A Dependent Security Type System for Concurrent Imperative Programs

Toby Murray, Robert Sison, Edward Pierzchalski and Christine Rizkallah

August 16, 2018

Abstract

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

The formalization includes the following parts:
• Notion of Dependent SIFUM-security and preliminary concepts:
  Preliminaries.thy, Security.thy
• Compositionality proof: Compositionality.thy
• Example language: Language.thy
• Type system for ensuring Dependent SIFUM-security and soundness proof:
  TypeSystem.thy
• Type system for ensuring sound use of modes and soundness proof: LocallySoundUseOfModes.thy

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

Contents

1 Preliminaries 2

2 Definition of the SIFUM-Security Property 4
  2.1 Evaluation of Concurrent Programs 5
  2.2 Low-equivalence and Strong Low Bisimulations 6
  2.3 SIFUM-Security 8
  2.4 Sound Mode Use 9
3 Compositionality Proof for SIFUM-Security Property 12

4 Language for Instantiating the SIFUM-Security Property 21
   4.1 Syntax ............................................. 21
   4.2 Semantics ......................................... 22
   4.3 Semantic Properties .............................. 24

5 Type System for Ensuring SIFUM-Security of Commands 27
   5.1 Typing Rules ....................................... 27
   5.2 Typing Soundness .................................. 41

6 Type System for Ensuring Locally Sound Use of Modes 52
   6.1 Typing Rules ....................................... 52
   6.2 Soundness of the Type System .................. 53

1 Preliminaries

theory Preliminaries
imports Main
begin

Possible modes for variables:

datatype Mode = AsmNoReadOrWrite | AsmNoWrite | GuarNoReadOrWrite | GuarNoWrite

We consider a two-element security lattice:

datatype Sec = High | Low

Sec forms a (complete) lattice:

instantiation Sec :: complete-lattice
begin

definition top-Sec-def: top = High

definition sup-Sec-def: sup d1 d2 = (if (d1 = High ∨ d2 = High) then High else Low)

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

definition bot-Sec-def: bot = Low

definition less-eq-Sec-def: d1 ≤ d2 = (d1 = d2 ∨ d1 = Low)

definition less-Sec-def: d1 < d2 = (d1 = Low ∧ d2 = High)

definition Sup-Sec-def: Sup S = (if (High ∈ S) then High else Low)

definition Inf-Sec-def: Inf S = (if (Low ∈ S) then Low else High)

instance (proof)
Memories are mappings from variables to values

\[ \text{type-synonym} \quad (\text{'var, 'val}) \text{Mem} = \text{'var} \Rightarrow \text{'val} \]

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

\[ \text{type-synonym} \quad \text{'var Mds} = \text{Mode} \Rightarrow \text{'var set} \]

Local configurations:

\[ \text{type-synonym} \quad (\text{'com, 'var, 'val}) \text{LocalConf} = (\text{'com} \times \text{'var Mds}) \times (\text{'var, 'val}) \text{Mem} \]

Global configurations:

\[ \text{type-synonym} \quad (\text{'com, 'var, 'val}) \text{GlobalConf} = (\text{'com} \times \text{'var Mds}) \text{list} \times (\text{'var, 'val}) \text{Mem} \]

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

\[
\text{locale}sifum-security-init = \\
\text{fixes} \quad \text{dma} :: (\text{'Var, 'Val}) \text{Mem} \Rightarrow \text{'Var} \Rightarrow \text{Sec} \\
\text{fixes} \quad \text{C-vars} :: \text{'Var} \Rightarrow \text{'Var set} \\
\text{fixes} \quad \text{C} :: \text{'Var set} \\
\text{fixes} \quad \text{eval} :: (\text{'Com, 'Var, 'Val}) \text{LocalConf rel} \\
\text{fixes} \quad \text{some-val} :: \text{'Val} \\
\text{fixes} \quad \text{INIT} :: (\text{'Var, 'Val}) \text{Mem} \Rightarrow \text{bool} \\
\text{assumes} \quad \text{deterministic}: \quad [ (\text{lc}, \text{lc'}) \in \text{eval}; (\text{lc}, \text{lc''}) \in \text{eval} ] \Rightarrow \text{lc'} = \text{lc''} \\
\text{assumes} \quad \text{finite-memory}: \quad \text{finite} \{ (\text{x} :: \text{'Var}). \text{True} \} \\
\text{defines} \quad \text{C} \equiv \bigcup x. \text{C-vars} x \\
\text{assumes} \quad \text{C-vars-C}: \quad x \in \text{C} \Rightarrow \text{C-vars} x = \{ \} \\
\text{assumes} \quad \text{dna-C-vars}: \quad \forall x \in \text{C-vars} y. \quad \text{mem}_1 x = \text{mem}_2 x \Rightarrow \text{dna mem}_{1 y} \text{mem}_{2 y} \\
\text{assumes} \quad \text{C-Low}: \quad \forall x \in \text{C}. \quad \text{dna mem} x = \text{Low} \\
\]

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

context sifum-security-init begin

lemma C-vars-subset-C:
\[ \text{C-vars} x \subseteq \text{C} \]
(proof)

lemma dna-C:
\[ \forall x \in C. \text{mem}_1 x = \text{mem}_2 x \Rightarrow \text{dma mem}_1 = \text{dma mem}_2 \]

\[\langle\text{proof}\rangle\]

end

**Lemma** my-trancl-induct [consumes 1, case-names base step]:
\[ [(a, b) \in r^+; \\
\land x\ y. [(x, y) \in r; P x] \Rightarrow P y] \Rightarrow P b \]
\[\langle\text{proof}\rangle\]

**Lemma** my-trancl-step-induct [consumes 1, case-names base step]:
\[ [(a, b) \in r^+; \\
\land x\ y. (x, y) \in r \Rightarrow P x y; \\
\land x\ y\ z. P x y \Rightarrow (y, z) \in r \Rightarrow P x z] \Rightarrow P a b \]
\[\langle\text{proof}\rangle\]

**Lemma** my-trancl-big-step-induct [consumes 1, case-names base step]:
\[ [(a, b) \in r^+; \\
\land x\ y. (x, y) \in r \Rightarrow P x y; \\
\land x\ y\ z. (x, y) \in r \Rightarrow P x y \Rightarrow (y, z) \in r \Rightarrow P x z] \Rightarrow P a b \]
\[\langle\text{proof}\rangle\]

**Lemmas** my-trancl-step-induct3 = my-trancl-step-induct [of \((a\ x, a\ y)\) \(\Rightarrow\) \((b\ x, b\ y)\), split-format (complete), consumes 1, case-names step]

**Lemmas** my-trancl-big-step-induct3 = my-trancl-big-step-induct [of \((a\ x, a\ y)\) \(\Rightarrow\) \((b\ x, b\ y)\), 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 \to ('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 =

\(\lambda x. \text{case} (A x) \text{ of} \\
\text{Some} (v_1, v_2) \Rightarrow \text{if} \text{first then} v_1 \text{ else} v_2 \\
\text{None} \Rightarrow \text{mem} x)\)
abbreviation apply-adaptation₁ ::
(′Var, ′Val) Mem ⇒ (′Var, ′Val) adaptation ⇒ (′Var, ′Val) Mem
(- [ ] [900, 0] 1000)
where mem [ ] A ≡ apply-adaptation True mem A

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

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

context sifum-security-init begin

2.1 Evaluation of Concurrent Programs

abbreviation eval-abv :: (′Com, ′Var, ′Val) LocalConf ⇒ (-, -, -) LocalConf ⇒ bool
(infixl → 70)
where
x → y ≡ (x, y) ∈ eval

abbreviation conf-abv :: ′Com ⇒ ′Var Mds ⇒ (′Var, ′Val) Mem ⇒ (′Var, ′Val) Mem ⇒ (-, -, -) LocalConf
((-, -, -) [0, 0, 0] 1000)
where
( c, mds, mem ) ≡ ((c, mds), mem)

inductive-set meval :: ((-, -, -) GlobalConf × nat × (-, -, -) GlobalConf) set
and meval-abv :: - ⇒ - ⇒ - ⇒ bool (- →, - 70)
where
conf → k conf’ ≡ (conf, k, conf’) ∈ meval |
meval-intro [iff]: ( cms ! n, mem ) → ( cm’, mem’ ); n < length cms ] →
((cms, mem), n, (cms [n := cm’], mem’)) ∈ meval

inductive-cases meval-elim [elim]: ((cms, mem), k, (cms’, mem’)) ∈ meval

inductive neval :: (′Com, ′Var, ′Val) LocalConf ⇒ nat ⇒ (-, -, -) LocalConf ⇒ bool
(infixl → 70)
where
neval-0: x = y ⇒ x →⁰ y |
neval-S-n: x → y ⇒ y →ⁿ z ⇒ x →ⁿSuc n z

inductive-cases neval-ZeroE: neval 0 y

5
inductive-cases neval-SucE: neval x (Suc n) y

lemma neval-det:
- x ®^n y ⇒ x ®^n y' ⇒ y = y'
(proof)

lemma neval-Suc-simp [simp]:
- neval x (Suc 0) y = x ® y
(proof)

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

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

abbreviation meval-sched-abv :: (-, -, -) GlobalConf ⇒ nat list ⇒ (-, -, -) GlobalConf ⇒ bool (- → - 70)
where
- c ®^ns c' ≡ meval-sched ns c c'

lemma meval-sched-det:
- meval-sched ns c c' ⇒ meval-sched ns c c'' ⇒ c' = c''
(proof)

2.2 Low-equivalence and Strong Low Bisimulations

definition low-eq :: ('Var, 'Val) Mem ⇒ (-, -) Mem ⇒ bool (infixl =l 80)
where
- mem1 =l mem2 ≡ (∀ x. dma mem1 x = Low ⇒ mem1 x = mem2 x)

definition low-mds-eq :: 'Var Mds ⇒ ('Var, 'Val) Mem ⇒ (-, -) Mem ⇒ bool (- =l [-100, 100] 80)
where
- (mem1 =mds l mem2) ≡ (∀ x. dma mem1 x = Low ∧ (x ∈ C ∨ x /∈ mds Asm-NoReadOrWrite) ⇒ mem1 x = mem2 x)

lemma low-eq-low-mds-eq:
(mem₁ =^l mem₂) = (mem₁ =^l (λm. { }) mem₂)  
⟨proof⟩

**Lemma** low-mds-eq-dma:

\[(mem₁ = mds mem₂) \implies dma mem₁ = dma mem₂\]

⟨proof⟩

**Lemma** low-mds-eq-sym:

\[(mem₁ = mds mem₂) \implies (mem₂ = mds mem₁)\]

⟨proof⟩

**Lemma** low-eq-sym:

\[(mem₁ = l mem₂) \implies (mem₂ = l mem₁)\]

⟨proof⟩

**Lemma** [simp]: mem =^l mem' \implies mem = mds mem'

⟨proof⟩

**Lemma** [simp]: (\forall mds. mem = mds mem') \implies mem =^l mem'

⟨proof⟩

**Lemma** High-not-in-C [simp]:

\[dma mem₁ x = High \implies x \notin C\]

⟨proof⟩

**Definition**

closed-glob-consistent :: ((('Com', 'Var', 'Val) LocalConf)) rel ⇒ bool

**Where**

closed-glob-consistent \( R = \)

(\forall c₁ mds mem₁ c₂ mem₂. ((c₁, mds, mem₁), (c₂, mds, mem₂)) \in R →

(\forall A, (\forall x. \text{case } A \text{ of } \text{Some } (v,v') \Rightarrow (mem₁ x \neq v \lor mem₂ x \neq v') \Rightarrow \neg \text{var-asm-not-written mds } x) \land

(\forall x_0. \text{dma } (mem₁ \parallel 1 A) x \neq \text{dma mem₁ } x \rightarrow \neg \text{var-asm-not-written mds } x) \land

(\forall x_0. \text{dma } (mem₁ \parallel 1 A) x = \text{Low} \land (x \notin mds \text{ AsmNoReadOrWrite } \lor x \in C) \rightarrow (\text{mem₁ } \parallel 1 A) x = (\text{mem₂ } \parallel 2 A) x) \rightarrow

((c₁, mds, mem₁[|₁ A ]), (c₂, mds, mem₂[|₂ A ])) \in R))

**Definition**

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

**Where**

strong-low-bisim-mm \( R \equiv \)

sym \( R \land \)

closed-glob-consistent \( R \land \)

(\forall c₁ mds mem₁ c₂ mem₂. ((c₁, mds, mem₁), (c₂, mds, mem₂)) \in R →

(mem₁ = mds mem₂) \land

(\forall c₁' mds' mem₁'. (c₁, mds, mem₁) \leadsto (c₁', mds', mem₁')) →

(\forall c₁ mds mem₁ c₂ mem₂. ((c₁, mds, mem₁), (c₂, mds, mem₂)) \in R →

(mem₁ = mds mem₂) \land

(\forall c₁' mds' mem₁'. (c₁, mds, mem₁) \leadsto (c₁', mds', mem₁')) →

\( (mem₁ = mds mem₂) \land \)

(\forall c₁' mds' mem₁'. (c₁, mds, mem₁) \leadsto (c₁', mds', mem₁')) →

7
(\exists c_2', \text{mem}_2'. (c_2', \text{mds}, \text{mem}_2') \rightarrow (c_2', \text{mds}', \text{mem}_2') \land \\
(\langle c_1', \text{mds}', \text{mem}_1'\rangle, (c_2', \text{mds}', \text{mem}_2'\rangle) \in R))

\textbf{inductive-set mm-equiv :: ('Com, 'Var, 'Val) LocalConf rel}

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

\textbf{('Com, 'Var, 'Val) LocalConf \Rightarrow bool (infix \approx 60)

where}

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

\text{mm-equiv-intro \} if: \} strong-low-bisis-mm R ; (lc_1, lc_2) \in R \} \implies (lc_1, lc_2) \in mm-equiv

\textbf{inductive-cases mm-equiv-elim [elim]: (c_1, \text{mds}, \text{mem}_1) \approx (c_2, \text{mds}, \text{mem}_2)}

\textbf{definition low-indistinguishable :: 'Var Mds \Rightarrow 'Com \Rightarrow 'Com \Rightarrow bool}

\text{(- \sim - [100, 100] 80) where}

\text{c_1 \sim_{mds} c_2 = (\forall \text{mem}_1 \text{mem}_2. \text{mem}_1 =_{mds}\text{mem}_2 \rightarrow (c_1, \text{mds}, \text{mem}_1) \approx (c_2, \text{mds}, \text{mem}_2))}

\textbf{2.3 SIFUM-Security}

\textbf{definition}

\text{com-sifum-secure :: 'Com \times 'Var Mds \Rightarrow bool}

\text{where}

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

\textbf{definition}

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

\text{where}

\text{prog-sifum-secure-cont cmds =}

(\forall \text{mem}_1 \text{mem}_2. \text{INIT mem}_1 \land \text{INIT mem}_2 \land \text{mem}_1 =_{mds} \text{mem}_2 \rightarrow 
(\forall \text{sched cms}_1' \text{mem}_1').

(\text{cmds}, \text{mem}_1) \rightarrow_{\text{sched}} (\text{cms}_1', \text{mem}_1') \rightarrow 
(\exists \text{cms}_2' \text{mem}_2'. (\text{cmds}, \text{mem}_2) \rightarrow_{\text{sched}} (\text{cms}_2', \text{mem}_2') \land 
\text{map snd \text{cms}_1' = map snd \text{cms}_2' \land }

\text{length \text{cms}_2' = length \text{cms}_1' \land }

(\forall x. \text{dma \text{mem}_1'} x = \text{Low} \land (x \in C \lor (\forall i < \text{length \text{ cms}_1'}. 
\text{x} \notin \text{ snd (\text{ cms}_1' ! i) AsmNoReadOrWrite}) \rightarrow \text{mem}_1' x = 
\text{mem}_2' x)))))

\textbf{lemma prog-sifum-secure-cont-def2:

prog-sifum-secure-cont cmds =}

(\forall \text{mem}_1 \text{mem}_2. \text{INIT mem}_1 \land \text{INIT mem}_2 \land \text{mem}_1 =_{mds} \text{mem}_2 \rightarrow 
(\forall \text{sched cms}_1' \text{mem}_1'.

