-dom Γ $\mathcal{S}$ (*Acq v* -) = *dom* Γ |
add-anno-dom Γ $\mathcal{S}$ (*Rel v AsmNoReadOrWrite*) = (*if v* ∉ *fst* $\mathcal{S}$ *then dom* Γ − {*v*} *else dom* Γ) |
add-anno-dom Γ $\mathcal{S}$ (*Rel v AsmNoWrite*) = (*if v* ∉ *snd* $\mathcal{S}$ *then dom* Γ − {*v*} *else dom* Γ) |
add-anno-dom Γ $\mathcal{S}$ (*Rel v* -) = *dom* Γ

definition
add-anno :: (*'Var*,*'BExp*) *TyEnv* ⇒ *'Var Stable* ⇒ *'Var ModeUpd* ⇒ (*'Var*,*'BExp*) *TyEnv* (- ⊕- - [*120*, *120*, *120*] *1000*)
where
Γ ⊕$_{\mathcal{S}}$ *upd* = *restrict-map* (λ*x*. *Some* (*to-total* Γ *x*)) (*add-anno-dom* Γ $\mathcal{S}$ *upd*)

lemma *add-anno-acq-AsmNoReadOrWrite-idemp* [*simp*]:
 $v \in dom\ \Gamma \vee v \in \mathcal{C} \Longrightarrow \Gamma \oplus_{\mathcal{S}} (Acq\ v\ AsmNoReadOrWrite) = \Gamma$
 apply(*safe* | *clarsimp simp*: *add-anno-def to-total-def*)+
  apply(*rule ext*)
  apply(*clarsimp simp*: *restrict-map-def* | *safe*)+
  apply(*case-tac* Γ x, *fastforce*+)[*5*]
 apply(*rule ext*)
 apply(*clarsimp simp*: *restrict-map-def* | *safe*)+
 apply(*case-tac* Γ x, *fastforce*+)
 apply(*safe* | *clarsimp simp*: *add-anno-def to-total-def*)+
 apply(*rule ext*)
 apply(*clarsimp simp*: *restrict-map-def* | *safe*)+
 apply(*case-tac* Γ x, *fastforce*+)
 done

lemma *add-anno-rel-AsmNoReadOrWrite-idemp* [*simp*]:
 $[\![v \notin dom\ \Gamma;\ v \notin fst\ \mathcal{S}]\!] \Longrightarrow \Gamma \oplus_{\mathcal{S}} (Rel\ v\ AsmNoReadOrWrite) = \Gamma$
 apply(*subgoal-tac* $v \notin dom\ \Gamma$)
  apply(*safe* | *clarsimp simp*: *add-anno-def to-total-def*)+
 apply(*clarsimp simp*: *restrict-map-def* | *safe*)+
 apply(*erule-tac* P=($\lambda x.$ *if* $x \in dom\ \Gamma \wedge x \neq v$
 then Some (*if* $x \in dom\ \Gamma$ *then the* (Γ x) *else dma-type* x) *else None*) = Γ
in *notE*)
 apply(*rule ext*)
 apply(*case-tac* Γ x, *fastforce*+)
 done

lemma *add-anno-acq-AsmNoReadOrWrite* [*simp*]:
 assumes *notin* [*simp*]: $v \notin dom\ \Gamma$
 shows $v \notin \mathcal{C} \Longrightarrow \Gamma \oplus_{\mathcal{S}} (Acq\ v\ AsmNoReadOrWrite) = (\Gamma(v \mapsto dma\text{-}type\ v))$
 apply(*safe* | *clarsimp simp*: *add-anno-def to-total-def*)+
 apply(*clarsimp simp*: *restrict-map-def* | *safe*)+
 apply(*rule ext*)
 apply(*auto intro*: *sym*)
 done

lemma *add-anno-rel-AsmNoReadOrWrite* [*simp*]:
 assumes *isin* [*simp*]: $v \in dom\ \Gamma$
 shows $v \notin fst\ \mathcal{S} \Longrightarrow \Gamma \oplus_{\mathcal{S}} (Rel\ v\ AsmNoReadOrWrite) = restrict\text{-}map\ \Gamma\ ((dom\ \Gamma) - \{v\})$
 apply(*safe* | *clarsimp simp*: *add-anno-def to-total-def*)+
 apply(*clarsimp simp*: *restrict-map-def* | *safe*)+
 apply(*rule ext*)
 apply(*auto intro*: *sym*)
 done

lemma *add-anno-acq-AsmNoWrite-idemp* [*simp*]:
 $v \in dom\ \Gamma \vee v \in \mathcal{C} \Longrightarrow \Gamma \oplus_{\mathcal{S}} (Acq\ v\ AsmNoWrite) = \Gamma$
 apply(*safe* | *clarsimp simp*: *add-anno-def to-total-def*)+

apply(*rule ext*)
apply(*clarsimp simp*: *restrict-map-def* | *safe*)+
apply(*case-tac* Γ *x*, *fastforce*+)[*5*]
apply(*rule ext*)
apply(*clarsimp simp*: *restrict-map-def* | *safe*)+
apply(*case-tac* Γ *x*, *fastforce*+)
apply(*safe* | *clarsimp simp*: *add-anno-def to-total-def*)+
apply(*rule ext*)
apply(*clarsimp simp*: *restrict-map-def* | *safe*)+
apply(*case-tac* Γ *x*, *fastforce*+)
done

lemma *add-anno-rel-AsmNoWrite-idemp* [*simp*]:
⟦*v* ∉ *dom* Γ; *v* ∉ *snd* S⟧ ⟹ Γ ⊕S (*Rel v AsmNoWrite*) = Γ
apply(*subgoal-tac v* ∉ *dom* Γ)
apply(*safe* | *clarsimp simp*: *add-anno-def to-total-def*)+
apply(*clarsimp simp*: *restrict-map-def* | *safe*)+
apply(*erule-tac P*=(λ*x*. *if x* ∈ *dom* Γ ∧ *x* ≠ *v*
then Some (*if x* ∈ *dom* Γ *then the* (Γ *x*) *else dma-type x*) *else None*) = Γ
in *notE*)
apply(*rule ext*)
apply(*case-tac* Γ *x*, *fastforce*+)
done

lemma *add-anno-acq-AsmNoWrite* [*simp*]:
assumes *notin* [*simp*]: *v* ∉ *dom* Γ
shows *v* ∉ C ⟹ Γ ⊕S (*Acq v AsmNoWrite*) = (Γ(*v* ↦ *dma-type v*))
apply(*safe* | *clarsimp simp*: *add-anno-def to-total-def*)+
apply(*clarsimp simp*: *restrict-map-def* | *safe*)+
apply(*rule ext*)
apply(*auto intro*: *sym*)
done

lemma *add-anno-rel-AsmNoWrite* [*simp*]:
assumes *isin* [*simp*]: *v* ∈ *dom* Γ
shows *v* ∉ *snd* S ⟹ Γ ⊕S (*Rel v AsmNoWrite*) = *restrict-map* Γ ((*dom* Γ) − {*v*})
apply(*safe* | *clarsimp simp*: *add-anno-def to-total-def*)+
apply(*clarsimp simp*: *restrict-map-def* | *safe*)+
apply(*rule ext*)
apply(*auto intro*: *sym*)
done

fun
add-anno-stable :: *'Var Stable* ⇒ *'Var ModeUpd* ⇒ *'Var Stable*
where
add-anno-stable S (*Acq v AsmNoReadOrWrite*) = (*fst* S, *snd* S ∪ {*v*}) |
add-anno-stable S (*Acq v AsmNoWrite*) = (*fst* S ∪ {*v*}, *snd* S) |
add-anno-stable S (*Acq v* -) = S |

add-anno-stable $\mathcal{S}$ (*Rel v AsmNoReadOrWrite*) = (*fst* $\mathcal{S}$, *snd* $\mathcal{S}$ − {*v*}) |
add-anno-stable $\mathcal{S}$ (*Rel v AsmNoWrite*) = (*fst* $\mathcal{S}$ − {*v*}, *snd* $\mathcal{S}$) |
add-anno-stable $\mathcal{S}$ (*Rel v* -) = $\mathcal{S}$

definition
pred-entailment :: *'BExp preds* ⇒ *'BExp preds* ⇒ *bool* (**infix** ⊢ *70*)
where
$P \vdash P' \equiv \forall\, mem.\ pred\ P\ mem \longrightarrow pred\ P'\ mem$

We give a predicate interpretation of subtype and then prove it has the correct semantic property.

definition
subtype :: *'BExp Type* ⇒ *'BExp preds* ⇒ *'BExp Type* ⇒ *bool* (- ≤:_ - [*120*, *120*, *120*] *1000*)
where
$t \leq:_P t' \equiv (P \cup t') \vdash t$

definition
type-max :: *'BExp Type* ⇒ (*'Var*,*'Val*) *Mem* ⇒ *Sec*
where
type-max t mem ≡ *if* (∀ *p*∈*t*. ev_B *mem p*) *then Low else High*

lemma *type-stable-is-sufficient'*:
⟦*type-stable* $\mathcal{S}$ *t*⟧ ⟹
∀ *mem mem'*. (∀ *x*. *stable* $\mathcal{S}$ *x* ⟶ *mem x* = *mem' x*) ⟶ *type-max t mem* = *type-max t mem'*
using *type-stable-is-sufficient*
unfolding *type-max-def image-def*
by (*metis* (*no-types*, *lifting*) *eval-vars-det*$_B$)

lemma *subtype-sound*:
$t \leq:_P t' \Longrightarrow \forall\, mem.\ pred\ P\ mem \longrightarrow type\text{-}max\ t\ mem \leq type\text{-}max\ t'\ mem$
apply(*fastforce simp*: *subtype-def pred-entailment-def pred-def type-max-def less-eq-Sec-def*)
done

lemma *subtype-complete*:
assumes *a*: ⋀*mem*. *pred P mem* ⟹ *type-max t mem* ≤ *type-max t' mem*
shows $t \leq:_P t'$
unfolding *subtype-def pred-entailment-def*
proof (*clarify*)
fix *mem*
assume *p*: *pred* ($P \cup t'$) *mem*
hence *pred P mem*
unfolding *pred-def* **by** *blast*
with *a* **have** *tmax*: *type-max t mem* ≤ *type-max t' mem* **by** *blast*
from *p* **have** *t'*: *pred t' mem*
unfolding *pred-def* **by** *blast*
from *t'* **have** *type-max t' mem* = *Low*
unfolding *type-max-def pred-def* **by** *force*

with *tmax* **have** *type-max t mem* $\leq$ *Low*
by *simp*
hence *type-max t mem* = *Low*
unfolding *less-eq-Sec-def* **by** *blast*
thus *pred t mem*
unfolding *type-max-def pred-def* **by** (*auto split*: *if-splits*)
qed

lemma *subtype-correct*:
($t \leq:_P t'$) = ($\forall$ *mem*. *pred P mem* $\longrightarrow$ *type-max t mem* $\leq$ *type-max t' mem*)
apply(*rule iffI*)
apply(*simp add*: *subtype-sound*)
apply(*simp add*: *subtype-complete*)
done

definition
type-equiv :: *'BExp Type* $\Rightarrow$ *'BExp preds* $\Rightarrow$ *'BExp Type* $\Rightarrow$ *bool* (- =:_ - [*120*, *120*, *120*] *1000*)
where
$t =:_P t' \equiv t \leq:_P t' \wedge t' \leq:_P t$

lemma *subtype-refl* [*simp*]:
$t \leq:_P t$
by(*simp add*: *subtype-def pred-entailment-def pred-def*)

lemma *type-equiv-refl* [*simp*]:
$t =:_P t$
by (*simp add*: *type-equiv-def*)

definition
anno-type-stable :: (*'Var*,*'BExp*) *TyEnv* $\Rightarrow$ *'Var Stable* $\Rightarrow$ *'Var ModeUpd* $\Rightarrow$ *bool*
where
anno-type-stable Γ $\mathcal{S}$ *upd* $\equiv$ (*case upd of* (*Rel v m*) $\Rightarrow$
($v \in \mathcal{C} \wedge v \notin$ *add-anno-dom* Γ $\mathcal{S}$ *upd*) $\longrightarrow$
($\forall x \in$ *dom* Γ. $v \notin$ *vars-of-type* (*the* (Γ *x*)))
| (*Acq v m*) $\Rightarrow$
($v \notin \mathcal{C} \wedge v \in$ *add-anno-dom* Γ $\mathcal{S}$ *upd* $-$ *dom* Γ) $\longrightarrow$
($\forall x \in \mathcal{C}$-*vars v*. *stable* $\mathcal{S}$ *x*))

definition
anno-type-sec :: (*'Var*,*'BExp*) *TyEnv* $\Rightarrow$ *'Var Stable* $\Rightarrow$ *'BExp preds* $\Rightarrow$ *'Var ModeUpd* $\Rightarrow$ *bool*
where
anno-type-sec Γ $\mathcal{S}$ *P upd* $\equiv$ (*case upd of* (*Rel v AsmNoReadOrWrite*) $\Rightarrow$
($v \in$ *add-anno-dom* Γ $\mathcal{S}$ *upd* $\longrightarrow$ (*the* (Γ *v*)) $\leq:_P$ (*dma-type v*))
| - $\Rightarrow$ *True*)

definition
types-stable :: (*'Var*,*'BExp*) *TyEnv* ⇒ *'Var Stable* ⇒ *bool*
where
types-stable Γ $\mathcal{S}$ ≡ ∀ *x*∈*dom* Γ. *type-stable* $\mathcal{S}$ (*the* (Γ *x*))

definition
tyenv-wellformed :: *'Var Mds* ⇒ (*'Var*,*'BExp*) *TyEnv* ⇒ *'Var Stable* ⇒ *'BExp preds* ⇒ *bool*
where
tyenv-wellformed mds Γ $\mathcal{S}$ *P* ≡
mds-consistent mds Γ $\mathcal{S}$ *P* ∧
types-wellformed Γ ∧ *types-stable* Γ $\mathcal{S}$

lemma *subset-entailment*:
P' ⊆ *P* ⟹ *P* ⊢ *P'*
apply(*auto simp*: *pred-entailment-def pred-def*)
done

lemma *pred-entailment-refl* [*simp*]:
P ⊢ *P*
by(*simp add*: *pred-entailment-def*)

lemma *pred-entailment-mono*:
P ⊢ *P'* ⟹ *P* ⊆ *P''* ⟹ *P''* ⊢ *P'*
by(*auto simp*: *pred-entailment-def pred-def*)

lemma *type-equiv-subset*:
type-equiv t P t' ⟹ *P* ⊆ *P'* ⟹ *type-equiv t P' t'*
apply(*auto simp*: *type-equiv-def subtype-def intro*: *pred-entailment-mono*)
done

definition
context-equiv :: (*'Var*,*'BExp*) *TyEnv* ⇒ *'BExp preds* ⇒ (*'Var*,*'BExp*) *TyEnv* ⇒ *bool* (- =:_ - [*120*, *120*, *120*] *1000*)
where
Γ =:$_P$ Γ' ≡ *dom* Γ = *dom* Γ' ∧
(∀ *x*∈*dom* Γ'. *type-equiv* (*the* (Γ *x*)) *P* (*the* (Γ' *x*)))

lemma *context-equiv-refl*[*simp*]:
context-equiv Γ *P* Γ
by(*simp add*: *context-equiv-def*)

lemma *context-equiv-subset*:
context-equiv Γ *P* Γ' ⟹ *P* ⊆ *P'* ⟹ *context-equiv* Γ *P'* Γ'
apply(*auto simp*: *context-equiv-def intro*: *type-equiv-subset*)
done

lemma *pred-entailment-trans*:
P ⊢ *P'* ⟹ *P'* ⊢ *P''* ⟹ *P* ⊢ *P''*

by(*auto simp*: *pred-entailment-def*)

lemma *pred-un* [*simp*]:
pred $(P \cup P')$ *mem* = (*pred P mem* $\wedge$ *pred* P' *mem*)
apply(*auto simp*: *pred-def*)
done

lemma *pred-entailment-un*:
$P \vdash P' \Longrightarrow P \vdash P'' \Longrightarrow P \vdash (P' \cup P'')$
apply(*subst pred-entailment-def*)
apply *clarsimp*
apply(*fastforce simp*: *pred-entailment-def*)
done

lemma *pred-entailment-mono-un*:
$P \vdash P' \Longrightarrow (P \cup P'') \vdash (P' \cup P'')$
apply(*auto simp*: *pred-entailment-def pred-def*)
done

lemma *subtype-trans*:
$t \leq:_P t' \Longrightarrow t' \leq:_P t'' \Longrightarrow P \vdash P' \Longrightarrow t \leq:_P t''$
$t \leq:_P' t' \Longrightarrow t' \leq:_P t'' \Longrightarrow P \vdash P' \Longrightarrow t \leq:_P t''$
apply(*clarsimp simp*: *subtype-def*)
apply(*rule pred-entailment-trans*)
prefer *2*
apply *assumption*
apply(*rule pred-entailment-un*)
apply(*blast intro*: *subset-entailment*)
apply(*rule pred-entailment-trans*)
prefer *2*
apply *assumption*
apply(*blast intro*: *pred-entailment-mono-un*)
apply(*clarsimp simp*: *subtype-def*)
apply(*rule pred-entailment-trans*)
prefer *2*
apply *assumption*
apply(*rule pred-entailment-un*)
apply(*blast intro*: *pred-entailment-mono*)
apply(*blast intro*: *subset-entailment*)
done

lemma *type-equiv-trans*:
type-equiv t P $t' \Longrightarrow$ *type-equiv* t' P' $t'' \Longrightarrow P \vdash P' \Longrightarrow$ *type-equiv t P* t''
apply(*auto simp*: *type-equiv-def intro*: *subtype-trans*)
done

lemma *context-equiv-trans*:
context-equiv Γ *P* $\Gamma' \Longrightarrow$ *context-equiv* Γ' P' $\Gamma'' \Longrightarrow P \vdash P' \Longrightarrow$ *context-equiv* Γ *P* Γ''

apply(*force simp*: *context-equiv-def intro*: *type-equiv-trans*)
done

lemma *un-pred-entailment-mono*:
$(P \cup P') \vdash P'' \Longrightarrow P''' \vdash P \Longrightarrow (P''' \cup P') \vdash P''$
unfolding *pred-entailment-def pred-def*
apply *blast*
done

lemma *subtype-entailment*:
$t \leq:_P t' \Longrightarrow P' \vdash P \Longrightarrow t \leq:_{P'} t'$
apply(*auto simp*: *subtype-def intro*: *un-pred-entailment-mono*)
done

lemma *type-equiv-entailment*:
type-equiv $t\ P\ t' \Longrightarrow P' \vdash P \Longrightarrow$ *type-equiv* $t\ P'\ t'$
apply(*auto simp*: *type-equiv-def intro*: *subtype-entailment*)
done

lemma *context-equiv-entailment*:
context-equiv $\Gamma\ P\ \Gamma' \Longrightarrow P' \vdash P \Longrightarrow$ *context-equiv* $\Gamma\ P'\ \Gamma'$
apply(*auto simp*: *context-equiv-def intro*: *type-equiv-entailment*)
done

inductive
has-type :: ('*Var*,'*BExp*) *TyEnv* ⇒ '*Var Stable* ⇒ '*BExp preds* ⇒ ('*Var*, '*AExp*, '*BExp*) *Stmt* ⇒ ('*Var*,'*BExp*) *TyEnv* ⇒ '*Var Stable* ⇒ '*BExp preds* ⇒ *bool*
(⊢ -,-,- {-} -,-,- [120, 120, 120, 120, 120, 120, 120] 1000)
where
stop-type [*intro*]: ⊢ Γ,$\mathcal{S}$,*P* {*Stop*} Γ,$\mathcal{S}$,*P* |
skip-type [*intro*] : ⊢ Γ,$\mathcal{S}$,*P* {*Skip*} Γ,$\mathcal{S}$,*P* |
$assign_{\mathcal{C}}$:
⟦$x \in \mathcal{C}$; Γ ⊢$_a$ $e \in t$; $P \vdash t$; (∀ v∈*dom* Γ. $x \notin$ *vars-of-type* (*the* (Γ v)));
P' = *restrict-preds-to-vars* (*assign-post* $P\ x\ e$) {v. *stable* $\mathcal{S}$ v};
∀ v. $x \in \mathcal{C}$-*vars* $v \wedge v \notin$ *snd* $\mathcal{S} \longrightarrow P \vdash$ (*to-total* Γ v) ∧
(*to-total* Γ v) ≤:$_{P'}$ (*dma-type* v) ⟧ ⟹
⊢ Γ,$\mathcal{S}$,*P* {$x \leftarrow e$} Γ,$\mathcal{S}$,P' |
$assign_1$:
⟦ $x \notin$ *dom* Γ ; $x \notin \mathcal{C}$; Γ ⊢$_a$ $e \in t$; $t \leq:_P$ (*dma-type* x);
P' = *restrict-preds-to-vars* (*assign-post* $P\ x\ e$) {v. *stable* $\mathcal{S}$ v} ⟧ ⟹
⊢ Γ,$\mathcal{S}$,*P* {$x \leftarrow e$} Γ,$\mathcal{S}$,P' |
$assign_2$:
⟦ $x \in$ *dom* Γ ; Γ ⊢$_a$ $e \in t$; *type-stable* $\mathcal{S}$ t; P' = *restrict-preds-to-vars* (*assign-post* $P\ x\ e$) {v. *stable* $\mathcal{S}$ v};
$x \notin$ *snd* $\mathcal{S} \longrightarrow t \leq:_{P'}$ (*dma-type* x) ⟧ ⟹
has-type Γ $\mathcal{S}$ P ($x \leftarrow e$) (Γ (x := *Some* t)) $\mathcal{S}$ P' |

if-type [*intro*]:
$[\![\ \Gamma \vdash_b e \in t;\ P \vdash t;$
$\vdash \Gamma,\mathcal{S},(P +_{\mathcal{S}} e)\ \{\ c_1\ \}\ \Gamma',\mathcal{S}',P';\ \vdash \Gamma,\mathcal{S},(P +_{\mathcal{S}} (\textit{bexp-neg}\ e))\ \{\ c_2\ \}\ \Gamma'',\mathcal{S}',P'';$
context-equiv $\Gamma'\ P'\ \Gamma'''$; *context-equiv* $\Gamma''\ P''\ \Gamma'''$; $P' \vdash P'''$; $P'' \vdash P'''$;
$\forall$ *mds. tyenv-wellformed mds* $\Gamma'\ \mathcal{S}'\ P' \longrightarrow$ *tyenv-wellformed mds* $\Gamma'''\ \mathcal{S}'\ P'''$;
$\forall$ *mds. tyenv-wellformed mds* $\Gamma''\ \mathcal{S}'\ P'' \longrightarrow$ *tyenv-wellformed mds* $\Gamma'''\ \mathcal{S}'\ P'''$
$]\!] \Longrightarrow$
$\vdash \Gamma,\mathcal{S},P\ \{\ \textit{If}\ e\ c_1\ c_2\ \}\ \Gamma''',\mathcal{S}',P'''\ |$
while-type [*intro*]: $[\![\ \Gamma \vdash_b e \in t\ ;\ P \vdash t; \vdash \Gamma,\mathcal{S},(P +_{\mathcal{S}} e)\ \{\ c\ \}\ \Gamma,\mathcal{S},P\ ]\!] \Longrightarrow \vdash \Gamma,\mathcal{S},P\ \{\ \textit{While}\ e\ c\ \}\ \Gamma,\mathcal{S},P\ |$
anno-type [*intro*]: $[\![\ \Gamma' = \Gamma \oplus_{\mathcal{S}} \textit{upd}\ ;\ \mathcal{S}' = \textit{add-anno-stable}\ \mathcal{S}\ \textit{upd};\ P' = \textit{restrict-preds-to-vars}\ P\ \{v.\ \textit{stable}\ \mathcal{S}'\ v\};$
$\vdash \Gamma',\mathcal{S}',P'\ \{\ c\ \}\ \Gamma'',\mathcal{S}'',P''\ ;\ c \neq \textit{Stop}\ ;$
$(\bigwedge x.\ (\textit{to-total}\ \Gamma\ x) \leq:_{P'}\ (\textit{to-total}\ \Gamma'\ x));$
$\textit{anno-type-stable}\ \Gamma\ \mathcal{S}\ \textit{upd};\ \textit{anno-type-sec}\ \Gamma\ \mathcal{S}\ P\ \textit{upd}]\!] \Longrightarrow \vdash \Gamma,\mathcal{S},P\ \{\ c@[\textit{upd}]\ \}\ \Gamma'',\mathcal{S}'',P''\ |$
seq-type [*intro*]: $[\![\vdash \Gamma,\mathcal{S},P\ \{\ c_1\ \}\ \Gamma',\mathcal{S}',P'\ ; \vdash \Gamma',\mathcal{S}',P'\ \{\ c_2\ \}\ \Gamma'',\mathcal{S}'',P''\ ]\!] \Longrightarrow \vdash \Gamma,\mathcal{S},P\ \{\ c_1\ ;;\ c_2\ \}\ \Gamma'',\mathcal{S}'',P''\ |$
sub : $[\![\vdash \Gamma_1,\mathcal{S},P_1\ \{\ c\ \}\ \Gamma_1',\mathcal{S}',P_1'$; *context-equiv* $\Gamma_2\ P_2\ \Gamma_1$; ($\forall$ *mds. tyenv-wellformed mds* $\Gamma_2\ \mathcal{S}\ P_2 \longrightarrow$ *tyenv-wellformed mds* $\Gamma_1\ \mathcal{S}\ P_1$);
($\forall$ *mds. tyenv-wellformed mds* $\Gamma_1'\ \mathcal{S}'\ P_1' \longrightarrow$ *tyenv-wellformed mds* $\Gamma_2'\ \mathcal{S}'\ P_2'$); *context-equiv* $\Gamma_1'\ P_1'\ \Gamma_2'$; $P_2 \vdash P_1$; $P_1' \vdash P_2'\ ]\!] \Longrightarrow \vdash \Gamma_2,\mathcal{S},P_2\ \{\ c\ \}\ \Gamma_2',\mathcal{S}',P_2'\ |$
await-type [*intro*]:
$[\![\ \Gamma \vdash_b e \in t;\ P \vdash t;$
$\vdash \Gamma,\mathcal{S},(P +_{\mathcal{S}} e)\ \{\ c\ \}\ \Gamma',\mathcal{S}',P'\ ]\!] \Longrightarrow$
$\vdash \Gamma,\mathcal{S},P\ \{\ \textit{Await}\ e\ c\ \}\ \Gamma',\mathcal{S}',P'$

lemma *sub'*:
$[\![$ *context-equiv* $\Gamma_2\ P_2\ \Gamma_1$;
($\forall$ *mds. tyenv-wellformed mds* $\Gamma_2\ \mathcal{S}\ P_2 \longrightarrow$ *tyenv-wellformed mds* $\Gamma_1\ \mathcal{S}\ P_1$);
($\forall$ *mds. tyenv-wellformed mds* $\Gamma_1'\ \mathcal{S}'\ P_1' \longrightarrow$ *tyenv-wellformed mds* $\Gamma_2'\ \mathcal{S}'\ P_2'$);
context-equiv $\Gamma_1'\ P_1'\ \Gamma_2'$;
$P_2 \vdash P_1$;
$P_1' \vdash P_2'$;
$\vdash \Gamma_1,\mathcal{S},P_1\ \{\ c\ \}\ \Gamma_1',\mathcal{S}',P_1'\ ]\!] \Longrightarrow$
$\vdash \Gamma_2,\mathcal{S},P_2\ \{\ c\ \}\ \Gamma_2',\mathcal{S}',P_2'$
by(*rule sub*)

lemma *assign$_2$-helper*:
$[\![\Gamma\ x = \textit{Some}\ t;\ \textit{has-type}\ \Gamma\ \mathcal{S}\ P\ (x \leftarrow e)\ (\Gamma(x \mapsto t))\ \mathcal{S}\ P']\!] \Longrightarrow \textit{has-type}\ \Gamma\ \mathcal{S}\ P\ (x \leftarrow e)\ \Gamma\ \mathcal{S}\ P'$
by (*simp add*:*map-upd-triv*)

lemma *conc'*:
$[\![\vdash \Gamma_1,\mathcal{S},P\ \{\ c\ \}\ \Gamma',\mathcal{S}',P';$
$\Gamma_1 = (\Gamma_2(x \mapsto t))$;
$x \in \textit{dom}\ \Gamma_2$;
$\textit{type-equiv}\ (\textit{the}\ (\Gamma_2\ x))\ P\ t$;

type-wellformed t;
type-stable $\mathcal{S}$ t $\rrbracket \Longrightarrow$
$\vdash \Gamma_2,\mathcal{S},P \{ c \} \Gamma',\mathcal{S}',P'$
apply(*erule sub*)
apply(*fastforce simp*: *context-equiv-def*)
apply(*clarsimp simp*: *tyenv-wellformed-def mds-consistent-def*)
apply(*rule conjI*)
apply *fastforce*
apply(*rule conjI*)
apply(*fastforce simp*: *types-wellformed-def*)
apply(*fastforce simp*: *types-stable-def*)
apply *blast*
apply *simp*+
done

lemma *tyenv-wellformed-subset*:
tyenv-wellformed mds Γ $\mathcal{S}$ $P \Longrightarrow P' \subseteq P \Longrightarrow$ *tyenv-wellformed mds* Γ $\mathcal{S}$ P'
apply(*auto simp*: *tyenv-wellformed-def mds-consistent-def*)
done

lemma *if-type'*:
$\llbracket \Gamma \vdash_b e \in t$;
$P \vdash t$;
$\vdash \Gamma,\mathcal{S},(P +_{\mathcal{S}} e) \{ c_1 \} \Gamma',\mathcal{S}',P'$;
$\vdash \Gamma,\mathcal{S},(P +_{\mathcal{S}} (\textit{bexp-neg } e)) \{ c_2 \} \Gamma',\mathcal{S}',P''$;
$P''' \subseteq P' \cap P'' \rrbracket \Longrightarrow$
$\vdash \Gamma,\mathcal{S},P \{ \textit{If } e\ c_1\ c_2 \} \Gamma',\mathcal{S}',P'''$
apply(*erule* (*3*) *if-type*)
apply(*rule context-equiv-refl*)
apply(*rule context-equiv-refl*)
apply(*blast intro*: *subset-entailment*)+
apply(*blast intro*: *tyenv-wellformed-subset*)+
done

lemma *skip-type'*:
$\llbracket \Gamma = \Gamma'; \mathcal{S} = \mathcal{S}'; P = P' \rrbracket \Longrightarrow \vdash \Gamma,\mathcal{S},P \{\textit{Skip}\} \Gamma',\mathcal{S}',P'$
using *skip-type* **by** *simp*

Some helper lemmas to discharge the assumption of the $\llbracket ?\Gamma' = ?\Gamma \oplus_? \mathcal{S}$ *?upd*; $?\mathcal{S}' =$ *add-anno-stable ?*$\mathcal{S}$ *?upd*; $?P' = ?P$ |‘ $\{v.$ *stable ?*$\mathcal{S}'$ $v\}$; $\vdash$ $?\Gamma',?\mathcal{S}',?P' \{?c\}$ $?\Gamma'',?\mathcal{S}'',?P''$; $?c \neq$ *Stop*; $\bigwedge x.$ *to-total ?*Γ $x \leq:_? P'$ *to-total* $?\Gamma'$ x; *anno-type-stable ?*Γ *?*$\mathcal{S}$ *?upd*; *anno-type-sec ?*Γ *?*$\mathcal{S}$ *?P ?upd*$\rrbracket \Longrightarrow \vdash$ $?\Gamma,?\mathcal{S},?P \{?c@[?upd]\}$ $?\Gamma'',?\mathcal{S}'',?P''$ rule.

lemma *anno-type-helpers* [*simp*]:
(*to-total* Γ x) $\leq:_P$ (*to-total* (*add-anno* Γ $\mathcal{S}$ (*buffer* $+=_m$ *AsmNoWrite*)) x)
(*to-total* Γ x) $\leq:_P$ (*to-total* (*add-anno* Γ $\mathcal{S}$ (*buffer* $+=_m$ *AsmNoReadOrWrite*)) x)
apply(*auto simp*: *to-total-def add-anno-def subtype-def intro*: *subset-entailment*)
done

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.

inductive-cases *has-type-elim*: $\vdash \Gamma,\mathcal{S},P \{ c \} \Gamma',\mathcal{S}',P'$
inductive-cases *has-type-stop-elim*: $\vdash \Gamma,\mathcal{S},P \{ Stop \} \Gamma',\mathcal{S}',P'$

definition *tyenv-eq* :: ('*Var*,'*BExp*) *TyEnv* ⇒ ('*Var*, '*Val*) *Mem* ⇒ ('*Var*, '*Val*) *Mem* ⇒ *bool*
(**infix** $=_1$ *60*)
where $mem_1 =_\Gamma mem_2 \equiv \forall\ x.$ (*type-max* (*to-total* Γ *x*) mem_1 = *Low* ⟶ mem_1 *x* = mem_2 *x*)

lemma *type-max-dma-type* [*simp*]:
type-max (*dma-type x*) *mem* = *dma mem x*
using *dma-correct* **unfolding** *type-max-def* **apply** *auto*
done

This result followed trivially for Mantel et al., but we need to know that the type environment is wellformed.

