

# A Dependent Security Type System for Concurrent Imperative Programs

Toby Murray, Robert Sison, Edward Pierzchalski and Christine Rizkallah

May 26, 2024

## 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

<b>1 Preliminaries</b>	<b>2</b>
<b>2 Definition of the SIFUM-Security Property</b>	<b>5</b>
2.1 Evaluation of Concurrent Programs . . . . .	5
2.2 Low-equivalence and Strong Low Bisimulations . . . . .	7
2.3 SIFUM-Security . . . . .	9
2.4 Sound Mode Use . . . . .	10

<b>3</b>	<b>Compositionality Proof for SIFUM-Security Property</b>	<b>14</b>
<b>4</b>	<b>Language for Instantiating the SIFUM-Security Property</b>	<b>59</b>
4.1	Syntax . . . . .	59
4.2	Semantics . . . . .	60
4.3	Semantic Properties . . . . .	62
<b>5</b>	<b>Type System for Ensuring SIFUM-Security of Commands</b>	<b>67</b>
5.1	Typing Rules . . . . .	67
5.2	Typing Soundness . . . . .	84
<b>6</b>	<b>Type System for Ensuring Locally Sound Use of Modes</b>	<b>152</b>
6.1	Typing Rules . . . . .	152
6.2	Soundness of the Type System . . . . .	153

## 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  $\vee$  d2 = High) then High else
Low)

```

```

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

```

```

definition bot-Sec-def: bot = Low

```

```

definition less-eq-Sec-def: d1  $\leq$  d2 = (d1 = d2  $\vee$  d1 = Low)

```

```

definition less-Sec-def: d1 < d2 = (d1 = Low  $\wedge$  d2 = High)

```

```

definition Sup-Sec-def: Sup S = (if (High  $\in$  S) then High else Low)

```

```

definition Inf-Sec-def: Inf S = (if (Low  $\in$  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  $\Rightarrow$  'val
```

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

```
type-synonym 'var Mds = Mode  $\Rightarrow$  'var set
```

Local configurations:

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

Global configurations:

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

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

```

locale sifum-security-init =
  fixes dma :: ('Var, 'Val) Mem  $\Rightarrow$  'Var  $\Rightarrow$  Sec
  fixes C-vars :: 'Var  $\Rightarrow$  'Var set
  fixes C :: 'Var set
  fixes eval :: ('Com, 'Var, 'Val) LocalConf rel
  fixes some-val :: 'Val
  fixes INIT :: ('Var, 'Val) Mem  $\Rightarrow$  bool
  assumes deterministic:  $\llbracket (lc, lc') \in eval; (lc, lc'') \in eval \rrbracket \Longrightarrow lc' = lc''$ 
  assumes finite-memory: finite  $\{(x::'Var). True\}$ 
  defines C  $\equiv \bigcup x. C\text{-vars } x$ 
  assumes C-vars-C:  $x \in C \Longrightarrow C\text{-vars } x = \{ \}$ 
  assumes dma-C-vars:  $\forall x \in C\text{-vars } y. mem_1 x = mem_2 x \Longrightarrow dma\ mem_1 y = dma$ 
  mem_2 y
  assumes C-Low:  $\forall x \in C. dma\ mem\ x = Low$ 

```

```

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

```

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

**lemma** *C-vars-subset-C*:

*C-vars*  $x \subseteq C$

**by**(*force simp: C-def*)

**lemma** *dma-C*:

$\forall x \in C. mem_1 x = mem_2 x \implies dma\ mem_1 = dma\ mem_2$

**proof**

**fix** *y*

**assume**  $\forall x \in C. mem_1 x = mem_2 x$

**hence**  $\forall x \in C\text{-vars } y. mem_1 x = mem_2 x$

**using** *C-vars-subset-C* **by** *blast*

**thus**  $dma\ mem_1 y = dma\ mem_2 y$

**by**(*rule dma-C-vars*)

**qed**

**end**

**lemma** *my-trancl-induct* [*consumes 1, case-names base step*]:

$\llbracket (a, b) \in r^+;$

$P\ a;$

$\bigwedge x\ y. \llbracket (x, y) \in r; P\ x \rrbracket \implies P\ y \rrbracket \implies P\ b$

**by** (*induct rule: trancl.induct, blast+*)

**lemma** *my-trancl-step-induct* [*consumes 1, case-names base step*]:

$\llbracket (a, b) \in r^+;$

$\bigwedge x\ y. (x, y) \in r \implies P\ x\ y;$

$\bigwedge x\ y\ z. P\ x\ y \implies (y, z) \in r \implies P\ x\ z \rrbracket \implies P\ a\ b$

**by** (*induct rule: trancl-induct, blast+*)

**lemma** *my-trancl-big-step-induct* [*consumes 1, case-names base step*]:

$\llbracket (a, b) \in r^+;$

$\bigwedge x\ y. (x, y) \in r \implies P\ x\ y;$

$\bigwedge x\ y\ z. (x, y) \in r^+ \implies P\ x\ y \implies (y, z) \in r \implies P\ y\ z \implies P\ x\ z \rrbracket \implies P\ a\ b$

**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*  $\rightarrow$  (*'val*  $\times$  *'val*)

**definition** *apply-adaptation* ::  
*bool*  $\Rightarrow$  (*'Var*, *'Val*) *Mem*  $\Rightarrow$  (*'Var*, *'Val*) *adaptation*  $\Rightarrow$  (*'Var*, *'Val*) *Mem*  
**where** *apply-adaptation first mem A* =  
 $(\lambda x. \text{case } (A \ x) \text{ of}$   
 $\quad \text{Some } (v_1, v_2) \Rightarrow \text{if first then } v_1 \text{ else } v_2$   
 $\quad | \text{None} \Rightarrow \text{mem } x)$

**abbreviation** *apply-adaptation<sub>1</sub>* ::  
(*'Var*, *'Val*) *Mem*  $\Rightarrow$  (*'Var*, *'Val*) *adaptation*  $\Rightarrow$  (*'Var*, *'Val*) *Mem*  
(-  $[[|_1$  -] [900, 0] 1000)  
**where** *mem*  $[[|_1$  *A*]  $\equiv$  *apply-adaptation True mem A*

**abbreviation** *apply-adaptation<sub>2</sub>* ::  
(*'Var*, *'Val*) *Mem*  $\Rightarrow$  (*'Var*, *'Val*) *adaptation*  $\Rightarrow$  (*'Var*, *'Val*) *Mem*  
(-  $[[|_2$  -] [900, 0] 1000)  
**where** *mem*  $[[|_2$  *A*]  $\equiv$  *apply-adaptation False mem A*

**definition**  
*var-asm-not-written* :: *'Var Mds*  $\Rightarrow$  *'Var*  $\Rightarrow$  *bool*  
**where**  
*var-asm-not-written mds x*  $\equiv$   $x \in mds \text{ AsmNoWrite} \vee x \in mds \text{ AsmNoReadOrWrite}$

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

### 2.1 Evaluation of Concurrent Programs

**abbreviation** *eval-abv* :: (*'Com*, *'Var*, *'Val*) *LocalConf*  $\Rightarrow$  (-, -, -) *LocalConf*  $\Rightarrow$  *bool*  
 $(\text{infixl } \rightsquigarrow 70)$   
**where**  
 $x \rightsquigarrow y \equiv (x, y) \in \text{eval}$

**abbreviation** *conf-abv* :: *'Com*  $\Rightarrow$  *'Var Mds*  $\Rightarrow$  (*'Var*, *'Val*) *Mem*  $\Rightarrow$  (-,-,-) *LocalConf*  
 $(\langle -, -, - \rangle [0, 0, 0] 1000)$   
**where**  
 $\langle c, mds, mem \rangle \equiv ((c, mds), mem)$

**inductive-set** *meval* :: ((-,-,-) *GlobalConf*  $\times$  *nat*  $\times$  (-,-,-) *GlobalConf*) *set*  
**and** *meval-abv* :: -  $\Rightarrow$  -  $\Rightarrow$  -  $\Rightarrow$  *bool* (-  $\rightsquigarrow$  - - 70)

**where**

$conf \rightsquigarrow_k conf' \equiv (conf, k, conf') \in meval \mid$   
 $meval-intro [iff]: \llbracket (cms ! n, mem) \rightsquigarrow (cm', mem'); n < length\ cms \rrbracket \implies$   
 $((cms, mem), n, (cms [n := cm'], mem')) \in meval$

**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**

$neval-0: x = y \implies x \rightsquigarrow^0 y \mid$   
 $neval-S-n: x \rightsquigarrow y \implies y \rightsquigarrow^n z \implies x \rightsquigarrow^{Suc\ n} z$

**inductive-cases**  $neval-ZeroE: neval\ x\ 0\ y$

**inductive-cases**  $neval-SucE: neval\ x\ (Suc\ n)\ y$

**lemma**  $neval-det:$

$x \rightsquigarrow^n y \implies x \rightsquigarrow^n y' \implies y = y'$   
**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**  $\bigwedge n. neval\ x\ n\ y \implies n = Suc\ 0 \implies x \rightsquigarrow y$   
**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**

$meval\text{-}sched :: nat\ list \Rightarrow ('Com, 'Var, 'Val)\ GlobalConf \Rightarrow (-, -, -)\ GlobalConf \Rightarrow bool$

**where**

$meval\text{-}sched \ []\ c\ c' = (c = c') \mid$   
 $meval\text{-}sched\ (n\#\ ns)\ c\ c' = (\exists\ c''.\ c \rightsquigarrow_n c'' \wedge meval\text{-}sched\ ns\ c''\ c')$

**abbreviation**

$meval\text{-}sched\text{-}abv :: (-, -, -)\ GlobalConf \Rightarrow nat\ list \Rightarrow (-, -, -)\ GlobalConf \Rightarrow bool\ (- \rightarrow_ - -\ 70)$

**where**

$c \rightarrow_{ns} c' \equiv meval\text{-}sched\ ns\ c\ c'$

**lemma** *meval-sched-det:*

$meval\text{-}sched\ ns\ c\ c' \Longrightarrow meval\text{-}sched\ ns\ c\ c'' \Longrightarrow c' = c''$

**apply**(*induct ns arbitrary: c*)

**apply**(*auto dest: deterministic*)

**done**

## 2.2 Low-equivalence and Strong Low Bisimulations

**definition**

$low\text{-}eq :: ('Var, 'Val)\ Mem \Rightarrow (-, -)\ Mem \Rightarrow bool\ (\mathbf{infixl} =^l\ 80)$

**where**

$mem_1 =^l mem_2 \equiv (\forall\ x.\ dma\ mem_1\ x = Low \longrightarrow mem_1\ x = mem_2\ x)$

**definition**

$low\text{-}mds\text{-}eq :: 'Var\ Mds \Rightarrow ('Var, 'Val)\ Mem \Rightarrow (-, -)\ Mem \Rightarrow bool$   
 $(- =^l - [100, 100]\ 80)$

**where**

$(mem_1 =_{mds}^l mem_2) \equiv (\forall\ x.\ dma\ mem_1\ x = Low \wedge (x \in \mathcal{C} \vee x \notin mds\ Asm\text{-}NoReadOrWrite) \longrightarrow mem_1\ x = mem_2\ x)$

**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:*

$(mem_1 =_{mds}^l mem_2) \Longrightarrow dma\ mem_1 = dma\ mem_2$

**apply**(*rule dma-C*)

**apply**(*simp add: low-mds-eq-def C-Low*)

**done**

**lemma** *low-mds-eq-sym:*

$(mem_1 =_{mds}^l mem_2) \Longrightarrow (mem_2 =_{mds}^l mem_1)$

**apply**(*frule low-mds-eq-dma*)

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

**done**

**lemma** *low-eq-sym*:

$(mem_1 =^l mem_2) \implies (mem_2 =^l mem_1)$   
**apply**(*simp add: low-eq-low-mds-eq low-mds-eq-sym*)  
**done**

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

**lemma** [*simp*]:  $(\forall m_{ds}. mem =_{m_{ds}^l} mem') \implies mem =^l mem'$   
**by** (*auto simp: low-mds-eq-def low-eq-def*)

**lemma** *High-not-in-C* [*simp*]:  
 $dma\ mem_1\ x = High \implies x \notin C$   
**apply**(*case-tac x \in C*)  
**by**(*simp add: C-Low*)

**definition**

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

**where**

*closed-glob-consistent*  $\mathcal{R} =$   
 $(\forall c_1\ m_{ds}\ mem_1\ c_2\ mem_2. (\langle c_1, m_{ds}, mem_1 \rangle, \langle c_2, m_{ds}, mem_2 \rangle) \in \mathcal{R} \longrightarrow$   
 $(\forall A. ((\forall x. case\ A\ x\ of\ Some\ (v,v') \Rightarrow (mem_1\ x \neq v \vee mem_2\ x \neq v')) \longrightarrow \neg$   
*var-asm-not-written*  $m_{ds}\ x \mid - \Rightarrow True) \wedge$   
 $(\forall x. dma\ (mem_1\ [\![\!_1\ A])\ ]\ x \neq dma\ mem_1\ x \longrightarrow \neg\ var-asm-not-written\ m_{ds}$   
 $x) \wedge$   
 $(\forall x. dma\ (mem_1\ [\![\!_1\ A])\ ]\ x = Low \wedge (x \notin m_{ds}\ AsmNoReadOrWrite \vee x \in$   
 $C) \longrightarrow (mem_1\ [\![\!_1\ A])\ ]\ x = (mem_2\ [\![\!_2\ A])\ ]\ x)) \longrightarrow$   
 $(\langle c_1, m_{ds}, mem_1[\![\!_1\ A] \rangle], \langle c_2, m_{ds}, mem_2[\![\!_2\ A] \rangle] \rangle) \in \mathcal{R}))$

**definition**

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

**where**

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

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

**and** *mm-equiv-abv* ::  $((Com, Var, Val)\ LocalConf) \Rightarrow$   
 $((Com, Var, Val)\ LocalConf) \Rightarrow bool$  (**infix**  $\approx 60$ )

**where**

*mm-equiv-abv*  $x\ y \equiv (x, y) \in mm-equiv \mid$



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

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

**definition** *low-indistinguishable* ::  $'Var\ Mds \Rightarrow 'Com \Rightarrow 'Com \Rightarrow \text{bool}$   
 $(- \sim_1 - [100, 100] 80)$

**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* ::  $'Com \times 'Var\ Mds \Rightarrow \text{bool}$

**where**

*com-sifum-secure cmd*  $\equiv \text{case cmd of } (c, mds_s) \Rightarrow c \sim_{mds_s} c$

**definition**

*prog-sifum-secure-cont* ::  $('Com \times 'Var\ Mds)\ \text{list} \Rightarrow \text{bool}$

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

$(\forall mem_1 mem_2. \text{INIT } mem_1 \wedge \text{INIT } mem_2 \wedge mem_1 =^l mem_2 \longrightarrow$   
 $(\forall \text{ sched } cms_1' mem_1'.$   
 $(cmds, mem_1) \rightarrow_{\text{sched}} (cms_1', mem_1') \longrightarrow$   
 $(\exists cms_2' mem_2'. (cmds, mem_2) \rightarrow_{\text{sched}} (cms_2', mem_2') \wedge$   
 $\text{map snd } cms_1' = \text{map snd } cms_2' \wedge$   
 $\text{length } cms_2' = \text{length } cms_1' \wedge$   
 $(\forall x. \text{dma } mem_1' x = \text{Low} \wedge (x \in \mathcal{C} \vee (\forall i < \text{length } cms_1'.$   
 $x \notin \text{snd } (cms_1' ! i) \text{ AsmNoReadOrWrite})) \longrightarrow mem_1' x =$   
 $mem_2' x))))$

**lemma** *prog-sifum-secure-cont-def2*:

*prog-sifum-secure-cont cmds*  $\equiv$

$(\forall mem_1 mem_2. \text{INIT } mem_1 \wedge \text{INIT } mem_2 \wedge mem_1 =^l mem_2 \longrightarrow$   
 $(\forall \text{ sched } cms_1' mem_1'.$   
 $(cmds, mem_1) \rightarrow_{\text{sched}} (cms_1', mem_1') \longrightarrow$   
 $(\exists cms_2' mem_2'. (cmds, mem_2) \rightarrow_{\text{sched}} (cms_2', mem_2') \wedge$   
 $(\forall cms_2' mem_2'. (cmds, mem_2) \rightarrow_{\text{sched}} (cms_2', mem_2') \longrightarrow$   
 $\text{map snd } cms_1' = \text{map snd } cms_2' \wedge$   
 $\text{length } cms_2' = \text{length } cms_1' \wedge$   
 $(\forall x. \text{dma } mem_1' x = \text{Low} \wedge (x \in \mathcal{C} \vee (\forall i < \text{length } cms_1'.$   
 $x \notin \text{snd } (cms_1' ! i) \text{ AsmNoReadOrWrite})) \longrightarrow mem_1' x =$   
 $mem_2' x))))$

**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

$subst :: ('a \multimap 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$

### where

$subst\ f\ mem = (\lambda\ x.\ case\ f\ x\ of$   
     $None \Rightarrow mem\ x\ |$   
     $Some\ v \Rightarrow v)$

### abbreviation

$subst-abv :: ('a \Rightarrow 'b) \Rightarrow ('a \multimap 'b) \Rightarrow ('a \Rightarrow 'b)\ (-\ [\mapsto]\ [900, 0]\ 1000)$

### where

$f\ [\mapsto]\ \sigma \equiv subst\ \sigma\ f$

**lemma** *subst-not-in-dom* :  $\llbracket x \notin dom\ \sigma \rrbracket \Longrightarrow mem\ [\mapsto]\ \sigma\ x = mem\ x$

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

$vars-C :: 'Var\ set \Rightarrow 'Var\ set$

### where

$vars-C\ X \equiv \bigcup_{x \in X}. C\text{-vars}\ x$

**lemma** *vars-C-subset-C*:

$vars-C\ X \subseteq C$

by(*auto simp: C-def vars-C-def*)

### definition

$doesnt-read-or-modify :: 'Com \Rightarrow 'Var \Rightarrow bool$

### where

$doesnt-read-or-modify\ c\ x \equiv doesnt-read-or-modify-vars\ c\ (\{x\} \cup C\text{-vars}\ x)$

### 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-read-or-modify*  $c\ x \implies$  *doesnt-modify*  $c\ x$

**by** (*fastforce simp: doesnt-modify-def doesnt-read-or-modify-def doesnt-read-or-modify-vars-def*)

*intro: noread-nowrite dma-C-vars*)

**inductive-set**

*loc-reach* ::  $('Com, 'Var, 'Val)\ LocalConf \Rightarrow ('Com, 'Var, 'Val)\ LocalConf\ set$

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

**where**

*refl* :  $\langle fst\ (fst\ lc), snd\ (fst\ lc), snd\ lc \rangle \in loc\text{-}reach\ lc \mid$

*step* :  $\llbracket \langle c', mds', mem' \rangle \in loc\text{-}reach\ lc;$

$\langle c', mds', mem' \rangle \rightsquigarrow \langle c'', mds'', mem'' \rangle \rrbracket \implies$

$\langle c'', mds'', mem'' \rangle \in loc\text{-}reach\ lc \mid$

*mem-diff* :  $\llbracket \langle c', mds', mem' \rangle \in loc\text{-}reach\ lc;$

$(\forall x. var\text{-}asm\text{-}not\text{-}written\ mds'\ x \longrightarrow mem'\ x = mem''\ x \wedge dma$

$mem'\ x = dma\ mem''\ x) \rrbracket \implies$

$\langle c', mds', mem'' \rangle \in loc\text{-}reach\ lc$

**lemma** *neval-loc-reach*:

*neval*  $lc'\ n\ lc'' \implies lc' \in loc\text{-}reach\ lc \implies lc'' \in loc\text{-}reach\ lc$

**proof** (*induct rule: neval.induct*)

**case** (*neval-0*  $x\ y$ )

**thus** *?case* **by** *simp*

**next**

**case** (*neval-S-n*  $x\ y\ n\ z$ )

**from**  $\langle x \in loc\text{-}reach\ lc \rangle$  **and**  $\langle x \rightsquigarrow y \rangle$  **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  $\Rightarrow$  bool

**where**

*locally-sound-mode-use* lc =

$(\forall c' mds' mem'. \langle c', mds', mem' \rangle \in \text{loc-reach } lc \longrightarrow$   
 $(\forall x. (x \in mds' \text{ GuarNoReadOrWrite} \longrightarrow \text{doesnt-read-or-modify } c' x) \wedge$   
 $(x \in mds' \text{ GuarNoWrite} \longrightarrow \text{doesnt-modify } c' x)))$

**definition**

*respects-own-guarantees* :: ('Com  $\times$  'Var Mds)  $\Rightarrow$  bool

**where**

*respects-own-guarantees* cm  $\equiv$

$(\forall x. (x \in (\text{snd } cm) \text{ GuarNoReadOrWrite} \longrightarrow \text{doesnt-read-or-modify } (\text{fst } cm) x)$   
 $\wedge$   
 $(x \in (\text{snd } cm) \text{ GuarNoWrite} \longrightarrow \text{doesnt-modify } (\text{fst } cm) x))$

**lemma** *locally-sound-mode-use-def2*:

*locally-sound-mode-use* lc  $\equiv \forall lc' \in \text{loc-reach } lc. \text{respects-own-guarantees } (\text{fst } lc')$

**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)  $\implies \text{respects-own-guarantees } cm$

**unfolding** *locally-sound-mode-use-def* *respects-own-guarantees-def*

**by** (metis *fst-conv* *loc-reach.refl*)

**definition**

*compatible-modes* :: ('Var Mds) list  $\Rightarrow$  bool

**where**

*compatible-modes* mdss =  $(\forall (i :: \text{nat}) x. i < \text{length } mdss \longrightarrow$   
 $(x \in (mdss ! i) \text{ AsmNoReadOrWrite} \longrightarrow$   
 $(\forall j < \text{length } mdss. j \neq i \longrightarrow x \in (mdss ! j) \text{ GuarNoReadOrWrite})) \wedge$   
 $(x \in (mdss ! i) \text{ AsmNoWrite} \longrightarrow$   
 $(\forall j < \text{length } mdss. j \neq i \longrightarrow x \in (mdss ! j) \text{ GuarNoWrite})))$

**definition**

*reachable-mode-states* :: ('Com, 'Var, 'Val) GlobalConf  $\Rightarrow ((\text{'Var Mds}) \text{ list}) \text{ set}$

**where**

*reachable-mode-states* gc  $\equiv$

$\{mdss. (\exists cms' mem' \text{ sched}. gc \rightarrow_{\text{sched}} (cms', mem') \wedge \text{map } \text{snd } cms' =$   
 $mdss)\}$

**definition**

*globally-sound-mode-use* :: ('Com, 'Var, 'Val) GlobalConf  $\Rightarrow$  bool  
**where**  
*globally-sound-mode-use* gc  $\equiv$   
 ( $\forall$  mdss. mdss  $\in$  reachable-mode-states gc  $\longrightarrow$  compatible-modes mdss)

**primrec**  
*sound-mode-use* :: (-, -, -) GlobalConf  $\Rightarrow$  bool  
**where**  
*sound-mode-use* (cms, mem) =  
 (list-all ( $\lambda$  cm. locally-sound-mode-use (cm, mem))) cms  $\wedge$   
 globally-sound-mode-use (cms, mem)

**lemma** mm-equiv-sym:  
 assumes equivalent:  $\langle c_1, mds_1, mem_1 \rangle \approx \langle c_2, mds_2, mem_2 \rangle$   
 shows  $\langle c_2, mds_2, mem_2 \rangle \approx \langle c_1, mds_1, mem_1 \rangle$   
**proof** –  
 from equivalent obtain  $\mathcal{R}$   
 where  $\mathcal{R}$ -bisim: strong-low-bisim-mm  $\mathcal{R} \wedge (\langle c_1, mds_1, mem_1 \rangle, \langle c_2, mds_2, mem_2 \rangle) \in \mathcal{R}$   
 by (metis mm-equiv.simps)  
 hence sym  $\mathcal{R}$   
 by (auto simp: strong-low-bisim-mm-def)  
 hence  $(\langle c_2, mds_2, mem_2 \rangle, \langle c_1, mds_1, mem_1 \rangle) \in \mathcal{R}$   
 by (metis  $\mathcal{R}$ -bisim symE)  
 thus ?thesis  
 by (metis  $\mathcal{R}$ -bisim mm-equiv.intros)

qed

**lemma** low-indistinguishable-sym:  $lc \sim_{m_{ds}} lc' \implies lc' \sim_{m_{ds}} lc$   
 apply (clarsimp simp: low-indistinguishable-def)  
 apply (rule mm-equiv-sym)  
 apply (blast dest: low-mds-eg-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<sub>1</sub>  $c_2$  mem<sub>2</sub>  $x$   
 assume  $\langle c_1, mds, mem_1 \rangle \approx \langle c_2, mds, mem_2 \rangle$   
 then obtain  $\mathcal{R}$  where

```

    strong-low-bisim-mm  $\mathcal{R} \wedge (\langle c_1, mds, mem_1 \rangle, \langle c_2, mds, mem_2 \rangle) \in \mathcal{R}$ 
  by blast
  thus  $mem_1 =_{mds^l} mem_2$  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**

$differing-vars :: ('Var, 'Val) Mem \Rightarrow (-, -) Mem \Rightarrow 'Var set$

**where**

$differing-vars mem_1 mem_2 \equiv \{x. mem_1 x \neq mem_2 x\}$

**definition**

$differing-vars-lists :: ('Var, 'Val) Mem \Rightarrow (-, -) Mem \Rightarrow$

$((-, -) Mem \times (-, -) Mem) list \Rightarrow nat \Rightarrow 'Var set$

**where**

$differing-vars-lists mem_1 mem_2 mems i \equiv$

$(differing-vars mem_1 (fst (mems ! i)) \cup differing-vars mem_2 (snd (mems ! i)))$

**lemma** *differing-finite*: *finite* (*differing-vars* *mem*<sub>1</sub> *mem*<sub>2</sub>)  
**by** (*metis UNIV-def Un-UNIV-left finite-Un finite-memory*)

**lemma** *differing-lists-finite*: *finite* (*differing-vars-lists* *mem*<sub>1</sub> *mem*<sub>2</sub> *mems* *i*)  
**by** (*simp add: differing-finite differing-vars-lists-def*)

**fun** *makes-compatible* ::

(*'Com, 'Var, 'Val*) *GlobalConf*  $\Rightarrow$   
(*'Com, 'Var, 'Val*) *GlobalConf*  $\Rightarrow$   
((-, -) *Mem*  $\times$  (-, -) *Mem*) *list*  $\Rightarrow$   
*bool*

**where**

*makes-compatible* (*cms*<sub>1</sub>, *mem*<sub>1</sub>) (*cms*<sub>2</sub>, *mem*<sub>2</sub>) *mems* =  
(*length cms*<sub>1</sub> = *length cms*<sub>2</sub>  $\wedge$  *length cms*<sub>1</sub> = *length mems*  $\wedge$   
 $(\forall i. i < \text{length } cms_1 \longrightarrow$   
 $(\forall \sigma. \text{dom } \sigma = \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i \longrightarrow$   
 $(cms_1 ! i, (\text{fst } (mems ! i)) [\mapsto \sigma]) \approx (cms_2 ! i, (\text{snd } (mems ! i)) [\mapsto \sigma])) \wedge$   
 $(\forall x. (mem_1 \ x = mem_2 \ x \vee dma \ mem_1 \ x = High \vee x \in \mathcal{C}) \longrightarrow$   
 $x \notin \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i)) \wedge$   
 $((\text{length } cms_1 = 0 \wedge mem_1 =^l mem_2) \vee (\forall x. \exists i. i < \text{length } cms_1 \wedge$   
 $x \notin \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i)))$

**lemma** *makes-compatible-intro* [*intro*]:

$\llbracket \text{length } cms_1 = \text{length } cms_2 \wedge \text{length } cms_1 = \text{length } mems;$   
 $(\wedge i \ \sigma. \llbracket i < \text{length } cms_1; \text{dom } \sigma = \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i \rrbracket$   
 $\implies$   
 $(cms_1 ! i, (\text{fst } (mems ! i)) [\mapsto \sigma]) \approx (cms_2 ! i, (\text{snd } (mems ! i)) [\mapsto \sigma]);$   
 $(\wedge i \ x. \llbracket i < \text{length } cms_1; mem_1 \ x = mem_2 \ x \vee dma \ mem_1 \ x = High \vee x \in$   
 $\mathcal{C} \rrbracket \implies$   
 $x \notin \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i);$   
 $(\text{length } cms_1 = 0 \wedge mem_1 =^l mem_2) \vee$   
 $(\forall x. \exists i. i < \text{length } cms_1 \wedge x \notin \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i) \rrbracket$   
 $\implies$   
*makes-compatible* (*cms*<sub>1</sub>, *mem*<sub>1</sub>) (*cms*<sub>2</sub>, *mem*<sub>2</sub>) *mems*  
**by** *auto*

**lemma** *compat-low*:

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

**proof** –

**assume**  $i < \text{length } cms_1$  **and**  $*$ :  $x \in \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i$  **and**  
*makes-compatible* (*cms*<sub>1</sub>, *mem*<sub>1</sub>) (*cms*<sub>2</sub>, *mem*<sub>2</sub>) *mems*

**then have**

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

**by** (*simp add: Let-def, blast*)  
**with** \* **show**  $\text{dma mem}_1 x = \text{Low}$   
**by** (*cases dma mem}\_1 x, blast*)  
**qed**

**lemma** *compat-different*:

$\llbracket \text{makes-compatible } (cms_1, mem_1) (cms_2, mem_2) mems;$   
 $i < \text{length } cms_1;$   
 $x \in \text{differing-vars-lists } mem_1 mem_2 mems i \rrbracket \implies mem_1 x \neq mem_2 x \wedge \text{dma}$   
 $mem_1 x = \text{Low} \wedge x \notin \mathcal{C}$   
**by** (*cases dma mem}\_1 x, auto*)

**lemma** *sound-modes-no-read* :

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

**proof** –

**fix**  $cms mem x i$   
**assume** *sound-modes: sound-mode-use* ( $cms, mem$ ) **and**  $i < \text{length } cms$   
**hence** *locally-sound-mode-use* ( $cms ! i, mem$ )  
**by** (*auto simp: sound-mode-use-def list-all-length*)  
**moreover**  
**assume**  $x \in (\text{map snd } cms ! i) \text{ GuarNoReadOrWrite}$   
**ultimately show** *doesnt-read-or-modify* ( $\text{fst } (cms ! i)$ )  $x$   
**apply** (*simp add: locally-sound-mode-use-def*)  
**using**  $\langle i < \text{length } cms \rangle \langle \text{locally-sound-mode-use } (cms ! i, mem) \rangle \text{ locally-sound-respects-guarantees}$   
*respects-own-guarantees-def* **by** *auto*  
**qed**

**lemma** *differing-vars-neg*:  $x \notin \text{differing-vars-lists } mem1 mem2 mems i \implies$

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

**lemma** *differing-vars-neg-intro*:

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

**lemma** *differing-vars-elim* [*elim*]:

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

**lemma** *makes-compatible-dma-eq*:

**assumes** *compat: makes-compatible* ( $cms_1, mem_1$ ) ( $cms_2, mem_2$ )  $mems$   
**assumes** *ile*:  $i < \text{length } cms_1$   
**assumes** *dom* $\sigma$ :  $\text{dom } \sigma = \text{differing-vars-lists } mem_1 mem_2 mems i$   
**shows**  $\text{dma } ((\text{fst } (mems ! i)) [\mapsto \sigma]) = \text{dma } mem_1$   
**proof**(*rule dma-C, clarify*)



**fix**  $x$   
**assume**  $x \in \mathcal{C}$   
**with** *compat ile* **have** *notin-diff*:  $x \notin \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i$   
**by** *simp*  
**hence**  $x \notin \text{dom } \sigma$   
**by**(*metis dom $\sigma$* )  
**hence** (*fst (mems ! i) [mapsto  $\sigma$ ]*)  $x = (\text{fst } (mems ! i)) \ x$   
**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*:  
 $\llbracket \text{fst } (mems ! i) \ x = \text{snd } (mems ! i) \ x;$   
 $x \notin \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i \rrbracket \implies$   
 $mem_1 \ x = mem_2 \ x$

**proof** –  
**assume**  $x \notin \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i$   
**hence**  $\text{fst } (mems ! i) \ x = mem_1 \ x \wedge \text{snd } (mems ! i) \ x = mem_2 \ x$   
**by** (*simp add: differing-vars-lists-def differing-vars-def*)  
**moreover assume**  $\text{fst } (mems ! i) \ x = \text{snd } (mems ! i) \ x$   
**ultimately show**  $mem_1 \ x = mem_2 \ x$  **by** *auto*  
**qed**

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

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

**lemma** *globally-sound-modes-compatible*:  
 $\llbracket \text{globally-sound-mode-use } (cms, mem) \rrbracket \implies \text{compatible-modes } (\text{map } \text{snd } cms)$   
**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<sub>1</sub>, mem<sub>1</sub>)*  
*sound-mode-use (cms<sub>2</sub>, mem<sub>2</sub>)*

**assumes** *compat*: *makes-compatible* ( $cms_1, mem_1$ ) ( $cms_2, mem_2$ ) *mems*  
**assumes** *modes-eq*:  $map\ snd\ cms_1 = map\ snd\ cms_2$   
**assumes** *ile*:  $i < length\ cms_1$   
**assumes** *x*:  $x \in differing-vars-lists\ mem_1\ mem_2\ mems\ i$   
**shows** *doesnt-read-or-modify* ( $fst\ (cms_1\ !\ i)$ )  $x \wedge doesnt-read-or-modify\ (fst\ (cms_2\ !\ i))\ x$   
**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*:  $mem_1\ x \neq mem_2\ x$   
**by** (*metis compat-different*)  
  
**from** *compat ile x* **have** *not-in-C*:  $x \notin C$   
**by** (*metis compat-different*)  
  
**with** *a* **and** *compat ile x* **obtain** *j* **where**  
*jprop*:  $j < length\ cms_1 \wedge x \notin differing-vars-lists\ mem_1\ mem_2\ mems\ j$   
**by** *fastforce*  
  
**let**  $?X_j = differing-vars-lists\ mem_1\ mem_2\ mems\ j$   
**obtain**  $\sigma :: 'Val \rightarrow 'Val$  **where** *dom* $\sigma$ :  $dom\ \sigma = ?X_j$   
**proof**  
**let**  $?s = \lambda x. if\ (x \in ?X_j)\ then\ Some\ some-val\ else\ None$   
**show** *dom*  $?s = ?X_j$  **unfolding** *dom-def* **by** *auto*  
**qed**  
**let**  $?mdss = map\ snd\ cms_1$  **and**  
 $?mems_{1j} = fst\ (mems\ !\ j)$  **and**  
 $?mems_{2j} = snd\ (mems\ !\ j)$   
  
**from** *jprop dom* $\sigma$  **have** *subst-eq*:  
 $?mems_{1j}\ [\mapsto\ \sigma]\ x = ?mems_{1j}\ x \wedge ?mems_{2j}\ [\mapsto\ \sigma]\ x = ?mems_{2j}\ x$   
**by** (*metis subst-not-in-dom*)  
  
**from** *compat jprop dom* $\sigma$   
**have** ( $cms_1\ !\ j, ?mems_{1j}\ [\mapsto\ \sigma]$ )  $\approx$  ( $cms_2\ !\ j, ?mems_{2j}\ [\mapsto\ \sigma]$ )  
**by** (*auto simp: Let-def*)  
  
**hence** *low-eq*:  $?mems_{1j}\ [\mapsto\ \sigma] = ?mdss\ !\ j\ ^l\ ?mems_{2j}\ [\mapsto\ \sigma]$  **using** *modes-eq*  
**by** (*metis (no-types) jprop len mm-equiv-low-eq nth-map surjective-pairing*)  
  
**with** *jprop* **and** *b* **have**  $x \in (?mdss\ !\ j)\ AsmNoReadOrWrite$   
**proof** –  
**{** **assume**  $x \notin (?mdss\ !\ j)\ AsmNoReadOrWrite$

**then have** *mems-eq*:  $?mems_1 j x = ?mems_2 j x$   
**using**  $\langle dma\ mem_1\ x = Low \rangle\ low\text{-}eq\ subst\text{-}eq$   
*makes-compatible-dma-eq*[*OF compat jprop*[*THEN conjunct1*] *dom* $\sigma$ ]  
*low-mds-eq-def*  
**by** (*metis (poly-guards-query)*)

**hence**  $mem_1\ x = mem_2\ x$   
**by** (*metis compat-different-vars jprop*)

**hence** *False* **by** (*metis b*)

}

**thus** *?thesis* **by** *metis*

**qed**

**hence**  $x \in (?mdss\ !\ i)\ GuarNoReadOrWrite$   
**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 (cms<sub>1</sub> ! i)) x*  $\wedge$  *doesnt-read-or-modify (fst (cms<sub>2</sub> ! i)) x* **using** *sound-modes ile*

**by** (*metis len modes-eq sound-modes-no-read*)

**qed**

**definition**

*vars-and-C* ::  $'Var\ set \Rightarrow 'Var\ set$

**where**

*vars-and-C*  $X \equiv X \cup vars\text{-}C\ X$

**fun** *change-respecting* ::

$('Com, 'Var, 'Val)\ LocalConf \Rightarrow$

$('Com, 'Var, 'Val)\ LocalConf \Rightarrow$

$'Var\ set \Rightarrow bool$

**where** *change-respecting* (*cms*, *mem*) (*cms'*, *mem'*)  $X =$

$((cms, mem) \rightsquigarrow (cms', mem') \wedge$

$(\forall \sigma. dom\ \sigma = vars\text{-}and\text{-}C\ X \longrightarrow (cms, mem\ [\mapsto \sigma]) \rightsquigarrow (cms', mem'\ [\mapsto$

$\sigma])))$

**lemma** *subst-overrides*:  $dom\ \sigma = dom\ \tau \implies mem\ [\mapsto \tau]\ [\mapsto \sigma] = mem\ [\mapsto \sigma]$

**unfolding** *subst-def*

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

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

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

**lemma** *dom-restrict-total*:  $dom\ (to\text{-}partial\ f\ \mid\ X) = X$

**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*:  $\forall f. \text{dom } f = Y \longrightarrow (cms, mem \ [\mapsto f]) \rightsquigarrow (cms', mem' \ [\mapsto f])$   
**assumes** *x-in-dom*:  $x \in Y$   
**shows**  $mem \ x = mem' \ x$   
**proof** –  
**let**  $?f' = \text{to-partial } mem \ |' \ Y$   
**have**  $\text{dom } ?f' = Y$   
**by** (*metis dom-restrict-total*)

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

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

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

**moreover**  
**have**  $x\text{-in-dom}' : x \in \text{dom } ?f'$   
**by** (*metis x-in-dom dom-restrict-total*)  
**hence**  $?f' \ x = \text{Some } (mem \ x)$   
**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*:  $(\forall \sigma. \text{dom } \sigma = X \longrightarrow (cms, mem \ [\mapsto \sigma]) \rightsquigarrow (cms', mem' \ [\mapsto \sigma]))$   
**assumes** *dom-subset*:  $\text{dom } \sigma \subseteq X$   
**shows**  $(cms, mem \ [\mapsto \sigma]) \rightsquigarrow (cms', mem' \ [\mapsto \sigma])$   
**proof** –  
**define**  $\sigma_X$  **where**  $\sigma_X \ x = (\text{if } x \in X \text{ then if } x \in \text{dom } \sigma \text{ then } \sigma \ x \text{ else Some } (mem \ x) \text{ else None})$  **for**  $x$

```

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

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

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

moreover from noread  $\langle \text{dom } \sigma_X = X \rangle$  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')  $X \implies$ 
  ( $\forall \sigma. \text{dom } \sigma = X \longrightarrow (\text{cms}, \text{mem} [\mapsto \sigma]) \rightsquigarrow (\text{cms}', \text{mem}' [\mapsto \sigma])$ )
  unfolding change-respecting.simps vars-and-C-def
  using change-respecting-subset' by blast

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

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

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

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

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

```

```

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

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

lemma C-vars-finite[simp]:
  finite (C-vars x)
  apply(rule finite-subset[OF - finite-memory])
  by blast

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

```

```

    apply(simp add: subst-def  $\sigma a$ )
    using  $\langle a \notin X \rangle$  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  $\langle a \in \{x\} \cup \mathcal{C}\text{-vars } x \rangle IH_X$ 
    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  $\sigma = A \cup B \implies$ 
mem  $[\mapsto (\sigma \mid' A)] [\mapsto (\sigma \mid' B)] = mem [\mapsto \sigma]$ 
by(fastforce simp: subst-def split: option.splits intro!: ext simp: restrict-map-def)

```

**lemma** *noread-exists-change-respecting*:

```

assumes fin: finite  $(X :: 'Var \text{ set})$ 
assumes eval:  $\langle c, mds, mem \rangle \rightsquigarrow \langle c', mds', mem' \rangle$ 
assumes noread:  $\forall x \in X. \text{ doesnt-read-or-modify } c \ x$ 
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  $[\mapsto \text{Map.empty}] = mem \ mem' [\mapsto \text{Map.empty}] = mem'$ 
    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-C-def vars-and-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  $\langle c, mds, mem \rangle \langle c', mds', mem' \rangle (\text{insert } x \ X)$ 
  proof
    show  $\langle c, mds, mem \rangle \rightsquigarrow \langle c', mds', mem' \rangle$  using insert by auto
  next
  fix  $\sigma :: 'Var \rightarrow 'Val$ 
  let  $\ ?\sigma_X = \sigma \mid' \text{ vars-and-C } X$ 
  let  $\ ?\sigma_x = \sigma \mid' (\{x\} \cup \mathcal{C}\text{-vars } x)$ 
  assume dom $\sigma$ : dom  $\sigma = \text{ vars-and-C } (\text{insert } x \ X)$ 

```

**hence**  $\text{dom } ?\sigma_X = \text{vars-and-}\mathcal{C} X$   
**by** (*metis dom-restrict inf-absorb2 subset-insertI vars-and-}\mathcal{C}\text{-mono}*)  
**from**  $\text{dom } \sigma$  **have**  $\text{dom } \sigma_x: \text{dom } ?\sigma_x = \{x\} \cup \mathcal{C}\text{-vars } x$   
**by** (*simp add: dom } \sigma \text{ vars-and-}\mathcal{C}\text{-def vars-}\mathcal{C}\text{-def, blast}*)  
**have**  $\text{dom } \sigma = \text{vars-and-}\mathcal{C} X \cup (\{x\} \cup \mathcal{C}\text{-vars } x)$   
**by** (*simp add: dom } \sigma \text{ vars-and-}\mathcal{C}\text{-def vars-}\mathcal{C}\text{-def, blast}*)  
**hence**  $\text{subst } \sigma: \bigwedge \text{ mem. mem } [\mapsto ?\sigma_X] [\mapsto ?\sigma_x] = \text{mem } [\mapsto \sigma]$   
**by** (*rule subst-restrict-twice*)  
**from** *insert* **have** *doesn't-read-or-modify*  $c x$  **by** *auto*  
**moreover from** *IH* **have**  $\text{eval}_X: \langle c, \text{mds}, \text{mem } [\mapsto ?\sigma_X] \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}'$   
 $[\mapsto ?\sigma_X] \rangle$   
**using**  $\langle \text{dom } ?\sigma_X = \text{vars-and-}\mathcal{C} X \rangle$   
**unfolding** *change-respecting.simps*  
**by** *auto*  
**ultimately have**  $\langle c, \text{mds}, \text{mem } [\mapsto ?\sigma_X] [\mapsto ?\sigma_x] \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}'$   
 $[\mapsto ?\sigma_X] [\mapsto ?\sigma_x] \rangle$   
**using** *subset-refl dom } \sigma\_x \text{ doesn't-read-or-modify-subst by metis}*  
**thus**  $\langle c, \text{mds}, \text{mem } [\mapsto \sigma] \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' [\mapsto \sigma] \rangle$   
**using** *subst } \sigma* **by** *metis*  
**qed**  
**qed**  
**qed**