(\text{cmds}, \text{mem}_1) \rightarrow_{\text{sched}} (\text{cms}_1', \text{mem}_1') \rightarrow 
(\exists \text{cms}_2' \text{mem}_2'. (\text{cmds}, \text{mem}_2) \rightarrow_{\text{sched}} (\text{cms}_2', \text{mem}_2') \land 
(\forall c \text{ms}_2' \text{mem}_2'. (\text{cmds}, \text{mem}_2) \rightarrow_{\text{sched}} (\text{cms}_2', \text{mem}_2') \land 
\text{map snd \text{cms}_1' = map snd \text{cms}_2' \land }

\text{}})
\[ \text{length } \text{cms}_2' = \text{length } \text{cms}_1' \wedge \\
(\forall x. \text{dma mem}_1' x = \text{Low} \wedge (x \in C \lor (\forall i < \text{length } \text{cms}_1'. \\
\quad x \notin \text{snd} (\text{cms}_1' ! i) \text{ AsmNoReadOrWrite}) \rightarrow \text{mem}_1' x = \\
\quad \text{mem}_2' x)) \]

\[ \langle \text{proof} \rangle \]

### 2.4 Sound Mode Use

**definition**

\[ \text{subst} :: (a \Rightarrow 'b) \Rightarrow (a \Rightarrow 'b) \Rightarrow (a \Rightarrow 'b) \]

**where**

\[ \text{subst } f \text{ mem} = (\lambda x. \text{case } f \text{ x of} \\
\quad \text{None } \Rightarrow \text{mem } x \mid \\
\quad \text{Some } v \Rightarrow v) \]

**abbreviation**

\[ \text{subst-abv} :: (a \Rightarrow 'b) \Rightarrow (a \Rightarrow 'b) \Rightarrow (a \Rightarrow 'b) \]

**where**

\[ f \; [\mapsto \sigma] \equiv \text{subst } \sigma \; f \]

**lemma**

\[ \text{subst-not-in-dom} : [x \notin \text{dom } \sigma ] \Rightarrow \text{mem } [\mapsto \sigma] x = \text{mem } x \]

\[ \langle \text{proof} \rangle \]

**definition**

\[ \text{doesnt-read-or-modify-vars} :: '\text{Com} \Rightarrow '\text{Var set} \Rightarrow \text{bool} \]

**where**

\[ \text{doesnt-read-or-modify-vars} \; c \; X = (\forall \; \text{mds mem c' mds'} \; \text{mem}'. \\
\quad \langle c', \text{mds'}, \text{mem}' \rangle \rightarrow (\langle c', \text{mds'}, \text{mem}' \rangle) \rightarrow \\
\quad ((\forall x \in X. (\forall v. (c, \text{mds}, \text{mem} (x := v)) \rightarrow (c', \text{mds'}, \text{mem}' (x := v))))) \]

**definition**

\[ \text{vars-C} :: '\text{Var set} \Rightarrow '\text{Var set} \]

**where**

\[ \text{vars-C} \; X \equiv \bigcup \; x \in X. \; \text{C-vars } x \]

**lemma**

\[ \text{vars-C-subset-C} : \]

\[ \text{vars-C} \; X \subseteq C \]

\[ \langle \text{proof} \rangle \]

**definition**

\[ \text{doesnt-read-or-modify} :: '\text{Com} \Rightarrow '\text{Var} \Rightarrow \text{bool} \]

**where**

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

**definition**

\[ \text{doesnt-modify} :: '\text{Com} \Rightarrow '\text{Var} \Rightarrow \text{bool} \]

**where**

\[ \forall x. \text{dma mem}_1' x = \text{Low} \wedge (x \in C \\
\quad \lor (\forall i < \text{length } \text{cms}_1'. \\
\quad x \notin \text{snd} (\text{cms}_1' ! i) \text{ AsmNoReadOrWrite}) \rightarrow \text{mem}_1' x = \\
\quad \text{mem}_2' x) \]

\[ \langle \text{proof} \rangle \]
\[ \text{doesn't-modify } c \ x = (\forall \ mds \ mem \ c' \ mds' \ mem'. (\langle c, mds, mem \rangle \Rightarrow \langle c', mds', mem' \rangle)) \implies \text{mem } x = \text{mem'} x \land \text{dma mem } x = \text{dma mem'} x \]

**Lemma noread-nowrite:**
- **Assumes** step: \( (c, mds, mem) \Rightarrow (c', mds', mem') \)
- **Assumes** noread: \( (\forall v. (c, mds, \text{mem}(x := v)) \Rightarrow (c', mds', \text{mem}'(x := v))) \)
- **Shows** \( \text{mem } x = \text{mem'} x \)

**Proof**

**Lemma doesn't-read-or-modify-doesn't-modify:**
- doesn't-read-or-modify \( c \ x \) \( \Rightarrow \) doesn't-modify \( c \ x \)

**Proof**

**Inductive Set**

\[ \text{loc-reach} :: ('Com, 'Var, 'Val) \text{LocalConf} \Rightarrow ('Com, 'Var, 'Val) \text{LocalConf set} \]

For \( lc :: (-, -, -) \text{LocalConf} \)

- **Reflexivity**
  \[ \text{refl} : \{ \text{fst} (\text{fst } lc), \text{snd} (\text{fst } lc), \text{snd } lc \} \in \text{loc-reach } lc \]
- **Step**
  \[ \{ c', mds', mem' \} \in \text{loc-reach } lc \]
  \[ (c', mds', mem') \Rightarrow (c'', mds'', mem'') \] \( \Rightarrow \)
  \[ (c'', mds'', mem'') \in \text{loc-reach } lc \]

- **Mem-diff**
  \[ (\forall x. \text{var-asm-not-written } mds' x \Rightarrow \text{mem'} x = \text{mem''} x \land \text{dma mem} x = \text{dma mem''} x) \] \( \Rightarrow \)
  \[ \{ c', mds', mem'' \} \in \text{loc-reach } lc \]

**Lemma neval-loc-reach:**
- neval \( lc' \ n \) \( lc'' \) \( \Rightarrow \) \( lc' \in \text{loc-reach } lc \) \( \Rightarrow \) \( lc'' \in \text{loc-reach } lc \)

**Proof**

**Definition**

\[ \text{locally-sound-mode-use} :: (-, -, -) \text{LocalConf} \Rightarrow \text{bool} \]

- **Where**
  \[ \text{locally-sound-mode-use } lc = \]
  \[ (\forall c' mds' mem'. (c', mds', mem') \in \text{loc-reach } lc \Rightarrow \]
  \[ (\forall x. (x \in mds' \text{GuarNoReadOrWrite } \Rightarrow \text{doesn't-read-or-modify } c' x) \land \]
  \[ (x \in mds' \text{GuarNoWrite } \Rightarrow \text{doesn't-modify } c' x))) \]

**Definition**

\[ \text{respects-own-guarantees} :: ('Com \times 'Var Mds) \Rightarrow \text{bool} \]

- **Where**
  \[ \text{respects-own-guarantees } cm \equiv \]
  \[ (\forall x. (x \in (\text{snd } cm) \text{GuarNoReadOrWrite } \Rightarrow \text{doesn't-read-or-modify } (\text{fst } cm) x) \land \]
\((x \in (\text{snd \ cm}) \text{ GuarNoWrite} \rightarrow \text{doesnt-modify\ (fst \ cm \ x)})\)

**Lemma** locally-sound-mode-use-def2:
locally-sound-mode-use \(lc \equiv \forall \text{lc}' \in \text{loc-reach \ lc}. \text{respects-own-guarantees (fst \ lc')}
\)
\(\langle \text{proof} \rangle\)

**Lemma** locally-sound-respects-guarantees:
locally-sound-mode-use \((cm, \text{mem}) \implies \text{respects-own-guarantees cm}
\)
\(\langle \text{proof} \rangle\)

**Definition**
compatible-modes :: \((\text{\'Var \ Mds}) \text{ list} \Rightarrow \text{bool}\)
where
compatible-modes \(mdss \equiv \forall (i :: \text{nat}) \ x. i < \text{length \ mdss} \rightarrow\)
\((x \in (\text{mdss} ! i) \text{ AsmNoReadOrWrite} \rightarrow\)
\((\forall j < \text{length \ mdss}. j \neq i \rightarrow x \in (\text{mdss} ! j) \text{ GuarNoReadOrWrite})) \land\)
\((x \in (\text{mdss} ! i) \text{ AsmNoWrite} \rightarrow\)
\((\forall j < \text{length \ mdss}. j \neq i \rightarrow x \in (\text{mdss} ! j) \text{ GuarNoWrite}))\)

**Definition**
reachable-mode-states :: \((\text{\'Com, \'Var, \'Val}) \text{ GlobalConf} \Rightarrow ((\text{\'Var \ Mds}) \text{ list}) \text{ set}\)
where
reachable-mode-states \(gc \equiv\)
\(\{ \text{mdss}. (\exists \text{cms' mem' sched. gc} \rightarrow \text{sched (cms', mem')} \land \text{map snd cms'} = \text{mdss})\}\)

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

**Primrec**
sound-mode-use :: \((-,-,-) \text{ GlobalConf} \Rightarrow \text{bool}\)
where
sound-mode-use \((\text{cms, mem}) =\)
\((\text{list-all (\lambda cm. locally-sound-mode-use (cm, mem)) cms} \land\)
\(\text{globally-sound-mode-use (cms, mem)})\)

**Lemma** mm-equiv-sym:
assumes equivalent: \(\langle c_1, \text{mds}_1, \text{mem}_1 \rangle \approx \langle c_2, \text{mds}_2, \text{mem}_2 \rangle\)
shows \(\langle c_2, \text{mds}_2, \text{mem}_2 \rangle \approx \langle c_1, \text{mds}_1, \text{mem}_1 \rangle\)
\(\langle \text{proof} \rangle\)

**Lemma** low-indistinguishable-sym: \(lc \sim_{\text{mds}} \text{lc}' \implies \text{lc'} \sim_{\text{mds}} \text{lc}\)
\(\langle \text{proof} \rangle\)

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

11
lemma mm-equiv-strong-low-bisim: strong-low-bisim-mm mm-equiv

3 Compositionality Proof for SIFUM-Security Property

theory Compositionality
imports Security
begin

context sifum-security-init
begin

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

definition differing-vars-lists :: ('Var', 'Val) Mem ⇒ (\_, \_) Mem ⇒ ((\_, \_) Mem × (\_, \_) Mem) list ⇒ nat ⇒ 'Var set
where differing-vars-lists mem_1 mem_2 mems i ≡ (differing-vars mem_1 (fst (mems ! i)) union differing-vars mem_2 (snd (mems ! i)))

lemma differing-finite: finite (differing-vars mem_1 mem_2)
(proof)

lemma differing-lists-finite: finite (differing-vars-lists mem_1 mem_2 mems i)
(proof)

fun makes-compatible ::
('Com', 'Var', 'Val) GlobalConf ⇒
('Com', 'Var', 'Val) GlobalConf ⇒
((\_, \_) Mem × (\_, \_) Mem) list ⇒ bool
where makes-compatible (cms_1, mem_1) (cms_2, mem_2) mems =
(length cms_1 = length cms_2 ∧ length cms_1 = length mems ∧
(\forall i. i < length cms_1 →→...
\((\forall \sigma. \text{dom } \sigma = \text{differing-vars-lists mem}_1 \text{ mem}_2 \text{ mems } i \rightarrow\) \\
\quad (\text{cms}_1 ! i, (\text{fst } (\text{mems } ! i)) \rightarrow \sigma]) \approx (\text{cms}_2 ! i, (\text{snd } (\text{mems } ! i)) \rightarrow \sigma))\) \\
\quad (\forall x. (\text{mem}_1 x = \text{mem}_2 x \lor \text{dma mem}_1 x = \text{High } \lor x \in C) \rightarrow x \notin \text{differing-vars-lists mem}_1 \text{ mem}_2 \text{ mems } i)\) \\
\quad ((\text{length cms}_1 = 0 \land \text{mem}_1 = \text{mem}_2) \lor (\forall x. \exists i. i < \text{length cms}_1 \land x \notin \text{differing-vars-lists mem}_1 \text{ mem}_2 \text{ mems } i))\)

\textbf{lemma makes-compatible-intro [intro]:} \\
\quad [ \text{length cms}_1 = \text{length cms}_2 \land \text{length cms}_1 = \text{length mems}; \\
\quad (\land i \sigma. [ i < \text{length cms}_1; \text{dom } \sigma = \text{differing-vars-lists mem}_1 \text{ mem}_2 \text{ mems } i ] \rightarrow (\text{cms}_1 ! i, (\text{fst } (\text{mems } ! i)) \rightarrow \sigma]) \approx (\text{cms}_2 ! i, (\text{snd } (\text{mems } ! i)) \rightarrow \sigma)); \\
\quad (\land i x. [ i < \text{length cms}_1; \text{mem}_1 x = \text{mem}_2 x \lor \text{dma mem}_1 x = \text{High } \lor x \in C ] \rightarrow x \notin \text{differing-vars-lists mem}_1 \text{ mem}_2 \text{ mems } i); \\
\quad (\text{length cms}_1 = 0 \land \text{mem}_1 = \text{mem}_2) \lor (\forall x. \exists i. i < \text{length cms}_1 \land x \notin \text{differing-vars-lists mem}_1 \text{ mem}_2 \text{ mems } i)] \rightarrow \\
\quad \text{makes-compatible } (\text{cms}_1, \text{mem}_1) (\text{cms}_2, \text{mem}_2) \text{ mems} \) \\
\quad (\text{proof})

\textbf{lemma compat-low:} \\
\quad [ \text{makes-compatible } (\text{cms}_1, \text{mem}_1) (\text{cms}_2, \text{mem}_2) \text{ mems}; \\
\quad i < \text{length cms}_1; \\
\quad x \in \text{differing-vars-lists mem}_1 \text{ mem}_2 \text{ mems } i ] \rightarrow \text{dma mem}_1 x = \text{Low} \) \\
\quad (\text{proof})

\textbf{lemma compat-different:} \\
\quad [ \text{makes-compatible } (\text{cms}_1, \text{mem}_1) (\text{cms}_2, \text{mem}_2) \text{ mems}; \\
\quad i < \text{length cms}_1; \\
\quad x \in \text{differing-vars-lists mem}_1 \text{ mem}_2 \text{ mems } i ] \rightarrow \text{mem}_1 x \neq \text{mem}_2 x \land \text{dma mem}_1 x = \text{Low } \land x \notin C \) \\
\quad (\text{proof})

\textbf{lemma sound-modes-no-read :} \\
\quad [ \text{sound-mode-use } (\text{cms}, \text{mem}); x \in (\text{map snd cms } ! i) \text{ GuarNoReadOrWrite}; i < \text{length cms } ] \implies \text{doesn't-read-or-modify } (\text{fst } (\text{cms } ! i)) x \) \\
\quad (\text{proof})

\textbf{lemma differing-vars-neg: } x \notin \text{differing-vars-lists mem}_1 \text{ mem}_2 \text{ mems } i \implies \\
\quad (\text{fst } (\text{mems } ! i) x = \text{mem}_1 x \land \text{snd } (\text{mems } ! i) x = \text{mem}_2 x) \) \\
\quad (\text{proof})

\textbf{lemma differing-vars-neg-intro:} \\
\quad [ \text{mem}_1 x = \text{fst } (\text{mems } ! i) x; \\
\quad \text{mem}_2 x = \text{snd } (\text{mems } ! i) x ] \implies x \notin \text{differing-vars-lists mem}_1 \text{ mem}_2 \text{ mems } i
lemma differing-vars-elim [elim]:
\[ x \in \text{differing-vars-lists mem}_1 \ \text{mem}_2 \ \text{mems} \ i \implies \]
\[ (\text{fst} (\text{mems} \ ! \ i) \neq \text{mem}_1 \ x) \lor (\text{snd} (\text{mems} \ ! \ i) \neq \text{mem}_2 \ x) \]
(\text{proof})

lemma makes-compatible-dma-eq:
\[ \text{assumes compat: makes-compatible (cms}_1, \ \text{mem}_1) (cms}_2, \ \text{mem}_2) \ \text{mems} \]
\[ \text{assumes ile: } i < \text{length cms}_1 \]
\[ \text{assumes dom}\sigma; \ \text{dom} \ \sigma = \text{differing-vars-lists mem}_1 \ \text{mem}_2 \ \text{mems} \ i \]
\[ \text{shows dma \ ((fst (mems \ ! \ i)) \mapsto \sigma)} = \text{dma mem}_1 \]
(\text{proof})

lemma compat-different-vars:
\[ \text{[ \ [ \ \text{fst} (\text{mems} \ ! \ i) \ x = \text{snd} (\text{mems} \ ! \ i) \ x; \]
\text{x \not\in \text{differing-vars-lists mem}_1 \ \text{mem}_2 \ \text{mems} \ i \ ] \implies \]
\text{mem}_1 \ x = \text{mem}_2 \ x \]
(\text{proof})

lemma differing-vars-subst [rule-format]:
\[ \text{assumes dom}\sigma; \ \text{dom} \ \sigma \supseteq \text{differing-vars mem}_1 \ \text{mem}_2 \]
\[ \text{shows mem}_1 \ [\mapsto \sigma] = \text{mem}_2 \ [\mapsto \sigma] \]
(\text{proof})

lemma mm-equiv-low-eq:
\[ \text{[ \ [ \langle c}_1, \ \text{mds}, \ \text{mem}_1 \ \rangle \ \approx \langle c}_2, \ \text{mds}, \ \text{mem}_2 \ \rangle \ ] \implies \text{mem}_1 = \text{mds}^l \ \text{mem}_2 \]
(\text{proof})

lemma globally-sound-modes-compatible:
\[ \text{[ globally-sound-mode-use (cms, mem) ] \implies \compatible-modes (map snd cms) } \]
(\text{proof})

lemma compatible-different-no-read :
\[ \text{assumes sound-modes: sound-mode-use (cms}_1, \ \text{mem}_1) \]
\[ \text{sound-mode-use (cms}_2, \ \text{mem}_2) \]
\[ \text{assumes compat: makes-compatible (cms}_1, \ \text{mem}_1) (cms}_2, \ \text{mem}_2) \ \text{mems} \]
\[ \text{assumes modes-eq: map snd cms}_1 = \text{map snd cms}_2 \]
\[ \text{assumes ile: } i < \text{length cms}_1 \]
\[ \text{assumes x: } x \in \text{differing-vars-lists mem}_1 \ \text{mem}_2 \ \text{mems} \ i \]
\[ \text{shows } \text{doesnt-read-or-modify \ (fst (cms}_1 \ ! \ i)) \ x \ \land \ \text{doesnt-read-or-modify (fst (cms}_2 \ ! \ i)) \ x \]
(\text{proof})