lemma *tyenv-eq-sym'*:
dom Γ ∩ $\mathcal{C}$ = {} ⟹ *types-wellformed* Γ ⟹ $mem_1 =_\Gamma mem_2$ ⟹ $mem_2 =_\Gamma mem_1$
proof(*clarsimp simp*: *tyenv-eq-def*)
fix *x*
assume *a*: ∀ *x*. *type-max* (*to-total* Γ *x*) mem_1 = *Low* ⟶ mem_1 *x* = mem_2 *x*
assume *b*: *dom* Γ ∩ $\mathcal{C}$ = {}
from *a b* **have** *eq-*$\mathcal{C}$: ∀ *x*∈$\mathcal{C}$. mem_1 *x* = mem_2 *x*
by (*fastforce simp*: *to-total-def* $\mathcal{C}$*-Low type-max-dma-type split*: *if-splits*)
hence *dma* mem_1 = *dma* mem_2
by (*rule dma-*$\mathcal{C}$)
hence *dma-type-eq*: *type-max* (*dma-type x*) mem_1 = *type-max* (*dma-type x*) mem_2
by(*simp*)
assume *c*: *types-wellformed* Γ

assume *d*: *type-max* (*to-total* Γ *x*) mem_2 = *Low*
show mem_2 *x* = mem_1 *x*
proof(*cases x* ∈ *dom* Γ)
assume *in-dom*: *x* ∈ *dom* Γ
from *this* **obtain** *t* **where** *t*: Γ *x* = *Some t* **by** *blast*
from *this in-dom c* **have** *type-wellformed t* **by** (*force simp*: *types-wellformed-def*)
hence ∀ *x*∈*vars-of-type t*. mem_1 *x* = mem_2 *x*
using *eq-*$\mathcal{C}$ **unfolding** *type-wellformed-def* **by** *blast*
hence *t-eq*: *type-max t* mem_1 = *type-max t* mem_2
unfolding *type-max-def* **using** $eval\text{-}vars\text{-}det_B$
by *fastforce*
with *in-dom t* **have** *to-total* Γ *x* = *t*
by (*auto simp*: *to-total-def*)

with *t-eq* **have** *type-max* (*to-total* Γ *x*) *mem*$_2$ = *type-max* (*to-total* Γ *x*) *mem*$_1$
by *simp*
with *d* **have** *type-max* (*to-total* Γ *x*) *mem*$_1$ = *Low* **by** *simp*
with *a* **show** *?thesis* **by** (*metis sym*)
next
assume *x* ∉ *dom* Γ
hence *to-total* Γ *x* = *dma-type x*
by (*auto simp*: *to-total-def*)
with *dma-type-eq* **have** *type-max* (*to-total* Γ *x*) *mem*$_2$ = *type-max* (*to-total* Γ *x*) *mem*$_1$ **by** *simp*
with *d* **have** *type-max* (*to-total* Γ *x*) *mem*$_1$ = *Low* **by** *simp*
with *a* **show** *?thesis* **by** (*metis sym*)
qed
qed

lemma *tyenv-eq-sym*:
tyenv-wellformed mds Γ $\mathcal{S}$ *P* ⟹ *mem*$_1$ =$_\Gamma$ *mem*$_2$ ⟹ *mem*$_2$ =$_\Gamma$ *mem*$_1$
apply(*rule tyenv-eq-sym'*)
apply(*fastforce simp*: *tyenv-wellformed-def mds-consistent-def*)
apply(*simp add*: *tyenv-wellformed-def*)
by *assumption*

inductive-set $\mathcal{R}_1$ :: ('*Var*,'*BExp*) *TyEnv* ⇒ '*Var Stable* ⇒ '*BExp preds* ⇒ (('*Var*, '*AExp*, '*BExp*) *Stmt*, '*Var*, '*Val*) *LocalConf rel*
and $\mathcal{R}_1$-*abv* ::
(('*Var*, '*AExp*, '*BExp*) *Stmt*, '*Var*, '*Val*) *LocalConf* ⇒
('*Var*,'*BExp*) *TyEnv* ⇒ '*Var Stable* ⇒ '*BExp preds* ⇒
(('*Var*, '*AExp*, '*BExp*) *Stmt*, '*Var*, '*Val*) *LocalConf* ⇒
bool (- $\mathcal{R}^1$-,-,- - [*120*, *120*, *120*, *120*, *120*] *1000*)
for Γ′ :: ('*Var*,'*BExp*) *TyEnv*
and $\mathcal{S}'$ :: '*Var Stable*
and *P*′ :: '*BExp preds*
where
x $\mathcal{R}^1_{\Gamma,\mathcal{S},P}$ *y* ≡ (*x*, *y*) ∈ $\mathcal{R}_1$ Γ $\mathcal{S}$ *P* |
intro [*intro*!] : ⟦ ⊢ Γ,$\mathcal{S}$,*P* { *c* } Γ′,$\mathcal{S}'$,*P*′ ; *tyenv-wellformed mds* Γ $\mathcal{S}$ *P* ; *mem*$_1$ =$_\Gamma$ *mem*$_2$;
pred P mem$_1$; *pred P mem*$_2$; ∀ *x*∈*dom* Γ. *x*∉*mds AsmNoReadOrWrite* ⟶ *type-max* (*the* (Γ *x*)) *mem*$_1$ ≤ *dma mem*$_1$ *x*⟧ ⟹
⟨*c*, *mds*, *mem*$_1$⟩ $\mathcal{R}^1_{\Gamma',\mathcal{S}',P'}$ ⟨*c*, *mds*, *mem*$_2$⟩

inductive $\mathcal{R}_3$-*aux* :: (('*Var*, '*AExp*, '*BExp*) *Stmt*, '*Var*, '*Val*) *LocalConf* ⇒
('*Var*,'*BExp*) *TyEnv* ⇒ '*Var Stable* ⇒ '*BExp preds* ⇒ (('*Var*, '*AExp*, '*BExp*) *Stmt*, '*Var*, '*Val*) *LocalConf* ⇒
bool (- $\mathcal{R}^3$-,-,- - [*120*, *120*, *120*, *120*, *120*] *1000*)
and $\mathcal{R}_3$ :: ('*Var*,'*BExp*) *TyEnv* ⇒ '*Var Stable* ⇒ '*BExp preds* ⇒ (('*Var*, '*AExp*, '*BExp*) *Stmt*, '*Var*, '*Val*) *LocalConf rel*
where
$\mathcal{R}_3$ Γ′ $\mathcal{S}'$ *P*′ ≡ {(*lc*$_1$, *lc*$_2$). $\mathcal{R}_3$-*aux lc*$_1$ Γ′ $\mathcal{S}'$ *P*′ *lc*$_2$} |
intro$_1$ [*intro*] : ⟦ ⟨*c*$_1$, *mds*, *mem*$_1$⟩ $\mathcal{R}^1_{\Gamma,\mathcal{S},P}$ ⟨*c*$_2$, *mds*, *mem*$_2$⟩; ⊢ Γ,$\mathcal{S}$,*P* { *c* }

Γ',$\mathcal{S}'$,P' $[\![\Longrightarrow$
$\langle Seq\ c_1\ c,\ mds,\ mem_1\rangle\ \mathcal{R}^3{}_{\Gamma',\mathcal{S}',P'}\ \langle Seq\ c_2\ c,\ mds,\ mem_2\rangle$ |
intro$_3$ [*intro*] : $[\![\ \langle c_1,\ mds,\ mem_1\rangle\ \mathcal{R}^3{}_{\Gamma,\mathcal{S},P}\ \langle c_2,\ mds,\ mem_2\rangle;\ \vdash \Gamma,\mathcal{S},P\ \{\ c\ \}$
Γ',$\mathcal{S}'$,P' $[\![\Longrightarrow$
$\langle Seq\ c_1\ c,\ mds,\ mem_1\rangle\ \mathcal{R}^3{}_{\Gamma',\mathcal{S}',P'}\ \langle Seq\ c_2\ c,\ mds,\ mem_2\rangle$

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

inductive-set $\mathcal{R}$ :: (*'Var*,*'BExp*) *TyEnv* ⇒ *'Var Stable* ⇒ *'BExp preds* ⇒
((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf rel*
and $\mathcal{R}$-*abv* ::
((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf* ⇒
(*'Var*,*'BExp*) *TyEnv* ⇒ *'Var Stable* ⇒ *'BExp preds* ⇒
((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf* ⇒
bool (- $\mathcal{R}^u$-,-,- - [*120*, *120*, *120*, *120*, *120*] *1000*)
for Γ :: (*'Var*,*'BExp*) *TyEnv*
and $\mathcal{S}$ :: *'Var Stable*
and P :: *'BExp preds*
where
$x\ \mathcal{R}^u{}_{\Gamma,\mathcal{S},P}\ y \equiv (x,\ y) \in \mathcal{R}\ \Gamma\ \mathcal{S}\ P$ |
intro$_1$: $lc\ \mathcal{R}^1{}_{\Gamma,\mathcal{S},P}\ lc' \Longrightarrow (lc,\ lc') \in \mathcal{R}\ \Gamma\ \mathcal{S}\ P$ |
intro$_3$: $lc\ \mathcal{R}^3{}_{\Gamma,\mathcal{S},P}\ lc' \Longrightarrow (lc,\ lc') \in \mathcal{R}\ \Gamma\ \mathcal{S}\ P$

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

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

lemma $\mathcal{R}_1$-*mem-eq*: $\langle c_1,\ mds,\ mem_1\rangle\ \mathcal{R}^1{}_{\Gamma',\mathcal{S}',P'}\ \langle c_2,\ mds,\ mem_2\rangle \Longrightarrow mem_1 =_{mds}{}^l\ mem_2$
proof (*erule* $\mathcal{R}_1$-*elim*)
fix $\Gamma\ \mathcal{S}\ P$
assume *wf*: *tyenv-wellformed mds* $\Gamma\ \mathcal{S}\ P$
hence *mds-consistent*: *mds-consistent mds* $\Gamma\ \mathcal{S}\ P$
unfolding *tyenv-wellformed-def* **by** *blast*

assume *tyenv-eq*: $mem_1 =_\Gamma mem_2$
assume *leq*: $\forall x{\in}dom\ \Gamma.\ x \notin mds\ AsmNoReadOrWrite \longrightarrow type\text{-}max\ (the\ (\Gamma\ x))\ mem_1 \leq dma\ mem_1\ x$
assume *pred*: *pred P* mem_1

show $mem_1 =_{mds}{}^{l} mem_2$
unfolding *low-mds-eq-def*
proof(*clarify*)
fix *x*
assume *is-Low*: *dma* $mem_1\ x = Low$
assume *is-readable*: $x \in \mathcal{C} \vee x \notin mds\ AsmNoReadOrWrite$
show $mem_1\ x = mem_2\ x$
proof(*cases* $x \in dom\ \Gamma$)
assume *in-dom*: $x \in dom\ \Gamma$
with *mds-consistent* **have** $x \notin \mathcal{C}$
unfolding *mds-consistent-def* **by** *blast*
with *is-readable* **have** $x \notin mds\ AsmNoReadOrWrite$
by *blast*

with *in-dom leq* **have** *type-max* (*to-total* $\Gamma\ x$) $mem_1 \leq dma\ mem_1\ x$
unfolding *to-total-def*
by *auto*
with *is-Low* **have** *type-max* (*to-total* $\Gamma\ x$) $mem_1 = Low$
by(*simp add*: *less-eq-Sec-def*)
with *tyenv-eq* **show** *?thesis*
unfolding *tyenv-eq-def* **by** *blast*
next
assume *nin-dom*: $x \notin dom\ \Gamma$
with *is-Low* **have** *type-max* (*to-total* $\Gamma\ x$) $mem_1 = Low$
unfolding *to-total-def*
by *simp*
with *tyenv-eq* **show** *?thesis*
unfolding *tyenv-eq-def* **by** *blast*
qed
qed
qed

lemma $\mathcal{R}_1$*-dma-eq*:
$\langle c_1,\ mds,\ mem_1\rangle\ {\mathcal{R}^1}_{\Gamma',\mathcal{S}',P'}\ \langle c_2,\ mds,\ mem_2\rangle \Longrightarrow dma\ mem_1 = dma\ mem_2$
apply(*drule* $\mathcal{R}_1$*-mem-eq*)
apply(*erule low-mds-eq-dma*)
done

lemma *bisim-simple-*$\mathcal{R}_1$:
$\langle c,\ mds,\ mem\rangle\ {\mathcal{R}^1}_{\Gamma,\mathcal{S},P}\ \langle c',\ mds',\ mem'\rangle \Longrightarrow c = c'$
apply(*cases rule*: $\mathcal{R}_1$*.cases*, *simp*+)
done

lemma *bisim-simple-*$\mathcal{R}_3$:
lc ${\mathcal{R}^3}_{\Gamma,\mathcal{S},P}$ *lc'* $\Longrightarrow$ (*fst* (*fst lc*)) = (*fst* (*fst lc'*))
apply(*induct rule*: $\mathcal{R}_3$*-aux.induct*)
using *bisim-simple-*$\mathcal{R}_1$ **apply** *clarsimp*
apply *simp*
done

lemma *bisim-simple-*$\mathcal{R}_u$:
lc ${\mathcal{R}^u}_{\Gamma,\mathcal{S},P}$ *lc'* $\Longrightarrow$ (*fst* (*fst lc*)) = (*fst* (*fst lc'*))
apply(*induct rule*: $\mathcal{R}$*.induct*)
apply *clarsimp*
apply(*cases rule*: $\mathcal{R}_1$*.cases*, *simp+*)
apply(*cases rule*: $\mathcal{R}_3$*-aux.cases*, *simp+*)
apply *blast*
using *bisim-simple-*$\mathcal{R}_3$ **apply** *clarsimp*
done

lemma $\mathcal{C}$*-eq-type-max-eq*:
assumes *wf*: *type-wellformed t*
assumes $\mathcal{C}$*-eq*: $\forall\, x{\in}\mathcal{C}.\ mem_1\ x = mem_2\ x$
shows *type-max t* mem_1 = *type-max t* mem_2
proof −
have $\forall\, x{\in}$*vars-of-type t*. $mem_1\ x = mem_2\ x$
using *wf* $\mathcal{C}$*-eq* **unfolding** *type-wellformed-def* **by** *blast*
thus *?thesis*
unfolding *type-max-def* **using** *eval-vars-det*$_B$ **by** *fastforce*
qed

lemma *vars-of-type-eq-type-max-eq*:
assumes *mem-eq*: $\forall\, x{\in}$*vars-of-type t*. $mem_1\ x = mem_2\ x$
shows *type-max t* mem_1 = *type-max t* mem_2
proof −
from *assms* **show** *?thesis*
unfolding *type-max-def* **using** *eval-vars-det*$_B$ **by** *fastforce*
qed

lemma $\mathcal{R}_1$*-sym*: *sym* ($\mathcal{R}_1\ \Gamma'\ \mathcal{S}'\ P'$)
unfolding *sym-def*
proof *clarsimp*
fix *c mds mem c' mds' mem'*
assume *in-*$\mathcal{R}_1$: $\langle c, mds, mem\rangle$ ${\mathcal{R}^1}_{\Gamma',\mathcal{S}',P'}$ $\langle c', mds', mem'\rangle$
then obtain Γ $\mathcal{S}$ *P* **where**
stuff: $c' = c$ $mds' = mds$ $\vdash \Gamma,\mathcal{S},P\ \{c\}\ \Gamma',\mathcal{S}',P'$ *tyenv-wellformed mds* Γ $\mathcal{S}$ *P*
mem $=_\Gamma$ *mem' pred P mem pred P mem'*
$\forall\, x{\in}$*dom* Γ. $x \notin$ *mds AsmNoReadOrWrite* $\longrightarrow$ *type-max* (*the* (Γ *x*)) *mem* $\leq$ *dma*

mem x
using $\mathcal{R}_1$*-elim'* **by** *blast+*
from *stuff* **have** *stuff'*: $mem' =_\Gamma mem$
by (*metis tyenv-eq-sym*)

have $\forall x{\in}dom\ \Gamma.\ x \notin mds\ AsmNoReadOrWrite \longrightarrow type\text{-}max\ (the\ (\Gamma\ x))\ mem' \leq dma\ mem'\ x$
proof −
from *in-*$\mathcal{R}_1$ **have** *dma mem = dma mem'*
using $\mathcal{R}_1$*-dma-eq stuff* **by** *metis*
moreover have $\forall x{\in}dom\ \Gamma.$ *type-max* (*the* (Γ *x*)) *mem* = *type-max* (*the* (Γ *x*)) *mem'*
proof
fix *x*
assume $x \in dom\ \Gamma$
hence *type-wellformed* (*the* (Γ *x*))
using ‹*tyenv-wellformed mds* Γ $\mathcal{S}$ *P*›
by(*auto simp*: *tyenv-wellformed-def types-wellformed-def*)
moreover have $\forall x{\in}\mathcal{C}.$ *mem x* = *mem' x*
using *in-*$\mathcal{R}_1$ $\mathcal{R}_1$*-mem-eq* $\mathcal{C}$*-Low stuff*
unfolding *low-mds-eq-def* **by** *auto*
ultimately
show *type-max* (*the* (Γ *x*)) *mem* = *type-max* (*the* (Γ *x*)) *mem'*
using $\mathcal{C}$*-eq-type-max-eq* **by** *blast*
qed
ultimately show *?thesis*
using *stuff*(*8*) **by** *fastforce*
qed
with *stuff stuff'*
show $\langle c', mds', mem'\rangle\ {\mathcal{R}^1}_{\Gamma',\mathcal{S}',P'}\ \langle c, mds, mem\rangle$
by (*metis* (*no-types*) $\mathcal{R}_1$*.intro*)
qed

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

lemma $\mathcal{R}$*-mds* [*simp*]: $\langle c_1, mds, mem_1\rangle\ {\mathcal{R}^u}_{\Gamma,\mathcal{S},P}\ \langle c_2, mds', mem_2\rangle \Longrightarrow mds = mds'$

apply (*rule* $\mathcal{R}$*-elim'*)
apply (*auto*)
apply (*metis* $\mathcal{R}_1$*-elim'*)
apply (*insert* $\mathcal{R}_3$*-elim'*)
by *blast*

lemma $\mathcal{R}$*-sym*: *sym* ($\mathcal{R}$ Γ $\mathcal{S}$ P)
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 $\mathcal{S}$ P
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 $\mathcal{S}$ P
using $\mathcal{R}$.*intro*$_1$ [*of* Γ $\mathcal{S}$ P] **and** $\mathcal{R}$.*intro*$_3$ [*of* - Γ $\mathcal{S}$ P]
using $\mathcal{R}_1$*-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' $\mathcal{S}'$ P')
unfolding *closed-glob-consistent-def*
proof (*clarify*)
fix c_1 *mds* mem_1 c_2 mem_2 A
assume *R1*: $\langle c_1, mds, mem_1\rangle$ $\mathcal{R}^1{}_{\Gamma',\mathcal{S}',P'}$ $\langle c_2, mds, mem_2\rangle$
hence [*simp*]: $c_2 = c_1$ **by** *blast*
assume *A-updates-vars*: $\forall x.$ *case* A x *of None* $\Rightarrow$ *True* | *Some* (v, v') $\Rightarrow$ mem_1 $x \neq v \vee mem_2\ x \neq v' \longrightarrow \neg$ *var-asm-not-written mds x*
assume *A-updates-dma*: $\forall x.$ *dma* mem_1 [$\|_1$ A] $x \neq$ *dma* mem_1 $x \longrightarrow \neg$ *var-asm-not-written mds x*
assume *A-updates-sec*: $\forall x.$ *dma* mem_1 [$\|_1$ A] $x =$ *Low* $\wedge$ ($x \notin$ *mds AsmNoReadOrWrite* $\vee$ $x \in \mathcal{C}$) $\longrightarrow$ mem_1 [$\|_1$ A] $x = mem_2$ [$\|_2$ A] x
from *R1* **obtain** Γ $\mathcal{S}$ P **where** Γ*-props*: $\vdash$ Γ,$\mathcal{S}$,P { c_1 } Γ',$\mathcal{S}'$,P' $mem_1 =_\Gamma mem_2$ *tyenv-wellformed mds* Γ $\mathcal{S}$ P
pred P mem_1 *pred* P mem_2
$\forall x \in dom\ \Gamma.\ x \notin mds\ AsmNoReadOrWrite \longrightarrow$ *type-max* (*the* (Γ x)) $mem_1 \leq$ *dma* mem_1 x
by *force*

from Γ*-props*(*3*) **have** *stable-not-written*: $\forall x.$ *stable* $\mathcal{S}$ $x \longrightarrow$ *var-asm-not-written mds x*
by(*auto simp*: *tyenv-wellformed-def mds-consistent-def stable-def var-asm-not-written-def*)
with *A-updates-vars* **have** *stable-unchanged*$_1$: $\forall x.$ *stable* $\mathcal{S}$ $x \longrightarrow$ (mem_1 [$\|_1$ A]) $x = mem_1\ x$ **and**
stable-unchanged$_2$: $\forall x.$ *stable* $\mathcal{S}$ $x \longrightarrow$ (mem_2 [$\|_2$ A]) $x = mem_2\ x$

by(*auto simp*: *apply-adaptation-def split*: *option.splits*)

from *stable-not-written A-updates-dma*
have *stable-unchanged-dma*$_1$: $\forall x.$ *stable* $\mathcal{S}$ $x \longrightarrow$ *dma* (*mem*$_1$ [$\|_1$ *A*]) x = *dma* *mem*$_1$ x
by(*blast*)

have *tyenv-eq'*: *mem*$_1$ [$\|_1$ *A*] $=_\Gamma$ *mem*$_2$ [$\|_2$ *A*]
proof(*clarsimp simp*: *tyenv-eq-def*)
fix x
assume *a*: *type-max* (*to-total* Γ x) *mem*$_1$ [$\|_1$ *A*] = *Low*
show *mem*$_1$ [$\|_1$ *A*] x = *mem*$_2$ [$\|_2$ *A*] x
proof(*cases* $x \in$ *dom* Γ)
assume *in-dom*: $x \in$ *dom* Γ
with Γ-*props*(*3*) **have** *var-asm-not-written mds x*
by(*auto simp*: *tyenv-wellformed-def mds-consistent-def var-asm-not-written-def stable-def*)
hence [*simp*]: *mem*$_1$ [$\|_1$ *A*] x = *mem*$_1$ x **and** [*simp*]: *mem*$_2$ [$\|_2$ *A*] x = *mem*$_2$ x
using *A-updates-vars* **by**(*auto simp*: *apply-adaptation-def split*: *option.splits*)
from *in-dom a* **obtain** t_x **where** Γ_x: Γ x = *Some* t_x **and** t_x-*Low'*: *type-max* t_x (*mem*$_1$ [$\|_1$ *A*]) = *Low*
by(*auto simp*: *to-total-def*)
have t_x-*unchanged*: *type-max* t_x (*mem*$_1$ [$\|_1$ *A*]) = *type-max* t_x *mem*$_1$
proof −
from Γ_x Γ-*props*(*3*) **have** t_x-*stable*: *type-stable* $\mathcal{S}$ t_x **and** t_x-*wellformed*: *type-wellformed* t_x
by(*force simp*: *tyenv-wellformed-def types-stable-def types-wellformed-def*)+
from t_x-*stable* t_x-*wellformed stable-unchanged*$_1$ **show** *?thesis*
using *type-stable-is-sufficient'*
by *blast*
qed
with t_x-*Low'* **have** t_x-*Low*: *type-max* t_x *mem*$_1$ = *Low* **by** *simp*
with Γ_x Γ-*props*(*2*) **have** *mem*$_1$ x = *mem*$_2$ x
by(*force simp*: *tyenv-eq-def to-total-def split*: *if-splits*)
thus *?thesis* **by** *simp*
next
assume *nin-dom*: $x \notin$ *dom* Γ
with *a* **have** *is-Low'*: *dma* (*mem*$_1$[$\|_1$ *A*]) x = *Low*
by(*simp add*: *to-total-def*)
show *?thesis*
proof(*cases* $x \notin$ *mds AsmNoReadOrWrite* $\vee$ $x \in \mathcal{C}$)
assume $x \notin$ *mds AsmNoReadOrWrite* $\vee$ $x \in \mathcal{C}$
with *is-Low'* **show** *?thesis*
using *A-updates-sec* **by** *blast*
next
assume $\neg$ ($x \notin$ *mds AsmNoReadOrWrite* $\vee$ $x \in \mathcal{C}$)
hence $x \in$ *mds AsmNoReadOrWrite* **and** $x \notin \mathcal{C}$
by *auto*

```
      with nin-dom Γ-props(3) have False
        by(auto simp: tyenv-wellformed-def mds-consistent-def stable-def)
      thus ?thesis by blast
    qed
  qed
qed

have sec': ∀x∈dom Γ. x ∉ mds AsmNoReadOrWrite ⟶ type-max (the (Γ x))
(mem₁ [||₁ A]) ≤ dma (mem₁ [||₁ A]) x
proof(intro ballI impI)
  fix x
  assume readable: x ∉ mds AsmNoReadOrWrite
  assume in-dom: x ∈ dom Γ
  with Γ-props(3) have var-asm-not-written mds x
   by(auto simp: tyenv-wellformed-def mds-consistent-def var-asm-not-written-def
stable-def)
  hence [simp]: dma mem₁ [||₁ A] x = dma mem₁ x
    using A-updates-dma by(auto simp: apply-adaptation-def split: option.splits)
  from in-dom obtain tₓ where Γₓ: Γ x = Some tₓ
    by(auto simp: to-total-def)
  have tₓ-unchanged: type-max tₓ (mem₁ [||₁ A]) = type-max tₓ mem₁
  proof −
      from Γₓ Γ-props(3) have tₓ-stable: type-stable 𝒮 tₓ and tₓ-wellformed:
type-wellformed tₓ
      by(force simp: tyenv-wellformed-def types-stable-def types-wellformed-def)+
    from tₓ-stable tₓ-wellformed stable-unchanged₁ show ?thesis
      using type-stable-is-sufficient'
      by blast
  qed
  with Γₓ have [simp]:type-max (the (Γ x)) (mem₁ [||₁ A]) = type-max (the (Γ
x)) mem₁
    by simp
  show type-max (the (Γ x)) mem₁ [||₁ A] ≤ dma mem₁ [||₁ A] x
    apply simp
    using in-dom readable Γ-props by metis
qed

from stable-unchanged₁ stable-unchanged₂ Γ-props(3) have ∀p∈P. evᴮ (mem₁
[||₁ A]) p = evᴮ mem₁ p ∧ evᴮ (mem₂ [||₂ A]) p = evᴮ mem₂ p
  apply(intro ballI)
  apply(rule conjI)
  by(rule eval-vars-detᴮ,force simp: tyenv-wellformed-def mds-consistent-def sta-
ble-def)+

hence pred P (mem₁ [||₁ A]) = pred P mem₁ and
      pred P (mem₂ [||₂ A]) = pred P mem₂
  by(simp add: pred-def)+

with Γ-props tyenv-eq' sec'
```

show $\langle c_1,\ mds,\ mem_1\ [\|_1\ A]\rangle\ \mathcal{R}^1{}_{\Gamma',\mathcal{S}',P'}\ \langle c_2,\ mds,\ mem_2\ [\|_2\ A]\rangle$
by *auto*
qed

lemma $\mathcal{R}_3$*-closed-glob-consistent*:
closed-glob-consistent $(\mathcal{R}_3\ \Gamma'\ \mathcal{S}'\ P')$
unfolding *closed-glob-consistent-def*
proof(*clarsimp*)
fix c_1 *mds* mem_1 c_2 mem_2 *A*
assume *in-*$\mathcal{R}_3$: $\langle c_1,\ mds,\ mem_1\rangle\ \mathcal{R}^3{}_{\Gamma',\mathcal{S}',P'}\ \langle c_2,\ mds,\ mem_2\rangle$
assume *A-modifies-vars*: $\forall x.$ *case A x of None* $\Rightarrow$ *True* | *Some* $(v,\ v') \Rightarrow mem_1\ x \neq v \vee mem_2\ x \neq v' \longrightarrow \neg$ *var-asm-not-written mds x*
assume *A-modifies-dma*: $\forall x.$ *dma* $mem_1\ [\|_1\ A]\ x \neq$ *dma* $mem_1\ x \longrightarrow \neg$ *var-asm-not-written mds x*
assume *A-modifies-sec*: $\forall x.$ *dma* $mem_1\ [\|_1\ A]\ x =$ *Low* $\wedge$ ($x \in$ *mds AsmNoReadOrWrite* $\longrightarrow x \in \mathcal{C}) \longrightarrow mem_1\ [\|_1\ A]\ x = mem_2\ [\|_2\ A]\ x$

define lc_1 **where** $lc_1 \equiv \langle c_1,\ mds,\ mem_1\rangle$
define lc_2 **where** $lc_2 \equiv \langle c_2,\ mds,\ mem_2\rangle$
from lc_1*-def* lc_2*-def in-*$\mathcal{R}_3$ **have** $lc_1\ \mathcal{R}^3{}_{\Gamma',\mathcal{S}',P'}\ lc_2$ **by** *simp*
from *this* lc_1*-def* lc_2*-def A-modifies-vars A-modifies-dma A-modifies-sec*
show $\langle c_1,\ mds,\ mem_1\ [\|_1\ A]\rangle\ \mathcal{R}^3{}_{\Gamma',\mathcal{S}',P'}\ \langle c_2,\ mds,\ mem_2\ [\|_2\ A]\rangle$
proof(*induct arbitrary*: c_1 *mds* mem_1 c_2 *mds* mem_2 *rule*: $\mathcal{R}_3$*-aux.induct*)
case ($intro_1$ c_1 *mds* mem_1 Γ $\mathcal{S}$ *P* c_2 mem_2 *c* Γ' $\mathcal{S}'$ P')
show *?case*
apply (*rule* $\mathcal{R}_3$*-aux.intro*$_1$[*OF - intro*$_1$(*2*)])
using $\mathcal{R}_1$*-closed-glob-consistent intro*$_1$
unfolding *closed-glob-consistent-def* **by** *blast*
next
case ($intro_3$ c_1 *mds* mem_1 Γ $\mathcal{S}$ *P* c_2 mem_2 *c* Γ' $\mathcal{S}'$ P')
show *?case*
apply(*rule* $\mathcal{R}_3$*-aux.intro*$_3$)
apply(*rule intro*$_3$(*2*))
using *intro*$_3$ **apply** *simp+*
done
qed
qed

lemma $\mathcal{R}$*-closed-glob-consistent*: *closed-glob-consistent* $(\mathcal{R}\ \Gamma'\ \mathcal{S}'\ P')$
unfolding *closed-glob-consistent-def*
proof (*clarify*, *erule* $\mathcal{R}$*-elim*, *simp-all*)
fix c_1 *mds* mem_1 c_2 mem_2 *A*
assume *R1*: $\langle c_1,\ mds,\ mem_1\rangle\ \mathcal{R}^1{}_{\Gamma',\mathcal{S}',P'}\ \langle c_2,\ mds,\ mem_2\rangle$
and *A-modifies-vars*: $\forall x.$ *case A x of None* $\Rightarrow$ *True* | *Some* $(v,\ v') \Rightarrow mem_1\ x \neq v \vee mem_2\ x \neq v' \longrightarrow \neg$ *var-asm-not-written mds x*
and *A-modifies-dma*: $\forall x.$ *dma* $(mem_1\ [\|_1\ A])\ x \neq$ *dma* $mem_1\ x \longrightarrow \neg$ *var-asm-not-written*

mds x
and *A-modifies-sec*: $\forall x.$ *dma* mem_1 $[\|_1$ *A*$]$ *x* = *Low* $\wedge$ (*x* $\in$ *mds AsmNoReadOrWrite* $\longrightarrow$ *x* $\in \mathcal{C}$) $\longrightarrow$ mem_1 $[\|_1$ *A*$]$ *x* = mem_2 $[\|_2$ *A*$]$ *x*
show
$\langle c_1,$ *mds*, mem_1 $[\|_1$ *A*$]\rangle$ $\mathcal{R}^u{}_{\Gamma',\mathcal{S}',P'}$ $\langle c_2,$ *mds*, mem_2 $[\|_2$ *A*$]\rangle$
apply(*rule intro*$_1$)
apply *clarify*
using $\mathcal{R}_1$*-closed-glob-consistent* **unfolding** *closed-glob-consistent-def*
using *R1 A-modifies-vars A-modifies-dma A-modifies-sec*
by *blast*
next
fix c_1 *mds* mem_1 c_2 mem_2 *x A*
assume *R3*: $\langle c_1,$ *mds*, $mem_1\rangle$ $\mathcal{R}^3{}_{\Gamma',\mathcal{S}',P'}$ $\langle c_2,$ *mds*, $mem_2\rangle$
and *A-modifies-vars*: $\forall x.$ *case A x of None* $\Rightarrow$ *True* | *Some* $(v,\ v')$ $\Rightarrow$ mem_1 *x* $\neq$ *v* $\vee$ mem_2 *x* $\neq$ v' $\longrightarrow$ $\neg$ *var-asm-not-written mds x*
and *A-modifies-dma*: $\forall x.$ *dma* $(mem_1$ $[\|_1$ *A*$])$ *x* $\neq$ *dma* mem_1 *x* $\longrightarrow$ $\neg$ *var-asm-not-written mds x*
and *A-modifies-sec*: $\forall x.$ *dma* mem_1 $[\|_1$ *A*$]$ *x* = *Low* $\wedge$ (*x* $\in$ *mds AsmNoReadOrWrite* $\longrightarrow$ *x* $\in \mathcal{C}$) $\longrightarrow$ mem_1 $[\|_1$ *A*$]$ *x* = mem_2 $[\|_2$ *A*$]$ *x*
show $\langle c_1,$ *mds*, mem_1 $[\|_1$ *A*$]\rangle$ $\mathcal{R}^u{}_{\Gamma',\mathcal{S}',P'}$ $\langle c_2,$ *mds*, mem_2 $[\|_2$ *A*$]\rangle$
apply(*rule intro*$_3$)
using $\mathcal{R}_3$*-closed-glob-consistent* **unfolding** *closed-glob-consistent-def*
using *R3 A-modifies-vars A-modifies-dma A-modifies-sec*
by *blast*
qed