**lemma** *update-nth-eq*:

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

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

**lemma** *mm-equiv-step*:

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

**lemma** *change-respecting-doesnt-modify*:

**assumes** *cr*: *change-respecting*  $(\text{cms}, \text{mem}) (\text{cms}', \text{mem}') X$   
**assumes** *eval*:  $(\text{cms}, \text{mem}) \rightsquigarrow (\text{cms}', \text{mem}')$   
**assumes** *x-in-dom*:  $x \in X \cup \text{vars-}\mathcal{C} X$   
**shows**  $\text{mem } x = \text{mem}' x$   
**using** *change-respecting-doesnt-modify'* [**where**  $Y = X \cup \text{vars-}\mathcal{C} X$ , *OF eval*] *cr*



*change-respecting.simps vars-and-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**  $\bigwedge y. y \in \mathcal{C}\text{-vars } x \implies \text{mem } y = \text{mem}' y$   
**proof** –  
**fix** *y*  
**assume**  $y \in \mathcal{C}\text{-vars } x$   
**hence**  $y \in \text{vars-}\mathcal{C} X$   
**using** *x-in-dom* **by** (*auto simp: vars-}\mathcal{C}\text{-def}*)  
**thus**  $\text{mem } y = \text{mem}' y$   
**using** *cr eval change-respecting-doesnt-modify* **by** *blast*  
**qed**  
**thus** *?thesis* **by** (*metis dma-}\mathcal{C}\text{-vars}*)  
**qed**

**definition** *restrict-total* :: (*'a*  $\Rightarrow$  *'b*)  $\Rightarrow$  *'a set*  $\Rightarrow$  *'a*  $\rightarrow$  *'b*  
**where** *restrict-total f A = to-partial f |' A*

**lemma** *differing-empty-eq*:  
 $\llbracket \text{differing-vars mem mem}' = \{\} \rrbracket \implies \text{mem} = \text{mem}'$   
**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*  $\Rightarrow$  *'Var Mds*  $\Rightarrow$  (*'Var*,*'Val*) *Mem*  
 $\Rightarrow$  (*'Var*,*'Val*) *Mem*  $\Rightarrow$  *bool*  
**where** *globally-consistent A mds mem<sub>1</sub> mem<sub>2</sub>*  $\equiv$   
 $(\forall x. \text{case } A \text{ of } \text{Some } (v, v') \Rightarrow (\text{mem}_1 x \neq v \vee \text{mem}_2 x \neq v') \longrightarrow \neg \text{var-asm-not-written } mds \text{ } x \mid - \Rightarrow \text{True}) \wedge$   
 $(\forall x. \text{dma mem}_1 \llbracket \llbracket_1 A \rrbracket \rrbracket x \neq \text{dma mem}_1 x \longrightarrow \neg \text{var-asm-not-written } mds \text{ } x)$   
 $\wedge$   
 $(\forall x. \text{dma } (\text{mem}_1 \llbracket \llbracket_1 A \rrbracket \rrbracket) x = \text{Low} \wedge (x \notin mds \text{ } \text{AsmNoReadOrWrite} \vee x \in \mathcal{C}) \longrightarrow (\text{mem}_1 \llbracket \llbracket_1 A \rrbracket \rrbracket) x = (\text{mem}_2 \llbracket \llbracket_2 A \rrbracket \rrbracket) x)$

**lemma** *globally-consistent-adapt-bisim*:  
**assumes** *bisim*:  $\langle c_1, mds, mem_1 \rangle \approx \langle c_2, mds, mem_2 \rangle$   
**assumes** *globally-consistent*: *globally-consistent A mds mem<sub>1</sub> mem<sub>2</sub>*

**shows**  $\langle c_1, mds, mem_1 \llbracket_1 A \rrbracket \rangle \approx \langle c_2, mds, mem_2 \llbracket_2 A \rrbracket \rangle$   
**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-C-eq:

$(a, b) \approx (a', b') \implies snd\ a = snd\ a' \implies$   
 $\forall x \in \mathcal{C}. b\ x = b'\ x$   
**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 C-Low)

**lemma** apply-adaptation-not-in-dom:

$x \notin dom\ A \implies apply\ adaptation\ b\ blah\ A\ x = blah\ x$   
**apply**(simp add: apply-adaptation-def domIff split: option.splits)  
**done**

**lemma** makes-compatible-invariant:

**assumes** sound-modes: sound-mode-use (cms<sub>1</sub>, mem<sub>1</sub>)  
sound-mode-use (cms<sub>2</sub>, mem<sub>2</sub>)  
**assumes** compat: makes-compatible (cms<sub>1</sub>, mem<sub>1</sub>) (cms<sub>2</sub>, mem<sub>2</sub>) mems  
**assumes** modes-eq: map snd cms<sub>1</sub> = map snd cms<sub>2</sub>  
**assumes** eval: (cms<sub>1</sub>, mem<sub>1</sub>)  $\rightsquigarrow_k$  (cms<sub>1</sub>' , mem<sub>1</sub>' )  
**obtains** cms<sub>2</sub>' mem<sub>2</sub>' mems' **where**  
map snd cms<sub>1</sub>' = map snd cms<sub>2</sub>'  $\wedge$   
(cms<sub>2</sub>, mem<sub>2</sub>)  $\rightsquigarrow_k$  (cms<sub>2</sub>' , mem<sub>2</sub>' )  $\wedge$   
makes-compatible (cms<sub>1</sub>' , mem<sub>1</sub>' ) (cms<sub>2</sub>' , mem<sub>2</sub>' ) mems'

**proof** –

**let** ?X =  $\lambda i. differing\ vars\ lists\ mem_1\ mem_2\ mems\ i$   
**from** sound-modes compat modes-eq **have**  
 $a: \forall i < length\ cms_1. \forall x \in (?X\ i). doesnt\ read\ or\ modify\ (fst\ (cms_1\ !\ i))\ x \wedge$   
 $doesnt\ read\ or\ modify\ (fst\ (cms_2\ !\ i))\ x$   
**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<sub>1</sub> = length cms<sub>2</sub>

**by** (metis length-map)

**let** ?mds<sub>k</sub> = snd (cms<sub>1</sub> ! k) **and**  
?mds<sub>k</sub>' = snd (cms<sub>1</sub>' ! k) **and**  
?mems<sub>1</sub>k = fst (mems ! k) **and**  
?mems<sub>2</sub>k = snd (mems ! k) **and**  
?n = length cms<sub>1</sub>

**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**  $\bigwedge \sigma. dom \sigma = ?X k \implies ?mems_1 k [\mapsto \sigma] = mem_1 [\mapsto \sigma]$   
**using** *differing-vars-subst differing-vars-lists-def*  
**by** (*metis Un-upper1 Un-subset-iff*)

**hence**

*eval* $_{\sigma}$ :  $\bigwedge \sigma. dom \sigma = ?X k \implies (cms_1 ! k, ?mems_1 k [\mapsto \sigma]) \rightsquigarrow (cms_1' ! k, mem_1' [\mapsto \sigma])$   
**by** (*metis change-respecting-subst[rule-format, where X=?X k] c*)

**moreover**

**with** *b* **and** *compat* **have**

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

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

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

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

**then obtain** *c<sub>h</sub>* *mem<sub>h</sub>* **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**

$\forall \sigma. dom \sigma = ?X k \longrightarrow$

$(cms_2 ! k, ?mems_2k [\mapsto h] [\mapsto \sigma]) \rightsquigarrow \langle c_h, ?mds_k', mem_h [\mapsto \sigma] \rangle$   
**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*:  $\bigwedge \sigma. dom \sigma = ?X k \implies$

$(cms_2 ! k, ?mems_2k [\mapsto \sigma]) \rightsquigarrow \langle c_h, ?mds_k', mem_h [\mapsto \sigma] \rangle \wedge$   
 $(cms_1' ! k, mem_1' [\mapsto \sigma]) \approx \langle c_h, ?mds_k', mem_h [\mapsto \sigma] \rangle$   
**using** *h-prop*

**by** (*metis deterministic*)

**let**  $?\sigma\text{-}mem_2 = \text{to-partial } mem_2 \mid' ?X k$

**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 $\sigma$ -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 $_2'$ -def c $_2'$ -def*

**by** *metis*

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

**with** *i b equal-size* **have**  $(cms_2, mem_2) \rightsquigarrow_k (cms_2', mem_2')$

**by** (*metis meval-intro*)

**moreover**

**from** *equal-size* **have** *new-length*:  $length cms_1' = length cms_2'$

**unfolding** *cms $_2'$ -def*

**by** (*metis eval length-list-update meval-elim*)

**with** *modes-eq* **have**  $map snd cms_1' = map snd cms_2'$

**unfolding** *cms $_2'$ -def*

**by** (*metis b map-update snd-conv*)

**moreover**

— 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*:  $\bigwedge x. \llbracket x \in ?X k \rrbracket \implies$   
 $mem_1 x = mem_1' x \wedge mem_2 x = mem_2' x \wedge dma\ mem_1 x = dma\ mem_1' x$   
**proof**(*intro conjI*)

**fix**  $x$   
**assume**  $x \in ?X k$   
**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**  $x \in ?X k$

**hence** *eq-mem<sub>2</sub>*:  $?σ\text{-}mem_2 x = Some (mem_2 x)$   
**by** (*metis restrict-in to-partial-def*)  
**hence**  $?mems_2 k \llbracket \vdash h \rrbracket \llbracket \vdash ?σ\text{-}mem_2 \rrbracket x = mem_2 x$   
**by** (*auto simp: subst-def*)

**moreover have**  $mem_h \llbracket \vdash ?σ\text{-}mem_2 \rrbracket x = mem_2 x$   
**by** (*auto simp: subst-def ⟨x ∈ ?X k⟩ eq-mem<sub>2</sub>*)

**ultimately have**  $?mems_2 k \llbracket \vdash h \rrbracket \llbracket \vdash ?σ\text{-}mem_2 \rrbracket x = mem_h \llbracket \vdash ?σ\text{-}mem_2 \rrbracket x$   
**by** *auto*

**thus**  $mem_2 x = mem_2' x$   
**by** (*metis ⟨mem<sub>2</sub> = ?mems<sub>2</sub> k ⟩ domσ-mem<sub>2</sub> domh mem<sub>2</sub>'-def*)

*subst-overrides*)

**next**

**fix**  $x$   
**assume**  $x \in ?X k$   
**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**  $mems'\text{-}k$  **where**  $mems'\text{-}k x =$   
*(if*  $x \notin ?X k$   
*then*  $(mem_1' x, mem_2' x)$   
*else*  $(?mems_1 k x, ?mems_2 k x)$  **for**  $x$

**define**  $mems'\text{-}i$  **where**  $mems'\text{-}i i x =$   
*(if*  $((mem_1 x \neq mem_1' x \vee mem_2 x \neq mem_2' x) \wedge$   
 $(mem_1' x = mem_2' x \vee dma\ mem_1' x = High))$   
*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, snd ∘ mems'-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, snd ∘ mems'-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*:  
 $\bigwedge i. \llbracket i < \text{length } \text{cms}_1; i \neq k \rrbracket \implies \text{mems}' ! i = (\text{fst} \circ \text{mems}'\text{-}i \ i, \text{snd} \circ \text{mems}'\text{-}i \ i)$   
**unfolding** *mems'-def*  
**by** *auto*

**have** *mems'-i-1 [simp]*:  
 $\bigwedge i x. \llbracket i \neq k; i < \text{length } \text{cms}_1;$   
 $\text{mem}_1 x \neq \text{mem}_1' x \vee \text{mem}_2 x \neq \text{mem}_2' x;$   
 $\text{mem}_1' x = \text{mem}_2' x \vee \text{dma } \text{mem}_1' x = \text{High} \rrbracket \implies$   
 $\text{fst} (\text{mems}' ! i) x = \text{mem}_1' x \wedge \text{snd} (\text{mems}' ! i) x = \text{mem}_2' x$   
**unfolding** *mems'-i-def mems'-i-simp*  
**by** *auto*

**have** *mems'-i-2 [simp]*:  
 $\bigwedge i x. \llbracket i \neq k; i < \text{length } \text{cms}_1;$   
 $\text{mem}_1 x \neq \text{mem}_1' x \vee \text{mem}_2 x \neq \text{mem}_2' x;$   
 $\text{mem}_1' x \neq \text{mem}_2' x; \text{dma } \text{mem}_1' x = \text{Low} \rrbracket \implies$   
 $\text{fst} (\text{mems}' ! i) x = \text{some-val} \wedge \text{snd} (\text{mems}' ! i) x = \text{some-val}$   
**unfolding** *mems'-i-def mems'-i-simp*  
**by** *auto*

**have** *mems'-i-3 [simp]*:  
 $\bigwedge i x. \llbracket i \neq k; i < \text{length } \text{cms}_1;$   
 $\text{mem}_1 x = \text{mem}_1' x; \text{mem}_2 x = \text{mem}_2' x;$   
 $\text{dma } \text{mem}_1 x = \text{High} \wedge \text{dma } \text{mem}_1' x = \text{Low} \rrbracket \implies$   
 $\text{fst} (\text{mems}' ! i) x = \text{mem}_1 x \wedge \text{snd} (\text{mems}' ! i) x = \text{mem}_1 x$   
**unfolding** *mems'-i-def mems'-i-simp*  
**by** *auto*

**have** *mems'-i-4 [simp]*:  
 $\bigwedge i x. \llbracket i \neq k; i < \text{length } \text{cms}_1;$   
 $\text{mem}_1 x = \text{mem}_1' x; \text{mem}_2 x = \text{mem}_2' x;$   
 $\text{dma } \text{mem}_1 x = \text{Low} \vee \text{dma } \text{mem}_1' x = \text{High};$   
 $\text{dma } \text{mem}_1' x = \text{dma } \text{mem}_1 x \rrbracket \implies$   
 $\text{fst} (\text{mems}' ! i) x = \text{fst} (\text{mems}' ! i) x \wedge \text{snd} (\text{mems}' ! i) x = \text{snd} (\text{mems}' ! i) x$   
**unfolding** *mems'-i-def mems'-i-simp*  
**by** *auto*

**have** *mems'-i-5 [simp]*:  
 $\bigwedge i x. \llbracket i \neq k; i < \text{length } \text{cms}_1;$   
 $\text{mem}_1 x = \text{mem}_1' x; \text{mem}_2 x = \text{mem}_2' x;$   
 $\text{dma } \text{mem}_1 x = \text{Low} \wedge \text{dma } \text{mem}_1' x = \text{High};$   
 $\text{dma } \text{mem}_1' x \neq \text{dma } \text{mem}_1 x \rrbracket \implies$   
 $\text{fst} (\text{mems}' ! i) x = \text{mem}_1' x \wedge \text{snd} (\text{mems}' ! i) x = \text{mem}_2' x$   
**unfolding** *mems'-i-def mems'-i-simp*  
**by** *auto*

**have** *mems'-i-cases*:

$\wedge P i x.$

$\llbracket i \neq k; i < \text{length } cms_1;$

$\llbracket mem_1 x \neq mem_1' x \vee mem_2 x \neq mem_2' x;$

$mem_1' x = mem_2' x \vee dma\ mem_1' x = High;$

$fst (mems' ! i) x = mem_1' x; snd (mems' ! i) x = mem_2' x \rrbracket \implies P x;$

$\llbracket mem_1 x \neq mem_1' x \vee mem_2 x \neq mem_2' x;$

$mem_1' x \neq mem_2' x; dma\ mem_1' x = Low;$

$fst (mems' ! i) x = some\text{-}val; snd (mems' ! i) x = some\text{-}val \rrbracket \implies P x;$

$\llbracket mem_1 x = mem_1' x; mem_2 x = mem_2' x; dma\ mem_1 x = High;$

$dma\ mem_1' x = Low;$

$fst (mems' ! i) x = mem_1 x; snd (mems' ! i) x = mem_1 x \rrbracket \implies P x;$

$\llbracket mem_1 x = mem_1' x; mem_2 x = mem_2' x; dma\ mem_1 x = Low \vee dma\ mem_1' x = High;$

$dma\ mem_1' x = dma\ mem_1 x;$

$fst (mems' ! i) x = fst (mems ! i) x; snd (mems' ! i) x = snd (mems ! i) x$

$\rrbracket \implies P x;$

$\llbracket mem_1 x = mem_1' x; mem_2 x = mem_2' x; dma\ mem_1 x = Low; dma\ mem_1' x = High;$

$fst (mems' ! i) x = mem_1' x; snd (mems' ! i) x = mem_2' x \rrbracket \implies P x$      $\rrbracket$

$\implies P x$

**using** *mems'-i-1 mems'-i-2 mems'-i-3 mems'-i-4 mems'-i-5*

**by** (*metis (full-types) Sec.exhaust*)

**let**  $?X' = \lambda i. \text{differing-vars-lists } mem_1' mem_2' mems' i$

**have** *len-unchanged*:  $\text{length } cms_1' = \text{length } cms_1$

**by** (*metis cms\_2'-def equal-size length-list-update new-length*)

**have** *mm-equiv'*:  $(cms_1' ! k, \text{subst } (? \sigma\text{-}mem_2) mem_1') \approx (c_h, snd (cms_1' ! k), mem_2')$

**apply**(*simp add: mem\_2'-def*)

**apply**(*rule g[THEN conjunct2]*)

**apply**(*rule dom-restrict-total*)

**done**

**hence** *C-subst-eq*:  $\forall x \in C. (\text{subst } (? \sigma\text{-}mem_2) mem_1') x = mem_2' x$

**apply**(*rule mm-equiv-C-eq*)

**by** *simp*

**have** *low-mds-eq'*:  $(\text{subst } (? \sigma\text{-}mem_2) mem_1') =_{snd (cms_1' ! k)} mem_2'$

**apply**(*rule mm-equiv-low-eq[where c\_1=fst (cms\_1' ! k)]*)

**apply**(*force intro: mm-equiv'*)

**done**

**have** *C-subst-eq-idemp*:  $\wedge x. x \in C \implies (\text{subst } (? \sigma\text{-}mem_2) mem_1') x = mem_1' x$

**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:  $\bigwedge x. x \in C \implies mem_1' x = mem_2' x$ 
by simp

have not-control:  $\bigwedge x i. i < length\ cms_1' \implies x \in ?X' i \implies x \notin C$ 
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  $i < length\ cms_1'$ 
  have  $i < length\ cms_1'$  by (metis len-unchanged <i < length cms1'>)
  assume  $x \in ?X' i$ 
  assume  $x \in C$ 
  have  $x \notin ?X' i$ 
    using compat <i < length cms1'> len-unchanged new-length
    by (metis <x ∈ C> compat-different)
  from  $\langle x \in C \rangle$  have  $mem_1' x = mem_2' x$  by(rule C-eq)
  from  $\langle x \in C \rangle$  have dma  $mem_1' x = Low$  by(simp add: C-Low)
  show False
  proof(cases i = k)
    assume eq[simp]:  $i = k$ 
    show ?thesis
    using  $\langle x \notin ?X' i \rangle \langle x \in ?X' i \rangle$ 
    by(force simp: differing-vars-lists-def differing-vars-def)
  next
    assume neg:  $i \neq k$ 
    thus ?thesis
    using  $\langle x \in ?X' i \rangle \langle x \notin ?X' i \rangle \langle x \in C \rangle$  C-Low  $\langle mem_1' x = mem_2' x \rangle$ 
    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  $length\ cms_1' = length\ cms_1$ 
    by (metis cms2'-def equal-size length-list-update new-length)
  then show  $length\ cms_1' = length\ cms_2' \wedge length\ cms_1' = length\ mems'$ 
    using compat new-length
    unfolding mems'-def
    by auto
  next
    fix i
    fix  $\sigma :: 'Var \rightarrow 'Val$ 
    let ?mems1'i = fst (mems' ! i)

```

**let**  $?mems_2' i = snd (mems' ! i)$   
**assume**  $i-le: i < length cms_1'$   
**assume**  $dom\sigma: dom \sigma = ?X' i$   
**show**  $(cms_1' ! i, (fst (mems' ! i)) [\mapsto \sigma]) \approx (cms_2' ! i, (snd (mems' ! i)) [\mapsto \sigma])$

**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**  $\sigma'$  **where**  $\sigma' x =$   
*(if*  $x \in ?X k$   
*then*  $if x \in ?X' k$   
*then*  $\sigma x$   
*else*  $Some (mem_1' x)$   
*else*  $None)$  **for**  $x$   
**have**  $dom\sigma': dom \sigma' = ?X k$   
**using**  $\sigma'-def$  [*abs-def*]  
**apply** (*clarsimp*, *safe*)  
**by** (*metis* *domI domIff*, *metis*  $\langle i = k \rangle domD dom\sigma$ )  
**have** *diff-vars-impl* [*simp*]:  $\bigwedge x. x \in ?X' k \implies x \in ?X k$   
**proof** (*rule ccontr*)  
**fix**  $x$   
**assume**  $x \notin ?X k$   
**hence**  $mem_1 x = ?mems_1 k x \wedge mem_2 x = ?mems_2 k x$   
**by** (*metis* *differing-vars-neg*)  
**from**  $\langle x \notin ?X k \rangle$  **have**  $?mems_1' i x = mem_1' x \wedge ?mems_2' i x = mem_2' x$   
**by** *auto*  
**moreover**  
**assume**  $x \in ?X' k$   
**hence**  $mem_1' x \neq ?mems_1' i x \vee mem_2' x \neq ?mems_2' i x$   
**by** (*metis*  $\langle i = k \rangle$  *differing-vars-elim*)  
**ultimately show** *False*  
**by** *auto*  
**qed**

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

**proof** (*rule ext*)

**fix**  $x$

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

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

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

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

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

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

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

**by** (*auto simp: subst-def*)

**moreover**  
**from**  $\text{dom}\sigma'$  **and**  $\langle x \in ?X' k \rangle$  **have**  $x \in \text{dom } \sigma'$  **by** *simp*

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

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

**hence**  $?mems_1'i [\mapsto \sigma] x = ?mems_1'i x$   
**using**  $\text{dom}\sigma$   
**by** (*metis*  $\langle i = k \rangle$  *subst-not-in-dom*)  
**show** *?thesis*  
**proof**(*case-tac*  $x \in ?X k$ )  
**assume**  $x \in ?X k$   
**from**  $\langle x \notin ?X' k \rangle$  **have**  $\text{mem}_1' x = ?mems_1'i x$   
**by**(*metis* *differing-vars-neg*  $\langle i = k \rangle$ )  
**then have**  $\sigma' x = \text{Some } (?mems_1'i x)$   
**unfolding**  $\sigma'$ -*def*  
**using**  $\text{dom}\sigma'$  *domh*  
**by**(*simp* *add*:  $\langle x \in ?X k \rangle \langle x \notin ?X' k \rangle$ )  
**hence**  $\text{mem}_1' [\mapsto \sigma'] x = ?mems_1'i x$   
**unfolding** *subst-def*  
**by** (*metis* *option.simps(5)*)  
**thus** *?thesis*  
**by** (*metis*  $\langle ?mems_1'i [\mapsto \sigma] x = ?mems_1'i x \rangle$ )  
**next**  
**assume**  $x \notin ?X k$   
**then have**  $\text{mem}_1' [\mapsto \sigma'] x = \text{mem}_1' x$   
**by** (*metis*  $\text{dom}\sigma'$  *subst-not-in-dom*)  
**moreover**  
**have**  $?mems_1'i x = \text{mem}_1' x$   
**by** (*metis*  $\langle i = k \rangle \langle x \notin ?X' k \rangle$  *differing-vars-neg*)  
**ultimately show** *?thesis*  
**by** (*metis*  $\langle ?mems_1'i [\mapsto \sigma] x = ?mems_1'i x \rangle$ )  
**qed**  
**qed**  
**qed**

**moreover have**  $?mems_2'i [\mapsto \sigma] = \text{mem}_h [\mapsto \sigma']$   
**proof** (*rule ext*)  
**fix**  $x$

**show**  $?mems_2'i [\mapsto \sigma] x = \text{mem}_h [\mapsto \sigma'] x$   
**proof** (*cases*  $x \in ?X' k$ )  
**assume**  $x \in ?X' k$

**then obtain**  $v$  **where**  $\sigma x = \text{Some } v$   
**using**  $\text{dom}\sigma$   
**by**  $(\text{metis } \text{dom}D \langle i = k \rangle)$   
**hence**  $?mems_2' i \ [\mapsto \sigma] x = v$   
**using**  $\langle x \in ?X' k \rangle \text{dom}\sigma$   
**unfolding**  $\text{subst-def}$   
**by**  $(\text{metis } \text{option.simps}(5))$   
**moreover**  
**from**  $\langle x \in ?X' k \rangle$  **have**  $x \in ?X k$   
**by**  $\text{auto}$   
**hence**  $x \in \text{dom}(\sigma')$   
**by**  $(\text{metis } \text{dom}\sigma' \langle x \in ?X' k \rangle)$   
**hence**  $\text{mem}_2' \ [\mapsto \sigma'] x = v$   
**using**  $\text{dom}\sigma' c$   
**unfolding**  $\text{subst-def}$   
**by**  $(\text{metis } \sigma'\text{-def } \langle \sigma x = \text{Some } v \rangle \text{diff-vars-impl } \text{option.simps}(5) \langle x \in ?X' k \rangle)$

**ultimately show**  $?thesis$   
**by**  $(\text{metis } \text{dom}\sigma' \text{dom-restrict-total } \text{mem}_2'\text{-def } \text{subst-overrides})$   
**next**  
**assume**  $x \notin ?X' k$

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

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

**hence**  $\text{mem}_1 x = \text{mem}_1' x \wedge \text{mem}_2 x = \text{mem}_2' x$  **by**  $(\text{metis } x\text{-unchanged})$

**moreover from**  $\langle x \notin ?X' k \rangle \langle i = k \rangle$  **have**  $?mems_1' i x = \text{mem}_1' x \wedge ?mems_2' i x = \text{mem}_2' x$   
**by** $(\text{metis } \text{differing-vars-neg})$

**moreover from**  $\langle x \in ?X k \rangle$  **have**  $\text{fst}(\text{mems} ! i) x \neq \text{mem}_1 x \vee \text{snd}(\text{mems} ! i) x \neq \text{mem}_2 x$   
**by** $(\text{metis } \text{differing-vars-elim } \langle i = k \rangle)$

**moreover from**  $\langle x \in ?X k \rangle$  **have**  $\text{fst}(\text{mems}' ! i) x = \text{fst}(\text{mems} ! i) x$   
 $\wedge$   
 $\text{snd}(\text{mems}' ! i) x = \text{snd}(\text{mems} ! i) x$   
**by** $(\text{metis } \text{mems}'\text{-k-2 } \langle i = k \rangle)$

**ultimately have**  $\text{False}$  **by**  $\text{auto}$

```

      thus ?thesis by blast
    next
      assume  $x \notin ?X k$ 
      hence  $x \notin \text{dom } \sigma'$  by (simp add: dom $\sigma'$ )
      then have  $\text{mem}_h [\mapsto \sigma'] x = \text{mem}_h x$ 
        by (metis subst-not-in-dom)
      moreover
      have ? $\text{mems}_2' i x = \text{mem}_2' x$ 
        by (metis  $\langle i = k \rangle \text{ mems}'\text{-}k\text{-}1 \langle x \notin ?X k \rangle$ )

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

ultimately show
  ( $\text{cms}_1' ! i, (\text{fst } (\text{mems}' ! i)) [\mapsto \sigma] \approx (\text{cms}_2' ! i, (\text{snd } (\text{mems}' ! i)) [\mapsto \sigma])$ )
  using dom $\sigma$  dom $\sigma'$  g b  $\langle i = k \rangle$ 
  by (metis c $_2'$ -def cms $_2'$ -def equal-size nth-list-update-eq)

next
  assume  $i \neq k$ 
  define  $\sigma'$  where  $\sigma' x =$ 
    (if  $x \in ?X i$ 
     then if  $x \in ?X' i$ 
      then  $\sigma x$ 
      else Some ( $\text{mem}_1' x$ )
     else None) for  $x$ 
  let ? $\text{mems}_1 i = \text{fst } (\text{mems} ! i)$  and
      ? $\text{mems}_2 i = \text{snd } (\text{mems} ! i)$ 
  have dom  $\sigma' = ?X i$ 
    unfolding  $\sigma'$ -def
    apply auto
    apply (metis option.simps(2))
    by (metis domD dom $\sigma$ )

  have  $o: \bigwedge x.$ 
    ((? $\text{mems}_1' i [\mapsto \sigma] x \neq ?\text{mems}_1 i [\mapsto \sigma'] x \vee$ 
     ? $\text{mems}_2' i [\mapsto \sigma] x \neq ?\text{mems}_2 i [\mapsto \sigma'] x$ )  $\wedge$ 
     (dma mem $_1 x = \text{Low} \vee \text{dma mem}_1' x = \text{High}$ )  $\wedge$ 
     (dma mem $_1' x = \text{dma mem}_1 x$ )
      $\longrightarrow (\text{mem}_1' x \neq \text{mem}_1 x \vee \text{mem}_2' x \neq \text{mem}_2 x)$ )
  proof -
    fix  $x$ 
    {

```

**assume** *eq-mem*:  $mem_1' x = mem_1 x \wedge mem_2' x = mem_2 x$   
**and** *clas*:  $dma mem_1 x = Low \vee dma mem_1' x = High$   
**and** *clas-eq*:  $dma mem_1' x = dma mem_1 x$   
**hence** *mems'-simp*:  $?mems_1' i x = ?mems_1 i x \wedge ?mems_2' i x = ?mems_2 i$   
*x*  
**using** *mems'-i-4*  
**by** (*metis*  $\langle i \neq k \rangle b i$ -le length-list-update)  
**have**  
 $?mems_1' i [\mapsto \sigma] x = ?mems_1 i [\mapsto \sigma'] x \wedge ?mems_2' i [\mapsto \sigma] x = ?mems_2 i$   
 $[\mapsto \sigma'] x$   
**proof** (*cases*  $x \in ?X' i$ )  
**assume**  $x \in ?X' i$   
**hence**  $?mems_1' i x \neq mem_1' x \vee ?mems_2' i x \neq mem_2' x$   
**by** (*metis* *differing-vars-neg-intro*)  
**hence**  $x \in ?X i$   
**using** *eq-mem mems'-simp*  
**by** (*metis* *differing-vars-neg*)  
**hence**  $\sigma' x = \sigma x$   
**by** (*metis*  $\sigma'$ -def  $\langle x \in ?X' i \rangle$ )  
**thus** *?thesis*  
**by** (*clarsimp simp: subst-def mems'-simp split: option.splits*)  
**next**  
**assume**  $x \notin ?X' i$   
**hence**  $?mems_1' i x = mem_1' x \wedge ?mems_2' i x = mem_2' x$   
**by** (*metis* *differing-vars-neg*)  
**hence**  $x \notin ?X i$   
**using** *eq-mem mems'-simp*  
**by** (*auto simp: differing-vars-neg-intro*)  
**thus** *?thesis*  
**by** (*metis*  $\langle dom \sigma' = ?X i \rangle \langle x \notin ?X' i \rangle dom \sigma mems'$ -simp  
*subst-not-in-dom*)  
**qed**  
**}**  
**thus** *?thesis x* **by** *blast*  
**qed**

**have** *o-downgrade*:  $\bigwedge x. x \notin ?X' i \wedge (subst \sigma (fst (mems' ! i)) x \neq subst \sigma' (fst (mems ! i)) x \vee$   
 $subst \sigma (snd (mems' ! i)) x \neq subst \sigma' (snd (mems ! i)) x) \wedge$   
 $(dma mem_1 x = High \wedge dma mem_1' x = Low) \longrightarrow$   
 $mem_1' x \neq mem_1 x \vee mem_2' x \neq mem_2 x$

**proof** –

**fix**  $x$  {

**assume** *mem-eq*:  $mem_1' x = mem_1 x \wedge mem_2' x = mem_2 x$

**and** *clas*:  $(dma mem_1 x = High \wedge dma mem_1' x = Low)$

**and** *notin*:  $x \notin ?X' i$

**hence** *mems'-simp* [*simp*]:  $?mems_1' i x = mem_1 x \wedge ?mems_2' i x = mem_1$

$x$

```

using mems'-i-3
by (metis  $\langle i \neq k \rangle$  b i-le length-list-update)
have
   $?mems_1' i [\mapsto \sigma] x = ?mems_1 i [\mapsto \sigma'] x \wedge ?mems_2' i [\mapsto \sigma] x = ?mems_2 i$ 
 $[\mapsto \sigma'] x$ 
proof (cases  $x \in ?X' i$ )
  assume  $x \in ?X' i$ 
  thus ?thesis using notin by blast
next
  assume  $x \notin ?X' i$ 
  hence  $?mems_1' i x = mem_1' x \wedge ?mems_2' i x = mem_2' x$ 
  by (metis differing-vars-neg)
  moreover have  $x \notin ?X i$ 
  using clas compat i-le len-unchanged
  by (force)
  ultimately show ?thesis
    using dom $\sigma$   $\langle dom \sigma' = ?X i \rangle \langle x \notin ?X' i \rangle$  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:
   $\bigwedge x. mem_1' x \neq mem_1 x \vee mem_2' x \neq mem_2 x \vee$ 
   $dma mem_1' x \neq dma mem_1 x \vee dma mem_2' x \neq dma mem_2 x \implies$ 
   $\neg var-asm-not-written (snd (cms_1 ! i)) x$ 
proof –
  fix  $x$ 
  assume  $mem_1' x \neq mem_1 x \vee mem_2' x \neq mem_2 x \vee dma mem_1' x \neq$ 
   $dma mem_1 x \vee dma mem_2' x \neq dma mem_2 x$ 
  hence modified:  $\neg (doesnt-modify (fst (cms_1 ! k)) x) \vee \neg (doesnt-modify$ 
   $(fst (cms_2 ! k)) x)$ 
  using b i
  unfolding doesnt-modify-def
  by (metis surjective-pairing)
  hence modified-r:  $\neg (doesnt-read-or-modify (fst (cms_1 ! k)) x) \vee \neg$ 
   $(doesnt-read-or-modify (fst (cms_2 ! k)) x)$  using doesnt-read-or-modify-doesnt-modify
  by fastforce

from sound-modes have loc-modes:
  locally-sound-mode-use  $(cms_1 ! k, mem_1) \wedge$ 
  locally-sound-mode-use  $(cms_2 ! k, mem_2)$ 
  unfolding sound-mode-use.simps
  by (metis b equal-size list-all-length)

```

**moreover**  
**have**  $\text{snd} (cms_1 ! k) = \text{snd} (cms_2 ! k)$   
**by** (*metis b equal-size modes-eq nth-map*)  
**have**  $(cms_1 ! k, mem_1) \in \text{loc-reach} (cms_1 ! k, mem_1)$   
**using** *loc-reach.refl* **by** *auto*  
**hence** *guars*:  
 $x \in \text{snd} (cms_1 ! k) \text{ GuarNoWrite} \longrightarrow \text{doesnt-modify} (\text{fst} (cms_1 ! k))$   
 $x \wedge$   
 $x \in \text{snd} (cms_2 ! k) \text{ GuarNoWrite} \longrightarrow \text{doesnt-modify} (\text{fst} (cms_1 ! k))$   
 $x \wedge$   
 $x \in \text{snd} (cms_1 ! k) \text{ GuarNoReadOrWrite} \longrightarrow \text{doesnt-read-or-modify}$   
 $(\text{fst} (cms_1 ! k)) x \wedge$   
 $x \in \text{snd} (cms_2 ! k) \text{ GuarNoReadOrWrite} \longrightarrow \text{doesnt-read-or-modify}$   
 $(\text{fst} (cms_1 ! k)) x$   
**using** *loc-modes*  
**unfolding** *locally-sound-mode-use-def*  $\langle \text{snd} (cms_1 ! k) = \text{snd} (cms_2 ! k) \rangle$   
**by** (*metis loc-reach.refl surjective-pairing*)

**hence**  $x \notin \text{snd} (cms_1 ! k) \text{ GuarNoWrite} \wedge x \notin \text{snd} (cms_1 ! k) \text{ GuarNoReadOrWrite}$   
**using** *modified modified-r loc-modes locally-sound-mode-use-def*  
**by** (*metis (no-types, lifting)  $\langle cms_2, mem_2 \rangle \rightsquigarrow_k \langle cms_2', mem_2' \rangle$  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<sub>1</sub>)*  
**by** (*metis globally-sound-modes-compatible sound-mode-use.simps*)

**ultimately show** (*?thesis x*)  
**unfolding** *compatible-modes-def var-asm-not-written-def*  
**using**  $\langle i \neq k \rangle$  *i-le*  
**by** (*metis (no-types) b length-list-update length-map nth-map*)

**qed**

**from** *o o-downgrade* **have**  
 $p: \bigwedge x. \llbracket ?mems_1' i [\mapsto \sigma] x \neq ?mems_1 i [\mapsto \sigma'] x \vee$   
 $?mems_2' i [\mapsto \sigma] x \neq ?mems_2 i [\mapsto \sigma'] x;$   
 $x \notin ?X' i \vee ((dma mem_1 x = Low \vee dma mem_1' x = High) \wedge$   
 $(dma mem_1' x = dma mem_1 x)) \rrbracket \implies$   
 $\neg \text{var-asm-not-written} (\text{snd} (cms_1 ! i)) x$

**proof** –  
**fix**  $x$   
**assume** *mems-neq*:  
 $?mems_1' i [\mapsto \sigma] x \neq ?mems_1 i [\mapsto \sigma'] x \vee ?mems_2' i [\mapsto \sigma] x \neq ?mems_2 i$   
 $[\mapsto \sigma'] x$   
**and** *nin*:  
 $x \notin ?X' i \vee ((dma mem_1 x = Low \vee dma mem_1' x = High) \wedge$   
 $(dma mem_1' x = dma mem_1 x))$



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  $q'$ :

$\bigwedge x. \llbracket dma\ mem_1 x = Low; dma\ mem_1' x = Low;$   
 $?mems_1' i \llbracket \mapsto \sigma \rrbracket x \neq ?mems_1 i \llbracket \mapsto \sigma \rrbracket x \vee$   
 $?mems_2' i \llbracket \mapsto \sigma \rrbracket x \neq ?mems_2 i \llbracket \mapsto \sigma \rrbracket x;$   
 $x \notin ?X' i \rrbracket \implies$   
 $mem_1' x = mem_2' x$

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

)

have  $i < length\ cms_1$

**by** (metis  $cms_2'$ -def equal-size i-le length-list-update new-length)

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

bisim:  $(cms_1 ! i, ?mems_1 i \llbracket \mapsto \sigma \rrbracket) \approx (cms_2 ! i, ?mems_2 i \llbracket \mapsto \sigma \rrbracket)$

**by** auto

**define**  $\sigma'_k$  **where**  $\sigma'_k x = (if\ x \in ?X\ k\ then\ Some\ (undefined::'Val)\ else$

None) **for**  $x$

**have**  $dom\ \sigma'_k = ?X\ k$  **unfolding**  $\sigma'_k$ -def **by** (simp add: dom-def)

**with** compat **and**  $\langle dom\ \sigma'_k = ?X\ k \rangle$  **and**  $b$  **have**

bisim<sub>k</sub>:  $(cms_1 ! k, ?mems_1 k \llbracket \mapsto \sigma'_k \rrbracket) \approx (cms_2 ! k, ?mems_2 k \llbracket \mapsto \sigma'_k \rrbracket)$

**by** auto

have  $q$ -downgrade:

$\bigwedge x. \llbracket dma\ mem_1 x = High; dma\ mem_1' x = Low;$   
 $?mems_1' i \llbracket \mapsto \sigma \rrbracket x \neq ?mems_1 i \llbracket \mapsto \sigma \rrbracket x \vee$   
 $?mems_2' i \llbracket \mapsto \sigma \rrbracket x \neq ?mems_2 i \llbracket \mapsto \sigma \rrbracket x;$   
 $x \notin ?X' i \rrbracket \implies$   
 $mem_1' x = mem_2' x$

**by** (metis (erased, opaque-lifting)  $\langle i \neq k \rangle$  compat-different-vars i-le len-unchanged mems'-i-2 o-downgrade)

have  $q$ :  $\bigwedge x. \llbracket dma\ mem_1' x = Low;$