definition vars-and-\text{\textit{C}} :: \ '\text{Var set} \Rightarrow \ '\text{Var set} \]
where
\text{vars-and-\text{\textit{C}}} \ X \equiv X \cup \text{vars-\text{\textit{C}}} \ X
fun change-respecting ::
('Com, 'Var, 'Val) LocalConf ⇒
('Com, 'Var, 'Val) LocalConf ⇒
'Var set ⇒ bool
where change-respecting (cms, mem) (cms', mem') X =
((cms, mem) ⊢ (cms', mem') ∧
(∀ σ. dom σ = vars-and-C X −→ (cms, mem [σ ↦]) ⊢ (cms', mem' [σ ↦ σ])))

lemma subst-overrides: dom σ = dom τ ⇒ mem [σ ↦ τ] [σ ↦ σ] = mem [σ ↦ σ]
⟨proof⟩

definition to-partial :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b)
where to-partial f = λ x. Some (f x)
lemma dom-restrict-total: dom (to-partial f |[' X) = X
⟨proof⟩

lemma change-respecting-doesnt-modify':
assumes eval: (cms, mem) ⊢ (cms', mem')
assumes cr: ∀ f. dom f = Y −→ (cms, mem [σ ↦ f]) ⊢ (cms', mem' [σ ↦ f])
assumes x-in-dom: x ∈ Y
shows mem x = mem' x
⟨proof⟩

lemma change-respecting-subset':
assumes step: (cms, mem) ⊢ (cms', mem')
assumes norvd: (∀ σ. dom σ = X −→ (cms, mem [σ ↦ σ]) ⊢ (cms', mem' [σ ↦ σ]))
assumes dom-subset: dom σ ⊆ X
shows (cms, mem [σ ↦ σ]) ⊢ (cms', mem' [σ ↦ σ])
⟨proof⟩

lemma change-respecting-subst:
change-respecting (cms, mem) (cms', mem') X ⇒
(∀ σ. dom σ = X −→ (cms, mem [σ ↦ σ]) ⊢ (cms', mem' [σ ↦ σ]))
⟨proof⟩

lemma change-respecting-intro [iff]:
[ (⟨ c, mds, mem ⟩ ⊢ ⟨ c', mds', mem' ⟩;
  (λ f. dom f = vars-and-C X ⇒
  (⟨ c, mds, mem [σ ↦ f] ⟩ ⊢ ⟨ c', mds', mem' [σ ↦ f] ⟩)) ]
⇒ change-respecting ⟨ c, mds, mem ⟩ ⟨ c', mds', mem' ⟩ X
⟨proof⟩

lemma vars-C-mono:
X ⊆ Y ⇒ vars-C X ⊆ vars-C Y

15
lemma vars-C-Un:
var-C (X ∪ Y) = (var-C X ∪ var-C Y)

lemma vars-C-insert:
var-C (insert x Y) = (var-C {x}) ∪ (var-C Y)

lemma vars-C-empty[simp]:
var-C {} = {}

lemma C-vars-of-C-vars-empty:
x ∈ C-vars y ⟹ C-vars x = {}

lemma vars-and-C-mono:
X ⊆ X' ⟹ var-and-C X ⊆ var-and-C X'

lemma C-vars-finite[simp]:
finit (C-vars x)

lemma finite-dom:
finit (dom (σ::Var ⇒ 'Val option))

lemma doesn't-read-or-modify-subst:
assumes noread: doesn't-read-or-modify c x
assumes step: (c, mds, mem) → (c', mds', mem')
assumes subset: X ⊆ {x} ∪ C-vars x
shows ⋀ x. dom σ = X ⟹ (c, mds, mem↦→ σ) → (c', mds', mem'↦→ σ)

lemma subst-restrict-twice:
dom σ = A ∪ B ⟹
mem [↦→ (σ | ' A)] [↦→ (σ | ' B)] = mem [↦→ σ]

lemma noread-exists-change-respecting:
assumes fin: finite (X :: 'Var set)
assumes eval: (c, mds, mem) → (c', mds', mem')
assumes noread: ∀ x ∈ X. doesn't-read-or-modify c x
shows change-respecting (c, mds, mem) (c', mds', mem' ↦ X

(\proof)

lemma noread-exists-change-respecting:
assumes fin: finite (X :: 'Var set)
assumes eval: (c, mds, mem) → (c', mds', mem')
assumes noread: ∀ x ∈ X. doesn't-read-or-modify c x
shows change-respecting (c, mds, mem) (c', mds', mem' ↦ X

(\proof)
lemma update-nth-eq:
\[ [ \, xs = ys \, ; \, n < \text{length} \, xs \, ] \implies xs = ys \, [ n := xs \, ! \, n ] \]
(proof)

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: snd \, \text{cms}_1 = snd \, \text{cms}_2
assumes step: (\text{cms}_1, \text{mem}_1) \rightsquigarrow (\text{cms}_1', \text{mem}_1')
shows \exists \, e_2', \text{mem}_2'. (\text{cms}_2, \text{mem}_2) \rightsquigarrow (e_2', \text{snd} \, \text{cms}_1', \text{mem}_2') \land
(\text{cms}_1', \text{mem}_1') \approx (e_2', \text{snd} \, \text{cms}_1', \text{mem}_2 ')
(proof)

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
(proof)

lemma change-respecting-doesnt-modify-dma:
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
shows \text{dma mem} \, x = \text{dma mem}' \, x
(proof)

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

lemma differing-empty-eq:
\[ [ \, \text{differing-vars mem mem}' = \{\} \, ] \implies \text{mem} = \text{mem}' \]
(proof)

lemma adaptation-finite:
   finite (dom (A::('Var,'Val) adaptation))
   (proof)

definition globally-consistent :: ('Var,'Val) adaptation ⇒ 'Var Mds ⇒ ('Var,'Val) Mem
   ⇒ ('Var,'Val) Mem ⇒ bool
   where globally-consistent A mds mem_1 mem_2 ≡
   (∀ \, x. \, \text{case} \, A \, x \, \text{of} \, \text{Some} \, (v,v') \Rightarrow (\text{mem}_1 \, x \neq v \, \lor \, \text{mem}_2 \, x \neq v') \implies \neg \, \text{var-asm-not-written mds} \, x \, |\, - \Rightarrow \text{True}) \land
   (∀ \, x. \, \text{dma mem}_1 \, [||_1 \, A] \, x \neq \text{dma mem}_1 \, x \implies \neg \, \text{var-asm-not-written mds} \, x) \land
\[ (\forall x. \text{dma} (\text{mem}_1 [1] A) \ x = \text{Low} \land (x \notin \text{mds AsmNoReadOrWrite}) \lor x \in C) \rightarrow (\text{mem}_1 [1] A) x = (\text{mem}_2 [2] A) x \]

**Lemma globally-consistent-adapt-bisim:**

** Assumes bisim: \( \langle c_1, \text{mds}, \text{mem}_1 \rangle \approx \langle c_2, \text{mds}, \text{mem}_2 \rangle \)**

** Assumes globally-consistent: globally-consistent \( A \) \( \text{mds} \) \( \text{mem}_1 \) \( \text{mem}_2 \)**

** Shows \( \langle c_1, \text{mds}, \text{mem}_1 [1] A \rangle \approx \langle c_2, \text{mds}, \text{mem}_2 [2] A \rangle \)**

**Proof**

**Lemma mm-equiv-C-eq:**

\((a, b) \approx (a', b') \Rightarrow \text{snd } a = \text{snd } a' \Rightarrow \forall x \in C. \ b x = b' x \)**

**Proof**

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

\( x \notin \text{dom } A \Rightarrow \text{apply-adaptation } b \text{ blah } A x = \text{blah } x \)**

**Proof**

**Lemma makes-compatible-invariant:**

** Assumes sound-modes: sound-mode-use \( \langle c_1, \text{mds}, \text{mem}_1 \rangle \)**

** Assumes sound-mode-use \( \langle c_2, \text{mds}, \text{mem}_2 \rangle \)**

** Assumes compat: makes-compatible \( \langle c_1, \text{mds}, \text{mem}_1 \rangle \langle c_2, \text{mds}, \text{mem}_2 \rangle \) \( \text{mems} \)**

** Assumes modes-eq: \( \text{map snd } \text{cms}_1 = \text{map snd } \text{cms}_2 \)**

** Assumes eval: \( \langle c_1, \text{mds}, \text{mem}_1 \rangle \approx_k \langle c_1', \text{mds}, \text{mem}_1' \rangle \)**

** Obtains \( \text{cms}_2, \text{mem}_2' \) \( \text{mems}' \) where \( \text{map snd } \text{cms}_1' = \text{map snd } \text{cms}_2' \)

\( \langle \text{cms}_2, \text{mem}_2 \rangle \approx_k \langle \text{cms}_2', \text{mem}_2' \rangle \) \( \approx_k \langle \text{cms}_1', \text{mem}_1' \rangle \) \( \langle \text{cms}_2', \text{mem}_2' \rangle \) \( \text{mems}' \)**

**Proof**

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 \( \langle c_1, \text{mds}, \text{mem}_1 \rangle \langle c_2, \text{mds}, \text{mem}_2 \rangle \) \( \text{mems} \)**

** Assumes modes-eq: \( \text{map snd } \text{cms}_1 = \text{map snd } \text{cms}_2 \)**

** Assumes x-low: \( \text{dna } \text{mem}_1 x = \text{Low} \)**

** Assumes x-readable: \( x \in C \lor (\forall i < \text{length } \text{cms}_1, x \notin \text{snd } (\text{cms}_1 ! i) \text{AsmNoReadOrWrite}) \)**

** Shows \( \text{mem}_1 x = \text{mem}_2 x \)**

**Proof**

**Lemma loc-reach-subset:**

** Assumes eval: \( \langle c, \text{mds}, \text{mem} \rangle \approx \langle c', \text{mds}', \text{mem}' \rangle \)**

** Shows \( \text{loc-reach } \langle c', \text{mds}', \text{mem}' \rangle \subseteq \text{loc-reach } \langle c, \text{mds}, \text{mem} \rangle \)**

**Proof**

**Lemma locally-sound-modes-invariant:**

** Assumes sound-modes: locally-sound-mode-use \( \langle c, \text{mds}, \text{mem} \rangle \)**
assumes eval: (c, mds, mem) → (c', mds', mem')
shows locally-sound-mode-use (c', mds', mem')
(proof)

lemma meval-sched-one:
(cms, mem) →ₖ (cms', mem') → (cms, mem) →ₖ [k] (cms', mem')
(proof)

lemma meval-sched-ConsI:
(cms, mem) →ₖ (cms', mem') → (cms', mem') → sched (cms'', mem'') → (cms, mem) →ₖ (k#sched) (cms'', mem'')
(proof)

lemma reachable-modes-subset:
assumes eval: (cms, mem) →ₖ (cms', mem')
shows reachable-mode-states (cms', mem') ⊆ reachable-mode-states (cms, mem)
(proof)

lemma globally-sound-modes-invariant:
assumes globally-sound: globally-sound-mode-use (cms, mem)
assumes eval: (cms, mem) →ₖ (cms', mem')
shows globally-sound-mode-use (cms', mem')
(proof)

lemma loc-reach-mem-diff-subset:
assumes mem-diff: ∀ x. var-asm-not-written mds x → mem₁ x = mem₂ x ∧ dma mem₁ x = dma mem₂ x
shows (c', mds', mem') ∈ loc-reach (c, mds, mem₁) → (c', mds', mem') ∈ loc-reach (c, mds, mem₂)
(proof)

lemma loc-reach-mem-diff-eq:
assumes mem-diff: ∀ x. var-asm-not-written mds x → mem' x = mem x ∧ dma mem' x = dma mem x
shows loc-reach (c, mds, mem) = loc-reach (c, mds, mem')
(proof)

lemma sound-modes-invariant:
assumes sound-modes: sound-mode-use (cms, mem)
assumes eval: (cms, mem) →ₖ (cms', mem')
shows sound-mode-use (cms', mem')
(proof)

lemma app-Cons-rewrite:
ns @ (a # ms) = (ns @ [a]) @ ms
(proof)
lemma meval-sched-app-iff:
\[(\text{cms}_1, \text{mem}_1) \rightarrow_{\text{ns}@[n]} (\text{cms}_1', \text{mem}_1') = \exists \text{cms}_1'' \text{ mem}_1''. (\text{cms}_1, \text{mem}_1) \rightarrow_{\text{ns}} (\text{cms}_1'', \text{mem}_1'') \land (\text{cms}_1'', \text{mem}_1'') \rightarrow_{\text{ms}} (\text{cms}_1', \text{mem}_1')\]
(proof)

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:
\[(\text{cms}_1, \text{mem}_1') \rightarrow_{\text{ns}@[n]} (\text{cms}_1', \text{mem}_1') \Rightarrow \exists \text{cms}_1'' \text{ mem}_1''. (\text{cms}_1, \text{mem}_1) \rightarrow_{\text{ns}} (\text{cms}_1'', \text{mem}_1'') \land (\text{cms}_1'', \text{mem}_1'') \rightarrow_{\text{ns}} (\text{cms}_1', \text{mem}_1')\]
(proof)

lemma meval-sched-snocI:
\[(\text{cms}_1, \text{mem}_1') \rightarrow_{\text{ns}} (\text{cms}_1'', \text{mem}_1'') \land (\text{cms}_1'', \text{mem}_1'') \rightarrow_{\text{ns}} (\text{cms}_1', \text{mem}_1') \Rightarrow (\text{cms}_1, \text{mem}_1) \rightarrow_{\text{ns}@[n]} (\text{cms}_1', \text{mem}_1')\]
(proof)

lemma makes-compatible-eval-sched:
assumes compat: makes-compatible (\text{cms}_1, \text{mem}_1) (\text{cms}_2, \text{mem}_2)\ mems
assumes modes-eq: map snd \text{cms}_1 = map snd \text{cms}_2
assumes sound-modes: sound-mode-use (\text{cms}_1, \text{mem}_1) sound-mode-use (\text{cms}_2, \text{mem}_2)
assumes eval: (\text{cms}_1, \text{mem}_1) \rightarrow_{\text{sched}} (\text{cms}_1', \text{mem}_1')
shows \exists \text{cms}_2' \text{ mem}_2' \text{ mems}'. sound-mode-use (\text{cms}_1', \text{mem}_1') \land sound-mode-use (\text{cms}_2', \text{mem}_2') \land map snd \text{cms}_1' = map snd \text{cms}_2' \land (\text{cms}_2, \text{mem}_2) \rightarrow_{\text{sched}} (\text{cms}_2', \text{mem}_2') \land makes-compatible (\text{cms}_1', \text{mem}_1') (\text{cms}_2', \text{mem}_2') \text{ mems}'
(proof)

lemma differing-vars-initially-empty:
i < n \Rightarrow x \notin differing-vars-lists \text{mem}_1 \text{ mem}_2 (\text{zip} (\text{replicate n mem}_1) (\text{replicate n mem}_2)) i
(proof)

lemma compatible-refl:
assumes coms-secure: list-all com-sifum-secure \text{cmds}
assumes low-eq: \text{mem}_1 =' \text{mem}_2
shows makes-compatible (\text{cmds}, \text{mem}_1) (\text{cmds}, \text{mem}_2)
(\text{replicate (length \text{cmds}) (mem}_1, \text{mem}_2))
(proof)

theorem sifum-compositionality-cont:
assumes com-secure: list-all com-sifum-secure \text{cmds}
assumes sound-modes: \( \forall \) mem. INIT mem \( \rightarrow \) sound-mode-use (cmds, mem)

shows prog-sifum-secure-cont cmds

(proof)

end

end

4 Language for Instantiating the SIFUM-Security Property

theory Language

imports Preliminaries

begin

4.1 Syntax

datatype \( \text{'var ModeUpd} = \text{Acq 'var Mode} (\text{infix} +_{m} 75) \)

| Rel 'var Mode (\text{infix} -_{m} 75)

datatype (\text{'var, 'aexp, 'bexp}) Stmt = Assign 'var 'aexp (\text{infix} ← 130)

| Skip

| ModeDecl (\text{'var, 'aexp, 'bexp}) Stmt 'var ModeUpd (\text{infix} \[ \text{-0}[-] [0, 0] \] 150)

| Seq (\text{'var, 'aexp, 'bexp}) Stmt (\text{'var, 'aexp, 'bexp}) Stmt (\text{infixr} ;; 150)

| If 'bexp (\text{'var, 'aexp, 'bexp}) Stmt (\text{'var, 'aexp, 'bexp}) Stmt

| While 'bexp (\text{'var, 'aexp, 'bexp}) Stmt

| Await 'bexp (\text{'var, 'aexp, 'bexp}) Stmt

| Stop

type-synonym (\text{'var, 'aexp, 'bexp}) EvalCxt = (\text{'var, 'aexp, 'bexp}) Stmt list

locale sifum-lang-no-dma =

fixes eval_A :: ('Var,'Val) Mem \( \Rightarrow \) 'AExp \( \Rightarrow \) 'Val

fixes eval_B :: ('Var,'Val) Mem \( \Rightarrow \) 'BExp \( \Rightarrow \) bool

fixes aexp-vars :: 'AExp \( \Rightarrow \) 'Var set

fixes bexp-vars :: 'BExp \( \Rightarrow \) 'Var set

assumes Var-finite : finite \{x :: 'Var\}. True

assumes eval-vars-det_A : \( \forall \) x \( \in \) aexp-vars e. mem1 x = mem2 x \( \Rightarrow \) eval_A mem1 e = eval_A mem2 e

assumes eval-vars-det_B : \( \forall \) x \( \in \) bexp-vars b. mem1 x = mem2 x \( \Rightarrow \) eval_B mem1 b = eval_B mem2 b

locale sifum-lang = sifum-lang-no-dma eval_A eval_B aexp-vars bexp-vars

for eval_A :: ('Var,'Val) Mem \( \Rightarrow \) 'AExp \( \Rightarrow \) 'Val

and eval_B :: ('Var,'Val) Mem \( \Rightarrow \) 'BExp \( \Rightarrow \) bool

and aexp-vars :: 'AExp \( \Rightarrow \) 'Var set

and bexp-vars :: 'BExp \( \Rightarrow \) 'Var set+

fixes dma :: 'Var \( \Rightarrow \) Sec
context sifum-lang-no-dma
begin

notation (latex output)
Seq (; - 60)

notation (Rule output)
Seq (; - 60)

notation (Rule output)
If (if - then - else - fi 50)

notation (Rule output)
While (while - do - done)