lemma *mode-update-add-anno*:
mds-consistent mds Γ $\mathcal{S}$ *P* $\Longrightarrow$
mds-consistent (*update-modes upd mds*)
(Γ $\oplus_{\mathcal{S}}$ *upd*)
(*add-anno-stable* $\mathcal{S}$ *upd*)
(*P* |‘ {*v. stable* (*add-anno-stable* $\mathcal{S}$ *upd*) *v*})
apply(*case-tac upd*)
apply(*rename-tac v m*)
apply(*case-tac m*)
apply((*clarsimp simp*: *add-anno-def mds-consistent-def stable-def restrict-preds-to-vars-def* | *safe* | *fastforce*)+)[4]
apply(*rename-tac v m*)
apply(*case-tac m*)
apply((*clarsimp simp*: *add-anno-def mds-consistent-def stable-def restrict-preds-to-vars-def* | *safe* | *fastforce*)+)[4]
done

lemma *add-anno-acq-GuarNoReadOrWrite* [*simp*]:
Γ $\oplus_{\mathcal{S}}$ (*v* $+=_m$ *GuarNoReadOrWrite*) = Γ
apply(*clarsimp simp*: *add-anno-def to-total-def restrict-map-def*)

apply(*rule ext*)
apply(*clarsimp* | *safe*)+
by (*metis option.collapse prod.collapse*)

lemma *add-anno-rel-GuarNoReadOrWrite* [*simp*]:
$\Gamma \oplus_{\mathcal{S}} (v -=_m \mathit{GuarNoReadOrWrite}) = \Gamma$
apply(*clarsimp simp*: *add-anno-def to-total-def restrict-map-def*)
apply(*rule ext*)
apply(*clarsimp* | *safe*)+
by (*metis option.collapse*)

lemma *add-anno-acq-GuarNoWrite* [*simp*]:
$\Gamma \oplus_{\mathcal{S}} (v +=_m \mathit{GuarNoWrite}) = \Gamma$
apply(*clarsimp simp*: *add-anno-def to-total-def restrict-map-def*)
apply(*rule ext*)
apply(*clarsimp* | *safe*)+
by (*metis option.collapse prod.collapse*)

lemma *add-anno-rel-GuarNoWrite* [*simp*]:
$\Gamma \oplus_{\mathcal{S}} (v -=_m \mathit{GuarNoWrite}) = \Gamma$
apply(*clarsimp simp*: *add-anno-def to-total-def restrict-map-def*)
apply(*rule ext*)
apply(*clarsimp* | *safe*)+
by (*metis option.collapse*)

lemma *dom-add-anno-rel*:
$\forall x{\in}\mathit{dom}\ (\Gamma \oplus_{\mathcal{S}} (v -=_m m)).\ (\Gamma \oplus_{\mathcal{S}} (v -=_m m))\ x = \Gamma\ x$
apply(*clarsimp simp*: *add-anno-def to-total-def restrict-map-def split*: *if-splits*)
apply(*case-tac m*)
apply(*auto split*: *if-splits*)
done

lemma *types-wellformed-mode-update*:
types-wellformed $\Gamma \Longrightarrow$
types-wellformed $(\Gamma \oplus_{\mathcal{S}} \mathit{upd})$
apply(*clarsimp simp*: *types-wellformed-def*)
apply(*case-tac upd*)
apply(*rename-tac v t v' m*)
apply(*case-tac m*)
apply *clarsimp*
apply(*case-tac* $v' \in \mathit{dom}\ \Gamma \vee v' \in \mathcal{C}$)
apply(*clarsimp*, *force*)
apply(*simp split*: *if-splits*)
apply(*drule sym*, *fastforce*)
apply(*clarsimp* | *force*)+
apply(*case-tac* $v' \in \mathit{dom}\ \Gamma \vee v' \in \mathcal{C}$)
apply *clarsimp*
apply *force*
apply(*simp split*: *if-splits*)

apply(*drule sym*, *fastforce*)
apply(*clarsimp* | *force*)+
using *dom-add-anno-rel*[*THEN bspec*, *OF domI*]
apply (*metis domI option.sel*)
done

lemma *types-stable-mode-update*:
types-stable Γ $\mathcal{S}$ $\Longrightarrow$ *types-wellformed* Γ $\Longrightarrow$ *anno-type-stable* Γ $\mathcal{S}$ *upd*
$\Longrightarrow$ *types-stable* (Γ $\oplus_{\mathcal{S}}$ *upd*) (*add-anno-stable* $\mathcal{S}$ *upd*)
apply(*clarsimp simp*: *types-stable-def*)
apply(*rename-tac x c f C*)
apply(*case-tac upd*)
apply(*rename-tac v m*)
apply(*case-tac m*)
apply *clarsimp*
apply(*case-tac v* $\in$ *dom* Γ $\vee$ *v* $\in$ $\mathcal{C}$)
apply *clarsimp*
apply(*force simp*: *stable-def*)
apply(*simp split*: *if-splits*)
using $\mathcal{C}$*-vars-correct*
apply(*fastforce simp*: *anno-type-stable-def stable-def*)
apply(*force simp*: *stable-def*)
apply *clarsimp*
apply(*case-tac v* $\in$ *dom* Γ $\vee$ *v* $\in$ $\mathcal{C}$)
apply *clarsimp*
apply(*force simp*: *stable-def*)
apply(*simp split*: *if-splits*)
using $\mathcal{C}$*-vars-correct*
apply(*fastforce simp*: *anno-type-stable-def stable-def*)
apply(*force simp*: *stable-def*)
apply(*force simp*: *stable-def*)
apply(*force simp*: *stable-def*)
apply(*rename-tac v m*)
apply(*subgoal-tac* Γ *x* = *Some* (*C*))
prefer *2*
using *dom-add-anno-rel*
apply (*metis domI*)
apply(*case-tac m*)
apply(*clarsimp simp*: *anno-type-stable-def split*: *if-splits*)
apply(*clarsimp simp*: *stable-def types-wellformed-def type-wellformed-def*)
using $\mathcal{C}$*-vars-correct*
apply (*metis* (*mono-tags*, *lifting*) *UN-I contra-subsetD domI option.sel*)
apply(*clarsimp simp*: *stable-def types-wellformed-def type-wellformed-def*)
using $\mathcal{C}$*-vars-correct*
apply (*metis* (*mono-tags*, *lifting*) *UN-I contra-subsetD domI option.sel*)
apply(*clarsimp simp*: *anno-type-stable-def split*: *if-splits*)
apply(*clarsimp simp*: *stable-def types-wellformed-def type-wellformed-def*)
apply (*metis* (*mono-tags*, *lifting*) *UN-I contra-subsetD domI option.sel*)

apply(*clarsimp simp*: *stable-def*)
apply (*metis* (*no-types*, *lifting*) *domI option.sel snd-conv subsetD type-wellformed-def types-wellformed-def*)
apply(*clarsimp simp*: *anno-type-stable-def split*: *if-splits*)
apply(*clarsimp simp*: *stable-def*)
apply (*metis* (*no-types*, *lifting*) *domI option.sel snd-conv subsetD type-wellformed-def types-wellformed-def*)
apply(*clarsimp simp*: *stable-def*)
apply (*metis* (*no-types*, *lifting*) *domI option.sel snd-conv subsetD type-wellformed-def types-wellformed-def*)
done

lemma *tyenv-wellformed-mode-update*:
tyenv-wellformed mds Γ $\mathcal{S}$ *P* $\Longrightarrow$ *anno-type-stable* Γ $\mathcal{S}$ *upd* $\Longrightarrow$
tyenv-wellformed (*update-modes upd mds*)
(Γ $\oplus_{\mathcal{S}}$ *upd*)
(*add-anno-stable* $\mathcal{S}$ *upd*)
(*P* |‘ {*v*. *stable* (*add-anno-stable* $\mathcal{S}$ *upd*) *v*})
apply(*clarsimp simp*: *tyenv-wellformed-def*)
apply(*rule conjI*)
apply(*blast intro*: *mode-update-add-anno*)
apply(*rule conjI*)
apply(*blast intro*: *types-wellformed-mode-update*)
apply(*blast intro*: *types-stable-mode-update*)
done

lemma *stop-cxt* :
$[\![\vdash \Gamma,\mathcal{S},P \{ c \} \Gamma',\mathcal{S}',P' ; c = Stop]\!] \Longrightarrow$
context-equiv Γ *P* $\Gamma' \wedge \mathcal{S}' = \mathcal{S} \wedge P \vdash P' \wedge$ ($\forall$ *mds*. *tyenv-wellformed mds* Γ $\mathcal{S}$ *P* $\longrightarrow$ *tyenv-wellformed mds* Γ' $\mathcal{S}$ *P'*)
apply (*induct rule*: *has-type.induct*)
apply (*auto intro*: *context-equiv-trans context-equiv-entailment pred-entailment-trans*)
done

lemma *tyenv-wellformed-preds-update*:
P' = *P''* |‘ {*v*. *stable* $\mathcal{S}$ *v*} $\Longrightarrow$
tyenv-wellformed mds Γ $\mathcal{S}$ *P* $\Longrightarrow$ *tyenv-wellformed mds* Γ $\mathcal{S}$ *P'*
apply(*clarsimp simp*: *tyenv-wellformed-def*)
apply(*clarsimp simp*: *mds-consistent-def*)
apply(*auto simp*: *restrict-preds-to-vars-def add-pred-def split*: *if-splits*)
done

lemma *mds-consistent-preds-tyenv-update*:
P' = *P''* |‘ {*v*. *stable* $\mathcal{S}$ *v*} $\Longrightarrow$ *v* $\in$ *dom* Γ $\Longrightarrow$
mds-consistent mds Γ $\mathcal{S}$ *P* $\Longrightarrow$ *mds-consistent mds* (Γ(*v* $\mapsto$ *t*)) $\mathcal{S}$ *P'*
apply(*clarsimp simp*: *mds-consistent-def*)

apply(*auto simp*: *restrict-preds-to-vars-def add-pred-def split*: *if-splits*)
done

lemma *pred-preds-update*:
assumes *mem′-def*: $mem' = mem\ (x := ev_A\ mem\ e)$
assumes *P′-def*: $P' = (assign\text{-}post\ P\ x\ e)\ |`\ \{v.\ stable\ \mathcal{S}\ v\}$
assumes *pred-P*: *pred P mem*
shows *pred P′ mem′*
proof −
from *P′-def* **have** *P′-def′*: $P' \subseteq assign\text{-}post\ P\ x\ e$
by(*auto simp*: *restrict-preds-to-vars-def add-pred-def split*: *if-splits*)
have *pred* (*assign-post P x e*) *mem′*
using *assign-post-valid pred-P mem′-def* **by** *blast*
with *P′-def′* **show** *?thesis*
unfolding *pred-def* **by** *blast*
qed

lemma *types-wellformed-update*:
types-wellformed $\Gamma \Longrightarrow$ *type-wellformed* $t \Longrightarrow$ *types-wellformed* $(\Gamma(x \mapsto t))$
by(*auto simp*: *types-wellformed-def*)

lemma *types-stable-update*:
types-stable $\Gamma\ \mathcal{S} \Longrightarrow$ *type-stable* $\mathcal{S}\ t \Longrightarrow$ *types-stable* $(\Gamma(x \mapsto t))\ \mathcal{S}$
by(*auto simp*: *types-stable-def*)

lemma *tyenv-wellformed-sub*:
$[\![P_1 \subseteq P_2;$
$\Gamma_2 = \Gamma_1;$ *tyenv-wellformed mds* $\Gamma_2\ \mathcal{S}\ P_2]\!] \Longrightarrow$
tyenv-wellformed mds $\Gamma_1\ \mathcal{S}\ P_1$
apply(*clarsimp simp*: *tyenv-wellformed-def* | *safe*)+
apply(*fastforce simp*: *mds-consistent-def*)
done

abbreviation
tyenv-sec :: *'Var Mds* ⇒ (*'Var*,*'BExp*) *TyEnv* ⇒ (*'Var*,*'Val*) *Mem* ⇒ *bool*
where
tyenv-sec mds Γ *mem* ≡ ∀ *x*∈*dom* Γ. *x*∉*mds AsmNoReadOrWrite* ⟶ *type-max* (*the* (Γ *x*)) *mem* ≤ *dma mem x*

lemma *tyenv-sec-mode-update*:
$(\forall x.\ \ (to\text{-}total\ \Gamma\ x) \leq_{:P''} (to\text{-}total\ \Gamma''\ x)) \Longrightarrow pred\ P''\ mem \Longrightarrow \mathcal{S} = (mds\ AsmNoWrite,\ mds\ AsmNoReadOrWrite) \Longrightarrow$
$anno\text{-}type\text{-}sec\ \Gamma\ \mathcal{S}\ P\ upd \Longrightarrow \mathcal{S}'' = add\text{-}anno\text{-}stable\ \mathcal{S}\ upd \Longrightarrow (\forall p{\in}P.\ \forall v{\in}bexp\text{-}vars\ p.\ stable\ \mathcal{S}\ v) \Longrightarrow$
$P'' = P\ |`\ \{v.\ stable\ \mathcal{S}''\ v\} \Longrightarrow$
$\Gamma'' = \Gamma \oplus_{\mathcal{S}} upd \Longrightarrow tyenv\text{-}sec\ mds\ \Gamma\ mem \Longrightarrow tyenv\text{-}sec\ (update\text{-}modes\ upd\ mds)\ \Gamma''\ mem$
apply(*case-tac upd*)

apply(*rename-tac v m*)
apply(*case-tac m*)
apply(*auto simp*: *add-anno-def to-total-def*)[*4*]
apply(*rename-tac v m*)
apply(*case-tac m*)
apply(*subgoal-tac* $v \in$ *mds AsmNoWrite* $\longrightarrow$ $P'' = P$)
by(*auto simp*: *add-anno-def to-total-def dest*: *subtype-sound split*: *if-splits simp*: *anno-type-sec-def restrict-preds-to-vars-def stable-def*)

lemma *tyenv-sec-eq*:
$\forall x{\in}\mathcal{C}$. *mem* x = *mem'* x $\Longrightarrow$ *types-wellformed* Γ $\Longrightarrow$ *tyenv-sec mds* Γ *mem* = *tyenv-sec mds* Γ *mem'*
apply(*rule ball-cong*[*OF HOL.refl*])
apply(*rename-tac x*)
apply(*rule imp-cong*[*OF HOL.refl*])
apply(*subgoal-tac type-max* (*the* (Γ x)) *mem* = *type-max* (*the* (Γ x)) *mem'*)
using *dma-*$\mathcal{C}$ **apply** *fastforce*
apply(*force intro*: $\mathcal{C}$*-eq-type-max-eq simp*: *types-wellformed-def*)
done

lemma *context-equiv-tyenv-sec*:
context-equiv Γ_2 P_2 Γ_1 $\Longrightarrow$
pred P_2 *mem* $\Longrightarrow$ *tyenv-sec mds* Γ_2 *mem* $\Longrightarrow$ *tyenv-sec mds* Γ_1 *mem*
apply(*clarsimp simp*: *context-equiv-def type-equiv-def*)
apply(*rename-tac x y*)
apply(*rule-tac y=type-max* (*the* (Γ_2 x)) *mem* **in** *order-trans*)
apply(*rule subtype-sound*[*rule-format*])
apply *force*
apply *assumption*
apply *force*
done

lemma *add-pred-entailment*:
$P +_{\mathcal{S}} p \vdash P$
apply(*rule subset-entailment*)
apply(*rule add-pred-subset*)
done

lemma *preservation-no-await*:
⟦$\vdash \Gamma,\mathcal{S},P \{ c \} \Gamma',\mathcal{S}',P'$;
$\langle c, mds, mem\rangle \rightsquigarrow \langle c', mds', mem'\rangle$;
no-await c⟧ $\Longrightarrow$
$\exists \Gamma'' \mathcal{S}'' P''.$ ($\vdash \Gamma'',\mathcal{S}'',P'' \{ c' \} \Gamma',\mathcal{S}',P'$) $\wedge$
(*tyenv-wellformed mds* Γ $\mathcal{S}$ P $\wedge$ *pred P mem* $\wedge$ *tyenv-sec mds* Γ *mem* $\longrightarrow$
tyenv-wellformed mds' Γ'' $\mathcal{S}''$ P'' $\wedge$
pred P'' *mem'* $\wedge$
tyenv-sec mds' Γ'' *mem'*)

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

case (*anno-type* Γ'' Γ $\mathcal{S}$ *upd* $\mathcal{S}''$ P'' P c_1 Γ' $\mathcal{S}'$ P')
hence *step*: $\langle c_1, \textit{update-modes upd mds}, \textit{mem}\rangle \rightsquigarrow \langle c', \textit{mds}', \textit{mem}'\rangle$
by (*metis upd-elim*)
from ‹*no-await* (c_1@[*upd*])› *no-await.cases* **have** *no-await* c_1 **by** *fast*
with *step anno-type*(*5*) **obtain** Γ''' $\mathcal{S}'''$ P''' **where**
$\vdash \Gamma''',\mathcal{S}''',P''' \{ c' \} \Gamma',\mathcal{S}',P' \wedge$
(*tyenv-wellformed* (*update-modes upd mds*) Γ'' $\mathcal{S}''$ P'' $\wedge$ *pred* P'' *mem* $\wedge$ *tyenv-sec* (*update-modes upd mds*) Γ'' *mem* $\longrightarrow$
tyenv-wellformed mds' Γ''' $\mathcal{S}'''$ P''' $\wedge$ *pred* P''' *mem'* $\wedge$ *tyenv-sec mds'* Γ''' *mem'*)
by *blast*
moreover
have *tyenv-wellformed mds* Γ $\mathcal{S}$ P $\longrightarrow$ *tyenv-wellformed* (*update-modes upd mds*) Γ'' $\mathcal{S}''$ P''
using *anno-type*
apply *auto*
by (*metis tyenv-wellformed-mode-update*)
moreover
have *pred*: *pred P mem* $\longrightarrow$ *pred* P'' *mem*
using *anno-type*
by (*auto simp*: *pred-def restrict-preds-to-vars-def*)
moreover
have *tyenv-wellformed mds* Γ $\mathcal{S}$ P $\wedge$ *pred P mem* $\wedge$ *tyenv-sec mds* Γ *mem* $\longrightarrow$ *tyenv-sec* (*update-modes upd mds*) Γ'' *mem*
apply(*rule impI*)
apply(*rule tyenv-sec-mode-update*)
using *anno-type* **apply** *fastforce*
using *anno-type pred* **apply** *fastforce*
using *anno-type* **apply** *fastforce*
using *anno-type* **apply**(*fastforce simp*: *tyenv-wellformed-def mds-consistent-def*)
using *anno-type* **apply** *fastforce*
apply(*fastforce simp*: *tyenv-wellformed-def mds-consistent-def*)
apply(*fastforce simp*: *tyenv-wellformed-def mds-consistent-def*)
using *anno-type* **apply**(*fastforce simp*: *tyenv-wellformed-def mds-consistent-def*)
by *simp*
ultimately show *?case*
by *blast*
next
case *stop-type*
with *stop-no-eval* **show** *?case* **by** *auto*
next
case *skip-type*
hence $c' = \textit{Stop} \wedge \textit{mds}' = \textit{mds} \wedge \textit{mem}' = \textit{mem}$
by (*metis skip-elim*)
thus *?case*
by (*metis stop-type*)
next

case ($assign_1$ x Γ e t P P' $\mathcal{S}$ c' mds)
hence upd: $c' = Stop \wedge mds' = mds \wedge mem' = mem\ (x := ev_A\ mem\ e)$
by ($metis$ $assign\text{-}elim$)
from $assign_1(2)$ upd **have** $\mathcal{C}$-eq: $\forall x{\in}\mathcal{C}.\ mem\ x = mem'\ x$
by $auto$
from upd **have** $\vdash \Gamma,\mathcal{S},P'\ \{c'\}\ \Gamma,\mathcal{S},P'$
by ($metis$ $stop\text{-}type$)
moreover have $tyenv\text{-}wellformed\ mds\ \Gamma\ \mathcal{S}\ P \longrightarrow tyenv\text{-}wellformed\ mds'\ \Gamma\ \mathcal{S}\ P'$
using upd $tyenv\text{-}wellformed\text{-}preds\text{-}update$ $assign_1$ **by** $metis$
moreover have $pred\ P\ mem \longrightarrow pred\ P'\ mem'$
using $pred\text{-}preds\text{-}update$ $assign_1$ upd **by** $metis$

moreover have $tyenv\text{-}wellformed\ mds\ \Gamma\ \mathcal{S}\ P\ \wedge\ tyenv\text{-}sec\ mds\ \Gamma\ mem \longrightarrow tyenv\text{-}sec\ mds\ \Gamma\ mem'$
using $tyenv\text{-}sec\text{-}eq[OF\ \mathcal{C}\text{-}eq$, **where** $\Gamma{=}\Gamma]$
unfolding $tyenv\text{-}wellformed\text{-}def$ **by** $blast$
ultimately show $?case$
by ($metis$ upd)
next
case ($assign_2$ x Γ e t $\mathcal{S}$ P' P c' mds)
hence upd: $c' = Stop \wedge mds' = mds \wedge mem' = mem\ (x := ev_A\ mem\ e)$
by ($metis$ $assign\text{-}elim$)
let $?\Gamma' = \Gamma\ (x \mapsto t)$
from upd **have** ty: $\vdash\ ?\Gamma',\mathcal{S},P'\ \{c'\}\ ?\Gamma',\mathcal{S},P'$
by ($metis$ $stop\text{-}type$)
have wf: $tyenv\text{-}wellformed\ mds\ \Gamma\ \mathcal{S}\ P \longrightarrow tyenv\text{-}wellformed\ mds'\ ?\Gamma'\ \mathcal{S}\ P'$
proof
assume $tyenv\text{-}wf$: $tyenv\text{-}wellformed\ mds\ \Gamma\ \mathcal{S}\ P$
hence wf: $types\text{-}wellformed\ \Gamma$
unfolding $tyenv\text{-}wellformed\text{-}def$ **by** $blast$
hence $type\text{-}wellformed\ t$
using $assign_2(2)$ $type\text{-}aexpr\text{-}type\text{-}wellformed$
by $blast$
with wf **have** wf': $types\text{-}wellformed\ ?\Gamma'$
using $types\text{-}wellformed\text{-}update$ **by** $metis$
from $tyenv\text{-}wf$ **have** $stable'$: $types\text{-}stable\ ?\Gamma'\ \mathcal{S}$
using $types\text{-}stable\text{-}update$
$assign_2(3)$
unfolding $tyenv\text{-}wellformed\text{-}def$ **by** $blast$
have m: $mds\text{-}consistent\ mds\ \Gamma\ \mathcal{S}\ P$
using $tyenv\text{-}wf$ **unfolding** $tyenv\text{-}wellformed\text{-}def$ **by** $blast$
from $assign_2(4)$ $assign_2(1)$
have $mds\text{-}consistent\ mds'\ (\Gamma(x \mapsto t))\ \mathcal{S}\ P'$
apply($rule$ $mds\text{-}consistent\text{-}preds\text{-}tyenv\text{-}update$)
using upd m **by** $simp$
from wf' $stable'$ $this$ **show** $tyenv\text{-}wellformed\ mds'\ ?\Gamma'\ \mathcal{S}\ P'$
unfolding $tyenv\text{-}wellformed\text{-}def$ **by** $blast$
qed
have p: $pred\ P\ mem \longrightarrow pred\ P'\ mem'$

using *pred-preds-update assign*$_2$ *upd* **by** *metis*
have *sec*: *tyenv-wellformed mds* Γ $\mathcal{S}$ *P* $\Longrightarrow$ *pred P mem* $\Longrightarrow$ *tyenv-sec mds* Γ *mem* $\Longrightarrow$ *tyenv-sec mds' ?*Γ' *mem'*
proof(*clarify*)
assume *wf*: *tyenv-wellformed mds* Γ $\mathcal{S}$ *P*
assume *pred*: *pred P mem*
assume *sec*: *tyenv-sec mds* Γ *mem*
from *pred p* **have** *pred'*: *pred P' mem'* **by** *blast*
fix *v t'*
assume Γv: $(\Gamma(x \mapsto t))$ *v* = *Some t'*
assume *v* $\notin$ *mds' AsmNoReadOrWrite*
show *type-max* (*the* $((\Gamma(x \mapsto t))$ *v*)) *mem'* $\leq$ *dma mem' v*
proof(*cases v* = *x*)
assume [*simp*]: *v* = *x*
hence [*simp*]: (*the* $((\Gamma(x \mapsto t))$ *v*)) = *t* **and** *t-def*: *t* = *t'*
using Γv **by** *auto*
from ‹*v* $\notin$ *mds' AsmNoReadOrWrite*› *upd wf* **have** *readable*: *v* $\notin$ *snd* $\mathcal{S}$
by(*auto simp*: *tyenv-wellformed-def mds-consistent-def*)
with *assign*$_2$(*5*) **have** *t* $\leq_{:P}'$ (*dma-type x*) **by** *fastforce*
with *pred'* **show** *?thesis*
using *type-max-dma-type subtype-correct*
by *fastforce*
next
assume *neq*: *v* $\neq$ *x*
hence [*simp*]: $((\Gamma(x \mapsto t))$ *v*) = Γ *v*
by *simp*
with Γv **have** Γv: Γ *v* = *Some t'* **by** *simp*
with *sec upd* ‹*v* $\notin$ *mds' AsmNoReadOrWrite*› **have** *f-leq*: *type-max t' mem* $\leq$ *dma mem v*
by *auto*
have $\mathcal{C}$*-eq*: $\forall\, x{\in}\mathcal{C}.$ *mem x* = *mem' x*
using *wf assign*$_2$(*1*) *upd* **by**(*auto simp*: *tyenv-wellformed-def mds-consistent-def*)
hence *dma-eq*: *dma mem* = *dma mem'*
by(*rule dma-*$\mathcal{C}$)
have *f-eq*: *type-max t' mem* = *type-max t' mem'*
apply(*rule* $\mathcal{C}$*-eq-type-max-eq*)
using Γv *wf* **apply**(*force simp*: *tyenv-wellformed-def types-wellformed-def*)
by(*rule* $\mathcal{C}$*-eq*)
from *neq* Γv *f-leq dma-eq f-eq* **show** *?thesis*
by *simp*
qed
qed
from *ty wf p sec* **show** *?case*
by *blast*
next
case (*assign*$_{\mathcal{C}}$ *x* Γ *e t P P'* $\mathcal{S}$ *c' mds*)

hence *upd*: *c'* = *Stop* $\wedge$ *mds'* = *mds* $\wedge$ *mem'* = *mem* (*x* := *ev*$_A$ *mem e*)
by (*metis assign-elim*)

hence $\vdash \Gamma,\mathcal{S},P' \{c'\} \Gamma,\mathcal{S},P'$
by (*metis stop-type*)
moreover have *tyenv-wellformed mds* Γ $\mathcal{S}$ $P \longrightarrow$ *tyenv-wellformed mds'* Γ $\mathcal{S}$ P'
using *upd tyenv-wellformed-preds-update assign*$_\mathcal{C}$ **by** *metis*
moreover have *pred P mem* $\longrightarrow$ *pred P' mem'*
using *pred-preds-update assign*$_\mathcal{C}$ *upd* **by** *metis*
moreover have *tyenv-wellformed mds* Γ $\mathcal{S}$ $P \wedge$ *pred P mem* $\wedge$ *tyenv-sec mds* Γ *mem* $\Longrightarrow$ *tyenv-sec mds'* Γ *mem'*
proof(*clarify*)
fix *v t'*
assume *wf*: *tyenv-wellformed mds* Γ $\mathcal{S}$ P
assume *pred*: *pred P mem*
hence *pred'*: *pred P' mem'* **using** ‹*pred P mem* $\longrightarrow$ *pred P' mem'*› **by** *blast*
assume *sec*: *tyenv-sec mds* Γ *mem*
assume Γv: Γ *v* = *Some t'*
assume *readable'*: *v* $\notin$ *mds' AsmNoReadOrWrite*
with *upd* **have** *readable*: *v* $\notin$ *mds AsmNoReadOrWrite* **by** *simp*
with *wf* **have** *v* $\notin$ *snd* $\mathcal{S}$ **by**(*auto simp*: *tyenv-wellformed-def mds-consistent-def*)
show *type-max* (*the* (Γ *v*)) *mem'* $\leq$ *dma mem' v*
proof(*cases x* $\in$ $\mathcal{C}$*-vars v*)
assume *x* $\in$ $\mathcal{C}$*-vars v*
with *assign*$_\mathcal{C}$(*6*) ‹*v* $\notin$ *snd* $\mathcal{S}$› **have** (*to-total* Γ *v*) $\leq_{:P'}$ (*dma-type v*) **by** *blast*
from *pred'* Γv *subtype-correct this* **show** *?thesis*
using *type-max-dma-type* **by**(*auto simp*: *to-total-def split*: *if-splits*)
next
assume *x* $\notin$ $\mathcal{C}$*-vars v*
hence $\forall$ *v'*$\in$$\mathcal{C}$*-vars v*. *mem v'* = *mem' v'*
using *upd* **by** *auto*
hence *dma-eq*: *dma mem v* = *dma mem' v*
by(*rule dma-*$\mathcal{C}$*-vars*)
from Γv *assign*$_\mathcal{C}$(*4*) **have** *x* $\notin$ *vars-of-type t'* **by** *force*
have *type-wellformed t'*
using *wf* Γv **by**(*force simp*: *tyenv-wellformed-def types-wellformed-def*)
with ‹*x* $\notin$ *vars-of-type t'*› *upd* **have** *f-eq*: *type-max t' mem* = *type-max t' mem'*
using *vars-of-type-eq-type-max-eq* **by** *fastforce*
from *sec* Γv *readable* **have** *type-max t' mem* $\leq$ *dma mem v*
by *auto*
with *f-eq dma-eq* Γv **show** *?thesis*
by *simp*
qed
qed
ultimately show *?case*
by (*metis*)
next
case (*if-type* Γ *e t P* $\mathcal{S}$ *th* Γ' $\mathcal{S}'$ *P' el* Γ'' *P''* Γ''' *P'''* *c' mds*)
from *if-type*(*13*)
show *?case*
proof(*rule if-elim*)

```
   assume [simp]: ev_B mem e and [simp]: c' = th and [simp]: mem' = mem and
[simp]: mds' = mds
    from if-type(3) have  ⊢ Γ,S,P +_S e {c'} Γ',S',P' by simp
    hence ⊢ Γ,S,P +_S e {c'} Γ''',S',P'''
      apply(rule sub)
           apply simp+
         using if-type apply blast
        using if-type apply blast
       apply simp
      using if-type apply(blast intro: pred-entailment-trans)
      done
    moreover have tyenv-wellformed mds Γ S P ⟶ tyenv-wellformed mds' Γ S
(P +_S e)
      by(auto simp: tyenv-wellformed-def mds-consistent-def add-pred-def)
    moreover have pred P mem ⟶ pred (P +_S e) mem'
      by(auto simp: pred-def add-pred-def)
    moreover have tyenv-sec mds Γ mem ⟶ tyenv-sec mds' Γ mem'
      by(simp)
    ultimately show ?case by blast
  next
    assume [simp]: ¬ ev_B mem e and [simp]: c' = el and [simp]: mem' = mem
and [simp]: mds' = mds
    from if-type(5) have  ⊢ Γ,S,P +_S bexp-neg e {c'} Γ'',S',P'' by simp
    hence ⊢ Γ,S,P +_S bexp-neg e {c'} Γ''',S',P'''
      apply(rule sub)
           apply simp+
         using if-type apply blast
        using if-type apply blast
       apply simp
      using if-type apply(blast intro: pred-entailment-trans)
      done
    moreover have tyenv-wellformed mds Γ S P ⟶ tyenv-wellformed mds' Γ S
(P +_S bexp-neg e)
      by(auto simp: tyenv-wellformed-def mds-consistent-def add-pred-def)
    moreover have pred P mem ⟶ pred (P +_S bexp-neg e) mem'
      by(auto simp: pred-def add-pred-def bexp-neg-negates)
    moreover have tyenv-sec mds Γ mem ⟶ tyenv-sec mds' Γ mem'
      by(simp)
    ultimately show ?case by blast
  qed
next
  case (while-type Γ e t P S c c' mds)
  hence [simp]: mds' = mds ∧ c' = If e (c ;; While e c) Stop ∧ mem' = mem
    by (metis while-elim)
  have  ⊢ Γ,S,P {c'} Γ,S,P
    apply simp
    apply(rule if-type)
           apply(rule while-type(1))
          apply(rule while-type(2))
```