```

      ?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 (?mems1 i [↦ σ']) = dma mem1
  apply(rule dma-C)
  apply(rule ballI)
  apply(case-tac x ∈ ?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 mem2 |' differing-vars-lists mem1
mem2 mems k)) mem1') = dma mem1'
  using compat b
  by(force intro!: dma-C subst-not-in-dom)

have A-correct:
  ∧ x.
  ?mems1 i [↦ σ'] [|1 A] x = ?mems1' i [↦ σ] x ∧
  ?mems2 i [↦ σ'] [|2 A] x = ?mems2' i [↦ σ] x
proof -
  fix x
  show ?thesis x
  (is ?Eq1 ∧ ?Eq2)
  proof (cases x ∈ ?Δ)
  assume x ∈ ?Δ
  hence diff:
    ?mems1' i [↦ σ] x ≠ ?mems1 i [↦ σ'] x ∨ ?mems2' i [↦ σ] x ≠ ?mems2
[↦ σ'] x
  by (auto simp: differing-vars-def)
  show ?thesis
  proof (cases dma (?mems1' i [↦ σ]) x)
  assume dma (?mems1' i [↦ σ]) x = High
  from ⟨dma (?mems1' i [↦ σ]) x = High⟩ have A-simp [simp]:
    A x = Some (?mems1' i [↦ σ] x, ?mems2' i [↦ σ] x)
  unfolding A-def
  by (metis ⟨x ∈ ?Δ⟩)
  from A-simp have ?Eq1 ?Eq2
  unfolding A-def apply-adaptation-def
  by auto
  thus ?thesis
  by auto
  next
  assume dma (?mems1' i [↦ σ]) x = Low
  show ?thesis
  proof (cases x ∈ ?X' i)
  assume x ∈ ?X' i

```

**then obtain**  $v$  **where**  $\sigma x = \text{Some } v$   
**by** (*metis domD dom $\sigma$* )  
**hence eq:**  $?mems_1'i \ [\vdash \sigma] \ x = v \wedge ?mems_2'i \ [\vdash \sigma] \ x = v$   
**unfolding** *subst-def*  
**by** *auto*  
**moreover**  
**from**  $\langle x \in ?X' \ i \rangle$  **and**  $\langle dma \ ( ?mems_1'i \ [\vdash \sigma]) \ x = Low \rangle$  **have**  $A\text{-simp}$

[*simp*]:

$A \ x = (\text{case } \sigma \ x \ \text{of}$   
 $\quad \text{Some } v \Rightarrow \text{Some } (v, v)$   
 $\quad | \ \text{None} \Rightarrow \text{None})$   
**unfolding** *A-def*  
**by** (*metis Sec.simps(1)  $\langle x \in ?\Delta \rangle$* )  
**ultimately show** *?thesis*  
**using**  $domA \ \langle x \in ?\Delta \rangle \ \langle \sigma \ x = \text{Some } v \rangle$   
**by** (*auto simp: apply-adaptation-def*)

**next**  
**assume**  $x \notin ?X' \ i$

**hence**  $A\text{-simp}$  [*simp*]:  $A \ x = \text{Some } (mem_1' \ x, mem_1' \ x)$   
**unfolding** *A-def*  
**using**  $\langle x \in ?\Delta \rangle \ \langle x \notin ?X' \ i \rangle \ \langle dma \ ( ?mems_1'i \ [\vdash \sigma]) \ x = Low \rangle$   
**by** *auto*

**from**  $q$  **have**  $mem_1' \ x = mem_2' \ x$   
**by** (*metis  $\langle dma \ ( ?mems_1'i \ [\vdash \sigma]) \ x = Low \rangle \ diff \ \langle x \notin ?X' \ i \rangle \ dma\text{-eq}$* )

*dma-eq''*)

**from**  $\langle x \notin ?X' \ i \rangle$  **have**  
 $?mems_1'i \ [\vdash \sigma] \ x = ?mems_1'i \ x \wedge ?mems_2'i \ [\vdash \sigma] \ x = ?mems_2'i \ x$   
**by** (*metis dom $\sigma$  subst-not-in-dom*)  
**moreover**  
**from**  $\langle x \notin ?X' \ i \rangle$  **have**  $?mems_1'i \ x = mem_1' \ x \wedge ?mems_2'i \ x =$

*mem\_2' \ x*

**by** (*metis differing-vars-neg*)  
**ultimately show** *?thesis*  
**using**  $\langle mem_1' \ x = mem_2' \ x \rangle$   
**by** (*auto simp: apply-adaptation-def*)

**qed**  
**qed**

**next**  
**assume**  $x \notin ?\Delta$   
**hence**  $A \ x = None$   
**by** (*metis domA domIff*)  
**from**  $\langle A \ x = None \rangle$  **have**  $x \notin dom \ A$   
**by** (*metis domIff*)  
**from**  $\langle x \notin ?\Delta \rangle$  **have**  $?mems_1 \ i \ [\vdash \sigma] \ [\|_1 \ A] \ x = ?mems_1' \ i \ [\vdash \sigma] \ x \wedge$

```

      ?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 =

```

$High) \wedge (dma\ mem_1' x = dma\ mem_1 x))$   
**hence**  $x \in ?X' i ((dma\ mem_1 x = High \wedge dma\ mem_1' x = Low) \vee$   
 $(dma\ mem_1' x \neq dma\ mem_1 x))$   
**by**  $(metis\ Sec.exhaust)+$   
  
**thus**  $?thesis$  **by**  $(fastforce\ simp\ add:\ modifies-no-var-asm-not-written)$   
**qed**  
  
**next**  
  
**have**  $reclas: (\forall x. dma ((subst\ \sigma' (fst (mems ! i))) [\![\!_1 A]) x \neq dma (subst$   
 $\sigma' (fst (mems ! i))) x \longrightarrow$   
 $\neg var-asm-not-written (snd (cms_1 ! i)) x)$   
**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**  $(metis\ \langle i < length\ cms_1 \rangle\ equal-size\ modes-eq\ nth-map)$   
  
**hence**  $low-mds-eq: (subst\ \sigma' (fst (mems ! i))) =_{snd (cms_1 ! i)}^l (subst\ \sigma'$   
 $(snd (mems ! i)))$   
**apply**  $-$   
**apply**  $(rule\ mm-equiv-low-eq[\mathbf{where}\ c_1=fst (cms_1 ! i)\ \mathbf{and}\ c_2=fst (cms_2$   
 $! i)])$   
**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**  $low-mds-eq_k: (subst\ \sigma'_k (fst (mems ! k))) =_{snd (cms_1 ! k)}^l (subst$   
 $\sigma'_k (snd (mems ! k)))$   
**apply**  $-$   
**apply**  $(rule\ mm-equiv-low-eq[\mathbf{where}\ c_1=fst (cms_1 ! k)\ \mathbf{and}\ c_2=fst (cms_2$   
 $! k)])$   
**using**  $bisim_k$   
**by**  $(metis\ prod.collapse)$   
  
**have**  $eq: \forall x. dma ((subst\ \sigma' (fst (mems ! i))) [\![\!_1 A]) x = Low \wedge (x \in$   
 $(snd (cms_1 ! i))\ AsmNoReadOrWrite \longrightarrow x \in \mathcal{C}) \longrightarrow$   
 $(subst\ \sigma' (fst (mems ! i))) [\![\!_1 A] x = (subst\ \sigma' (snd (mems ! i))) [\![\!_2 A] x$   
  
**apply**  $(clarsimp\ simp:\ adapt-eq\ dma-eq)$   
**apply**  $(case-tac\ x \in dom\ \sigma)$   
**apply**  $(force\ simp:\ subst-def)$   
**apply**  $(simp\ add:\ subst-not-in-dom)$   
**apply**  $(simp\ add:\ dom\ \sigma)$   
**apply**  $(clarsimp\ simp:\ differing-vars-lists-def\ differing-vars-def)$

```

apply(case-tac  $i = k$ )
apply(simp add:  $\langle i \neq k \rangle$ )
apply(erule mems'-i-cases)
  apply(rule  $\langle i < \text{length } cms_1 \rangle$  [simplified len-unchanged])
  apply force
  apply fastforce
  apply clarsimp
apply clarsimp

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

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

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

with bisim have  $(cms_1 ! i, ?mems_1 i [\mapsto \sigma'] [\|_1 A]) \approx (cms_2 ! i, ?mems_2 i [\mapsto \sigma'] [\|_2 A])$ 
using 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  $\langle i \neq k \rangle$  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 cases: 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: cases)+
  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
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:  $x \in \mathcal{C} \vee (\forall i < \text{length } cms_1. x \notin \text{snd } (cms_1 ! i) \text{ AsmNoReadOrWrite})$   
**shows**  $mem_1 x = mem_2 x$   
**proof** –  
**let**  $?X = \lambda i. \text{differing-vars-lists } mem_1 mem_2 mems i$   
**from**  $compat$  **have**  $(\text{length } cms_1 = 0 \wedge mem_1 =^l mem_2) \vee (\forall x. \exists j. j < \text{length } cms_1 \wedge x \notin ?X j)$   
**by** *auto*  
**thus**  $mem_1 x = mem_2 x$   
**proof**  
**assume**  $\text{length } cms_1 = 0 \wedge mem_1 =^l mem_2$   
**with**  $x$ -low **show**  $?thesis$   
**by** (*simp add: low-eq-def*)  
**next**  
**assume**  $\forall x. \exists j. j < \text{length } cms_1 \wedge x \notin ?X j$   
**then obtain**  $j$  **where**  $j$ -prop:  $j < \text{length } cms_1 \wedge x \notin ?X j$   
**by** *auto*  
**let**  $?mems_1 j = \text{fst } (mems ! j)$  **and**  
 $?mems_2 j = \text{snd } (mems ! j)$   
  
**obtain**  $\sigma :: 'Var \rightarrow 'Val$  **where**  $dom \sigma: dom \sigma = ?X j$   
**by** (*metis dom-restrict-total*)  
  
**from**  $compat$   $x$ -low *makes-compatible-dma-eq*  $j$ -prop  $\langle dom \sigma = ?X j \rangle$   
**have**  $x$ -low:  $dma (?mems_1 j [\mapsto \sigma]) x = Low$   
**by** *metis*  
  
**from**  $dom \sigma$   $compat$  **and**  $j$ -prop **have**  $(cms_1 ! j, ?mems_1 j [\mapsto \sigma]) \approx (cms_2 ! j, ?mems_2 j [\mapsto \sigma])$   
**by** *auto*  
  
**moreover**  
**have**  $\text{snd } (cms_1 ! j) = \text{snd } (cms_2 ! j)$   
**using** *modes-eq*  
**by** (*metis j-prop length-map nth-map*)  
  
**ultimately have**  $?mems_1 j [\mapsto \sigma] = \text{snd } (cms_1 ! j)^l ?mems_2 j [\mapsto \sigma]$   
**using** *modes-eq j-prop*  
**by** (*metis mm-equiv-low-eq prod.collapse*)  
**hence**  $?mems_1 j x = ?mems_2 j x$   
**using**  $x$ -low  $x$ -readable  $j$ -prop  $\langle dom \sigma = ?X j \rangle$   
**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**  $loc\text{-}reach \langle c', mds', mem' \rangle \subseteq loc\text{-}reach \langle c, mds, mem \rangle$

**proof** (*clarify*)

**fix**  $c'' mds'' mem''$

**from** *eval* **have**  $\langle c', mds', mem' \rangle \in loc\text{-}reach \langle c, mds, mem \rangle$

**by** (*metis loc-reach.refl loc-reach.step surjective-pairing*)

**assume**  $\langle c'', mds'', mem'' \rangle \in loc\text{-}reach \langle c', mds', mem' \rangle$

**thus**  $\langle c'', mds'', mem'' \rangle \in loc\text{-}reach \langle c, mds, mem \rangle$

**apply** *induct*

**apply** (*metis*  $\langle \langle c', mds', mem' \rangle \in loc\text{-}reach \langle c, mds, mem \rangle \rangle$  *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**  $\langle c', mds', mem' \rangle \in loc\text{-}reach \langle c, mds, mem \rangle$

**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*:

$(cms, mem) \rightsquigarrow_k (cms', mem') \implies$   
 $(cms, mem) \rightarrow_{[k]} (cms', mem')$

**apply** (*simp*)

**done**

**lemma** *meval-sched-ConsI*:

$(cms, mem) \rightsquigarrow_k (cms', mem') \implies$   
 $(cms', mem') \rightarrow_{sched} (cms'', mem'') \implies$   
 $(cms, mem) \rightarrow_{(k\#sched)} (cms'', mem'')$

**by** *fastforce*

**lemma** *reachable-modes-subset*:

**assumes** *eval*:  $(cms, mem) \rightsquigarrow_k (cms', mem')$

**shows**  $reachable\text{-}mode\text{-}states \langle cms', mem' \rangle \subseteq reachable\text{-}mode\text{-}states \langle cms, mem \rangle$

**proof**

**fix** *mdss*

**assume**  $mdss \in reachable\text{-}mode\text{-}states \langle cms', mem' \rangle$

**thus**  $mdss \in reachable\text{-}mode\text{-}states \langle cms, mem \rangle$

**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:  $\forall x. \text{var-asm-not-written mds } x \longrightarrow \text{mem}_1 x = \text{mem}_2 x \wedge \text{dma mem}_1 x = \text{dma mem}_2 x$*   
**shows**  $\langle c', \text{mds}', \text{mem}' \rangle \in \text{loc-reach } \langle c, \text{mds}, \text{mem}_1 \rangle \implies \langle c', \text{mds}', \text{mem}' \rangle \in \text{loc-reach } \langle c, \text{mds}, \text{mem}_2 \rangle$   
**proof** –  
**let**  $?lc = \langle c', \text{mds}', \text{mem}' \rangle$   
**assume**  $?lc \in \text{loc-reach } \langle c, \text{mds}, \text{mem}_1 \rangle$   
**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:  $\forall x. \text{var-asm-not-written mds } x \longrightarrow \text{mem}' x = \text{mem } x \wedge \text{dma mem}' x = \text{dma mem } x$*   
**shows**  $\text{loc-reach } \langle c, \text{mds}, \text{mem} \rangle = \text{loc-reach } \langle c, \text{mds}, \text{mem}' \rangle$   
**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*:  $\forall i < \text{length } cms. \text{locally-sound-mode-use}$   
(*cms ! i*, *mem*)  
**unfolding** *sound-mode-use-def*  
**by** *simp* (*metis list-all-length*)  
**from** *eval* **obtain** *k cms<sub>k</sub>'* **where**  
*ev*: (*cms ! k*, *mem*)  $\rightsquigarrow$  (*cms<sub>k</sub>'*, *mem'*)  $\wedge k < \text{length } cms \wedge cms' = cms [k :=$   
*cms<sub>k</sub>']*  
**by** (*metis meval-elim*)  
**hence** *length cms* = *length cms'*  
**by** *auto*  
**have**  $\bigwedge i. i < \text{length } cms' \implies \text{locally-sound-mode-use } (cms' ! i, mem')$   
**proof** –  
**fix** *i*  
**assume** *i-le*: *i < length cms'*  
**thus** *?thesis i*  
**proof** (*cases i = k*)  
**assume** *i = k*  
**thus** *?thesis*  
**using** *i-le ev loc-sound*  
**by** (*metis (opaque-lifting, no-types) locally-sound-modes-invariant nth-list-update*  
*surj-pair*)  
**next**  
**assume** *i  $\neq$  k*  
**hence** *cms' ! i = cms ! i*  
**by** (*metis ev nth-list-update-neq*)  
**from** *sound-modes* **have** *compatible-modes* (*map snd cms*)  
**unfolding** *sound-mode-use.simps*  
**by** (*metis globally-sound-modes-compatible*)  
**hence**  $\bigwedge x. \text{var-asm-not-written } (snd (cms ! i)) x \implies x \in \text{snd } (cms ! k)$   
*GuarNoWrite  $\vee x \in \text{snd } (cms ! k)$  GuarNoReadOrWrite*  
**unfolding** *compatible-modes-def*  
**by** (*metis (no-types)  $\langle i \neq k \rangle \langle \text{length } cms = \text{length } cms' \rangle ev i-le \text{length-map}$*   
*nth-map var-asm-not-written-def*)  
**hence**  $\bigwedge x. \text{var-asm-not-written } (snd (cms ! i)) x \implies \text{doesn't-modify } (fst (cms$   
*! k)) x*  
**using** *ev loc-sound*  
**unfolding** *locally-sound-mode-use-def*  
**by** (*metis loc-reach.refl surjective-pairing doesn't-read-or-modify-doesn't-modify*)  
**with** *ev* **have**  $\bigwedge x. \text{var-asm-not-written } (snd (cms ! i)) x \implies mem\ x = mem'$   
*x  $\wedge dma\ mem\ x = dma\ mem'\ x$*   
**unfolding** *doesn't-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:
  (cms1, mem1) →ns@ms (cms1', mem1') =
  (∃ cms1'' mem1''. (cms1, mem1) →ns (cms1'', mem1'') ∧ (cms1'', mem1'') →ms
  (cms1', mem1'))
apply (induct ns arbitrary: ms cms1 mem1)
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:
  (cms1, mem1) →ns@[n] (cms1', mem1') ⇒
  ∃ cms1'' mem1''. (cms1, mem1) →ns (cms1'', mem1'') ∧ (cms1'', mem1'') ↪n
  (cms1', mem1'))
apply (fastforce dest: meval-sched-appD)
done

lemma meval-sched-snocI:
  (cms1, mem1) →ns (cms1'', mem1'') ∧ (cms1'', mem1'') ↪n (cms1', mem1') ⇒
  (cms1, mem1) →ns@[n] (cms1', mem1')
apply (fastforce intro: meval-sched-appI)
done

lemma makes-compatible-eval-sched:
assumes compat: makes-compatible (cms1, mem1) (cms2, mem2) mems
assumes modes-eq: map snd cms1 = map snd cms2
assumes sound-modes: sound-mode-use (cms1, mem1) sound-mode-use (cms2, mem2)
assumes eval: (cms1, mem1) →sched (cms1', mem1')
shows ∃ cms2' mem2' mems'. sound-mode-use (cms1', mem1') ∧
sound-mode-use (cms2', mem2') ∧
map snd cms1' = map snd cms2' ∧

```

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

**proof** –

**from** *eval show* ?thesis

**proof** (induct sched arbitrary: cms<sub>1</sub>' mem<sub>1</sub>' rule: rev-induct)

case Nil

hence cms<sub>1</sub>' = cms<sub>1</sub> ∧ mem<sub>1</sub>' = mem<sub>1</sub>

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<sub>1</sub>'' mem<sub>1</sub>'' where eval'':

(cms<sub>1</sub>, mem<sub>1</sub>) →<sub>ns</sub> (cms<sub>1</sub>'', mem<sub>1</sub>'') ∧ (cms<sub>1</sub>'', mem<sub>1</sub>'') ~<sub>n</sub> (cms<sub>1</sub>', mem<sub>1</sub>') )

by (metis meval-sched-snocD prod-cases3 snd-conv)

hence (cms<sub>1</sub>'', mem<sub>1</sub>'') ~<sub>n</sub> (cms<sub>1</sub>', mem<sub>1</sub>') ..

moreover

from eval'' obtain cms<sub>2</sub>'' mem<sub>2</sub>'' mems'' where IH:

sound-mode-use (cms<sub>1</sub>'', mem<sub>1</sub>'') ∧

sound-mode-use (cms<sub>2</sub>'', mem<sub>2</sub>'') ∧

map snd cms<sub>1</sub>'' = map snd cms<sub>2</sub>'' ∧

(cms<sub>2</sub>, mem<sub>2</sub>) →<sub>ns</sub> (cms<sub>2</sub>'', mem<sub>2</sub>'') ∧

makes-compatible (cms<sub>1</sub>'', mem<sub>1</sub>'') (cms<sub>2</sub>'', mem<sub>2</sub>'') mems''

using snoc

by metis

ultimately obtain cms<sub>2</sub>' mem<sub>2</sub>' mems' where

map snd cms<sub>1</sub>' = map snd cms<sub>2</sub>' ∧

(cms<sub>2</sub>'', mem<sub>2</sub>'') ~<sub>n</sub> (cms<sub>2</sub>', mem<sub>2</sub>') ∧

makes-compatible (cms<sub>1</sub>', mem<sub>1</sub>') (cms<sub>2</sub>', mem<sub>2</sub>') 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*:

$i < n \implies x \notin \text{differing-vars-lists } mem_1 \ mem_2 \ (\text{zip } (\text{replicate } n \ mem_1) \ (\text{replicate } n \ mem_2)) \ i$

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<sub>1</sub> =<sup>l</sup> mem<sub>2</sub>

shows makes-compatible (cmds, mem<sub>1</sub>)

(cmds, mem<sub>2</sub>)



(replicate (length cmds) (mem<sub>1</sub>, mem<sub>2</sub>))

**proof** –

**let** ?n = length cmds

**let** ?mems = replicate ?n (mem<sub>1</sub>, mem<sub>2</sub>) **and**

?mdss = map snd cmds

**let** ?X = differing-vars-lists mem<sub>1</sub> mem<sub>2</sub> ?mems

**have** diff-empty:  $\forall i < ?n. ?X\ i = \{\}$

**by** (metis differing-vars-initially-empty ex-in-conv min.idem zip-replicate)

**show** ?thesis

**proof**

**show** length cmds = length cmds  $\wedge$  length cmds = length ?mems

**by** auto

**next**

**fix** i **and**  $\sigma :: 'Var \Rightarrow 'Val\ option$

**let** ?mems<sub>1</sub>i = fst (?mems ! i) **and** ?mems<sub>2</sub>i = snd (?mems ! i)

**let** ?mdss<sub>i</sub> = ?mdss ! i

**assume** i:  $i < \text{length}\ cmds$

**assume** dom $\sigma$ : dom  $\sigma =$

differing-vars-lists mem<sub>1</sub> mem<sub>2</sub>

(replicate (length cmds) (mem<sub>1</sub>, mem<sub>2</sub>)) i

**from** i **have** ?mems<sub>1</sub>i = mem<sub>1</sub> ?mems<sub>2</sub>i = mem<sub>2</sub>

**by** auto

**with** dom $\sigma$  **have** [simp]: dom  $\sigma = \{\}$  **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**  $\bigwedge mem_1\ mem_2. mem_1 = ?mdss_i^l\ mem_2 \implies$

(cmds ! i, mem<sub>1</sub>)  $\approx$  (cmds ! i, mem<sub>2</sub>)

**using** com-sifum-secure-def low-indistinguishable-def

**by** auto

**moreover**

**have**  $\bigwedge x. \sigma\ x = None$  **using**  $\langle dom\ \sigma = \{\} \rangle$

**by** (metis domIff empty-iff)

**hence** [simp]:  $\bigwedge mem. mem\ [\mapsto\ \sigma] = mem$

**by**(simp add: subst-def)

**from** i **have** ?mems<sub>1</sub>i = mem<sub>1</sub> ?mems<sub>2</sub>i = mem<sub>2</sub>

**by** auto

**with** low-eq **have** ?mems<sub>1</sub>i  $[\mapsto\ \sigma] = ?mdss_i^l\ ?mems_2i\ [\mapsto\ \sigma]$

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

**ultimately show** (cmds ! i, ?mems<sub>1</sub>i  $[\mapsto\ \sigma]$ )  $\approx$  (cmds ! i, ?mems<sub>2</sub>i  $[\mapsto\ \sigma]$ )

**by** simp

**next**

```

fix  $i\ x$ 
assume  $i < \text{length } \text{cmds}$ 
with diff-empty show  $x \notin ?X\ i$  by auto
next
show  $(\text{length } \text{cmds} = 0 \wedge \text{mem}_1 =^l \text{mem}_2) \vee (\forall x. \exists i < \text{length } \text{cmds}. x \notin$ 
 $?X\ i)$ 
using diff-empty
by  $(\text{metis } \text{bot-less } \text{bot-nat-def } \text{empty-iff } \text{length-zip } \text{low-eq } \text{min-0L})$ 
qed
qed

```

**theorem** *sifum-compositionality-cont:*

```

assumes com-secure: list-all com-sifum-secure cmds
assumes sound-modes:  $\forall \text{ mem}. \text{INIT } \text{mem} \longrightarrow \text{sound-mode-use } (\text{cmds}, \text{mem})$ 
shows prog-sifum-secure-cont cmds
unfolding prog-sifum-secure-cont-def
using assms
proof (clarify)
fix  $\text{mem}_1\ \text{mem}_2 :: 'Var \Rightarrow 'Val$ 
fix  $\text{sched } \text{cms}_1'\ \text{mem}_1'$ 
let  $?n = \text{length } \text{cmds}$ 
let  $?mems = \text{zip } (\text{replicate } ?n\ \text{mem}_1) (\text{replicate } ?n\ \text{mem}_2)$ 
assume  $\text{INIT}_1: \text{INIT } \text{mem}_1$  and  $\text{INIT}_2: \text{INIT } \text{mem}_2$ 
assume low-eq:  $\text{mem}_1 =^l \text{mem}_2$ 
with com-secure have compat:
 $\text{makes-compatible } (\text{cmds}, \text{mem}_1) (\text{cmds}, \text{mem}_2) ?mems$ 
by  $(\text{metis } \text{compatible-refl } \text{fst-conv } \text{length-replicate } \text{map-replicate } \text{snd-conv } \text{zip-eq-conv}$ 
 $\text{INIT}_1\ \text{INIT}_2)$ 

```

**also assume**  $\text{eval}: (\text{cmds}, \text{mem}_1) \rightarrow_{\text{sched}} (\text{cms}_1', \text{mem}_1')$

**ultimately obtain**  $\text{cms}_2'\ \text{mem}_2'\ \text{mems}'$

```

where  $p: \text{map } \text{snd } \text{cms}_1' = \text{map } \text{snd } \text{cms}_2' \wedge$ 
 $(\text{cmds}, \text{mem}_2) \rightarrow_{\text{sched}} (\text{cms}_2', \text{mem}_2') \wedge$ 
 $\text{makes-compatible } (\text{cms}_1', \text{mem}_1') (\text{cms}_2', \text{mem}_2')\ \text{mems}'$ 
using sound-modes makes-compatible-eval-sched INIT1 INIT2
by blast

```

```

thus  $\exists \text{cms}_2'\ \text{mem}_2'. (\text{cmds}, \text{mem}_2) \rightarrow_{\text{sched}} (\text{cms}_2', \text{mem}_2') \wedge$ 
 $\text{map } \text{snd } \text{cms}_1' = \text{map } \text{snd } \text{cms}_2' \wedge$ 
 $\text{length } \text{cms}_2' = \text{length } \text{cms}_1' \wedge$ 
 $(\forall x. \text{dma } \text{mem}_1'\ x = \text{Low} \wedge (x \in \mathcal{C} \vee (\forall i < \text{length } \text{cms}_1'. x$ 
 $\notin \text{snd } (\text{cms}_1'\ i)\ \text{AsmNoReadOrWrite}))$ 
 $\longrightarrow \text{mem}_1'\ x = \text{mem}_2'\ x)$ 
using p compat-low-eq
by  $(\text{metis } \text{length-map})$ 
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 ← 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 evalA :: ('Var, 'Val) Mem ⇒ 'AExp ⇒ 'Val
  fixes evalB :: ('Var, 'Val) Mem ⇒ 'BExp ⇒ bool
  fixes aexp-vars :: 'AExp ⇒ 'Var set
  fixes bexp-vars :: 'BExp ⇒ 'Var set
  assumes Var-finite : finite {(x :: 'Var). True}
  assumes eval-vars-detA : [ [ ∀ x ∈ aexp-vars e. mem1 x = mem2 x ] ] ⇒ evalA
mem1 e = evalA mem2 e
  assumes eval-vars-detB : [ [ ∀ x ∈ bexp-vars b. mem1 x = mem2 x ] ] ⇒ evalB
mem1 b = evalB mem2 b
```

```
locale sifum-lang = sifum-lang-no-dma evalA evalB aexp-vars bexp-vars
  for evalA :: ('Var, 'Val) Mem ⇒ 'AExp ⇒ 'Val
  and evalB :: ('Var, 'Val) Mem ⇒ 'BExp ⇒ bool
  and aexp-vars :: 'AExp ⇒ 'Var set
  and bexp-vars :: 'BExp ⇒ 'Var set+
  fixes dma :: 'Var ⇒ 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**  $conf_w-abv :: ('Var, 'AExp, 'BExp) Stmt \Rightarrow$   
 $'Var Mds \Rightarrow ('Var, 'Val) Mem \Rightarrow (-,-) LocalConf$   
 $(\langle -, -, - \rangle_w [0, 120, 120] 100)$   
**where**  
 $\langle c, mds, mem \rangle_w \equiv ((c, mds), mem)$

## 4.2 Semantics

**primrec**  $update-modes :: 'Var ModeUpd \Rightarrow 'Var Mds \Rightarrow 'Var Mds$

**where**

$update-modes (Acq\ x\ m)\ mds = mds\ (m := insert\ x\ (mds\ m)) \mid$

$update-modes (Rel\ x\ m)\ mds = mds\ (m := \{y.\ y \in mds\ m \wedge y \neq x\})$

**fun**  $updated-var :: 'Var ModeUpd \Rightarrow 'Var$

**where**

$updated-var (Acq\ x\ -) = x \mid$

$updated-var (Rel\ x\ -) = x$

**fun**  $updated-mode :: 'Var ModeUpd \Rightarrow Mode$

**where**

$updated-mode (Acq\ -\ m) = m \mid$

$updated-mode (Rel\ -\ m) = m$

**inductive-set**  $eval_w-simple :: (('Var, 'AExp, 'BExp) Stmt \times ('Var, 'Val) Mem)$   
 $rel$

**and**  $eval_w-simple-abv :: (('Var, 'AExp, 'BExp) Stmt \times ('Var, 'Val) Mem) \Rightarrow$   
 $('Var, 'AExp, 'BExp) Stmt \times ('Var, 'Val) Mem \Rightarrow bool$

(**infixr**  $\rightsquigarrow_s$  60)

**where**

$c \rightsquigarrow_s c' \equiv (c, c') \in eval_w-simple \mid$

$assign: ((x \leftarrow e, mem), (Stop, mem\ (x := eval_A\ mem\ e))) \in eval_w-simple \mid$

$skip: ((Skip, mem), (Stop, mem)) \in eval_w-simple \mid$

$seq-stop: ((Seq\ Stop\ c, mem), (c, mem)) \in eval_w-simple \mid$

*if-true*:  $\llbracket \text{eval}_B \text{ mem } b \rrbracket \Longrightarrow ((\text{If } b \text{ t } e, \text{ mem}), (t, \text{ mem})) \in \text{eval}_w\text{-simple} \mid$   
*if-false*:  $\llbracket \neg \text{eval}_B \text{ mem } b \rrbracket \Longrightarrow ((\text{If } b \text{ t } e, \text{ mem}), (e, \text{ mem})) \in \text{eval}_w\text{-simple} \mid$   
*while*:  $((\text{While } b \text{ c}, \text{ mem}), (\text{If } b \text{ (c ;; While } b \text{ c) Stop}, \text{ mem})) \in \text{eval}_w\text{-simple}$

**lemma** *cond*:

$((\text{If } b \text{ t } e, \text{ mem}), (\text{if eval}_B \text{ mem } b \text{ then } t \text{ else } e, \text{ mem})) \in \text{eval}_w\text{-simple}$   
**apply**(*case-tac eval<sub>B</sub> mem b*)  
**apply**(*auto intro: eval<sub>w</sub>-simple.intros*)  
**done**

**primrec** *cxt-to-stmt* :: ('Var, 'AExp, 'BExp) EvalCxt  $\Rightarrow$  ('Var, 'AExp, 'BExp) Stmt

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

**lemma** *rtrancl-mono-proof[mono]*:

$(\bigwedge a \ b. x \ a \ b \longrightarrow y \ a \ b) \Longrightarrow \text{rtranclp } x \ a \ b \longrightarrow \text{rtranclp } y \ a \ b$   
**apply** (*rule impI, rotate-tac, induct rule: rtranclp.induct*)  
**apply** *simp-all*  
**apply** (*metis rtranclp.intros*)  
**done**

**lemma** *trancl-mono-proof[mono]*:

$(\bigwedge a \ b. x \ a \ b \longrightarrow y \ a \ b) \Longrightarrow \text{tranclp } x \ a \ b \longrightarrow \text{tranclp } y \ a \ b$   
**apply** (*rule impI, rotate-tac, induct rule: tranclp.induct*)  
**apply** *simp-all*  
**apply** *blast*  
**by** *fastforce*

**inductive** *no-await* :: ('Var, 'AExp, 'BExp) Stmt  $\Rightarrow$  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  $\Rightarrow$  bool **where**

*is-final* Stop |  
*is-final* c  $\Longrightarrow$  *is-final* (c@[m])

**inductive-set** *eval<sub>w</sub>* :: ('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf rel

**and** *eval<sub>w</sub>-abv* :: ('Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf  $\Rightarrow$

((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf*  $\Rightarrow$  *bool*

(**infixr**  $\rightsquigarrow_w$  60)

**where**

$c \rightsquigarrow_w c' \equiv (c, c') \in \text{eval}_w \mid$   
*unannotated*:  $\llbracket (c, \text{mem}) \rightsquigarrow_s (c', \text{mem}') \rrbracket$   
 $\implies (\langle \text{cxt-to-stmt } E \ c, \text{ mds}, \text{ mem} \rangle_w, \langle \text{cxt-to-stmt } E \ c', \text{ mds}, \text{ mem}' \rangle_w) \in \text{eval}_w \mid$   
*seq*:  $\llbracket \langle c_1, \text{ mds}, \text{ mem} \rangle_w \rightsquigarrow_w \langle c_1', \text{ mds}', \text{ mem}' \rangle_w \rrbracket \implies (\langle (c_1 ;; c_2), \text{ mds}, \text{ mem} \rangle_w,$   
 $\langle (c_1' ;; c_2), \text{ mds}', \text{ mem}' \rangle_w) \in \text{eval}_w \mid$   
*decl*:  $\llbracket \langle c, \text{update-modes } \mu \ \text{ mds}, \text{ mem} \rangle_w \rightsquigarrow_w \langle c', \text{ mds}', \text{ mem}' \rangle_w \rrbracket \implies$   
 $(\langle \text{cxt-to-stmt } E \ (\text{ModeDecl } c \ \mu), \text{ mds}, \text{ mem} \rangle_w, \langle \text{cxt-to-stmt } E \ c', \text{ mds}',$   
 $\text{ mem}' \rangle_w) \in \text{eval}_w \mid$

*await*:  $\llbracket \text{eval}_B \ \text{mem } b; \text{ no-await } c_1;$   
 $\langle c_1, \text{ mds}, \text{ mem} \rangle_w, \langle c_2, \text{ mds}', \text{ mem}' \rangle_w) \in \text{eval}_w^+;$   
*is-final*  $c_2 \rrbracket \implies$   
 $(\langle \text{Await } b \ c_1, \text{ mds}, \text{ mem} \rangle_w, \langle c_2, \text{ mds}', \text{ mem}' \rangle_w) \in \text{eval}_w$

**abbreviation** *eval<sub>w</sub>-plus* ::

((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf*  $\Rightarrow$   
 ((*'Var*, *'AExp*, *'BExp*) *Stmt*, *'Var*, *'Val*) *LocalConf*  $\Rightarrow$  *bool* (-  $\rightsquigarrow_w^+$ )

-) **where**  
 $\text{ctx} \rightsquigarrow_w^+ \text{ctx}' \equiv (\text{ctx}, \text{ctx}') \in \text{eval}_w^+$

### 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*:  $((c, \text{ mds}), \text{ mem}), ((c', \text{ mds}'), \text{ mem}') \in \text{eval}_w$

**inductive-cases** *stop-no-eval'* [*elim*]:  $((\text{Stop}, \text{ mem}), (c', \text{ mem}')) \in \text{eval}_w\text{-simple}$

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

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

**lemma** *cxt-inv*:

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

**lemma** *cxt-inv-assign*:

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

**lemma** *cxt-inv-skip*:

$\llbracket \text{cxt-to-stmt } E \ c = \text{Skip} \rrbracket \implies c = \text{Skip} \wedge E = []$   
**by** (*metis Stmt.simps(23) cxt-inv*)

**lemma** *cxt-inv-stop*:

$\text{cxt-to-stmt } E \ c = \text{Stop} \implies c = \text{Stop} \wedge E = []$   
**by** (*metis Stmt.simps(49) cxt-inv*)

**lemma** *cxt-inv-if*:

$cxt\text{-to-stmt } E \ c = \text{If } e \ p \ q \implies c = \text{If } e \ p \ q \wedge E = []$   
**by** (*metis Stmt.simps(43) cxt-inv*)

**lemma** *cxt-inv-anno*:

$cxt\text{-to-stmt } E \ c = c'@[mu] \implies c = c'@[mu] \wedge E = []$   
**using** *cxt-inv* **by** *blast*

**lemma** *cxt-inv-await*:

$cxt\text{-to-stmt } E \ c = \text{Await } e \ p \implies c = \text{Await } e \ p \wedge E = []$   
**by** (*metis Stmt.simps(47) cxt-inv*)

**lemma** *cxt-inv-while*:

$cxt\text{-to-stmt } E \ c = \text{While } e \ p \implies c = \text{While } e \ p \wedge E = []$   
**by** (*metis Stmt.simps(45) cxt-inv*)

**lemma** *skip-elim* [*elim*]:

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

**apply** (*erule eval-elim*)

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

**apply** (*metis Stmt.simps(24)*)

**apply** (*metis Stmt.simps(21) cxt-inv-skip*)

**by** *simp*

**lemma** *assign-elim* [*elim*]:

$\langle x \leftarrow e, mds, mem \rangle_w \rightsquigarrow_w \langle c', mds', mem' \rangle_w \implies c' = \text{Stop} \wedge mds = mds' \wedge mem' = mem \ (x := \text{eval}_A \ mem \ e)$

**apply** (*erule eval-elim*)

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

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

**apply** *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!*]:  $(\text{If } b \ p \ q, mem) \rightsquigarrow_s (c', mem')$

**lemma** *if-elim* [*elim*]:

$\bigwedge P.$

$\llbracket \langle \text{If } b \ p \ q, mds, mem \rangle_w \rightsquigarrow_w \langle c', mds', mem' \rangle_w ;$

$\llbracket c' = p; mem' = mem ; mds' = mds ; \text{eval}_B \ mem \ b \rrbracket \implies P ;$

$\llbracket c' = q; mem' = mem ; mds' = mds ; \neg \text{eval}_B \ mem \ b \rrbracket \implies P \rrbracket \implies P$

**apply** (*erule eval-elim*)

**apply** (*metis (no-types) cxt-inv-if cxt-to-stmt.simps(1) if-elim'*)

**apply** (*metis Stmt.simps(43)*)

**apply** (*metis Stmt.simps(35) cxt-inv-if*)

by *simp*

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

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

**lemma** *while-elim* [*elim*]:

$\llbracket \langle \text{While } e \ c, \ mds, \ mem \rangle_w \rightsquigarrow_w \langle c', \ mds', \ mem' \rangle_w \rrbracket \implies c' = \text{If } e \ (c \ ; \ ; \ \text{While } e \ c) \ \text{Stop} \wedge \ mds' = mds \wedge \ mem' = mem$

**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*]:  $\langle c@[upd], \ mem \rangle \rightsquigarrow_s \langle c', \ mem' \rangle$

**lemma** *upd-elim* [*elim*]:

$\langle c@[upd], \ mds, \ mem \rangle_w \rightsquigarrow_w \langle c', \ mds', \ mem' \rangle_w \implies \langle c, \ \text{update-modes } upd \ mds, \ mem \rangle_w \rightsquigarrow_w \langle c', \ mds', \ mem' \rangle_w$

**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*]:

$c_1 \ ; \ ; \ c_2 = \text{cxt-to-stmt } E \ c \implies (E = [] \wedge c = c_1 \ ; \ ; \ c_2) \vee (\exists \ c' \ cs. E = c' \ \# \ cs \wedge c = c_1 \wedge c_2 = \text{cxt-to-stmt } cs \ c')$

**apply** (*cases E*)

**apply** (*metis cxt-to-stmt.simps(1)*)

**by** (*metis Stmt.simps(3) cxt-to-stmt.simps(2)*)

**inductive-cases** *seq-elim'* [*elim*]:  $\langle c_1 \ ; \ ; \ c_2, \ mem \rangle \rightsquigarrow_s \langle c', \ mem' \rangle$

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

**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*]:

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

**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*:  
 $c ;; \text{cxt-to-stmt } E \ c' = \text{cxt-to-stmt } (c' \# E) \ c$   
**by** (*metis cxt-to-stmt.simps*(2))

**lemma** *seq-elim* [*elim*]:  
 $\llbracket \langle c_1 ;; c_2, \text{mds}, \text{mem} \rangle_w \rightsquigarrow_w \langle c', \text{mds}', \text{mem}' \rangle_w ; c_1 \neq \text{Stop} \rrbracket \implies$   
 $(\exists c_1'. \langle c_1, \text{mds}, \text{mem} \rangle_w \rightsquigarrow_w \langle c_1', \text{mds}', \text{mem}' \rangle_w \wedge c' = c_1' ;; c_2)$   
**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*:  $\text{Stop} = \text{cxt-to-stmt } E \ c \implies c = \text{Stop}$   
**by** (*metis Stmt.simps*(50) *cxt-to-stmt.simps*(1) *cxt-to-stmt.simps*(2) *neg-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*:  $\text{mem}' = \text{mem}$   
**assumes** *upd*:  $\text{mds}' = \text{update-modes } \text{upd } \text{mds}$   
**shows**  $(\langle \text{Skip}@[\text{upd}], \text{mds}, \text{mem} \rangle_w, \langle \text{Stop}, \text{mds}', \text{mem}' \rangle_w) \in \text{eval}_w$   
**using** *assms decl-eval\_w*  
**by** *auto*

**lemma** *assign-eval\_w'*:  
 $\llbracket \text{mds} = \text{mds}'; \text{mem}' = \text{mem}(x := \text{eval}_A \ \text{mem } e) \rrbracket \implies$   
 $\langle x \leftarrow e, \text{mds}, \text{mem} \rangle_w \rightsquigarrow_w \langle \text{Stop}, \text{mds}', \text{mem}' \rangle_w$

**using** *assign-eval<sub>w</sub>*  
**by** *simp*

**lemma** *seq-decl-elim*:

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

**lemma** *seq-assign-elim*:

$\langle\langle x \leftarrow e \rangle\rangle ; c, \text{mds}, \text{mem} \rangle_w \rightsquigarrow_w \langle c', \text{mds}', \text{mem}' \rangle_w \implies$   
 $c' = \text{Stop} ; c \wedge \text{mds}' = \text{mds} \wedge \text{mem}' = \text{mem}(x := \text{eval}_A \text{ mem } e)$   
**apply** (*drule seq-elim, simp*)  
**apply** (*erule exE, clarsimp*)  
**apply** (*drule assign-elim, clarsimp*)  
**done**

**lemma** *no-await-trans*:

$\llbracket \text{no-await } c ; \langle c, \text{mds}, \text{mem} \rangle_w \rightsquigarrow_w \langle c', \text{mds}', \text{mem}' \rangle_w \rrbracket \implies \text{no-await } c'$   
**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]*:  $\llbracket \text{no-await } c \rrbracket \implies c \neq \text{Await } b \text{ } c'$

**using** *no-await.cases Stmt.distinct by fast*

**lemma** *no-await-trancl-impl*:

$\llbracket \text{ctx} \rightsquigarrow_w^+ \text{ctx}' \rrbracket \implies \text{no-await (fst (fst ctx))} \longrightarrow \text{no-await (fst (fst ctx'))}$   
**apply** (*erule trancl.induct, clarsimp*)  
**using** *no-await-trans apply blast*

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

**lemma** *no-await-trancl*:  
 $\llbracket \langle ctx \rightsquigarrow_w^+ ctx' ; no\text{-}await (fst (fst ctx)) \rrbracket \implies no\text{-}await (fst (fst ctx'))$   
**using** *no-await-trancl-impl* **by** *blast*

**lemma** *await-elim*:  
 $\llbracket \langle Await\ b\ c_1, mds, mem \rangle_w \rightsquigarrow_w \langle c_2, mds', mem \rangle_w \rrbracket \implies$   
 $eval_B\ mem\ b \wedge no\text{-}await\ c_1 \wedge is\text{-}final\ c_2 \wedge$   
 $\langle c_1, mds, mem \rangle_w \rightsquigarrow_w^+ \langle c_2, mds', mem \rangle_w$   
**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  $\Gamma$  to track all stable (i.e. *AsmNoWrite* or *AsmNoReadOrWrite*),

non- $\mathcal{C}$  variables.