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

abbreviation conf_w-abv :: ('Var, 'AExp, 'BExp) Stmt ⇒ 'Var Mds ⇒ 'Var Mds
⇒ ('Var, 'Val) Mem ⇒ (.,.,.) LocalConf
((.,.,.)_w |0, 120, 120| 100)
where
(c, mds, mem)w ≡ ((c, mds), mem)

4.2 Semantics

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

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

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

inductive-set eval_w-simple :: (('Var, 'AExp, 'BExp) Stmt × ('Var, 'Val) Mem) rel
and eval_w-simple-abv :: (('Var, 'AExp, 'BExp) Stmt × ('Var, 'Val) Mem) ⇒
(('Var, 'AExp, 'BExp) Stmt × ('Var, 'Val) Mem) ⇒ bool
(infixr ⇝, 60)
where

\[ c \sim_c c' \equiv (c, c') \in \text{eval}_w\text{-simple} | \]
assign: \((x \leftarrow e, \text{mem})\), \((\text{Stop}, \text{mem} (x := \text{eval}_A \text{mem} e))\) \in \text{eval}_w\text{-simple} |
skip: (Stop, mem) \in eval_w\text{-simple} |
seq-stop: ((Seq Stop c, mem), (c, mem)) \in eval_w\text{-simple} |
if-true: \([\text{eval}_B \text{mem} b] \Rightarrow ((\text{If} \ b \ t \ e, \text{mem}), (t, \text{mem})) \in \text{eval}_w\text{-simple} |
if-false: \[\neg \text{eval}_B \text{mem} b] \Rightarrow ((\text{If} \ b \ t \ e, \text{mem}), (e, \text{mem})) \in \text{eval}_w\text{-simple} |
while: ((\text{While} \ b \ c, \text{mem}), (\text{If} \ b \ (c ;; \text{While} \ b \ c) \text{Stop}, \text{mem})) \in \text{eval}_w\text{-simple}

lemma cond:
\((\text{If} \ b \ t \ e, \text{mem}), (\text{if} \ \text{eval}_B \text{mem} b \ \text{then} \ t \ \text{else} \ e, \text{mem})) \in \text{eval}_w\text{-simple} \rangle

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

where
\[ \text{cxt-to-stmt} [] c = c | \]
\[ \text{cxt-to-stmt} (c \# cs) c' = \text{Seq} c' (\text{cxt-to-stmt} cs c) \]

lemma rtrancl-mono-proof[mono]
(\(\forall a. b. x a b \rightarrow y a b\)) \Rightarrow rtranclp x a b \rightarrow rtranclp y a b \rangle

lemma trancl-mono-proof[mono]
(\(\forall a. b. x a b \rightarrow y a b\)) \Rightarrow tranclp x a b \rightarrow tranclp y a b \rangle

inductive no-await :: ('Var, 'AExp, 'BExp) Stmt \Rightarrow bool where
\[ \text{no-await} (x \leftarrow c) | \]
\[ \text{no-await} c1 \Rightarrow \text{no-await} c2 \Rightarrow \text{no-await} (c1 ;; c2) | \]
\[ \text{no-await} c1 \Rightarrow \text{no-await} c2 \Rightarrow \text{no-await} (\text{If} \ b \ c1 \ c2) | \]
\[ \text{no-await} c \Rightarrow \text{no-await} (\text{While} \ b \ c) | \]
\[ \text{no-await} \text{Skip} | \]
\[ \text{no-await} \text{Stop} | \]
\[ \text{no-await} c \Rightarrow \text{no-await} (c@[m]) \]

inductive is-final :: ('Var, 'AExp, 'BExp) Stmt \Rightarrow bool where
\[ \text{is-final} \text{Stop} | \]
\[ \text{is-final} c \Rightarrow \text{is-final} (c@[m]) \]

inductive-set eval_w :: ('Var, 'AExp, 'BExp) Stmt, ('Var, 'Val) LocalConf rel
and eval_w-abe :: ('(Var, 'AExp, 'BExp) Stmt, ('Var, 'Val) LocalConf) \Rightarrow
\((('Var, 'AExp, 'BExp) Stmt, ('Var, 'Val) LocalConf) \Rightarrow \text{bool} \)
\(\text{infixr} \rightarrow_w 60\)

where
The following lemmas simplify working with evaluation contexts in the evalu-ation of expressions.

<table>
<thead>
<tr>
<th>Lemma</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lemma cxt-inv</strong>:</td>
<td>$c \rightsquigarrow_w c' \equiv (c, c') \in eval_w$</td>
</tr>
<tr>
<td><strong>lemma cxt-inv-assign</strong>:</td>
<td>$\text{seq} : \langle \langle c_1, mds, mem \rangle_w, \langle c_1', mds', mem' \rangle_w \rangle \Rightarrow \langle \langle c_1 :: c_2, mds, mem \rangle_w, \langle c_1', mds', mem' \rangle_w \rangle \in eval_w$</td>
</tr>
<tr>
<td><strong>lemma cxt-inv-stop</strong>:</td>
<td>$\text{decl} : \langle \langle c, \text{update-modes} ma mds, mem \rangle_w, \langle c', mds', mem' \rangle_w \rangle \Rightarrow \langle \langle \text{cxt-to-stmt} E (\text{ModeDecl} c \text{ mu}), mds, mem \rangle_w, \langle \text{cxt-to-stmt} E c', mds', mem' \rangle_w \rangle \in eval_w$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>abbreviation eval_w-plus ::</strong></td>
<td>$((\text{Var}, \text{AExp}, \text{BExp}) \text{ Stmt}, \text{Var}, \text{Val}) \text{ LocalConf} \Rightarrow \langle ((\text{Var}, \text{AExp}, \text{BExp}) \text{ Stmt}, \text{Var}, \text{Val}) \text{ LocalConf} \Rightarrow \text{ bool } (- \rightsquigarrow_w^+)$</td>
</tr>
</tbody>
</table>

- **where**

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

### 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**: $\langle (c, mds, mem), (c', mds', mem') \rangle \in eval_w$  

**inductive-cases stop-no-elim' [elim]**: $\langle (\text{Stop}, \text{mem}), (c', \text{mem'}) \rangle \in eval_w$  

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

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

**lemma cxt-inv**: $\lbrack \langle \text{cxt-to-stmt} E c = c' ; \land p q. c' \neq \text{Seq} p q \rangle \Rightarrow E = [] \land c' = c \rbrack$  

**lemma cxt-inv-assign**: $\lbrack \langle \text{cxt-to-stmt} E c = x \leftarrow e \rangle \Rightarrow e = x \leftarrow e \land E = [] \rbrack$  

**lemma cxt-inv-skip**: $\lbrack \langle \text{cxt-to-stmt} E c = \text{Skip} \rangle \Rightarrow e = \text{Skip} \land E = [] \rbrack$  

**lemma cxt-inv-stop**: $\langle \text{cxt-to-stmt} E c = \text{Stop} \rangle \Rightarrow e = \text{Stop} \land E = []$  

**lemma cxt-inv-if**: $\langle \text{cxt-to-stmt} E c = \text{If} e p q \Rightarrow c = \text{If} e p q \land E = [] \rbrack$  

24
lemma ctx-inv-anno:
\[\text{ctx-to-stmt } E \ c = c'@[\mu] \implies c = c'@[\mu] \land E = []\]
⟨proof⟩

lemma ctx-inv-await:
\[\text{ctx-to-stmt } E \ c = \text{Await } e \ p \implies c = \text{Await } e \ p \land E = []\]
⟨proof⟩

lemma ctx-inv-while:
\[\text{ctx-to-stmt } E \ c = \text{While } e \ p \implies c = \text{While } e \ p \land E = []\]
⟨proof⟩

lemma skip-elim [elim]:
\[\langle \text{Skip} \ mds \ mem \rangle_w \rightsquigarrow_w \langle c' \ mds' \ mem' \rangle_w \implies c' = \text{Stop} \land mds = mds' \land mem = mem'\]
⟨proof⟩

lemma assign-elim [elim]:
\[\langle x \leftarrow e \ mds \ mem \rangle_w \rightsquigarrow_w \langle c' \ mds' \ mem' \rangle_w \implies c' = \text{Stop} \land mds = mds' \land mem' = \text{mem}(x := \text{eval}_A \ mem \ e)\]
⟨proof⟩

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

lemma if-elim [elim]:
\[\forall P. \ \langle (\text{If } b \ p \ q \ mem) \rangle_w \rightsquigarrow_w \langle c' \ mds' \ mem' \rangle_w \implies c' = \text{If } e \ (c ;; \text{While } e \ c) \text{Stop} \land mds' = mds \land mem' = mem\]
⟨proof⟩

inductive-cases await-elim' [elim!]: (Await b p \ mds \ mem) \rightsquigarrow_w \ (c', mds', \ mem')

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

lemma while-elim [elim]:
\[\langle (\text{While } e \ c \ mds \ mem) \rangle_w \rightsquigarrow_w \langle c' \ mds' \ mem' \rangle_w \implies c' = \text{If } e \ (c ;; \text{While } e \ c) \text{Stop} \land mds' = mds \land mem' = mem\]
⟨proof⟩

inductive-cases upd-elim' [elim]: (c@[upd] \ mds \ mem) \rightsquigarrow_w \ (c', \ mds', \ mem')

lemma upd-elim [elim]:
\[\langle c@[upd] \ mds \ mem \rangle_w \rightsquigarrow_w \langle c' \ mds' \ mem' \rangle_w \implies \langle c \ \text{update-modes } upd \ mds, mem \rangle_w \rightsquigarrow_w \langle c' \ mds' \ mem' \rangle_w\]
⟨proof⟩

lemma ctx-seq-elim [elim]:
\[c_1 ;; c_2 = \text{ctx-to-stmt } E \ c \implies (E = [] \land c = c_1 ;; c_2) \lor (\exists \ c' \ cs. \ E = c' # cs)\]
\[ \land c = c_1 \land c_2 = \text{ctxt-to-stmt} \ cs \ c' \]

\[ \langle \text{proof} \rangle \]

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

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

\[ \langle \text{proof} \rangle \]

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

\[ \langle \text{Stop} :: c, \text{mds}, \text{mem} \rangle_w \rightsquigarrow_w \langle c', \text{mds}', \text{mem}' \rangle_w \implies c' = c \land \text{mds}' = \text{mds} \land \text{mem}' = \text{mem} \]

\[ \langle \text{proof} \rangle \]

**lemma cxt-stmt-seq:**

\[ c :: \text{ctxt-to-stmt} \ E \ c' = \text{ctxt-to-stmt}(c' \# E) \ c \]

\[ \langle \text{proof} \rangle \]

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

\[ \[ \langle c_1 :: c_2, \text{mds}, \text{mem} \rangle_w \rightsquigarrow_w \langle c', \text{mds}', \text{mem}' \rangle_w \implies c_1 \neq \text{Stop} \] \implies \\
\[ \exists c_1', (c_1', \text{mds}, \text{mem})_w \rightsquigarrow_w \langle c_1', \text{mds}', \text{mem}' \rangle_w \land c' = c_1' :: c_2 \]

\[ \langle \text{proof} \rangle \]

**lemma stop-cxt:**

\[ \text{Stop} = \text{ctxt-to-stmt} \ E \ c' \implies c = \text{Stop} \]

\[ \langle \text{proof} \rangle \]

**lemmas decl-eval \( w \):**

- **assumes mem-unchanged:** \( \text{mem}' = \text{mem} \)
- **assumes upd:** \( \text{mds}' = \text{update-modes} \, \text{upd} \, \text{mds} \)
- **shows** \( (\langle \text{Skip}@\text{[upd]} \rangle, \text{mds}, \text{mem})_w, (\langle \text{Stop}, \text{mds}', \text{mem}' \rangle)_w \) \( \in \text{eval}_w \)

\[ \langle \text{proof} \rangle \]

**lemma assign-eval \( w \):**

\[ [\text{mds} = \text{mds'}; \text{mem}' = \text{mem}(x := \text{eval}_A \text{mem} \ e)] \implies \\
\[ (x \leftarrow e, \text{mds}, \text{mem})_w \rightsquigarrow_w \langle \text{Stop}, \text{mds}', \text{mem}' \rangle_w \]

\[ \langle \text{proof} \rangle \]

26
lemma seq-decl-elim:

\[ (\text{Skip@} \text{upd}) ; c, \text{mds}, \text{mem}_w \Rightarrow \text{c'}, \text{mds}', \text{mem}'_w \Rightarrow \]
\[ c' = \text{Stop} ; c \land \text{mem}' = \text{mem} \land \text{mds}' = \text{update-modes} \text{upd mds} \]
(proof)

lemma seq-assign-elim:

\[ (x \leftarrow e) ; c, \text{mds}, \text{mem}_w \Rightarrow \text{c'}, \text{mds}', \text{mem}'_w \Rightarrow \]
\[ c' = \text{Stop} ; c \land \text{mds}' = \text{mds} \land \text{mem}' = \text{mem}(x := \text{eval}_A \text{mem} e) \]
(proof)

lemma no-await-trans:

\[ [\text{no-await} c, (c, \text{mds}, \text{mem})_w \Rightarrow (c', \text{mds}', \text{mem}'_w)] \Rightarrow \text{no-await} c' \]
(proof)

lemma no-await-no-await[elim]: [no-await] \Rightarrow c \neq \text{Await b} e'
(proof)

lemma no-await-trancl-impl:

\[ \text{ctx} \Rightarrow w + \text{ctx}' \Rightarrow \text{no-await} (\text{fst (fst ctx)}) \Rightarrow \text{no-await} (\text{fst (fst ctx')}) \]
(proof)

lemma no-await-trancl:

\[ \text{ctx} \Rightarrow w + \text{ctx}', \text{no-await} (\text{fst (fst ctx)}) \Rightarrow \text{no-await} (\text{fst (fst ctx')}) \]
(proof)

lemma await-elim:

\[ [(\text{Await b} c_1, \text{mds}, \text{mem})_w \Rightarrow (c_2, \text{mds}', \text{mem}'_w)] \Rightarrow \]
\[ \text{eval}_B \text{mem} b \land \text{no-await} c_1 \land \text{is-final} c_2 \land \]
\[ (c_1, \text{mds}, \text{mem})_w \Rightarrow w + (c_2, \text{mds}', \text{mem}'_w) \]
(proof)

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 \text{AsmNoReadOrWrite} assumption is assigned the value in variable \text{input}, but where \text{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.

\[
\text{type-synonym } 'BExp \text{ Type } = 'BExp \text{ set}
\]

We require $\Gamma$ to track all stable (i.e. AsmNoWrite or AsmNoReadOrWrite), non-$C$ 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.

\[
\text{type-synonym } (\Var, 'BExp) \text{ TyEnv } = \Var \rightarrow 'BExp \text{ Type}
\]

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.

\[
\text{type-synonym } \Var \text{ Stable } = (\Var \text{ set } \times \Var \text{ set})
\]

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 $(\Var, 'Val) \text{ Mem } \Rightarrow \text{ bool or even } (\Var, 'Val) \text{ Mem set}$), 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-synonyms**

\[ \text{bexp preds} = \text{bexp set} \]

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

**definition**

\[ \text{pred} :: \text{BExp preds} \Rightarrow (\text{Var}, \text{Val}) \text{ Mem} \Rightarrow \text{bool} \]

**where**

\[ \text{pred } P \equiv \lambda \text{mem}. (\forall p \in P. \text{eval} \text{B mem} p) \]

end

**locale** sifum-types =

sifum-lang-no-dma ev\_A ev\_B aexp-vars bexp-vars + sifum-security dma C-vars C eval\_w undefined

**for** ev\_A :: (\text{Var}, \text{Val}) \text{ Mem} \Rightarrow \text{AExp} \Rightarrow \text{Val}

**and** ev\_B :: (\text{Var}, \text{Val}) \text{ Mem} \Rightarrow \text{BExp} \Rightarrow \text{bool}

**and** aexp-vars :: \text{AExp} \Rightarrow \text{Var set}

**and** bexp-vars :: \text{BExp} \Rightarrow \text{Var set}

and dma :: (\text{Var}, \text{Val}) \text{ Mem} \Rightarrow \text{Var} \Rightarrow \text{Sec}

and C-vars :: \text{Var} \Rightarrow \text{Var set}

and C :: \text{Var set} +

**fixes** bexp-neg :: \text{BExp} \Rightarrow \text{BExp}

assumes bexp-neg-negates: \( \lambda \text{mem } e. (\text{ev} \text{B mem (bexp-neg e)}) = (\neg (\text{ev} \text{B mem e})) \)

**fixes** assign-post :: \text{BExp preds} \Rightarrow \text{Var} \Rightarrow \text{AExp} \Rightarrow \text{BExp preds}

assumes assign-post-valid: \( \lambda \text{mem } P \text{ mem} \Rightarrow \text{pred} (\text{assign-post } P \text{ x } e) \)