apply(*rule seq-type*)
apply(*rule while-type*(*3*))
apply(*rule has-type.while-type*)
apply(*rule while-type*(*1*))
apply(*rule while-type*(*2*))
apply(*rule while-type*(*3*))
apply(*rule stop-type*)
apply *simp*
apply *simp*
apply *simp*
apply(*rule add-pred-entailment*)
apply *simp*+
by(*blast intro*!: *tyenv-wellformed-subset add-pred-subset*)
thus *?case*
by *fastforce*
next
case (*seq-type* Γ $\mathcal{S}$ P c_1 Γ_1 $\mathcal{S}_1$ P_1 c_2 Γ_2 $\mathcal{S}_2$ P_2 c' *mds*)
thus *?case*
proof (*cases* c_1 = *Stop*)
assume [*simp*]: c_1 = *Stop*
with *seq-type* **have** [*simp*]: *mds'* = *mds* **and** [*simp*]: c' = c_2 **and** [*simp*]: *mem'* = *mem*
by (*metis seq-stop-elim*)+
have *context-eq*: *context-equiv* Γ P Γ_1 **and** [*simp*]: $\mathcal{S}_1$ = $\mathcal{S}$ **and** *entail*: $P \vdash P_1$ **and**
wf-imp: $\forall$ *mds*. *tyenv-wellformed mds* Γ $\mathcal{S}$ $P \longrightarrow$ *tyenv-wellformed mds* Γ_1 $\mathcal{S}$ P_1
using *stop-cxt seq-type*(*1*) **by** *simp*+
have $\vdash \Gamma$,$\mathcal{S}$,P {c_2} Γ_2,$\mathcal{S}_2$,P_2
apply(*rule sub*)
using *seq-type*(*3*) **apply** *simp*
apply(*rule context-eq*)
apply(*rule wf-imp*)
apply *simp*+
apply(*rule entail*)
apply(*rule pred-entailment-refl*)
done
thus *?case*
by *fastforce*
next
assume $c_1 \neq$ *Stop*
then obtain c_1' **where** *step*: $\langle c_1, mds, mem\rangle \rightsquigarrow \langle c_1', mds', mem'\rangle \wedge c' = (c_1' ;; c_2)$
by (*metis seq-elim seq-type.prems*)
then have *no-await* c_1 **using** ‹*no-await* (c_1 ;; c_2)› *no-await.cases* **by** *blast*
then obtain Γ''' $\mathcal{S}'''$ P''' **where** $\vdash \Gamma'''$,$\mathcal{S}'''$,P''' {c_1'} Γ_1,$\mathcal{S}_1$,P_1 $\wedge$
(*tyenv-wellformed mds* Γ $\mathcal{S}$ $P \wedge$ *pred* P *mem* $\wedge$ *tyenv-sec mds* Γ *mem* $\longrightarrow$
tyenv-wellformed mds' Γ''' $\mathcal{S}'''$ $P''' \wedge$ *pred* P''' *mem'* $\wedge$ *tyenv-sec mds'* Γ''' *mem'*)

```
    using step seq-type(2)
    by blast
  moreover
  from seq-type have ⊢ Γ1,S1,P1 {c2} Γ2,S2,P2 by auto
  moreover
  ultimately show ?case
    apply (rule-tac x = Γ''' in exI)
    using ‹⟨c1, mds, mem⟩ ⇝ ⟨c1', mds', mem'⟩ ∧ c' = c1' ;; c2› by blast
 qed
next
 case (sub Γ1 S P1 c Γ1' S' P1' Γ2 P2 Γ2' P2' c' mds)
 then obtain Γ'' S'' P'' where stuff: ⊢ Γ'',S'',P'' { c' } Γ1',S',P1' ∧
  (tyenv-wellformed mds Γ1 S P1 ∧ pred P1 mem ∧ tyenv-sec mds Γ1 mem ⟶
   tyenv-wellformed mds' Γ'' S'' P'' ∧ pred P'' mem' ∧ tyenv-sec mds' Γ'' mem')
  by force

 have imp: tyenv-wellformed mds Γ2 S P2 ∧ pred P2 mem ∧ tyenv-sec mds Γ2
mem ⟹
        tyenv-wellformed mds Γ1 S P1 ∧ pred P1 mem ∧ tyenv-sec mds Γ1 mem
  apply(rule conjI)
   using sub(5) sub(4) tyenv-wellformed-sub unfolding pred-def
   apply blast
  apply(rule conjI)
   using local.pred-def pred-entailment-def sub.hyps(7) apply auto[1]
  using sub(3) context-equiv-tyenv-sec unfolding pred-def by blast
 show ?case
  apply (rule-tac x = Γ'' in exI, rule-tac x = S'' in exI, rule-tac x=P'' in exI)

  apply (rule conjI)
   apply(rule has-type.sub)
        apply(rule stuff[THEN conjunct1])
       apply simp+
     apply(rule sub(5))
    apply(rule sub(6))
   apply simp
  using sub apply blast
 using imp stuff apply blast
 done
next
 case (await-type Γ e t P S c Γ' S' P' c' mds)
  show ?case using no-await-no-await await-type.prems by blast
qed


lemma preservation-no-await-plus:
 ⟦⟨c, mds, mem⟩ ⇝+ ⟨c', mds', mem'⟩;
  ⊢ Γ,S,P { c } Γ',S',P';
  no-await c⟧ ⟹
   no-await c' ∧ (∃ Γ'' S'' P''. (⊢ Γ'',S'',P'' { c' } Γ',S',P') ∧
```

```
    (tyenv-wellformed mds Γ 𝒮 P ∧ pred P mem ∧ tyenv-sec mds Γ mem ⟶
    tyenv-wellformed mds' Γ'' 𝒮'' P'' ∧ pred P'' mem' ∧ tyenv-sec mds' Γ'' mem'))
  apply (induct arbitrary: Γ 𝒮 P rule: my-trancl-step-induct3)
   using preservation-no-await no-await-trans apply fast
  using preservation-no-await no-await-trans by metis


lemma preservation:
  assumes typed: ⊢ Γ,𝒮,P { c } Γ',𝒮',P'
  assumes eval: ⟨c, mds, mem⟩ ⇝ ⟨c', mds', mem'⟩
  shows ∃ Γ'' 𝒮'' P''. (⊢ Γ'',𝒮'',P'' { c' } Γ',𝒮',P') ∧
                  (tyenv-wellformed mds Γ 𝒮 P ∧ pred P mem ∧ tyenv-sec mds Γ
mem ⟶
                    tyenv-wellformed mds' Γ'' 𝒮'' P'' ∧ pred P'' mem' ∧ tyenv-sec
mds' Γ'' mem')
  using typed eval
proof (induct arbitrary: c' mds rule: has-type.induct)

  case (anno-type Γ'' Γ 𝒮 upd 𝒮'' P'' P c₁ Γ' 𝒮' P')
  hence ⟨c₁, update-modes upd mds, mem⟩ ⇝ ⟨c', mds', mem'⟩
    by (metis upd-elim)
  with anno-type(5) obtain Γ''' 𝒮''' P''' where
    ⊢ Γ''',𝒮''',P''' { c' } Γ',𝒮',P' ∧
      (tyenv-wellformed (update-modes upd mds) Γ'' 𝒮'' P'' ∧ pred P'' mem ∧
tyenv-sec (update-modes upd mds) Γ'' mem ⟶
       tyenv-wellformed mds' Γ''' 𝒮''' P''' ∧ pred P''' mem' ∧ tyenv-sec mds' Γ'''
mem')
    by blast

  moreover
  have tyenv-wellformed mds Γ 𝒮 P ⟶ tyenv-wellformed (update-modes upd mds)
Γ'' 𝒮'' P''
    using anno-type
    apply auto
    by (metis tyenv-wellformed-mode-update)
  moreover
  have pred: pred P mem ⟶ pred P'' mem
    using anno-type
    by (auto simp: pred-def restrict-preds-to-vars-def)
  moreover
  have tyenv-wellformed mds Γ 𝒮 P ∧ pred P mem ∧ tyenv-sec mds Γ mem ⟶
tyenv-sec (update-modes upd mds) Γ'' mem
    apply(rule impI)
    apply(rule tyenv-sec-mode-update)
          using anno-type apply fastforce
         using anno-type pred apply fastforce
        using anno-type apply fastforce
      using anno-type apply(fastforce simp: tyenv-wellformed-def mds-consistent-def)
       using anno-type apply fastforce
```

apply(*fastforce simp*: *tyenv-wellformed-def mds-consistent-def*)
apply(*fastforce simp*: *tyenv-wellformed-def mds-consistent-def*)
using *anno-type* **apply**(*fastforce simp*: *tyenv-wellformed-def mds-consistent-def*)
by *simp*
ultimately show *?case*
by *blast*
next
case *stop-type*
with *stop-no-eval* **show** *?case* **..**
next
case *skip-type*
hence $c' = Stop \wedge mds' = mds \wedge mem' = mem$
by (*metis skip-elim*)
thus *?case*
by (*metis stop-type*)
next
case ($assign_1$ x Γ e t P P' $\mathcal{S}$ c' mds)
hence *upd*: $c' = Stop \wedge mds' = mds \wedge mem' = mem\ (x := ev_A\ mem\ e)$
by (*metis assign-elim*)
from $assign_1(2)$ *upd* **have** $\mathcal{C}$*-eq*: $\forall x{\in}\mathcal{C}.\ mem\ x = mem'\ x$
by *auto*
from *upd* **have** $\vdash \Gamma,\mathcal{S},P'\ \{c'\}\ \Gamma,\mathcal{S},P'$
by (*metis stop-type*)
moreover have *tyenv-wellformed mds* Γ $\mathcal{S}$ $P \longrightarrow$ *tyenv-wellformed mds'* Γ $\mathcal{S}$ P'
using *upd tyenv-wellformed-preds-update* $assign_1$ **by** *metis*
moreover have *pred* P *mem* $\longrightarrow$ *pred* P' *mem'*
using *pred-preds-update* $assign_1$ *upd* **by** *metis*

moreover have *tyenv-wellformed mds* Γ $\mathcal{S}$ P $\wedge$ *tyenv-sec mds* Γ *mem* $\longrightarrow$ *tyenv-sec mds* Γ *mem'*
using *tyenv-sec-eq*[*OF* $\mathcal{C}$*-eq*, **where** $\Gamma{=}\Gamma$]
unfolding *tyenv-wellformed-def* **by** *blast*
ultimately show *?case*
by (*metis upd*)
next
case ($assign_2$ x Γ e t $\mathcal{S}$ P' P c' mds)
hence *upd*: $c' = Stop \wedge mds' = mds \wedge mem' = mem\ (x := ev_A\ mem\ e)$
by (*metis assign-elim*)
let $?\Gamma' = \Gamma\ (x \mapsto t)$
from *upd* **have** *ty*: $\vdash\ ?\Gamma',\mathcal{S},P'\ \{c'\}\ ?\Gamma',\mathcal{S},P'$
by (*metis stop-type*)
have *wf*: *tyenv-wellformed mds* Γ $\mathcal{S}$ $P \longrightarrow$ *tyenv-wellformed mds'* $?\Gamma'$ $\mathcal{S}$ P'
proof
assume *tyenv-wf*: *tyenv-wellformed mds* Γ $\mathcal{S}$ P
hence *wf*: *types-wellformed* Γ
unfolding *tyenv-wellformed-def* **by** *blast*
hence *type-wellformed t*
using $assign_2(2)$ *type-aexpr-type-wellformed*
by *blast*

with *wf* **have** *wf'*: *types-wellformed* ?Γ′
using *types-wellformed-update* **by** *metis*
from *tyenv-wf* **have** *stable'*: *types-stable* ?Γ′ $\mathcal{S}$
using *types-stable-update*
assign$_2$(*3*)
unfolding *tyenv-wellformed-def* **by** *blast*
have *m*: *mds-consistent mds* Γ $\mathcal{S}$ *P*
using *tyenv-wf* **unfolding** *tyenv-wellformed-def* **by** *blast*
from *assign*$_2$(*4*) *assign*$_2$(*1*)
have *mds-consistent mds'* (Γ($x \mapsto t$)) $\mathcal{S}$ *P'*
apply(*rule mds-consistent-preds-tyenv-update*)
using *upd m* **by** *simp*
from *wf' stable' this* **show** *tyenv-wellformed mds'* ?Γ′ $\mathcal{S}$ *P'*
unfolding *tyenv-wellformed-def* **by** *blast*
qed
have *p*: *pred P mem* $\longrightarrow$ *pred P' mem'*
using *pred-preds-update assign*$_2$ *upd* **by** *metis*
have *sec*: *tyenv-wellformed mds* Γ $\mathcal{S}$ *P* $\Longrightarrow$ *pred P mem* $\Longrightarrow$ *tyenv-sec mds* Γ *mem* $\Longrightarrow$ *tyenv-sec mds'* ?Γ′ *mem'*
proof(*clarify*)
assume *wf*: *tyenv-wellformed mds* Γ $\mathcal{S}$ *P*
assume *pred*: *pred P mem*
assume *sec*: *tyenv-sec mds* Γ *mem*
from *pred p* **have** *pred'*: *pred P' mem'* **by** *blast*
fix *v t'*
assume Γ*v*: (Γ($x \mapsto t$)) *v* = *Some t'*
assume *v* ∉ *mds' AsmNoReadOrWrite*
show *type-max* (*the* ((Γ($x \mapsto t$)) *v*)) *mem'* ≤ *dma mem' v*
proof(*cases v* = *x*)
assume [*simp*]: *v* = *x*
hence [*simp*]: (*the* ((Γ($x \mapsto t$)) *v*)) = *t* **and** *t-def*: *t* = *t'*
using Γ*v* **by** *auto*
from ‹*v* ∉ *mds' AsmNoReadOrWrite*› *upd wf* **have** *readable*: *v* ∉ *snd* $\mathcal{S}$
by(*auto simp*: *tyenv-wellformed-def mds-consistent-def*)
with *assign*$_2$(*5*) **have** *t* ≤:$_P$ *'* (*dma-type x*) **by** *fastforce*
with *pred'* **show** *?thesis*
using *type-max-dma-type subtype-correct*
by *fastforce*
next
assume *neq*: *v* ≠ *x*
hence [*simp*]: ((Γ($x \mapsto t$)) *v*) = Γ *v*
by *simp*
with Γ*v* **have** Γ*v*: Γ *v* = *Some t'* **by** *simp*
with *sec upd* ‹*v* ∉ *mds' AsmNoReadOrWrite*› **have** *f-leq*: *type-max t' mem* ≤ *dma mem v*
by *auto*
have $\mathcal{C}$*-eq*: ∀ *x*∈$\mathcal{C}$. *mem x* = *mem' x*
using *wf assign*$_2$(*1*) *upd* **by**(*auto simp*: *tyenv-wellformed-def mds-consistent-def*)
hence *dma-eq*: *dma mem* = *dma mem'*

```
      by(rule dma-C)
    have f-eq: type-max t' mem = type-max t' mem'
      apply(rule C-eq-type-max-eq)
       using Γv wf apply(force simp: tyenv-wellformed-def types-wellformed-def)
      by(rule C-eq)
    from neq Γv f-leq dma-eq f-eq show ?thesis
      by simp
  qed
 qed
 from ty wf p sec show ?case
  by blast
next
 case (assign_C x Γ e t P P' S c' mds)

 hence upd: c' = Stop ∧ mds' = mds ∧ mem' = mem (x := ev_A mem e)
  by (metis assign-elim)
 hence ⊢ Γ,S,P' {c'} Γ,S,P'
  by (metis stop-type)
 moreover have tyenv-wellformed mds Γ S P ⟶ tyenv-wellformed mds' Γ S P'
  using upd tyenv-wellformed-preds-update assign_C by metis
 moreover have pred P mem ⟶ pred P' mem'
  using pred-preds-update assign_C upd by metis
 moreover have tyenv-wellformed mds Γ S P ∧ pred P mem ∧ tyenv-sec mds Γ
mem ⟹ tyenv-sec mds' Γ mem'
 proof(clarify)
  fix v t'
  assume wf: tyenv-wellformed mds Γ S P
  assume pred: pred P mem
  hence pred': pred P' mem' using ‹pred P mem ⟶ pred P' mem'› by blast
  assume sec: tyenv-sec mds Γ mem
  assume Γv: Γ v = Some t'
  assume readable': v ∉ mds' AsmNoReadOrWrite
  with upd have readable: v ∉ mds AsmNoReadOrWrite by simp
 with wf have v ∉ snd S by(auto simp: tyenv-wellformed-def mds-consistent-def)
  show type-max (the (Γ v)) mem' ≤ dma mem' v
  proof(cases x ∈ C-vars v)
   assume x ∈ C-vars v
   with assign_C(6) ‹v ∉ snd S› have (to-total Γ v) ≤:_P' (dma-type v) by blast
   from pred' Γv subtype-sound[OF this] show ?thesis
    using type-max-dma-type by(auto simp: to-total-def split: if-splits)
  next
   assume x ∉ C-vars v
   hence ∀ v'∈C-vars v. mem v' = mem' v'
    using upd by auto
   hence dma-eq: dma mem v = dma mem' v
    by(rule dma-C-vars)
   from Γv assign_C(4) have x ∉ vars-of-type t' by force
   have type-wellformed t'
    using wf Γv by(force simp: tyenv-wellformed-def types-wellformed-def)
```

with ‹$x \notin$ *vars-of-type* t'› *upd* **have** *f-eq*: *type-max* t' *mem* = *type-max* t' *mem*$'$
using *vars-of-type-eq-type-max-eq* **by** *fastforce*
from *sec* Γv *readable* **have** *type-max* t' *mem* $\leq$ *dma mem v*
by *auto*
with *f-eq dma-eq* Γv **show** *?thesis*
by *simp*
qed
qed
ultimately show *?case*
by (*metis stop-type*)
next
case (*if-type* Γ *e t P* $\mathcal{S}$ *th* Γ' $\mathcal{S}'$ P' *el* Γ'' P'' Γ''' P''' c' *mds*)
from *if-type*(*13*)
show *?case*
proof(*rule if-elim*)
assume [*simp*]: ev_B *mem e* **and** [*simp*]: $c' = th$ **and** [*simp*]: *mem*$' =$ *mem* **and** [*simp*]: *mds*$' =$ *mds*
from *if-type*(*3*) **have** $\vdash \Gamma,\mathcal{S},P +_{\mathcal{S}} e \{c'\} \Gamma',\mathcal{S}',P'$ **by** *simp*
hence $\vdash \Gamma,\mathcal{S},P +_{\mathcal{S}} e \{c'\} \Gamma''',\mathcal{S}',P'''$
apply(*rule sub*)
apply *simp+*
using *if-type* **apply** *blast*
using *if-type* **apply** *blast*
apply *simp*
using *if-type* **apply**(*blast intro*: *pred-entailment-trans*)
done
moreover have *tyenv-wellformed mds* Γ $\mathcal{S}$ $P \longrightarrow$ *tyenv-wellformed mds*$' \Gamma$ $\mathcal{S}$ $(P +_{\mathcal{S}} e)$
by(*auto simp*: *tyenv-wellformed-def mds-consistent-def add-pred-def*)
moreover have *pred P mem* $\longrightarrow$ *pred* $(P +_{\mathcal{S}} e)$ *mem*$'$
by(*auto simp*: *pred-def add-pred-def*)
moreover have *tyenv-sec mds* Γ *mem* $\longrightarrow$ *tyenv-sec mds*$' \Gamma$ *mem*$'$
by(*simp*)
ultimately show *?case* **by** *blast*
next
assume [*simp*]: $\neg$ ev_B *mem e* **and** [*simp*]: $c' = el$ **and** [*simp*]: *mem*$' =$ *mem* **and** [*simp*]: *mds*$' =$ *mds*
from *if-type*(*5*) **have** $\vdash \Gamma,\mathcal{S},P +_{\mathcal{S}}$ *bexp-neg e* $\{c'\} \Gamma'',\mathcal{S}',P''$ **by** *simp*
hence $\vdash \Gamma,\mathcal{S},P +_{\mathcal{S}}$ *bexp-neg e* $\{c'\} \Gamma''',\mathcal{S}',P'''$
apply(*rule sub*)
apply *simp+*
using *if-type* **apply** *blast*
using *if-type* **apply** *blast*
apply *simp*
using *if-type* **apply**(*blast intro*: *pred-entailment-trans*)
done
moreover have *tyenv-wellformed mds* Γ $\mathcal{S}$ $P \longrightarrow$ *tyenv-wellformed mds*$' \Gamma$ $\mathcal{S}$ $(P +_{\mathcal{S}}$ *bexp-neg e*$)$

```
    by(auto simp: tyenv-wellformed-def mds-consistent-def add-pred-def)
   moreover have pred P mem ⟶ pred (P +_S bexp-neg e) mem'
    by(auto simp: pred-def add-pred-def bexp-neg-negates)
   moreover have tyenv-sec mds Γ mem ⟶ tyenv-sec mds' Γ mem'
    by(simp)
   ultimately show ?case by blast
  qed
next
  case (while-type Γ e t P S c c' mds)
  hence [simp]: mds' = mds ∧ c' = If e (c ;; While e c) Stop ∧ mem' = mem
   by (metis while-elim)
  have ⊢ Γ,S,P {c'} Γ,S,P
   apply simp
   apply(rule if-type)
          apply(rule while-type(1))
         apply(rule while-type(2))
        apply(rule seq-type)
         apply(rule while-type(3))
        apply(rule has-type.while-type)
          apply(rule while-type(1))
         apply(rule while-type(2))
        apply(rule while-type(3))
       apply(rule stop-type)
      apply simp
     apply simp
    apply simp
   apply(rule add-pred-entailment)
   apply simp+
  by(blast intro!: tyenv-wellformed-subset add-pred-subset)
  thus ?case
   by fastforce
next
  case (seq-type Γ S P c1 Γ1 S1 P1 c2 Γ2 S2 P2 c' mds)
  thus ?case
  proof (cases c1 = Stop)
   assume [simp]: c1 = Stop
   with seq-type have [simp]: mds' = mds and [simp]: c' = c2 and [simp]: mem'
= mem
    by (metis seq-stop-elim)+
   have context-eq: context-equiv Γ P Γ1 and [simp]: S1 = S and entail: P ⊢ P1
and
          wf-imp: ∀ mds. tyenv-wellformed mds Γ S P ⟶ tyenv-wellformed mds
Γ1 S P1
    using stop-cxt seq-type(1) by simp+
   have ⊢ Γ,S,P {c2} Γ2,S2,P2
    apply(rule sub)
         using seq-type(3) apply simp
        apply(rule context-eq)
       apply(rule wf-imp)
```

```
        apply simp+
       apply(rule entail)
      apply(rule pred-entailment-refl)
      done
    thus ?case
      by fastforce
  next
    assume c1 ≠ Stop
   then obtain c1' where ⟨c1, mds, mem⟩ ⇝ ⟨c1', mds', mem'⟩ ∧ c' = (c1' ;; c2)
      by (metis seq-elim seq-type.prems)
    then obtain Γ''' S''' P''' where ⊢ Γ''',S''',P''' {c1'} Γ1,S1,P1 ∧
      (tyenv-wellformed mds Γ S P ∧ pred P mem ∧ tyenv-sec mds Γ mem ⟶
       tyenv-wellformed mds' Γ''' S''' P''' ∧ pred P''' mem' ∧ tyenv-sec mds' Γ'''
mem')
      using seq-type(2)
      by force
    moreover
    from seq-type have ⊢ Γ1,S1,P1 {c2} Γ2,S2,P2 by auto
    moreover
    ultimately show ?case
      apply (rule-tac x = Γ''' in exI)
      using ‹⟨c1, mds, mem⟩ ⇝ ⟨c1', mds', mem'⟩ ∧ c' = c1' ;; c2› by blast
  qed
next
  case (sub Γ1 S P1 c Γ1' S' P1' Γ2 P2 Γ2' P2' c' mds)
  then obtain Γ'' S'' P'' where stuff: ⊢ Γ'',S'',P'' { c' } Γ1',S',P1' ∧
    (tyenv-wellformed mds Γ1 S P1 ∧ pred P1 mem ∧ tyenv-sec mds Γ1 mem ⟶
     tyenv-wellformed mds' Γ'' S'' P'' ∧ pred P'' mem' ∧ tyenv-sec mds' Γ'' mem')
    by force

  have imp: tyenv-wellformed mds Γ2 S P2 ∧ pred P2 mem ∧ tyenv-sec mds Γ2
mem ⟹
          tyenv-wellformed mds Γ1 S P1 ∧ pred P1 mem ∧ tyenv-sec mds Γ1 mem
    apply(rule conjI)
     using sub(5) sub(4) tyenv-wellformed-sub unfolding pred-def
     apply blast
    apply(rule conjI)
     using local.pred-def pred-entailment-def sub.hyps(7) apply auto[1]
    using sub(3) context-equiv-tyenv-sec unfolding pred-def by blast
  show ?case
    apply (rule-tac x = Γ'' in exI, rule-tac x = S'' in exI, rule-tac x=P'' in exI)

    apply (rule conjI)
     apply(rule has-type.sub)
          apply(rule stuff[THEN conjunct1])
         apply simp+
       apply(rule sub(5))
      apply(rule sub(6))
     apply simp
```

using *sub* **apply** *blast*
using *imp stuff* **apply** *blast*
done
next
case (*await-type* Γ *e t P* $\mathcal{S}$ *c* Γ' $\mathcal{S}'$ *P' c' mds*)
from *this* **show** *?case*
apply *simp*
apply (*drule await-elim*, *clarsimp*)
apply (*drule preservation-no-await-plus*[*of c mds mem c' mds' mem'* Γ $\mathcal{S}$ *P* $+_{\mathcal{S}}$ *e* Γ' $\mathcal{S}'$ *P'*], *assumption*+)
apply (*subgoal-tac* ⟦ *tyenv-wellformed mds* Γ $\mathcal{S}$ *P* ⟧ ⟹ *tyenv-wellformed mds* Γ $\mathcal{S}$ *P* $+_{\mathcal{S}}$ *e*) **defer**
apply (*unfold add-pred-def*)[*1*]
apply (*case-tac pred-stable* $\mathcal{S}$ *e*, *clarsimp*)
apply (*unfold tyenv-wellformed-def*, *clarsimp*)[*1*]
apply (*unfold mds-consistent-def*, *clarsimp*)[*1*]
apply *clarsimp*
apply (*subgoal-tac pred P mem* ⟹ *pred P* $+_{\mathcal{S}}$ *e mem*) **defer**
apply (*unfold add-pred-def*)[*1*]
apply (*case-tac pred-stable* $\mathcal{S}$ *e*, *clarsimp*)
apply (*unfold pred-def*, *clarsimp*)[*1*]
apply *clarsimp*
apply *clarsimp*
using *has-type.sub* **by** (*metis context-equiv-refl pred-entailment-refl*)
qed

inductive-cases *await-type-elim*: ⊢ Γ,$\mathcal{S}$,*P* {*Await b ca*} Γ',$\mathcal{S}'$,*P'*

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⟩ $\mathcal{R}^3{}_{\Gamma',\mathcal{S}',P'}$ ⟨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*)

lemma ev_A*-eq*:
assumes *tyenv-eq*: mem_1 $=_\Gamma$ mem_2
assumes *pred*: *pred P* mem_1
assumes *e-type*: $\Gamma \vdash_a e \in t$
assumes *subtype*: $t \leq:_P$ (*dma-type v*)
assumes *is-Low*: *dma* mem_1 *v* = *Low*
shows ev_A mem_1 *e* = ev_A mem_2 *e*

proof(*rule eval-vars-det*$_A$, *clarify*)
fix *x*
assume *in-vars*: $x \in$ *aexp-vars e*
have *type-max* (*to-total* Γ *x*) mem_1 = *Low*
proof −
from *subtype-sound*[*OF subtype*] *pred* **have** *type-max t* $mem_1 \leq$ *dma* mem_1 *v*
by(*auto*)
with *is-Low* **have** *type-max t* mem_1 = *Low* **by**(*auto simp*: *less-eq-Sec-def*)
with *e-type in-vars* **show** *?thesis*
apply −
apply(*erule type-aexpr.cases*)
using *Sec.exhaust* **by**(*auto simp*: *type-max-def split*: *if-splits*)
qed
thus mem_1 *x* = mem_2 *x*
using *tyenv-eq* **unfolding** *tyenv-eq-def* **by** *blast*
qed

lemma *ev*$_A$*-eq′*:
assumes *tyenv-eq*: $mem_1 =_\Gamma mem_2$
assumes *pred*: *pred P* mem_1
assumes *e-type*: $\Gamma \vdash_a e \in t$
assumes *subtype*: $P \vdash t$
shows *ev*$_A$ mem_1 *e* = *ev*$_A$ mem_2 *e*
proof(*rule eval-vars-det*$_A$, *clarify*)
fix *x*
assume *in-vars*: $x \in$ *aexp-vars e*
have *type-max* (*to-total* Γ *x*) mem_1 = *Low*
proof −
from *subtype pred* **have** *type-max t* $mem_1 \leq$ *Low*
by(*auto simp*: *type-max-def pred-entailment-def pred-def*)
hence *type-max t* mem_1 = *Low* **by**(*auto simp*: *less-eq-Sec-def*)
with *e-type in-vars* **show** *?thesis*
apply −
apply(*erule type-aexpr.cases*)
using *Sec.exhaust* **by**(*auto simp*: *type-max-def split*: *if-splits*)
qed
thus mem_1 *x* = mem_2 *x*
using *tyenv-eq* **unfolding** *tyenv-eq-def* **by** *blast*
qed

lemma *ev*$_B$*-eq′*:
assumes *tyenv-eq*: $mem_1 =_\Gamma mem_2$
assumes *pred*: *pred P* mem_1
assumes *e-type*: $\Gamma \vdash_b e \in t$
assumes *subtype*: $P \vdash t$
shows *ev*$_B$ mem_1 *e* = *ev*$_B$ mem_2 *e*
proof(*rule eval-vars-det*$_B$, *clarify*)
fix *x*
assume *in-vars*: $x \in$ *bexp-vars e*

have *type-max* (*to-total* Γ *x*) mem_1 = *Low*
proof −
from *subtype pred* **have** *type-max t* $mem_1 \leq$ *Low*
by(*auto simp*: *type-max-def pred-entailment-def pred-def*)
hence *type-max t* mem_1 = *Low* **by**(*auto simp*: *less-eq-Sec-def*)
with *e-type in-vars* **show** *?thesis*
apply −
apply(*erule type-bexpr.cases*)
using *Sec.exhaust* **by**(*auto simp*: *type-max-def split*: *if-splits*)
qed
thus mem_1 *x* = mem_2 *x*
using *tyenv-eq* **unfolding** *tyenv-eq-def* **by** *blast*
qed

lemma *R1-equiv-entailment*:
⟨*c*, *mds*, *mem*⟩ $\mathcal{R}^1{}_{\Gamma',\mathcal{S}',P'}$ ⟨*c'*, *mds'*, *mem'*⟩ ⟹
context-equiv Γ' P' Γ'' ⟹ $P' \vdash P''$ ⟹
∀ *mds*. *tyenv-wellformed mds* Γ' $\mathcal{S}'$ P' ⟶ *tyenv-wellformed mds* Γ'' $\mathcal{S}'$ P'' ⟹
⟨*c*, *mds*, *mem*⟩ $\mathcal{R}^1{}_{\Gamma'',\mathcal{S}',P''}$ ⟨*c'*, *mds'*, *mem'*⟩
apply(*induct rule*: $\mathcal{R}_1$.*induct*)
apply(*rule* $\mathcal{R}_1$.*intro*)
apply(*blast intro*: *sub context-equiv-refl pred-entailment-refl*)+
done

lemma *R3-equiv-entailment*:
⟨*c*, *mds*, *mem*⟩ $\mathcal{R}^3{}_{\Gamma',\mathcal{S}',P'}$ ⟨*c'*, *mds'*, *mem'*⟩ ⟹
context-equiv Γ' P' Γ'' ⟹ $P' \vdash P''$ ⟹
∀ *mds*. *tyenv-wellformed mds* Γ' $\mathcal{S}'$ P' ⟶ *tyenv-wellformed mds* Γ'' $\mathcal{S}'$ P'' ⟹
⟨*c*, *mds*, *mem*⟩ $\mathcal{R}^3{}_{\Gamma'',\mathcal{S}',P''}$ ⟨*c'*, *mds'*, *mem'*⟩
apply(*induct rule*: $\mathcal{R}_3$-*aux*.*induct*)
apply(*erule* $\mathcal{R}_3$-*aux*.*intro*$_1$)
apply(*blast intro*: *sub context-equiv-refl tyenv-wellformed-subset subset-entailment*)
apply(*erule* $\mathcal{R}_3$-*aux*.*intro*$_3$)
apply(*blast intro*: *sub context-equiv-refl tyenv-wellformed-subset subset-entailment*)
done

lemma *R-equiv-entailment*:
lc_1 $\mathcal{R}^u{}_{\Gamma',\mathcal{S}',P'}$ lc_2 ⟹
context-equiv Γ' P' Γ'' ⟹ $P' \vdash P''$ ⟹
∀ *mds*. *tyenv-wellformed mds* Γ' $\mathcal{S}'$ P' ⟶ *tyenv-wellformed mds* Γ'' $\mathcal{S}'$ P'' ⟹
lc_1 $\mathcal{R}^u{}_{\Gamma'',\mathcal{S}',P''}$ lc_2
apply(*induct rule*: $\mathcal{R}$.*induct*)
apply *clarsimp*
apply(*rule* $\mathcal{R}$.*intro*$_1$)
apply(*fastforce intro*: *R1-equiv-entailment*)
apply(*rule* $\mathcal{R}$.*intro*$_3$)
apply(*fastforce intro*: *R3-equiv-entailment*)
done

lemma *context-equiv-tyenv-eq*:
tyenv-eq Γ *mem mem'* $\Longrightarrow$ *context-equiv* Γ P Γ' $\Longrightarrow$ *pred* P *mem* $\Longrightarrow$ *tyenv-eq* Γ' *mem mem'*
apply(*clarsimp simp*: *tyenv-eq-def to-total-def context-equiv-def split*: *if-splits simp*: *type-equiv-def*)
using *subtype-trans subtype-sound*
by (*metis domI less-eq-Sec-def option.sel*)