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

We decouple the requirement for inclusion in  $\Gamma$  from a variable's classification so that we don't need to be updating  $\Gamma$  each time we alter a control variable. Even if we tried to keep  $\Gamma$  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*  $\rightarrow$  *'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  $\Gamma$  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  $\Gamma$  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  $\Gamma$  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<sub>B</sub> mem p*)

**end**

```

locale sifum-types =
  sifum-lang-no-dma evA evB aexp-vars bexp-vars + sifum-security dma C-vars C
  evalw undefined
  for evA :: ('Var, 'Val) Mem ⇒ 'AExp ⇒ 'Val
  and evB :: ('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 C-vars :: 'Var ⇒ 'Var set
  and C :: 'Var set +

  fixes bexp-neg :: 'BExp ⇒ 'BExp
  assumes bexp-neg-negates:  $\bigwedge mem\ e. (ev_B\ mem\ (bexp-neg\ e)) = (\neg (ev_B\ mem\ e))$ 

  fixes assign-post :: 'BExp preds ⇒ 'Var ⇒ 'AExp ⇒ 'BExp preds
  assumes assign-post-valid:  $\bigwedge mem. pred\ P\ mem \implies pred (assign-post\ P\ x\ e)$ 
  (mem(x := evA mem e))
  fixes dma-type :: 'Var ⇒ 'BExp set
  assumes dma-correct:
    dma mem x = (if ( $\forall e \in dma-type\ x. ev_B\ mem\ e$ ) then Low else High)
  assumes C-vars-correct:
    C-vars x = ( $\bigcup (bexp-vars\ 'dma-type\ x)$ )
  fixes pred-False :: 'BExp
  assumes pred-False-is-False:  $\neg ev_B\ mem\ pred-False$ 
  assumes bexp-vars-pred-False: bexp-vars pred-False = {}

```

```

locale sifum-types-assign =
  sifum-lang-no-dma evA evB aexp-vars bexp-vars + sifum-security dma C-vars C
  evalw undefined
  for evA :: ('Var, 'Val) Mem ⇒ 'AExp ⇒ 'Val
  and evB :: ('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 C-vars :: 'Var ⇒ 'Var set
  and C :: 'Var set +

  fixes bexp-neg :: 'BExp ⇒ 'BExp
  assumes bexp-neg-negates:  $\bigwedge mem\ e. (ev_B\ mem\ (bexp-neg\ e)) = (\neg (ev_B\ mem\ e))$ 
  fixes dma-type :: 'Var ⇒ 'BExp set
  assumes dma-correct:
    dma mem x = (if ( $\forall e \in dma-type\ x. ev_B\ mem\ e$ ) then Low else High)
  assumes C-vars-correct:
    C-vars x = ( $\bigcup (bexp-vars\ 'dma-type\ x)$ )
  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**

$P +_S e \equiv (if\ (\forall x \in bexp-vars\ e. stable\ \mathcal{S}\ x)\ then\ P \cup \{e\}\ else\ P)$

**lemma** *add-pred-subset*:

$P \subseteq P +_S p$

**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  $\subseteq$  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-detB
    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-detA
    apply fastforce
    using eval-vars-detB
    apply (metis (mono-tags, lifting) ComplD fun-upd-other mem-Collect-eq sin-
gletonI subset-eq)
    using dma-correct apply blast
    using C-vars-correct pred-False-is-False bexp-vars-pred-False apply blast+
  done
```

```
context sifum-types
begin
```

**abbreviation**

```
mm-equiv-abv2 :: (-, -, -) LocalConf  $\Rightarrow$  (-, -, -) LocalConf  $\Rightarrow$  bool
(infix  $\approx$  60)
```

**where**

```
mm-equiv-abv2 c c'  $\equiv$  mm-equiv-abv c c'
```

**abbreviation**

```
eval-abv2 :: (-, 'Var, 'Val) LocalConf  $\Rightarrow$  (-, -, -) LocalConf  $\Rightarrow$  bool
(infixl  $\rightsquigarrow$  70)
```

**where**

```
x  $\rightsquigarrow$  y  $\equiv$  (x, y)  $\in$  evalw
```

**abbreviation**

```
eval-plus-abv :: (-, 'Var, 'Val) LocalConf  $\Rightarrow$  (-, -, -) LocalConf  $\Rightarrow$  bool
(infixl  $\rightsquigarrow^+$  70)
```

**where**

```
x  $\rightsquigarrow^+$  y  $\equiv$  (x, y)  $\in$  evalw+
```

**abbreviation**

```
no-eval-abv :: (-, 'Var, 'Val) LocalConf  $\Rightarrow$  bool
(-  $\rightsquigarrow$   $\perp$ )
```

**where**

```
x  $\rightsquigarrow$   $\perp$   $\equiv$   $\forall$  y. (x, y)  $\notin$  evalw
```

**abbreviation**

$low\text{-indistinguishable}\text{-abv} :: 'Var\ Mds \Rightarrow ('Var, 'AExp, 'BExp)\ Stmt \Rightarrow (-, -, -)$   
 $Stmt \Rightarrow bool$   
 $(- \sim_1 - [100, 100] 80)$

**where**

$c \sim_{mds} c' \equiv low\text{-indistinguishable}\ mds\ c\ c'$

**abbreviation**

$vars\text{-of}\text{-type} :: 'BExp\ Type \Rightarrow 'Var\ set$

**where**

$vars\text{-of}\text{-type}\ t \equiv \bigcup (bexp\text{-vars}\ 't)$

**definition**

$type\text{-wellformed} :: 'BExp\ Type \Rightarrow bool$

**where**

$type\text{-wellformed}\ t \equiv vars\text{-of}\text{-type}\ t \subseteq \mathcal{C}$

**lemma** *dma-type-wellformed* [simp]:

$type\text{-wellformed}\ (dma\text{-type}\ x)$

**apply**(*clarsimp simp: type-wellformed-def C-def | safe*)**+**

**using** *C-vars-correct* **apply** *blast*

**done**

**definition**

$to\text{-total} :: ('Var, 'BExp)\ TyEnv \Rightarrow 'Var \Rightarrow 'BExp\ Type$

**where**

$to\text{-total}\ \Gamma \equiv \lambda v. \text{if } v \in \text{dom } \Gamma \text{ then the } (\Gamma\ v) \text{ else } dma\text{-type}\ v$

**definition**

$types\text{-wellformed} :: ('Var, 'BExp)\ TyEnv \Rightarrow bool$

**where**

$types\text{-wellformed}\ \Gamma \equiv \forall x \in \text{dom } \Gamma. type\text{-wellformed}\ (\text{the } (\Gamma\ x))$

**lemma** *to-total-type-wellformed*:

$types\text{-wellformed}\ \Gamma \Longrightarrow$

$type\text{-wellformed}\ (to\text{-total}\ \Gamma\ x)$

**by**(*auto simp: to-total-def types-wellformed-def*)

**lemma** *Un-type-wellformed*:

$\forall t \in ts. type\text{-wellformed}\ t \Longrightarrow type\text{-wellformed}\ (\bigcup\ ts)$

**apply**(*clarsimp simp: type-wellformed-def | safe*)**+**

**by**(*fastforce simp: C-def elim!: subsetCE*)

**inductive**

$type\text{-aexpr} :: ('Var, 'BExp)\ TyEnv \Rightarrow 'AExp \Rightarrow 'BExp\ Type \Rightarrow bool\ (- \vdash_a - \in -)$   
 $[120, 120, 120] 1000)$

**where**

$type\text{-aexpr}\ [intro!]: \Gamma \vdash_a e \in \bigcup (\text{image } (\lambda x. to\text{-total}\ \Gamma\ x) (aexp\text{-vars}\ e))$



**lemma** *type-aexprI*:  
 $t = \bigcup (\text{image } (\lambda x. \text{to-total } \Gamma x) (\text{aexp-vars } e)) \implies \Gamma \vdash_a e \in t$   
**apply**(*erule ssubst*)  
**apply**(*rule type-aexpr.intros*)  
**done**

**lemma** *type-aexpr-type-wellformed*:  
 $\text{types-wellformed } \Gamma \implies \Gamma \vdash_a e \in t \implies \text{type-wellformed } t$   
**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*]:  $\Gamma \vdash_a e \in t$

**inductive**

*type-bexpr* :: (*'Var, 'BExp*) *TyEnv*  $\Rightarrow$  *'BExp*  $\Rightarrow$  *'BExp Type*  $\Rightarrow$  *bool* ( $- \vdash_b - \in -$   
[120, 120, 120] 1000)

**where**

*type-bexpr* [*intro!*]:  $\Gamma \vdash_b e \in \bigcup (\text{image } (\lambda x. \text{to-total } \Gamma x) (\text{bexp-vars } e))$

**lemma** *type-bexprI*:  
 $t = \bigcup (\text{image } (\lambda x. \text{to-total } \Gamma x) (\text{bexp-vars } e)) \implies \Gamma \vdash_b e \in t$   
**apply**(*erule ssubst*)  
**apply**(*rule type-bexpr.intros*)  
**done**

**lemma** *type-bexpr-type-wellformed*:  
 $\text{types-wellformed } \Gamma \implies \Gamma \vdash_b e \in t \implies \text{type-wellformed } t$   
**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*]:  $\Gamma \vdash_b e \in t$

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**

$pred\text{-}stable :: 'Var \text{ Stable} \Rightarrow 'BExp \Rightarrow bool$

**where**

$pred\text{-}stable \mathcal{S} p \equiv \forall x \in bexp\text{-}vars \ p. \ stable \ \mathcal{S} \ x$

**abbreviation**

$type\text{-}stable :: 'Var \ \text{Stable} \Rightarrow 'BExp \ \text{Type} \Rightarrow bool$

**where**

$type\text{-}stable \ \mathcal{S} \ t \equiv (\forall p \in t. \ pred\text{-}stable \ \mathcal{S} \ p)$

**lemma** *type-stable-is-sufficient*:

$\llbracket type\text{-}stable \ \mathcal{S} \ t \rrbracket \Longrightarrow$

$\forall mem \ mem'. (\forall x. \ stable \ \mathcal{S} \ x \longrightarrow mem \ x = mem' \ x) \longrightarrow (ev_B \ mem) \ 't = (ev_B \ mem') \ 't$

**apply** (*clarsimp simp: type-wellformed-def image-def*)

**apply** *safe*

**using** *eval-vars-det<sub>B</sub> apply blast+*

**done**

**definition**

$mds\text{-}consistent :: 'Var \ Mds \Rightarrow ('Var, 'BExp) \ TyEnv \Rightarrow 'Var \ \text{Stable} \Rightarrow 'BExp \ preds \Rightarrow bool$

**where**

$mds\text{-}consistent \ mds \ \Gamma \ \mathcal{S} \ P \equiv$   
 $(\mathcal{S} = (mds \ AsmNoWrite, mds \ AsmNoReadOrWrite)) \wedge$   
 $(dom \ \Gamma = \{x. \ x \notin \mathcal{C} \wedge \ stable \ \mathcal{S} \ x\}) \wedge$   
 $(\forall p \in P. \ pred\text{-}stable \ \mathcal{S} \ p)$

**fun**

$add\text{-}anno\text{-}dom :: ('Var, 'BExp) \ TyEnv \Rightarrow 'Var \ \text{Stable} \Rightarrow 'Var \ \text{ModeUpd} \Rightarrow 'Var \ \text{set}$

**where**

$add\text{-}anno\text{-}dom \ \Gamma \ \mathcal{S} \ (Acq \ v \ AsmNoReadOrWrite) = (if \ v \notin \mathcal{C} \ \text{then} \ dom \ \Gamma \cup \{v\} \ \text{else} \ dom \ \Gamma) \ |$

$add\text{-}anno\text{-}dom \ \Gamma \ \mathcal{S} \ (Acq \ v \ AsmNoWrite) = (if \ v \notin \mathcal{C} \ \text{then} \ dom \ \Gamma \cup \{v\} \ \text{else} \ dom \ \Gamma) \ |$

$add\text{-}anno\text{-}dom \ \Gamma \ \mathcal{S} \ (Acq \ v \ -) = dom \ \Gamma \ |$

$add\text{-}anno\text{-}dom \ \Gamma \ \mathcal{S} \ (Rel \ v \ AsmNoReadOrWrite) = (if \ v \notin \text{fst} \ \mathcal{S} \ \text{then} \ dom \ \Gamma - \{v\} \ \text{else} \ dom \ \Gamma) \ |$

$add\text{-}anno\text{-}dom \ \Gamma \ \mathcal{S} \ (Rel \ v \ AsmNoWrite) = (if \ v \notin \text{snd} \ \mathcal{S} \ \text{then} \ dom \ \Gamma - \{v\} \ \text{else} \ dom \ \Gamma) \ |$

$add\text{-}anno\text{-}dom \ \Gamma \ \mathcal{S} \ (Rel \ v \ -) = dom \ \Gamma$

**definition**

$add\text{-}anno :: ('Var, 'BExp) \ TyEnv \Rightarrow 'Var \ \text{Stable} \Rightarrow 'Var \ \text{ModeUpd} \Rightarrow ('Var, 'BExp) \ TyEnv \ (- \oplus - \ [120, 120, 120] \ 1000)$

**where**

$\Gamma \oplus_{\mathcal{S}} \ upd = restrict\text{-}map \ (\lambda x. \ Some \ (to\text{-}total \ \Gamma \ x)) \ (add\text{-}anno\text{-}dom \ \Gamma \ \mathcal{S} \ upd)$

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

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

**lemma** *add-anno-acq-AsmNoReadOrWrite* [*simp*]:  
**assumes** *notin* [*simp*]:  $v \notin \text{dom } \Gamma$   
**shows**  $v \notin \mathcal{C} \implies \Gamma \oplus_{\mathcal{S}} (\text{Acq } v \text{ AsmNoReadOrWrite}) = (\Gamma(v \mapsto \text{dma-type } v))$   
**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-AsmNoReadOrWrite* [*simp*]:  
**assumes** *isin* [*simp*]:  $v \in \text{dom } \Gamma$   
**shows**  $v \notin \text{fst } \mathcal{S} \implies \Gamma \oplus_{\mathcal{S}} (\text{Rel } v \text{ AsmNoReadOrWrite}) = \text{restrict-map } \Gamma ((\text{dom } \Gamma) - \{v\})$   
**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*]:  
 $v \in \text{dom } \Gamma \vee v \in \mathcal{C} \implies \Gamma \oplus_{\mathcal{S}} (\text{Acq } v \text{ AsmNoWrite}) = \Gamma$   
**apply**(*safe* | *clarsimp simp: add-anno-def to-total-def*) +

```

apply(rule ext)
apply(clarsimp simp: restrict-map-def | safe)+
apply(case-tac  $\Gamma$  x, fastforce+)[5]
apply(rule ext)
apply(clarsimp simp: restrict-map-def | safe)+
apply(case-tac  $\Gamma$  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  $\Gamma$  x, fastforce+)
done

```

```

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

```

```

lemma add-anno-acq-AsmNoWrite [simp]:
  assumes notin [simp]:  $v \notin \text{dom } \Gamma$ 
  shows  $v \notin \mathcal{C} \implies \Gamma \oplus_{\mathcal{S}} (\text{Acq } v \text{ AsmNoWrite}) = (\Gamma(v \mapsto \text{dma-type } v))$ 
  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]:  $v \in \text{dom } \Gamma$ 
  shows  $v \notin \text{snd } \mathcal{S} \implies \Gamma \oplus_{\mathcal{S}} (\text{Rel } v \text{ AsmNoWrite}) = \text{restrict-map } \Gamma ((\text{dom } \Gamma) - \{v\})$ 
  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

```

```

fun
  add-anno-stable :: 'Var Stable  $\Rightarrow$  'Var ModeUpd  $\Rightarrow$  'Var Stable
where
  add-anno-stable  $\mathcal{S}$  (Acq v AsmNoReadOrWrite) = (fst  $\mathcal{S}$ , snd  $\mathcal{S} \cup \{v\}$ ) |
  add-anno-stable  $\mathcal{S}$  (Acq v AsmNoWrite) = (fst  $\mathcal{S} \cup \{v\}$ , snd  $\mathcal{S}$ ) |
  add-anno-stable  $\mathcal{S}$  (Acq v -) =  $\mathcal{S}$  |

```

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

**definition**

$pred\text{-}entailment :: 'BExp\ preds \Rightarrow 'BExp\ preds \Rightarrow bool$  (**infix**  $\vdash$  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**

$subtype :: 'BExp\ Type \Rightarrow 'BExp\ preds \Rightarrow 'BExp\ Type \Rightarrow bool$  ( $- \leq :-$   $-$  [120, 120, 120] 1000)

**where**

$t \leq :_P t' \equiv (P \cup t') \vdash t$

**definition**

$type\text{-}max :: 'BExp\ Type \Rightarrow ('Var, 'Val)\ Mem \Rightarrow Sec$

**where**

$type\text{-}max\ t\ mem \equiv if\ (\forall p \in t. ev_B\ mem\ p)\ then\ Low\ else\ High$

**lemma** *type-stable-is-sufficient'*:

$\llbracket type\text{-}stable\ \mathcal{S}\ t \rrbracket \Longrightarrow$   
 $\forall mem\ mem'. (\forall x. stable\ \mathcal{S}\ x \longrightarrow mem\ x = mem'\ x) \longrightarrow type\text{-}max\ t\ mem =$   
 $type\text{-}max\ t\ mem'$   
**using** *type-stable-is-sufficient*  
**unfolding** *type-max-def image-def*  
**by** (*metis (no-types, lifting) eval-vars-det<sub>B</sub>*)

**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**  $a: \bigwedge mem. pred\ P\ mem \Longrightarrow type\text{-}max\ t\ mem \leq type\text{-}max\ t'\ mem$

**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**  $tmax: type\text{-}max\ t\ mem \leq type\text{-}max\ t'\ mem$  **by** *blast*

**from**  $p$  **have**  $t': pred\ t'\ mem$

**unfolding** *pred-def* **by** *blast*

**from**  $t'$  **have**  $type\text{-}max\ t'\ mem = Low$

**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**  
 $anno-type-stable\ \Gamma\ \mathcal{S}\ upd \equiv (case\ upd\ of\ (Rel\ v\ m) \Rightarrow$   
 $(v \in \mathcal{C} \wedge v \notin add-anno-dom\ \Gamma\ \mathcal{S}\ upd) \longrightarrow$   
 $(\forall x \in dom\ \Gamma. v \notin vars-of-type\ (the\ (\Gamma\ x)))$   
 $\quad | (Acq\ v\ m) \Rightarrow$   
 $(v \notin \mathcal{C} \wedge v \in add-anno-dom\ \Gamma\ \mathcal{S}\ upd - dom\ \Gamma) \longrightarrow$   
 $(\forall x \in \mathcal{C}\ vars\ v. stable\ \mathcal{S}\ x))$

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

**definition**

$$\text{types-stable} :: ('Var, 'BExp) TyEnv \Rightarrow 'Var Stable \Rightarrow bool$$
**where**

$$\text{types-stable } \Gamma \mathcal{S} \equiv \forall x \in \text{dom } \Gamma. \text{ type-stable } \mathcal{S} (\text{the } (\Gamma x))$$
**definition**

$$\text{tyenv-wellformed} :: 'Var Mds \Rightarrow ('Var, 'BExp) TyEnv \Rightarrow 'Var Stable \Rightarrow 'BExp \text{ preds} \Rightarrow bool$$
**where**

$$\begin{aligned} \text{tyenv-wellformed mds } \Gamma \mathcal{S} P &\equiv \\ \text{mds-consistent mds } \Gamma \mathcal{S} P &\wedge \\ \text{types-wellformed } \Gamma \wedge \text{types-stable } \Gamma \mathcal{S} & \end{aligned}$$
**lemma subset-entailment:**

$$P' \subseteq P \Longrightarrow P \vdash P'$$

$$\text{apply}(\text{auto simp: pred-entailment-def pred-def})$$
**done****lemma pred-entailment-refl [simp]:**

$$P \vdash P$$

$$\text{by}(\text{simp add: pred-entailment-def})$$
**lemma pred-entailment-mono:**

$$P \vdash P' \Longrightarrow P \subseteq P'' \Longrightarrow P'' \vdash P'$$

$$\text{by}(\text{auto simp: pred-entailment-def pred-def})$$
**lemma type-equiv-subset:**

$$\text{type-equiv } t P t' \Longrightarrow P \subseteq P' \Longrightarrow \text{type-equiv } t P' t'$$

$$\text{apply}(\text{auto simp: type-equiv-def subtype-def intro: pred-entailment-mono})$$
**done****definition**

$$\text{context-equiv} :: ('Var, 'BExp) TyEnv \Rightarrow 'BExp \text{ preds} \Rightarrow ('Var, 'BExp) TyEnv \Rightarrow bool \text{ (- =: - [120, 120, 120] 1000)}$$
**where**

$$\begin{aligned} \Gamma =:_{P} \Gamma' &\equiv \text{dom } \Gamma = \text{dom } \Gamma' \wedge \\ &(\forall x \in \text{dom } \Gamma'. \text{ type-equiv } (\text{the } (\Gamma x)) P (\text{the } (\Gamma' x))) \end{aligned}$$
**lemma context-equiv-refl[simp]:**

$$\text{context-equiv } \Gamma P \Gamma$$

$$\text{by}(\text{simp add: context-equiv-def})$$
**lemma context-equiv-subset:**

$$\text{context-equiv } \Gamma P \Gamma' \Longrightarrow P \subseteq P' \Longrightarrow \text{context-equiv } \Gamma P' \Gamma'$$

$$\text{apply}(\text{auto simp: context-equiv-def intro: type-equiv-subset})$$
**done****lemma pred-entailment-trans:**

$$P \vdash P' \Longrightarrow P' \vdash P'' \Longrightarrow P \vdash P''$$

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

**lemma** *pred-un [simp]*:

$\text{pred } (P \cup P') \text{ mem} = (\text{pred } P \text{ mem} \wedge \text{pred } P' \text{ mem})$

**apply**(*auto simp: pred-def*)

**done**

**lemma** *pred-entailment-un*:

$P \vdash P' \implies P \vdash P'' \implies P \vdash (P' \cup P'')$

**apply**(*subst pred-entailment-def*)

**apply** *clarsimp*

**apply**(*fastforce simp: pred-entailment-def*)

**done**

**lemma** *pred-entailment-mono-un*:

$P \vdash P' \implies (P \cup P'') \vdash (P' \cup P'')$

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

**done**

**lemma** *subtype-trans*:

$t \leq_P t' \implies t' \leq_{P'} t'' \implies P \vdash P' \implies t \leq_P t''$

$t \leq_{P'} t' \implies t' \leq_P t'' \implies P \vdash P' \implies t \leq_P t''$

**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*:

$\text{type-equiv } t P t' \implies \text{type-equiv } t' P' t'' \implies P \vdash P' \implies \text{type-equiv } t P t''$

**apply**(*auto simp: type-equiv-def intro: subtype-trans*)

**done**

**lemma** *context-equiv-trans*:

$\text{context-equiv } \Gamma P \Gamma' \implies \text{context-equiv } \Gamma' P' \Gamma'' \implies P \vdash P' \implies \text{context-equiv } \Gamma P \Gamma''$



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

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

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

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

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

**inductive**

*has-type* :: ('Var, 'BExp) TyEnv  $\Rightarrow$  'Var Stable  $\Rightarrow$  'BExp preds  $\Rightarrow$  ('Var, 'AExp, 'BExp) Stmt  $\Rightarrow$  ('Var, 'BExp) TyEnv  $\Rightarrow$  'Var Stable  $\Rightarrow$  'BExp preds  $\Rightarrow$  bool  
 $(\vdash \ - \ , \ - \ \{-\} \ - \ , \ - \ [120, 120, 120, 120, 120, 120, 120, 120] \ 1000)$

**where**

*stop-type* [*intro*]:  $\vdash \Gamma, \mathcal{S}, P \ \{\text{Stop}\} \ \Gamma, \mathcal{S}, P \mid$

*skip-type* [*intro*]:  $\vdash \Gamma, \mathcal{S}, P \ \{\text{Skip}\} \ \Gamma, \mathcal{S}, P \mid$

*assign<sub>C</sub>* :

$\llbracket x \in \mathcal{C}; \Gamma \vdash_a e \in t; P \vdash t; (\forall v \in \text{dom } \Gamma. x \notin \text{vars-of-type } (\text{the } (\Gamma \ v))) \rrbracket$ ;

$P' = \text{restrict-preds-to-vars } (\text{assign-post } P \ x \ e) \ \{v. \text{stable } \mathcal{S} \ v\}$ ;

$\forall v. x \in \mathcal{C}\text{-vars } v \wedge v \notin \text{snd } \mathcal{S} \longrightarrow P \vdash (\text{to-total } \Gamma \ v) \wedge$

$(\text{to-total } \Gamma \ v) \leq_{:P'} (\text{dma-type } v) \rrbracket \implies$

$\vdash \Gamma, \mathcal{S}, P \ \{x \leftarrow e\} \ \Gamma, \mathcal{S}, P' \mid$

*assign<sub>1</sub>* :

$\llbracket x \notin \text{dom } \Gamma; x \notin \mathcal{C}; \Gamma \vdash_a e \in t; t \leq_{:P} (\text{dma-type } x) \rrbracket$ ;

$P' = \text{restrict-preds-to-vars } (\text{assign-post } P \ x \ e) \ \{v. \text{stable } \mathcal{S} \ v\} \rrbracket \implies$

$\vdash \Gamma, \mathcal{S}, P \ \{x \leftarrow e\} \ \Gamma, \mathcal{S}, P' \mid$

*assign<sub>2</sub>* :

$\llbracket x \in \text{dom } \Gamma; \Gamma \vdash_a e \in t; \text{type-stable } \mathcal{S} \ t; P' = \text{restrict-preds-to-vars } (\text{assign-post } P \ x \ e) \ \{v. \text{stable } \mathcal{S} \ v\} \rrbracket$ ;

$x \notin \text{snd } \mathcal{S} \longrightarrow t \leq_{:P'} (\text{dma-type } x) \rrbracket \implies$

*has-type*  $\Gamma \ \mathcal{S} \ P \ (x \leftarrow e) \ (\Gamma \ (x := \text{Some } t)) \ \mathcal{S} \ P' \mid$

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

**lemma** *sub'*:

$\llbracket \text{context-equiv } \Gamma_2 \ P_2 \ \Gamma_1;$   
 $(\forall \text{ mds. } \text{tyenv-wellformed mds } \Gamma_2 \ \mathcal{S} \ P_2 \longrightarrow \text{tyenv-wellformed mds } \Gamma_1 \ \mathcal{S} \ P_1);$   
 $(\forall \text{ mds. } \text{tyenv-wellformed mds } \Gamma_1' \ \mathcal{S}' \ P_1' \longrightarrow \text{tyenv-wellformed mds } \Gamma_2' \ \mathcal{S}' \ P_2');$   
*context-equiv*  $\Gamma_1' \ P_1' \ \Gamma_2';$   
 $P_2 \vdash P_1;$   
 $P_1' \vdash P_2';$   
 $\vdash \Gamma_1, \mathcal{S}, P_1 \{ c \} \Gamma_1', \mathcal{S}', P_1' \rrbracket \Longrightarrow$   
 $\vdash \Gamma_2, \mathcal{S}, P_2 \{ c \} \Gamma_2', \mathcal{S}', P_2'$   
**by**(*rule sub*)

**lemma** *assign<sub>2</sub>-helper*:

$\llbracket \Gamma \ x = \text{Some } t; \text{has-type } \Gamma \ \mathcal{S} \ P \ (x \leftarrow e) \ (\Gamma(x \mapsto t)) \ \mathcal{S} \ P' \rrbracket \Longrightarrow \text{has-type } \Gamma \ \mathcal{S} \ P$   
 $(x \leftarrow e) \ \Gamma \ \mathcal{S} \ P'$   
**by** (*simp add:map-upd-triv*)

**lemma** *conc'*:

$\llbracket \vdash \Gamma_1, \mathcal{S}, P \{ c \} \Gamma', \mathcal{S}', P';$   
 $\Gamma_1 = (\Gamma_2(x \mapsto t));$   
 $x \in \text{dom } \Gamma_2;$   
*type-equiv* (*the*  $(\Gamma_2 \ x)$ )  $P \ t;$

```

    type-wellformed t;
    type-stable S t ] =>
  ⊢ Γ2, S, P { c } Γ', 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*:  
 $tyenv\text{-wellformed}\ mds\ \Gamma\ S\ P \implies P' \subseteq P \implies tyenv\text{-wellformed}\ mds\ \Gamma\ S\ P'$   
**apply**(auto simp: tyenv-wellformed-def mds-consistent-def)  
**done**

**lemma** *if-type'*:  
 $\llbracket \Gamma \vdash_b e \in t; P \vdash t; \vdash \Gamma, S, (P +_S e) \{ c_1 \} \Gamma', S', P'; \vdash \Gamma, S, (P +_S (bexp\text{-neg}\ e)) \{ c_2 \} \Gamma', S', P''; P''' \subseteq P' \cap P'' \rrbracket \implies \vdash \Gamma, S, P \{ If\ e\ c_1\ c_2 \} \Gamma', S', P'''$   
**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'*:  
 $\llbracket \Gamma = \Gamma'; S = S'; P = P' \rrbracket \implies \vdash \Gamma, S, P \{ Skip \} \Gamma', S', P'$   
**using** skip-type **by** simp

Some helper lemmas to discharge the assumption of the  $\llbracket ?\Gamma' = ?\Gamma \oplus ?S\ ?upd; ?S' = add\text{-anno}\text{-stable}\ ?S\ ?upd; ?P' = ?P \mid' \{ v.\ local.\ stable\ ?S'\ v \}; \vdash ?\Gamma', ?S', ?P' \{ ?c \} ?\Gamma'', ?S'', ?P''; ?c \neq Stop; \bigwedge x.\ to\text{-total}\ ?\Gamma\ x \leq ?P' \ to\text{-total}\ ?\Gamma'\ x; anno\text{-type}\text{-stable}\ ?\Gamma\ ?S\ ?upd; anno\text{-type}\text{-sec}\ ?\Gamma\ ?S\ ?P\ ?upd \rrbracket \implies \vdash ?\Gamma, ?S, ?P \{ ?c @ [ ?upd ] \} ?\Gamma'', ?S'', ?P''$  rule.

**lemma** *anno-type-helpers* [simp]:  
 $(to\text{-total}\ \Gamma\ x) \leq_P (to\text{-total}\ (add\text{-anno}\ \Gamma\ S\ (buffer\ +=_m\ AsmNoWrite))\ x)$   
 $(to\text{-total}\ \Gamma\ x) \leq_P (to\text{-total}\ (add\text{-anno}\ \Gamma\ S\ (buffer\ +=_m\ AsmNoReadOrWrite))\ x)$   
**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* ::  $('Var, 'BExp) TyEnv \Rightarrow ('Var, 'Val) Mem \Rightarrow ('Var, 'Val) Mem \Rightarrow bool$

(**infix** =<sub>1</sub> 60)

**where**  $mem_1 =_{\Gamma} mem_2 \equiv \forall x. (type-max (to-total \Gamma x) mem_1 = Low \longrightarrow mem_1 x = mem_2 x)$

**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'*:

$dom \Gamma \cap \mathcal{C} = \{\} \Longrightarrow types-wellformed \Gamma \Longrightarrow mem_1 =_{\Gamma} mem_2 \Longrightarrow mem_2 =_{\Gamma} mem_1$

**proof**(*clarsimp simp: tyenv-eq-def*)

**fix** *x*

**assume** *a*:  $\forall x. type-max (to-total \Gamma x) mem_1 = Low \longrightarrow mem_1 x = mem_2 x$

**assume** *b*:  $dom \Gamma \cap \mathcal{C} = \{\}$

**from** *a b* **have** *eq-C*:  $\forall x \in \mathcal{C}. mem_1 x = mem_2 x$

**by** (*fastforce simp: to-total-def C-Low type-max-dma-type split: if-splits*)

**hence**  $dma mem_1 = dma mem_2$

**by** (*rule dma-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*  $\Gamma$

**assume** *d*:  $type-max (to-total \Gamma x) mem_2 = Low$

**show**  $mem_2 x = mem_1 x$

**proof**(*cases x \in dom \Gamma*)

**assume** *in-dom*:  $x \in dom \Gamma$

**from** *this* **obtain** *t* **where**  $t: \Gamma x = Some t$  **by** *blast*

**from** *this in-dom c* **have** *type-wellformed t* **by** (*force simp: types-wellformed-def*)

**hence**  $\forall x \in vars-of-type t. mem_1 x = mem_2 x$

**using** *eq-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-vars-det<sub>B</sub>*

**by** *fastforce*

**with** *in-dom t* **have**  $to-total \Gamma x = t$

**by** (*auto simp: to-total-def*)

**with**  $t\text{-eq}$  **have**  $\text{type-max } (to\text{-total } \Gamma \ x) \ mem_2 = \text{type-max } (to\text{-total } \Gamma \ x) \ mem_1$   
**by**  $\text{simp}$   
**with**  $d$  **have**  $\text{type-max } (to\text{-total } \Gamma \ x) \ mem_1 = Low$  **by**  $\text{simp}$   
**with**  $a$  **show**  $?thesis$  **by**  $(metis \ sym)$   
**next**  
**assume**  $x \notin dom \ \Gamma$   
**hence**  $to\text{-total } \Gamma \ x = dma\text{-type } x$   
**by**  $(auto \ simp: \ to\text{-total}\text{-def})$   
**with**  $dma\text{-type}\text{-eq}$  **have**  $\text{type-max } (to\text{-total } \Gamma \ x) \ mem_2 = \text{type-max } (to\text{-total } \Gamma \ x) \ mem_1$  **by**  $\text{simp}$   
**with**  $d$  **have**  $\text{type-max } (to\text{-total } \Gamma \ x) \ mem_1 = Low$  **by**  $\text{simp}$   
**with**  $a$  **show**  $?thesis$  **by**  $(metis \ sym)$   
**qed**  
**qed**

**lemma**  $tyenv\text{-eq}\text{-sym}$ :

$tyenv\text{-wellformed } mds \ \Gamma \ \mathcal{S} \ P \implies mem_1 =_{\Gamma} mem_2 \implies mem_2 =_{\Gamma} mem_1$   
**apply** $(rule \ tyenv\text{-eq}\text{-sym})$   
**apply** $(fastforce \ simp: \ tyenv\text{-wellformed}\text{-def } mds\text{-consistent}\text{-def})$   
**apply** $(simp \ add: \ tyenv\text{-wellformed}\text{-def})$   
**by**  $assumption$

**inductive-set**  $\mathcal{R}_1 :: ('Var, 'BExp) \ TyEnv \Rightarrow 'Var \ Stable \Rightarrow 'BExp \ preds \Rightarrow (('Var, 'AExp, 'BExp) \ Stmt, 'Var, 'Val) \ LocalConf \ rel$

**and**  $\mathcal{R}_1\text{-abv} ::$   
 $(('Var, 'AExp, 'BExp) \ Stmt, 'Var, 'Val) \ LocalConf \Rightarrow$   
 $('Var, 'BExp) \ TyEnv \Rightarrow 'Var \ Stable \Rightarrow 'BExp \ preds \Rightarrow$   
 $(('Var, 'AExp, 'BExp) \ Stmt, 'Var, 'Val) \ LocalConf \Rightarrow$   
 $bool \ (- \ \mathcal{R}_1 \text{-,,-} \ - \ [120, 120, 120, 120, 120] \ 1000)$   
**for**  $\Gamma' :: ('Var, 'BExp) \ TyEnv$   
**and**  $S' :: 'Var \ Stable$   
**and**  $P' :: 'BExp \ preds$

**where**

$x \ \mathcal{R}_1^1_{\Gamma, \mathcal{S}, P} \ y \equiv (x, y) \in \mathcal{R}_1 \ \Gamma \ \mathcal{S} \ P \mid$   
 $intro \ [intro!] : \llbracket \vdash \ \Gamma, \mathcal{S}, P \ \{ \ c \} \ \Gamma', S', P' ; \ tyenv\text{-wellformed } mds \ \Gamma \ \mathcal{S} \ P ; \ mem_1 =_{\Gamma} \ mem_2 ;$   
 $\text{pred } P \ mem_1 ; \text{pred } P \ mem_2 ; \forall x \in dom \ \Gamma. \ x \notin mds \ AsmNoReadOrWrite$   
 $\longrightarrow \text{type-max } (the \ (\Gamma \ x)) \ mem_1 \leq dma \ mem_1 \ x \rrbracket \implies$   
 $\langle c, mds, mem_1 \rangle \ \mathcal{R}_1^1_{\Gamma', \mathcal{S}', P'} \ \langle c, mds, mem_2 \rangle$

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

$bool \ (- \ \mathcal{R}_3 \text{-,,-} \ - \ [120, 120, 120, 120, 120] \ 1000)$   
**and**  $\mathcal{R}_3 :: ('Var, 'BExp) \ TyEnv \Rightarrow 'Var \ Stable \Rightarrow 'BExp \ preds \Rightarrow (('Var, 'AExp, 'BExp) \ Stmt, 'Var, 'Val) \ LocalConf \ rel$

**where**

$\mathcal{R}_3 \ \Gamma' \ S' \ P' \equiv \{(lc_1, lc_2). \ \mathcal{R}_3\text{-aux } lc_1 \ \Gamma' \ S' \ P' \ lc_2\} \mid$   
 $intro_1 \ [intro] : \llbracket \langle c_1, mds, mem_1 \rangle \ \mathcal{R}_1^1_{\Gamma, \mathcal{S}, P} \ \langle c_2, mds, mem_2 \rangle ; \vdash \ \Gamma, \mathcal{S}, P \ \{ \ c \} \rrbracket$

$$\begin{aligned}
\Gamma', \mathcal{S}', P' \Vdash & \langle \text{Seq } c_1 \ c, \text{ mds}, \text{ mem}_1 \rangle \mathcal{R}^3_{\Gamma', \mathcal{S}', P'} \langle \text{Seq } c_2 \ c, \text{ mds}, \text{ mem}_2 \rangle \mid \\
& \text{intro}_3 \ [\text{intro}] : \Vdash \langle c_1, \text{ mds}, \text{ mem}_1 \rangle \mathcal{R}^3_{\Gamma, \mathcal{S}, P} \langle c_2, \text{ mds}, \text{ mem}_2 \rangle; \vdash \Gamma, \mathcal{S}, P \{ c \} \\
\Gamma', \mathcal{S}', P' \Vdash & \langle \text{Seq } c_1 \ c, \text{ mds}, \text{ mem}_1 \rangle \mathcal{R}^3_{\Gamma', \mathcal{S}', P'} \langle \text{Seq } c_2 \ c, \text{ mds}, \text{ mem}_2 \rangle
\end{aligned}$$

**definition**

$$\begin{aligned}
\text{weak-bisim} :: & ((\text{'Var}, \text{'AExp}, \text{'BExp}) \text{ Stmt}, \text{'Var}, \text{'Val}) \text{ LocalConf rel} \Rightarrow \\
& ((\text{'Var}, \text{'AExp}, \text{'BExp}) \text{ Stmt}, \text{'Var}, \text{'Val}) \text{ LocalConf rel} \Rightarrow \text{bool}
\end{aligned}$$

**where**

$$\begin{aligned}
\text{weak-bisim } \mathcal{T}_1 \ \mathcal{T} & \equiv \forall \ c_1 \ c_2 \ \text{ mds} \ \text{ mem}_1 \ \text{ mem}_2 \ c_1' \ \text{ mds}' \ \text{ mem}_1'. \\
& ((\langle c_1, \text{ mds}, \text{ mem}_1 \rangle, \langle c_2, \text{ mds}, \text{ mem}_2 \rangle) \in \mathcal{T}_1 \wedge \\
& (\langle c_1, \text{ mds}, \text{ mem}_1 \rangle \rightsquigarrow \langle c_1', \text{ mds}', \text{ mem}_1' \rangle)) \longrightarrow \\
& (\exists \ c_2' \ \text{ mem}_2'. \langle c_2, \text{ mds}, \text{ mem}_2 \rangle \rightsquigarrow \langle c_2', \text{ mds}', \text{ mem}_2' \rangle \wedge \\
& (\langle c_1', \text{ mds}', \text{ mem}_1' \rangle, \langle c_2', \text{ mds}', \text{ mem}_2' \rangle) \in \mathcal{T})
\end{aligned}$$

**inductive-set**  $\mathcal{R} :: (\text{'Var}, \text{'BExp}) \text{ TyEnv} \Rightarrow \text{'Var Stable} \Rightarrow \text{'BExp preds} \Rightarrow$

$$\begin{aligned}
& ((\text{'Var}, \text{'AExp}, \text{'BExp}) \text{ Stmt}, \text{'Var}, \text{'Val}) \text{ LocalConf rel} \\
& \text{and } \mathcal{R}\text{-abv} ::
\end{aligned}$$

$$\begin{aligned}
& ((\text{'Var}, \text{'AExp}, \text{'BExp}) \text{ Stmt}, \text{'Var}, \text{'Val}) \text{ LocalConf} \Rightarrow \\
& (\text{'Var}, \text{'BExp}) \text{ TyEnv} \Rightarrow \text{'Var Stable} \Rightarrow \text{'BExp preds} \Rightarrow \\
& ((\text{'Var}, \text{'AExp}, \text{'BExp}) \text{ Stmt}, \text{'Var}, \text{'Val}) \text{ LocalConf} \Rightarrow \\
& \text{bool} \ (- \ \mathcal{R}^u \ \_ \ \_ \ \_ \ [120, 120, 120, 120, 120] \ 1000)
\end{aligned}$$

**for**  $\Gamma :: (\text{'Var}, \text{'BExp}) \text{ TyEnv}$

**and**  $\mathcal{S} :: \text{'Var Stable}$

**and**  $P :: \text{'BExp preds}$

**where**

$$\begin{aligned}
x \ \mathcal{R}^u_{\Gamma, \mathcal{S}, P} \ y & \equiv (x, y) \in \mathcal{R} \ \Gamma \ \mathcal{S} \ P \mid \\
\text{intro}_1: \text{lc } \mathcal{R}^1_{\Gamma, \mathcal{S}, P} \ \text{lc}' & \Longrightarrow (\text{lc}, \text{lc}') \in \mathcal{R} \ \Gamma \ \mathcal{S} \ P \mid \\
\text{intro}_3: \text{lc } \mathcal{R}^3_{\Gamma, \mathcal{S}, P} \ \text{lc}' & \Longrightarrow (\text{lc}, \text{lc}') \in \mathcal{R} \ \Gamma \ \mathcal{S} \ P
\end{aligned}$$

**inductive-cases**  $\mathcal{R}_1\text{-elim} \ [\text{elim}] : \langle c_1, \text{ mds}, \text{ mem}_1 \rangle \mathcal{R}^1_{\Gamma, \mathcal{S}, P} \langle c_2, \text{ mds}, \text{ mem}_2 \rangle$

**inductive-cases**  $\mathcal{R}_3\text{-elim} \ [\text{elim}] : \langle c_1, \text{ mds}, \text{ mem}_1 \rangle \mathcal{R}^3_{\Gamma, \mathcal{S}, P} \langle c_2, \text{ mds}, \text{ mem}_2 \rangle$

**inductive-cases**  $\mathcal{R}\text{-elim} \ [\text{elim}] : (\langle c_1, \text{ mds}, \text{ mem}_1 \rangle, \langle c_2, \text{ mds}, \text{ mem}_2 \rangle) \in \mathcal{R} \ \Gamma \ \mathcal{S} \ P$

**inductive-cases**  $\mathcal{R}\text{-elim}' : (\langle c_1, \text{ mds}, \text{ mem}_1 \rangle, \langle c_2, \text{ mds}_2, \text{ mem}_2 \rangle) \in \mathcal{R} \ \Gamma \ \mathcal{S} \ P$

**inductive-cases**  $\mathcal{R}_1\text{-elim}' : \langle c_1, \text{ mds}, \text{ mem}_1 \rangle \mathcal{R}^1_{\Gamma, \mathcal{S}, P} \langle c_2, \text{ mds}_2, \text{ mem}_2 \rangle$

**inductive-cases**  $\mathcal{R}_3\text{-elim}' : \langle c_1, \text{ mds}, \text{ mem}_1 \rangle \mathcal{R}^3_{\Gamma, \mathcal{S}, P} \langle c_2, \text{ mds}_2, \text{ mem}_2 \rangle$

**lemma**  $\mathcal{R}_1\text{-mem-eq} : \langle c_1, \text{ mds}, \text{ mem}_1 \rangle \mathcal{R}^1_{\Gamma', \mathcal{S}', P'} \langle c_2, \text{ mds}, \text{ mem}_2 \rangle \Longrightarrow \text{mem}_1$   
 $=_{\text{ mds }^l} \text{ mem}_2$

**proof** (*erule*  $\mathcal{R}_1\text{-elim}$ )

**fix**  $\Gamma \ \mathcal{S} \ P$

**assume** *wf*: *tyenv-wellformed*  $\text{ mds } \ \Gamma \ \mathcal{S} \ P$

**hence** *mds-consistent*: *mds-consistent*  $\text{ mds } \ \Gamma \ \mathcal{S} \ P$

**unfolding** *tyenv-wellformed-def* **by** *blast*

**assume** *tyenv-eq*:  $mem_1 =_{\Gamma} mem_2$   
**assume** *leq*:  $\forall x \in dom \Gamma. x \notin mds \text{ AsmNoReadOrWrite} \longrightarrow type-max (the (\Gamma x))$   
 $mem_1 \leq dma mem_1 x$   
**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*:  $x \in C \vee x \notin mds \text{ AsmNoReadOrWrite}$   
**show**  $mem_1 x = mem_2 x$   
**proof**(*cases*  $x \in dom \Gamma$ )  
**assume** *in-dom*:  $x \in dom \Gamma$   
**with** *mds-consistent* **have**  $x \notin C$   
**unfolding** *mds-consistent-def* **by** *blast*  
**with** *is-readable* **have**  $x \notin mds \text{ AsmNoReadOrWrite}$   
**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** *R<sub>1</sub>-dma-eq*:

$\langle c_1, mds, mem_1 \rangle \mathcal{R}_{\Gamma', S', P'}^1 \langle c_2, mds, mem_2 \rangle \implies dma mem_1 = dma mem_2$   
**apply**(*drule R<sub>1</sub>-mem-eq*)  
**apply**(*erule low-mds-eq-dma*)  
**done**

**lemma** *bisim-simple-R<sub>1</sub>*:

$\langle c, mds, mem \rangle \mathcal{R}_{\Gamma, S, P}^1 \langle c', mds', mem' \rangle \implies c = c'$   
**apply**(*cases rule: R<sub>1</sub>.cases, simp+*)  
**done**

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

**lemma** *bisim-simple- $\mathcal{R}_u$* :  
 $lc \mathcal{R}_{\Gamma, \mathcal{S}, P}^u lc' \implies (fst (fst lc)) = (fst (fst lc'))$   
**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** *C-eq-type-max-eq*:  
**assumes** *wf: type-wellformed t*  
**assumes** *C-eq:  $\forall x \in \mathcal{C}. mem_1 x = mem_2 x$*   
**shows** *type-max t mem<sub>1</sub> = type-max t mem<sub>2</sub>*  
**proof** –  
**have**  $\forall x \in vars\text{-of-type } t. mem_1 x = mem_2 x$   
**using** *wf C-eq unfolding type-wellformed-def by blast*  
**thus** *?thesis*  
**unfolding** *type-max-def using eval-vars-det<sub>B</sub> by fastforce*  
**qed**

**lemma** *vars-of-type-eq-type-max-eq*:  
**assumes** *mem-eq:  $\forall x \in vars\text{-of-type } t. mem_1 x = mem_2 x$*   
**shows** *type-max t mem<sub>1</sub> = type-max t mem<sub>2</sub>*  
**proof** –  
**from** *assms show ?thesis*  
**unfolding** *type-max-def using eval-vars-det<sub>B</sub> 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- $\mathcal{R}_1$ :  $\langle c, mds, mem \rangle \mathcal{R}_{\Gamma', \mathcal{S}', P'}^1 \langle c', mds', mem' \rangle$*   
**then obtain**  $\Gamma \mathcal{S} P$  **where**  
*stuff:  $c' = c \ mds' = mds \vdash \Gamma, \mathcal{S}, P \{c\} \Gamma', \mathcal{S}', P' \ \text{tyenv-wellformed mds } \Gamma \ \mathcal{S} \ P$*   
 *$mem =_{\Gamma} mem' \ \text{pred } P \ mem \ \text{pred } P \ mem'$*   
 $\forall x \in dom \ \Gamma. x \notin mds \ \text{AsmNoReadOrWrite} \implies \text{type-max } (the (\Gamma \ x)) \ mem \leq dma$



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

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

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

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

**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* ( $\mathcal{R} \Gamma \mathcal{S} P$ )  
**unfolding** *sym-def*  
**proof** (*clarify*)  
**fix**  $c_1$   $mds$   $mem_1$   $c_2$   $mds_2$   $mem_2$   
**assume** *asm*:  $(\langle c_1, mds, mem_1 \rangle, \langle c_2, mds_2, mem_2 \rangle) \in \mathcal{R} \Gamma \mathcal{S} P$   
**with**  $\mathcal{R}$ -*mds* **have** [*simp*]:  $mds_2 = mds$   
**by** *blast*  
**from** *asm* **show**  $(\langle c_2, mds_2, mem_2 \rangle, \langle c_1, mds, mem_1 \rangle) \in \mathcal{R} \Gamma \mathcal{S} P$   
**using**  $\mathcal{R}$ .*intro*<sub>1</sub> [*of*  $\Gamma \mathcal{S} P$ ] **and**  $\mathcal{R}$ .*intro*<sub>3</sub> [*of* -  $\Gamma \mathcal{S} P$ ]  
**using**  $\mathcal{R}_1$ -*sym* [*of*  $\Gamma$ ] **and**  $\mathcal{R}_3$ -*sym* [*of*  $\Gamma$ ]  
**apply** *simp*  
**apply** (*erule*  $\mathcal{R}$ -elim)  
**by** (*auto simp: sym-def*)  
**qed**

**lemma**  $\mathcal{R}_1$ -closed-glob-consistent: *closed-glob-consistent* ( $\mathcal{R}_1 \Gamma' \mathcal{S}' P'$ )  
**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*:  $\forall x. \text{case } A \ x \text{ of } None \Rightarrow True \mid Some \ (v, v') \Rightarrow mem_1 \ x \neq v \vee mem_2 \ x \neq v' \longrightarrow \neg \text{var-asm-not-written } mds \ x$   
**assume**  $A$ -*updates-dma*:  $\forall x. \text{dma } mem_1 \ [\![\!]_1 A] \ x \neq \text{dma } mem_1 \ x \longrightarrow \neg \text{var-asm-not-written } mds \ x$   
**assume**  $A$ -*updates-sec*:  $\forall x. \text{dma } mem_1 \ [\![\!]_1 A] \ x = Low \wedge (x \notin mds \text{ AsmNoReadOrWrite} \vee x \in \mathcal{C}) \longrightarrow mem_1 \ [\![\!]_1 A] \ x = mem_2 \ [\![\!]_2 A] \ x$   
**from**  $R1$  **obtain**  $\Gamma \mathcal{S} P$  **where**  $\Gamma$ -*props*:  $\vdash \Gamma, \mathcal{S}, P \{ c_1 \} \Gamma', \mathcal{S}', P' \ mem_1 =_{\Gamma} mem_2$  *tyenv-wellformed*  $mds \ \Gamma \ \mathcal{S} \ P$   
 $\text{pred } P \ mem_1 \ \text{pred } P \ mem_2$   
 $\forall x \in \text{dom } \Gamma. \ x \notin mds \ \text{AsmNoReadOrWrite} \longrightarrow$   
*type-max* (*the* ( $\Gamma \ x$ ))  $mem_1 \leq \text{dma } mem_1 \ x$   
**by** *force*  
**from**  $\Gamma$ -*props*(3) **have** *stable-not-written*:  $\forall x. \text{stable } \mathcal{S} \ x \longrightarrow \text{var-asm-not-written } mds \ x$   
**by** (*auto simp: tyenv-wellformed-def mds-consistent-def stable-def var-asm-not-written-def*)  
**with**  $A$ -*updates-vars* **have** *stable-unchanged*<sub>1</sub>:  $\forall x. \text{stable } \mathcal{S} \ x \longrightarrow (mem_1 \ [\![\!]_1 A]) \ x = mem_1 \ x$  **and**  
*stable-unchanged*<sub>2</sub>:  $\forall x. \text{stable } \mathcal{S} \ x \longrightarrow (mem_2 \ [\![\!]_2 A]) \ x = mem_2 \ x$