\( (\text{mem}(x := \text{ev} \text{A mem e})) \)

**fixes** dma-type :: \text{Var} \Rightarrow \text{BExp set}

assumes dma-correct:

\( \text{dma mem } x = (\text{if } (\forall e \in \text{dma-type } x. \text{ev} \text{B mem e}) \text{ then Low else High}) \)

assumes C-vars-correct:

\( C\text{-vars } x = (\bigcup (\text{bexp-vars } \text{ dma-type } x)) \)

**fixes** pred-False :: \text{BExp}

assumes pred-False-is-False: \( \neg \text{ev} \text{B mem pred-False} \)

assumes bexp-vars-pred-False: \text{bexp-vars pred-False} = \{\}

**locale** sifum-types-assign =

sifum-lang-no-dma ev\_A ev\_B aexp-vars bexp-vars + sifum-security dma C-vars C eval\_w undefined

**for** ev\_A :: (\text{Var}, \text{Val}) \text{ Mem} \Rightarrow \text{AExp} \Rightarrow \text{Val}

**and** ev\_B :: (\text{Var}, \text{Val}) \text{ Mem} \Rightarrow \text{BExp} \Rightarrow \text{bool}

**and** aexp-vars :: \text{AExp} \Rightarrow \text{Var set}

**and** bexp-vars :: \text{BExp} \Rightarrow \text{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: \( \forall e. (ev_B \text{ mem } (bexp-neg e)) = (\neg (ev_B \text{ mem } e)) \)

fixes dma-type :: 'Var ⇒ 'BExp set
assumes dma-type-correct:
\( \text{ dma mem } x = (if \( (\forall e \in \text{ dma-type } x. ev_B \text{ mem } e) \) then Low else High) \)

assumes C-vars-correct:
\( C \text{-vars } x = (\bigcup (\text{ bexp-vars } \ ' \text{ dma-type } x)) \)

fixes pred-False :: 'BExp
assumes pred-False-is-False: \( \neg ev_B \text{ mem } pred-False \)
 fixes bexp-vars-pred-False: bexp-vars pred-False = \{\}

fixes bexp-assign :: 'Var ⇒ 'AExp ⇒ 'BExp
assumes bexp-assign-eval: \( \forall e. (ev_B \text{ mem } (bexp-assign x e)) = (mem x = (ev_A \text{ mem } e)) \)
assumes bexp-assign-vars: \( \forall e. (bexp-vars (bexp-assign x e)) = aexp-vars e \cup \{x\} \)

context sifum-lang-no-dma begin

definition
stable :: 'Var Stable ⇒ 'Var ⇒ bool
where
\( \text{ stable } S x \equiv x \in (\text{ fst } S \cup \text{ snd } S) \)

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

lemma add-pred-subset:
\( P \subseteq P +_S p \)
(proof)

definition
restrict-preds-to-vars :: 'BExp preds ⇒ 'Var set ⇒ 'BExp preds (- | ' - [120, 120] 1000)
where
\( P |' V \equiv \{e. e \in P \land \text{ bexp-vars } e \subseteq V\} \)

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

the most simple assignment postcondition transformer

**definition**

assign-post :: 'BExp preds ⇒ 'Var ⇒ 'AExp ⇒ 'BExp preds

where

assign-post P x e ≡
(if x ∈ (aexp-vars e) then
(restrict-preds-to-vars P (−{x}))
else
(restrict-preds-to-vars P (−{x}) ∪ {beexp-assign x e}))

end

**sublocale** sifum-types-assign ⊆ sifum-types - - - - - - - assign-post

(proof)

**context** sifum-types

begin

**abbreviation**

mm-equiv-abv2 :: (-, -, -) LocalConf ⇒ (-, -, -) LocalConf ⇒ bool
(infix ≈ 60)

where

mm-equiv-abv2 c c’ ≡ mm-equiv-abv c c’

**abbreviation**

eval-abv2 :: (-, 'Var, 'Val) LocalConf ⇒ (-, -, -) LocalConf ⇒ bool
(infixl ⇝ 70)

where

x ⇝ y ≡ (x, y) ∈ eval_w

**abbreviation**

eval-plus-abv :: (-, 'Var, 'Val) LocalConf ⇒ (-, -, -) LocalConf ⇒ bool
(infixl ⇝+ 70)

where

x ⇝+ y ≡ (x, y) ∈ eval_w⁺

**abbreviation**

no-eval-abv :: (-, 'Var, 'Val) LocalConf ⇒ bool
(- ⇝ ⊥)

where

x ⇝ ⊥ ≡ ∀ y. (x, y) ∉ eval_w

**abbreviation**

low-indistinguishable-abv :: 'Var Mds ⇒ ('Var, 'AExp, 'BExp) Stmt ⇒ (-, -, -) Stmt ⇒ bool
where
\[ c \sim_{mds} c' \equiv \text{low-indistinguishable mds } c c' \]

**abbreviation**

vars-of-type :: 'BExp Type ⇒ 'Var set

where
\[ \text{vars-of-type } t \equiv \bigcup (\text{bexp-vars ' } t) \]

**definition**

type-wellformed :: 'BExp Type ⇒ bool

where
\[ \text{type-wellformed } t \equiv \text{vars-of-type } t \subseteq C \]

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

\[ \text{type-wellformed } (\text{dma-type } x) \]

(proof)

**definition**

to-total :: ('Var,'BExp) TyEnv ⇒ 'Var ⇒ 'BExp Type

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

**definition**

types-wellformed :: ('Var,'BExp) TyEnv ⇒ bool

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

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

\[ \text{types-wellformed } \Gamma \implies \text{type-wellformed } (\text{to-total } \Gamma x) \]

(proof)

**lemma** Un-type-wellformed:

\[ \forall t \in ts. \text{type-wellformed } t \implies \text{type-wellformed } (\bigcup ts) \]

(proof)

**inductive**

type-aexpr :: ('Var,'BExp) TyEnv ⇒ 'AExp ⇒ 'BExp Type ⇒ bool (\[ - \sim_{-} [100, 100] 80 \])

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

**lemma** type-aexpr1:

\[ t = \bigcup (\text{image } (\lambda x. \text{to-total } \Gamma x) (\text{aexp-vars } e)) \implies \Gamma \vdash e \in t \]

(proof)

**lemma** type-aexpr-type-wellformed:
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.

\[
\text{abbreviation} \\
\text{pred-stable} :: 'Var \text{ Stable} \Rightarrow 'BExp \Rightarrow \text{bool} \\
\text{where} \\
pred-stable \ S \ p \equiv \forall x \in \text{bexp-vars} \ p. \ \text{stable} \ S \ x
\]

\[
\text{abbreviation} \\
type-stable :: 'Var \text{ Stable} \Rightarrow 'BExp \text{ Type} \Rightarrow \text{bool} \\
\text{where} \\
type-stable \ S \ t \equiv (\forall p \in t. \ \text{pred-stable} \ S \ p)
\]

\[
\text{lemma} \ \text{type-stable-is-sufficient}: \\
[type-stable \ S \ t] \implies \\
\forall \ \text{mem mem’}. \ (\forall x. \ \text{stable} \ S \ x \rightarrow \text{mem} \ x = \text{mem’} \ x) \rightarrow (\text{ev}_B \ \text{mem}) \ ' t = (\text{ev}_B \ \text{mem’}) \ ' t \\
\langle \text{proof} \rangle
\]

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

where

mds-consistent mds Γ S P ≡
(S = (mds AsmNoWrite, mds AsmNoReadOrWrite)) ∧
(dom Γ = \{ x. x \notin C ∧ stable S x \}) ∧
(∀ p ∈ P. pred-stable S p)

fun

add-anno-dom :: ('Var,'BExp) TyEnv ⇒ 'Var Stable ⇒ 'Var ModeUpd ⇒ 'Var

set

where

add-anno-dom Γ S (Acq v AsmNoReadOrWrite) = (if v \notin C then dom Γ ∪ \{ v \} else dom Γ) |
add-anno-dom Γ S (Acq v AsmNoWrite) = (if v \notin C then dom Γ ∪ \{ v \} else dom Γ) |
add-anno-dom Γ S (Rel v AsmNoReadOrWrite) = (if v \notin fst S then dom Γ − \{ v \} else dom Γ) |
add-anno-dom Γ S (Rel v AsmNoWrite) = (if v \notin snd S then dom Γ − \{ v \} else dom Γ) |
add-anno-dom Γ S (Rel v -) = dom Γ

definition

add-anno :: ('Var,'BExp) TyEnv ⇒ 'Var Stable ⇒ 'Var ModeUpd ⇒ 'Var

TyEnv (- ⊕ - [120, 120, 120] 1000)

where

Γ ⊕ upd = restrict-map (λx. Some (to-total Γ x)) (add-anno-dom Γ S upd)

lemma add-anno-acq-AsmNoReadOrWrite-idemp [simp]:
v \in dom Γ \lor v \in C ⇒ Γ ⊕ S (Acq v AsmNoReadOrWrite) = Γ

(proof)

lemma add-anno-rel-AsmNoReadOrWrite-idemp [simp]:
[v \notin dom Γ; v \notin fst S] ⇒ Γ ⊕ S (Rel v AsmNoReadOrWrite) = Γ

(proof)

lemma add-anno-acq-AsmNoWrite-idemp [simp]:
assumesnotin [simp]: v \notin dom Γ
shows v \notin C ⇒ Γ ⊕ S (Acq v AsmNoWrite) = (Γ(v \mapsto dma-type v))

(proof)

lemma add-anno-rel-AsmNoWrite [simp]:
assumesisin [simp]: v \in dom Γ
shows v \notin fst S ⇒ Γ ⊕ S (Rel v AsmNoWrite) = restrict-map Γ ((dom Γ) − \{ v \})

(proof)

lemma add-anno-acq-AsmNoWrite-idemp [simp]:
v \in dom Γ \lor v \in C ⇒ Γ ⊕ S (Acq v AsmNoWrite) = Γ

34
lemma add-anno-rel-AsmNoWrite-idemp [simp]:
\[ (v \notin \text{dom } \Gamma; \ v \notin \text{snd } S) \implies \Gamma \oplus_S (Rel v \text{ AsmNoWrite}) = \Gamma \]
(proof)

lemma add-anno-acq-AsmNoWrite [simp]:
assumes \( \notin \text{simp} \): \( v \notin \text{dom } \Gamma \)
shows \( v \notin C \implies \Gamma \oplus_S (\text{Acq } v \text{ AsmNoWrite}) = (\Gamma(v \mapsto \text{dma-type } v)) \)
(proof)

lemma add-anno-rel-AsmNoWrite [simp]:
assumes \( \text{isin simp} \): \( v \in \text{dom } \Gamma \)
shows \( v \notin \text{snd } S \implies \Gamma \oplus_S (\text{Rel } v \text{ AsmNoWrite}) = \text{restrict-map } \Gamma ((\text{dom } \Gamma) - \{v\}) \)
(proof)

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

definition pred-entailment :: 'BExp preds ⇒ 'BExp preds ⇒ bool (infix ⊢)
where
\( P \vdash P' \equiv \forall \text{mem. } P \text{ mem } \longrightarrow \text{pred } P' \text{ mem } \)

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

definition subtype :: 'BExp Type ⇒ 'BExp Type ⇒ bool (-≤:-
where
\( t \leq_S t' \equiv (P \cup t') \vdash t \)

definition type-max :: 'BExp Type ⇒ ('Var,'Val) Mem ⇒ Sec
where
\( \text{type-max } t \text{ mem } \equiv \text{if } (\forall p \in t. \text{evB } \text{mem } p) \text{ then Low else High } \)

lemma type-stable-is-sufficient':
\[ \text{type-stable } S t \implies \forall \text{mem mem'. } (\forall x. \text{stable } S x \longrightarrow \text{mem } x = \text{mem'} x) \longrightarrow \text{type-max } t \text{ mem } = \text{type-max } t \text{ mem'} \]

35
lemma subtype-sound:
\[ t \leq_P t' \implies \forall \text{mem. pred P mem} \implies \text{type-max } t \text{ mem} \leq \text{type-max } t' \text{ mem} \]

lemma subtype-complete:
assumes \( a: \forall \text{mem. pred P mem} \implies \text{type-max } t \text{ mem} \leq \text{type-max } t' \text{ mem} \)
shows \( t \leq_P t' \)

lemma subtype-correct:
\( (t \leq_P t') = (\forall \text{mem. pred P mem} \implies \text{type-max } t \text{ mem} \leq \text{type-max } t' \text{ mem}) \)

definition type-equiv :: 'BExp Type \Rightarrow 'BExp preds \Rightarrow 'BExp Type \Rightarrow bool (- =: - [120, 120, 120] 1000)
where
\( t =:=_P t' \equiv (\forall \text{mem. pred P mem} \implies \text{type-max } t \text{ mem} \leq \text{type-max } t' \text{ mem}) \)

lemma subtype-refl [simp]:
\( t \leq_P t \)

lemma type-equiv-refl [simp]:
\( t =:=_P t \)

definition anno-type-stable :: ('Var,'BExp) TyEnv \Rightarrow 'Var Stable \Rightarrow 'Var ModeUpd \Rightarrow bool
where
anno-type-stable \( \Gamma S \) upd \( \equiv (\text{case upd of } (\text{Rel v m}) \Rightarrow (v \in C \land v \notin \text{add-anno-dom } \Gamma S \text{ upd}) \implies \) 
\( (\forall x \in \text{dom } \Gamma. v \notin \text{vars-of-type } (\text{the } (\Gamma x))) \) 
\mid (\text{Acq v m}) \Rightarrow (v \notin C \land v \in \text{add-anno-dom } \Gamma S \text{ upd} - \text{dom } \Gamma) \implies \) 
\( (\forall x \in C\text{-vars } v. \text{stable } S x) \) 

definition anno-type-sec :: ('Var,'BExp) TyEnv \Rightarrow 'Var Stable \Rightarrow 'BExp preds \Rightarrow 'Var ModeUpd \Rightarrow bool
where
anno-type-sec \( \Gamma S P \) upd \( \equiv (\text{case upd of } (\text{Rel v AsmNoReadOrWrite}) \Rightarrow (v \in \text{add-anno-dom } \Gamma S \text{ upd} \implies (\text{the } (\Gamma v)) \leq_P \) 
\( (\text{dma-type } v)) \) 
\mid - \Rightarrow True) \)
definition
\text{types-stable} :: (\text{'Var}, \text{'BExp}) \text{TyEnv} \Rightarrow \text{'Var Stable} \Rightarrow \text{bool}
where
\text{types-stable} \, \Gamma \, S \equiv \forall x \in \text{dom} \, \Gamma . \text{type-stable} \, S \, (\text{the} \, (\Gamma \, x))

definition
\text{tyenv-wellformed} :: \text{'Var Mds} \Rightarrow (\text{'Var}, \text{'BExp}) \text{TyEnv} \Rightarrow \text{'Var Stable} \Rightarrow \text{'BExp preds} \Rightarrow \text{bool}
where
\text{tyenv-wellformed mds} \, \Gamma \, S \, P \equiv \text{mds-consistent mds} \, \Gamma \, S \, P \land \\
\text{types-wellformed} \, \Gamma \land \text{types-stable} \, \Gamma \, S

lemma \text{subset-entailment}:
\quad \quad \quad \text{P} \subseteq \text{P} \implies \text{P} \vdash \text{P}'
\langle \text{proof} \rangle

lemma \text{pred-entailment-refl} [\text{simp}]:
\quad \quad \quad \text{P} \vdash \text{P}
\langle \text{proof} \rangle

lemma \text{pred-entailment-mono}:
\quad \quad \quad \text{P} \vdash \text{P}' \implies \text{P} \subseteq \text{P}'' \implies \text{P}'' \vdash \text{P}'
\langle \text{proof} \rangle

lemma \text{type-equiv-subset}:
\quad \quad \quad \text{type-equiv} \, t \, \text{P} \, t \, \text{P}' \implies \text{P} \subseteq \text{P}' \implies \text{type-equiv} \, t \, \text{P} \, t' \text{P}'
\langle \text{proof} \rangle

definition
\text{context-equiv} :: (\text{'Var}, \text{'BExp}) \text{TyEnv} \Rightarrow \text{'BExp preds} \Rightarrow (\text{'Var}, \text{'BExp}) \text{TyEnv} \Rightarrow \text{bool} (- \equiv - \, [120, 120, 120] \, 1000)
where
\quad \quad \quad \Gamma = \, \vdash \, \Gamma' \equiv \text{dom} \, \Gamma = \text{dom} \, \Gamma' \land \\
\quad \quad \quad \forall x \in \text{dom} \, \Gamma'. \text{type-equiv} \, (\text{the} \, (\Gamma \, x)) \, \text{P} \, (\text{the} \, (\Gamma' \, x))

lemma \text{context-equiv-refl}[\text{simp}]:
\quad \quad \quad \text{context-equiv} \, \Gamma \, \text{P} \, \Gamma
\langle \text{proof} \rangle

lemma \text{context-equiv-subset}:
\quad \quad \quad \text{context-equiv} \, \Gamma \, \text{P} \, \Gamma' \implies \text{P} \subseteq \text{P}' \implies \text{context-equiv} \, \Gamma \, \text{P} \, \Gamma'
\langle \text{proof} \rangle

lemma \text{pred-entailment-trans}:
\quad \quad \quad \text{P} \, \vdash \text{P}' \implies \text{P}' \vdash \text{P}'' \implies \text{P} \vdash \text{P}''
\langle \text{proof} \rangle

lemma \text{pred-un} [\text{simp}]:

37
\[
\text{pred} \ (P \cup P') \ \text{mem} = (\text{pred} \ P \ \text{mem} \land \text{pred} \ P' \ \text{mem})
\]

proof

\textbf{lemma pred-entailment-un:}
\[P \vdash P' \Rightarrow P \vdash P'' \Rightarrow P \vdash (P' \cup P'')\]
proof

\textbf{lemma pred-entailment-mono-un:}
\[P \vdash P' \Rightarrow (P \cup P'') \vdash (P' \cup P'')\]
proof

\textbf{lemma subtype-trans:}
\[t \leq_p t' \Rightarrow t' \leq_p t'' \Rightarrow P \vdash P' \Rightarrow t \leq_p t''\]
\[t \leq_p t' \Rightarrow t' \leq_p t'' \Rightarrow P \vdash P' \Rightarrow t \leq_p t''\]
proof

\textbf{lemma type-equiv-trans:}
\[\text{type-equiv} \ t \ P \ t' \Rightarrow \text{type-equiv} \ t' \ P' \ t'' \Rightarrow P \vdash P' \Rightarrow \text{type-equiv} \ t \ P \ t''\]
proof

\textbf{lemma context-equiv-trans:}
\[\text{context-equiv} \ \Gamma \ P \ \Gamma' \Rightarrow \text{context-equiv} \ \Gamma' \ P' \ \Gamma'' \Rightarrow P \vdash P' \Rightarrow \text{context-equiv} \ \Gamma \ P \ \Gamma''\]
proof

\textbf{lemma un-pred-entailment-mono:}
\[(P \cup P') \vdash P'' \Rightarrow P'' \vdash (P'' \cup P') \vdash P''\]
proof

\textbf{lemma subtype-entailment:}
\[t \leq_p t' \Rightarrow P' \vdash P \Rightarrow t \leq_p t'\]
proof

\textbf{lemma type-equiv-entailment:}
\[\text{type-equiv} \ t \ P \ t' \Rightarrow P' \vdash P \Rightarrow \text{type-equiv} \ t \ P' \ t'\]
proof

\textbf{lemma context-equiv-entailment:}
\[\text{context-equiv} \ \Gamma \ P \ \Gamma' \Rightarrow P' \vdash P \Rightarrow \text{context-equiv} \ \Gamma \ P' \ \Gamma'\]
proof

\textbf{inductive}

\[\text{has-type} :: (\text{'Var}, \text{'BExp}) \ \text{TyEnv} \Rightarrow \text{'Var Stable} \Rightarrow \text{'BExp} \ \text{preds} \Rightarrow (\text{'Var}, \text{'AExp}, \text{'BExp}) \ \text{Stmt} \Rightarrow (\text{'Var}, \text{'BExp}) \ \text{TyEnv} \Rightarrow \text{'Var Stable} \Rightarrow \text{'BExp} \ \text{preds} \Rightarrow \text{bool} \ (\vdash \vdash \vdash \vdash \{\} \vdash \vdash \vdash \{120, 120, 120, 120, 120, 120, 120\} \ 1000)\]
where
stop-type [intro]: ⊢ Γ,S,P {Stop} Γ,S,P |
skip-type [intro]: ⊢ Γ,S,P {Skip} Γ,S,P |
assign₁:
  [ x ∈ C; Γ ⊢ a e ∈ t; P ⊢ t; (∀ v ∈ dom Γ, x /∈ vars-of-type (the (Γ v))); P′ = restrict-preds-to-vars (assign-post P x e) {v. stable S v};
  ∀ v, x ∈ C-vars v ∧ v /∈ snd S ⊢ P ⊢ (to-total Γ v) ∧ (to-total Γ v) ≤;p′ (dma-type v) ] ⇒
  ⊢ Γ,S,P { x ← e } Γ,S,P′ |
assign₂:
  [ x /∈ dom Γ; x ∈ C; Γ ⊢ a e ∈ t; t ≤;p (dma-type x); P′ = restrict-preds-to-vars (assign-post P x e) {v. stable S v} ] ⇒
  ⊢ Γ,S,P { x ← e } Γ,S,P′ |
if-type [intro]:
  [ Γ ⊢ a e ∈ t; P ⊢ t; 
    ⊢ Γ,S,(P +S e) { c₁ } Γ′,S′,P; ⊢ Γ,S,(P +S (bexp-neg e)) { c₂ } Γ′′,S′′,P″; 
    context-equiv Γ′ P′ Γ′′; context-equiv Γ′′ P″ P‴; P′ ⊢ P″; P‴; P‴ ⊢ P‴‴; 
    ∀ mds. tyenv-wellformed mds Γ′′ S′ P‴ → tyenv-wellformed mds Γ‴‴ S‴ P‴‴; 
    ∀ mds. tyenv-wellformed mds Γ‴‴ S‴ P‴‴ → tyenv-wellformed mds Γ‴‴ S‴ P‴‴ ] ⇒
  ⊢ Γ,S,P { if e c₁ c₂ } Γ‴‴ S‴,P‴‴ |
while-type [intro]: [ Γ ⊢ b e ∈ t ; P ⊢ t ; ⊢ Γ,S,(P +S e) { c } Γ,S,P ] ⇒
  ⊢ Γ,S,P { while e c } Γ,S,P |
anno-type [intro]: [ Γ′ = Γ ⊕S upd ; S′ = add-anno-stable S upd ; P′ = restrict-preds-to-vars 
P {v. stable S′ v} ;
    ⊢ Γ′,S′,P′ { c } Γ‴‴ S‴,P‴‴ ; e /∉ Stop ; 
    (∀ x. (to-total Γ x) ≤;p′ (to-total Γ′ x));
    anno-type-stable Γ S upd; anno-type-sec Γ S P upd ] ⇒
  ⊢ Γ,S,P { c(t[upd]) } Γ‴‴ S‴,P‴‴ |
seq-type [intro]: [ ⊢ Γ,S,P { c₁ } Γ′,S′,P′ ; ⊢ Γ′,S′,P′ { c₂ } Γ‴‴ S‴,P‴‴ ] ⇒
  ⊢ Γ,S,P { c₁ ; c₂ } Γ‴‴ S‴,P‴‴ |
sub: [ ⊢ Γ₁,S,P₁ { c } Γ₁′,S′,P₁{c} ; context-equiv Γ₂ P₂ Γ₁ ; (∀ mds. tyenv-wellformed 
mds Γ₂ S P₂ → tyenv-wellformed mds Γ₁ S P₁) ;
  (∀ mds. tyenv-wellformed mds Γ₁′ S′ P′₁ → tyenv-wellformed mds Γ₂′ S′ P′₂) ;
  context-equiv Γ₁′ P₁′ Γ₂′ ; P₂ ⊢ P₁ ; P₁′ ⊢ P₂′ ] ⇒
  ⊢ Γ₂,S,P₂ { c } Γ₂′,S′,P₂′ |
await-type [intro]:
  [ Γ ⊢ b e ∈ t ; P ⊢ t ; 
    ⊢ Γ,S,(P +S e) { c } Γ′,S′,P′ ] ⇒
  ⊢ Γ,S,P { Await e c } Γ′,S′,P′ |

**Lemma anno-type-helpers [simp]:**

\[ \text{(to-total } \Gamma x) \leq: P \text{ (to-total (add-anno } \Gamma S \text{ (buffer } +=m \text{ AsmNoWrite}) x) } \]
\[ \text{(to-total } \Gamma x) \leq: P \text{ (to-total (add-anno } \Gamma S \text{ (buffer } +=m \text{ AsmNoReadOrWrite}) x) } \]

\[ \langle \text{proof} \rangle \]
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.

\[
\text{inductive-cases has-type-elim: } \vdash \Gamma, S, P \{ \ c \} \Gamma', S', P' \]

\[
\text{inductive-cases has-type-stop-elim: } \vdash \Gamma, S, P \{ \text{Stop} \} \Gamma', S', P' \]

**Definition** tyenv-eq :: (\(\text{Var}, \text{BExp}\)) TyEnv \(\Rightarrow\) (\(\text{Var}, \text{Val}\)) Mem \(\Rightarrow\) (\(\text{Var}, \text{Val}\)) Mem

\[
\text{(infix } =_{\text{tyenv-eq}} \text{)}
\]

**Lemma** tyenv-eq-sym :: type-max \(\text{dma-type}\) mem \(\Rightarrow\) \(\text{tyenv-wellformed}\) mds \(\Gamma\)

\[
\text{mem} \in \text{dom} \Gamma \\
\text{where mem}_1 =_{\Gamma} \text{mem}_2 \iff \forall x. (\text{type-max (to-total } \Gamma x) \text{ mem}_1 = \text{Low} \rightarrow \text{mem}_1 x = \text{mem}_2 x)
\]

**Proof**

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

**Lemma** tyenv-eq-sym':

\[
\text{tyenv-wellformed mds } \Gamma S P \Rightarrow \text{mem}_1 =_{\Gamma} \text{mem}_2 \Rightarrow \text{mem}_2 =_{\Gamma} \text{mem}_1
\]

**Proof**

\[
\text{inductive-set } \mathcal{R}_1 :: (\text{\textquoteleft Var,\textquoteleft BExp}\text{) TyEnv} \Rightarrow \text{\textquoteleft Var Stable} \Rightarrow \text{\textquoteleft BExp preds} \Rightarrow (\text{\textquoteleft Var, \textquoteleft AExp, \textquoteleft BExp}\text{) Stmt, } \text{\textquoteleft Var, \textquoteleft Val}) \text{LocalConf rel}
\]

**and** \(\mathcal{R}_1\)-\text{aux} ::

\[
(\text{\textquoteleft Var, \textquoteleft AExp, \textquoteleft BExp}\text{) Stmt, } \text{\textquoteleft Var, \textquoteleft Val}) \text{LocalConf} \Rightarrow
\]

\[
(\text{\textquoteleft Var, \textquoteleft BExp}\text{) TyEnv} \Rightarrow \text{\textquoteleft Var Stable} \Rightarrow \text{\textquoteleft BExp preds} \Rightarrow
\]

\[
(\text{\textquoteleft Var, \textquoteleft AExp, \textquoteleft BExp}\text{) Stmt, } \text{\textquoteleft Var, \textquoteleft Val}) \text{LocalConf} \Rightarrow
\]

\[
\text{bool} \vdash (\text{\textquoteleft R}_1, \ldots, [120, 120, 120, 120, 120, 120] 1000)
\]

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

**and** \(S' :: \text{\textquoteleft Var Stable}

**and** \(P' :: \text{\textquoteleft BExp preds}

**where**

\[
x \text{\textquoteleft R}_1 \Gamma S P \quad y \equiv (x, y) \in \mathcal{R}_1 \Gamma S P \mid
\]

\[
\text{intro [intro]} : [ \vdash \Gamma, S, P \{ \ c \} \Gamma', S', P' ; \text{tyenv-wellformed mds } \Gamma S P ; \text{mem}_1 =_{\Gamma} \text{mem}_2;
\]

\[
\quad \Gamma P \text{mem}_1 ; \quad \text{pred } P \text{mem}_2; \forall x \in \text{dom } \Gamma. \quad x \notin \text{mds \ AsmNoReadOrWrite}
\]

\[
\rightarrow \text{type-max (the (\Gamma x)) mem}_1 \leq \text{dma mem}_1 x \quad \Rightarrow
\]

\[
\quad \{ c, \text{mds, mem}_1 \} \text{\textquoteleft R}_1 \Gamma', S', P' \{ c, \text{mds, mem}_2 \}
\]

**Inductive** \(\mathcal{R}_3\)-\text{aux} :: (\(\text{\textquoteleft Var, \textquoteleft AExp, \textquoteleft BExp}\text{) Stmt, } \text{\textquoteleft Var, \textquoteleft Val}) \text{LocalConf} \Rightarrow
\]

\[
(\text{\textquoteleft Var, \textquoteleft BExp}\text{) TyEnv} \Rightarrow \text{\textquoteleft Var Stable} \Rightarrow \text{\textquoteleft BExp preds} \Rightarrow (\text{\textquoteleft Var, \textquoteleft AExp, \textquoteleft BExp}\text{) Stmt, } \text{\textquoteleft Var, \textquoteleft Val}) \text{LocalConf} \Rightarrow
\]

41
\[\text{bool} (\cdot \mathcal{R}^3, \ldots, [120, 120, 120, 120, 120] 1000)\]

and \(\mathcal{R}_3 :: (\forall \text{'Var}, \text{'BExp}) \mathcal{T}_\text{Env} \Rightarrow \text{'Var Stable} \Rightarrow \text{'BExp preds} \Rightarrow ((\forall \text{'Var}, \text{'AExp}, \text{'BExp}) \text{Stmt}, \text{'Var}, \text{'Val}) \text{LocalConf rel}\)

where
\[
\mathcal{R}_3 \Gamma \Gamma' S' P' \equiv\begin{cases} \{(lc_1, lc_2). \mathcal{R}_3\text{-aux } lc_1 \Gamma \Gamma' S' P' lc_2\} \mid \\
\text{intro}_1 \text{[intro]} : \left[\{(c_1, \text{mds}, \text{mem}_1) \mathcal{R}_1^{\Gamma, S, P} (c_2, \text{mds}, \text{mem}_2)\} \parallel \Gamma, S, P \{ c \} \right] \\
\text{Γ′,S′,P′} \rightarrow (\text{Seq } c_1 c, \text{mds, mem}_1) \mathcal{R}_3^{\Gamma', S', P', P} (\text{Seq } c_2 c, \text{mds, mem}_2) \right]\end{cases}
\]

\[
\text{Γ′,S′,P′} \rightarrow (\text{Seq } c_1 c, \text{mds, mem}_1) \mathcal{R}_3^{\Gamma', S', P', P} (\text{Seq } c_2 c, \text{mds, mem}_2)
\]

definition

weak-bisim :: ((\forall \text{'Var}, \text{'AExp}, \text{'BExp}) \text{Stmt}, \text{'Var}, \text{'Val}) \text{LocalConf rel} \Rightarrow ((\forall \text{'Var}, \text{'AExp}, \text{'BExp}) \text{Stmt}, \text{'Var}, \text{'Val}) \text{LocalConf rel} \Rightarrow \text{bool}

where

weak-bisim \(\mathcal{T}_1 T \equiv \forall c_1 c_2 \text{mds mem}_1 \text{mem}_2 c_1 \text{'} \text{mds' mem}_1'\). \(\{(c_1, \text{mds}, \text{mem}_1), (c_2, \text{mds}, \text{mem}_2)\} \in \mathcal{T}_1 \land (c_1, \text{mds}, \text{mem}_1) \Rightarrow (c_1', \text{mds'}, \text{mem}_1')\) \(\exists c_2' \text{mem}_2'. (c_2, \text{mds, mem}_2) \Rightarrow (c_2', \text{mds', mem}_2') \land \{(c_1', \text{mds'}, \text{mem}_1'), (c_2', \text{mds', mem}_2')\} \in T\)

inductive-set \(\mathcal{R} :: (\forall \text{'Var}, \text{'BExp}) \mathcal{T}_\text{Env} \Rightarrow \text{'Var Stable} \Rightarrow \text{'BExp preds} \Rightarrow ((\forall \text{'Var}, \text{'AExp}, \text{'BExp}) \text{Stmt}, \text{'Var}, \text{'Val}) \text{LocalConf rel}\)

and \(\mathcal{R}_\text{abv} :: ((\forall \text{'Var}, \text{'AExp}, \text{'BExp}) \text{Stmt}, \text{'Var}, \text{'Val}) \text{LocalConf rel} \Rightarrow (\forall \text{'Var}, \text{'BExp}) \mathcal{T}_\text{Env} \Rightarrow \text{'Var Stable} \Rightarrow \text{'BExp preds} \Rightarrow ((\forall \text{'Var}, \text{'AExp}, \text{'BExp}) \text{Stmt}, \text{'Var}, \text{'Val}) \text{LocalConf rel} \Rightarrow \text{bool} (\cdot \mathcal{R}^n, \ldots, [120, 120, 120, 120, 120] 1000)\)

for \(\Gamma :: (\forall \text{'Var}, \text{'BExp}) \mathcal{T}_\text{Env}\)

and \(\mathcal{S} :: \text{'Var Stable}\)

and \(\mathcal{P} :: \text{'BExp preds}\)

where

\[x \mathcal{R}^n_{\Gamma, S, P} y \equiv (x, y) \in \mathcal{R} \Gamma S P \mid \]

\[
\text{intro}_1 : \text{le } \mathcal{R}_1^{\Gamma, S, P} \text{le'} \Rightarrow (\text{le, le'}) \in \mathcal{R} \Gamma S P \mid \]

\[
\text{intro}_2 : \text{le } \mathcal{R}_3^{\Gamma, S, P} \text{le'} \Rightarrow (\text{le, le'}) \in \mathcal{R} \Gamma S P
\]

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

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

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

inductive-cases \(\mathcal{R}_3\text{-elim' [elim]} : \langle c_1, \text{mds, mem}_1 \rangle \mathcal{R}_3^{\Gamma, S, P} \langle c_2, \text{mds, mem}_2 \rangle\)

lemma \(\mathcal{R}_1\text{-mem-eq} : \langle c_1, \text{mds, mem}_1 \rangle \mathcal{R}_1^{\Gamma', S', P'} \langle c_2, \text{mds, mem}_2 \rangle \Rightarrow \text{mem}_1\)
=_{mds \_1} mem_2

(\text{proof})

\text{lemma} \ \mathcal{R}_1\text{-dma-eq:}
\langle c_1, mds, mem_1 \rangle \mathcal{R}_1^{1'}\mathcal{S}'P' \langle c_2, mds, mem_2 \rangle \implies \text{dma mem}_1 = \text{dma mem}_2