lemma $\mathcal{R}$*-typed-step-no-await*:
$[\![\vdash \Gamma,\mathcal{S},P \{ c_1 \} \Gamma',\mathcal{S}',P' ;$
tyenv-wellformed mds Γ $\mathcal{S}$ P; $mem_1 =_\Gamma mem_2$; *pred* P mem_1;
pred P mem_2; *tyenv-sec mds* Γ mem_1;
$\langle c_1, mds, mem_1\rangle \rightsquigarrow \langle {c_1}', mds', {mem_1}'\rangle$; *no-await* c_1 $]\!] \Longrightarrow$
$(\exists\ {c_2}'\ {mem_2}'.\ \langle c_1, mds, mem_2\rangle \rightsquigarrow \langle {c_2}', mds', {mem_2}'\rangle \wedge$
$\langle {c_1}', mds', {mem_1}'\rangle\ {\mathcal{R}^u}_{\Gamma',\mathcal{S}',P'}\ \langle {c_2}', mds', {mem_2}'\rangle)$
proof (*induct arbitrary*: *mds* ${c_1}'$ *rule*: *has-type.induct*)
case (*seq-type* Γ $\mathcal{S}$ P c_1 Γ'' $\mathcal{S}''$ P'' c_2 Γ' $\mathcal{S}'$ P' *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** *context-equiv* Γ P Γ'' **and** $\mathcal{S} = \mathcal{S}''$ **and** $P \vdash P''$ **and**
(∀ *mds*. *tyenv-wellformed mds* Γ $\mathcal{S}$ P ⟶ *tyenv-wellformed mds* Γ'' $\mathcal{S}$ P'')
by (*metis stop-cxt*)+
hence $\vdash \Gamma,\mathcal{S},P \{ c_2 \} \Gamma',\mathcal{S}',P'$
apply −
apply(*rule sub*)
using *seq-type*(*3*) **apply** *simp*
by *auto*
have $\langle c_2, mds, mem_1\rangle\ {\mathcal{R}^1}_{\Gamma',\mathcal{S}',P'}\ \langle c_2, mds, mem_2\rangle$
apply (*rule* $\mathcal{R}_1$*.intro* [*of* Γ])
apply(*rule* ‹$\vdash \Gamma,\mathcal{S},P \{ c_2 \} \Gamma',\mathcal{S}',P'$›)
using *seq-type* **by** *auto*
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*)
by (*auto simp*: ‹$c_1 = Stop$› *seq-stop-eval*$_w$ $\mathcal{R}$*.intro*$_1$)
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*:

```
      ⟨c1, mds, mem1⟩ ⇝ ⟨c1'', mds', mem1'⟩ ∧ c1' = c1'' ;; c2
      by (metis seq-elim)
    with ‹no-await (c1 ;; c2)› have no-await c1 using no-await.cases by blast
    with seq-type(2) ‹no-await c1› obtain c2'' mem2' where c2''-props:
      ⟨c1, mds, mem2⟩ ⇝ ⟨c2'', mds', mem2'⟩ ∧ ⟨c1'', mds', mem1'⟩ R^u_{Γ'',S'',P''}
⟨c2'', mds', mem2'⟩
    using seq-type.prems(1) seq-type.prems(2) seq-type.prems(3) seq-type.prems(4)
seq-type.prems(5) c1''-props
      by blast
    hence ⟨c1'' ;; c2, mds', mem1'⟩ R^u_{Γ',S',P'} ⟨c2'' ;; c2, mds', mem2'⟩
      apply (rule conjE)
      apply (erule R-elim, auto)
       apply (metis R.intro3 R3-aux.intro1 seq-type(3))
      by (metis R.intro3 R3-aux.intro3 seq-type(3))
    moreover
    from c2''-props have ⟨c1 ;; c2, mds, mem2⟩ ⇝ ⟨c2'' ;; c2, mds', mem2'⟩
      by (metis eval_w.seq)
    ultimately show ?case
      by (metis c1''-props)
  qed
next
  case (anno-type Γ' Γ S upd S' P' P c Γ'' S'' P'' mds)
  have mem1 =_{Γ'} mem2
  proof(clarsimp simp: tyenv-eq-def)
    fix x
    assume a: type-max (to-total Γ' x) mem1 = Low
    hence type-max (to-total Γ x) mem1 = Low
    proof −
      from ‹pred P mem1› have pred P' mem1
        using anno-type.hyps(3)
        by(auto simp: restrict-preds-to-vars-def pred-def)
      with subtype-correct anno-type.hyps(7) a
      show ?thesis
        using less-eq-Sec-def by metis
    qed
    thus  mem1 x = mem2 x
      using anno-type.prems(2)
      unfolding tyenv-eq-def by blast
  qed

  have tyenv-wellformed mds Γ S P ⟶ tyenv-wellformed (update-modes upd mds)
Γ' S' P'
    using anno-type
    apply auto
    by (metis tyenv-wellformed-mode-update)
  moreover
  have pred: pred P mem1 ⟶ pred P' mem1
    using anno-type
    by (auto simp: pred-def restrict-preds-to-vars-def)
```

moreover
have *tyenv-wellformed mds* Γ $\mathcal{S}$ $P \wedge$ *pred* P $mem_1 \wedge$ *tyenv-sec mds* Γ $mem_1 \longrightarrow$

tyenv-sec (*update-modes upd mds*) Γ' mem_1
apply(*rule impI*)
apply(*rule tyenv-sec-mode-update*)
using *anno-type* **apply** *fastforce*
using *anno-type pred* **apply** *fastforce*
using *anno-type* **apply** *fastforce*
using *anno-type* **apply**(*fastforce simp*: *tyenv-wellformed-def mds-consistent-def*)
using *anno-type* **apply** *fastforce*
apply(*fastforce simp*: *tyenv-wellformed-def mds-consistent-def*)
apply(*fastforce simp*: *tyenv-wellformed-def mds-consistent-def*)
using *anno-type* **apply**(*fastforce simp*: *tyenv-wellformed-def mds-consistent-def*)
by *simp*
from ‹*no-await* (*c*@[*upd*])› **have** *no-await c* **using** *no-await.cases* **by** *blast*
ultimately obtain c_2' mem_2' **where** ($\langle c,$ *update-modes upd mds*, $mem_2\rangle \rightsquigarrow \langle c_2',$ *mds'*, $mem_2'\rangle \wedge$
$\langle c_1',$ *mds'*, $mem_1'\rangle$ $\mathcal{R}^u{}_{\Gamma'',\mathcal{S}'',P''}$ $\langle c_2',$ *mds'*, $mem_2'\rangle$)
using *anno-type*
apply *auto*
using ‹$mem_1 =_{\Gamma'} mem_2$› *local.pred-def restrict-preds-to-vars-def upd-elim* ‹*no-await c*›

using ‹*tyenv-wellformed mds* Γ $\mathcal{S}$ $P \wedge$ *pred* P $mem_1 \wedge$ ($\forall x{\in}dom\ \Gamma.\ x \notin$ *mds AsmNoReadOrWrite* $\longrightarrow$ *type-max* (*the* (Γ x)) $mem_1 \leq$ *dma* mem_1 x) $\longrightarrow$ ($\forall x{\in}dom\ \Gamma'.\ x \notin$ *update-modes upd mds AsmNoReadOrWrite* $\longrightarrow$ *type-max* (*the* (Γ' x)) $mem_1 \leq$ *dma* mem_1 x)› *mem-Collect-eq* **by** *fastforce* **try0**
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 $\mathcal{S}$ P *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** $\langle$*Stop*, *mds*, $mem_1\rangle$ $\mathcal{R}^1{}_{\Gamma,\mathcal{S},P}$ $\langle$*Stop*, *mds*, $mem_2\rangle$
by *auto*
thus *?case*
using $\mathcal{R}.intro_1$ **and** *unannotated* [**where** c = *Skip* **and** E = []]
apply *auto*
by (*metis* (*mono-tags*, *lifting*) $\mathcal{R}.intro_1$ *old.prod.case skip-eval*$_w$)
next

case (*assign*$_1$ x Γ e t P P' $\mathcal{S}$ *mds*)
hence *upd* [*simp*]: $c_1' = Stop$ $mds' = mds$ $mem_1' = mem_1$ $(x := ev_A\ mem_1\ e)$
using *assign-elim*
by (*auto*, *metis*)
from *assign*$_1$(*2*) *upd* **have** $\mathcal{C}$-*eq*: $\forall x{\in}\mathcal{C}.\ mem_1\ x = mem_1'\ x$
by *auto*
have *dma-eq* [*simp*]: *dma* mem_1 = *dma* mem_1'
using *dma-*$\mathcal{C}$ *assign*$_1$(*2*) **by** *simp*
have $mem_1\ (x := ev_A\ mem_1\ e) =_\Gamma mem_2\ (x := ev_A\ mem_2\ e)$
unfolding *tyenv-eq-def*
proof(*clarify*)
fix v
assume *is-Low'*: *type-max* (*to-total* Γ v) $(mem_1(x := ev_A\ mem_1\ e)) = Low$
show $(mem_1(x := ev_A\ mem_1\ e))\ v = (mem_2(x := ev_A\ mem_2\ e))\ v$
proof(*cases* $v \in dom\ \Gamma$)
assume [*simp*]: $v \in dom\ \Gamma$
then obtain t' **where** [*simp*]: $\Gamma\ v = Some\ t'$ **by** *force*
hence [*simp*]: (*to-total* Γ v) $= t'$
unfolding *to-total-def* **by** (*auto split*: *if-splits*)
have *type-max* t' mem_1 = *type-max* t' mem_1'
apply(*rule* $\mathcal{C}$*-eq-type-max-eq*)
using ‹$\Gamma\ v = Some\ t'$› *assign*$_1$(*6*)
unfolding *tyenv-wellformed-def types-wellformed-def*
apply (*metis* ‹$v \in dom\ \Gamma$› *option.sel*)

using *assign*$_1$(*2*) **apply** *simp*
done
with *is-Low'* **have** *is-Low*: *type-max* (*to-total* Γ v) $mem_1 = Low$
by *simp*
from *assign*$_1$(*1*) ‹$v \in dom\ \Gamma$› **have** $x \neq v$ **by** *auto*
thus *?thesis*
apply *simp*
using *is-Low assign*$_1$(*7*) **unfolding** *tyenv-eq-def* **by** *auto*
next
assume $v \notin dom\ \Gamma$
hence [*simp*]: $\bigwedge mem.$ *type-max* (*to-total* Γ v) *mem* = *dma mem v*
unfolding *to-total-def* **by** *simp*
with *is-Low'* **have** *dma* mem_1' $v = Low$ **by** *simp*
with *dma-eq* **have** *dma-v-Low*: *dma* mem_1 $v = Low$ **by** *simp*
hence *is-Low*: *type-max* (*to-total* Γ v) $mem_1 = Low$ **by** *simp*
show *?thesis*
proof(*cases* $x = v$)
assume $x \neq v$
thus *?thesis*
apply *simp*
using *is-Low assign*$_1$(*7*) **unfolding** *tyenv-eq-def* **by** *blast*
next
assume $x = v$
thus *?thesis*

apply *simp*
apply(*rule* ev_A*-eq*)
apply(*rule* $assign_1(7)$)
apply(*rule* $assign_1(8)$)
apply(*rule* $assign_1(3)$)
apply(*rule* $assign_1(4)$)
using *dma-v-Low* **by** *simp*
qed
qed
qed

moreover have *tyenv-wellformed mds* Γ $\mathcal{S}$ $P \longrightarrow$ *tyenv-wellformed mds'* Γ $\mathcal{S}$ P'
using *upd tyenv-wellformed-preds-update* $assign_1$ **by** *metis*
moreover have *pred* P $mem_1 \longrightarrow$ *pred* P' mem_1'
using *pred-preds-update* $assign_1$ *upd* **by** *metis*

moreover have *pred* P $mem_2 \longrightarrow$ *pred* P' $(mem_2(x := ev_A\ mem_2\ e))$
using *pred-preds-update* $assign_1$ *upd* **by** *metis*

moreover have *tyenv-wellformed mds* Γ $\mathcal{S}$ $P \ \wedge$ *tyenv-sec mds* Γ $mem_1 \longrightarrow$ *tyenv-sec mds* Γ mem_1'
using *tyenv-sec-eq*[*OF* $\mathcal{C}$*-eq*, **where** Γ=Γ]
unfolding *tyenv-wellformed-def* **by** *blast*

ultimately have $\mathcal{R}'$:
$\langle$*Stop*, *mds'*, $mem_1\ (x := ev_A\ mem_1\ e)\rangle\ \mathcal{R}^u{}_{\Gamma,\mathcal{S},P'}\ \langle$*Stop*, *mds'*, $mem_2\ (x := ev_A\ mem_2\ e)\rangle$
apply −
apply (*rule* $\mathcal{R}$*.intro*$_1$, *auto simp*: $assign_1$ *simp del*: *dma-eq*)
done

have *a*: $\langle x \leftarrow e, mds, mem_2\rangle \rightsquigarrow \langle$*Stop*, *mds'*, $mem_2\ (x := ev_A\ mem_2\ e)\rangle$
by (*auto*, *metis cxt-to-stmt.simps*(*1*) $eval_w$*.unannotated* $eval_w$*-simple.assign*)

from $\mathcal{R}'$ *a* **show** *?case*
using ‹$c_1' =$ *Stop*› **and** ‹$mem_1' = mem_1\ (x := ev_A\ mem_1\ e)$›
by *blast*
next
case ($assign_{\mathcal{C}}$ x Γ e t P P' $\mathcal{S}$ *mds*)
hence *upd* [*simp*]: $c_1' =$ *Stop* *mds'* = *mds* $mem_1' = mem_1\ (x := ev_A\ mem_1\ e)$
using *assign-elim*
by (*auto*, *metis*)
have $mem_1\ (x := ev_A\ mem_1\ e) =_\Gamma mem_2\ (x := ev_A\ mem_2\ e)$
unfolding *tyenv-eq-def*
proof(*clarify*)
fix v
assume *is-Low'*: *type-max* (*to-total* Γ v) $(mem_1(x := ev_A\ mem_1\ e)) =$ *Low*
show $(mem_1(x := ev_A\ mem_1\ e))\ v = (mem_2(x := ev_A\ mem_2\ e))\ v$
proof(*cases* $v \in$ *dom* Γ)

assume *in-dom* [*simp*]: $v \in dom\ \Gamma$
then obtain t' **where** Γv [*simp*]: $\Gamma\ v = Some\ t'$ **by** *force*
hence [*simp*]: $(to\text{-}total\ \Gamma\ v) = t'$
unfolding *to-total-def* **by** (*auto split*: *if-splits*)
from $assign_{\mathcal{C}}(4)$ **have** *x-nin-C*: $x \notin vars\text{-}of\text{-}type\ t'$
using *in-dom* Γv
by (*metis option.sel snd-conv*)
have Γv-*wf*: *type-wellformed* t'
using *in-dom* Γv $assign_{\mathcal{C}}(7)$ **unfolding** *tyenv-wellformed-def types-wellformed-def*
by (*metis option.sel*)

with *x-nin-C* **have** *f-eq*: *type-max* t' $mem_1 =$ *type-max* t' $mem_1{}'$
using *vars-of-type-eq-type-max-eq* **by** *simp*
with *is-Low'* **have** *is-Low*: *type-max* (*to-total* Γ v) $mem_1 = Low$
by *simp*
from $assign_{\mathcal{C}}(1)$ ‹$v \in dom\ \Gamma$› $assign_{\mathcal{C}}(7)$ **have** $x \neq v$
by(*auto simp*: *tyenv-wellformed-def mds-consistent-def*)
thus *?thesis*
apply *simp*
using *is-Low* $assign_{\mathcal{C}}(8)$ **unfolding** *tyenv-eq-def* **by** *auto*
next
assume *nin-dom*: $v \notin dom\ \Gamma$
hence [*simp*]: $\bigwedge mem.$ *type-max* (*to-total* Γ v) $mem = dma\ mem\ v$
unfolding *to-total-def* **by** *simp*
with *is-Low'* **have** $dma\ mem_1{}'\ v = Low$ **by** *simp*
show *?thesis*
proof(*cases* $x = v$)
assume $x = v$
thus *?thesis*
apply *simp*
apply(*rule* ev_A-*eq'*)
apply(*rule* $assign_{\mathcal{C}}(8)$)
apply(*rule* $assign_{\mathcal{C}}(9)$)
apply(*rule* $assign_{\mathcal{C}}(2)$)
by(*rule* $assign_{\mathcal{C}}(3)$)
next
assume [*simp*]: $x \neq v$
show *?thesis*
proof(*cases* $x \in \mathcal{C}$-*vars* v)
assume *in-*$\mathcal{C}$*-vars*: $x \in \mathcal{C}$-*vars* v
hence $v \notin \mathcal{C}$
using $\mathcal{C}$*-vars-*$\mathcal{C}$ **by** *auto*
with *nin-dom* **have** $v \notin snd\ \mathcal{S}$
using $assign_{\mathcal{C}}(7)$
by(*auto simp*: *tyenv-wellformed-def mds-consistent-def stable-def*)
with *in-*$\mathcal{C}$*-vars* **have** $P \vdash (to\text{-}total\ \Gamma\ v)$
using $assign_{\mathcal{C}}(6)$ **by** *blast*
with $assign_{\mathcal{C}}(9)$ **have** *type-max* (*to-total* Γ v) $mem_1 = Low$
by(*auto simp*: *type-max-def pred-def pred-entailment-def*)

```
        thus ?thesis
          using not-sym[OF ‹x ≠ v›]
          apply simp
          using assign_C(8)
          unfolding tyenv-eq-def by auto
      next
        assume x ∉ C-vars v
        with is-Low' have dma mem_1 v = Low
          using dma-C-vars ‹⋀mem. type-max (to-total Γ v) mem = dma mem v›
          by (metis fun-upd-other)
        thus ?thesis
          using not-sym[OF ‹x ≠ v›]
          apply simp
          using assign_C(8)
          unfolding tyenv-eq-def by auto
      qed
    qed
  qed
  qed

  moreover have tyenv-wellformed mds Γ S P ⟶ tyenv-wellformed mds' Γ S P'
    using upd tyenv-wellformed-preds-update assign_C by metis
  moreover have pred P mem_1 ⟶ pred P' mem_1'
    using pred-preds-update assign_C upd by metis
  moreover have pred P mem_2 ⟶ pred P' (mem_2(x := ev_A mem_2 e))
    using pred-preds-update assign_C upd by metis
  moreover have tyenv-wellformed mds Γ S P ∧ pred P mem_1 ∧ tyenv-sec mds
Γ mem_1 ⟹ tyenv-sec mds' Γ mem_1'
  proof(clarify)
    fix v t'
    assume wf: tyenv-wellformed mds Γ S P
    assume pred: pred P mem_1
    hence pred': pred P' mem_1' using ‹pred P mem_1 ⟶ pred P' mem_1'› by blast
    assume sec: tyenv-sec mds Γ mem_1
    assume Γv: Γ v = Some t'
    assume readable': v ∉ mds' AsmNoReadOrWrite
    with upd have readable: v ∉ mds AsmNoReadOrWrite by simp
   with wf have v ∉ snd S by(auto simp: tyenv-wellformed-def mds-consistent-def)
    show type-max (the (Γ v)) mem_1' ≤ dma mem_1' v
    proof(cases x ∈ C-vars v)
      assume x ∈ C-vars v
      with assign_C(6) ‹v ∉ snd S› have (to-total Γ v) ≤:_P' (dma-type v) by blast
      from pred' Γv subtype-correct this show ?thesis
        using type-max-dma-type by(auto simp: to-total-def split: if-splits)
    next
      assume x ∉ C-vars v
      hence ∀ v'∈C-vars v. mem_1 v' = mem_1' v'
        using upd by auto
      hence dma-eq: dma mem_1 v = dma mem_1' v
```

```
    by(rule dma-C-vars)
  from Γv assign_C(4) have x ∉ vars-of-type t' by force
  have type-wellformed t'
    using wf Γv by(force simp: tyenv-wellformed-def types-wellformed-def)
   with ‹x ∉ vars-of-type t'› upd have f-eq: type-max t' mem_1 = type-max t'
mem_1'
    using vars-of-type-eq-type-max-eq by fastforce
  from sec Γv readable have type-max t' mem_1 ≤ dma mem_1 v
    by auto
  with f-eq dma-eq Γv show ?thesis
    by simp
 qed
qed

ultimately have R':
 ⟨Stop, mds', mem_1 (x := ev_A mem_1 e)⟩ R^u_{Γ,S,P'} ⟨Stop, mds', mem_2 (x := ev_A
mem_2 e)⟩
 apply −
 apply (rule R.intro_1, auto simp: assign_C)
 done

have a: ⟨x ← e, mds, mem_2⟩ ⇝ ⟨Stop, mds', mem_2 (x := ev_A mem_2 e)⟩
by (auto, metis cxt-to-stmt.simps(1) eval_w.unannotated eval_w-simple.assign)

from R' a show ?case
 using ‹c_1' = Stop› and ‹mem_1' = mem_1 (x := ev_A mem_1 e)›
 by blast
next
case (assign_2 x Γ e t S P' P mds)
have upd [simp]: c_1' = Stop mds' = mds mem_1' = mem_1 (x := ev_A mem_1 e)
 using assign-elim[OF assign_2(11)]
 by auto
from ‹x ∈ dom Γ› ‹tyenv-wellformed mds Γ S P›
have x-nin-C: x ∉ C
 by(auto simp: tyenv-wellformed-def mds-consistent-def)
hence dma-eq [simp]: dma mem_1' = dma mem_1
 using dma-C assign_2
 by auto

let ?Γ' = Γ (x ↦ t)
have ⟨x ← e, mds, mem_2⟩ ⇝ ⟨Stop, mds, mem_2 (x := ev_A mem_2 e)⟩
 using assign_2
by (metis cxt-to-stmt.simps(1) eval_w-simplep.assign eval_wp.unannotated eval_wp-eval_w-eq)

moreover
have tyenv-eq': mem_1(x := ev_A mem_1 e) =_{Γ(x ↦ t)} mem_2(x := ev_A mem_2 e)
unfolding tyenv-eq-def
proof(clarify)
 fix v
```

assume *is-Low'*: *type-max* (*to-total* ($\Gamma(x \mapsto t)$) *v*) (*mem*$_1(x := ev_A\ mem_1\ e)$) = *Low*
show (*mem*$_1(x := ev_A\ mem_1\ e)$) *v* = (*mem*$_2(x := ev_A\ mem_2\ e)$) *v*
proof(*cases* $v = x$)
assume *neq*: $v \neq x$
hence *type-max* (*to-total* Γ *v*) *mem*$_1$ = *Low*
proof(*cases* $v \in dom\ \Gamma$)
assume $v \in dom\ \Gamma$
then obtain t' **where** [*simp*]: $\Gamma\ v = Some\ t'$ **by** *force*
hence [*simp*]: (*to-total* Γ *v*) = t'
unfolding *to-total-def* **by** (*auto split*: *if-splits*)
hence [*simp*]: (*to-total* ?Γ' *v*) = t'
using *neq* **by**(*auto simp*: *to-total-def*)
have *type-max* t' *mem*$_1$ = *type-max* t' *mem*$_1{}'$
apply(*rule* $\mathcal{C}$*-eq-type-max-eq*)
using *assign*$_2(6)$
apply(*clarsimp simp*: *tyenv-wellformed-def types-wellformed-def*)
using ‹*v* ∈ *dom* Γ› ‹Γ *v* = *Some* t'› **apply**(*metis option.sel*)
using *x-nin-*$\mathcal{C}$ **by** *simp*
from *this is-Low' neq neq*[*THEN not-sym*] **show** *type-max* (*to-total* Γ *v*) *mem*$_1$ = *Low*
by *auto*
next
assume $v \notin dom\ \Gamma$
with *is-Low' neq*
have *dma mem*$_1{}'$ *v* = *Low*
by(*auto simp*: *to-total-def split*: *if-splits*)
with *dma-eq* ‹*v* ∉ *dom* Γ› **show** *?thesis*
by(*auto simp*: *to-total-def split*: *if-splits*)
qed
with *neq assign*$_2(7)$ **show** (*mem*$_1(x := ev_A\ mem_1\ e)$) *v* = (*mem*$_2(x := ev_A\ mem_2\ e)$) *v*
by(*auto simp*: *tyenv-eq-def*)
next
assume *eq*[*simp*]: *v* = *x*
with *is-Low'* ‹*x* ∈ *dom* Γ› **have** *t-Low'*: *type-max t mem*$_1{}'$ = *Low*
by(*auto simp*: *to-total-def split*: *if-splits*)
have *wf-t*: *type-wellformed t*
using *type-aexpr-type-wellformed assign*$_2(2)$ *assign*$_2(6)$
by(*fastforce simp*: *tyenv-wellformed-def*)
with *t-Low'* ‹*x* ∉ $\mathcal{C}$› **have** *t-Low*: *type-max t mem*$_1$ = *Low*
using $\mathcal{C}$*-eq-type-max-eq*
by (*metis* (*no-types*, *lifting*) *fun-upd-other upd*(*3*))
show *?thesis*
proof(*simp*, *rule eval-vars-det*$_A$, *clarify*)
fix *y*
assume *in-vars*: *y* ∈ *aexp-vars e*
have *type-max* (*to-total* Γ *y*) *mem*$_1$ = *Low*
proof −

```
        from t-Low in-vars assign2(2) show ?thesis
          apply −
          apply(erule type-aexpr.cases)
          using Sec.exhaust by(auto simp: type-max-def  split: if-splits)
      qed
      thus mem1 y = mem2 y
        using assign2 unfolding tyenv-eq-def by blast
    qed
  qed
qed

from upd have ty: ⊢ ?Γ',S,P' {c1'} ?Γ',S,P'
  by (metis stop-type)
have wf: tyenv-wellformed mds Γ S P ⟶ tyenv-wellformed mds' ?Γ' S P'
proof
  assume tyenv-wf: tyenv-wellformed mds Γ S P
  hence wf: types-wellformed Γ
    unfolding tyenv-wellformed-def by blast
  hence  type-wellformed t
    using assign2(2) type-aexpr-type-wellformed
    by blast
  with wf have wf': types-wellformed ?Γ'
    using types-wellformed-update by metis
  from tyenv-wf have stable': types-stable ?Γ' S
    using types-stable-update
          assign2(3)
    unfolding tyenv-wellformed-def by blast
  have m: mds-consistent mds Γ S P
    using tyenv-wf unfolding tyenv-wellformed-def by blast
  from assign2(4) assign2(1)
  have mds-consistent mds' (Γ(x ↦ t)) S P'
    apply(rule mds-consistent-preds-tyenv-update)
    using upd m by simp
  from wf' stable' this show tyenv-wellformed mds' ?Γ' S P'
    unfolding tyenv-wellformed-def by blast
qed
have p: pred P mem1 ⟶ pred P' mem1'
  using pred-preds-update assign2 upd by metis
have p2: pred P mem2 ⟶ pred P' (mem2(x := ev_A mem2 e))
  using pred-preds-update assign2 upd by metis
 have sec: tyenv-wellformed mds Γ S P ⟹ pred P mem1 ⟹ tyenv-sec mds Γ
mem1 ⟹ tyenv-sec mds' ?Γ' mem1'
proof(clarify)
  assume wf: tyenv-wellformed mds Γ S P
  assume pred: pred P mem1
  assume sec: tyenv-sec mds Γ mem1
  from pred p have pred': pred P' mem1' by blast
  fix v t'
  assume Γv: (Γ(x ↦ t)) v = Some t'
```

assume $v \notin mds'$ *AsmNoReadOrWrite*
show *type-max* (*the* (($\Gamma(x \mapsto t)$) v)) $mem_1{}' \leq dma\ mem_1{}'\ v$
proof(*cases* $v = x$)
assume [*simp*]: $v = x$
hence [*simp*]: (*the* (($\Gamma(x \mapsto t)$) v)) $= t$ **and** *t-def*: $t = t'$
using Γv **by** *auto*
from ‹$v \notin mds'$ *AsmNoReadOrWrite*› *upd wf* **have** *readable*: $v \notin snd\ \mathcal{S}$
by(*auto simp*: *tyenv-wellformed-def mds-consistent-def*)
with *assign*$_2$(*5*) **have** $t \leq_{:P'} (dma\text{-}type\ x)$ **by** *fastforce*
with *pred*$'$ **show** *?thesis*
using *type-max-dma-type subtype-correct*
by *fastforce*
next
assume *neq*: $v \neq x$
hence [*simp*]: (($\Gamma(x \mapsto t)$) v) $= \Gamma\ v$
by *simp*
with Γv **have** Γv: $\Gamma\ v = Some\ t'$ **by** *simp*
with *sec upd* ‹$v \notin mds'$ *AsmNoReadOrWrite*› **have** *f-leq*: *type-max* t' mem_1 $\leq dma\ mem_1\ v$
by *auto*
have $\mathcal{C}$*-eq*: $\forall x{\in}\mathcal{C}.\ mem_1\ x = mem_1{}'\ x$
using *wf assign*$_2$(*1*) *upd* **by**(*auto simp*: *tyenv-wellformed-def mds-consistent-def*)
hence *dma-eq*: $dma\ mem_1 = dma\ mem_1{}'$
by(*rule dma-*$\mathcal{C}$)
have *f-eq*: *type-max* t' mem_1 = *type-max* t' $mem_1{}'$
apply(*rule* $\mathcal{C}$*-eq-type-max-eq*)
using Γv *wf* **apply**(*force simp*: *tyenv-wellformed-def types-wellformed-def*)
by(*rule* $\mathcal{C}$*-eq*)
from *neq* Γv *f-leq dma-eq f-eq* **show** *?thesis*
by *simp*
qed
qed

have $\langle Stop,\ mds,\ mem_1\ (x := ev_A\ mem_1\ e)\rangle\ \mathcal{R}^1{}_{?\Gamma',\mathcal{S},P'}\ \langle Stop,\ mds,\ mem_2\ (x := ev_A\ mem_2\ e)\rangle$
apply(*rule* $\mathcal{R}_1$*.intro*)
apply *blast*
using *wf assign*$_2$ **apply** *fastforce*
apply(*rule tyenv-eq*$'$)
using *p assign*$_2$ **apply** *fastforce*
using p_2 *assign*$_2$ **apply** *fastforce*
using *sec assign*$_2$
using *upd*(*2*) *upd*(*3*) **by** *blast*