```

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

  from stable-not-written A-updates-dma
  have stable-unchanged-dma1:  $\forall x. \text{stable } \mathcal{S} \ x \longrightarrow \text{dma } (\text{mem}_1 \llbracket \llbracket_1 A \rrbracket \rrbracket) \ x = \text{dma}$ 
 $\text{mem}_1 \ x$ 
    by(blast)

  have tyenv-eq':  $\text{mem}_1 \llbracket \llbracket_1 A \rrbracket \rrbracket =_{\Gamma} \text{mem}_2 \llbracket \llbracket_2 A \rrbracket \rrbracket$ 
  proof(clarsimp simp: tyenv-eq-def)
    fix x
    assume a: type-max (to-total  $\Gamma \ x$ )  $\text{mem}_1 \llbracket \llbracket_1 A \rrbracket \rrbracket = \text{Low}$ 
    show  $\text{mem}_1 \llbracket \llbracket_1 A \rrbracket \rrbracket \ x = \text{mem}_2 \llbracket \llbracket_2 A \rrbracket \rrbracket \ x$ 
    proof(cases  $x \in \text{dom } \Gamma$ )
      assume in-dom:  $x \in \text{dom } \Gamma$ 
      with  $\Gamma$ -props( $\mathcal{J}$ ) 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]:  $\text{mem}_1 \llbracket \llbracket_1 A \rrbracket \rrbracket \ x = \text{mem}_1 \ x$  and [simp]:  $\text{mem}_2 \llbracket \llbracket_2 A \rrbracket \rrbracket \ x = \text{mem}_2$ 
 $x$ 
        using A-updates-vars by(auto simp: apply-adaptation-def split: option.splits)
      from in-dom a obtain  $t_x$  where  $\Gamma_x: \Gamma \ x = \text{Some } t_x$  and  $t_x\text{-Low}'$ : type-max
 $t_x (\text{mem}_1 \llbracket \llbracket_1 A \rrbracket \rrbracket) = \text{Low}$ 
        by(auto simp: to-total-def)
      have  $t_x$ -unchanged: type-max  $t_x (\text{mem}_1 \llbracket \llbracket_1 A \rrbracket \rrbracket) = \text{type-max } t_x \text{ mem}_1$ 
      proof -
        from  $\Gamma_x \ \Gamma$ -props( $\mathcal{J}$ ) 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-unchanged1 show ?thesis
          using type-stable-is-sufficient'
          by blast
      qed
      with  $t_x\text{-Low}'$  have  $t_x\text{-Low}$ : type-max  $t_x \text{ mem}_1 = \text{Low}$  by simp
      with  $\Gamma_x \ \Gamma$ -props( $\mathcal{J}$ ) have  $\text{mem}_1 \ x = \text{mem}_2 \ x$ 
      by(force simp: tyenv-eq-def to-total-def split: if-splits)
      thus ?thesis by simp
    next
    assume nin-dom:  $x \notin \text{dom } \Gamma$ 
    with a have is-Low':  $\text{dma } (\text{mem}_1 \llbracket \llbracket_1 A \rrbracket \rrbracket) \ x = \text{Low}$ 
      by(simp add: to-total-def)
    show ?thesis
    proof(cases  $x \notin \text{mds } \text{AsmNoReadOrWrite} \vee x \in \mathcal{C}$ )
      assume  $x \notin \text{mds } \text{AsmNoReadOrWrite} \vee x \in \mathcal{C}$ 
      with is-Low' show ?thesis
      using A-updates-sec by blast
    next
    assume  $\neg (x \notin \text{mds } \text{AsmNoReadOrWrite} \vee x \in \mathcal{C})$ 
    hence  $x \in \text{mds } \text{AsmNoReadOrWrite}$  and  $x \notin \mathcal{C}$ 
      by auto
  
```

```

with nin-dom  $\Gamma$ -props( $\mathcal{P}$ ) have False
  by(auto simp: tyenv-wellformed-def mds-consistent-def stable-def)
thus ?thesis by blast
qed
qed
qed

have sec':  $\forall x \in \text{dom } \Gamma. x \notin \text{mds } \text{AsmNoReadOrWrite} \longrightarrow \text{type-max } (\text{the } (\Gamma x))$ 
(mem1  $\llbracket \_ \rrbracket_1 A$ )  $\leq$  dma (mem1  $\llbracket \_ \rrbracket_1 A$ ) x
proof(intro ballI impI)
  fix x
  assume readable:  $x \notin \text{mds } \text{AsmNoReadOrWrite}$ 
  assume in-dom:  $x \in \text{dom } \Gamma$ 
  with  $\Gamma$ -props( $\mathcal{P}$ ) 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  $\llbracket \_ \rrbracket_1 A$  x = dma mem1 x
  using A-updates-dma by(auto simp: apply-adaptation-def split: option.splits)
  from in-dom obtain tx where  $\Gamma_x: \Gamma x = \text{Some } t_x$ 
  by(auto simp: to-total-def)
  have tx-unchanged:  $\text{type-max } t_x (\text{mem}_1 \llbracket \_ \rrbracket_1 A) = \text{type-max } t_x \text{ mem}_1$ 
  proof –
    from  $\Gamma_x$   $\Gamma$ -props( $\mathcal{P}$ ) have tx-stable: type-stable  $\mathcal{S} t_x$  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  $\Gamma_x$  have [simp]:  $\text{type-max } (\text{the } (\Gamma x)) (\text{mem}_1 \llbracket \_ \rrbracket_1 A) = \text{type-max } (\text{the } (\Gamma$ 
x)) mem1
  by simp
  show  $\text{type-max } (\text{the } (\Gamma x)) \text{ mem}_1 \llbracket \_ \rrbracket_1 A \leq \text{dma mem}_1 \llbracket \_ \rrbracket_1 A$  x
  apply simp
  using in-dom readable  $\Gamma$ -props by metis
qed

from stable-unchanged1 stable-unchanged2  $\Gamma$ -props( $\mathcal{P}$ ) have  $\forall p \in P. \text{ev}_B (\text{mem}_1$ 
 $\llbracket \_ \rrbracket_1 A) p = \text{ev}_B \text{ mem}_1 p \wedge \text{ev}_B (\text{mem}_2 \llbracket \_ \rrbracket_2 A) p = \text{ev}_B \text{ mem}_2 p$ 
  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  $\llbracket \_ \rrbracket_1 A$ ) = pred P mem1 and
  pred P (mem2  $\llbracket \_ \rrbracket_2 A$ ) = pred P mem2
  by(simp add: pred-def)+

with  $\Gamma$ -props tyenv-eq' sec'

```

**show**  $\langle c_1, mds, mem_1 \llbracket_1 A \rrbracket \rangle \mathcal{R}^1_{\Gamma', \mathcal{S}', P'} \langle c_2, mds, mem_2 \llbracket_2 A \rrbracket \rangle$   
**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*:  $\forall x. \text{case } A \ x \text{ of } None \Rightarrow True \mid Some \ (v, v') \Rightarrow mem_1 \ x \neq v \vee mem_2 \ x \neq v' \longrightarrow \neg \text{var-asm-not-written } mds \ x$

**assume** *A-modifies-dma*:  $\forall x. \text{dma } mem_1 \llbracket_1 A \rrbracket \ x \neq \text{dma } mem_1 \ x \longrightarrow \neg \text{var-asm-not-written } mds \ x$

**assume** *A-modifies-sec*:  $\forall x. \text{dma } mem_1 \llbracket_1 A \rrbracket \ x = Low \wedge (x \in mds \ AsmNoReadOrWrite \longrightarrow x \in \mathcal{C}) \longrightarrow mem_1 \llbracket_1 A \rrbracket \ x = mem_2 \llbracket_2 A \rrbracket \ x$

**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<sub>1</sub>-def lc<sub>2</sub>-def in- $\mathcal{R}_3$*  **have**  $lc_1 \ \mathcal{R}^3_{\Gamma', \mathcal{S}', P'} \ lc_2$  **by** *simp*

**from** *this lc<sub>1</sub>-def lc<sub>2</sub>-def A-modifies-vars A-modifies-dma A-modifies-sec*

**show**  $\langle c_1, mds, mem_1 \llbracket_1 A \rrbracket \rangle \mathcal{R}^3_{\Gamma', \mathcal{S}', P'} \langle c_2, mds, mem_2 \llbracket_2 A \rrbracket \rangle$

**proof**(*induct arbitrary: c<sub>1</sub> mds mem<sub>1</sub> c<sub>2</sub> mds mem<sub>2</sub> rule:  $\mathcal{R}_3$ -aux.induct*)

**case** (*intro<sub>1</sub> c<sub>1</sub> mds mem<sub>1</sub>  $\Gamma \ \mathcal{S} \ P \ c_2 \ mem_2 \ c \ \Gamma' \ \mathcal{S}' \ P'$* )

**show** *?case*

**apply** (*rule  $\mathcal{R}_3$ -aux.intro<sub>1</sub>[OF - intro<sub>1</sub>(2)]*)

**using**  *$\mathcal{R}_1$ -closed-glob-consistent intro<sub>1</sub>*

**unfolding** *closed-glob-consistent-def* **by** *blast*

**next**

**case** (*intro<sub>3</sub> c<sub>1</sub> mds mem<sub>1</sub>  $\Gamma \ \mathcal{S} \ P \ c_2 \ mem_2 \ c \ \Gamma' \ \mathcal{S}' \ P'$* )

**show** *?case*

**apply**(*rule  $\mathcal{R}_3$ -aux.intro<sub>3</sub>*)

**apply**(*rule intro<sub>3</sub>(2)*)

**using** *intro<sub>3</sub> 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*:  $\forall x. \text{case } A \ x \text{ of } None \Rightarrow True \mid Some \ (v, v') \Rightarrow mem_1 \ x \neq v \vee mem_2 \ x \neq v' \longrightarrow \neg \text{var-asm-not-written } mds \ x$

**and** *A-modifies-dma*:  $\forall x. \text{dma} \ (mem_1 \llbracket_1 A \rrbracket) \ x \neq \text{dma } mem_1 \ x \longrightarrow \neg \text{var-asm-not-written}$

```

mds x
and A-modifies-sec:  $\forall x. \text{dma } \text{mem}_1 \llbracket \llbracket_1 A \rrbracket x = \text{Low} \wedge (x \in \text{mds } \text{AsmNoReadOrWrite} \longrightarrow x \in \mathcal{C}) \longrightarrow \text{mem}_1 \llbracket \llbracket_1 A \rrbracket x = \text{mem}_2 \llbracket \llbracket_2 A \rrbracket x$ 
show
   $\langle c_1, \text{mds}, \text{mem}_1 \llbracket \llbracket_1 A \rrbracket \rangle \mathcal{R}^u_{\Gamma', \mathcal{S}', P'} \langle c_2, \text{mds}, \text{mem}_2 \llbracket \llbracket_2 A \rrbracket \rangle$ 
apply(rule intro1)
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  $c_1 \text{ mds } \text{mem}_1 c_2 \text{ mem}_2 x A$ 
assume R3:  $\langle c_1, \text{mds}, \text{mem}_1 \rangle \mathcal{R}^3_{\Gamma', \mathcal{S}', P'} \langle c_2, \text{mds}, \text{mem}_2 \rangle$ 
and A-modifies-vars:  $\forall x. \text{case } A \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } (v, v') \Rightarrow \text{mem}_1 x \neq v \vee \text{mem}_2 x \neq v' \longrightarrow \neg \text{var-asm-not-written } \text{mds } x$ 
and A-modifies-dma:  $\forall x. \text{dma } (\text{mem}_1 \llbracket \llbracket_1 A \rrbracket) x \neq \text{dma } \text{mem}_1 x \longrightarrow \neg \text{var-asm-not-written } \text{mds } x$ 
and A-modifies-sec:  $\forall x. \text{dma } \text{mem}_1 \llbracket \llbracket_1 A \rrbracket x = \text{Low} \wedge (x \in \text{mds } \text{AsmNoReadOrWrite} \longrightarrow x \in \mathcal{C}) \longrightarrow \text{mem}_1 \llbracket \llbracket_1 A \rrbracket x = \text{mem}_2 \llbracket \llbracket_2 A \rrbracket x$ 
show  $\langle c_1, \text{mds}, \text{mem}_1 \llbracket \llbracket_1 A \rrbracket \rangle \mathcal{R}^u_{\Gamma', \mathcal{S}', P'} \langle c_2, \text{mds}, \text{mem}_2 \llbracket \llbracket_2 A \rrbracket \rangle$ 
apply(rule intro3)
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  $\Gamma \mathcal{S} P \Longrightarrow$ 
    mds-consistent (update-modes upd mds)
       $(\Gamma \oplus_{\mathcal{S}} \text{upd})$ 
       $(\text{add-anno-stable } \mathcal{S} \text{ upd})$ 
       $(P \mid \{v. \text{stable } (\text{add-anno-stable } \mathcal{S} \text{ upd}) v\})$ 
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]:
   $\Gamma \oplus_{\mathcal{S}} (v \text{ +=}_m \text{GuarNoReadOrWrite}) = \Gamma$ 
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-GuarNoReadOrWrite* [*simp*]:  
 $\Gamma \oplus_S (v \text{ --}_m \text{ GuarNoReadOrWrite}) = \Gamma$   
**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*]:  
 $\Gamma \oplus_S (v \text{ +=}_m \text{ GuarNoWrite}) = \Gamma$   
**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*]:  
 $\Gamma \oplus_S (v \text{ --}_m \text{ GuarNoWrite}) = \Gamma$   
**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*:  
 $\forall x \in \text{dom} (\Gamma \oplus_S (v \text{ --}_m m)). (\Gamma \oplus_S (v \text{ --}_m m)) x = \Gamma x$   
**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*  $\Gamma \implies$   
*types-wellformed* ( $\Gamma \oplus_S \text{ upd}$ )  
**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 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 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  $\Gamma \mathcal{S} \implies$  types-wellformed  $\Gamma \implies$  anno-type-stable  $\Gamma \mathcal{S} \text{ upd}$ 
   $\implies$  types-stable  $(\Gamma \oplus_{\mathcal{S}} \text{upd})$  (add-anno-stable  $\mathcal{S} \text{ upd}$ )
  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  $\Gamma \vee v \in C$ )
      apply clarsimp
      apply(force simp: stable-def)
      apply(simp split: if-splits)
      using 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  $\Gamma \vee v \in C$ )
      apply clarsimp
      apply(force simp: stable-def)
      apply(simp split: if-splits)
      using C-vars-correct
      apply(fastforce simp: anno-type-stable-def stable-def)
      apply(force simp: stable-def)
    apply(force simp: stable-def)
  apply(rename-tac v m)
  apply(subgoal-tac  $\Gamma x = \text{Some } (C)$ )
  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 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 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  $\Gamma$   $\mathcal{S}$   $P \implies$  anno-type-stable  $\Gamma$   $\mathcal{S}$   $upd \implies$ 
tyenv-wellformed (update-modes  $upd$  mds)
  ( $\Gamma \oplus_{\mathcal{S}} upd$ )
  (add-anno-stable  $\mathcal{S}$   $upd$ )
  ( $P \mid' \{v. \text{stable } (add-anno-stable \mathcal{S} \text{ } upd) v\}$ )
  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* :

```

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

```

**lemma** *tyenv-wellformed-preds-update*:

```

 $P' = P'' \mid' \{v. \text{stable } \mathcal{S} v\} \implies$ 
tyenv-wellformed mds  $\Gamma \mathcal{S} P \implies$  tyenv-wellformed mds  $\Gamma \mathcal{S} P'$ 
  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*:

```

 $P' = P'' \mid' \{v. \text{stable } \mathcal{S} v\} \implies v \in \text{dom } \Gamma \implies$ 
mds-consistent mds  $\Gamma \mathcal{S} P \implies$  mds-consistent mds ( $\Gamma(v \mapsto t)$ )  $\mathcal{S} P'$ 
  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*:  $P' = (assign-post P x e) \mid \{v. stable \mathcal{S} v\}$   
**assumes** *pred-P*:  $pred P mem$   
**shows**  $pred P' mem'$

**proof** –

**from** *P'-def* **have** *P'-def'*:  $P' \subseteq assign-post P x e$   
**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*:

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

**lemma** *types-stable-update*:

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

**lemma** *tyenv-wellformed-sub*:

$\llbracket P_1 \subseteq P_2; \Gamma_2 = \Gamma_1; tyenv-wellformed mds \Gamma_2 \mathcal{S} P_2 \rrbracket \implies$   
 $tyenv-wellformed mds \Gamma_1 \mathcal{S} P_1$   
**apply**(*clarsimp simp: tyenv-wellformed-def | safe*)  
**apply**(*fastforce simp: mds-consistent-def*)  
**done**

**abbreviation**

$tyenv-sec :: 'Var Mds \Rightarrow ('Var, 'BExp) TyEnv \Rightarrow ('Var, 'Val) Mem \Rightarrow bool$

**where**

$tyenv-sec mds \Gamma mem \equiv \forall x \in dom \Gamma. x \notin mds \ AsmNoReadOrWrite \longrightarrow type-max$   
 $(the (\Gamma x)) mem \leq dma mem x$

**lemma** *tyenv-sec-mode-update*:

$(\forall x. (to-total \Gamma x) \leq_{P''} (to-total \Gamma'' x)) \implies pred P'' mem \implies \mathcal{S} = (mds$   
 $AsmNoWrite, mds AsmNoReadOrWrite) \implies$   
 $anno-type-sec \Gamma \mathcal{S} P upd \implies \mathcal{S}'' = add-anno-stable \mathcal{S} upd \implies (\forall p \in P.$   
 $\forall v \in bexp-vars p. stable \mathcal{S} v) \implies$   
 $P'' = P \mid \{v. stable \mathcal{S}'' v\} \implies$   
 $\Gamma'' = \Gamma \oplus_{\mathcal{S}} upd \implies tyenv-sec mds \Gamma mem \implies tyenv-sec (update-modes upd$   
 $mds) \Gamma'' mem$   
**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 ∈ mds AsmNoWrite → 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*:

```

  ∀ x ∈ C. mem x = mem' x ⇒ types-wellformed Γ ⇒ 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-C apply fastforce
  apply(force intro: C-eq-type-max-eq simp: types-wellformed-def)
done

```

**lemma** *context-equiv-tyenv-sec*:

```

context-equiv Γ2 P2 Γ1 ⇒
  pred P2 mem ⇒ tyenv-sec mds Γ2 mem ⇒ tyenv-sec mds Γ1 mem
apply(clarsimp simp: context-equiv-def type-equiv-def)
apply(rename-tac x y)
apply(rule-tac y=type-max (the (Γ2 x)) mem in order-trans)
apply(rule subtype-sound[rule-format])
  apply force
  apply assumption
apply force
done

```

**lemma** *add-pred-entailment*:

```

P +S p ⊢ P
apply(rule subset-entailment)
apply(rule add-pred-subset)
done

```

**lemma** *preservation-no-await*:

```

[[⊢ Γ, S, P { c } Γ', S', P';
  ⟨c, mds, mem⟩ ∼ ⟨c', mds', mem'⟩;
  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')

```

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

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

**next**

**case** (*assign*<sub>1</sub>  $x \Gamma e t P P' \mathcal{S} c' mds$ )  
**hence** *upd*:  $c' = \text{Stop} \wedge mds' = mds \wedge mem' = mem (x := ev_A mem e)$   
**by** (*metis assign-elim*)  
**from** *assign*<sub>1</sub>(2) *upd* **have** *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*  $mds \Gamma \mathcal{S} P \longrightarrow tyenv\text{-wellformed} mds' \Gamma \mathcal{S} P'$   
**using** *upd tyenv-wellformed-preds-update assign*<sub>1</sub> **by** *metis*  
**moreover** **have** *pred*  $P mem \longrightarrow pred P' mem'$   
**using** *pred-preds-update assign*<sub>1</sub> *upd* **by** *metis*

**moreover** **have** *tyenv-wellformed*  $mds \Gamma \mathcal{S} P \wedge tyenv\text{-sec} mds \Gamma mem \longrightarrow tyenv\text{-sec} mds \Gamma mem'$   
**using** *tyenv-sec-eq[OF C-eq, where  $\Gamma = \Gamma$ ]*  
**unfolding** *tyenv-wellformed-def* **by** *blast*  
**ultimately show** *?case*  
**by** (*metis upd*)

**next**  
**case** (*assign*<sub>2</sub>  $x \Gamma e t \mathcal{S} P' P c' mds$ )  
**hence** *upd*:  $c' = \text{Stop} \wedge mds' = mds \wedge mem' = mem (x := ev_A mem e)$   
**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*  $mds \Gamma \mathcal{S} P \longrightarrow tyenv\text{-wellformed} mds' ? \Gamma' \mathcal{S} P'$   
**proof**  
**assume** *tyenv-wf*: *tyenv-wellformed*  $mds \Gamma \mathcal{S} P$   
**hence** *wf*: *types-wellformed*  $\Gamma$   
**unfolding** *tyenv-wellformed-def* **by** *blast*  
**hence** *type-wellformed*  $t$   
**using** *assign*<sub>2</sub>(2) *type-aexpr-type-wellformed*  
**by** *blast*  
**with** *wf* **have** *wf'*: *types-wellformed*  $? \Gamma'$   
**using** *types-wellformed-update* **by** *metis*  
**from** *tyenv-wf* **have** *stable'*: *types-stable*  $? \Gamma' \mathcal{S}$   
**using** *types-stable-update assign*<sub>2</sub>(3)  
**unfolding** *tyenv-wellformed-def* **by** *blast*  
**have** *m*: *mds-consistent*  $mds \Gamma \mathcal{S} P$   
**using** *tyenv-wf unfolding tyenv-wellformed-def* **by** *blast*  
**from** *assign*<sub>2</sub>(4) *assign*<sub>2</sub>(1)  
**have** *mds-consistent*  $mds' (\Gamma(x \mapsto t)) \mathcal{S} P'$   
**apply**(*rule mds-consistent-preds-tyenv-update*)  
**using** *upd m* **by** *simp*  
**from** *wf' stable'* **this show** *tyenv-wellformed*  $mds' ? \Gamma' \mathcal{S} P'$   
**unfolding** *tyenv-wellformed-def* **by** *blast*

**qed**  
**have** *p*: *pred*  $P mem \longrightarrow pred P' mem'$

```

    using pred-preds-update assign2 upd by metis
  have sec: tyenv-wellformed mds  $\Gamma$   $\mathcal{S}$   $P \implies$  pred  $P$  mem  $\implies$  tyenv-sec mds  $\Gamma$ 
  mem  $\implies$  tyenv-sec mds'  $\mathcal{?}\Gamma'$  mem'
  proof (clarify)
    assume wf: tyenv-wellformed mds  $\Gamma$   $\mathcal{S}$   $P$ 
    assume pred: pred  $P$  mem
    assume sec: tyenv-sec mds  $\Gamma$  mem
    from pred p have pred': pred  $P'$  mem' by blast
    fix v t'
    assume  $\Gamma v$ : ( $\Gamma(x \mapsto t)$ ) 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  $\Gamma v$  by auto
      from  $\langle v \notin$  mds' AsmNoReadOrWrite  $\rangle$  upd wf have readable: v  $\notin$  snd  $\mathcal{S}$ 
        by (auto simp: tyenv-wellformed-def mds-consistent-def)
      with assign2(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) =  $\Gamma v$ 
        by simp
      with  $\Gamma v$  have  $\Gamma v$ :  $\Gamma v$  = Some t' by simp
      with sec upd  $\langle v \notin$  mds' AsmNoReadOrWrite  $\rangle$  have f-leq: type-max t' mem  $\leq$ 
        dma mem v
        by auto
      have C-eq:  $\forall x \in \mathcal{C}. mem\ x = mem'\ x$ 
        using wf assign2(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  $\Gamma v$  wf apply (force simp: tyenv-wellformed-def types-wellformed-def)
        by (rule C-eq)
      from neq  $\Gamma v$  f-leq dma-eq f-eq show ?thesis
        by simp
    qed
  qed
  from ty wf p sec show ?case
    by blast
  next
  case (assignC x  $\Gamma$  e t  $P P' \mathcal{S} c' mds$ )

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

```

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

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



```

    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  $\Gamma \mathcal{S} P c_1 \Gamma_1 \mathcal{S}_1 P_1 c_2 \Gamma_2 \mathcal{S}_2 P_2 c' mds$ )
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  $\Gamma P \Gamma_1$  and [simp]:  $\mathcal{S}_1 = \mathcal{S}$  and entail:  $P \vdash P_1$ 
and
  wf-imp:  $\forall mds. tyenv-wellformed mds \Gamma \mathcal{S} P \longrightarrow tyenv-wellformed mds \Gamma_1 \mathcal{S} P_1$ 
  using stop-ctx seq-type(1) by simp+
  have  $\vdash \Gamma, \mathcal{S}, P \{c_2\} \Gamma_2, \mathcal{S}_2, P_2$ 
  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.premis)
  then have no-await  $c_1$  using  $\langle no-await (c_1 ;; c_2) \rangle no-await.cases$  by blast
  then obtain  $\Gamma''' \mathcal{S}''' P'''$  where  $\vdash \Gamma''', \mathcal{S}''', P''' \{c_1'\} \Gamma_1, \mathcal{S}_1, P_1 \wedge$ 
    ( $tyenv-wellformed mds \Gamma \mathcal{S} P \wedge pred P mem \wedge tyenv-sec mds \Gamma mem \longrightarrow$ 
     $tyenv-wellformed mds' \Gamma''' \mathcal{S}''' P''' \wedge pred P''' mem' \wedge tyenv-sec mds' \Gamma'''$ 
 $mem'$ )

```

```

    using step seq-type(2)
    by blast
  moreover
  from seq-type have  $\vdash \Gamma_1, \mathcal{S}_1, P_1 \{c_2\} \Gamma_2, \mathcal{S}_2, P_2$  by auto
  moreover
  ultimately show ?case
    apply (rule-tac  $x = \Gamma'''$  in exI)
    using  $\langle c_1, mds, mem \rangle \rightsquigarrow \langle c_1', mds', mem' \rangle \wedge c' = c_1' ;; c_2$  by blast
  qed
next
case (sub  $\Gamma_1 \mathcal{S} P_1 c \Gamma_1' \mathcal{S}' P_1' \Gamma_2 P_2 \Gamma_2' P_2' c' mds$ )
then obtain  $\Gamma'' \mathcal{S}'' P''$  where stuff:  $\vdash \Gamma'', \mathcal{S}'', P'' \{c'\} \Gamma_1', \mathcal{S}', P_1' \wedge$ 
  ( $tyenv\text{-wellformed } mds \Gamma_1 \mathcal{S} P_1 \wedge pred P_1 mem \wedge tyenv\text{-sec } mds \Gamma_1 mem \longrightarrow$ 
   $tyenv\text{-wellformed } mds' \Gamma'' \mathcal{S}'' P'' \wedge pred P'' mem' \wedge tyenv\text{-sec } mds' \Gamma'' mem'$ )
  by force

  have imp:  $tyenv\text{-wellformed } mds \Gamma_2 \mathcal{S} P_2 \wedge pred P_2 mem \wedge tyenv\text{-sec } mds \Gamma_2$ 
  mem  $\implies$ 
     $tyenv\text{-wellformed } mds \Gamma_1 \mathcal{S} P_1 \wedge pred P_1 mem \wedge tyenv\text{-sec } mds \Gamma_1 mem$ 
  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  $x = \Gamma''$  in exI, rule-tac  $x = \mathcal{S}''$  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  $\Gamma e t P \mathcal{S} c \Gamma' \mathcal{S}' P' c' mds$ )
  show ?case using no-await-no-await await-type.premis by blast
qed

lemma preservation-no-await-plus:
   $\llbracket \langle c, mds, mem \rangle \rightsquigarrow^+ \langle c', mds', mem' \rangle; \vdash \Gamma, \mathcal{S}, P \{c\} \Gamma', \mathcal{S}', P'; no\text{-await } c \rrbracket \implies$ 
   $no\text{-await } c' \wedge (\exists \Gamma'' \mathcal{S}'' P''. (\vdash \Gamma'', \mathcal{S}'', P'' \{c'\} \Gamma', \mathcal{S}', P') \wedge$ 

```

$(\text{tyenv-wellformed mds } \Gamma \mathcal{S} P \wedge \text{pred } P \text{ mem} \wedge \text{tyenv-sec mds } \Gamma \text{ mem} \longrightarrow$   
 $\text{tyenv-wellformed mds}' \Gamma'' \mathcal{S}'' P'' \wedge \text{pred } P'' \text{ mem}' \wedge \text{tyenv-sec mds}' \Gamma'' \text{ mem}'\wedge)$   
**apply** (*induct arbitrary:  $\Gamma \mathcal{S} P$  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:*  $\vdash \Gamma, \mathcal{S}, P \{c\} \Gamma', \mathcal{S}', P'$   
**assumes** *eval:*  $\langle c, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \rangle$   
**shows**  $\exists \Gamma'' \mathcal{S}'' P'' . (\vdash \Gamma'', \mathcal{S}'', P'' \{c'\} \Gamma', \mathcal{S}', P') \wedge$   
 $(\text{tyenv-wellformed mds } \Gamma \mathcal{S} P \wedge \text{pred } P \text{ mem} \wedge \text{tyenv-sec mds } \Gamma$   
 $\text{mem} \longrightarrow$   
 $\text{tyenv-wellformed mds}' \Gamma'' \mathcal{S}'' P'' \wedge \text{pred } P'' \text{ mem}' \wedge \text{tyenv-sec$   
 $\text{mds}' \Gamma'' \text{ mem}'\wedge)$   
**using** *typed eval*  
**proof** (*induct arbitrary:  $c'$  mds rule: has-type.induct*)

**case** (*anno-type*  $\Gamma'' \Gamma \mathcal{S} \text{ upd } \mathcal{S}'' P'' P c_1 \Gamma' \mathcal{S}' P'$ )  
**hence**  $\langle c_1, \text{update-modes upd mds}, \text{mem} \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \rangle$   
**by** (*metis upd-elim*)  
**with** *anno-type(5)* **obtain**  $\Gamma''' \mathcal{S}''' P'''$  **where**  
 $\vdash \Gamma''', \mathcal{S}''', P''' \{c'\} \Gamma', \mathcal{S}', P' \wedge$   
 $(\text{tyenv-wellformed (update-modes upd mds)} \Gamma'' \mathcal{S}'' P'' \wedge \text{pred } P'' \text{ mem} \wedge$   
 $\text{tyenv-sec (update-modes upd mds)} \Gamma'' \text{ mem} \longrightarrow$   
 $\text{tyenv-wellformed mds}' \Gamma''' \mathcal{S}''' P''' \wedge \text{pred } P''' \text{ mem}' \wedge \text{tyenv-sec mds}' \Gamma'''$   
 $\text{mem}'\wedge)$   
**by** *blast*

**moreover**

**have**  $\text{tyenv-wellformed mds } \Gamma \mathcal{S} P \longrightarrow \text{tyenv-wellformed (update-modes upd mds)}$   
 $\Gamma'' \mathcal{S}'' P''$

**using** *anno-type*

**apply** *auto*

**by** (*metis tyenv-wellformed-mode-update*)

**moreover**

**have**  $\text{pred: pred } P \text{ mem} \longrightarrow \text{pred } P'' \text{ mem}$

**using** *anno-type*

**by** (*auto simp: pred-def restrict-preds-to-vars-def*)

**moreover**

**have**  $\text{tyenv-wellformed mds } \Gamma \mathcal{S} P \wedge \text{pred } P \text{ mem} \wedge \text{tyenv-sec mds } \Gamma \text{ mem} \longrightarrow$   
 $\text{tyenv-sec (update-modes upd mds)} \Gamma'' \text{ mem}$