(\text{proof})

\text{lemma} \ \text{bisim-simple-\mathcal{R}_1:}
\langle c, mds, mem \rangle \mathcal{R}_1^{1'}\mathcal{S}'P' \langle c', mds', mem' \rangle \implies c = c'

(\text{proof})

\text{lemma} \ \text{bisim-simple-\mathcal{R}_3:}
le \mathcal{R}_3^{1'}\mathcal{S}'P' le' \implies (\text{fst (fst le)}) = (\text{fst (fst le')})

(\text{proof})

\text{lemma} \ \text{bisim-simple-\mathcal{R}_u:}
le \mathcal{R}_u^{1'}\mathcal{S}'P' le' \implies (\text{fst (fst le)}) = (\text{fst (fst le')})

(\text{proof})

\text{lemma} \ \mathcal{C}\text{-eq-type-max-eq:}
\text{assumes} \ \text{af: type-wellformed } t
\text{assumes} \ \mathcal{C}\text{-eq: } \forall x \in \mathcal{C}. \text{mem}_1 x = \text{mem}_2 x
\text{shows} \ \text{type-max } t \text{mem}_1 = \text{type-max } t \text{mem}_2

(\text{proof})

\text{lemma} \ \text{vars-of-type-eq-type-max-eq:}
\text{assumes} \ \text{mem-eq: } \forall x \in \text{vars-of-type } t. \text{mem}_1 x = \text{mem}_2 x
\text{shows} \ \text{type-max } t \text{mem}_1 = \text{type-max } t \text{mem}_2

(\text{proof})

\text{lemma} \ \mathcal{R}_1\text{-sym: } \text{sym (R}_1 \Gamma \mathcal{S}' P')

(\text{proof})

\text{lemma} \ \mathcal{R}_3\text{-sym: } \text{sym (R}_3 \Gamma S P)

(\text{proof})

\text{lemma} \ \mathcal{R}\text{-mds [simp]: } \langle c_1, mds, mem_1 \rangle \mathcal{R}_u^{1'}\mathcal{S}'P' \langle c_2, mds', mem_2 \rangle \implies mds = mds'

(\text{proof})

\text{lemma} \ \mathcal{R}\text{-sym: } \text{sym (R}_ \Gamma S P)

(\text{proof})
\textbf{lemma} $R_1$-closed-glob-consistent: closed-glob-consistent $(R_1 \; \Gamma' \; S' \; P')$

(\textit{proof})

\textbf{lemma} $R_3$-closed-glob-consistent:

closed-glob-consistent $(R_3 \; \Gamma' \; S' \; P')$

(\textit{proof})

\textbf{lemma} $R$-closed-glob-consistent: closed-glob-consistent $(R \; \Gamma' \; S' \; P')$

(\textit{proof})

\textbf{lemma} mode-update-add-anno:

mds-consistent mds $\Gamma$ $S$ $P$ $\implies$

mds-consistent (update-modes upd mds)

$(\Gamma \oplus_S \text{upd})$

(add-anno-stable $S$ upd)

($P \; |' \; \{ v. \; \text{stable} \; (add-anno-stable \; S \; \text{upd}) \; v \}$)

(\textit{proof})

\textbf{lemma} add-anno-acq-GuarNoReadOrWrite [simp]:

$\Gamma \oplus_S (v \; +=_m \; \text{GuarNoReadOrWrite}) = \Gamma$

(\textit{proof})

\textbf{lemma} add-anno-rel-GuarNoReadOrWrite [simp]:

$\Gamma \oplus_S (v \; -=_m \; \text{GuarNoReadOrWrite}) = \Gamma$

(\textit{proof})

\textbf{lemma} add-anno-acq-GuarNoWrite [simp]:

$\Gamma \oplus_S (v \; +=_m \; \text{GuarNoWrite}) = \Gamma$

(\textit{proof})

\textbf{lemma} add-anno-rel-GuarNoWrite [simp]:

$\Gamma \oplus_S (v \; -=_m \; \text{GuarNoWrite}) = \Gamma$

(\textit{proof})

\textbf{lemma} dom-add-anno-rel:

$\forall \; x \in \text{dom} \; (\Gamma \oplus_S (v \; -=_m \; m)). \; (\Gamma \oplus_S (v \; -=_m \; m)) \; x = \Gamma \; x$

(\textit{proof})

\textbf{lemma} types-wellformed-mode-update:

\begin{align*}
\text{types-wellformed} \; \Gamma & \implies \\
\text{types-wellformed} \; (\Gamma \oplus_S \text{upd})
\end{align*}

(\textit{proof})
lemma types-stable-mode-update:
  \[ \text{types-stable } \Gamma S \implies \text{types-wellformed } \Gamma \implies \text{anno-type-stable } \Gamma S \text{ upd} \]
  \[ \implies \text{types-stable } (\Gamma \oplus_S \text{ upd}) (\text{add-anno-stable } S \text{ upd}) \]
  \langle proof \rangle

lemma tyenv-wellformed-mode-update:
  \[ \text{tyenv-wellformed } \text{ mds } \Gamma S P \implies \text{anno-type-stable } \Gamma S \text{ upd} \]
  \[ \implies \text{tyenv-wellformed } (\text{update-modes } \text{ upd } \text{ mds}) \]
  \[ (\Gamma \oplus_S \text{ upd}) \]
  \[ (\text{add-anno-stable } S \text{ upd}) \]
  \[ (P \mid^; \{ \text{v. stable } (\text{add-anno-stable } S \text{ upd}) \text{ v} \}) \]
  \langle proof \rangle

lemma stop-cxt :
  \[ [\vdash \Gamma.S.P \{ c \} \Gamma',S',P' : c = \text{Stop}] \implies \]
  \[ \text{context-equiv } \Gamma P \Gamma' \wedge S' = S \wedge P \vdash P' \wedge (\forall \text{ mds. tyenv-wellformed mds } \Gamma S P) \]
  \[ \implies \text{tyenv-wellformed mds } \Gamma' S P' \]
  \langle proof \rangle

lemma tyenv-wellformed-preds-update:
  \[ P' = P'' |^; \{ \text{v. stable } S \text{ v} \} \implies \]
  \[ \text{tyenv-wellformed mds } \Gamma S P \implies \text{tyenv-wellformed mds } \Gamma S P' \]
  \langle proof \rangle

lemma mds-consistent-preds-tyenv-update:
  \[ P' = P'' |^; \{ \text{v. stable } S \text{ v} \} \implies v \in \text{dom } \Gamma \implies \]
  \[ \text{mds-consistent mds } \Gamma S P \implies \text{mds-consistent mds } (\Gamma(v \mapsto t)) S P' \]
  \langle proof \rangle

lemma pred-preds-update:
  assumes mem'-def : mem' = mem (x := ev_A mem e)
  assumes P'-def : P' = (assign-post P x e) |^; \{ \text{v. stable } S \text{ v} \}
  assumes pred-P : pred P mem
  shows pred P' mem'
  \langle proof \rangle

lemma types-wellformed-update:
  \[ \text{types-wellformed } \Gamma \implies \text{type-wellformed } t \implies \text{types-wellformed } (\Gamma(x \mapsto t)) \]
  \langle proof \rangle

lemma types-stable-update:
  \[ \text{types-stable } \Gamma S \implies \text{type-stable } S t \implies \text{types-stable } (\Gamma(x \mapsto t)) S \]
  \langle proof \rangle
**Lemma** tyenv-wellformed-sub:

\[ P_1 \subseteq P_2; \]
\[ \Gamma_2 = \Gamma_1; \text{ tyenv-wellformed mds } \Gamma_2 S P_2 \]\n\[ \Rightarrow \text{ tyenv-wellformed mds } \Gamma_1 S P_1 \]

(proof)

**Abbreviation**

\[ \text{ tyenv-sec } :: ' \text{ Var } Mds \Rightarrow (' \text{ Var }, 'BExp) \text{ TyEnv } \Rightarrow (' \text{ Var }, 'Val) \text{ Mem } \Rightarrow \text{ bool } \]

**Where**

\[ \text{ tyenv-sec mds } \Gamma \text{ mem } \equiv \forall x \in \text{ dom } \Gamma, x \notin \text{ mds } \text{ AsmNoReadOrWrite } \Rightarrow \text{ type-max } (\text{ the } (\Gamma x)) \text{ mem } \leq \text{ dma mem x } \]

**Lemma** tyenv-sec-mode-update:

\[ (\forall x. \text{ (to-total } (\Gamma x)) \leq_{P''} \text{ (to-total } (\Gamma'' x))) \Rightarrow \text{ pred } P'' \text{ mem } \Rightarrow S = (\text{ mds AsmNoWrite}, \text{ mds } \text{ AsmNoReadOrWrite}) \]
\[ \Rightarrow \text{ anno-type-sec } \Gamma S P \text{ upd } \Rightarrow S'' = \text{ add-anno-stable } S \text{ upd } \Rightarrow (\forall p \in P. \forall v \in \text{ bexp-vars p, stable } S v) \Rightarrow \]
\[ P'' = P | \{ v, \text{ stable } S'' v \} \Rightarrow \]
\[ \Gamma'' = \Gamma \oplus_{S} \text{ upd } \Rightarrow \text{ tyenv-sec mds } \Gamma \text{ mem } \Rightarrow \text{ tyenv-sec } (\text{ update-modes upd mds}) \Gamma'' \text{ mem } \]

(proof)

**Lemma** tyenv-sec-eq:

\[ \forall x \in C. \text{ mem x } = \text{ mem' } x \Rightarrow \text{ types-wellformed } \Gamma \Rightarrow \text{ tyenv-sec mds } \Gamma \text{ mem } = \text{ tyenv-sec mds } \Gamma \text{ mem'} \]

(proof)

**Lemma** context-equiv-tyenv-sec:

\[ \text{ context-equiv } \Gamma_2 P_2 \Gamma_1 \Rightarrow \]
\[ \text{ pred } P_2 \text{ mem } \Rightarrow \text{ tyenv-sec mds } \Gamma_2 \text{ mem } \Rightarrow \text{ tyenv-sec mds } \Gamma_1 \text{ mem } \]

(proof)

**Lemma** add-pred-entailment:

\[ P +_S P \vdash P \]

(proof)

**Lemma** preservation-no-await:

\[ \vdash \Gamma, S.P \{ c \} \Gamma', S', P'; \]
\[ \langle c, \text{ mds, mem } \rangle \Rightarrow \langle c', \text{ mds', mem' } \rangle; \]
\[ \text{ no-await } c \]
\[ \Rightarrow \exists \Gamma'' S'' P''. (\vdash \Gamma'', S'', P''. \{ c' \} \Gamma'', S', P') \land \]
\[ (\text{ tyenv-wellformed mds } \Gamma S P \land \text{ pred } P \text{ mem } \land \text{ tyenv-sec mds } \Gamma \text{ mem } \Rightarrow \]
\[ \text{ tyenv-wellformed mds' } \Gamma'' S'' P'' \land \]
\[ \text{ pred } P'' \text{ mem' } \land \]
\[ \text{ tyenv-sec mds' } \Gamma'' \text{ mem' }) \]

(proof)
lemma preservation-no-await-plus:
\[ (c, mds, mem) \rightsquigarrow^+ (c', mds', mem'); \]
\[ \vdash \Gamma, S, P \{ c \} \Gamma', S', P'; \]
\[ \text{no-await } c' \;\implies\; \text{no-await } c' \land (\exists \Gamma'', S'', P''. (\vdash \Gamma'', S'', P''. \{ c' \} \Gamma', S', P') \land \tyenv-wellformed mds \Gamma S P \land pred P mem \land tyenv-sec mds \Gamma mem \implies tyenv-wellformed mds' \Gamma'' S'' P'' \land pred P'' mem' \land tyenv-sec mds' \Gamma'' mem')) \]

\[ \langle \text{proof} \rangle \]

lemma preservation:
\[ \text{assumes typed: } \;\vdash \;\Gamma, S, P \{ c \} \Gamma', S', P' \]
\[ \text{assumes eval: } \;\langle c, mds, mem \rangle \rightsquigarrow \langle c', mds', mem' \rangle \]
\[ \text{shows } \exists \Gamma'', S'', P''. (\vdash \Gamma'', S'', P''. \{ c' \} \Gamma', S', P') \land \tyenv-wellformed mds \Gamma S P \land pred P mem \land tyenv-sec mds \Gamma mem \implies tyenv-wellformed mds' \Gamma'' S'' P'' \land pred P'' mem' \land tyenv-sec mds' \Gamma'' mem') \]

\[ \langle \text{proof} \rangle \]

inductive-cases await-type-elim:
\[ \vdash \Gamma, S, P \{ \text{Await } b \;\text{ca} \} \Gamma', S', P' \]

fun bisim-helper :: (('Var', 'AExp', 'BExp') Stmt, 'Var', 'Val) LocalConf ⇒
\[ (('Var', 'AExp', 'BExp') Stmt, 'Var', 'Val) LocalConf ⇒ \text{bool} \]
where
\[ \text{bisim-helper } \langle c_1, mds, mem_1 \rangle \langle c_2, mds_2, mem_2 \rangle = \text{mem}_1 =_{mds} \text{mem}_2 \]

lemma R₃-mem-eq: \[ \langle c_1, mds, mem_1 \rangle R₃ \Gamma', S', P' \langle c_2, mds, mem_2 \rangle \implies \text{mem}_1 =_{mds} \text{mem}_2 \]
\[ \langle \text{proof} \rangle \]

lemma ev₁A-eq:
\[ \text{assumes tyenv-eq: } \text{mem}_1 =_{Γ} \text{mem}_2 \]
\[ \text{assumes pred: } \text{pred } P \text{ mem}_1 \]
\[ \text{assumes e-type: } \Gamma \vdash_{a} e \in t \]
\[ \text{assumes subtype: } t \subseteq_{P} (\text{dma-type } v) \]
\[ \text{assumes is-Low: } \text{dma mem}_1 v = \text{Low} \]
\[ \text{shows } ev₁A \text{ mem}_1 e = ev₁A \text{ mem}_2 e \]
\[ \langle \text{proof} \rangle \]

lemma ev₁A-eq'::
\[ \text{assumes tyenv-eq: } \text{mem}_1 =_{Γ} \text{mem}_2 \]
\[ \text{assumes pred: } \text{pred } P \text{ mem}_1 \]
\[ \text{assumes e-type: } \Gamma \vdash_{a} e \in t \]
\[ \text{assumes subtype: } P \vdash_{t} \]
\[ \text{shows } ev₁A \text{ mem}_1 e = ev₁A \text{ mem}_2 e \]
(proof)

**Lemma ev\_B-eq':**

- **Assumes** tyenv-eq: mem\(_1 = \Gamma\) mem\(_2\)
- **Assumes** pred: pred \(P\) mem\(_1\)
- **Assumes** e-type: \(\Gamma \vdash e \in t\)
- **Assumes** subtype: \(P \vdash t\)
- **Shows** ev\_B mem\(_1\) \(e = ev\_B\) mem\(_2\) \(e\)

(\textit{proof})

**Lemma R1-equiv-entailment:**

\((c, mds, mem) \Rightarrow^R^1 (\Gamma, S', P', \langle c', mds', mem' \rangle) \Rightarrow\)

- **context-equiv** \(\Gamma' P' \Gamma'' \Rightarrow P'' \Rightarrow\)
- **\(\forall mds.\)** tyenv-wellformed \(mds\) \(\Gamma' S' P' \Rightarrow tyenv-wellformed mds\) \(\Gamma'' S' P'' \Rightarrow\)

\((c, mds, mem) \Rightarrow^R^1 (\Gamma, S', P', \langle c', mds', mem' \rangle) \Rightarrow\)

(\textit{proof})

**Lemma R3-equiv-entailment:**

\((c, mds, mem) \Rightarrow^R^3 (\Gamma, S', P', \langle c', mds', mem' \rangle) \Rightarrow\)

- **context-equiv** \(\Gamma' P' \Gamma'' \Rightarrow P'' \Rightarrow\)
- **\(\forall mds.\)** tyenv-wellformed \(mds\) \(\Gamma' S' P' \Rightarrow tyenv-wellformed mds\) \(\Gamma'' S' P'' \Rightarrow\)

\((c, mds, mem) \Rightarrow^R^3 (\Gamma, S', P', \langle c', mds', mem' \rangle) \Rightarrow\)

(\textit{proof})

**Lemma R-equiv-entailment:**

\(lc_1 \Rightarrow^R^u (\Gamma, S', P', \langle lc_2 \rangle) \Rightarrow\)

- **context-equiv** \(\Gamma' P' \Gamma'' \Rightarrow P'' \Rightarrow\)
- **\(\forall mds.\)** tyenv-wellformed \(mds\) \(\Gamma' S' P' \Rightarrow tyenv-wellformed mds\) \(\Gamma'' S' P'' \Rightarrow\)

\(lc_1 \Rightarrow^R^u (\Gamma, S', P', \langle lc_2 \rangle) \Rightarrow\)

(\textit{proof})

**Lemma context-equiv-tyenv-eq:**

\(tyenv-eq\) \(\Gamma\) mem mem\(_r\) \(\Rightarrow\) **context-equiv** \(\Gamma P\) \(\Gamma'') \(\Rightarrow\) pred \(P\) mem \(\Rightarrow\) tyenv-eq \(\Gamma'\) mem mem\(_r\)

(\textit{proof})

**Lemma R-typed-step-no-await:**

\(\parallel \Gamma, S, P \{ c_1 \} \Rightarrow (\Gamma, S', P')\)