ultimately have $\langle x \leftarrow e,\ mds,\ mem_2\rangle \rightsquigarrow \langle Stop,\ mds',\ mem_2\ (x := ev_A\ mem_2\ e)\rangle$
$\langle Stop,\ mds',\ mem_1\ (x := ev_A\ mem_1\ e)\rangle\ \mathcal{R}^u{}_{\Gamma(x \mapsto t),\mathcal{S},P'}\ \langle Stop,\ mds',\ mem_2\ (x := ev_A\ mem_2\ e)\rangle$
using $\mathcal{R}$*.intro*$_1$

```
    by auto
  thus ?case
    using ‹mds' = mds› ‹c1' = Stop› ‹mem1' = mem1(x := evA mem1 e)›
    by blast
next
  case (if-type Γ e t P S th Γ' S' P' el Γ'' P'' Γ''' P''')
  let ?P = if (evB mem1 e) then P +S e else P +S (bexp-neg e)
  from ‹⟨Stmt.If e th el, mds, mem1⟩ ⇝ ⟨c1', mds', mem1'⟩› have ty: ⊢ Γ,S,?P
{c1'} Γ''',S',P'''
  proof (rule if-elim)
    assume c1' = th mem1' = mem1 mds' = mds evB mem1 e
    with if-type(3)
    show ?thesis
      apply simp
      apply(erule sub)
        using if-type apply simp+
      done
  next
    assume c1' = el mem1' = mem1 mds' = mds ¬ evB mem1 e
    with if-type(5)
    show ?thesis
      apply simp
      apply(erule sub)
        using if-type apply simp+
      done
  qed
  have evB-eq [simp]: evB mem1 e = evB mem2 e
    apply(rule evB-eq')
      apply(rule ‹mem1 =Γ mem2›)
     apply(rule ‹pred P mem1›)
    apply(rule ‹Γ ⊢b e ∈ t›)
    by(rule ‹ P ⊢ t›)
  have (⟨c1', mds, mem1⟩, ⟨c1', mds, mem2⟩) ∈ R Γ''' S' P'''
    apply (rule intro1)
    apply clarify
    apply (rule R1.intro [where Γ = Γ and Γ' = Γ''' and S = S and P = ?P])
       apply(rule ty)
      using ‹tyenv-wellformed mds Γ S P›
      apply(auto simp: tyenv-wellformed-def mds-consistent-def add-pred-def)[1]
     apply(rule ‹mem1 =Γ mem2›)
   using ‹pred P mem1› apply(fastforce simp: pred-def add-pred-def bexp-neg-negates)
  using ‹pred P mem2› apply(fastforce simp: pred-def add-pred-def bexp-neg-negates)
    by(rule ‹tyenv-sec mds Γ mem1›)

  show ?case
  proof −
    from evB-eq if-type(13) have (⟨If e th el, mds, mem2⟩ ⇝ ⟨c1', mds, mem2⟩)
      apply (cases evB mem1 e)
       apply (subgoal-tac c1' = 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*))
using *if-type.prems*(*6*) **apply** *blast*
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*))
using *if-type.prems*(*6*) **by** *blast*
with ‹$\langle c_1', mds, mem_1\rangle$ $\mathcal{R}^u{}_{\Gamma'''\!,\mathcal{S}'\!,P'''}$ $\langle c_1', mds, mem_2\rangle$› **show** *?thesis*
by (*metis if-elim if-type.prems*(*6*))
qed
next
case (*while-type* Γ *e t P* $\mathcal{S}$ *c*)
hence [*simp*]: $c_1' = (If\ e\ (c\ ;;\ While\ e\ c)\ Stop)$ **and**
[*simp*]: $mds' = mds$ **and**
[*simp*]: $mem_1' = mem_1$
by (*auto simp*: *while-elim*)

with *while-type* **have** $\langle While\ e\ c, mds, mem_2\rangle \rightsquigarrow \langle c_1', mds, mem_2\rangle$
by (*metis cxt-to-stmt.simps*(*1*) $eval_w$*-simplep.while* $eval_w$*p.unannotated* $eval_w$*p-*$eval_w$*-eq*)

moreover have *ty*: $\vdash \Gamma,\mathcal{S},P\ \{c_1'\}\ \Gamma,\mathcal{S},P$
apply *simp*
apply(*rule if-type*)
apply(*rule while-type*(*1*))
apply(*rule while-type*(*2*))
apply(*rule seq-type*)
apply(*rule while-type*(*3*))
apply(*rule has-type.while-type*)
apply(*rule while-type*(*1*))
apply(*rule while-type*(*2*))
apply(*rule while-type*(*3*))
apply(*rule stop-type*)
apply *simp+*
apply(*rule add-pred-entailment*)
apply *simp*
apply(*blast intro*!: *add-pred-subset tyenv-wellformed-subset*)
done
moreover
have $\langle c_1', mds, mem_1\rangle$ $\mathcal{R}^1{}_{\Gamma,\mathcal{S},P}$ $\langle c_1', mds, mem_2\rangle$
apply (*rule* $\mathcal{R}_1$*.intro* [**where** $\Gamma = \Gamma$])
apply(*rule ty*)
using *while-type* **apply** *simp+*
done
hence $\langle c_1', mds, mem_1\rangle$ $\mathcal{R}^u{}_{\Gamma,\mathcal{S},P}$ $\langle c_1', mds, mem_2\rangle$
using $\mathcal{R}$*.intro*$_1$ **by** *auto*
ultimately show *?case*
by *fastforce*
next

case (*sub* Γ_1 $\mathcal{S}$ P_1 c $\Gamma_1{}'$ $\mathcal{S}'$ $P_1{}'$ Γ_2 P_2 $\Gamma_2{}'$ $P_2{}'$ *mds* $c_1{}'$)
have *imp*: *tyenv-wellformed mds* Γ_2 $\mathcal{S}$ P_2 $\wedge$ *pred* P_2 *mem*$_1$ $\wedge$ *pred* P_2 *mem*$_2$ $\wedge$ *tyenv-sec mds* Γ_2 *mem*$_1$ $\Longrightarrow$
tyenv-wellformed mds Γ_1 $\mathcal{S}$ P_1 $\wedge$ *pred* P_1 *mem*$_1$ $\wedge$ *pred* P_1 *mem*$_2$ $\wedge$ *tyenv-sec mds* Γ_1 *mem*$_1$
apply(*rule conjI*)
using *sub*(*5*) *sub*(*4*) *tyenv-wellformed-sub* **unfolding** *pred-def*
apply *blast*
apply(*rule conjI*)
using *local.pred-def pred-entailment-def sub.hyps*(*7*) **apply** *auto*[*1*]
apply(*rule conjI*)
using *local.pred-def pred-entailment-def sub.hyps*(*7*) **apply** *auto*[*1*]
using *sub*(*3*) *context-equiv-tyenv-sec* **unfolding** *pred-def* **by** *blast*

have *tyenv-eq*: *mem*$_1$ $=_{\Gamma_1}$ *mem*$_2$
using *context-equiv-tyenv-eq sub* **by** *blast*

from *imp tyenv-eq* **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*$_1{}'\rangle$ $\mathcal{R}^u{}_{\Gamma_1{}',\mathcal{S}',P_1{}'}$ $\langle c_2{}',$ *mds*$',$ *mem*$_2{}'\rangle$
using *sub* **by** *blast*
with *R-equiv-entailment* $\langle P_1{}' \vdash P_2{}'\rangle$ **show** *?case*
using *sub.hyps*(*6*) *sub.hyps*(*5*) **by** *blast*
next case (*await-type* Γ e t P $\mathcal{S}$ c Γ' $\mathcal{S}'$ P' Γ'' P'')
from *this* **show** *?case* **using** *no-await-no-await* **by** *blast*
qed

lemma *is-final-*$\mathcal{R}_u$*-is-final*:
$\langle c_1,$ *mds*, *mem*$_1\rangle$ $\mathcal{R}^u{}_{\Gamma,\mathcal{S},P}$ $\langle c_2,$ *mds*, *mem*$_2\rangle \Longrightarrow$ *is-final* $c_1 \Longrightarrow$ *is-final* c_2
by (*fastforce dest*: *bisim-simple-*$\mathcal{R}_u$)

lemma *pred-plus-impl*:
pred P *mem* $\Longrightarrow$ ev_B *mem* $e \Longrightarrow$ *pred* P $+_S$ e *mem*
unfolding *add-pred-def pred-def* **by** *simp*

lemma *my-*$\mathcal{R}_3$*-aux-induct* [*consumes 1*, *case-names intro*$_1$ *intro*$_3$]:
$[\![\langle c_1,$ *mds*, *mem*$_1\rangle$ $\mathcal{R}^3{}_{\Gamma,\mathcal{S},P}$ $\langle c_2,$ *mds*, *mem*$_2\rangle$;
$\bigwedge c_1$ *mds mem*$_1$ Γ $\mathcal{S}$ P c_2 *mem*$_2$ c Γ' $\mathcal{S}'$ P'.
$[\![\langle c_1,$ *mds*, *mem*$_1\rangle$ $\mathcal{R}^1{}_{\Gamma,\mathcal{S},P}$ $\langle c_2,$ *mds*, *mem*$_2\rangle$;
$\vdash \Gamma,\mathcal{S},P$ $\{c\}$ $\Gamma',\mathcal{S}',P']\!]$ $\Longrightarrow$
Q $(c_1$;; $c)$ *mds mem*$_1$ Γ' $\mathcal{S}'$ P' $(c_2$;; $c)$ *mds mem*$_2$;
$\bigwedge c_1$ *mds mem*$_1$ Γ $\mathcal{S}$ P c_2 *mem*$_2$ c Γ' $\mathcal{S}'$ P'.
$[\![\langle c_1,$ *mds*, *mem*$_1\rangle$ $\mathcal{R}^3{}_{\Gamma,\mathcal{S},P}$ $\langle c_2,$ *mds*, *mem*$_2\rangle$;
Q c_1 *mds mem*$_1$ Γ $\mathcal{S}$ P c_2 *mds mem*$_2$;
$\vdash \Gamma,\mathcal{S},P$ $\{c\}$ $\Gamma',\mathcal{S}',P']\!]$ $\Longrightarrow$
Q $(c_1$;; $c)$ *mds mem*$_1$ Γ' $\mathcal{S}'$ P' $(c_2$;; $c)$ *mds mem*$_2]\!]$ $\Longrightarrow$
Q c_1 *mds mem*$_1$ Γ $\mathcal{S}$ P c_2 *mds mem*$_2$
using $\mathcal{R}_3$*-aux.induct*[**where**
?x1.0 $= \langle c_1,$ *mds*, *mem*$_1\rangle$ **and**

?x2.0 = Γ **and**
?x3.0 = $\mathcal{S}$ **and**
?x4.0 = *P* **and**
?x5.0 = ⟨c_2, *mds*, mem_2⟩ **and**
?P = λctx_1 Γ $\mathcal{S}$ *P* ctx_2. *Q* (*fst* (*fst* ctx_1)) (*snd* (*fst* ctx_1)) (*snd* ctx_1) Γ $\mathcal{S}$ *P* (*fst* (*fst* ctx_2)) (*snd* (*fst* ctx_2)) (*snd* ctx_2)]
by *force*

lemma $\mathcal{R}$*-typed-step-plus*:
⟦⟨c_1, *mds*, mem_1⟩ $\rightsquigarrow^+$ ⟨c_1′, *mds*′, mem_1′⟩;
⊢ Γ,$\mathcal{S}$,*P* { c_1 } Γ′,$\mathcal{S}$′,*P*′;
no-await c_1;
tyenv-wellformed mds Γ $\mathcal{S}$ *P*;
mem_1 =Γ mem_2;
pred P mem_1;
pred P mem_2;
tyenv-sec mds Γ mem_1 ⟧ ⟹
(∃ c_2′ mem_2′. ⟨c_1, *mds*, mem_2⟩ $\rightsquigarrow^+$ ⟨c_2′, *mds*′, mem_2′⟩ ∧
⟨c_1′, *mds*′, mem_1′⟩ $\mathcal{R}^u{}_{\Gamma',\mathcal{S}',P'}$ ⟨c_2′, *mds*′, mem_2′⟩)
proof (*induct arbitrary*: Γ $\mathcal{S}$ *P* mem_2 *rule*: *my-trancl-big-step-induct3*)
case (*base* c_1 *mds* mem_1 c_1′ *mds*′ mem_1′)
from *this* **show** *?case* **using** $\mathcal{R}$*-typed-step-no-await bisim-simple-*$\mathcal{R}_u$ **by** *fast*
next
case (*step* c_1 *mds* mem_1 c_1′ *mds*′ mem_1′ c_1″ *mds*″ mem_1″)
from *this* **obtain** mem_2′ **where** *step*$_2$′: ⟨c_1, *mds*, mem_2⟩ $\rightsquigarrow^+$ ⟨c_1′, *mds*′, mem_2′⟩
and
rel$_2$′: ⟨c_1′, *mds*′, mem_1′⟩ $\mathcal{R}^u{}_{\Gamma',\mathcal{S}',P'}$ ⟨c_1′, *mds*′, mem_2′⟩
using *bisim-simple-*$\mathcal{R}_u$ **by** (*metis fst-conv*)
from *rel*$_2$′ **show** *?case*
proof(*cases rule*: $\mathcal{R}$*.cases*)
case (*intro*$_1$)
from *this* **obtain** Γ″ $\mathcal{S}$″ *P*″ **where**
⊢ Γ″,$\mathcal{S}$″,*P*″ {c_1′} Γ′,$\mathcal{S}$′,*P*′
tyenv-wellformed mds′ Γ″ $\mathcal{S}$″ *P*″
mem_1′ =Γ″ mem_2′
pred P″ mem_1′
pred P″ mem_2′
∀ *x*∈*dom* Γ″. *x* ∉ *mds*′ *AsmNoReadOrWrite* ⟶ *type-max* (*the* (Γ″ *x*)) mem_1′ ≤ *dma* mem_1′ *x*
using $\mathcal{R}_1$*.cases* **by** *auto*
from *step*$_2$′ ‹*no-await* c_1› *step.hyps*(*1*) *step.hyps*(*4*) *this* **obtain** mem_2″ **where**
step$_2$″: ⟨c_1′, *mds*′, mem_2′⟩ $\rightsquigarrow^+$ ⟨c_1″, *mds*″, mem_2″⟩ **and**
rel$_2$″: ⟨c_1″, *mds*″, mem_1″⟩ $\mathcal{R}^u{}_{\Gamma',\mathcal{S}',P'}$ ⟨c_1″, *mds*″, mem_2″⟩
using *no-await-trancl bisim-simple-*$\mathcal{R}_u$ **by** (*metis fst-conv*)
from *this step*$_2$′ **show** *?thesis* **using** *trancl-trans* **by** *fast*
next
case (*intro*$_3$)
from *intro*$_3$ *step.prems step.hyps*(*1*) *step.hyps*(*3*) *step.hyps*(*4*) **obtain** mem_2″

where
step'': $\langle c_1', mds', mem_2' \rangle \rightsquigarrow^+ \langle c_1'', mds'', mem_2'' \rangle$ **and**
rel'': $\langle c_1'', mds'', mem_1'' \rangle \ \mathcal{R}^u{}_{\Gamma',\mathcal{S}',P'} \ \langle c_1'', mds'', mem_2'' \rangle$
proof (*induct arbitrary*: *rule*: *my-*$\mathcal{R}_3$*-aux-induct*)
case (*intro*$_1$ $c_1'1'$ *mds'* mem_1' Γ''' $\mathcal{S}'''$ P''' $c_1'1$ mem_2' $c_1'2$ Γ'' $\mathcal{S}''$ P'')
from *intro*$_1(1)$ **obtain** Γv $\mathcal{S}v$ Pv **where** *pre-props*:
$\vdash \Gamma v,\mathcal{S}v,Pv \ \{c_1'1\} \ \Gamma''',\mathcal{S}''',P'''$
tyenv-wellformed mds' Γv $\mathcal{S}v$ Pv
$mem_1' =_{\Gamma v} mem_2'$
pred Pv mem_1'
pred Pv mem_2'
$c_1'1 = c_1'1'$
$\forall x{\in}dom\ \Gamma v.\ x \notin mds'\ AsmNoReadOrWrite \longrightarrow type\text{-}max\ (the\ (\Gamma v\ x))\ mem_1' \leq dma\ mem_1'\ x$
using $\mathcal{R}_1$*.cases* **by** *blast*
from *this intro*$_1$ **have** *typed*: $\vdash \Gamma v,\mathcal{S}v,Pv \ \{c_1'1 \ ;; \ c_1'2\} \ \Gamma'',\mathcal{S}'',P''$
using *has-type.seq-type* **by** *blast*
from *this pre-props* ‹*no-await* c_1› ‹$\langle c_1, mds, mem_1 \rangle \rightsquigarrow^+ \langle c_1'1 \ ;; \ c_1'2, mds', mem_1' \rangle$› *intro*$_1(13)$
obtain mem_2'' **where**
step: $\langle c_1'1 \ ;; \ c_1'2, mds', mem_2' \rangle \rightsquigarrow^+ \langle c_1'', mds'', mem_2'' \rangle \wedge$
$\langle c_1'', mds'', mem_1'' \rangle \ \mathcal{R}^u{}_{\Gamma'',\mathcal{S}'',P''} \ \langle c_1'', mds'', mem_2'' \rangle$
using *no-await-trancl bisim-simple-*$\mathcal{R}_u$ **by** (*metis fst-conv*)
from *this intro*$_1(3)$ **show** *?case* **using** *no-await-trancl bisim-simple-*$\mathcal{R}_u$ **by** *blast*
next
case (*intro*$_3$ $c_1'1'$ *mds'* mem_1' Γ''' $\mathcal{S}'''$ P''' $c_1'1$ mem_2' $c_1'2$ Γ'' $\mathcal{S}''$ P'')
from *intro*$_3(1)$ **obtain** Γv $\mathcal{S}v$ Pv **where** *pre-props*:
$\vdash \Gamma v,\mathcal{S}v,Pv \ \{c_1'1\} \ \Gamma''',\mathcal{S}''',P'''$
tyenv-wellformed mds' Γv $\mathcal{S}v$ Pv
$mem_1' =_{\Gamma v} mem_2'$
pred Pv mem_1'
pred Pv mem_2'
$c_1'1 = c_1'1'$
$\forall x{\in}dom\ \Gamma v.\ x \notin mds'\ AsmNoReadOrWrite \longrightarrow type\text{-}max\ (the\ (\Gamma v\ x))\ mem_1' \leq dma\ mem_1'\ x$
by (*induct arbitrary*: *rule*: *my-*$\mathcal{R}_3$*-aux-induct*)
(*blast elim*: $\mathcal{R}_1$*.cases*, *blast*)
from *this intro*$_1$ **have** *typed*: $\vdash \Gamma v,\mathcal{S}v,Pv \ \{c_1'1 \ ;; \ c_1'2\} \ \Gamma'',\mathcal{S}'',P''$
using *has-type.seq-type intro*$_3$*.hyps*(3) **by** *blast*

from *this pre-props* ‹*no-await* c_1› ‹$\langle c_1, mds, mem_1 \rangle \rightsquigarrow^+ \langle c_1'1 \ ;; \ c_1'2, mds', mem_1' \rangle$› *intro*$_3$
obtain mem_2'' **where**
step: $\langle c_1'1 \ ;; \ c_1'2, mds', mem_2' \rangle \rightsquigarrow^+ \langle c_1'', mds'', mem_2'' \rangle \wedge$
$\langle c_1'', mds'', mem_1'' \rangle \ \mathcal{R}^u{}_{\Gamma'',\mathcal{S}'',P''} \ \langle c_1'', mds'', mem_2'' \rangle$
proof −
assume *a1*: $\bigwedge mem_2''.\ \langle c_1'1 \ ;; \ c_1'2, mds', mem_2' \rangle \rightsquigarrow^+ \langle c_1'', mds'', mem_2'' \rangle \wedge$

$\langle c_1{}'', mds'', mem_1{}''\rangle\ \mathcal{R}^u{}_{\Gamma'',\mathcal{S}'',P''}\ \langle c_1{}'', mds'', mem_2{}''\rangle$
$\Longrightarrow$ *thesis*
thus *?thesis* **using** *intro*$_3$*.prems*(*11*)
using *a1* **by** (*metis* (*no-types*) *pre-props*(*2−*)
$\langle\langle c_1, mds, mem_1\rangle \rightsquigarrow^+ \langle c_1{}'1 ;; c_1{}'2, mds', mem_1{}''\rangle\rangle$ ‹*no-await* c_1›
bisim-simple-$\mathcal{R}_u$ *fst-conv no-await-trancl typed*)
qed
from *this intro*$_3$ **show** *?case* **using** *no-await-trancl bisim-simple-*$\mathcal{R}_u$ **by** *blast*
qed
thus *?thesis*
by (*meson step*$_2{}'$ *trancl-trans*)
qed
qed

lemma $\mathcal{R}$*-typed-step*:
$[\![\vdash \Gamma,\mathcal{S},P \{ c_1 \} \Gamma',\mathcal{S}',P' ;$
tyenv-wellformed mds $\Gamma\ \mathcal{S}\ P$; $mem_1 =_\Gamma mem_2$; *pred* $P\ mem_1$;
pred $P\ mem_2$; *tyenv-sec mds* $\Gamma\ mem_1$;
$\langle c_1, mds, mem_1\rangle \rightsquigarrow \langle c_1{}', mds', mem_1{}''\rangle\ ]\!] \Longrightarrow$
$(\exists\ c_2{}'\ mem_2{}'.\ \langle c_1, mds, mem_2\rangle \rightsquigarrow \langle c_2{}', mds', mem_2{}''\rangle \wedge$
$\langle c_1{}', mds', mem_1{}''\rangle\ \mathcal{R}^u{}_{\Gamma',\mathcal{S}',P'}\ \langle c_2{}', mds', mem_2{}''\rangle)$
proof (*induct arbitrary*: *mds* $c_1{}'$ *rule*: *has-type.induct*)
case (*seq-type* $\Gamma\ \mathcal{S}\ P\ c_1\ \Gamma''\ \mathcal{S}''\ P''\ c_2\ \Gamma'\ \mathcal{S}'\ P'\ mds$)
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** *context-equiv* $\Gamma\ P\ \Gamma''$ **and** $\mathcal{S} = \mathcal{S}''$ **and** $P \vdash P''$ **and**
$(\forall\, mds.\ tyenv\text{-}wellformed\ mds\ \Gamma\ \mathcal{S}\ P \longrightarrow tyenv\text{-}wellformed\ mds\ \Gamma''\ \mathcal{S}\ P'')$
by (*metis stop-cxt*)+
hence $\vdash \Gamma,\mathcal{S},P \{ c_2 \} \Gamma',\mathcal{S}',P'$
apply −
apply(*rule sub*)
using *seq-type*(*3*) **apply** *simp*
by *auto*
have $\langle c_2, mds, mem_1\rangle\ \mathcal{R}^1{}_{\Gamma',\mathcal{S}',P'}\ \langle c_2, mds, mem_2\rangle$
apply (*rule* $\mathcal{R}_1$*.intro* [*of* Γ])
apply(*rule* ‹$\vdash \Gamma,\mathcal{S},P \{ c_2 \} \Gamma',\mathcal{S}',P'$›)
using *seq-type* **by** *auto*
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*)
by (*auto simp*: ‹$c_1 = Stop$› *seq-stop-eval*$_w$ $\mathcal{R}$*.intro*$_1$)
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*:
$\langle c_1, mds, mem_2\rangle \rightsquigarrow \langle c_2'', mds', mem_2'\rangle \wedge \langle c_1'', mds', mem_1'\rangle \ \mathcal{R}^u{}_{\Gamma'',\mathcal{S}'',P''}\ \langle c_2'', mds', mem_2'\rangle$
using *seq-type.prems*(*1*) *seq-type.prems*(*2*) *seq-type.prems*(*3*) *seq-type.prems*(*4*) *seq-type.prems*(*5*) **by** *presburger*
hence $\langle c_1'' ;; c_2, mds', mem_1'\rangle \ \mathcal{R}^u{}_{\Gamma',\mathcal{S}',P'}\ \langle c_2'' ;; c_2, mds', mem_2'\rangle$
apply (*rule conjE*)
apply (*erule* $\mathcal{R}$*-elim*, *auto*)
apply (*metis* $\mathcal{R}$*.intro*$_3$ $\mathcal{R}_3$*-aux.intro*$_1$ *seq-type*(*3*))
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* $\Gamma'\ \Gamma\ \mathcal{S}$ *upd* $\mathcal{S}'\ P'\ P\ c\ \Gamma''\ \mathcal{S}''\ P''$ *mds*)
have $mem_1 =_{\Gamma'} mem_2$
proof(*clarsimp simp*: *tyenv-eq-def*)
fix *x*
assume *a*: *type-max* (*to-total* Γ' *x*) $mem_1 = Low$
hence *type-max* (*to-total* Γ *x*) $mem_1 = Low$
proof −
from ‹*pred P mem*$_1$› **have** *pred* P' mem_1
using *anno-type.hyps*(*3*)
by(*auto simp*: *restrict-preds-to-vars-def pred-def*)
with *subtype-sound*[*OF anno-type.hyps*(*7*)] *a*
show *?thesis*
using *less-eq-Sec-def* **by** *metis*
qed
thus $mem_1\ x = mem_2\ x$
using *anno-type.prems*(*2*)
unfolding *tyenv-eq-def* **by** *blast*
qed

have *tyenv-wellformed mds* $\Gamma\ \mathcal{S}\ P \longrightarrow$ *tyenv-wellformed* (*update-modes upd mds*) $\Gamma'\ \mathcal{S}'\ P'$
using *anno-type*
apply *auto*
by (*metis tyenv-wellformed-mode-update*)

moreover
have *pred*: *pred P mem*$_1$ $\longrightarrow$ *pred P*$'$ *mem*$_1$
using *anno-type*
by (*auto simp*: *pred-def restrict-preds-to-vars-def*)
moreover
have *tyenv-wellformed mds* Γ $\mathcal{S}$ *P* $\wedge$ *pred P mem*$_1$ $\wedge$ *tyenv-sec mds* Γ *mem*$_1$ $\longrightarrow$

tyenv-sec (*update-modes upd mds*) Γ' *mem*$_1$
apply(*rule impI*)
apply(*rule tyenv-sec-mode-update*)
using *anno-type* **apply** *fastforce*
using *anno-type pred* **apply** *fastforce*
using *anno-type* **apply** *fastforce*
using *anno-type* **apply**(*fastforce simp*: *tyenv-wellformed-def mds-consistent-def*)
using *anno-type* **apply** *fastforce*
apply(*fastforce simp*: *tyenv-wellformed-def mds-consistent-def*)
apply(*fastforce simp*: *tyenv-wellformed-def mds-consistent-def*)
using *anno-type* **apply**(*fastforce simp*: *tyenv-wellformed-def mds-consistent-def*)
by *simp*
ultimately obtain *c*$_2$$'$ *mem*$_2$$'$ **where** (⟨*c*, *update-modes upd mds*, *mem*$_2$⟩ $\rightsquigarrow$ ⟨*c*$_2$$'$, *mds*$'$, *mem*$_2$$''$⟩ $\wedge$
⟨*c*$_1$$'$, *mds*$'$, *mem*$_1$$''$⟩ $\mathcal{R}^u{}_{\Gamma'',\mathcal{S}'',P''}$ ⟨*c*$_2$$'$, *mds*$'$, *mem*$_2$$''$⟩)
using *anno-type*
apply *auto*
using ‹*mem*$_1$ $=_{\Gamma'}$ *mem*$_2$› *local.pred-def restrict-preds-to-vars-def upd-elim* **by** *fastforce*
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 $\mathcal{S}$ *P 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*$_1$⟩ $\mathcal{R}^1{}_{\Gamma,\mathcal{S},P}$ ⟨*Stop*, *mds*, *mem*$_2$⟩
by *auto*
thus *?case*
using $\mathcal{R}$*.intro*$_1$ **and** *unannotated* [**where** *c* = *Skip* **and** *E* = []]
apply *auto*
by (*metis* (*mono-tags*, *lifting*) $\mathcal{R}$*.intro*$_1$ *old.prod.case skip-eval*$_w$)
next
case (*assign*$_1$ *x* Γ *e t P P*$'$ $\mathcal{S}$ *mds*)
hence *upd* [*simp*]: *c*$_1$$'$ = *Stop mds*$'$ = *mds mem*$_1$$'$ = *mem*$_1$ (*x* := *ev*$_A$ *mem*$_1$ *e*)

```
  using assign-elim
  by (auto, metis)
from assign₁(2) upd have C-eq: ∀ x∈C. mem₁ x = mem₁' x
  by auto
have dma-eq [simp]: dma mem₁ = dma mem₁'
  using dma-C assign₁(2) by simp
have mem₁ (x := ev_A mem₁ e) =_Γ mem₂ (x := ev_A mem₂ e)
unfolding tyenv-eq-def
proof(clarify)
  fix v
  assume is-Low': type-max (to-total Γ v) (mem₁(x := ev_A mem₁ e)) = Low
  show (mem₁(x := ev_A mem₁ e)) v = (mem₂(x := ev_A mem₂ e)) v
  proof(cases v ∈ dom Γ)
    assume [simp]: v ∈ dom Γ
    then obtain t' where [simp]: Γ v = Some t' by force
    hence [simp]: (to-total Γ v) = t'
      unfolding to-total-def by (auto split: if-splits)
    have type-max t' mem₁ = type-max t' mem₁'
      apply(rule C-eq-type-max-eq)
       using ‹Γ v = Some t'› assign₁(6)
       unfolding tyenv-wellformed-def types-wellformed-def
       apply (metis ‹v ∈ dom Γ› option.sel)

      using assign₁(2) apply simp
      done
    with is-Low' have is-Low: type-max (to-total Γ v) mem₁ = Low
      by simp
    from assign₁(1) ‹v ∈ dom Γ› have x ≠ v by auto
    thus ?thesis
      apply simp
      using is-Low assign₁(7) unfolding tyenv-eq-def by auto
  next
    assume v ∉ dom Γ
    hence [simp]: ⋀mem. type-max (to-total Γ v) mem = dma mem v
      unfolding to-total-def by simp
    with is-Low' have dma mem₁' v = Low by simp
    with dma-eq have dma-v-Low: dma mem₁ v = Low by simp
    hence is-Low: type-max (to-total Γ v) mem₁ = Low by simp
    show ?thesis
    proof(cases x = v)
      assume x ≠ v
      thus ?thesis
        apply simp
        using is-Low assign₁(7) unfolding tyenv-eq-def by blast
    next
      assume x = v
      thus ?thesis
        apply simp
        apply(rule ev_A-eq)
```

apply(*rule assign*$_1$(*7*))
apply(*rule assign*$_1$(*8*))
apply(*rule assign*$_1$(*3*))
apply(*rule assign*$_1$(*4*))
using *dma-v-Low* **by** *simp*
qed
qed
qed

moreover have *tyenv-wellformed mds* Γ $\mathcal{S}$ *P* $\longrightarrow$ *tyenv-wellformed mds'* Γ $\mathcal{S}$ *P'*
using *upd tyenv-wellformed-preds-update assign*$_1$ **by** *metis*
moreover have *pred P mem*$_1$ $\longrightarrow$ *pred P' mem*$_1$*'*
using *pred-preds-update assign*$_1$ *upd* **by** *metis*

moreover have *pred P mem*$_2$ $\longrightarrow$ *pred P'* (*mem*$_2$(*x* := *ev*$_A$ *mem*$_2$ *e*))
using *pred-preds-update assign*$_1$ *upd* **by** *metis*

moreover have *tyenv-wellformed mds* Γ $\mathcal{S}$ *P* $\wedge$ *tyenv-sec mds* Γ *mem*$_1$ $\longrightarrow$ *tyenv-sec mds* Γ *mem*$_1$*'*
using *tyenv-sec-eq*[*OF* $\mathcal{C}$*-eq*, **where** Γ=Γ]
unfolding *tyenv-wellformed-def* **by** *blast*

ultimately have $\mathcal{R}'$:
⟨*Stop*, *mds'*, *mem*$_1$ (*x* := *ev*$_A$ *mem*$_1$ *e*)⟩ $\mathcal{R}^u{}_{\Gamma,\mathcal{S},P'}$ ⟨*Stop*, *mds'*, *mem*$_2$ (*x* := *ev*$_A$ *mem*$_2$ *e*)⟩
apply −
apply (*rule* $\mathcal{R}$*.intro*$_1$, *auto simp*: *assign*$_1$ *simp del*: *dma-eq*)
done

have *a*: ⟨*x* ← *e*, *mds*, *mem*$_2$⟩ ⇝ ⟨*Stop*, *mds'*, *mem*$_2$ (*x* := *ev*$_A$ *mem*$_2$ *e*)⟩
by (*auto*, *metis cxt-to-stmt.simps*(*1*) *eval*$_w$*.unannotated eval*$_w$*-simple.assign*)

from $\mathcal{R}'$ *a* **show** *?case*
using ‹*c*$_1$*'* = *Stop*› **and** ‹*mem*$_1$*'* = *mem*$_1$ (*x* := *ev*$_A$ *mem*$_1$ *e*)›
by *blast*
next
case (*assign*$_{\mathcal{C}}$ *x* Γ *e t P P'* $\mathcal{S}$ *mds*)
hence *upd* [*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*) =$_\Gamma$ *mem*$_2$ (*x* := *ev*$_A$ *mem*$_2$ *e*)
unfolding *tyenv-eq-def*
proof(*clarify*)
fix *v*
assume *is-Low'*: *type-max* (*to-total* Γ *v*) (*mem*$_1$(*x* := *ev*$_A$ *mem*$_1$ *e*)) = *Low*
show (*mem*$_1$(*x* := *ev*$_A$ *mem*$_1$ *e*)) *v* = (*mem*$_2$(*x* := *ev*$_A$ *mem*$_2$ *e*)) *v*
proof(*cases v* ∈ *dom* Γ)
assume *in-dom* [*simp*]: *v* ∈ *dom* Γ
then obtain *t'* **where** Γ*v* [*simp*]: Γ *v* = *Some t'* **by** *force*

hence [*simp*]: (*to-total* Γ *v*) = *t*′
unfolding *to-total-def* **by** (*auto split*: *if-splits*)
from *assign*$_C$(*4*) **have** *x-nin-C*: *x* ∉ *vars-of-type t*′
using *in-dom* Γ*v*
by (*metis option.sel snd-conv*)
have Γ*v-wf*: *type-wellformed t*′
using *in-dom* Γ*v assign*$_C$(*7*) **unfolding** *tyenv-wellformed-def types-wellformed-def*
by (*metis option.sel*)

with *x-nin-C* **have** *f-eq*: *type-max t*′ *mem*$_1$ = *type-max t*′ *mem*$_1$′
using *vars-of-type-eq-type-max-eq* **by** *simp*
with *is-Low*′ **have** *is-Low*: *type-max* (*to-total* Γ *v*) *mem*$_1$ = *Low*
by *simp*
from *assign*$_C$(*1*) ‹*v* ∈ *dom* Γ› *assign*$_C$(*7*) **have** *x* ≠ *v*
by(*auto simp*: *tyenv-wellformed-def mds-consistent-def*)
thus *?thesis*
apply *simp*
using *is-Low assign*$_C$(*8*) **unfolding** *tyenv-eq-def* **by** *auto*
next
assume *nin-dom*: *v* ∉ *dom* Γ
hence [*simp*]: ⋀*mem*. *type-max* (*to-total* Γ *v*) *mem* = *dma mem v*
unfolding *to-total-def* **by** *simp*
with *is-Low*′ **have** *dma mem*$_1$′ *v* = *Low* **by** *simp*
show *?thesis*
proof(*cases x* = *v*)
assume *x* = *v*
thus *?thesis*
apply *simp*
apply(*rule ev*$_A$*-eq*′)
apply(*rule assign*$_C$(*8*))
apply(*rule assign*$_C$(*9*))
apply(*rule assign*$_C$(*2*))
by(*rule assign*$_C$(*3*))
next
assume [*simp*]: *x* ≠ *v*
show *?thesis*
proof(*cases x* ∈ *C-vars v*)
assume *in-C-vars*: *x* ∈ *C-vars v*
hence *v* ∉ *C*
using *C-vars-C* **by** *auto*
with *nin-dom* **have** *v* ∉ *snd S*
using *assign*$_C$(*7*)
by(*auto simp*: *tyenv-wellformed-def mds-consistent-def stable-def*)
with *in-C-vars* **have** *P* ⊢ (*to-total* Γ *v*)
using *assign*$_C$(*6*) **by** *blast*
with *assign*$_C$(*9*) **have** *type-max* (*to-total* Γ *v*) *mem*$_1$ = *Low*
by(*auto simp*: *type-max-def pred-def pred-entailment-def*)
thus *?thesis*
using *not-sym*[*OF* ‹*x* ≠ *v*›]