**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 (assign1 x  $\Gamma$  e t P P' S c' mds)
hence upd:  $c' = Stop \wedge mds' = mds \wedge mem' = mem (x := ev_A mem e)$ 
  by (metis assign-elim)
from assign1(2) upd have 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  $\Gamma$  S P  $\longrightarrow$  tyenv-wellformed mds'  $\Gamma$  S P'
  using upd tyenv-wellformed-preds-update assign1 by metis
moreover have pred P mem  $\longrightarrow$  pred P' mem'
  using pred-preds-update assign1 upd by metis

  moreover have tyenv-wellformed mds  $\Gamma$  S P  $\wedge$  tyenv-sec mds  $\Gamma$  mem  $\longrightarrow$ 
  tyenv-sec mds  $\Gamma$  mem'
  using tyenv-sec-eq[OF C-eq, where  $\Gamma = \Gamma$ ]
  unfolding tyenv-wellformed-def by blast
  ultimately show ?case
  by (metis upd)
next
case (assign2 x  $\Gamma$  e t 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  $\Gamma$  S P  $\longrightarrow$  tyenv-wellformed mds' ? $\Gamma$ ' S P'
proof
  assume tyenv-wf: tyenv-wellformed mds  $\Gamma$  S P
  hence wf: types-wellformed  $\Gamma$ 
  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 ? $\Gamma'$ 
  using types-wellformed-update by metis
from tyenv-wf have stable': types-stable ? $\Gamma'$   $\mathcal{S}$ 
  using types-stable-update
    assign2(3)
  unfolding tyenv-wellformed-def by blast
have m: mds-consistent mds  $\Gamma$   $\mathcal{S}$   $P$ 
  using tyenv-wf unfolding tyenv-wellformed-def by blast
from assign2(4) assign2(1)
have mds-consistent mds' ( $\Gamma(x \mapsto t)$ )  $\mathcal{S}$   $P'$ 
  apply(rule mds-consistent-preds-tyenv-update)
  using upd m by simp
from wf' stable' this show tyenv-wellformed mds' ? $\Gamma'$   $\mathcal{S}$   $P'$ 
  unfolding tyenv-wellformed-def by blast
qed
have p: pred  $P$  mem  $\longrightarrow$  pred  $P'$  mem'
  using pred-preds-update assign2 upd by metis
have sec: tyenv-wellformed mds  $\Gamma$   $\mathcal{S}$   $P \implies$  pred  $P$  mem  $\implies$  tyenv-sec mds  $\Gamma$ 
mem  $\implies$  tyenv-sec mds' ? $\Gamma'$  mem'
proof(clarify)
  assume wf: tyenv-wellformed mds  $\Gamma$   $\mathcal{S}$   $P$ 
  assume pred: pred  $P$  mem
  assume sec: tyenv-sec mds  $\Gamma$  mem
from pred p have pred': pred  $P'$  mem' by blast
fix v t'
assume  $\Gamma v$ : ( $\Gamma(x \mapsto t)$ ) 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]: ( $\Gamma(x \mapsto t)$ ) v = t and t-def: t = t'
  using  $\Gamma v$  by auto
from  $\langle v \notin$  mds' AsmNoReadOrWrite  $\rangle$  upd wf have readable: v  $\notin$  snd  $\mathcal{S}$ 
  by(auto simp: tyenv-wellformed-def mds-consistent-def)
with assign2(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) =  $\Gamma$  v
  by simp
with  $\Gamma v$  have  $\Gamma v$ :  $\Gamma$  v = Some t' by simp
with sec upd  $\langle v \notin$  mds' AsmNoReadOrWrite  $\rangle$  have f-leq: type-max t' mem  $\leq$ 
dma mem v
  by auto
have C-eq:  $\forall x \in \mathcal{C}. mem$  x = mem' x
using wf assign2(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  $\Gamma v$  wf apply(force simp: tyenv-wellformed-def types-wellformed-def)
    by(rule C-eq)
  from neq  $\Gamma v$  f-leq dma-eq f-eq show ?thesis
    by simp
qed
qed
from ty wf p sec show ?case
  by blast
next
case (assignC x  $\Gamma$  e t P P' S c' mds)

  hence upd: c' = Stop  $\wedge$  mds' = mds  $\wedge$  mem' = mem (x := evA mem e)
    by (metis assign-elim)
  hence  $\vdash \Gamma, S, P' \{c'\} \Gamma, S, P'$ 
    by (metis stop-type)
  moreover have tyenv-wellformed mds  $\Gamma$  S P  $\longrightarrow$  tyenv-wellformed mds'  $\Gamma$  S P'
    using upd tyenv-wellformed-preds-update assignC by metis
  moreover have pred P mem  $\longrightarrow$  pred P' mem'
    using pred-preds-update assignC upd by metis
  moreover have tyenv-wellformed mds  $\Gamma$  S P  $\wedge$  pred P mem  $\wedge$  tyenv-sec mds  $\Gamma$ 
    mem  $\implies$  tyenv-sec mds'  $\Gamma$  mem'
  proof (clarify)
    fix v t'
    assume wf: tyenv-wellformed mds  $\Gamma$  S P
    assume pred: pred P mem
    hence pred': pred P' mem' using  $\langle$ pred P mem  $\longrightarrow$  pred P' mem' $\rangle$  by blast
    assume sec: tyenv-sec mds  $\Gamma$  mem
    assume  $\Gamma v$ :  $\Gamma v = \text{Some } t'$ 
    assume readable': v  $\notin$  mds' AsmNoReadOrWrite
    with upd have readable: v  $\notin$  mds AsmNoReadOrWrite by simp
    with wf have v  $\notin$  snd S by (auto simp: tyenv-wellformed-def mds-consistent-def)
    show type-max (the ( $\Gamma v$ )) mem'  $\leq$  dma mem' v
    proof (cases x  $\in$  C-vars v)
      assume x  $\in$  C-vars v
      with assignC(6)  $\langle$ v  $\notin$  snd S $\rangle$  have (to-total  $\Gamma v$ )  $\leq_{P'}$  (dma-type v) by blast
      from pred'  $\Gamma v$  subtype-sound[OF this] show ?thesis
        using type-max-dma-type by (auto simp: to-total-def split: if-splits)
    next
      assume x  $\notin$  C-vars v
      hence  $\forall v' \in \text{C-vars } v. \text{mem } v' = \text{mem}' v'$ 
        using upd by auto
      hence dma-eq: dma mem v = dma mem' v
        by (rule dma-C-vars)
      from  $\Gamma v$  assignC(4) have x  $\notin$  vars-of-type t' by force
      have type-wellformed t'
        using wf  $\Gamma v$  by (force simp: tyenv-wellformed-def types-wellformed-def)

```

**with**  $\langle x \notin \text{vars-of-type } t' \rangle \text{ upd}$  **have**  $f\text{-eq: type-max } t' \text{ mem} = \text{type-max } t' \text{ mem}'$   
**using**  $\text{vars-of-type-eq-type-max-eq}$  **by**  $\text{fastforce}$   
**from**  $\text{sec } \Gamma v \text{ readable}$  **have**  $\text{type-max } t' \text{ mem} \leq \text{dma mem } v$   
**by**  $\text{auto}$   
**with**  $f\text{-eq dma-eq } \Gamma v$  **show**  $?thesis$   
**by**  $\text{simp}$   
**qed**  
**qed**  
**ultimately show**  $?case$   
**by**  $(\text{metis stop-type})$   
**next**  
**case**  $(\text{if-type } \Gamma \ e \ t \ P \ \mathcal{S} \ \text{th } \Gamma' \ \mathcal{S}' \ P' \ \text{el } \Gamma'' \ P'' \ \Gamma''' \ P''' \ c' \ \text{mds})$   
**from**  $\text{if-type}(1\beta)$   
**show**  $?case$   
**proof** $(\text{rule if-elim})$   
**assume**  $[\text{simp}]: \text{ev}_B \text{ mem } e$  **and**  $[\text{simp}]: c' = \text{th}$  **and**  $[\text{simp}]: \text{mem}' = \text{mem}$  **and**  
 $[\text{simp}]: \text{mds}' = \text{mds}$   
**from**  $\text{if-type}(3)$  **have**  $\vdash \Gamma, \mathcal{S}, P +_{\mathcal{S}} e \{c'\} \Gamma', \mathcal{S}', P'$  **by**  $\text{simp}$   
**hence**  $\vdash \Gamma, \mathcal{S}, P +_{\mathcal{S}} e \{c'\} \Gamma''', \mathcal{S}', P'''$   
**apply** $(\text{rule sub})$   
**apply**  $\text{simp}+$   
**using**  $\text{if-type apply blast}$   
**using**  $\text{if-type apply blast}$   
**apply**  $\text{simp}$   
**using**  $\text{if-type apply}(\text{blast intro: pred-entailment-trans})$   
**done**  
**moreover have**  $\text{tyenv-wellformed mds } \Gamma \ \mathcal{S} \ P \longrightarrow \text{tyenv-wellformed mds}' \ \Gamma \ \mathcal{S}$   
 $(P +_{\mathcal{S}} e)$   
**by** $(\text{auto simp: tyenv-wellformed-def mds-consistent-def add-pred-def})$   
**moreover have**  $\text{pred } P \ \text{mem} \longrightarrow \text{pred } (P +_{\mathcal{S}} e) \ \text{mem}'$   
**by** $(\text{auto simp: pred-def add-pred-def})$   
**moreover have**  $\text{tyenv-sec mds } \Gamma \ \text{mem} \longrightarrow \text{tyenv-sec mds}' \ \Gamma \ \text{mem}'$   
**by** $(\text{simp})$   
**ultimately show**  $?case$  **by**  $\text{blast}$   
**next**  
**assume**  $[\text{simp}]: \neg \text{ev}_B \text{ mem } e$  **and**  $[\text{simp}]: c' = \text{el}$  **and**  $[\text{simp}]: \text{mem}' = \text{mem}$   
**and**  $[\text{simp}]: \text{mds}' = \text{mds}$   
**from**  $\text{if-type}(5)$  **have**  $\vdash \Gamma, \mathcal{S}, P +_{\mathcal{S}} \text{bexp-neg } e \{c'\} \Gamma'', \mathcal{S}', P''$  **by**  $\text{simp}$   
**hence**  $\vdash \Gamma, \mathcal{S}, P +_{\mathcal{S}} \text{bexp-neg } e \{c'\} \Gamma''', \mathcal{S}', P'''$   
**apply** $(\text{rule sub})$   
**apply**  $\text{simp}+$   
**using**  $\text{if-type apply blast}$   
**using**  $\text{if-type apply blast}$   
**apply**  $\text{simp}$   
**using**  $\text{if-type apply}(\text{blast intro: pred-entailment-trans})$   
**done**  
**moreover have**  $\text{tyenv-wellformed mds } \Gamma \ \mathcal{S} \ P \longrightarrow \text{tyenv-wellformed mds}' \ \Gamma \ \mathcal{S}$   
 $(P +_{\mathcal{S}} \text{bexp-neg } e)$

```

    by(auto simp: tyenv-wellformed-def mds-consistent-def add-pred-def)
  moreover have pred P mem  $\longrightarrow$  pred (P +S bexp-neg e) mem'
    by(auto simp: pred-def add-pred-def bexp-neg-negates)
  moreover have tyenv-sec mds  $\Gamma$  mem  $\longrightarrow$  tyenv-sec mds'  $\Gamma$  mem'
    by(simp)
  ultimately show ?case by blast
qed
next
case (while-type  $\Gamma$  e t P S c c' mds)
hence [simp]: mds' = mds  $\wedge$  c' = If e (c ;; While e c) Stop  $\wedge$  mem' = mem
  by (metis while-elim)
have  $\vdash \Gamma, \mathcal{S}, P \{c'\} \Gamma, \mathcal{S}, P$ 
  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 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  $\Gamma$  S P c1  $\Gamma_1$  S1 P1 c2  $\Gamma_2$  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  $\Gamma$  P  $\Gamma_1$  and [simp]: S1 = S and entail: P  $\vdash$  P1
  and
    wf-imp:  $\forall$  mds. tyenv-wellformed mds  $\Gamma$  S P  $\longrightarrow$  tyenv-wellformed mds
 $\Gamma_1$  S P1
    using stop-ctx seq-type(1) by simp+
  have  $\vdash \Gamma, \mathcal{S}, P \{c_2\} \Gamma_2, \mathcal{S}_2, P_2$ 
    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  $c_1 \neq \text{Stop}$ 
  then obtain  $c_1'$  where  $\langle c_1, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c_1', \text{mds}', \text{mem}' \rangle \wedge c' = (c_1' ;; c_2)$ 
    by (metis seq-elim seq-type.prem)
  then obtain  $\Gamma''' S''' P'''$  where  $\vdash \Gamma''', S''', P''' \{c_1'\} \Gamma_1, \mathcal{S}_1, P_1 \wedge$ 
    ( $\text{tyenv-wellformed mds } \Gamma \mathcal{S} P \wedge \text{pred } P \text{ mem} \wedge \text{tyenv-sec mds } \Gamma \text{ mem} \longrightarrow$ 
     $\text{tyenv-wellformed mds}' \Gamma''' S''' P''' \wedge \text{pred } P''' \text{ mem}' \wedge \text{tyenv-sec mds}' \Gamma'''$ 
 $\text{mem}'$ )
    using seq-type(2)
    by force
  moreover
  from seq-type have  $\vdash \Gamma_1, \mathcal{S}_1, P_1 \{c_2\} \Gamma_2, \mathcal{S}_2, P_2$  by auto
  moreover
  ultimately show ?case
    apply (rule-tac  $x = \Gamma'''$  in exI)
    using  $\langle c_1, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c_1', \text{mds}', \text{mem}' \rangle \wedge c' = c_1' ;; c_2$  by blast
qed
next
  case (sub  $\Gamma_1 \mathcal{S} P_1 c \Gamma_1' \mathcal{S}' P_1' \Gamma_2 P_2 \Gamma_2' P_2' c' \text{mds}$ )
  then obtain  $\Gamma'' S'' P''$  where  $\text{stuff}: \vdash \Gamma'', S'', P'' \{c'\} \Gamma_1', \mathcal{S}', P_1' \wedge$ 
    ( $\text{tyenv-wellformed mds } \Gamma_1 \mathcal{S} P_1 \wedge \text{pred } P_1 \text{ mem} \wedge \text{tyenv-sec mds } \Gamma_1 \text{ mem} \longrightarrow$ 
     $\text{tyenv-wellformed mds}' \Gamma'' S'' P'' \wedge \text{pred } P'' \text{ mem}' \wedge \text{tyenv-sec mds}' \Gamma'' \text{mem}'$ )
    by force

  have  $\text{imp}: \text{tyenv-wellformed mds } \Gamma_2 \mathcal{S} P_2 \wedge \text{pred } P_2 \text{ mem} \wedge \text{tyenv-sec mds } \Gamma_2$ 
 $\text{mem} \implies$ 
     $\text{tyenv-wellformed mds } \Gamma_1 \mathcal{S} P_1 \wedge \text{pred } P_1 \text{ mem} \wedge \text{tyenv-sec mds } \Gamma_1 \text{ mem}$ 
    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  $x = \Gamma''$  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  $\Gamma$  e t P S c  $\Gamma'$   $\mathcal{S}' P' c'$  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'  $\Gamma \mathcal{S} P +_{\mathcal{S}}$ 
e  $\Gamma' \mathcal{S}' P'$ ], assumption+)
  apply (subgoal-tac [ tyenv-wellformed mds  $\Gamma \mathcal{S} P$  ]  $\implies$  tyenv-wellformed mds
 $\Gamma \mathcal{S} P +_{\mathcal{S}} e$ ) 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  $\implies$  pred P +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*:  $\vdash \Gamma, \mathcal{S}, P \{ \text{Await } b \text{ ca} \} \Gamma', \mathcal{S}', P'$

```

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

```

```

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

```

```

lemma evA-eq:
  assumes tyenv-eq:  $mem_1 =_{\Gamma} mem_2$ 
  assumes pred: pred P mem1
  assumes e-type:  $\Gamma \vdash_a e \in t$ 
  assumes subtype:  $t \leq_P (dma\text{-type } v)$ 
  assumes is-Low:  $dma\ mem_1\ v = Low$ 
  shows  $ev_A\ mem_1\ e = ev_A\ mem_2\ e$ 

```

```

proof(rule eval-vars-detA, clarify)
  fix x
  assume in-vars: x ∈ aexp-vars e
  have type-max (to-total Γ x) mem1 = Low
  proof –
    from subtype-sound[OF subtype] pred have type-max t mem1 ≤ dma mem1 v
      by(auto)
    with is-Low have type-max t mem1 = 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 mem1 x = mem2 x
    using tyenv-eq unfolding tyenv-eq-def by blast
qed

```

```

lemma evA-eq':
  assumes tyenv-eq: mem1 =Γ mem2
  assumes pred: pred P mem1
  assumes e-type: Γ ⊢a e ∈ t
  assumes subtype: P ⊢ t
  shows evA mem1 e = evA mem2 e
proof(rule eval-vars-detA, clarify)
  fix x
  assume in-vars: x ∈ aexp-vars e
  have type-max (to-total Γ x) mem1 = Low
  proof –
    from subtype pred have type-max t mem1 ≤ Low
      by(auto simp: type-max-def pred-entailment-def pred-def)
    hence type-max t mem1 = 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 mem1 x = mem2 x
    using tyenv-eq unfolding tyenv-eq-def by blast
qed

```

```

lemma evB-eq':
  assumes tyenv-eq: mem1 =Γ mem2
  assumes pred: pred P mem1
  assumes e-type: Γ ⊢b e ∈ t
  assumes subtype: P ⊢ t
  shows evB mem1 e = evB mem2 e
proof(rule eval-vars-detB, clarify)
  fix x
  assume in-vars: x ∈ bexp-vars e

```

**have** *type-max* (to-total  $\Gamma$   $x$ )  $mem_1 = Low$   
**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*:

$\langle c, mds, mem \rangle \mathcal{R}^1_{\Gamma', \mathcal{S}', P'} \langle c', mds', mem' \rangle \implies$   
*context-equiv*  $\Gamma' P' \Gamma'' \implies P' \vdash P'' \implies$   
 $\forall mds. \text{tyenv-wellformed } mds \Gamma' \mathcal{S}' P' \longrightarrow \text{tyenv-wellformed } mds \Gamma'' \mathcal{S}' P'' \implies$   
 $\langle c, mds, mem \rangle \mathcal{R}^1_{\Gamma'', \mathcal{S}', P''} \langle c', mds', mem' \rangle$   
**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*:

$\langle c, mds, mem \rangle \mathcal{R}^3_{\Gamma', \mathcal{S}', P'} \langle c', mds', mem' \rangle \implies$   
*context-equiv*  $\Gamma' P' \Gamma'' \implies P' \vdash P'' \implies$   
 $\forall mds. \text{tyenv-wellformed } mds \Gamma' \mathcal{S}' P' \longrightarrow \text{tyenv-wellformed } mds \Gamma'' \mathcal{S}' P'' \implies$   
 $\langle c, mds, mem \rangle \mathcal{R}^3_{\Gamma'', \mathcal{S}', P''} \langle c', mds', mem' \rangle$   
**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*:

$lc_1 \mathcal{R}^u_{\Gamma', \mathcal{S}', P'} lc_2 \implies$   
*context-equiv*  $\Gamma' P' \Gamma'' \implies P' \vdash P'' \implies$   
 $\forall mds. \text{tyenv-wellformed } mds \Gamma' \mathcal{S}' P' \longrightarrow \text{tyenv-wellformed } mds \Gamma'' \mathcal{S}' P'' \implies$   
 $lc_1 \mathcal{R}^u_{\Gamma'', \mathcal{S}', P''} lc_2$   
**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*  $\Gamma$  *mem* *mem'*  $\implies$  *context-equiv*  $\Gamma$  *P*  $\Gamma'$   $\implies$  *pred P mem*  $\implies$  *tyenv-eq*  $\Gamma'$  *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** *R-typed-step-no-await*:

$\llbracket \vdash \Gamma, \mathcal{S}, P \{ c_1 \} \Gamma', \mathcal{S}', P' ;$

*tyenv-wellformed* *mds*  $\Gamma$   $\mathcal{S}$  *P*; *mem*<sub>1</sub> = <sub>$\Gamma$</sub>  *mem*<sub>2</sub>; *pred P mem*<sub>1</sub>;

*pred P mem*<sub>2</sub>; *tyenv-sec* *mds*  $\Gamma$  *mem*<sub>1</sub>;

$\langle c_1, \text{mds}, \text{mem}_1 \rangle \rightsquigarrow \langle c_1', \text{mds}', \text{mem}_1 \wedge \rangle$ ; *no-await* *c*<sub>1</sub>  $\rrbracket \implies$

$(\exists c_2' \text{mem}_2'. \langle c_1, \text{mds}, \text{mem}_2 \rangle \rightsquigarrow \langle c_2', \text{mds}', \text{mem}_2 \wedge \rangle \wedge$   
 $\langle c_1', \text{mds}', \text{mem}_1 \wedge \rangle \mathcal{R}_{\Gamma', \mathcal{S}', P'}^u \langle c_2', \text{mds}', \text{mem}_2 \wedge \rangle)$

**proof** (*induct arbitrary: mds c<sub>1</sub>' rule: has-type.induct*)

**case** (*seq-type*  $\Gamma$   $\mathcal{S}$  *P* *c*<sub>1</sub>  $\Gamma''$   $\mathcal{S}''$   $P''$  *c*<sub>2</sub>  $\Gamma'$   $\mathcal{S}'$   $P'$  *mds*)

**show** *?case*

**proof** (*cases c<sub>1</sub> = Stop*)

**assume** *c*<sub>1</sub> = *Stop*

**hence** [*simp*]: *c*<sub>1</sub>' = *c*<sub>2</sub> *mds'* = *mds* *mem*<sub>1</sub>' = *mem*<sub>1</sub>

**using** *seq-type*

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

**from** *seq-type*  $\langle c_1 = \text{Stop} \rangle$  **have** *context-equiv*  $\Gamma$  *P*  $\Gamma''$  **and**  $\mathcal{S} = \mathcal{S}''$  **and**  $P \vdash P''$  **and**

$(\forall \text{mds}. \text{tyenv-wellformed } \text{mds } \Gamma \mathcal{S} P \longrightarrow \text{tyenv-wellformed}$

*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**  $\langle c_2, \text{mds}, \text{mem}_1 \rangle \mathcal{R}_{\Gamma', \mathcal{S}', P'}^1 \langle c_2, \text{mds}, \text{mem}_2 \rangle$

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

**apply**(*rule*  $\langle \vdash \Gamma, \mathcal{S}, P \{ c_2 \} \Gamma', \mathcal{S}', P' \rangle$ )

**using** *seq-type* **by** *auto*

**thus** *?case*

**using**  $\mathcal{R}.\text{intro}_1$

**apply** *clarify*

**apply** (*rule-tac* *x = c<sub>2</sub>* **in** *exI*)

**apply** (*rule-tac* *x = mem<sub>2</sub>* **in** *exI*)

**by** (*auto simp:*  $\langle c_1 = \text{Stop} \rangle$  *seq-stop-eval<sub>w</sub>*  $\mathcal{R}.\text{intro}_1$ )

**next**

**assume** *c*<sub>1</sub>  $\neq$  *Stop*

**with**  $\langle c_1 ;; c_2, \text{mds}, \text{mem}_1 \rangle \rightsquigarrow \langle c_1', \text{mds}', \text{mem}_1 \wedge \rangle$  **obtain** *c*<sub>1</sub>'' **where** *c*<sub>1</sub>''-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**  $\langle no-await (c_1 ;; c_2) \rangle$  **have** *no-await*  $c_1$  **using** *no-await.cases* **by** *blast*  
**with** *seq-type(2)*  $\langle no-await c_1 \rangle$  **obtain**  $c_2'' mem_2'$  **where**  $c_2''$ -props:  
 $\langle c_1, mds, mem_2 \rangle \rightsquigarrow \langle c_2'', mds', mem_2 \rangle \wedge \langle c_1'', mds', mem_1 \rangle \mathcal{R}^u_{\Gamma'', S'', P''}$   
 $\langle c_2'', mds', mem_2 \rangle$   
**using** *seq-type.prem(1)* *seq-type.prem(2)* *seq-type.prem(3)* *seq-type.prem(4)*  
*seq-type.prem(5)*  $c_1''$ -props  
**by** *blast*  
**hence**  $\langle c_1'' ;; c_2, mds', mem_1 \rangle \mathcal{R}^u_{\Gamma', S', P'} \langle c_2'' ;; c_2, mds', mem_2 \rangle$   
**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**  $c_2''$ -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  $\Gamma' x$ ) mem<sub>1</sub> = Low*  
**hence** *type-max (to-total  $\Gamma x$ ) mem<sub>1</sub> = Low*  
**proof** –  
**from**  $\langle pred P mem_1 \rangle$  **have**  $pred P' mem_1$   
**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**  $mem_1 x = mem_2 x$   
**using** *anno-type.prem(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  $\Gamma$   $\mathcal{S}$   $P \wedge \text{pred } P \text{ mem}_1 \wedge \text{tyenv-sec } \text{mds } \Gamma \text{ mem}_1 \longrightarrow$ 

  tyenv-sec (update-modes upd mds)  $\Gamma'$  mem1
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  $\langle \text{no-await } (c@[upd]) \rangle$  have no-await c using no-await.cases by blast
ultimately obtain  $c_2' \text{ mem}_2'$  where  $(\langle c, \text{update-modes upd mds, mem}_2 \rangle \rightsquigarrow \langle c_2',$ 
 $\text{mds}', \text{mem}_2' \rangle \wedge$ 
 $\langle c_1', \text{mds}', \text{mem}_1 \rangle \mathcal{R}^u_{\Gamma'', \mathcal{S}'', P''} \langle c_2', \text{mds}', \text{mem}_2' \rangle)$ 
  using anno-type
  apply auto
using  $\langle \text{mem}_1 =_{\Gamma'} \text{mem}_2 \rangle$  local.pred-def restrict-preds-to-vars-def upd-elim  $\langle \text{no-await}$ 
 $c \rangle$ 

  using  $\langle \text{tyenv-wellformed } \text{mds } \Gamma \mathcal{S} P \wedge \text{pred } P \text{ mem}_1 \wedge (\forall x \in \text{dom } \Gamma. x \notin$ 
 $\text{mds } \text{AsmNoReadOrWrite} \longrightarrow \text{type-max } (\text{the } (\Gamma x)) \text{ mem}_1 \leq \text{dma mem}_1 x) \longrightarrow$ 
 $(\forall x \in \text{dom } \Gamma'. x \notin \text{update-modes upd mds } \text{AsmNoReadOrWrite} \longrightarrow \text{type-max } (\text{the}$ 
 $(\Gamma' x)) \text{ mem}_1 \leq \text{dma mem}_1 x) \rangle$  mem-Collect-eq by fastforce try0
  thus ?case
    apply (rule-tac  $x = c_2'$  in exI)
    apply (rule-tac  $x = \text{mem}_2'$  in exI)
    apply auto
    by (metis cxt-to-stmt.simps(1) evalw.decl)
next
  case stop-type
  with stop-no-eval show ?case by auto
next
  case (skip-type  $\Gamma \mathcal{S} P \text{ mds}$ )
  moreover
  with skip-type have [simp]:  $\text{mds}' = \text{mds } c_1' = \text{Stop mem}_1' = \text{mem}_1$ 
    using skip-elim
    by (metis, metis, metis)
  with skip-type have  $\langle \text{Stop, mds, mem}_1 \rangle \mathcal{R}^1_{\Gamma, \mathcal{S}, P} \langle \text{Stop, mds, mem}_2 \rangle$ 
    by auto
  thus ?case
    using  $\mathcal{R}.\text{intro}_1$  and unannotated [where  $c = \text{Skip}$  and  $E = []$ ]
    apply auto
    by (metis (mono-tags, lifting)  $\mathcal{R}.\text{intro}_1$  old.prod.case skip-evalw)
next

```

```

case (assign1 x Γ e t P P' S mds)
hence upd [simp]: c1' = Stop mds' = mds mem1' = mem1 (x := evA mem1 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 := evA mem1 e) =Γ mem2 (x := evA mem2 e)
unfolding tyenv-eq-def
proof (clarify)
  fix v
  assume is-Low': type-max (to-total Γ v) (mem1(x := evA mem1 e)) = Low
  show (mem1(x := evA mem1 e)) v = (mem2(x := evA 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 evA-eq)
  apply(rule assign1(7))
  apply(rule assign1(8))
  apply(rule assign1(3))
  apply(rule assign1(4))
using dma-v-Low by simp
qed
qed
qed

moreover have tyenv-wellformed mds  $\Gamma \mathcal{S} P \longrightarrow$  tyenv-wellformed mds'  $\Gamma \mathcal{S} P'$ 
using upd tyenv-wellformed-preds-update assign1 by metis
moreover have pred P mem1  $\longrightarrow$  pred P' mem1'
using pred-preds-update assign1 upd by metis

moreover have pred P mem2  $\longrightarrow$  pred P' (mem2(x := evA mem2 e))
using pred-preds-update assign1 upd by metis

moreover have tyenv-wellformed mds  $\Gamma \mathcal{S} P \wedge$  tyenv-sec mds  $\Gamma$  mem1  $\longrightarrow$ 
tyenv-sec mds  $\Gamma$  mem1'
using tyenv-sec-eq[OF C-eq, where  $\Gamma=\Gamma$ ]
unfolding tyenv-wellformed-def by blast

ultimately have  $\mathcal{R}'$ :
 $\langle \text{Stop}, \text{mds}', \text{mem}_1(x := \text{ev}_A \text{ mem}_1 e) \rangle \mathcal{R}_{\Gamma, \mathcal{S}, P'}^u \langle \text{Stop}, \text{mds}', \text{mem}_2(x := \text{ev}_A$ 
mem2 e)  $\rangle$ 
apply -
apply (rule  $\mathcal{R}.\text{intro}_1$ , auto simp: assign1 simp del: dma-eq)
done

have  $a: \langle x \leftarrow e, \text{mds}, \text{mem}_2 \rangle \rightsquigarrow \langle \text{Stop}, \text{mds}', \text{mem}_2(x := \text{ev}_A \text{ mem}_2 e) \rangle$ 
by (auto, metis cxt-to-stmt.simps(1) evalw.unannotated evalw-simple.assign)

from  $\mathcal{R}'$  a show ?case
using  $\langle c_1' = \text{Stop} \rangle$  and  $\langle \text{mem}_1' = \text{mem}_1(x := \text{ev}_A \text{ mem}_1 e) \rangle$ 
by blast
next
case (assignC x  $\Gamma$  e t P P'  $\mathcal{S}$  mds)
hence upd [simp]: c1' = Stop mds' = mds mem1' = mem1(x := evA mem1 e)
using assign-elim
by (auto, metis)
have  $\text{mem}_1(x := \text{ev}_A \text{ mem}_1 e) =_{\Gamma} \text{mem}_2(x := \text{ev}_A \text{ mem}_2 e)$ 
unfolding tyenv-eq-def
proof(clarify)
fix  $v$ 
assume is-Low': type-max (to-total  $\Gamma$  v) (mem1(x := evA mem1 e)) = Low
show  $(\text{mem}_1(x := \text{ev}_A \text{ mem}_1 e)) v = (\text{mem}_2(x := \text{ev}_A \text{ mem}_2 e)) v$ 
proof(cases v  $\in$  dom  $\Gamma$ )

```

```

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

with  $x \text{nin-} C$  have f-eq: type-max  $t' \text{ mem}_1 = \text{type-max } t' \text{ mem}_1'$ 
  using vars-of-type-eq-type-max-eq by simp
with  $\text{is-Low}'$  have is-Low: type-max  $(\text{to-total } \Gamma v) \text{ mem}_1 = \text{Low}$ 
  by simp
from assignC(1)  $\langle v \in \text{dom } \Gamma \rangle$  assignC(7) have  $x \neq v$ 
  by(auto simp: tyenv-wellformed-def mds-consistent-def)
thus ?thesis
  apply simp
  using is-Low assignC(8) unfolding tyenv-eq-def by auto
next
assume nin-dom:  $v \notin \text{dom } \Gamma$ 
hence [simp]:  $\bigwedge \text{mem. type-max } (\text{to-total } \Gamma v) \text{ mem} = \text{dma mem } v$ 
  unfolding to-total-def by simp
with  $\text{is-Low}'$  have dma mem1'  $v = \text{Low}$  by simp
show ?thesis
proof(cases  $x = v$ )
  assume  $x = v$ 
  thus ?thesis
  apply simp
  apply(rule evA-eq')
  apply(rule assignC(8))
  apply(rule assignC(9))
  apply(rule assignC(2))
  by(rule assignC(3))
next
assume [simp]:  $x \neq v$ 
show ?thesis
proof(cases  $x \in \mathcal{C}\text{-vars } v$ )
  assume in- $\mathcal{C}$ -vars:  $x \in \mathcal{C}\text{-vars } v$ 
  hence  $v \notin \mathcal{C}$ 
  using  $\mathcal{C}$ -vars- $\mathcal{C}$  by auto
with nin-dom have  $v \notin \text{snd } \mathcal{S}$ 
  using assignC(7)
  by(auto simp: tyenv-wellformed-def mds-consistent-def stable-def)
with in- $\mathcal{C}$ -vars have  $P \vdash (\text{to-total } \Gamma v)$ 
  using assignC(6) by blast
with assignC(9) have type-max  $(\text{to-total } \Gamma v) \text{ mem}_1 = \text{Low}$ 
  by(auto simp: type-max-def pred-def pred-entailment-def)

```

```

thus ?thesis
  using not-sym[OF ‹ $x \neq v$ ›]
  apply simp
  using assignC(8)
  unfolding tyenv-eq-def by auto
next
  assume  $x \notin \mathcal{C}\text{-vars } v$ 
  with is-Low' have dma mem1 v = Low
    using dma-C-vars ‹ $\wedge \text{mem. type-max (to-total } \Gamma v) \text{ mem} = \text{dma mem } v$ ›
    by (metis fun-upd-other)
  thus ?thesis
    using not-sym[OF ‹ $x \neq v$ ›]
    apply simp
    using assignC(8)
    unfolding tyenv-eq-def by auto
qed
qed
qed
qed

moreover have tyenv-wellformed mds  $\Gamma \mathcal{S} P \longrightarrow$  tyenv-wellformed mds'  $\Gamma \mathcal{S} P'$ 
  using upd tyenv-wellformed-preds-update assignC by metis
moreover have pred P mem1  $\longrightarrow$  pred P' mem1'
  using pred-preds-update assignC upd by metis
moreover have pred P mem2  $\longrightarrow$  pred P' (mem2(x := evA mem2 e))
  using pred-preds-update assignC upd by metis
moreover have tyenv-wellformed mds  $\Gamma \mathcal{S} P \wedge$  pred P mem1  $\wedge$  tyenv-sec mds
 $\Gamma \text{ mem}_1 \Longrightarrow$  tyenv-sec mds'  $\Gamma \text{ mem}_1'$ 
proof (clarify)
  fix v t'
  assume wf: tyenv-wellformed mds  $\Gamma \mathcal{S} P$ 
  assume pred: pred P mem1
  hence pred': pred P' mem1' using ‹pred P mem1  $\longrightarrow$  pred P' mem1'› by blast
  assume sec: tyenv-sec mds  $\Gamma \text{ mem}_1$ 
  assume  $\Gamma v$ :  $\Gamma v = \text{Some } t'$ 
  assume readable':  $v \notin \text{mds}' \text{ AsmNoReadOrWrite}$ 
  with upd have readable:  $v \notin \text{mds} \text{ AsmNoReadOrWrite}$  by simp
  with wf have  $v \notin \text{snd } \mathcal{S}$  by (auto simp: tyenv-wellformed-def mds-consistent-def)
  show type-max (the ( $\Gamma v$ )) mem1'  $\leq$  dma mem1' v
  proof (cases  $x \in \mathcal{C}\text{-vars } v$ )
    assume  $x \in \mathcal{C}\text{-vars } v$ 
    with assignC(6) ‹ $v \notin \text{snd } \mathcal{S}$ › have (to-total  $\Gamma v$ )  $\leq_{P'}$  (dma-type v) by blast
    from pred'  $\Gamma v$  subtype-correct this show ?thesis
    using type-max-dma-type by (auto simp: to-total-def split: if-splits)
  next
    assume  $x \notin \mathcal{C}\text{-vars } v$ 
    hence  $\forall v' \in \mathcal{C}\text{-vars } v. \text{ mem}_1 v' = \text{ mem}_1' v'$ 
    using upd by auto
    hence dma-eq: dma mem1 v = dma mem1' v

```

```

    by(rule dma-C-vars)
  from  $\Gamma v$  assignC(4) have  $x \notin \text{vars-of-type } t'$  by force
  have type-wellformed  $t'$ 
    using wf  $\Gamma v$  by(force simp: tyenv-wellformed-def types-wellformed-def)
  with  $\langle x \notin \text{vars-of-type } t' \rangle$  upd have f-eq: type-max  $t'$  mem1 = type-max  $t'$ 
mem1'
    using vars-of-type-eq-type-max-eq by fastforce
  from sec  $\Gamma v$  readable have type-max  $t'$  mem1 ≤ dma mem1 v
    by auto
  with f-eq dma-eq  $\Gamma v$  show ?thesis
    by simp
qed
qed

ultimately have  $\mathcal{R}'$ :
   $\langle \text{Stop}, \text{mds}', \text{mem}_1 (x := \text{ev}_A \text{ mem}_1 e) \rangle \mathcal{R}_{\Gamma, \mathcal{S}, P'}^u \langle \text{Stop}, \text{mds}', \text{mem}_2 (x := \text{ev}_A$ 
mem2 e)  $\rangle$ 
  apply –
  apply (rule  $\mathcal{R}.$ intro1, auto simp: assignC)
  done

have a:  $\langle x \leftarrow e, \text{mds}, \text{mem}_2 \rangle \rightsquigarrow \langle \text{Stop}, \text{mds}', \text{mem}_2 (x := \text{ev}_A \text{ mem}_2 e) \rangle$ 
by (auto, metis cxt-to-stmt.simps(1) evalw.unannotated evalw-simple.assign)

from  $\mathcal{R}'$  a show ?case
  using  $\langle c_1' = \text{Stop} \rangle$  and  $\langle \text{mem}_1' = \text{mem}_1 (x := \text{ev}_A \text{ mem}_1 e) \rangle$ 
  by blast
next
case (assign2  $x \Gamma e t \mathcal{S} P' P \text{ mds}$ )
have upd [simp]:  $c_1' = \text{Stop}$   $\text{mds}' = \text{mds}$   $\text{mem}_1' = \text{mem}_1 (x := \text{ev}_A \text{ mem}_1 e)$ 
  using assign-elim[OF assign2(11)]
  by auto
from  $\langle x \in \text{dom } \Gamma \rangle \langle \text{tyenv-wellformed } \text{mds } \Gamma \mathcal{S} P \rangle$ 
have x-nin-C:  $x \notin \mathcal{C}$ 
  by(auto simp: tyenv-wellformed-def mds-consistent-def)
hence dma-eq [simp]: dma mem1' = dma mem1
  using dma-C assign2
  by auto

let ? $\Gamma'$  =  $\Gamma (x \mapsto t)$ 
have  $\langle x \leftarrow e, \text{mds}, \text{mem}_2 \rangle \rightsquigarrow \langle \text{Stop}, \text{mds}, \text{mem}_2 (x := \text{ev}_A \text{ mem}_2 e) \rangle$ 
  using assign2
  by (metis cxt-to-stmt.simps(1) evalw-simplep.assign evalwp.unannotated evalwp-evalw-eq)

moreover
have tyenv-eq': mem1( $x := \text{ev}_A \text{ mem}_1 e$ ) = $\Gamma(x \mapsto t)$  mem2( $x := \text{ev}_A \text{ mem}_2 e$ )
unfolding tyenv-eq-def
proof(clarify)
  fix v

```

```

assume is-Low': type-max (to-total ( $\Gamma(x \mapsto t)$ ) v) (mem1(x := evA mem1 e))
= Low
show (mem1(x := evA mem1 e) v = (mem2(x := evA mem2 e) v)
proof(cases v = x)
  assume neq: v  $\neq$  x
  hence type-max (to-total  $\Gamma$  v) mem1 = 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  $\Gamma$  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' mem1 = type-max t' mem1'
      apply(rule C-eq-type-max-eq)
      using assign2(6)
      apply(clarsimp simp: tyenv-wellformed-def types-wellformed-def)
      using  $\langle v \in \text{dom } \Gamma \rangle \langle \Gamma v = \text{Some } t' \rangle$  apply(metis option.sel)
      using x-nin-C by simp
    from this is-Low' neq neq[THEN not-sym] show type-max (to-total  $\Gamma$  v)
mem1 = Low
      by auto
  next
    assume v  $\notin$  dom  $\Gamma$ 
    with is-Low' neq
    have dma mem1' v = Low
      by(auto simp: to-total-def split: if-splits)
    with dma-eq  $\langle v \notin \text{dom } \Gamma \rangle$  show ?thesis
      by(auto simp: to-total-def split: if-splits)
  qed
with neq assign2(7) show (mem1(x := evA mem1 e) v = (mem2(x := evA
mem2 e) v)
  by(auto simp: tyenv-eq-def)
next
  assume eq[simp]: v = x
  with is-Low'  $\langle x \in \text{dom } \Gamma \rangle$  have t-Low': type-max t mem1' = Low
    by(auto simp: to-total-def split: if-splits)
  have wf-t: type-wellformed t
    using type-aexpr-type-wellformed assign2(2) assign2(6)
    by(fastforce simp: tyenv-wellformed-def)
  with t-Low'  $\langle x \notin \mathcal{C} \rangle$  have t-Low: type-max t mem1 = Low
    using C-eq-type-max-eq
    by (metis (no-types, lifting) fun-upd-other upd(3))
  show ?thesis
proof(simp, rule eval-vars-detA, clarify)
  fix y
  assume in-vars: y  $\in$  aexp-vars e
  have type-max (to-total  $\Gamma$  y) mem1 = 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 := evA 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  $v \notin mds' \text{ AsmNoReadOrWrite}$ 
show  $\text{type-max (the ((}\Gamma(x \mapsto t)) v)) \text{ mem}_1' \leq \text{dma mem}_1' v$ 
proof ( $\text{cases } v = x$ )
  assume  $[simp]: v = x$ 
  hence  $[simp]: (\text{the ((}\Gamma(x \mapsto t)) v)) = t$  and  $t\text{-def}: t = t'$ 
    using  $\Gamma v$  by  $\text{auto}$ 
  from  $\langle v \notin mds' \text{ AsmNoReadOrWrite} \rangle \text{ upd wf}$  have  $\text{readable}: v \notin \text{snd } \mathcal{S}$ 
    by ( $\text{auto simp: tyenv-wellformed-def mds-consistent-def}$ )
  with  $\text{assign}_2(5)$  have  $t \leq_{P'} (\text{dma-type } x)$  by  $\text{fastforce}$ 
  with  $\text{pred}'$  show  $?thesis$ 
    using  $\text{type-max-dma-type subtype-correct}$ 
    by  $\text{fastforce}$ 
next
  assume  $\text{neq}: v \neq x$ 
  hence  $[simp]: ((\Gamma(x \mapsto t)) v) = \Gamma v$ 
    by  $\text{simp}$ 
  with  $\Gamma v$  have  $\Gamma v: \Gamma v = \text{Some } t'$  by  $\text{simp}$ 
  with  $\text{sec upd } \langle v \notin mds' \text{ AsmNoReadOrWrite} \rangle$  have  $f\text{-leq}: \text{type-max } t' \text{ mem}_1$ 
 $\leq \text{dma mem}_1 v$ 
    by  $\text{auto}$ 
  have  $\mathcal{C}\text{-eq}: \forall x \in \mathcal{C}. \text{mem}_1 x = \text{mem}_1' x$ 
    using  $\text{wf assign}_2(1) \text{ upd}$  by ( $\text{auto simp: tyenv-wellformed-def mds-consistent-def}$ )
  hence  $\text{dma-eq}: \text{dma mem}_1 = \text{dma mem}_1'$ 
    by ( $\text{rule dma-C}$ )
  have  $f\text{-eq}: \text{type-max } t' \text{ mem}_1 = \text{type-max } t' \text{ mem}_1'$ 
    apply ( $\text{rule C-eq-type-max-eq}$ )
    using  $\Gamma v \text{ wf}$  apply ( $\text{force simp: tyenv-wellformed-def types-wellformed-def}$ )
    by ( $\text{rule C-eq}$ )
  from  $\text{neq } \Gamma v f\text{-leq dma-eq } f\text{-eq}$  show  $?thesis$ 
    by  $\text{simp}$ 
qed
qed

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

  ultimately have  $\langle x \leftarrow e, mds, \text{mem}_2 \rangle \rightsquigarrow \langle \text{Stop}, mds', \text{mem}_2 (x := \text{ev}_A \text{ mem}_2$ 
 $e) \rangle$ 
     $\langle \text{Stop}, mds', \text{mem}_1 (x := \text{ev}_A \text{ mem}_1 e) \rangle \mathcal{R}^u_{\Gamma(x \mapsto t), \mathcal{S}, P'} \langle \text{Stop}, mds', \text{mem}_2 (x$ 
 $:= \text{ev}_A \text{ mem}_2 e) \rangle$ 
    using  $\mathcal{R}.\text{intro}_1$ 

```

```

    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 ⟨⟨Stmnt.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.premis(6) apply blast
    apply (subgoal-tac c1' = 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.premis(6) by blast
    with ⟨c1', mds, mem1⟩  $\mathcal{R}^u_{\Gamma''', \mathcal{S}', P'''} \langle c1', mds, mem2 \rangle$  show ?thesis
    by (metis if-elim if-type.premis(6))
qed
next
case (while-type  $\Gamma e t P \mathcal{S} c$ )
hence [simp]: c1' = (If e (c ;; While e c) Stop) and
[simp]: mds' = mds and
[simp]: mem1' = mem1
by (auto simp: while-elim)

with while-type have ⟨While e c, mds, mem2⟩  $\rightsquigarrow$  ⟨c1', mds, mem2⟩
by (metis cxt-to-stmt.simps(1) eval_w-simplep.while eval_w.p.unannotated eval_w.p-eval_w-eq)

moreover have ty:  $\vdash \Gamma, \mathcal{S}, P \{c1'\} \Gamma, \mathcal{S}, P$ 
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 ⟨c1', mds, mem1⟩  $\mathcal{R}^1_{\Gamma, \mathcal{S}, P} \langle c1', mds, mem2 \rangle$ 
    apply (rule  $\mathcal{R}_1.intro$  [where  $\Gamma = \Gamma$ ])
    apply (rule ty)
    using while-type apply simp+
done
hence ⟨c1', mds, mem1⟩  $\mathcal{R}^u_{\Gamma, \mathcal{S}, P} \langle c1', mds, mem2 \rangle$ 
    using  $\mathcal{R}.intro_1$  by auto
ultimately show ?case
    by fastforce
next

```

**case** (*sub*  $\Gamma_1 \mathcal{S} P_1 c \Gamma_1' \mathcal{S}' P_1' \Gamma_2 P_2 \Gamma_2' P_2' \text{ mds } c_1'$ )  
**have** *imp*: *tyenv-wellformed* *mds*  $\Gamma_2 \mathcal{S} P_2 \wedge \text{pred } P_2 \text{ mem}_1 \wedge \text{pred } P_2 \text{ mem}_2 \wedge$   
*tyenv-sec* *mds*  $\Gamma_2 \text{ mem}_1 \implies$   
*tyenv-wellformed* *mds*  $\Gamma_1 \mathcal{S} P_1 \wedge \text{pred } P_1 \text{ mem}_1 \wedge \text{pred } P_1 \text{ mem}_2 \wedge$   
*tyenv-sec* *mds*  $\Gamma_1 \text{ mem}_1$   
**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*:  $\text{mem}_1 =_{\Gamma_1} \text{mem}_2$   
**using** *context-equiv-tyenv-eq sub* **by** *blast*  
  
**from** *imp tyenv-eq* **obtain**  $c_2' \text{ mem}_2'$  **where** *c\_2'-props*:  $\langle c, \text{mds}, \text{mem}_2 \rangle \rightsquigarrow \langle c_2',$   
*mds', mem\_2' \rangle  
 $\langle c_1', \text{mds}', \text{mem}_1' \rangle \mathcal{R}^u_{\Gamma_1', \mathcal{S}', P_1'} \langle c_2', \text{mds}', \text{mem}_2' \rangle$   
**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*  $\Gamma e t P \mathcal{S} c \Gamma' \mathcal{S}' P' \Gamma'' P''$ )  
**from** *this* **show** *?case* **using** *no-await-no-await* **by** *blast*  
**qed***

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

**lemma** *pred-plus-impl*:  
 $\text{pred } P \text{ mem} \implies \text{ev}_B \text{ mem } e \implies \text{pred } P +_{\mathcal{S}} e \text{ mem}$   
**unfolding** *add-pred-def pred-def* **by** *simp*

**lemma** *my- $\mathcal{R}_3$ -aux-induct* [*consumes 1, case-names intro<sub>1</sub> intro<sub>3</sub>*]:  
 $\llbracket \langle c_1, \text{mds}, \text{mem}_1 \rangle \mathcal{R}^3_{\Gamma, \mathcal{S}, P} \langle c_2, \text{mds}, \text{mem}_2 \rangle;$   
 $\wedge c_1 \text{ mds mem}_1 \Gamma \mathcal{S} P c_2 \text{ mem}_2 c \Gamma' \mathcal{S}' P'.$   
 $\llbracket \langle c_1, \text{mds}, \text{mem}_1 \rangle \mathcal{R}^1_{\Gamma, \mathcal{S}, P} \langle c_2, \text{mds}, \text{mem}_2 \rangle;$   
 $\vdash \Gamma, \mathcal{S}, P \{c\} \Gamma', \mathcal{S}', P' \rrbracket \implies$   
 $Q (c_1 ;; c) \text{ mds mem}_1 \Gamma' \mathcal{S}' P' (c_2 ;; c) \text{ mds mem}_2;$   
 $\wedge c_1 \text{ mds mem}_1 \Gamma \mathcal{S} P c_2 \text{ mem}_2 c \Gamma' \mathcal{S}' P'.$   
 $\llbracket \langle c_1, \text{mds}, \text{mem}_1 \rangle \mathcal{R}^3_{\Gamma, \mathcal{S}, P} \langle c_2, \text{mds}, \text{mem}_2 \rangle;$   
 $Q c_1 \text{ mds mem}_1 \Gamma \mathcal{S} P c_2 \text{ mds mem}_2;$   
 $\vdash \Gamma, \mathcal{S}, P \{c\} \Gamma', \mathcal{S}', P' \rrbracket \implies$   
 $Q (c_1 ;; c) \text{ mds mem}_1 \Gamma' \mathcal{S}' P' (c_2 ;; c) \text{ mds mem}_2 \rrbracket \implies$   
 $Q c_1 \text{ mds mem}_1 \Gamma \mathcal{S} P c_2 \text{ mds mem}_2$   
**using**  *$\mathcal{R}_3$ -aux.induct*[**where**  
 $?x1.0 = \langle c_1, \text{mds}, \text{mem}_1 \rangle$ ] **and**

$?x2.0 = \Gamma$  **and**  
 $?x3.0 = \mathcal{S}$  **and**  
 $?x4.0 = P$  **and**  
 $?x5.0 = \langle c_2, mds, mem_2 \rangle$  **and**  
 $?P = \lambda ctx_1 \Gamma \mathcal{S} P ctx_2. Q (fst (fst ctx_1)) (snd (fst ctx_1)) (snd ctx_1) \Gamma \mathcal{S} P (fst (fst ctx_2)) (snd (fst ctx_2)) (snd ctx_2)]$   
**by force**

**lemma**  $\mathcal{R}$ -typed-step-plus:

$\llbracket \langle c_1, mds, mem_1 \rangle \rightsquigarrow^+ \langle c_1', mds', mem_1' \rangle;$   
 $\vdash \Gamma, \mathcal{S}, P \{ c_1 \} \Gamma', \mathcal{S}', P';$   
 $no\text{-}await\ c_1;$   
 $tyenv\text{-}wellformed\ mds\ \Gamma\ \mathcal{S}\ P;$   
 $mem_1 =_{\Gamma} mem_2;$   
 $pred\ P\ mem_1;$   
 $pred\ P\ mem_2;$   
 $tyenv\text{-}sec\ mds\ \Gamma\ mem_1 \rrbracket \implies$   
 $(\exists\ c_2' mem_2'. \langle c_1, mds, mem_2 \rangle \rightsquigarrow^+ \langle c_2', mds', mem_2' \rangle \wedge$   
 $\langle c_1', mds', mem_1' \rangle \mathcal{R}_{\Gamma', \mathcal{S}', P'}^u \langle c_2', mds', mem_2' \rangle)$

**proof** (induct arbitrary:  $\Gamma \mathcal{S} P mem_2$  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**  $step_2': \langle c_1, mds, mem_2 \rangle \rightsquigarrow^+ \langle c_1', mds', mem_2' \rangle$

**and**

$rel_2': \langle c_1', mds', mem_1' \rangle \mathcal{R}_{\Gamma', \mathcal{S}', P'}^u \langle c_1', mds', mem_2' \rangle$

**using** *bisim-simple- $\mathcal{R}_u$*  **by** (*metis fst-conv*)

**from**  $rel_2'$  **show**  $?case$

**proof** (*cases rule:  $\mathcal{R}$ .cases*)

**case** (*intro<sub>1</sub>*)

**from this obtain**  $\Gamma''\ \mathcal{S}''\ P''$  **where**

$\vdash \Gamma'', \mathcal{S}'', P'' \{ c_1' \} \Gamma', \mathcal{S}', P'$

$tyenv\text{-}wellformed\ mds'\ \Gamma''\ \mathcal{S}''\ P''$

$mem_1' =_{\Gamma''} mem_2'$

$pred\ P''\ mem_1'$

$pred\ P''\ mem_2'$

$\forall x \in dom\ \Gamma''. x \notin mds'\ AsmNoReadOrWrite \longrightarrow type\text{-}max\ (the\ (\Gamma''\ x))\ mem_1' \leq dma\ mem_1'\ x$

**using**  $\mathcal{R}_1.cases$  **by auto**

**from**  $step_2'$   $\langle no\text{-}await\ c_1 \rangle$   $step.hyps(1)$   $step.hyps(4)$  **this obtain**  $mem_2''$  **where**

$step_2'': \langle c_1', mds', mem_2' \rangle \rightsquigarrow^+ \langle c_1'', mds'', mem_2'' \rangle$  **and**

$rel_2'': \langle c_1'', mds'', mem_1'' \rangle \mathcal{R}_{\Gamma', \mathcal{S}', P'}^u \langle c_1'', mds'', mem_2'' \rangle$

**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<sub>3</sub>*)

**from**  $intro_3$   $step.premis$   $step.hyps(1)$   $step.hyps(3)$   $step.hyps(4)$  **obtain**  $mem_2''$

where

$step'' : \langle c_1', mds', mem_2' \rangle \rightsquigarrow^+ \langle c_1'', mds'', mem_2'' \rangle$  and  
 $rel'' : \langle c_1'', mds'', mem_1'' \rangle \mathcal{R}_{\Gamma', \mathcal{S}', P'}^u \langle c_1'', mds'', mem_2'' \rangle$

**proof** (induct arbitrary: rule: *my- $\mathcal{R}_3$ -aux-induct*)

**case** ( $intro_1$   $c_1'1'$   $mds'$   $mem_1'$   $\Gamma'''$   $\mathcal{S}'''$   $P'''$   $c_1'1$   $mem_2'$   $c_1'2$   $\Gamma''$   $\mathcal{S}''$   $P''$ )

**from**  $intro_1(1)$  **obtain**  $\Gamma v$   $\mathcal{S} v$   $P v$  **where** pre-props:

$\vdash \Gamma v, \mathcal{S} v, P v \{c_1'1\} \Gamma''', \mathcal{S}''', P'''$   
*tyenv-wellformed*  $mds' \Gamma v \mathcal{S} v P v$   
 $mem_1' =_{\Gamma v} mem_2'$   
 $pred P v mem_1'$   
 $pred P v mem_2'$   
 $c_1'1 = c_1'1'$

$\forall x \in dom \Gamma v. x \notin mds' \text{ AsmNoReadOrWrite} \longrightarrow type\text{-max (the } (\Gamma v x)) mem_1'$   
 $\leq dma mem_1' x$

**using**  $\mathcal{R}_1.cases$  **by** *blast*

**from** *this*  $intro_1$  **have** *typed*:  $\vdash \Gamma v, \mathcal{S} v, P v \{c_1'1 ;; c_1'2\} \Gamma'', \mathcal{S}'', P''$

**using** *has-type.seq-type* **by** *blast*

**from** *this* pre-props  $\langle no\text{-await } c_1 \rangle \langle \langle c_1, mds, mem_1 \rangle \rightsquigarrow^+ \langle c_1'1 ;; c_1'2, mds', mem_1' \rangle \rangle$   $intro_1(13)$

**obtain**  $mem_2''$  **where**

*step*:  $\langle c_1'1 ;; c_1'2, mds', mem_2' \rangle \rightsquigarrow^+ \langle c_1'', mds'', mem_2'' \rangle \wedge$   
 $\langle c_1'', mds'', mem_1'' \rangle \mathcal{R}_{\Gamma'', \mathcal{S}'', P''}^u \langle c_1'', mds'', mem_2'' \rangle$

**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** ( $intro_3$   $c_1'1'$   $mds'$   $mem_1'$   $\Gamma'''$   $\mathcal{S}'''$   $P'''$   $c_1'1$   $mem_2'$   $c_1'2$   $\Gamma''$   $\mathcal{S}''$   $P''$ )

**from**  $intro_3(1)$  **obtain**  $\Gamma v$   $\mathcal{S} v$   $P v$  **where** pre-props:

$\vdash \Gamma v, \mathcal{S} v, P v \{c_1'1\} \Gamma''', \mathcal{S}''', P'''$   
*tyenv-wellformed*  $mds' \Gamma v \mathcal{S} v P v$   
 $mem_1' =_{\Gamma v} mem_2'$   
 $pred P v mem_1'$   
 $pred P v mem_2'$   
 $c_1'1 = c_1'1'$

$\forall x \in dom \Gamma v. x \notin mds' \text{ AsmNoReadOrWrite} \longrightarrow type\text{-max (the } (\Gamma v x)) mem_1'$   
 $\leq dma mem_1' x$

**by** (*induct arbitrary: rule: my- $\mathcal{R}_3$ -aux-induct*)

(*blast elim:  $\mathcal{R}_1.cases$ , blast*)

**from** *this*  $intro_1$  **have** *typed*:  $\vdash \Gamma v, \mathcal{S} v, P v \{c_1'1 ;; c_1'2\} \Gamma'', \mathcal{S}'', P''$

**using** *has-type.seq-type intro\_3.hyps(3)* **by** *blast*

**from** *this* pre-props  $\langle no\text{-await } c_1 \rangle \langle \langle c_1, mds, mem_1 \rangle \rightsquigarrow^+ \langle c_1'1 ;; c_1'2, mds', mem_1' \rangle \rangle$   $intro_3$

**obtain**  $mem_2''$  **where**

*step*:  $\langle c_1'1 ;; c_1'2, mds', mem_2' \rangle \rightsquigarrow^+ \langle c_1'', mds'', mem_2'' \rangle \wedge$   
 $\langle c_1'', mds'', mem_1'' \rangle \mathcal{R}_{\Gamma'', \mathcal{S}'', P''}^u \langle c_1'', mds'', mem_2'' \rangle$

**proof** –

**assume**  $a1 : \bigwedge mem_2''. \langle c_1'1 ;; c_1'2, mds', mem_2' \rangle \rightsquigarrow^+ \langle c_1'', mds'', mem_2'' \rangle$

$\wedge$

$\langle c_1'', mds'', mem_1'' \rangle \mathcal{R}_{\Gamma'', \mathcal{S}'', P''}^u \langle c_1'', mds'', mem_2'' \rangle$

$\implies$  *thesis*

**thus** *?thesis* **using** *intro3.prem*(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*

*c1>*

*bisim-simple- $\mathcal{R}_u$  fst-conv no-await-trancl typed*)

**qed**

**from** *this* *intro3* **show** *?case* **using** *no-await-trancl bisim-simple- $\mathcal{R}_u$*  **by** *blast*

**qed**

**thus** *?thesis*

**by** (*meson* *step2'* *trancl-trans*)

**qed**

**qed**

**lemma**  *$\mathcal{R}$ -typed-step*:

$\llbracket \vdash \Gamma, \mathcal{S}, P \{ c_1 \} \Gamma', \mathcal{S}', P' ;$

*tyenv-wellformed* *mds*  $\Gamma \mathcal{S} P ; mem_1 =_{\Gamma} mem_2 ; pred P mem_1 ;$

*pred* *P* *mem2* ; *tyenv-sec* *mds*  $\Gamma mem_1 ;$

$\langle c_1, mds, mem_1 \rangle \rightsquigarrow \langle c_1', mds', mem_1' \rangle \rrbracket \implies$

$(\exists c_2' mem_2'. \langle c_1, mds, mem_2 \rangle \rightsquigarrow \langle c_2', mds', mem_2' \rangle \wedge$

$\langle c_1', mds', mem_1' \rangle \mathcal{R}_{\Gamma', \mathcal{S}', P'}^u \langle c_2', mds', mem_2' \rangle)$

**proof** (*induct arbitrary: mds* *c1'* *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* *c1 = Stop*)

**assume** *c1 = Stop*

**hence** [*simp*]: *c1' = c2 mds' = mds mem1' = mem1*

**using** *seq-type*

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

**from** *seq-type* *<c1 = Stop>* **have** *context-equiv*  $\Gamma P \Gamma''$  **and**  $\mathcal{S} = \mathcal{S}''$  **and**  $P \vdash P''$  **and**

$(\forall mds. tyenv-wellformed mds \Gamma \mathcal{S} P \longrightarrow tyenv-wellformed$

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

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

**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 = c2* **in** *exI*)

```

    apply (rule-tac x = mem2 in exI)
    by (auto simp: ⟨c1 = Stop⟩ seq-stop-evalw R.intro1)
next
  assume c1 ≠ Stop
  with ⟨c1 ;; c2, mds, mem1⟩ ∼ ⟨c1', mds', mem1'⟩ obtain c1'' where
c1''-props:
  ⟨c1, mds, mem1⟩ ∼ ⟨c1'', mds', mem1'⟩ ∧ c1' = c1'' ;; c2
  by (metis seq-elim)
  with seq-type(2) obtain c2'' mem2' where c2''-props:
  ⟨c1, mds, mem2⟩ ∼ ⟨c2'', mds', mem2'⟩ ∧ ⟨c1'', mds', mem1'⟩ RuΓ'',S'',P''
  ⟨c2'', mds', mem2'⟩
  using seq-type.premis(1) seq-type.premis(2) seq-type.premis(3) seq-type.premis(4)
  seq-type.premis(5) by presburger
  hence ⟨c1'' ;; c2, mds', mem1'⟩ RuΓ',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 evalw.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-sound[OF anno-type.hyps(7)] a
    show ?thesis
    using less-eq-Sec-def by metis
  qed
  thus mem1 x = mem2 x
  using anno-type.premis(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 S P ∧ pred P mem1 ∧ tyenv-sec mds Γ mem1 →

  tyenv-sec (update-modes upd mds) Γ' mem1
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 c2' mem2' where (⟨c, update-modes upd mds, mem2⟩ ∼ ⟨c2',
mds', mem2'⟩ ∧
⟨c1', mds', mem1'⟩ RuΓ'', S'', P'' ⟨c2', mds', mem2'⟩)
  using anno-type
  apply auto
  using ⟨mem1 =Γ' mem2⟩ local.pred-def restrict-preds-to-vars-def upd-elim by
fastforce
thus ?case
  apply (rule-tac x = c2' in exI)
  apply (rule-tac x = mem2' in exI)
  apply auto
  by (metis cxt-to-stmt.simps(1) evalw.decl)
next
case stop-type
with stop-no-eval show ?case by auto
next
case (skip-type Γ S P mds)
moreover
with skip-type have [simp]: mds' = mds c1' = Stop mem1' = mem1
  using skip-elim
  by (metis, metis, metis)
with skip-type have ⟨Stop, mds, mem1'⟩ R1Γ, S, P ⟨Stop, mds, mem2'⟩
  by auto
thus ?case
  using R.intro1 and unannotated [where c = Skip and E = []]
  apply auto
  by (metis (mono-tags, lifting) R.intro1 old.prod.case skip-evalw)
next
case (assign1 x Γ e t P P' S mds)
hence upd [simp]: c1' = Stop mds' = mds mem1' = mem1 (x := evA mem1 e)

```

```

using assign-elim
by (auto, metis)
from assign1(2) upd have C-eq:  $\forall x \in \mathcal{C}. \text{mem}_1 x = \text{mem}_1' x$ 
by auto
have dma-eq [simp]:  $\text{dma mem}_1 = \text{dma mem}_1'$ 
using dma-C assign1(2) by simp
have  $\text{mem}_1(x := \text{ev}_A \text{ mem}_1 e) =_{\Gamma} \text{mem}_2(x := \text{ev}_A \text{ mem}_2 e)$ 
unfolding tyenv-eq-def
proof(clarify)
  fix v
  assume is-Low':  $\text{type-max}(\text{to-total } \Gamma v) (\text{mem}_1(x := \text{ev}_A \text{ mem}_1 e)) = \text{Low}$ 
  show  $(\text{mem}_1(x := \text{ev}_A \text{ mem}_1 e)) v = (\text{mem}_2(x := \text{ev}_A \text{ mem}_2 e)) v$ 
  proof(cases v ∈ dom Γ)
    assume [simp]:  $v \in \text{dom } \Gamma$ 
    then obtain t' where [simp]:  $\Gamma v = \text{Some } t'$  by force
    hence [simp]:  $(\text{to-total } \Gamma v) = t'$ 
    unfolding to-total-def by (auto split: if-splits)
    have  $\text{type-max } t' \text{ mem}_1 = \text{type-max } t' \text{ mem}_1'$ 
    apply(rule C-eq-type-max-eq)
    using  $\langle \Gamma v = \text{Some } t' \rangle \text{ assign}_1(6)$ 
    unfolding tyenv-wellformed-def types-wellformed-def
    apply (metis  $\langle v \in \text{dom } \Gamma \rangle \text{ option.sel}$ )

    using assign1(2) apply simp
    done
  with is-Low' have is-Low:  $\text{type-max}(\text{to-total } \Gamma v) \text{ mem}_1 = \text{Low}$ 
  by simp
  from assign1(1)  $\langle v \in \text{dom } \Gamma \rangle$  have  $x \neq v$  by auto
  thus ?thesis
  apply simp
  using is-Low assign1(7) unfolding tyenv-eq-def by auto
next
  assume  $v \notin \text{dom } \Gamma$ 
  hence [simp]:  $\bigwedge \text{mem}. \text{type-max}(\text{to-total } \Gamma v) \text{ mem} = \text{dma mem } v$ 
  unfolding to-total-def by simp
  with is-Low' have  $\text{dma mem}_1' v = \text{Low}$  by simp
  with dma-eq have dma-v-Low:  $\text{dma mem}_1 v = \text{Low}$  by simp
  hence is-Low:  $\text{type-max}(\text{to-total } \Gamma v) \text{ mem}_1 = \text{Low}$  by simp
  show ?thesis
  proof(cases x = v)
    assume  $x \neq v$ 
    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 evA-eq)

```



```

      apply(rule assign1(7))
      apply(rule assign1(8))
      apply(rule assign1(3))
      apply(rule assign1(4))
      using dma-v-Low by simp
    qed
  qed
qed

moreover have tyenv-wellformed mds  $\Gamma \mathcal{S} P \longrightarrow$  tyenv-wellformed mds'  $\Gamma \mathcal{S} P'$ 
  using upd tyenv-wellformed-preds-update assign1 by metis
moreover have pred P mem1  $\longrightarrow$  pred P' mem1'
  using pred-preds-update assign1 upd by metis

moreover have pred P mem2  $\longrightarrow$  pred P' (mem2(x := evA mem2 e))
  using pred-preds-update assign1 upd by metis

moreover have tyenv-wellformed mds  $\Gamma \mathcal{S} P \wedge$  tyenv-sec mds  $\Gamma$  mem1  $\longrightarrow$ 
  tyenv-sec mds  $\Gamma$  mem1'
  using tyenv-sec-eq[OF C-eq, where  $\Gamma = \Gamma$ ]
  unfolding tyenv-wellformed-def by blast

ultimately have  $\mathcal{R}'$ :
   $\langle \text{Stop}, \text{mds}', \text{mem}_1(x := \text{ev}_A \text{mem}_1 e) \rangle \mathcal{R}_{\Gamma, \mathcal{S}, P'}^u \langle \text{Stop}, \text{mds}', \text{mem}_2(x := \text{ev}_A$ 
  mem2 e)  $\rangle$ 
  apply -
  apply (rule  $\mathcal{R}.$ intro1, auto simp: assign1 simp del: dma-eq)
  done

have a:  $\langle x \leftarrow e, \text{mds}, \text{mem}_2 \rangle \rightsquigarrow \langle \text{Stop}, \text{mds}', \text{mem}_2(x := \text{ev}_A \text{mem}_2 e) \rangle$ 
  by (auto, metis cxt-to-stmt.simps(1) evalw.unannotated evalw-simple.assign)

from  $\mathcal{R}'$  a show ?case
  using  $\langle c_1' = \text{Stop} \rangle$  and  $\langle \text{mem}_1' = \text{mem}_1(x := \text{ev}_A \text{mem}_1 e) \rangle$ 
  by blast
next
case (assignC x  $\Gamma$  e t P P'  $\mathcal{S}$  mds)
hence upd [simp]:  $c_1' = \text{Stop}$  mds' = mds mem1' = mem1(x := evA mem1 e)
  using assign-elim
  by (auto, metis)
have mem1(x := evA mem1 e) = $\Gamma$  mem2(x := evA mem2 e)
  unfolding tyenv-eq-def
  proof(clarify)
    fix v
    assume is-Low': type-max (to-total  $\Gamma$  v) (mem1(x := evA mem1 e)) = Low
    show (mem1(x := evA mem1 e)) v = (mem2(x := evA mem2 e)) v
    proof(cases v  $\in$  dom  $\Gamma$ )
      assume in-dom [simp]: v  $\in$  dom  $\Gamma$ 
      then obtain t' where  $\Gamma v$  [simp]:  $\Gamma v = \text{Some } t'$  by force

```

```

hence [simp]: (to-total  $\Gamma$  v) = t'
  unfolding to-total-def by (auto split: if-splits)
from assignC(4) have x-nin-C: x  $\notin$  vars-of-type t'
  using in-dom  $\Gamma$  v
  by (metis option.sel snd-conv)
have  $\Gamma$ v-wf: type-wellformed t'
using in-dom  $\Gamma$  v assignC(7) unfolding tyenv-wellformed-def types-wellformed-def
  by (metis option.sel)

with x-nin-C have f-eq: type-max t' mem1 = type-max t' mem1'
  using vars-of-type-eq-type-max-eq by simp
with is-Low' have is-Low: type-max (to-total  $\Gamma$  v) mem1 = Low
  by simp
from assignC(1)  $\langle v \in \text{dom } \Gamma \rangle$  assignC(7) have x  $\neq$  v
  by (auto simp: tyenv-wellformed-def mds-consistent-def)
thus ?thesis
  apply simp
  using is-Low assignC(8) unfolding tyenv-eq-def by auto
next
assume nin-dom: v  $\notin$  dom  $\Gamma$ 
hence [simp]:  $\bigwedge$  mem. type-max (to-total  $\Gamma$  v) mem = dma mem v
  unfolding to-total-def by simp
with is-Low' have dma mem1' v = Low by simp
show ?thesis
proof(cases x = v)
  assume x = v
  thus ?thesis
  apply simp
  apply(rule evA-eq')
  apply(rule assignC(8))
  apply(rule assignC(9))
  apply(rule assignC(2))
  by(rule assignC(3))
next
assume [simp]: x  $\neq$  v
show ?thesis
proof(cases x  $\in$  C-vars v)
  assume in-C-vars: x  $\in$  C-vars v
  hence v  $\notin$  C
  using C-vars-C by auto
with nin-dom have v  $\notin$  snd  $\mathcal{S}$ 
  using assignC(7)
  by(auto simp: tyenv-wellformed-def mds-consistent-def stable-def)
with in-C-vars have P  $\vdash$  (to-total  $\Gamma$  v)
  using assignC(6) by blast
with assignC(9) have type-max (to-total  $\Gamma$  v) mem1 = Low
  by(auto simp: type-max-def pred-def pred-entailment-def)
thus ?thesis
  using not-sym[OF  $\langle x \neq v \rangle$ ]

```

**apply** *simp*  
**using** *assign<sub>C</sub>(8)*  
**unfolding** *tyenv-eq-def* **by** *auto*  
**next**  
**assume**  $x \notin \mathcal{C}\text{-vars } v$   
**with** *is-Low'* **have**  $\text{dma mem}_1 v = \text{Low}$   
**using** *dma-C-vars*  $\langle \wedge \text{mem. type-max (to-total } \Gamma v) \text{ mem} = \text{dma mem } v \rangle$   
**by** (*metis fun-upd-other*)  
**thus** *?thesis*  
**using** *not-sym[OF*  $\langle x \neq v \rangle$   
**apply** *simp*  
**using** *assign<sub>C</sub>(8)*  
**unfolding** *tyenv-eq-def* **by** *auto*  
**qed**  
**qed**  
**qed**  
**qed**

**moreover** **have**  $\text{tyenv-wellformed mds } \Gamma \mathcal{S} P \longrightarrow \text{tyenv-wellformed mds}' \Gamma \mathcal{S} P'$   
**using** *upd tyenv-wellformed-preds-update assign<sub>C</sub>* **by** *metis*  
**moreover** **have**  $\text{pred } P \text{ mem}_1 \longrightarrow \text{pred } P' \text{ mem}_1'$   
**using** *pred-preds-update assign<sub>C</sub> upd* **by** *metis*  
**moreover** **have**  $\text{pred } P \text{ mem}_2 \longrightarrow \text{pred } P' (\text{mem}_2(x := \text{ev}_A \text{ mem}_2 e))$   
**using** *pred-preds-update assign<sub>C</sub> upd* **by** *metis*  
**moreover** **have**  $\text{tyenv-wellformed mds } \Gamma \mathcal{S} P \wedge \text{pred } P \text{ mem}_1 \wedge \text{tyenv-sec mds } \Gamma \text{ mem}_1 \Longrightarrow \text{tyenv-sec mds}' \Gamma \text{ mem}_1'$   
**proof** (*clarify*)  
**fix**  $v t'$   
**assume** *wf: tyenv-wellformed mds*  $\Gamma \mathcal{S} P$   
**assume** *pred: pred*  $P \text{ mem}_1$   
**hence** *pred': pred*  $P' \text{ mem}_1'$  **using**  $\langle \text{pred } P \text{ mem}_1 \longrightarrow \text{pred } P' \text{ mem}_1' \rangle$  **by** *blast*  
**assume** *sec: tyenv-sec mds*  $\Gamma \text{ mem}_1$   
**assume**  $\Gamma v: \Gamma v = \text{Some } t'$   
**assume** *readable': v*  $\notin \text{mds}' \text{AsmNoReadOrWrite}$   
**with** *upd* **have** *readable: v*  $\notin \text{mds} \text{AsmNoReadOrWrite}$  **by** *simp*  
**with** *wf* **have**  $v \notin \text{snd } \mathcal{S}$  **by** (*auto simp: tyenv-wellformed-def mds-consistent-def*)  
**show** *type-max (the*  $(\Gamma v)) \text{ mem}_1' \leq \text{dma mem}_1' v$   
**proof** (*cases*  $x \in \mathcal{C}\text{-vars } v$ )  
**assume**  $x \in \mathcal{C}\text{-vars } v$   
**with** *assign<sub>C</sub>(6)*  $\langle v \notin \text{snd } \mathcal{S} \rangle$  **have**  $(\text{to-total } \Gamma v) \leq_{P'} (\text{dma-type } v)$  **by** *blast*  
**from** *pred'  $\Gamma v$  subtype-sound[OF this]* **show** *?thesis*  
**using** *type-max-dma-type* **by** (*auto simp: to-total-def split: if-splits*)  
**next**  
**assume**  $x \notin \mathcal{C}\text{-vars } v$   
**hence**  $\forall v' \in \mathcal{C}\text{-vars } v. \text{mem}_1 v' = \text{mem}_1' v'$   
**using** *upd* **by** *auto*  
**hence** *dma-eq: dma mem*<sub>1</sub>  $v = \text{dma mem}_1' v$   
**by** (*rule dma-C-vars*)  
**from**  $\Gamma v$  *assign<sub>C</sub>(4)* **have**  $x \notin \text{vars-of-type } t'$  **by** *force*

```

have type-wellformed  $t'$ 
  using wf  $\Gamma v$  by (force simp: tyenv-wellformed-def types-wellformed-def)
  with  $\langle x \notin \text{vars-of-type } t' \rangle$  upd have f-eq: type-max  $t'$   $\text{mem}_1 = \text{type-max } t'$ 
 $\text{mem}_1'$ 
  using vars-of-type-eq-type-max-eq by fastforce
  from sec  $\Gamma v$  readable have type-max  $t'$   $\text{mem}_1 \leq \text{dma } \text{mem}_1 v$ 
  by auto
  with f-eq dma-eq  $\Gamma v$  show ?thesis
  by simp
qed
qed

ultimately have  $\mathcal{R}'$ :
   $\langle \text{Stop}, \text{mds}', \text{mem}_1 (x := \text{ev}_A \text{mem}_1 e) \rangle \mathcal{R}_{\Gamma, \mathcal{S}, P'}^u \langle \text{Stop}, \text{mds}', \text{mem}_2 (x := \text{ev}_A$ 
 $\text{mem}_2 e) \rangle$ 
  apply –
  apply (rule  $\mathcal{R}.\text{intro}_1$ , auto simp: assign $\mathcal{C}$ )
  done

have  $a$ :  $\langle x \leftarrow e, \text{mds}, \text{mem}_2 \rangle \rightsquigarrow \langle \text{Stop}, \text{mds}', \text{mem}_2 (x := \text{ev}_A \text{mem}_2 e) \rangle$ 
by (auto,metis cxt-to-stmt.simps(1) eval $_w$ .unannotated eval $_w$ -simple.assign)

from  $\mathcal{R}'$   $a$  show ?case
  using  $\langle c_1' = \text{Stop} \rangle$  and  $\langle \text{mem}_1' = \text{mem}_1 (x := \text{ev}_A \text{mem}_1 e) \rangle$ 
  by blast
next
case (assign $_2$   $x \Gamma e t \mathcal{S} P' P \text{mds}$ )
have upd [simp]:  $c_1' = \text{Stop}$   $\text{mds}' = \text{mds}$   $\text{mem}_1' = \text{mem}_1 (x := \text{ev}_A \text{mem}_1 e)$ 
  using assign-elim[OF assign $_2(11)$ ]
  by auto
from  $\langle x \in \text{dom } \Gamma \rangle \langle \text{tyenv-wellformed } \text{mds } \Gamma \mathcal{S} P \rangle$ 
have x-nin-C:  $x \notin \mathcal{C}$ 
  by (auto simp: tyenv-wellformed-def mds-consistent-def)
hence dma-eq [simp]:  $\text{dma } \text{mem}_1' = \text{dma } \text{mem}_1$ 
  using dma-C assign $_2$ 
  by auto

let  $\mathcal{R}' = \Gamma (x \mapsto t)$ 
have  $\langle x \leftarrow e, \text{mds}, \text{mem}_2 \rangle \rightsquigarrow \langle \text{Stop}, \text{mds}, \text{mem}_2 (x := \text{ev}_A \text{mem}_2 e) \rangle$ 
  using 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':  $\text{mem}_1(x := \text{ev}_A \text{mem}_1 e) =_{\Gamma(x \mapsto t)} \text{mem}_2(x := \text{ev}_A \text{mem}_2 e)$ 
unfolding tyenv-eq-def
proof (clarify)
  fix  $v$ 
  assume is-Low': type-max (to-total ( $\Gamma(x \mapsto t)$ )  $v$ ) ( $\text{mem}_1(x := \text{ev}_A \text{mem}_1 e)$ )
  = Low

```

```

show (mem1(x := evA mem1 e)) v = (mem2(x := evA mem2 e)) v
proof(cases v = x)
  assume neq: v ≠ x
  hence type-max (to-total Γ v) mem1 = Low
  proof(cases v ∈ dom Γ)
    assume 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)
    hence [simp]: (to-total ?Γ' v) = t'
      using neq by(auto simp: to-total-def)
    have type-max t' mem1 = type-max t' mem1'
      apply(rule C-eq-type-max-eq)
      using assign2(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-C by simp
    from this is-Low' neq neq[THEN not-sym] show type-max (to-total Γ v)
mem1 = Low
  by auto
  next
    assume v ∉ dom Γ
    with is-Low' neq
    have dma mem1' 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 assign2(7) show (mem1(x := evA mem1 e)) v = (mem2(x := evA
mem2 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 mem1' = Low
      by(auto simp: to-total-def split: if-splits)
    have wf-t: type-wellformed t
      using type-aexpr-type-wellformed assign2(2) assign2(6)
      by(fastforce simp: tyenv-wellformed-def)
    with t-Low' ⟨x ∉ C⟩ have t-Low: type-max t mem1 = Low
      using C-eq-type-max-eq
      by (metis (no-types, lifting) fun-upd-other upd(3))
    show ?thesis
  proof(simp, rule eval-vars-detA, clarify)
    fix y
    assume in-vars: y ∈ aexp-vars e
    have type-max (to-total Γ y) mem1 = 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 := evA 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 (( $\Gamma(x \mapsto t)$ ) v)) = t and t-def: t = t'
    using  $\Gamma v$  by auto
  from  $\langle v \notin \text{mds}' \text{ AsmNoReadOrWrite} \rangle \text{ upd wf}$  have readable: v  $\notin$  snd  $\mathcal{S}$ 
    by (auto simp: tyenv-wellformed-def mds-consistent-def)
  with assign2(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]: (( $\Gamma(x \mapsto t)$ ) v) =  $\Gamma v$ 
    by simp
  with  $\Gamma v$  have  $\Gamma v$ :  $\Gamma v = \text{Some } t'$  by simp
  with sec upd  $\langle v \notin \text{mds}' \text{ AsmNoReadOrWrite} \rangle$  have f-leq: type-max t' mem1
 $\leq$  dma mem1 v
    by auto
  have C-eq:  $\forall x \in \mathcal{C}. \text{mem}_1 x = \text{mem}_1' x$ 
    using wf assign2(1) upd by (auto simp: tyenv-wellformed-def mds-consistent-def)
  hence dma-eq: dma mem1 = dma mem1'
    by (rule dma-C)
  have f-eq: type-max t' mem1 = type-max t' mem1'
    apply (rule C-eq-type-max-eq)
    using  $\Gamma v$  wf apply (force simp: tyenv-wellformed-def types-wellformed-def)
    by (rule C-eq)
  from neq  $\Gamma v$  f-leq dma-eq f-eq show ?thesis
    by simp
qed
qed

  have  $\langle \text{Stop}, \text{mds}, \text{mem}_1 (x := \text{ev}_A \text{mem}_1 e) \rangle \mathcal{R}^1_{\mathcal{I}\Gamma', \mathcal{S}, P'} \langle \text{Stop}, \text{mds}, \text{mem}_2 (x$ 
:= evA mem2 e)  $\rangle$ 
    apply (rule  $\mathcal{R}_1$ .intro)
      apply blast
      using wf assign2 apply fastforce
      apply (rule tyenv-eq')
      using p assign2 apply fastforce
      using p2 assign2 apply fastforce
      using sec assign2
      using upd(2) upd(3) by blast

  ultimately have  $\langle x \leftarrow e, \text{mds}, \text{mem}_2 \rangle \rightsquigarrow \langle \text{Stop}, \text{mds}', \text{mem}_2 (x := \text{ev}_A \text{mem}_2$ 
e)  $\rangle$ 
     $\langle \text{Stop}, \text{mds}', \text{mem}_1 (x := \text{ev}_A \text{mem}_1 e) \rangle \mathcal{R}^u_{\Gamma(x \mapsto t), \mathcal{S}, P'} \langle \text{Stop}, \text{mds}', \text{mem}_2 (x$ 
:= evA mem2 e)  $\rangle$ 
      using  $\mathcal{R}$ .intro1
      by auto
  thus ?case

```

```

using  $\langle mds' = mds \rangle \langle c_1' = Stop \rangle \langle mem_1' = mem_1(x := ev_A mem_1 e) \rangle$ 
by blast
next
case (if-type  $\Gamma e t P S th \Gamma' S' P' el \Gamma'' P'' \Gamma''' P'''$ )
let  $?P = if (ev_B mem_1 e) then P +_S e else P +_S (bexp-neg e)$ 
from  $\langle \langle Stmt.If e th el, mds, mem_1 \rangle \rightsquigarrow \langle c_1', mds', mem_1 \rangle \rangle$  have  $ty: \vdash \Gamma, S, ?P$ 
 $\{c_1\} \Gamma''', S', P'''$ 
proof (rule if-elim)
  assume  $c_1' = th mem_1' = mem_1 mds' = mds ev_B mem_1 e$ 
  with if-type(3)
  show ?thesis
  apply simp
  apply(erule sub)
  using if-type apply simp+
  done
next
assume  $c_1' = el mem_1' = mem_1 mds' = mds \neg ev_B mem_1 e$ 
with if-type(5)
show ?thesis
apply simp
apply(erule sub)
using if-type apply simp+
done
qed
have evB-eq [simp]:  $ev_B mem_1 e = ev_B mem_2 e$ 
apply(rule evB-eq')
  apply(rule  $\langle mem_1 =_{\Gamma} mem_2 \rangle$ )
  apply(rule  $\langle pred P mem_1 \rangle$ )
  apply(rule  $\langle \Gamma \vdash_b e \in t \rangle$ )
by(rule  $\langle P \vdash t \rangle$ )
have  $(\langle c_1', mds, mem_1 \rangle, \langle c_1', mds, mem_2 \rangle) \in \mathcal{R} \Gamma''' S' P'''$ 
apply (rule intro1)
apply clarify
apply (rule  $\mathcal{R}_1.intro$  [where  $\Gamma = \Gamma$  and  $\Gamma' = \Gamma'''$  and  $S = S$  and  $P = ?P$ ])
  apply(rule ty)
  using  $\langle tyenv-wellformed mds \Gamma S P \rangle$ 
  apply(auto simp: tyenv-wellformed-def mds-consistent-def add-pred-def)[1]
  apply(rule  $\langle mem_1 =_{\Gamma} mem_2 \rangle$ )
using  $\langle pred P mem_1 \rangle$  apply(fastforce simp: pred-def add-pred-def bexp-neg-negates)
using  $\langle pred P mem_2 \rangle$  apply(fastforce simp: pred-def add-pred-def bexp-neg-negates)
by(rule  $\langle tyenv-sec mds \Gamma mem_1 \rangle$ )

show ?case
proof –
from evB-eq if-type(13) have  $(\langle If e th el, mds, mem_2 \rangle \rightsquigarrow \langle c_1', mds, mem_2 \rangle)$ 
apply (cases  $ev_B mem_1 e$ )
apply (subgoal-tac  $c_1' = th$ )
apply clarify
apply (metis cxt-to-stmt.simps(1) evalw-simplep.if-true evalw.p.unannotated)

```