```
          apply simp
          using assign_C(8)
          unfolding tyenv-eq-def by auto
        next
          assume x ∉ C-vars v
          with is-Low' have dma mem_1 v = Low
           using dma-C-vars ‹⋀mem. type-max (to-total Γ v) mem = dma mem v›
           by (metis fun-upd-other)
          thus ?thesis
           using not-sym[OF ‹x ≠ v›]
           apply simp
           using assign_C(8)
           unfolding tyenv-eq-def by auto
        qed
      qed
    qed
  qed

  moreover have tyenv-wellformed mds Γ S P ⟶ tyenv-wellformed mds' Γ S P'
    using upd tyenv-wellformed-preds-update assign_C by metis
  moreover have pred P mem_1 ⟶ pred P' mem_1'
    using pred-preds-update assign_C upd by metis
  moreover have pred P mem_2 ⟶ pred P' (mem_2(x := ev_A mem_2 e))
    using pred-preds-update assign_C upd by metis
  moreover have tyenv-wellformed mds Γ S P ∧ pred P mem_1 ∧ tyenv-sec mds
Γ mem_1 ⟹ tyenv-sec mds' Γ mem_1'
  proof(clarify)
    fix v t'
    assume wf: tyenv-wellformed mds Γ S P
    assume pred: pred P mem_1
    hence pred': pred P' mem_1' using ‹pred P mem_1 ⟶ pred P' mem_1'› by blast
    assume sec: tyenv-sec mds Γ mem_1
    assume Γv: Γ v = Some t'
    assume readable': v ∉ mds' AsmNoReadOrWrite
    with upd have readable: v ∉ mds AsmNoReadOrWrite by simp
   with wf have v ∉ snd S by(auto simp: tyenv-wellformed-def mds-consistent-def)
    show type-max (the (Γ v)) mem_1' ≤ dma mem_1' v
    proof(cases x ∈ C-vars v)
      assume x ∈ C-vars v
      with assign_C(6) ‹v ∉ snd S› have (to-total Γ v) ≤:_P' (dma-type v) by blast
      from pred' Γv subtype-sound[OF this] show ?thesis
        using type-max-dma-type by(auto simp: to-total-def split: if-splits)
    next
      assume x ∉ C-vars v
      hence ∀ v'∈C-vars v. mem_1 v' = mem_1' v'
        using upd by auto
      hence dma-eq: dma mem_1 v = dma mem_1' v
        by(rule dma-C-vars)
      from Γv assign_C(4) have x ∉ vars-of-type t' by force
```

have *type-wellformed* t'
using *wf* Γv **by**(*force simp*: *tyenv-wellformed-def types-wellformed-def*)
with ‹$x \notin$ *vars-of-type* t'› *upd* **have** *f-eq*: *type-max* t' mem_1 = *type-max* t' $mem_1{}'$
using *vars-of-type-eq-type-max-eq* **by** *fastforce*
from *sec* Γv *readable* **have** *type-max* t' $mem_1 \leq$ *dma* mem_1 v
by *auto*
with *f-eq dma-eq* Γv **show** *?thesis*
by *simp*
qed
qed

ultimately have $\mathcal{R}'$:
$\langle Stop, mds', mem_1\ (x := ev_A\ mem_1\ e)\rangle\ \mathcal{R}^u{}_{\Gamma,\mathcal{S},P'}\ \langle Stop, mds', mem_2\ (x := ev_A\ mem_2\ e)\rangle$
apply −
apply (*rule* $\mathcal{R}$.*intro*$_1$, *auto simp*: *assign*$_C$)
done

have *a*: $\langle x \leftarrow e, mds, mem_2\rangle \rightsquigarrow \langle Stop, mds', mem_2\ (x := ev_A\ mem_2\ e)\rangle$
by (*auto*, *metis cxt-to-stmt.simps*(*1*) *eval*$_w$.*unannotated eval*$_w$*-simple.assign*)

from $\mathcal{R}'$ *a* **show** *?case*
using ‹$c_1{}'$ = *Stop*› **and** ‹$mem_1{}'$ = $mem_1\ (x := ev_A\ mem_1\ e)$›
by *blast*
next
case (*assign*$_2$ x Γ e t $\mathcal{S}$ P' P *mds*)
have *upd* [*simp*]: $c_1{}'$ = *Stop* *mds'* = *mds* $mem_1{}'$ = $mem_1\ (x := ev_A\ mem_1\ e)$
using *assign-elim*[*OF assign*$_2$(*11*)]
by *auto*
from ‹$x \in$ *dom* Γ› ‹*tyenv-wellformed mds* Γ $\mathcal{S}$ P›
have *x-nin-*$\mathcal{C}$: $x \notin \mathcal{C}$
by(*auto simp*: *tyenv-wellformed-def mds-consistent-def*)
hence *dma-eq* [*simp*]: *dma* $mem_1{}'$ = *dma* mem_1
using *dma-*$\mathcal{C}$ *assign*$_2$
by *auto*

let *?*Γ' = $\Gamma\ (x \mapsto t)$
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*$_w$*-simplep.assign eval*$_w$*p.unannotated eval*$_w$*p-eval*$_w$*-eq*)

moreover
have *tyenv-eq'*: $mem_1(x := ev_A\ mem_1\ e) =_{\Gamma(x \mapsto t)} mem_2(x := ev_A\ mem_2\ e)$
unfolding *tyenv-eq-def*
proof(*clarify*)
fix v
assume *is-Low'*: *type-max* (*to-total* ($\Gamma(x \mapsto t)$) v) ($mem_1(x := ev_A\ mem_1\ e)$) = *Low*

show $(mem_1(x := ev_A\ mem_1\ e))\ v = (mem_2(x := ev_A\ mem_2\ e))\ v$
proof(*cases* $v = x$)
assume *neq*: $v \neq x$
hence *type-max* (*to-total* Γ v) $mem_1 = Low$
proof(*cases* $v \in dom\ \Gamma$)
assume $v \in dom\ \Gamma$
then obtain t' **where** [*simp*]: $\Gamma\ v = Some\ t'$ **by** *force*
hence [*simp*]: (*to-total* Γ v) $= t'$
unfolding *to-total-def* **by** (*auto split*: *if-splits*)
hence [*simp*]: (*to-total* $?\Gamma'$ v) $= t'$
using *neq* **by**(*auto simp*: *to-total-def*)
have *type-max* t' $mem_1 =$ *type-max* t' $mem_1{}'$
apply(*rule* $\mathcal{C}$*-eq-type-max-eq*)
using $assign_2(6)$
apply(*clarsimp simp*: *tyenv-wellformed-def types-wellformed-def*)
using ‹$v \in dom\ \Gamma$› ‹$\Gamma\ v = Some\ t'$› **apply**(*metis option.sel*)
using *x-nin-*$\mathcal{C}$ **by** *simp*
from *this is-Low'* *neq* *neq*[*THEN not-sym*] **show** *type-max* (*to-total* Γ v) $mem_1 = Low$
by *auto*
next
assume $v \notin dom\ \Gamma$
with *is-Low'* *neq*
have *dma* $mem_1{}'$ $v = Low$
by(*auto simp*: *to-total-def* *split*: *if-splits*)
with *dma-eq* ‹$v \notin dom\ \Gamma$› **show** *?thesis*
by(*auto simp*: *to-total-def* *split*: *if-splits*)
qed
with *neq* $assign_2(7)$ **show** $(mem_1(x := ev_A\ mem_1\ e))\ v = (mem_2(x := ev_A\ mem_2\ e))\ v$
by(*auto simp*: *tyenv-eq-def*)
next
assume *eq*[*simp*]: $v = x$
with *is-Low'* ‹$x \in dom\ \Gamma$› **have** *t-Low'*: *type-max* *t* $mem_1{}' = Low$
by(*auto simp*: *to-total-def split*: *if-splits*)
have *wf-t*: *type-wellformed t*
using *type-aexpr-type-wellformed* $assign_2(2)$ $assign_2(6)$
by(*fastforce simp*: *tyenv-wellformed-def*)
with *t-Low'* ‹$x \notin \mathcal{C}$› **have** *t-Low*: *type-max* *t* $mem_1 = Low$
using $\mathcal{C}$*-eq-type-max-eq*
by (*metis* (*no-types*, *lifting*) *fun-upd-other* $upd(3)$)
show *?thesis*
proof(*simp*, *rule eval-vars-det*$_A$, *clarify*)
fix *y*
assume *in-vars*: $y \in aexp\text{-}vars\ e$
have *type-max* (*to-total* Γ y) $mem_1 = Low$
proof −
from *t-Low in-vars* $assign_2(2)$ **show** *?thesis*
apply −

```
        apply(erule type-aexpr.cases)
        using Sec.exhaust by(auto simp: type-max-def  split: if-splits)
      qed
      thus mem₁ y = mem₂ y
        using assign₂ unfolding tyenv-eq-def by blast
    qed
  qed
qed

from upd have ty: ⊢ ?Γ',S,P' {c₁'} ?Γ',S,P'
  by (metis stop-type)
have wf: tyenv-wellformed mds Γ S P ⟶ tyenv-wellformed mds' ?Γ' S P'
proof
  assume tyenv-wf: tyenv-wellformed mds Γ S P
  hence wf: types-wellformed Γ
    unfolding tyenv-wellformed-def by blast
  hence  type-wellformed t
    using assign₂(2) type-aexpr-type-wellformed
    by blast
  with wf have wf': types-wellformed ?Γ'
    using types-wellformed-update by metis
  from tyenv-wf have stable': types-stable ?Γ' S
    using types-stable-update
        assign₂(3)
    unfolding tyenv-wellformed-def by blast
  have m: mds-consistent mds Γ S P
    using tyenv-wf unfolding tyenv-wellformed-def by blast
  from assign₂(4) assign₂(1)
  have mds-consistent mds' (Γ(x ↦ t)) S P'
    apply(rule mds-consistent-preds-tyenv-update)
    using upd m by simp
  from wf' stable' this show tyenv-wellformed mds' ?Γ' S P'
    unfolding tyenv-wellformed-def by blast
qed
have p: pred P mem₁ ⟶ pred P' mem₁'
  using pred-preds-update assign₂ upd by metis
have p₂: pred P mem₂ ⟶ pred P' (mem₂(x := ev_A mem₂ e))
  using pred-preds-update assign₂ upd by metis
 have sec: tyenv-wellformed mds Γ S P ⟹ pred P mem₁ ⟹ tyenv-sec mds Γ
mem₁ ⟹ tyenv-sec mds' ?Γ' mem₁'
proof(clarify)
  assume wf: tyenv-wellformed mds Γ S P
  assume pred: pred P mem₁
  assume sec: tyenv-sec mds Γ mem₁
  from pred p have pred': pred P' mem₁' by blast
  fix v t'
  assume Γv: (Γ(x ↦ t)) v = Some t'
  assume v ∉ mds' AsmNoReadOrWrite
  show type-max (the ((Γ(x ↦ t)) v)) mem₁' ≤ dma mem₁' v
```

proof(*cases* $v = x$)
assume [*simp*]: $v = x$
hence [*simp*]: (*the* ((Γ($x \mapsto t$)) v)) $= t$ **and** *t-def*: $t = t'$
using Γv **by** *auto*
from ‹$v \notin$ *mds'* *AsmNoReadOrWrite*› *upd wf* **have** *readable*: $v \notin$ *snd* $\mathcal{S}$
by(*auto simp*: *tyenv-wellformed-def mds-consistent-def*)
with *assign*$_2$(*5*) **have** $t \leq_{:P}{}'$ (*dma-type x*) **by** *fastforce*
with *pred'* **show** *?thesis*
using *type-max-dma-type subtype-sound*
by *fastforce*
next
assume *neq*: $v \neq x$
hence [*simp*]: ((Γ($x \mapsto t$)) v) $=$ Γ v
by *simp*
with Γv **have** Γv: Γ v = *Some* t' **by** *simp*
with *sec upd* ‹$v \notin$ *mds'* *AsmNoReadOrWrite*› **have** *f-leq*: *type-max* t' *mem*$_1$ $\leq$ *dma mem*$_1$ v
by *auto*
have $\mathcal{C}$*-eq*: $\forall x{\in}\mathcal{C}$. *mem*$_1$ x = *mem*$_1{}'$ x
using *wf assign*$_2$(*1*) *upd* **by**(*auto simp*: *tyenv-wellformed-def mds-consistent-def*)
hence *dma-eq*: *dma mem*$_1$ = *dma mem*$_1{}'$
by(*rule dma-*$\mathcal{C}$)
have *f-eq*: *type-max* t' *mem*$_1$ = *type-max* t' *mem*$_1{}'$
apply(*rule* $\mathcal{C}$*-eq-type-max-eq*)
using Γv *wf* **apply**(*force simp*: *tyenv-wellformed-def types-wellformed-def*)
by(*rule* $\mathcal{C}$*-eq*)
from *neq* Γv *f-leq dma-eq f-eq* **show** *?thesis*
by *simp*
qed
qed

have ⟨*Stop*, *mds*, *mem*$_1$ (*x* := *ev*$_A$ *mem*$_1$ *e*)⟩ $\mathcal{R}^1{}_{?\Gamma',\mathcal{S},P'}$ ⟨*Stop*, *mds*, *mem*$_2$ (*x* := *ev*$_A$ *mem*$_2$ *e*)⟩
apply(*rule* $\mathcal{R}_1$*.intro*)
apply *blast*
using *wf assign*$_2$ **apply** *fastforce*
apply(*rule tyenv-eq'*)
using *p assign*$_2$ **apply** *fastforce*
using *p*$_2$ *assign*$_2$ **apply** *fastforce*
using *sec assign*$_2$
using *upd*(*2*) *upd*(*3*) **by** *blast*

ultimately have ⟨*x* ← *e*, *mds*, *mem*$_2$⟩ ⇝ ⟨*Stop*, *mds'*, *mem*$_2$ (*x* := *ev*$_A$ *mem*$_2$ *e*)⟩
⟨*Stop*, *mds'*, *mem*$_1$ (*x* := *ev*$_A$ *mem*$_1$ *e*)⟩ $\mathcal{R}^u{}_{\Gamma(x \mapsto t),\mathcal{S},P'}$ ⟨*Stop*, *mds'*, *mem*$_2$ (*x* := *ev*$_A$ *mem*$_2$ *e*)⟩
using $\mathcal{R}$*.intro*$_1$
by *auto*
thus *?case*

```
    using ‹mds′ = mds› ‹c₁′ = Stop› ‹mem₁′ = mem₁(x := ev_A mem₁ e)›
    by blast
next
  case (if-type Γ e t P 𝒮 th Γ′ 𝒮′ P′ el Γ″ P″ Γ‴ P‴)
  let ?P = if (ev_B mem₁ e) then P +_𝒮 e else P +_𝒮 (bexp-neg e)
  from ‹⟨Stmt.If e th el, mds, mem₁⟩ ⇝ ⟨c₁′, mds′, mem₁′⟩› have ty: ⊢ Γ,𝒮,?P
{c₁′} Γ‴,𝒮′,P‴
  proof (rule if-elim)
    assume c₁′ = th mem₁′ = mem₁ mds′ = mds ev_B mem₁ e
    with if-type(3)
    show ?thesis
      apply simp
      apply(erule sub)
        using if-type apply simp+
      done
  next
    assume c₁′ = el mem₁′ = mem₁ mds′ = mds ¬ ev_B mem₁ e
    with if-type(5)
    show ?thesis
      apply simp
      apply(erule sub)
        using if-type apply simp+
      done
  qed
  have ev_B-eq [simp]: ev_B mem₁ e = ev_B mem₂ e
    apply(rule ev_B-eq′)
      apply(rule ‹mem₁ =_Γ mem₂›)
     apply(rule ‹pred P mem₁›)
    apply(rule ‹Γ ⊢_b e ∈ t›)
    by(rule ‹ P ⊢ t›)
  have (⟨c₁′, mds, mem₁⟩, ⟨c₁′, mds, mem₂⟩) ∈ ℛ Γ‴ 𝒮′ P‴
    apply (rule intro₁)
    apply clarify
    apply (rule ℛ₁.intro [where Γ = Γ and Γ′ = Γ‴ and 𝒮 = 𝒮 and P = ?P])
       apply(rule ty)
      using ‹tyenv-wellformed mds Γ 𝒮 P›
      apply(auto simp: tyenv-wellformed-def mds-consistent-def add-pred-def)[1]
     apply(rule ‹mem₁ =_Γ mem₂›)
    using ‹pred P mem₁› apply(fastforce simp: pred-def add-pred-def bexp-neg-negates)
   using ‹pred P mem₂› apply(fastforce simp: pred-def add-pred-def bexp-neg-negates)
    by(rule ‹tyenv-sec mds Γ mem₁›)

  show ?case
  proof −
    from ev_B-eq if-type(13) have (⟨If e th el, mds, mem₂⟩ ⇝ ⟨c₁′, mds, mem₂⟩)
      apply (cases ev_B mem₁ e)
       apply (subgoal-tac c₁′ = th)
        apply clarify
       apply (metis cxt-to-stmt.simps(1) eval_w-simplep.if-true eval_wp.unannotated
```

```
eval_w p-eval_w-eq if-type(8))
      using if-type.prems(6) apply blast
     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))
     using if-type.prems(6) by blast
   with ‹⟨c_1', mds, mem_1⟩ R^u_{Γ''',S',P'''} ⟨c_1', mds, mem_2⟩› show ?thesis
   by (metis if-elim if-type.prems(6))
 qed
next
 case (while-type Γ e t P S c)
 hence [simp]: c_1' = (If e (c ;; While e c) Stop) and
   [simp]: mds' = mds and
   [simp]: mem_1' = mem_1
   by (auto simp: while-elim)

 with while-type have ⟨While e c, mds, mem_2⟩ ⇝ ⟨c_1', mds, mem_2⟩
  by (metis cxt-to-stmt.simps(1) eval_w-simplep.while eval_w p.unannotated eval_w p-eval_w-eq)