```

evalwp-evalw-eq if-type(8))
  using if-type.premis(6) apply blast
  apply (subgoal-tac c1' = el)
  apply (metis (opaque-lifting, mono-tags) cxt-to-stmt.simps(1) evalw.unannotated
evalw-simple.if-false if-type(8))
  using if-type.premis(6) by blast
  with ⟨c1', mds, mem1⟩  $\mathcal{R}^u_{\Gamma, \mathcal{S}, P}$  ⟨c1', mds, mem2⟩ show ?thesis
  by (metis if-elim if-type.premis(6))
qed
next
case (while-type  $\Gamma$  e t P  $\mathcal{S}$  c)
hence [simp]: c1' = (If e (c ;; While e c) Stop) and
[simp]: mds' = mds and
[simp]: mem1' = mem1
by (auto simp: while-elim)

with while-type have ⟨While e c, mds, mem2⟩  $\rightsquigarrow$  ⟨c1', mds, mem2⟩
by (metis cxt-to-stmt.simps(1) evalw-simplep.while evalwp.unannotated evalwp-evalw-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 ⟨c1', mds, mem1⟩  $\mathcal{R}^1_{\Gamma, \mathcal{S}, P}$  ⟨c1', mds, mem2⟩
  apply (rule  $\mathcal{R}_1$ .intro [where  $\Gamma = \Gamma$ ])
  apply (rule ty)
  using while-type apply simp+
done
hence ⟨c1', mds, mem1⟩  $\mathcal{R}^u_{\Gamma, \mathcal{S}, P}$  ⟨c1', mds, mem2⟩
  using  $\mathcal{R}$ .intro1 by auto
ultimately show ?case
  by fastforce
next
case (sub  $\Gamma_1$   $\mathcal{S}$  P1 c  $\Gamma_1'$   $\mathcal{S}'$  P1'  $\Gamma_2$  P2  $\Gamma_2'$  P2' mds c1')
have imp: tyenv-wellformed mds  $\Gamma_2$   $\mathcal{S}$  P2  $\wedge$  pred P2 mem1  $\wedge$  pred P2 mem2  $\wedge$ 

```

$tyenv\text{-}sec\ mds\ \Gamma_2\ mem_1 \implies$   
 $tyenv\text{-}wellformed\ mds\ \Gamma_1\ \mathcal{S}\ P_1 \wedge pred\ P_1\ mem_1 \wedge pred\ P_1\ mem_2 \wedge$   
 $tyenv\text{-}sec\ mds\ \Gamma_1\ mem_1$   
**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**  $c_2'\text{-props}: \langle c, mds, mem_2 \rangle \rightsquigarrow \langle c_2',$   
 $mds', mem_2' \rangle$   
 $\langle c_1', mds', mem_1 \rangle \mathcal{R}^u_{\Gamma_1', \mathcal{S}', P_1'} \langle c_2', mds', mem_2' \rangle$   
**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  $\Gamma\ e\ t\ P\ \mathcal{S}\ c\ \Gamma'\ \mathcal{S}'\ P'$ )  
**from** await-type.premis **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.premis  $\langle \Gamma \vdash_b e \in t \rangle \langle P \vdash t \rangle$  **have**  $pred\ P +_{\mathcal{S}} e\ mem_1\ pred\ P +_{\mathcal{S}}$   
 $e\ mem_2\ ev_B\ mem_2\ e$   
**using** pred-plus-impl  $\langle ev_B\ mem_1\ e \rangle \langle pred\ P\ mem_1 \rangle$   $ev_B\text{-}eq'$   
**by** blast+  
**from** await-type.premis  $\langle \Gamma \vdash_b e \in t \rangle \langle P \vdash t \rangle$  **have** wellformed: tyenv-wellformed  
 $mds\ \Gamma\ \mathcal{S}\ P +_{\mathcal{S}}\ e$   
**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  $\langle is\text{-}final\ c_1' \rangle \langle no\text{-}await\ c \rangle \langle tyenv\text{-}wellformed\ mds\ \Gamma\ \mathcal{S}\ P +_{\mathcal{S}}\ e \rangle$  await-type.premis  
 $\langle pred\ P +_{\mathcal{S}}\ e\ mem_1 \rangle \langle pred\ P +_{\mathcal{S}}\ e\ mem_2 \rangle$   
**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  $\langle is\text{-}final\ c_2' \rangle \langle ev_B\ mem_2\ e \rangle \langle no\text{-}await\ c \rangle \langle \Gamma \vdash_b e \in t \rangle \langle P \vdash t \rangle$   
 $\langle pred\ P +_{\mathcal{S}}\ e\ mem_2 \rangle$  step rel **show** ?case  
**using** eval\_w.intros(4) **by** blast  
**qed**

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

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

**lemma**  $\mathcal{R}$ -*to- $\mathcal{R}_3$* :  $\llbracket \langle c_1, mds, mem_1 \rangle \mathcal{R}^u_{\Gamma, \mathcal{S}, P} \langle c_2, mds, mem_2 \rangle ; \vdash \Gamma, \mathcal{S}, P \{ c \} \Gamma', \mathcal{S}', P' \rrbracket \implies$   
 $\langle c_1 ;; c, mds, mem_1 \rangle \mathcal{R}^3_{\Gamma', \mathcal{S}', P'} \langle c_2 ;; c, mds, mem_2 \rangle$   
**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  $\Gamma \mathcal{S} P c_2 mem_2 c \Gamma' \mathcal{S}' P'$* )  
**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*  $\langle c_1 = Stop \rangle$  *cxt-to-stmt.simps(1)* *eval\_w-simplep.seq-stop*  
*eval\_w.p.unannotated eval\_w.p-eval\_w-eq intro\_1.premseq-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)  $\langle c_1 = Stop \rangle$  intro\_1.premseq-stop-elim*  
*stop-cxt tyenv-wellformed-sub*)  
  **using**  $\langle c_1 = Stop \rangle$  *intro\_1.premseq-stop-elim stop-cxt context-equiv-tyenv-eq*

```

    apply metis

    using ⟨c1 = Stop⟩ intro1.prems pred-entailment-def seq-stop-elim stop-ctx
  apply blast
    using pred-entailment-def stop-ctx apply blast

    apply (metis (no-types, lifting) ⟨c1 = Stop⟩ context-equiv-def intro1.prems
less-eq-Sec-def seq-stop-elim stop-ctx subtype-sound type-equiv-def)
    using intro1.prems seq-stop-elim by auto
  next
    assume c1 ≠ Stop
    from intro1
    obtain c1'' where ⟨c1, mds, mem1⟩ ∼ ⟨c1'', mds', mem1'⟩ ∧ c1' = (c1'' ;;
c)
      by (metis ⟨c1 ≠ Stop⟩ intro1.prems seq-elim)
    with intro1
    obtain c2'' mem2' where ⟨c2, mds, mem2⟩ ∼ ⟨c2'', mds', mem2'⟩ ⟨c1'',
mds', mem1'⟩  $\mathcal{R}_{\Gamma, \mathcal{S}, P}^u$  ⟨c2'', mds', mem2'⟩
      using  $\mathcal{R}_1$ -weak-bisim and weak-bisim-def
      by blast
    thus ?thesis
      using intro1(2)  $\mathcal{R}$ -to- $\mathcal{R}_3$ 
      apply (rule-tac x = c2'' ;; c in exI)
      apply (rule-tac x = mem2' in exI)
      apply (rule conjI)
      apply (metis evalw.seq)
      apply auto
      apply (rule  $\mathcal{R}$ .intro3)

      by (simp add:  $\mathcal{R}$ -to- $\mathcal{R}_3$  ⟨c1, mds, mem1⟩ ∼ ⟨c1'', mds', mem1'⟩ ∧ c1' =
c1'' ;; c)
    qed
  next
    case (intro3 c1 mds mem1  $\Gamma$   $\mathcal{S}$  P c2 mem2 c  $\Gamma'$   $\mathcal{S}'$  P')
    thus ?case
      apply (cases c1 = Stop)
      apply blast
    proof –
      assume c1 ≠ Stop
      then obtain c1'' where ⟨c1, mds, mem1⟩ ∼ ⟨c1'', mds', mem1'⟩ c1' = (c1''
;; c)
        by (metis intro3.prems seq-elim)
      then obtain c2'' mem2' where ⟨c2, mds, mem2⟩ ∼ ⟨c2'', mds', mem2'⟩
⟨c1'', mds', mem1'⟩  $\mathcal{R}_{\Gamma, \mathcal{S}, P}^u$  ⟨c2'', mds', mem2'⟩
        using intro3(2) by metis
      thus ?thesis
        apply (rule-tac x = c2'' ;; c in exI)
        apply (rule-tac x = mem2' in exI)
        apply (rule conjI)

```

```

    apply (metis eval_w.seq)
  apply (erule R-elim)
  apply simp-all
  apply (metis R.intro3 R-to-R3 <c1'', mds', mem1^> R^u_{\Gamma, S, P} <c2'', mds',
mem2^> <c1' = c1'' ;; c> intro3(3))
  apply (metis (lifting) R.intro3 R-to-R3 <c1'', mds', mem1^> R^u_{\Gamma, S, P}
<c2'', mds', mem2^> <c1' = c1'' ;; c> intro3(3))
  done
qed
qed
}
thus ?thesis
  unfolding weak-bisim-def
  by auto
qed

```

```

lemma R-bisim: strong-low-bisim-mm (R \Gamma' S' P')
  unfolding strong-low-bisim-mm-def
proof (auto)
  from R-sym show sym (R \Gamma' S' P') .
next
  from R-closed-glob-consistent show closed-glob-consistent (R \Gamma' S' P') .
next
  fix c1 mds mem1 c2 mem2
  assume <c1, mds, mem1> R^u_{\Gamma', S', P'} <c2, mds, mem2>
  thus mem1 =_{mds} mem2
    apply (rule R-elim)
    by (auto simp: R1-mem-eq R3-mem-eq)
next
  fix c1 mds mem1 c2 mem2 c1' mds' mem1'
  assume eval: <c1, mds, mem1> \rightsquigarrow <c1', mds', mem1^>
  assume R: <c1, mds, mem1> R^u_{\Gamma', S', P'} <c2, mds, mem2>
  from R show \exists c2' mem2'. <c2, mds, mem2> \rightsquigarrow <c2', mds', mem2^> \wedge
    <c1', mds', mem1^> R^u_{\Gamma', S', P'} <c2', mds', mem2^>
    apply (rule R-elim)
    apply (insert R1-weak-bisim R3-weak-bisim eval weak-bisim-def)
    apply auto
  done
qed

```

```

lemma Typed-in-R:
  assumes typeable: \vdash \Gamma, S, P \{ c \} \Gamma', S', P'
  assumes wf: tyenv-wellformed mds \Gamma S P
  assumes mem-eq: \forall x. type-max (to-total \Gamma x) mem1 = Low \longrightarrow mem1 x =
mem2 x
  assumes pred1: pred P mem1
  assumes pred2: pred P mem2

```

**assumes** *tyenv-sec*: *tyenv-sec mds*  $\Gamma$  *mem*<sub>1</sub>  
**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*  $\Gamma'$ ])  
**apply** *clarify*  
**apply** (*rule*  $\mathcal{R}_1.intro$  [*of*  $\Gamma$ ])  
    **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*  $\Gamma \mathcal{S} P$   
**assumes** *mem-eq*:  $\forall x. type-max (to-total \Gamma x) mem_1 = Low \longrightarrow mem_1 x = mem_2 x$   
**assumes** *pred*<sub>1</sub>: *pred*  $P mem_1$   
**assumes** *pred*<sub>2</sub>: *pred*  $P mem_2$   
**assumes** *tyenv-sec*: *tyenv-sec mds*  $\Gamma mem_1$   
**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**

$\Gamma$ -*of-mds* ::  $'Var Mds \Rightarrow ('Var, 'BExp) TyEnv$

**where**

$\Gamma$ -*of-mds* *mds*  $\equiv (\lambda x. if x \notin \mathcal{C} \wedge x \in mds \text{ AsmNoWrite} \cup mds \text{ AsmNoReadOrWrite}$   
*then*

*if*  $x \in mds \text{ AsmNoReadOrWrite}$  *then*  
    *Some* ( $\{pred\text{-False}\}$ )  
    *else*  
    *Some* (*dma-type*  $x$ )  
    *else None*)

**definition**

$\mathcal{S}$ -*of-mds* ::  $'Var Mds \Rightarrow 'Var Stable$

**where**

$\mathcal{S}$ -*of-mds* *mds*  $\equiv (mds \text{ AsmNoWrite}, mds \text{ AsmNoReadOrWrite})$

**definition**

*mds-yields-stable-types* ::  $'Var Mds \Rightarrow bool$

**where**

*mds-yields-stable-types* *mds*  $\equiv \forall x. x \in mds \text{ AsmNoWrite} \cup mds \text{ AsmNoReadOrWrite} \longrightarrow$

$(\forall v \in \mathcal{C}\text{-vars } x. v \in mds \text{ AsmNoWrite} \cup mds \text{ AsmNoReadOrWrite})$

**inductive**

$type\text{-}global :: (('Var, 'AExp, 'BExp) Stmt \times 'Var Mds) list \Rightarrow bool$   
 $(\vdash - [120] 1000)$   
**where**  
 $\llbracket list\text{-}all (\lambda (c,m). (\exists \Gamma' \mathcal{S}' P'. \vdash (\Gamma\text{-of}\text{-}mds\ m), (\mathcal{S}\text{-of}\text{-}mds\ m), \{\} \{ c \} \Gamma', \mathcal{S}', P'))$   
 $\wedge mds\text{-}yields\text{-}stable\text{-}types\ m) cs ;$   
 $\quad \forall mem. sound\text{-}mode\text{-}use\ (cs, mem)$   
 $\rrbracket \Longrightarrow$   
 $type\text{-}global\ cs$

**inductive-cases**  $type\text{-}global\text{-}elim: \vdash cs$

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

**lemma**  $\Gamma\text{-of}\text{-}mds\text{-}tyenv\text{-}sec:$   
 $tyenv\text{-}sec\ m (\Gamma\text{-of}\text{-}mds\ m) mem_1$   
**apply**( $auto\ simp: \Gamma\text{-of}\text{-}mds\text{-}def$ )  
**done**

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

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

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

**theorem**  $type\text{-}soundness\text{-}global:$   
**assumes**  $typeable: \vdash cs$   
**shows**  $prog\text{-}sifum\text{-}secure\text{-}cont\ cs$

```

using typeable
apply (rule type-global-elim)
apply (subgoal-tac  $\forall c \in \text{set } cs. \text{com-sifum-secure } c$ )
  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 evA evB aexp-vars bexp-vars + sifum-security dma C-vars C
  evalw undefined
  for evA :: ('Var, 'Val) Mem  $\Rightarrow$  'AExp  $\Rightarrow$  'Val
  and evB :: ('Var, 'Val) Mem  $\Rightarrow$  'BExp  $\Rightarrow$  bool
  and aexp-vars :: 'AExp  $\Rightarrow$  'Var set
  and bexp-vars :: 'BExp  $\Rightarrow$  'Var set
  and dma :: ('Var, 'Val) Mem  $\Rightarrow$  'Var  $\Rightarrow$  Sec
  and C-vars :: 'Var  $\Rightarrow$  'Var set
  and C :: 'Var set

context sifum-modes
begin

abbreviation
  eval-abv-modes :: (-, 'Var, 'Val) LocalConf  $\Rightarrow$  (-, -, -) LocalConf  $\Rightarrow$  bool
  (infixl  $\rightsquigarrow$  70)
where
   $x \rightsquigarrow y \equiv (x, y) \in \text{eval}_w$ 

fun
  update-annos :: 'Var Mds  $\Rightarrow$  'Var ModeUpd list  $\Rightarrow$  'Var Mds

```



(**infix**  $\oplus$  140)

**where**

$$\text{update-annos } mds \ [] = mds \mid$$

$$\text{update-annos } mds (a \# as) = \text{update-annos } (\text{update-modes } a \ mds) \ as$$

**fun**

$$\text{annotate} :: ('Var, 'AExp, 'BExp) \text{ Stmt} \Rightarrow 'Var \ \text{ModeUpd} \ \text{list} \Rightarrow ('Var, 'AExp, 'BExp) \text{ Stmt}$$

(**infix**  $\otimes$  140)

**where**

$$\text{annotate } c \ [] = c \mid$$

$$\text{annotate } c (a \# as) = (\text{annotate } c \ as)@[a]$$

**inductive**

$$\text{mode-type} :: 'Var \ Mds \Rightarrow ('Var, 'AExp, 'BExp) \text{ Stmt} \Rightarrow 'Var \ Mds \Rightarrow \text{bool} \ (\vdash \ - \ \{ \ - \} \ -)$$

**where**

$$\text{skip}: \vdash \ mds \ \{ \ \text{Skip} \ \otimes \ \text{annos} \} \ (m \oplus \ \text{annos}) \ \mid$$

$$\text{assign}: \llbracket x \notin mds \ \text{GuarNoWrite} \cup mds \ \text{GuarNoReadOrWrite} ; \ \text{aexp-vars } e \cap mds \ \text{GuarNoReadOrWrite} = \{\} ;$$

$$\forall v. (x \in \mathcal{C}\text{-vars } v \longrightarrow v \notin mds \ \text{GuarNoWrite} \cup mds \ \text{GuarNoReadOrWrite})$$

$$\wedge$$

$$(\mathcal{C}\text{-vars } v \cap \text{aexp-vars } e \neq \{\} \longrightarrow v \notin mds \ \text{GuarNoReadOrWrite}) \rrbracket$$

$$\implies$$

$$\vdash \ mds \ \{ (x \leftarrow e) \ \otimes \ \text{annos} \} \ (m \oplus \ \text{annos}) \ \mid$$

$$\text{if-}: \llbracket \vdash \ (m \oplus \ \text{annos}) \ \{ \ c_1 \} \ mds'' ;$$

$$\vdash \ (m \oplus \ \text{annos}) \ \{ \ c_2 \} \ mds'' ;$$

$$\text{bexp-vars } e \cap mds \ \text{GuarNoReadOrWrite} = \{\} ;$$

$$\forall v. \mathcal{C}\text{-vars } v \cap \text{bexp-vars } e \neq \{\} \longrightarrow v \notin mds \ \text{GuarNoReadOrWrite} \rrbracket \implies$$

$$\vdash \ mds \ \{ \ \text{If } e \ c_1 \ c_2 \ \otimes \ \text{annos} \} \ mds'' \ \mid$$

$$\text{while}: \llbracket mds' = m \oplus \ \text{annos} ; \vdash \ mds' \ \{ \ c \} \ mds' ; \text{bexp-vars } e \cap mds' \ \text{GuarNoReadOrWrite} = \{\} ;$$

$$\forall v. \mathcal{C}\text{-vars } v \cap \text{bexp-vars } e \neq \{\} \longrightarrow v \notin mds' \ \text{GuarNoReadOrWrite} \rrbracket$$

$$\implies$$

$$\vdash \ mds \ \{ \ \text{While } e \ c \ \otimes \ \text{annos} \} \ mds' \ \mid$$

$$\text{seq}: \llbracket \vdash \ mds \ \{ \ c_1 \} \ mds' ; \vdash \ mds' \ \{ \ c_2 \} \ mds'' \rrbracket \implies \vdash \ mds \ \{ \ c_1 \ ; \ ; \ c_2 \} \ mds'' \ \mid$$

$$\text{sub}: \llbracket \vdash \ mds_2 \ \{ \ c \} \ mds_2' ; mds_1 \leq mds_2 ; mds_2' \leq mds_1' \rrbracket \implies$$

$$\vdash \ mds_1 \ \{ \ c \} \ mds_1'$$

## 6.2 Soundness of the Type System

**lemma** *cxt-eval*:

$$\llbracket \langle \text{cxt-to-stmt} \ [] \ c, \ mds, \ \text{mem} \rangle \rightsquigarrow \langle \text{cxt-to-stmt} \ [] \ c', \ mds', \ \text{mem}' \rangle \rrbracket \implies$$

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

**by** *auto*

**lemma** *update-preserves-le*:

$$mds_1 \leq mds_2 \implies (mds_1 \oplus \ \text{annos}) \leq (mds_2 \oplus \ \text{annos})$$

**proof** (*induct annos arbitrary: mds<sub>1</sub> mds<sub>2</sub>*)

```

case Nil
thus ?case by simp
next
case (Cons a annos mds1 mds2)
hence update-modes a mds1 ≤ update-modes a mds2
  by (case-tac a, auto simp: le-fun-def)
with Cons show ?case
  by auto
qed

```

**lemma** *doesn't-read-annos*:

```

doesn't-read-or-modify c x  $\implies$  doesn't-read-or-modify (c ⊗ annos) x
unfolding doesn't-read-or-modify-def doesn't-read-or-modify-vars-def
apply clarify
apply (induct annos)
apply simp
apply (metis (lifting))
apply clarsimp
apply (rule cxt-eval)
apply (rule evalw.decl)
apply (metis cxt-eval evalw.decl upd-elim)+
done

```

**lemma** *doesn't-modify-annos*:

```

doesn't-modify c x  $\implies$  doesn't-modify (c ⊗ annos) x
unfolding doesn't-modify-def
apply clarsimp
apply (induct annos)
apply simp
by (metis annotate.simps(2) upd-elim)

```

**lemma** *stop-loc-reach*:

```

[[ ⟨c', mds', mem⟩ ∈ loc-reach ⟨Stop, mds, mem⟩ ]]  $\implies$ 
c' = Stop ∧ mds' = mds
apply (induct rule: loc-reach.induct)
by (auto simp: stop-no-eval)

```

**lemma** *stop-doesn't-access*:

```

doesn't-modify Stop x ∧ doesn't-read-or-modify Stop x
unfolding doesn't-modify-def and doesn't-read-or-modify-def doesn't-read-or-modify-vars-def
using stop-no-eval
by auto

```

**lemma** *skip-eval-step*:

```

⟨Skip ⊗ annos, mds, mem⟩  $\rightsquigarrow$  ⟨Stop, mds ⊕ annos, mem⟩
apply (induct annos arbitrary: mds)

```

**apply** *simp*  
**apply** (*metis cxt-to-stmt.simps*(1) *eval<sub>w</sub>.unannotated eval<sub>w</sub>-simple.skip*)  
**apply** *simp*  
**apply** (*insert eval<sub>w</sub>.decl*)  
**apply** (*rule cxt-eval*)  
**apply** (*rule eval<sub>w</sub>.decl*)  
**by** *auto*

**lemma** *skip-eval-elim*:

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

**lemma** *skip-doesnt-read*:

*doesnt-read-or-modify* (*Skip*  $\otimes$  *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*  $\otimes$  *annos*) *x*  
**apply** (*rule doesnt-modify-annos*)  
**apply** (*clarsimp simp: doesnt-modify-def*)  
**by** (*metis skip-elim*)+

**lemma** *skip-loc-reach*:

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

**lemma** *skip-doesnt-access*:

$\llbracket lc \in \text{loc-reach} \langle \text{Skip} \otimes \text{annos}, \text{mds}, \text{mem} \rangle ; lc = \langle c', \text{mds}', \text{mem}' \rangle \rrbracket \implies$   
*doesnt-read-or-modify* *c' x*  $\wedge$  *doesnt-modify* *c' x*  
**apply** (*subgoal-tac* ( $c' = \text{Stop} \wedge \text{mds}' = (\text{mds} \oplus \text{annos})$ )  $\vee$  ( $c' = \text{Skip} \otimes \text{annos} \wedge \text{mds}' = \text{mds}$ ))  
**apply** (*rule conjI, erule disjE*)  
**apply** (*simp add: doesnt-read-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*:

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

**lemma** *assign-annos-eval*:

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

**lemma** *assign-annos-eval-elim*:

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

**lemma** *mem-upd-commute*:

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

**lemma** *assign-doesnt-read*:

$\llbracket y \neq x; y \notin \text{aexp-vars } e; x \notin \mathcal{C}\text{-vars } y; \mathcal{C}\text{-vars } y \cap \text{aexp-vars } e = \{\} \rrbracket \implies$   
*doesnt-read-or-modify*  $((x \leftarrow e) \otimes \text{annos}) y$

**apply** (*rule doesnt-read-annos*)

**proof** –

**assume**  $y \neq x$   
 $y \notin \text{aexp-vars } e$   
 $x \notin \mathcal{C}\text{-vars } y$   
 $\mathcal{C}\text{-vars } y \cap \text{aexp-vars } e = \{\}$

**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*)<sup>+</sup>

**apply** (*rename-tac mds mem c' mds' mem'*)

**apply** (*rule impI*)

**apply** (*subgoal-tac c' = Stop  $\wedge$  mds' = mds  $\wedge$  mem' = mem (x := ev<sub>A</sub> 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 evalw.unannotated)
apply (metis evalw-simple.assign eval-vars-detA 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 evalw.unannotated)
apply (subgoal-tac va  $\notin$  aexp-vars e)
apply (metis evalw-simple.assign eval-vars-detA fun-upd-apply)
apply blast
apply (metis mem-upd-commute)
apply (metis assign-elim)
done

```

qed

**lemma** *assign-loc-reach*:

```

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

```

**lemma** *if-doesnt-modify*:

```

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

```

**lemma** *vars-eval<sub>B</sub>*:

```

 $x \notin bexp\text{-}vars \ e \implies ev_B mem e = ev_B (mem (x := v)) e$ 
by (metis (lifting) eval-vars-detB fun-upd-other)

```

**lemma** *if-doesnt-read*:

```

 $x \notin bexp\text{-}vars \ e \implies \mathcal{C}\text{-}vars \ x \cap bexp\text{-}vars \ e = \{\}$   $\implies$  doesnt-read-or-modify (If  $e \ c_1 \ c_2 \otimes \ annos$ )  $x$ 
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 evB mem e)
apply (subgoal-tac c' = c1 ∧ mds' = mds ∧ mem' = mem)
  prefer 2
  apply auto[1]
apply clarsimp
apply (rule cxt-eval)
apply (rule evalw.unannotated)
apply (rule evalw-simple.if-true)
apply (metis (lifting) vars-evalB)
apply (subgoal-tac c' = c2 ∧ mds' = mds ∧ mem' = mem)
  prefer 2
  apply auto[1]
apply clarsimp
apply (rule cxt-eval)
apply (rule evalw.unannotated)
apply (rule evalw-simple.if-false)
apply (metis (lifting) vars-evalB)
apply (rename-tac mds mem c' mds' mem' va v)
apply (case-tac evB mem e)
apply (subgoal-tac c' = c1 ∧ mds' = mds ∧ mem' = mem)
  prefer 2
  apply auto[1]
apply clarsimp
apply (rule cxt-eval)
apply (rule evalw.unannotated)
apply (rule evalw-simple.if-true)
apply (subgoal-tac va ∉ bexp-vars e)
  apply (metis (lifting) vars-evalB)
apply blast
apply (subgoal-tac c' = c2 ∧ mds' = mds ∧ mem' = mem)
  prefer 2
  apply auto[1]
apply clarsimp
apply (rule cxt-eval)
apply (rule evalw.unannotated)
apply (rule evalw-simple.if-false)
apply (subgoal-tac va ∉ bexp-vars e)
  apply (metis (lifting) vars-evalB)
apply blast
done

```

**lemma** *if-eval-true*:

```

[[ evB mem e ] ⇒⇒
⟨If e c1 c2 ⊗ annos, mds, mem⟩ ∼∼ ⟨c1, mds ⊕ annos, mem⟩
apply (induct annos arbitrary: mds)
apply simp
apply (metis cxt-eval evalw.unannotated evalw-simple.if-true)
by (metis annotate.simps(2) cxt-eval evalw.decl update-annos.simps(2))

```

**lemma** *if-eval-false*:

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

**lemma** *if-eval-elim*:

$\llbracket \langle \text{If } e \ c_1 \ c_2 \otimes \text{ annos}, \text{ mds}, \text{ mem} \rangle \rightsquigarrow \langle c', \text{ mds}', \text{ mem}' \rangle \rrbracket \implies$   
 $((c' = c_1 \wedge \text{ev}_B \text{ mem } e) \vee (c' = c_2 \wedge \neg \text{ev}_B \text{ mem } e)) \wedge \text{ mds}' = \text{ mds} \oplus \text{ annos} \wedge$   
 $\text{ mem}' = \text{ mem}$   
**apply** (*rule ccontr*)  
**apply** (*cases ev<sub>B</sub> mem e*)  
**apply** (*insert if-eval-true deterministic*)  
**apply** *blast*  
**using** *if-eval-false deterministic*  
**by** *blast*

**lemma** *if-eval-elim'*:

$\llbracket \langle \text{If } e \ c_1 \ c_2, \text{ mds}, \text{ mem} \rangle \rightsquigarrow \langle c', \text{ mds}', \text{ mem}' \rangle \rrbracket \implies$   
 $((c' = c_1 \wedge \text{ev}_B \text{ mem } e) \vee (c' = c_2 \wedge \neg \text{ev}_B \text{ mem } e)) \wedge \text{ mds}' = \text{ mds} \wedge \text{ mem}' =$   
 $\text{ mem}$   
**using** *if-eval-elim [where annos = []]*  
**by** *auto*

**lemma** *loc-reach-refl'*:

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

**lemma** *if-loc-reach*:

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

**apply** (*metis loc-reach.step*)  
**by** (*metis loc-reach.mem-diff*)

**lemma** *if-loc-reach'*:

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

**lemma** *seq-loc-reach*:

$\llbracket \langle c', mds', mem \rangle \in \text{loc-reach} \langle c_1 ;; c_2, mds, mem \rangle \rrbracket \implies$   
 $(\exists c''. c' = c'' ;; c_2 \wedge \langle c'', mds', mem \rangle \in \text{loc-reach} \langle c_1, mds, mem \rangle) \vee$   
 $(\exists c'' mds'' mem''. \langle \text{Stop}, mds'', mem'' \rangle \in \text{loc-reach} \langle c_1, mds, mem \rangle \wedge$   
 $\langle c', mds', mem \rangle \in \text{loc-reach} \langle c_2, mds'', mem'' \rangle)$

**apply** (*induct rule: loc-reach.induct*)  
**apply** *simp*  
**apply** (*metis loc-reach-refl'*)  
**apply** *simp*  
**apply** (*metis (no-types) loc-reach.step loc-reach-refl' seq-elim seq-stop-elim*)  
**by** (*metis (lifting) loc-reach.mem-diff*)

**lemma** *seq-doesnt-read*:

$\llbracket \text{doesnt-read-or-modify } c \ x \rrbracket \implies \text{doesnt-read-or-modify} \langle c ;; c' \rangle \ x$   
**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' \wedge mds' = mds \wedge mem' = mem*)  
**apply** *simp*  
**apply** (*metis cxt-eval eval\_w.unannotated eval\_w-simple.seq-stop*)  
**apply** (*metis (lifting) seq-stop-elim*)  
**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' \wedge mds' = mds \wedge mem' = mem*)  
**apply** *simp*  
**apply** (*metis cxt-eval eval\_w.unannotated eval\_w-simple.seq-stop*)  
**apply** (*metis (lifting) seq-stop-elim*)  
**apply** (*metis (lifting, no-types) eval\_w.seq seq-elim*)  
**done**

**lemma** *seq-doesnt-modify*:

$\llbracket \text{doesnt-modify } c \ x \rrbracket \implies \text{doesnt-modify} \langle c ;; c' \rangle \ x$   
**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  $c = \text{Stop}$ )
apply clarsimp
apply (metis (lifting) seq-stop-elim)
apply (metis (no-types) seq-elim)
done

```

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

```

lemma seq-stop-elim:  $\langle \text{Stop} ;; c, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \rangle \implies$ 
 $c' = c \wedge \text{mds}' = \text{mds} \wedge \text{mem}' = \text{mem}$ 
apply (erule seq-stop-elim')
apply (metis evalw.unannotated seq-stop-elim)
apply (metis evalw.seq seq-stop-elim)
by (metis (lifting) Stmt.simps(34) Stmt.simps(42) cxt-seq-elim)

```

```

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

```

```

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

```

```

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

```

```

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

```

**lemma** *while-eval-elim'*:

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

**lemma** *while-doesnt-read*:

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

**lemma** *while-doesnt-modify*:

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

**lemma** *disjE3*:

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

**lemma** *disjE5*:

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

**lemma** *if-doesnt-read'*:

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

**theorem** *mode-type-sound*:

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

**using** *typeable mode-le*

**proof** (*induct arbitrary*:  $\text{ mds}_2 \ \text{ mds}'_2 \ \text{ mem}' \ \text{ mem}$  *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**  $\forall mem. \text{locally-sound-mode-use} \langle (x \leftarrow e) \otimes \text{annos}, mds_2, mem \rangle$   
**unfolding** *locally-sound-mode-use-def*  
**proof** (*clarify*)  
**fix**  $mem\ c'\ mds'\ mem'\ y$   
**assume**  $asm: \langle c', mds', mem' \rangle \in \text{loc-reach} \langle (x \leftarrow e) \otimes \text{annos}, mds_2, mem \rangle$   
**hence**  $c' = (x \leftarrow e) \otimes \text{annos} \wedge mds' = mds_2 \vee c' = \text{Stop} \wedge mds' = mds_2 \oplus$   
*annos*  
**using** *assign-loc-reach* **by** *blast*  
**thus**  $(y \in mds'\ \text{GuarNoReadOrWrite} \longrightarrow \text{doesnt-read-or-modify}\ c'\ y) \wedge$   
 $(y \in mds'\ \text{GuarNoWrite} \longrightarrow \text{doesnt-modify}\ c'\ y)$   
**proof**  
**assume**  $c' = (x \leftarrow e) \otimes \text{annos} \wedge mds' = mds_2$   
**thus** *?thesis*  
**proof** (*safe*)  
**assume**  $nin: y \in mds_2\ \text{GuarNoReadOrWrite}$   
**hence**  $nin: y \in mds\ \text{GuarNoReadOrWrite}$   
**using** *assign.prem*s **unfolding** *le-fun-def* **by** *blast*  
**hence**  $y \notin \text{aexp-vars}\ e$   
**by** (*metis IntD2 IntI assign.hyps(2) assign.prem*s *empty-iff inf-apply*  
*le-iff-inf*)  
**moreover from**  $nin\ \text{assign.hyps}(3)$  **have**  $\mathcal{C}\text{-vars}\ y \cap \text{aexp-vars}\ e = \{\}$   
**by** (*meson contra-subsetD*)  
**moreover from**  $nin\ \text{assign.hyps}$  **have**  $x \notin \mathcal{C}\text{-vars}\ y \wedge x \neq y$   
**by** *blast*  
**ultimately show** *doesnt-read-or-modify*  $((x \leftarrow e) \otimes \text{annos})\ y$   
**using** *assign-doesnt-read*  
**by** *fastforce*  
**next**  
**assume**  $y \in mds_2\ \text{GuarNoWrite}$   
**hence**  $nin: y \in mds\ \text{GuarNoWrite}$   
**using** *assign.prem*s **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*  $((x \leftarrow e) \otimes \text{annos})\ y$   
**by** *blast*  
**qed**  
**next**  
**assume**  $c' = \text{Stop} \wedge mds' = mds_2 \oplus \text{annos}$   
**with** *stop-doesnt-access* **show** *?thesis* **by** *blast*  
**qed**  
**qed**  
**thus** *?case*  
**apply** *clarsimp*  
**by** (*metis assign.prem*s *assign-annos-eval assign-loc-reach stop-no-eval up-*  
*date-preserves-le*)  
**next**  
**case** (*if- mds annos c<sub>1</sub> mds'' c<sub>2</sub> e*)  
**let**  $?c = (\text{If}\ e\ c_1\ c_2) \otimes \text{annos}$   
**from** *if-* **have** *modes-le'*:  $mds_2 \oplus \text{annos} \leq mds \oplus \text{annos}$

```

  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*:  $\bigwedge c' mds' mem' mem.$   
 $\langle c', mds', mem \rangle \in loc\text{-}reach \langle While\ e\ c \otimes annos, mds_2, mem \rangle \implies$   
 $c' = While\ e\ c \otimes annos \wedge mds' = mds_2 \vee$   
 $c' = While\ e\ c \wedge mds' \leq mdsa \vee$   
 $c' = Stmt.If\ e\ (c ;; While\ e\ c)\ Stop \wedge mds' \leq mdsa \vee$   
 $c' = Stop \wedge mds' \leq mdsa \vee$   
 $(\exists c'' mem'' mds_3.$   
 $c' = c'' ;; While\ e\ c \wedge$   
 $mds_3 \leq mdsa \wedge \langle c'', mds', mem \rangle \in loc\text{-}reach \langle c, mds_3, mem'' \rangle)$

**proof** –  
**fix**  $mem\ c' mds' mem'$   
**assume**  $\langle c', mds', mem \rangle \in loc\text{-}reach \langle While\ e\ c \otimes annos, mds_2, mem \rangle$   
**thus** *?thesis*  $c' mds' mem' mem$   
**apply** (*induct rule*: *loc-reach.induct*)  
**apply** *simp-all*  
**apply** (*erule* *disjE*)  
**apply** *simp*  
**apply** (*metis*  $\langle mds_2 \oplus annos \leq mds \oplus annos \rangle$  *while.hyps(1)* *while-eval-elim*)  
**apply** (*erule* *disjE*)  
**apply** (*metis* *while-eval-elim'*)  
**apply** (*erule* *disjE*)  
**apply** *simp*  
**apply** (*metis* *if-eval-elim'* *loc-reach-refl'*)  
**apply** (*erule* *disjE*)  
**apply** (*metis* *stop-no-eval*)  
**apply** (*erule* *exE*)  
**apply** (*rename-tac*  $c' mds' mem' c'' mds'' mem'' c''a$ )  
**apply** (*case-tac*  $c''a = Stop$ )  
**apply** (*insert* *while.hyps(3)*)  
**apply** (*metis* *seq-stop-elim* *while.hyps(3)*)  
**apply** (*metis* *loc-reach.step* *seq-elim*)  
**by** (*metis* (*full-types*) *loc-reach.mem-diff*)

**qed**  
**from** *while* **show** *?case*  
**proof** (*safe*)  
**fix**  $mem$   
**assume**  $\langle Stop, mds_2', mem \rangle \in loc\text{-}reach \langle While\ e\ c \otimes annos, mds_2, mem \rangle$   
**thus**  $mds_2' \leq mds \oplus annos$   
**by** (*metis* *Stmt.distinct(35)* *Stmt.distinct(43)* *annotate.elims* *update-annos.simps(1)*  
*while.hyps(1)* *while.prem*s *while-loc-reach*)

**next**  
**fix**  $mem$   
**from** *while* **have**  $a: bexp\text{-}vars\ e \cap (mds_2 \oplus annos)\ GuarNoReadOrWrite = \{\}$   
**by** (*metis* (*lifting*, *no-types*) *Int-empty-right* *Int-left-commute*  $\langle mds_2 \oplus annos \leq mds \oplus annos \rangle$  *inf-fun-def* *le-iff-inf*)  
**from** *while* **have**  $b: \forall v. C\text{-}vars\ v \cap bexp\text{-}vars\ e \neq \{\} \longrightarrow v \notin (mds_2 \oplus annos)$

*GuarNoReadOrWrite*  
**by** (*meson*  $\langle mds_2 \oplus annos \leq mds \oplus annos \rangle$  *le-fun-def subsetCE*)  
**show** *locally-sound-mode-use*  $\langle \text{While } e \ c \ \otimes \ annos, \ mds_2, \ mem \rangle$   
**unfolding** *locally-sound-mode-use-def*  
**apply** (*rule allI*)  
**apply** (*rule impI*)  
**proof** –  
**fix**  $c' \ mds' \ mem'$   
**define**  $lc$  **where**  $lc = \langle \text{While } e \ c \ \otimes \ annos, \ mds_2, \ mem \rangle$   
**assume**  $\langle c', \ mds', \ mem' \rangle \in \text{loc-reach } lc$   
**thus**  $\forall x. (x \in mds' \ \text{GuarNoReadOrWrite} \longrightarrow \text{doesnt-read-or-modify } c' \ x) \wedge$   
 $(x \in mds' \ \text{GuarNoWrite} \longrightarrow \text{doesnt-modify } c' \ x)$   
**apply** (*simp add: lc-def*)  
**apply** (*drule while-loc-reach*)  
**apply** (*rule allI*)  
**apply** (*erule disjE5*)  
**proof** (*clarsimp*)  
**fix**  $x$   
**show**  $(x \in mds_2 \ \text{GuarNoReadOrWrite} \longrightarrow \text{doesnt-read-or-modify } (\text{While } e \ c \ \otimes \ annos) \ x) \wedge$   
 $(x \in mds_2 \ \text{GuarNoWrite} \longrightarrow \text{doesnt-modify } (\text{While } e \ c \ \otimes \ annos) \ x)$   
**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**  $a: c' = \text{Stmt.If } e \ (c \ ; \ ; \ \text{While } e \ c) \ \text{Stop} \wedge mds' \leq mdsa$   
**hence**  $mds' \leq mdsa$  **by** *blast*  
**let**  $?c' = \text{If } e \ (c \ ; \ ; \ \text{While } e \ c) \ \text{Stop}$   
**have**  $x \in mds' \ \text{GuarNoReadOrWrite} \longrightarrow \text{doesnt-read-or-modify } ?c' \ x$   
**apply** *clarify*  
**apply** (*rule if-doesnt-read'*)  
**apply** (*metis IntI*  $\langle mds' \leq mdsa \rangle$  *empty-iff le-fun-def rev-subsetD*  
*while.hyps(1) while.hyps(4)*)  
**by** (*metis IntI*  $\langle mds' \leq mdsa \rangle$  *empty-iff le-fun-def rev-subsetD* *while.hyps(1)*  
*while.hyps(5)*)  
**moreover**  
**have**  $x \in mds' \ \text{GuarNoWrite} \longrightarrow \text{doesnt-modify } ?c' \ x$   
**by** (*metis annotate.simps(1) if-doesnt-modify*)  
**ultimately show**  $(x \in mds' \ \text{GuarNoReadOrWrite} \longrightarrow \text{doesnt-read-or-modify}$   
 $c' \ x) \wedge$   
 $(x \in mds' \ \text{GuarNoWrite} \longrightarrow \text{doesnt-modify } c' \ x)$   
**using**  $a$  **by** *simp*  
**next**  
**fix**  $x$   
**assume**  $c' = \text{Stop} \wedge mds' \leq mdsa$   
**thus**  $(x \in mds' \ \text{GuarNoReadOrWrite} \longrightarrow \text{doesnt-read-or-modify } c' \ x) \wedge$   
 $(x \in mds' \ \text{GuarNoWrite} \longrightarrow \text{doesnt-modify } c' \ x)$

```

    by (simp, metis stop-doesnt-access)
next
  fix x
  assume  $\exists c'' mem'' mds_3.$ 
     $c' = c'' ;; While\ e\ c \wedge$ 
     $mds_3 \leq mdsa \wedge$ 
     $\langle c'', mds', mem \rangle$ 
     $\in loc\ reach\ \langle c, mds_3, mem' \rangle$ 
  from this obtain
     $c'' mem'' mds_3$ 
  where  $mds_3 \leq mdsa$  and [simp]:  $c' = c'' ;; While\ e\ c$ 
  and  $\langle c'', mds', mem \rangle \in loc\ reach\ \langle c, mds_3, mem' \rangle$  by blast
  thus  $(x \in mds'\ GuarNoReadOrWrite \longrightarrow doesnt\ read\ or\ modify\ c'\ x) \wedge$ 
     $(x \in mds'\ GuarNoWrite \longrightarrow doesnt\ modify\ c'\ x)$ 
    apply (clarsimp, safe)
      apply (rule seq-doesnt-read)
      apply (insert while(3))
      apply (metis  $\langle mds_3 \leq mdsa \rangle$  locally-sound-mode-use-def while.hyps(1))
      apply (rule seq-doesnt-modify)
      by (metis  $\langle mds_3 \leq mdsa \rangle$  locally-sound-mode-use-def while.hyps(1))
next
  fix x
  assume  $c' = While\ e\ c \wedge mds' \leq mdsa$ 
  thus  $(x \in mds'\ GuarNoReadOrWrite \longrightarrow doesnt\ read\ or\ modify\ c'\ x) \wedge$ 
     $(x \in mds'\ GuarNoWrite \longrightarrow doesnt\ modify\ c'\ x)$ 
    unfolding doesnt-read-or-modify-def doesnt-read-or-modify-vars-def and
doents-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  $\langle Stop, mds_2', mem \rangle \in loc\ reach\ \langle c_1 ;; c_2, mds_2, mem \rangle$ 
    then obtain  $mds_2'' mem''$  where  $\langle Stop, mds_2'', mem'' \rangle \in loc\ reach\ \langle c_1, mds_2,$ 
mem  $\rangle$  and
       $\langle Stop, mds_2', mem \rangle \in loc\ reach\ \langle c_2, mds_2'', mem'' \rangle$ 
    using seq-split by blast
    thus  $mds_2' \leq mds''$ 
    using seq by blast
  next
  fix mem
  from seq show locally-sound-mode-use  $\langle c_1 ;; c_2, mds_2, mem \rangle$ 
    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
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](http://isa-afp.org/entries/SIFUM_Type_Systems.shtml).
- [MSPR16] Toby Murray, Robert Sison, Edward Pierzhalski, 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.