 moreover have ty: ⊢ Γ,S,P {c_1'} Γ,S,P
   apply simp
   apply(rule if-type)
            apply(rule while-type(1))
           apply(rule while-type(2))
          apply(rule seq-type)
           apply(rule while-type(3))
          apply(rule has-type.while-type)
            apply(rule while-type(1))
           apply(rule while-type(2))
          apply(rule while-type(3))
         apply(rule stop-type)
        apply simp+
     apply(rule add-pred-entailment)
    apply simp
   apply(blast intro!: add-pred-subset tyenv-wellformed-subset)
   done
 moreover
 have ⟨c_1', mds, mem_1⟩ R^1_{Γ,S,P} ⟨c_1', mds, mem_2⟩
   apply (rule R_1.intro [where Γ = Γ])
       apply(rule ty)
      using while-type apply simp+
   done
 hence ⟨c_1', mds, mem_1⟩ R^u_{Γ,S,P} ⟨c_1', mds, mem_2⟩
   using R.intro_1 by auto
 ultimately show ?case
   by fastforce
next
 case (sub Γ_1 S P_1 c Γ_1' S' P_1' Γ_2 P_2 Γ_2' P_2' mds c_1')
 have imp: tyenv-wellformed mds Γ_2 S P_2 ∧ pred P_2 mem_1 ∧ pred P_2 mem_2 ∧
```

tyenv-sec mds Γ_2 *mem*$_1$ $\Longrightarrow$
tyenv-wellformed mds Γ_1 $\mathcal{S}$ P_1 $\wedge$ *pred* P_1 *mem*$_1$ $\wedge$ *pred* P_1 *mem*$_2$ $\wedge$ *tyenv-sec mds* Γ_1 *mem*$_1$
apply(*rule conjI*)
using *sub*(*5*) *sub*(*4*) *tyenv-wellformed-sub* **unfolding** *pred-def*
apply *blast*
apply(*rule conjI*)
using *local.pred-def pred-entailment-def sub.hyps*(*7*) **apply** *auto*[*1*]
apply(*rule conjI*)
using *local.pred-def pred-entailment-def sub.hyps*(*7*) **apply** *auto*[*1*]
using *sub*(*3*) *context-equiv-tyenv-sec* **unfolding** *pred-def* **by** *blast*

have *tyenv-eq*: *mem*$_1$ $=_{\Gamma_1}$ *mem*$_2$
using *context-equiv-tyenv-eq sub* **by** *blast*

from *imp tyenv-eq* **obtain** *c*$_2$′ *mem*$_2$′ **where** *c*$_2$′*-props*: $\langle c, mds, mem_2\rangle \rightsquigarrow \langle c_2', mds', mem_2'\rangle$
$\langle c_1', mds', mem_1'\rangle$ $\mathcal{R}^u{}_{\Gamma_1',\mathcal{S}',P_1'}$ $\langle c_2', mds', mem_2'\rangle$
using *sub* **by** *blast*
with *R-equiv-entailment* ‹*P*$_1$′ $\vdash$ *P*$_2$′› **show** *?case*
using *sub.hyps*(*6*) *sub.hyps*(*5*) **by** *blast*
next case (*await-type* Γ *e t P* $\mathcal{S}$ *c* Γ' $\mathcal{S}'$ P')
from *await-type.prems* **have** *ev*$_B$ *mem*$_1$ *e no-await c is-final c*$_1$′ **and** *step*: $\langle c, mds, mem_1\rangle \rightsquigarrow^+ \langle c_1', mds', mem_1'\rangle$
using *await-elim* **by** *simp*+
from *await-type.prems* ‹$\Gamma \vdash_b e \in t$› ‹$P \vdash t$› **have** *pred P* $+_{\mathcal{S}}$ *e mem*$_1$ *pred P* $+_{\mathcal{S}}$ *e mem*$_2$ *ev*$_B$ *mem*$_2$ *e*
using *pred-plus-impl* ‹*ev*$_B$ *mem*$_1$ *e*› ‹*pred P mem*$_1$› *ev*$_B$*-eq*′
by *blast*+
from *await-type.prems* ‹$\Gamma \vdash_b e \in t$› ‹$P \vdash t$› **have** *wellformed*: *tyenv-wellformed mds* Γ $\mathcal{S}$ *P* $+_{\mathcal{S}}$ *e*
apply (*unfold add-pred-def*)[*1*]
apply (*case-tac pred-stable* $\mathcal{S}$ *e*, *clarsimp*)
apply (*unfold tyenv-wellformed-def*, *clarsimp*)[*1*]
apply (*unfold mds-consistent-def*, *clarsimp*)[*1*]
by *clarsimp*
from *step* ‹*is-final c*$_1$′› ‹*no-await c*› ‹*tyenv-wellformed mds* Γ $\mathcal{S}$ *P* $+_{\mathcal{S}}$ *e*› *await-type.prems* ‹*pred P* $+_{\mathcal{S}}$ *e mem*$_1$› ‹*pred P* $+_{\mathcal{S}}$ *e mem*$_2$›
obtain *c*$_2$′ *mem*$_2$′ **where** *step*: $\langle c, mds, mem_2\rangle \rightsquigarrow^+ \langle c_2', mds', mem_2'\rangle$ **and**
rel: $\langle c_1', mds', mem_1'\rangle$ $\mathcal{R}^u{}_{\Gamma',\mathcal{S}',P'}$ $\langle c_2', mds', mem_2'\rangle$ *is-final c*$_2$′
using $\mathcal{R}$*-typed-step-plus await-type.hyps*(*3*) *is-final-*$\mathcal{R}_u$*-is-final* **by** *meson*
from *wellformed* ‹*is-final c*$_2$′› ‹*ev*$_B$ *mem*$_2$ *e*› ‹*no-await c*› ‹$\Gamma \vdash_b e \in t$› ‹$P \vdash t$› ‹*pred P* $+_{\mathcal{S}}$ *e mem*$_2$› *step rel* **show** *?case*
using *eval*$_w$*.intros*(*4*) **by** *blast*
qed

lemma $\mathcal{R}_1$*-weak-bisim*:
weak-bisim ($\mathcal{R}_1$ Γ' $\mathcal{S}'$ P') ($\mathcal{R}$ Γ' $\mathcal{S}'$ P')

unfolding *weak-bisim-def*
apply *clarsimp*
apply(*erule* $\mathcal{R}_1$*-elim*)
apply(*blast intro*: $\mathcal{R}$*-typed-step*)
done

lemma $\mathcal{R}$*-to-*$\mathcal{R}_3$: $[\![\langle c_1,\ mds,\ mem_1\rangle\ \mathcal{R}^u{}_{\Gamma,\mathcal{S},P}\ \langle c_2,\ mds,\ mem_2\rangle\ ;\vdash \Gamma,\mathcal{S},P\ \{\ c\ \}\ \Gamma',\mathcal{S}',P'\ ]\!] \Longrightarrow$
$\langle c_1\ ;;\ c,\ mds,\ mem_1\rangle\ \mathcal{R}^3{}_{\Gamma',\mathcal{S}',P'}\ \langle c_2\ ;;\ c,\ mds,\ mem_2\rangle$
apply (*erule* $\mathcal{R}$*-elim*)
by *auto*

lemma $\mathcal{R}_3$*-weak-bisim*:
weak-bisim $(\mathcal{R}_3\ \Gamma'\ \mathcal{S}'\ P')\ (\mathcal{R}\ \Gamma'\ \mathcal{S}'\ P')$
proof −
{
fix c_1 *mds* mem_1 c_2 mem_2 $c_1{}'$ *mds'* $mem_1{}'$
assume *case3*: $(\langle c_1,\ mds,\ mem_1\rangle, \langle c_2,\ mds,\ mem_2\rangle) \in \mathcal{R}_3\ \Gamma'\ \mathcal{S}'\ P'$
assume *eval*: $\langle c_1,\ mds,\ mem_1\rangle \rightsquigarrow \langle c_1{}',\ mds',\ mem_1{}'\rangle$
have $\exists\ c_2{}'\ mem_2'.\ \langle c_2,\ mds,\ mem_2\rangle \rightsquigarrow \langle c_2{}',\ mds',\ mem_2{}'\rangle \wedge \langle c_1{}',\ mds',\ mem_1{}'\rangle\ \mathcal{R}^u{}_{\Gamma',\mathcal{S}',P'}\ \langle c_2{}',\ mds',\ mem_2{}'\rangle$
using *case3 eval*
apply *simp*

proof (*induct arbitrary*: $c_1{}'$ *rule*: $\mathcal{R}_3$*-aux.induct*)
case (*intro*$_1$ c_1 *mds* mem_1 Γ $\mathcal{S}$ P c_2 mem_2 c Γ' $\mathcal{S}'$ P')
hence [*simp*]: $c_2 = c_1$
by (*metis* (*lifting*) $\mathcal{R}_1$*-elim*)
thus *?case*
proof (*cases* c_1 = *Stop*)
assume [*simp*]: c_1 = *Stop*
from *intro*$_1$(*1*) **show** *?thesis*
apply (*rule* $\mathcal{R}_1$*-elim*)
apply *simp*
apply (*rule-tac* x = c **in** *exI*)
apply (*rule-tac* x = mem_2 **in** *exI*)
apply (*rule conjI*)
apply (*metis* ‹c_1 = *Stop*› *cxt-to-stmt.simps*(*1*) *eval*$_w$*-simplep.seq-stop eval*$_w$*p.unannotated eval*$_w$*p-eval*$_w$*-eq intro*$_1$*.prems seq-stop-elim*)
apply (*rule* $\mathcal{R}$*.intro*$_1$, *clarify*)
apply (*subgoal-tac* $c_1{}' = c$)
apply *simp*
apply (*rule* $\mathcal{R}_1$*.intro*)
apply(*rule intro*$_1$(*2*))
apply (*metis* (*no-types*, *lifting*) ‹c_1 = *Stop*› *intro*$_1$*.prems seq-stop-elim stop-cxt tyenv-wellformed-sub*)
using ‹c_1 = *Stop*› *intro*$_1$*.prems seq-stop-elim stop-cxt context-equiv-tyenv-eq*

apply *metis*

using ‹c_1 = *Stop*› *intro*$_1$.*prems pred-entailment-def seq-stop-elim stop-cxt*
apply *blast*
using *pred-entailment-def stop-cxt* **apply** *blast*

apply (*metis* (*no-types*, *lifting*) ‹c_1 = *Stop*› *context-equiv-def intro*$_1$.*prems less-eq-Sec-def seq-stop-elim stop-cxt subtype-sound type-equiv-def*)
using *intro*$_1$.*prems seq-stop-elim* **by** *auto*
next
assume $c_1 \neq$ *Stop*
from *intro*$_1$
obtain $c_1{}''$ **where** $\langle c_1, mds, mem_1\rangle \rightsquigarrow \langle c_1{}'', mds', mem_1{}'\rangle \wedge c_1{}' = (c_1{}'' ;; c)$
by (*metis* ‹$c_1 \neq$ *Stop*› *intro*$_1$.*prems seq-elim*)
with *intro*$_1$
obtain $c_2{}''$ $mem_2{}'$ **where** $\langle c_2, mds, mem_2\rangle \rightsquigarrow \langle c_2{}'', mds', mem_2{}'\rangle$ $\langle c_1{}'', mds', mem_1{}'\rangle$ $\mathcal{R}^u{}_{\Gamma,\mathcal{S},P}$ $\langle c_2{}'', mds', mem_2{}'\rangle$
using $\mathcal{R}_1$-*weak-bisim* **and** *weak-bisim-def*
by *blast*
thus *?thesis*
using *intro*$_1$(*2*) $\mathcal{R}$-*to-*$\mathcal{R}_3$
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 *auto*
apply (*rule* $\mathcal{R}$.*intro*$_3$)

by (*simp add*: $\mathcal{R}$-*to-*$\mathcal{R}_3$ ‹$\langle c_1, mds, mem_1\rangle \rightsquigarrow \langle c_1{}'', mds', mem_1{}'\rangle \wedge c_1{}' = c_1{}'' ;; c$›)
qed
next
case (*intro*$_3$ c_1 *mds* mem_1 Γ $\mathcal{S}$ P c_2 mem_2 c Γ' $\mathcal{S}'$ P')
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** $\langle c_2, mds, mem_2\rangle \rightsquigarrow \langle c_2{}'', mds', mem_2{}'\rangle$ $\langle c_1{}'', mds', mem_1{}'\rangle$ $\mathcal{R}^u{}_{\Gamma,\mathcal{S},P}$ $\langle c_2{}'', mds', mem_2{}'\rangle$
using *intro*$_3$(*2*) **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-*$\mathcal{R}_3$ $\langle\langle c_1'', mds', mem_1'\rangle\ \mathcal{R}^u{}_{\Gamma,\mathcal{S},P}\ \langle c_2'', mds', mem_2'\rangle\rangle$ $\langle c_1' = c_1''\ ;;\ c\rangle$ $intro_3(\mathcal{3})$)
apply (*metis* (*lifting*) $\mathcal{R}.intro_3$ $\mathcal{R}$*-to-*$\mathcal{R}_3$ $\langle\langle c_1'', mds', mem_1'\rangle\ \mathcal{R}^u{}_{\Gamma,\mathcal{S},P}\ \langle c_2'', mds', mem_2'\rangle\rangle$ $\langle c_1' = c_1''\ ;;\ c\rangle$ $intro_3(\mathcal{3})$)
done
qed
qed
}
thus *?thesis*
unfolding *weak-bisim-def*
by *auto*
qed

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

lemma *Typed-in-*$\mathcal{R}$:
assumes *typeable*: $\vdash \Gamma,\mathcal{S},P\ \{\ c\ \}\ \Gamma',\mathcal{S}',P'$
assumes *wf*: *tyenv-wellformed mds* $\Gamma\ \mathcal{S}\ P$
assumes *mem-eq*: $\forall\ x.$ *type-max* (*to-total* $\Gamma\ x$) $mem_1 = Low \longrightarrow mem_1\ x = mem_2\ x$
assumes $pred_1$: *pred* $P\ mem_1$
assumes $pred_2$: *pred* $P\ mem_2$

assumes *tyenv-sec*: *tyenv-sec mds* Γ mem_1
shows $\langle c, mds, mem_1\rangle$ $\mathcal{R}^u{}_{\Gamma',\mathcal{S}',P'}$ $\langle c, mds, mem_2\rangle$
apply (*rule* $\mathcal{R}$.*intro*$_1$ [*of* Γ'])
apply *clarify*
apply (*rule* $\mathcal{R}_1$.*intro* [*of* Γ])
apply(*rule typeable*)
apply(*rule wf*)
using *mem-eq* **apply**(*fastforce simp*: *tyenv-eq-def*)
using *assms* **by** *simp*+

theorem *type-soundness*:
assumes *well-typed*: $\vdash \Gamma,\mathcal{S},P \{ c \} \Gamma',\mathcal{S}',P'$
assumes *wf*: *tyenv-wellformed mds* Γ $\mathcal{S}$ P
assumes *mem-eq*: $\forall$ *x. type-max* (*to-total* Γ *x*) mem_1 = *Low* $\longrightarrow$ mem_1 *x* = mem_2 *x*
assumes *pred*$_1$: *pred P* mem_1
assumes *pred*$_2$: *pred P* mem_2
assumes *tyenv-sec*: *tyenv-sec mds* Γ mem_1
shows $\langle c, mds, mem_1\rangle \approx \langle c, mds, mem_2\rangle$
using $\mathcal{R}$-*bisim Typed-in-*$\mathcal{R}$
by (*metis assms mem-eq mm-equiv.simps well-typed*)

definition
Γ-*of-mds* :: *'Var Mds* $\Rightarrow$ (*'Var*,*'BExp*) *TyEnv*
where
Γ-*of-mds mds* $\equiv$ (λ*x. if x* $\notin \mathcal{C} \wedge$ *x* $\in$ *mds AsmNoWrite* $\cup$ *mds AsmNoReadOrWrite then*
if x $\in$ *mds AsmNoReadOrWrite then*
Some ({*pred-False*})
else
Some (*dma-type x*)
else None)

definition
$\mathcal{S}$-*of-mds* :: *'Var Mds* $\Rightarrow$ *'Var Stable*
where
$\mathcal{S}$-*of-mds mds* $\equiv$ (*mds AsmNoWrite*, *mds AsmNoReadOrWrite*)

definition
mds-yields-stable-types :: *'Var Mds* $\Rightarrow$ *bool*
where
mds-yields-stable-types mds $\equiv \forall$ *x. x* $\in$ *mds AsmNoWrite* $\cup$ *mds AsmNoReadOrWrite* $\longrightarrow$
($\forall$ *v*$\in\mathcal{C}$-*vars x. v* $\in$ *mds AsmNoWrite* $\cup$ *mds AsmNoReadOrWrite*)

inductive

type-global :: (('*Var*, '*AExp*, '*BExp*) *Stmt* × '*Var Mds*) *list* ⇒ *bool*
(⊢ - [*120*] *1000*)
where
⟦ *list-all* (λ (*c*,*m*). (∃ Γ′ $\mathcal{S}$′ *P*′. ⊢ (Γ-*of-mds m*),($\mathcal{S}$-*of-mds m*),{} { *c* } Γ′,$\mathcal{S}$′,*P*′)
∧ *mds-yields-stable-types m*) *cs* ;
∀ *mem*. *sound-mode-use* (*cs*, *mem*)
⟧ ⟹
type-global cs

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

lemma *of-mds-tyenv-wellformed*: *mds-yields-stable-types m* ⟹ *tyenv-wellformed*
m (Γ-*of-mds m*) ($\mathcal{S}$-*of-mds m*) {}
apply(*auto simp*: *tyenv-wellformed-def* Γ-*of-mds-def* $\mathcal{S}$-*of-mds-def mds-consistent-def*
stable-def
types-wellformed-def types-stable-def mds-yields-stable-types-def
type-wellformed-def dma-$\mathcal{C}$*-vars* $\mathcal{C}$*-def bexp-vars-pred-False*
$\mathcal{C}$*-vars-correct*
split: *if-splits*)
done

lemma Γ-*of-mds-tyenv-sec*:
tyenv-sec m (Γ-*of-mds m*) *mem*$_1$
apply(*auto simp*: Γ-*of-mds-def*)
done

lemma *type-max-pred-False* [*simp*]:
type-max {*pred-False*} *mem* = *High*
apply(*simp add*: *type-max-def pred-False-is-False*)
done

lemma *typed-secure*:
⟦ ⊢ (Γ-*of-mds m*),($\mathcal{S}$-*of-mds m*),{} { *c* } Γ′,$\mathcal{S}$′,*P*′; *mds-yields-stable-types m* ⟧ ⟹
com-sifum-secure (*c*,*m*)
apply (*clarsimp simp*: *com-sifum-secure-def low-indistinguishable-def*)
apply (*erule type-soundness*)
apply(*erule of-mds-tyenv-wellformed*)
apply(*auto simp*: *to-total-def split*: *if-split simp*: Γ-*of-mds-def low-mds-eq-def*)[*1*]
apply(*fastforce simp*: *pred-def type-max-def*)
apply(*fastforce simp*: *pred-def*)
by(*rule* Γ-*of-mds-tyenv-sec*)

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*
shows *prog-sifum-secure-cont cs*

using *typeable*
apply (*rule type-global-elim*)
apply (*subgoal-tac* $\forall\ c \in$ *set cs. com-sifum-secure c*)
apply(*rule sifum-compositionality-cont*)
using *list-all-set* **apply** *fastforce*
apply *fastforce*
apply(*drule list-all-iff*[*THEN iffD1*])
apply *clarsimp*
apply(*rename-tac c m*)
apply(*drule-tac x*=(*c*,*m*) **in** *bspec*)
apply *assumption*
apply *clarsimp*
using *typed-secure* **by** *blast*

end
end

6 Type System for Ensuring Locally Sound Use of Modes

theory *LocallySoundModeUse*
imports *Security Language*
begin

6.1 Typing Rules

locale *sifum-modes* =
sifum-lang-no-dma ev_A ev_B *aexp-vars bexp-vars* + *sifum-security dma* $\mathcal{C}$*-vars* $\mathcal{C}$ $eval_w$ *undefined*
for ev_A :: (*'Var*, *'Val*) *Mem* ⇒ *'AExp* ⇒ *'Val*
and ev_B :: (*'Var*, *'Val*) *Mem* ⇒ *'BExp* ⇒ *bool*
and *aexp-vars* :: *'AExp* ⇒ *'Var set*
and *bexp-vars* :: *'BExp* ⇒ *'Var set*
and *dma* :: (*'Var*,*'Val*) *Mem* ⇒ *'Var* ⇒ *Sec*
and $\mathcal{C}$*-vars* :: *'Var* ⇒ *'Var set*
and $\mathcal{C}$ :: *'Var set*

context *sifum-modes*
begin

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

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

(**infix** $\oplus$ *140*)
where
update-annos mds [] = *mds* |
update-annos mds (*a* # *as*) = *update-annos* (*update-modes a mds*) *as*

fun
annotate :: ('*Var*, '*AExp*, '*BExp*) *Stmt* ⇒ '*Var ModeUpd list* ⇒ ('*Var*, '*AExp*, '*BExp*) *Stmt*
(**infix** $\otimes$ *140*)
where
annotate c [] = *c* |
annotate c (*a* # *as*) = (*annotate c as*)@[*a*]

inductive
mode-type :: '*Var Mds* ⇒ ('*Var*, '*AExp*, '*BExp*) *Stmt* ⇒ '*Var Mds* ⇒ *bool* (⊢ - { - } -)
where
skip: ⊢ *mds* { *Skip* ⊗ *annos* } (*mds* ⊕ *annos*) |
assign: ⟦ *x* ∉ *mds GuarNoWrite* ∪ *mds GuarNoReadOrWrite* ; *aexp-vars e* ∩ *mds GuarNoReadOrWrite* = {};
∀ *v*. (*x* ∈ *C-vars v* ⟶ *v* ∉ *mds GuarNoWrite* ∪ *mds GuarNoReadOrWrite*) ∧
(*C-vars v* ∩ *aexp-vars e* ≠ {} ⟶ *v* ∉ *mds GuarNoReadOrWrite*)⟧
⟹
⊢ *mds* { (*x* ← *e*) ⊗ *annos* } (*mds* ⊕ *annos*) |
if-: ⟦ ⊢ (*mds* ⊕ *annos*) { *c*$_1$ } *mds*'' ;
⊢ (*mds* ⊕ *annos*) { *c*$_2$ } *mds*'' ;
bexp-vars e ∩ *mds GuarNoReadOrWrite* = {};
∀ *v*. *C-vars v* ∩ *bexp-vars e* ≠ {} ⟶ *v* ∉ *mds GuarNoReadOrWrite*⟧ ⟹
⊢ *mds* { *If e c*$_1$ *c*$_2$ ⊗ *annos* } *mds*'' |
while: ⟦ *mds*' = *mds* ⊕ *annos* ; ⊢ *mds*' { *c* } *mds*' ; *bexp-vars e* ∩ *mds*' *GuarNoReadOrWrite* = {};
∀ *v*. *C-vars v* ∩ *bexp-vars e* ≠ {} ⟶ *v* ∉ *mds*' *GuarNoReadOrWrite*⟧
⟹
⊢ *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$_1$ ≤ *mds*$_2$ ⟹ (*mds*$_1$ ⊕ *annos*) ≤ (*mds*$_2$ ⊕ *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-or-modify c x $\Longrightarrow$ *doesnt-read-or-modify* (*c* $\otimes$ *annos*) *x*
unfolding *doesnt-read-or-modify-def doesnt-read-or-modify-vars-def*
apply *clarify*
apply (*induct annos*)
apply *simp*
apply (*metis* (*lifting*))
apply *clarsimp*
apply (*rule cxt-eval*)
apply (*rule* $eval_w$*.decl*)
apply (*metis cxt-eval* $eval_w$*.decl upd-elim*)+
done

lemma *doesnt-modify-annos*:
doesnt-modify c x $\Longrightarrow$ *doesnt-modify* (*c* $\otimes$ *annos*) *x*
unfolding *doesnt-modify-def*
apply *clarsimp*
apply (*induct annos*)
apply *simp*
by (*metis annotate.simps*(*2*) *upd-elim*)

lemma *stop-loc-reach*:
$[\![\langle c', mds', mem' \rangle \in$ *loc-reach* $\langle$*Stop*, *mds*, *mem*$\rangle$ $]\!] \Longrightarrow$
$c' =$ *Stop* $\wedge$ $mds' = mds$
apply (*induct rule*: *loc-reach.induct*)
by (*auto simp*: *stop-no-eval*)

lemma *stop-doesnt-access*:
doesnt-modify Stop x $\wedge$ *doesnt-read-or-modify Stop x*
unfolding *doesnt-modify-def* **and** *doesnt-read-or-modify-def doesnt-read-or-modify-vars-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*:
⟦ ⟨*Skip* ⊗ *annos*, *mds*, *mem*⟩ ⇝ ⟨*c'*, *mds'*, *mem'*⟩ ⟧ ⟹ *c'* = *Stop* ∧ *mds'* = *mds* ⊕ *annos* ∧ *mem'* = *mem*
apply (*rule ccontr*)
apply (*insert skip-eval-step deterministic*)
apply *clarify*
apply *clarsimp*
by *metis*+

lemma *skip-doesnt-read*:
doesnt-read-or-modify (*Skip* ⊗ *annos*) *x*
apply (*rule doesnt-read-annos*)
apply (*clarsimp simp*: *doesnt-read-or-modify-def doesnt-read-or-modify-vars-def*)
by (*metis annotate.simps*(*1*) *skip-elim skip-eval-step*)+

lemma *skip-doesnt-write*:
doesnt-modify (*Skip* ⊗ *annos*) *x*
apply (*rule doesnt-modify-annos*)
apply (*clarsimp simp*: *doesnt-modify-def*)
by (*metis skip-elim*)+

lemma *skip-loc-reach*:
⟦ ⟨*c'*, *mds'*, *mem'*⟩ ∈ *loc-reach* ⟨*Skip* ⊗ *annos*, *mds*, *mem*⟩ ⟧ ⟹
(*c'* = *Stop* ∧ *mds'* = (*mds* ⊕ *annos*)) ∨ (*c'* = *Skip* ⊗ *annos* ∧ *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* ∈ *loc-reach* ⟨*Skip* ⊗ *annos*, *mds*, *mem*⟩ ; *lc* = ⟨*c'*, *mds'*, *mem'*⟩ ⟧ ⟹ *doesnt-read-or-modify c' x* ∧ *doesnt-modify c' x*
apply (*subgoal-tac* (*c'* = *Stop* ∧ *mds'* = (*mds* ⊕ *annos*)) ∨ (*c'* = *Skip* ⊗ *annos* ∧ *mds'* = *mds*))
apply (*rule conjI*, *erule disjE*)
apply (*simp add*: *doesnt-read-or-modify-def doesnt-read-or-modify-vars-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*:
⟦ $x \neq y$; $x \notin \mathcal{C}$-*vars* 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 dma-*$\mathcal{C}$*-vars*)

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 *clarsimp*
by (*metis* (*lifting*))

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*:
⟦ $y \neq x$; $y \notin$ *aexp-vars* e; $x \notin \mathcal{C}$-*vars* y; $\mathcal{C}$-*vars* $y \cap$ *aexp-vars* $e = \{\}$ ⟧ ⟹
doesnt-read-or-modify (($x \leftarrow e$) ⊗ *annos*) y
apply (*rule doesnt-read-annos*)
proof −
assume $y \neq x$
$y \notin$ *aexp-vars* e
$x \notin \mathcal{C}$-*vars* y
$\mathcal{C}$-*vars* $y \cap$ *aexp-vars* $e = \{\}$
thus *doesnt-read-or-modify* ($x \leftarrow e$) y
unfolding *doesnt-read-or-modify-def doesnt-read-or-modify-vars-def*
apply −
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 *clarify*
apply (*rule conjI*)
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*)
apply *clarify*
apply (*rename-tac va v*)
apply (*subgoal-tac mem* (x := ev_A *mem* e, *va* := v) = *mem* (*va* := v, x := ev_A *mem* e))
apply *simp*
apply (*rule cxt-eval*)
apply (*rule* $eval_w$*.unannotated*)
apply (*subgoal-tac va* $\notin$ *aexp-vars e*)
apply (*metis* $eval_w$*-simple.assign eval-vars-det*$_A$ *fun-upd-apply*)
apply *blast*
apply (*metis mem-upd-commute*)
apply (*metis assign-elim*)
done
qed

lemma *assign-loc-reach*:
⟦ ⟨c', *mds*$'$, *mem*$'$⟩ ∈ *loc-reach* ⟨(x ← e) ⊗ *annos*, *mds*, *mem*⟩ ⟧ ⟹
(c' = *Stop* ∧ *mds*$'$ = (*mds* ⊕ *annos*)) ∨ (c' = (x ← e) ⊗ *annos* ∧ *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 ⊗ *annos*) x
apply (*rule doesnt-modify-annos*)
by (*auto simp*: *doesnt-modify-def*)

lemma $vars\text{-}eval_B$:
x ∉ *bexp-vars e* ⟹ 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* ⟹ $\mathcal{C}$*-vars* x ∩ *bexp-vars e* = {} ⟹ *doesnt-read-or-modify* (*If e* c_1 c_2 ⊗ *annos*) x
apply (*rule doesnt-read-annos*)
apply (*clarsimp simp*: *doesnt-read-or-modify-def doesnt-read-or-modify-vars-def*, *safe*)
apply (*rename-tac mds mem c*$'$ *mds*$'$ *mem*$'$ v)

apply (*case-tac* ev_B *mem e*)
apply (*subgoal-tac* $c' = c_1 \wedge mds' = mds \wedge mem' = mem$)
prefer *2*
apply *auto*[*1*]
apply *clarsimp*
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* $c' = c_2 \wedge mds' = mds \wedge mem' = mem$)
prefer *2*
apply *auto*[*1*]
apply *clarsimp*
apply (*rule cxt-eval*)
apply (*rule* $eval_w$*.unannotated*)
apply (*rule* $eval_w$*-simple.if-false*)
apply (*metis* (*lifting*) *vars-eval*$_B$)
apply (*rename-tac mds mem* c' mds' mem' *va v*)
apply (*case-tac* ev_B *mem e*)
apply (*subgoal-tac* $c' = c_1 \wedge mds' = mds \wedge mem' = mem$)
prefer *2*
apply *auto*[*1*]
apply *clarsimp*
apply (*rule cxt-eval*)
apply (*rule* $eval_w$*.unannotated*)
apply (*rule* $eval_w$*-simple.if-true*)
apply (*subgoal-tac va* $\notin$ *bexp-vars e*)
apply (*metis* (*lifting*) *vars-eval*$_B$)
apply *blast*
apply (*subgoal-tac* $c' = c_2 \wedge mds' = mds \wedge mem' = mem$)
prefer *2*
apply *auto*[*1*]
apply *clarsimp*
apply (*rule cxt-eval*)
apply (*rule* $eval_w$*.unannotated*)
apply (*rule* $eval_w$*-simple.if-false*)
apply (*subgoal-tac va* $\notin$ *bexp-vars e*)
apply (*metis* (*lifting*) *vars-eval*$_B$)
apply *blast*
done

lemma *if-eval-true*:
$[\![ev_B\ mem\ e]\!] \Longrightarrow$
$\langle If\ e\ c_1\ c_2 \otimes annos,\ mds,\ mem\rangle \rightsquigarrow \langle c_1,\ mds \oplus annos,\ mem\rangle$
apply (*induct annos arbitrary*: *mds*)
apply *simp*
apply (*metis cxt-eval* $eval_w$*.unannotated* $eval_w$*-simple.if-true*)
by (*metis annotate.simps*(*2*) *cxt-eval* $eval_w$*.decl update-annos.simps*(*2*))

lemma *if-eval-false*:
⟦ ¬ 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 *blast*
using *if-eval-false deterministic*
by *blast*

lemma *if-eval-elim*′:
⟦ ⟨*If e* c_1 c_2, *mds*, *mem*⟩ ⇝ ⟨c', *mds*′, *mem*′⟩ ⟧ ⟹
((*c*′ = c_1 ∧ ev_B *mem e*) ∨ (*c*′ = c_2 ∧ ¬ ev_B *mem e*)) ∧ *mds*′ = *mds* ∧ *mem*′ = *mem*
using *if-eval-elim* [**where** *annos* = []]
by *auto*

lemma *loc-reach-refl*′:
⟨*c*, *mds*, *mem*⟩ ∈ *loc-reach* ⟨*c*, *mds*, *mem*⟩
apply (*subgoal-tac* ∃ *lc*. *lc* ∈ *loc-reach lc* ∧ *lc* = ⟨*c*, *mds*, *mem*⟩)
apply *blast*
by (*metis loc-reach.refl fst-conv snd-conv*)

lemma *if-loc-reach*:
⟦ ⟨c', *mds*′, *mem*′⟩ ∈ *loc-reach* ⟨*If e* c_1 c_2 ⊗ *annos*, *mds*, *mem*⟩ ⟧ ⟹
(*c*′ = *If e* c_1 c_2 ⊗ *annos* ∧ *mds*′ = *mds*) ∨
(∃ *mem*″. ⟨c', *mds*′, *mem*′⟩ ∈ *loc-reach* ⟨c_1, *mds* ⊕ *annos*, *mem*″⟩) ∨
(∃ *mem*″. ⟨c', *mds*′, *mem*′⟩ ∈ *loc-reach* ⟨c_2, *mds* ⊕ *annos*, *mem*″⟩)
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'*:
⟦ ⟨*c'*, *mds'*, *mem'*⟩ ∈ *loc-reach* ⟨*If e* c_1 c_2, *mds*, *mem*⟩ ⟧ ⟹
(*c'* = *If e* c_1 c_2 ∧ *mds'* = *mds*) ∨
(∃ *mem''*. ⟨*c'*, *mds'*, *mem'*⟩ ∈ *loc-reach* ⟨c_1, *mds*, *mem''*⟩) ∨
(∃ *mem''*. ⟨*c'*, *mds'*, *mem'*⟩ ∈ *loc-reach* ⟨c_2, *mds*, *mem''*⟩)
using *if-loc-reach* [**where** *annos* = []]
by *simp*

lemma *seq-loc-reach*:
⟦ ⟨*c'*, *mds'*, *mem'*⟩ ∈ *loc-reach* ⟨c_1 ;; c_2, *mds*, *mem*⟩ ⟧ ⟹
(∃ *c''*. *c'* = *c''* ;; c_2 ∧ ⟨*c''*, *mds'*, *mem'*⟩ ∈ *loc-reach* ⟨c_1, *mds*, *mem*⟩) ∨
(∃ *c'' mds'' mem''*. ⟨*Stop*, *mds''*, *mem''*⟩ ∈ *loc-reach* ⟨c_1, *mds*, *mem*⟩ ∧
⟨*c'*, *mds'*, *mem'*⟩ ∈ *loc-reach* ⟨c_2, *mds''*, *mem''*⟩)
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-or-modify c x* ⟧ ⟹ *doesnt-read-or-modify* (*c* ;; *c'*) *x*
apply (*clarsimp simp*: *doesnt-read-or-modify-def doesnt-read-or-modify-vars-def*
| *safe*)+
apply (*rename-tac mds mem c'a mds' mem' v*)
apply (*case-tac c* = *Stop*)
apply *clarsimp*
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*)
apply (*metis* (*lifting*, *no-types*) $eval_w$*.seq seq-elim*)
apply (*rename-tac mds mem c'a mds' mem' va v*)
apply (*case-tac c* = *Stop*)
apply *clarsimp*
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*)
apply (*metis* (*lifting*, *no-types*) $eval_w$*.seq seq-elim*)
done

lemma *seq-doesnt-modify*:
⟦ *doesnt-modify c x* ⟧ ⟹ *doesnt-modify* (*c* ;; *c'*) *x*
apply (*clarsimp simp*: *doesnt-modify-def* | *safe*)+
apply (*case-tac c* = *Stop*)

apply *clarsimp*
apply (*metis* (*lifting*) *seq-stop-elim*)
apply (*metis* (*no-types*) *seq-elim*)
apply (*case-tac c = Stop*)
apply *clarsimp*
apply (*metis* (*lifting*) *seq-stop-elim*)
apply (*metis* (*no-types*) *seq-elim*)
done

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

lemma *seq-stop-elim*: $\langle Stop\ ;;\ c,\ mds,\ mem\rangle \rightsquigarrow \langle c',\ mds',\ mem'\rangle \Longrightarrow$
$c' = c \wedge mds' = mds \wedge mem' = mem$
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*(*34*) *Stmt.simps*(*42*) *cxt-seq-elim*)

lemma *seq-split*:
$[\![\langle Stop,\ mds',\ mem'\rangle \in loc\text{-}reach\ \langle c_1\ ;;\ c_2,\ mds,\ mem\rangle\ ]\!] \Longrightarrow$
$\exists\ mds''\ mem''.\ \langle Stop,\ mds'',\ mem''\rangle \in loc\text{-}reach\ \langle c_1,\ mds,\ mem\rangle \wedge$
$\langle Stop,\ mds',\ mem'\rangle \in loc\text{-}reach\ \langle c_2,\ mds'',\ mem''\rangle$
apply (*drule seq-loc-reach*)
by (*metis Stmt.simps*(*49*))

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

lemma *while-eval'*:
$\langle While\ e\ c,\ mds,\ mem\rangle \rightsquigarrow \langle If\ e\ (c\ ;;\ While\ e\ c)\ Stop,\ mds,\ mem\rangle$
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'*:

⟦ ⟨*While e c*, *mds*, *mem*⟩ ⇝ ⟨*c'*, *mds'*, *mem'*⟩ ⟧ ⟹
(*c'* = *If e* (*c* ;; *While e c*) *Stop* ∧ *mds'* = *mds* ∧ *mem'* = *mem*)
using *while-eval-elim* [**where** *annos* = []]
by *auto*

lemma *while-doesnt-read*:
⟦ *x* ∉ *bexp-vars e* ⟧ ⟹ *doesnt-read-or-modify* (*While e c* ⊗ *annos*) *x*
unfolding *doesnt-read-or-modify-def doesnt-read-or-modify-vars-def*
using *while-eval while-eval-elim*
by *metis*

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

lemma *disjE3*:
⟦ *A* ∨ *B* ∨ *C* ; *A* ⟹ *P* ; *B* ⟹ *P* ; *C* ⟹ *P* ⟧ ⟹ *P*
by *auto*

lemma *disjE5*:
⟦ *A* ∨ *B* ∨ *C* ∨ *D* ∨ *E* ; *A* ⟹ *P* ; *B* ⟹ *P* ; *C* ⟹ *P* ; *D* ⟹ *P* ; *E* ⟹ *P* ⟧
⟹ *P*
by *auto*

lemma *if-doesnt-read'*:
x ∉ *bexp-vars e* ⟹ *C*-*vars x* ∩ *bexp-vars e* = {} ⟹ *doesnt-read-or-modify* (*If e* c_1 c_2) *x*
using *if-doesnt-read* [**where** *annos* = []]
by *auto*

theorem *mode-type-sound*:
assumes *typeable*: ⊢ mds_1 { *c* } mds_1'
assumes *mode-le*: $mds_2 \leq mds_1$
shows ∀ *mem*. (⟨*Stop*, mds_2', *mem'*⟩ ∈ *loc-reach* ⟨*c*, mds_2, *mem*⟩ ⟶ $mds_2' \leq mds_1'$) ∧
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 (*clarsimp*, *intro conjI*)
apply (*metis* (*lifting*) *skip-eval-step skip-loc-reach stop-no-eval update-preserves-le*)
apply (*simp add*: *locally-sound-mode-use-def*)
by (*metis annotate.simps skip-doesnt-access*)
next
case (*assign x mds e annos*)

hence $\forall$ *mem. locally-sound-mode-use* $\langle(x \leftarrow e) \otimes annos,\ mds_2,\ mem\rangle$
unfolding *locally-sound-mode-use-def*
proof (*clarify*)
fix *mem c′ mds′ mem′ y*
assume *asm*: $\langle c', mds', mem'\rangle \in$ *loc-reach* $\langle(x \leftarrow e) \otimes annos,\ mds_2,\ mem\rangle$
hence $c' = (x \leftarrow e) \otimes annos \wedge mds' = mds_2 \vee c' = Stop \wedge mds' = mds_2 \oplus annos$
using *assign-loc-reach* **by** *blast*
thus $(y \in mds'\ GuarNoReadOrWrite \longrightarrow doesnt\text{-}read\text{-}or\text{-}modify\ c'\ y) \wedge (y \in mds'\ GuarNoWrite \longrightarrow doesnt\text{-}modify\ c'\ y)$
proof
assume $c' = (x \leftarrow e) \otimes annos \wedge mds' = mds_2$
thus *?thesis*
proof (*safe*)
assume *nin*: $y \in mds_2\ GuarNoReadOrWrite$
hence *nin*: $y \in mds\ GuarNoReadOrWrite$
using *assign.prems* **unfolding** *le-fun-def* **by** *blast*
hence $y \notin aexp\text{-}vars\ e$
by (*metis IntD2 IntI assign.hyps*(2) *assign.prems empty-iff inf-apply le-iff-inf*)
moreover from *nin assign.hyps*(3) **have** $\mathcal{C}\text{-}vars\ y \cap aexp\text{-}vars\ e = \{\}$
by (*meson contra-subsetD*)
moreover from *nin assign.hyps* **have** $x \notin \mathcal{C}\text{-}vars\ y \wedge x \neq y$
by *blast*
ultimately show *doesnt-read-or-modify* $((x \leftarrow e) \otimes annos)\ y$
using *assign-doesnt-read*
by *fastforce*
next
assume $y \in mds_2\ GuarNoWrite$
hence *nin*: $y \in mds\ GuarNoWrite$
using *assign.prems* **unfolding** *le-fun-def* **by** *blast*
hence $x \neq y \wedge x \notin \mathcal{C}\text{-}vars\ y$
using *assign* **by** *blast*
with *assign-doesnt-modify* **show** *doesnt-modify* $((x \leftarrow e) \otimes annos)\ y$
by *blast*
qed
next
assume $c' = Stop \wedge mds' = mds_2 \oplus annos$
with *stop-doesnt-access* **show** *?thesis* **by** *blast*
qed
qed
thus *?case*
apply *clarsimp*
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 ⟨Stop, mds₂', mem'⟩ ∈ loc-reach ⟨If e c₁ c₂ ⊗ annos, mds₂, mem⟩
    with modes-le' and if- show mds₂' ≤ mds''
      by (metis if-eval-false if-eval-true if-loc-reach stop-no-eval)
  next
    fix mem c' mds' mem' x
    assume ⟨c', mds', mem'⟩ ∈ loc-reach ⟨If e c₁ c₂ ⊗ annos, mds₂, mem⟩
    hence (c' = If e c₁ c₂ ⊗ annos ∧ mds' = mds₂) ∨
          (∃ mem''. ⟨c', mds', mem'⟩ ∈ loc-reach ⟨c₁, mds₂ ⊕ annos, mem''⟩) ∨
          (∃ mem''. ⟨c', mds', mem'⟩ ∈ loc-reach ⟨c₂, mds₂ ⊕ annos, mem''⟩)
      using if-loc-reach by blast
    thus (x ∈ mds' GuarNoReadOrWrite ⟶ doesnt-read-or-modify c' x) ∧
         (x ∈ mds' GuarNoWrite ⟶ doesnt-modify c' x)
    proof
      assume c' = If e c₁ c₂ ⊗ annos ∧ mds' = mds₂
      thus ?thesis
      proof (safe)
        assume x ∈ mds₂ GuarNoReadOrWrite
        hence nin: x ∈ mds GuarNoReadOrWrite
          using if- unfolding le-fun-def by auto
        with ‹bexp-vars e ∩ mds GuarNoReadOrWrite = {}› have x ∉ bexp-vars e
          by (metis IntD2 disjoint-iff-not-equal)
        moreover from if-(6) nin have 𝒞-vars x ∩ bexp-vars e = {}
          by blast
        ultimately show doesnt-read-or-modify (If e c₁ c₂ ⊗ annos) x
          using if-doesnt-read by blast
      next
        assume x ∈ mds₂ GuarNoWrite
        thus doesnt-modify (If e c₁ c₂ ⊗ annos) x
          using if-doesnt-modify by blast
      qed
    next
      assume (∃ mem''. ⟨c', mds', mem'⟩ ∈ loc-reach ⟨c₁, mds₂ ⊕ annos, mem''⟩)
∨
             (∃ mem''. ⟨c', mds', mem'⟩ ∈ loc-reach ⟨c₂, mds₂ ⊕ annos, mem''⟩)
      with if- show ?thesis
        by (metis locally-sound-mode-use-def modes-le')
    qed
  qed
next
```

case (*while mdsa mds annos c e*)
hence $mds_2 \oplus annos \leq mds \oplus annos$
by (*metis* (*lifting*) *update-preserves-le*)
have *while-loc-reach*: $\bigwedge c'\ mds'\ mem'\ mem.$
$\langle c', mds', mem' \rangle \in$ *loc-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 (*safe*)
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*) *Stmt.distinct*(*43*) *annotate.elims update-annos.simps*(*1*) *while.hyps*(*1*) *while.prems while-loc-reach*)
next
fix *mem*
from *while* **have** *a*: *bexp-vars e* ∩ $(mds_2 \oplus annos)$ *GuarNoReadOrWrite* = {}
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*)
from *while* **have** *b*: $\forall\, v.$ *C-vars v* ∩ *bexp-vars e* ≠ {} ⟶ $v \notin (mds_2 \oplus annos)$

GuarNoReadOrWrite
by (*meson* ‹$mds_2 \oplus annos \leq mds \oplus annos$› *le-fun-def subsetCE*)
show *locally-sound-mode-use* ⟨*While e c* ⊗ *annos*, mds_2, *mem*⟩
unfolding *locally-sound-mode-use-def*
apply (*rule allI*)+
apply (*rule impI*)
proof −
fix c' mds' mem'
define *lc* **where** *lc* = ⟨*While e c* ⊗ *annos*, mds_2, *mem*⟩
assume ⟨c', mds', mem'⟩ ∈ *loc-reach lc*
thus ∀ *x*. (*x* ∈ mds' *GuarNoReadOrWrite* ⟶ *doesnt-read-or-modify* c' *x*) ∧
(*x* ∈ mds' *GuarNoWrite* ⟶ *doesnt-modify* c' *x*)
apply (*simp add*: *lc-def*)
apply (*drule while-loc-reach*)
apply (*rule allI*)
apply (*erule disjE5*)
proof (*clarsimp*)
fix *x*
show (*x* ∈ mds_2 *GuarNoReadOrWrite* ⟶ *doesnt-read-or-modify* (*While e c* ⊗ *annos*) *x*) ∧
(*x* ∈ mds_2 *GuarNoWrite* ⟶ *doesnt-modify* (*While e c* ⊗ *annos*) *x*)
unfolding *doesnt-read-or-modify-def doesnt-read-or-modify-vars-def* **and** *doesnt-modify-def*
using *while-eval* **and** *while-eval-elim*
by *blast*
next
fix *x*
assume *a*: c' = *Stmt.If e* (*c* ;; *While e c*) *Stop* ∧ $mds' \leq mdsa$
hence $mds' \leq mdsa$ **by** *blast*
let *?c'* = *If e* (*c* ;; *While e c*) *Stop*
have *x* ∈ mds' *GuarNoReadOrWrite* ⟶ *doesnt-read-or-modify ?c' x*
apply *clarify*
apply (*rule if-doesnt-read'*)
apply (*metis IntI* ‹$mds' \leq mdsa$› *empty-iff le-fun-def rev-subsetD while.hyps*(*1*) *while.hyps*(*4*))
by (*metis IntI* ‹$mds' \leq mdsa$› *empty-iff le-fun-def rev-subsetD while.hyps*(*1*) *while.hyps*(*5*))
moreover
have *x* ∈ mds' *GuarNoWrite* ⟶ *doesnt-modify ?c' x*
by (*metis annotate.simps*(*1*) *if-doesnt-modify*)
ultimately show (*x* ∈ mds' *GuarNoReadOrWrite* ⟶ *doesnt-read-or-modify* c' *x*) ∧
(*x* ∈ mds' *GuarNoWrite* ⟶ *doesnt-modify* c' *x*)
using *a* **by** *simp*
next
fix *x*
assume c' = *Stop* ∧ $mds' \leq mdsa$
thus (*x* ∈ mds' *GuarNoReadOrWrite* ⟶ *doesnt-read-or-modify* c' *x*) ∧
(*x* ∈ mds' *GuarNoWrite* ⟶ *doesnt-modify* c' *x*)

```
        by (simp, metis stop-doesnt-access)
      next
        fix x
        assume ∃ c'' mem'' mds₃.
            c' = c'' ;; While e c ∧
            mds₃ ≤ mdsa ∧
            ⟨c'', mds', mem'⟩
            ∈ loc-reach ⟨c, mds₃, mem''⟩
        from this obtain
         c'' mem'' mds₃
        where mds₃ ≤ mdsa and [simp]: c' = c'' ;; While e c
        and ⟨c'', mds', mem'⟩ ∈ loc-reach ⟨c, mds₃, mem''⟩ by blast
        thus (x ∈ mds' GuarNoReadOrWrite ⟶ doesnt-read-or-modify c' x) ∧
              (x ∈ mds' GuarNoWrite ⟶ doesnt-modify c' x)
          apply (clarsimp, safe)
           apply (rule seq-doesnt-read)
           apply (insert while(3))
           apply (metis ‹mds₃ ≤ mdsa› locally-sound-mode-use-def while.hyps(1))
          apply (rule seq-doesnt-modify)
          by (metis ‹mds₃ ≤ mdsa› locally-sound-mode-use-def while.hyps(1))
      next
        fix x
        assume c' = While e c ∧ mds' ≤ mdsa
        thus (x ∈ mds' GuarNoReadOrWrite ⟶ doesnt-read-or-modify c' x) ∧
              (x ∈ mds' GuarNoWrite ⟶ doesnt-modify c' x)
          unfolding doesnt-read-or-modify-def doesnt-read-or-modify-vars-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 (safe)
    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 clarsimp
    by (metis (opaque-lifting, no-types) inf-absorb2 le-infI1)
qed

end

end
```

References

[GMS14] Sylvia Grewe, Heiko Mantel, and Daniel Schoepe. A formalisation of assumptions and guarantees for compositional noninterference. *Archive of Formal Proofs*, 2014. http://isa-afp.org/entries/SIFUM_Type_Systems.shtml.

[MSPR16] Toby Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. Compositional verification and refinement of concurrent value-dependent noninterference. In *IEEE Computer Security Foundations Symposium*, Lisbon, Portugal, June 2016.

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

[Mur15] Toby Murray. On high-assurance information-flow-secure programming languages. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 43–48, Prague, Czech Republic, July 2015.