- tyenv-wellformed \(mds\) \(\Gamma S P\); mem\(_1 = \Gamma\) mem\(_2\); **pred** \(P\) mem\(_1\):
- \(pred\) mem\(_2\); tyenv-sec \(mds\) \(\Gamma\) mem\(_1\):
- \(\langle c_1, mds, mem_1 \rangle \Rightarrow \langle c_1', mds', mem_1' \rangle\); no-await \(c_1\)
- \(\exists c_2'\) mem\(_1'\); \(\langle c_2', mds', mem_2' \rangle \Rightarrow \langle c_1, mds, mem_1 \rangle\) \(\Rightarrow\)

\(\langle c_1, mds, mem_1 \rangle \Rightarrow^R^u (\Gamma, S', P' \langle c_2', mds', mem_2' \rangle)\)

(\textit{proof})

**Lemma is-final-\(R_a\)-is-final:**

\((c_1, mds, mem_1) \Rightarrow^R^u (\Gamma, S, P \langle c_2, mds, mem_2 \rangle) \Rightarrow is-final\) \(c_1\) \(\Rightarrow is-final\) \(c_2\)
\[\text{lemma pred-plus-impl:}\]
\[\text{pred } P \text{ mem } \implies \text{ ev}_B \text{ mem } e \implies \text{ pred } P +_S e \text{ mem}\]
\[\langle \text{proof} \rangle\]

\[\text{lemma } \mathcal{R}_3\text{-aux-induct [consumes 1, case-names intro1 intrs3]}:\]
\[
[(c_1, \text{mds, mem}_1) \mathcal{R}_3^S \Gamma, S, P \langle c_2, \text{mds, mem}_2 \rangle; \]
\[
\bigwedge \text{mds mem}_1 \Gamma \ S \ P \ c_2 \ \text{mem}_2 \ c \ \Gamma' \ S' \ P'.
\]
\[
[(c_1, \text{mds, mem}_1) \mathcal{R}_3^S \Gamma, S, P \langle c_2, \text{mds, mem}_2 \rangle; \]
\[
\vdash \Gamma, S, P \{c_1\} \Gamma', S', P' \implies
\]
\[Q \ (c_1 \ V C; \ c) \ \text{mds mem}_1 \ \Gamma' \ S' \ P' \ (c_2 \ V C; \ c) \ \text{mds mem}_2; \]
\[
\bigwedge \text{mds mem}_1 \Gamma \ S \ P \ c_2 \ \text{mds mem}_2;
\]
\[
(c_1, \text{mds, mem}_1) \mathcal{R}_3^S \Gamma, S, P \langle c_2, \text{mds, mem}_2 \rangle;
\]
\[
\vdash \Gamma, S, P \{c_1\} \Gamma', S', P' \implies
\]
\[Q \ (c_1 \ V C; \ c) \ \text{mds mem}_1 \ \Gamma' \ S' \ P' \ (c_2 \ V C; \ c) \ \text{mds mem}_2 \implies
\]
\[Q \ c_1 \ \text{mds mem}_1 \ \Gamma \ S \ P \ c_2 \ \text{mds mem}_2
\]
\[\langle \text{proof} \rangle\]

\[\text{lemma } \mathcal{R}\text{-typed-step-plus:}\]
\[
[(c_1, \text{mds, mem}_1) \xrightarrow{t} (c_1', \text{mds}', \text{mem}_1'); \]
\[
\vdash \Gamma, S, P \{c_1\} \Gamma', S', P';
\]
\[\text{no-await } c_1;\]
\[\text{tyenv-sec mem} \ \text{mds} \ \Gamma \ S \ P;\]
\[\text{mem}_1 = \Gamma \ \text{mem}_2;\]
\[\text{pred } P \ \text{mem}_1;\]
\[\text{pred } P \ \text{mem}_2;\]
\[\text{tyenv-sec mem} \ \text{mds} \ \Gamma \ \text{mem}_1 \implies
\]
\[\exists c_2' \ \text{mem}_2'. (c_1, \text{mds, mem}_2) \xrightarrow{t} (c_2', \text{mds}', \text{mem}_2') \land
\]
\[\langle c_1', \text{mds}', \text{mem}_1' \rangle \mathcal{R}_3^S \Gamma', S', P' \langle c_2', \text{mds}', \text{mem}_2' \rangle
\]
\[\langle \text{proof} \rangle\]

\[\text{lemma } \mathcal{R}\text{-typed-step:}\]
\[
[\vdash \Gamma, S, P \{c_1\} \Gamma', S', P';\]
\[\text{tyenv-sec mem} \ \text{mds} \ \Gamma \ S \ P; \ \text{mem}_1 = \Gamma \ \text{mem}_2; \ \text{pred } P \ \text{mem}_1;\]
\[\text{pred } P \ \text{mem}_2;\]
\[\text{tyenv-sec mem} \ \text{mds} \ \Gamma \ \text{mem}_1;\]
\[\langle c_1, \text{mds, mem}_1 \rangle \xrightarrow{t} \langle c_1', \text{mds}', \text{mem}_1' \rangle \implies
\]
\[\exists c_2' \ \text{mem}_2'. (c_1, \text{mds, mem}_2) \xrightarrow{t} (c_2', \text{mds}', \text{mem}_2') \land
\]
\[\langle c_1', \text{mds}', \text{mem}_1' \rangle \mathcal{R}_3^S \Gamma', S', P' \langle c_2', \text{mds}', \text{mem}_2' \rangle
\]
\[\langle \text{proof} \rangle\]

\[\text{lemma } \mathcal{R}_1\text{-weak-bisim:}\]
\[\text{weak-bisim } (\mathcal{R}_1 \ \Gamma' \ S' \ P') \ (\mathcal{R} \ \Gamma' \ S' \ P')\]
\[\langle \text{proof} \rangle\]
lemma $\mathcal{R}$-to-$\mathcal{R}_3$: \[ \langle c_1, \text{mds}, \text{mem}_1 \rangle \mathcal{R}^u \Gamma, \mathcal{S}, \mathcal{P} \langle c_2, \text{mds}, \text{mem}_2 \rangle ; \vdash \Gamma, \mathcal{S}, \mathcal{P} \{ c \} \quad \Gamma', \mathcal{S}', \mathcal{P}' \] \implies \langle c_1 \triangleright; c, \text{mds}, \text{mem}_1 \rangle \mathcal{R}^3 \Gamma', \mathcal{S}', \mathcal{P}' \langle c_2 \triangleright; c, \text{mds}, \text{mem}_2 \rangle \]

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

lemma $\mathcal{R}$-bisim: strong-low-bisim-mm ($\mathcal{R} \Gamma' \mathcal{S}' \mathcal{P}'$)
(proof)

lemma Typed-in-$\mathcal{R}$:
assumes typeable: $\vdash \Gamma, \mathcal{S}, \mathcal{P} \{ c \} \Gamma', \mathcal{S}', \mathcal{P}'$
assumes wf: tyenv-wellformed mds $\Gamma \mathcal{S} \mathcal{P}$
assumes mem-eq: $\forall x. \text{type-max} (\text{to-total} \Gamma x) \text{mem}_1 = \text{Low} \rightarrow \text{mem}_1 x = \text{mem}_2 x$
assumes pred_1: pred $P$ mem_1
assumes pred_2: pred $P$ mem_2
assumes tyenv-sec: tyenv-sec mds $\Gamma \text{mem}_1$
shows $\langle c, \text{mds}, \text{mem}_1 \rangle \mathcal{R}^u \Gamma', \mathcal{S}', \mathcal{P}' \langle c, \text{mds}, \text{mem}_2 \rangle$
(proof)

theorem type-soundness:
assumes well-typed: $\vdash \Gamma, \mathcal{S}, \mathcal{P} \{ c \} \Gamma', \mathcal{S}', \mathcal{P}'$
assumes wf: tyenv-wellformed mds $\Gamma \mathcal{S} \mathcal{P}$
assumes mem-eq: $\forall x. \text{type-max} (\text{to-total} \Gamma x) \text{mem}_1 = \text{Low} \rightarrow \text{mem}_1 x = \text{mem}_2 x$
assumes pred_1: pred $P$ mem_1
assumes pred_2: pred $P$ mem_2
assumes tyenv-sec: tyenv-sec mds $\Gamma \text{mem}_1$
shows $\langle c, \text{mds}, \text{mem}_1 \rangle \approx \langle c, \text{mds}, \text{mem}_2 \rangle$
(proof)

definition $\Gamma$-of-mds :: 'Var Mds => ('Var, 'BExp) TyEnv
where
$\Gamma$-of-mds mds \equiv (\lambda x. \text{if} x \notin C \land x \in \text{mds AsmNoWrite} \cup \text{mds AsmNoReadOrWrite} \text{then}
  \text{if} x \in \text{mds AsmNoReadOrWrite} \text{then}
    \text{Some} \{ \{\text{pred-False}\} \}
  \text{else}
    \text{Some} \{ \text{dma-type} x \}
  \text{else} \text{None})$

definition
\( \mathcal{S} \text{-of-mds} :: '\text{Var} \text{ Mds} \Rightarrow '\text{Var Stable} \)

**where**

\( \mathcal{S} \text{-of-mds} \text{ mds} \equiv (\text{mds AsmNoWrite}, \text{mds AsmNoReadOrWrite}) \)

**definition**

\( \text{mds-yields-stable-types} :: '\text{Var} \text{ Mds} \Rightarrow \text{bool} \)

**where**

\( \text{mds-yields-stable-types} \text{ mds} \equiv \forall x. x \in \text{mds AsmNoWrite} \cup \text{mds AsmNoReadOrWrite} \rightarrow (\forall v \in C \text{-vars} x. v \in \text{mds AsmNoWrite} \cup \text{mds AsmNoReadOrWrite}) \)

**inductive**

\( \text{type-global} :: (('\text{Var}, '\text{AExp}, '\text{BExp}) \text{Stmt} \times '\text{Var} \text{ Mds}) \text{ list} \Rightarrow \text{bool} \)

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

**where**

\[
\text{list-all} (\lambda (c, \text{m}). (\exists \Gamma' \ S' \ P'). \vdash (\text{\text{of-mds} m},(\text{\text{of-mds} m}),\{\} \; \{ c \} \; \Gamma',S',P')) \land \text{mds-yields-stable-types} \; \text{m} \; \text{cs} ;
\forall \text{mem. sound-mode-use} \; (\text{cs}, \text{mem})
\]

\( \text{type-global} \; \text{cs} \)

**inductive-cases**

\( \text{type-global-elim}: \vdash \text{cs} \)

**lemma**

\( \text{of-mds-tyenv-wellformed}: \text{mds-yields-stable-types} \; \text{m} \Rightarrow \text{tyenv-wellformed} \; \text{m} \; (\text{\text{of-mds} m}) \; (\text{\text{of-mds} m}) \; \{\} \)

\( \langle \text{proof} \rangle \)

**lemma**

\( \text{\text{Gamma-of-mds-tyenv-sec}}: \text{tyenv-sec} \; \text{m} \; (\text{\text{Gamma-of-mds} m}) \; \text{mem}_1 \)

\( \langle \text{proof} \rangle \)

**lemma**

\( \text{type-max-pred-False} \; [\text{simp}]: \text{type-max} \; \{\text{pred-False}\} \; \text{mem} = \text{High} \)

\( \langle \text{proof} \rangle \)

**lemma**

\( \text{typed-secure}: \text{\[\vdash (\text{\text{Gamma-of-mds} m},(\text{\text{of-mds} m}),\{\} \; \{ c \} \; \Gamma',S',P'; \text{mds-yields-stable-types} \; \text{m}]) \Rightarrow \text{com-sifum-secure} \; (c,m) \]

\( \langle \text{proof} \rangle \)

**lemma**

\( \text{list-all-set}: \forall x \in \text{set} \; \text{xs.} \; P \; x \Rightarrow \text{list-all} \; P \; \text{xs} \)

\( \langle \text{proof} \rangle \)

**theorem**

\( \text{type-soundness-global}: \)

**assumes**

\( \text{typeable}: \vdash \text{cs} \)

**shows**

\( \text{prog-sifum-secure-cont} \; \text{cs} \)
6 Type System for Ensuring Locally Sound Use of Modes

theory LocallySoundModeUse
imports Security Language
begin

6.1 Typing Rules

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

abbreviation
eval-abv-modes :: (-, Val) LocalConf ⇒ (-, -) LocalConf ⇒ bool
  (infixl ⇝ 70)
where
  x ⇝ y ≡ (x, y) ∈ eval_w

fun
  update-annos :: 'Var Mds ⇒ 'Var ModeUpd list ⇒ 'Var Mds
  (infix ⊕ 140)
where
  update-annos mds [] = mds |
  update-annos mds (a # as) = update-annos (update-modes a mds) as

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

mode-type :: 'Var Mds ⇒ ('Var, 'AExp, 'BExp) Stmt ⇒ 'Var Mds ⇒ bool (⊢ - { - } -)

where

skip: ⊢ mds { Skip ⊗ annos } (mds ⊕ annos) |

assign: [ x ∉ mds GuarNoWrite ∪ mds GuarNoReadOrWrite ; aexp-vars e ⊙ mds GuarNoReadOrWrite = {} ];
∀ v. (x ∈ C-vars v → v ∉ mds GuarNoWrite ∪ mds GuarNoReadOrWrite)
∧ (C-vars v ∩ aexp-vars e ≠ {} → v ∉ mds GuarNoReadOrWrite)]
⇒
⊢ mds { (x ← e) ⊗ annos } (mds ⊕ annos) |

if: [ [ ⊢ (mds ⊕ annos) { c₁ } mds'' ];
 ⊢ (mds ⊕ annos) { c₂ } mds'' ];
 bexp-vars e ⊙ mds GuarNoReadOrWrite = {} ;
∀ v. C-vars v ∩ bexp-vars e ≠ {} → v ∉ mds GuarNoReadOrWrite] ⇒
⊢ mds { If e c₁ c₂ ⊗ annos } mds'' |

while: [ mds'' = mds ⊕ annos ; ⊢ mds' { c } mds'' ; bexp-vars e ⊙ mds' GuarNoReadOrWrite = {} ];
∀ v. C-vars v ∩ bexp-vars e ≠ {} → v ∉ mds' GuarNoReadOrWrite]
⇒
⊢ mds { While e c ⊗ annos } mds' |

seq: [ ⊢ mds { c₁ } mds'' ; ⊢ mds' { c₂ } mds'' ] ⇒ ⊢ mds { c₁ ; c₂ } mds'' |

sub: [ ⊢ mds₂ { c } mds₂' ; mds₁ ≤ mds₂ ; mds₂' ≤ mds₁' ] ⇒
⊢ mds₁ { c } mds₁'

6.2 Soundness of the Type System

lemma cxt-eval:
[ ≫ { cxt-to-stmt [] c, mds, mem } ≃ { cxt-to-stmt [] c', mds', mem'} ] ≫
(c, mds, mem) ≃ { c', mds', mem'}
(proof)

lemma update-preserves-le:
mds₁ ≤ mds₂ ⇒ (mds₁ ⊕ annos) ≤ (mds₂ ⊕ annos)
(proof)

lemma doesnt-read-annos:
doesnt-read-or-modify c x ⇒ doesnt-read-or-modify (c ⊗ annos) x
(proof)

lemma doesnt-modify-annos:
doesnt-modify c x ⇒ doesnt-modify (c ⊗ annos) x
(proof)
lemma stop-loc-reach:
\[ \{ \langle c', mds', \text{mem}' \rangle \in \text{loc-reach} \langle \text{Stop}, mds, \text{mem} \rangle \} \implies c' = \text{Stop} \land mds' = mds \]
(proof)

lemma stop-doesnt-access:
\[ \text{doesn't-modify} \text{ Stop} x \land \text{doesn't-read-or-modify} \text{ Stop} x \]
(proof)

lemma skip-eval-step:
\( (\text{Skip} \otimes \text{annos}, mds, \text{mem}) \rightsquigarrow (\text{Stop} \otimes \text{annos}, mds, \text{mem}) \)
(proof)

lemma skip-eval-elim:
\[ \{ \langle \text{Skip} \otimes \text{annos}, mds, \text{mem} \rangle \rightsquigarrow \langle \text{c}', mds', \text{mem}' \rangle \} \implies c' = \text{Stop} \land mds' = mds \]
(proof)

lemma skip-doesnt-read:
\[ \text{doesn't-read-or-modify} (\text{Skip} \otimes \text{annos}) \text{ x} \]
(proof)

lemma skip-doesnt-write:
\[ \text{doesn't-modify} (\text{Skip} \otimes \text{annos}) \text{ x} \]
(proof)

lemma skip-loc-reach:
\[ \{ \langle \text{c}', mds', \text{mem}' \rangle \in \text{loc-reach} \langle \text{Skip} \otimes \text{annos}, mds, \text{mem} \rangle \} \implies (c' = \text{Stop} \land mds' = (mds \oplus \text{annos})) \lor (c' = \text{Skip} \otimes \text{annos} \land mds' = mds) \]
(proof)

lemma skip-doesnt-access:
\[ \{ \text{lc} \in \text{loc-reach} \langle \text{Skip} \otimes \text{annos}, mds, \text{mem} \rangle ; \text{lc} = \langle \text{c}', mds', \text{mem}' \rangle \} \implies \text{doesn't-read-or-modify} \text{ c}' \text{ x} \land \text{doesn't-modify} \text{ c}' \text{ x} \]
(proof)

lemma assign-doesnt-modify:
\[ \{ x \neq y; x \notin \text{C-vars} y \} \implies \text{doesn't-modify} ((x \leftarrow e) \otimes \text{annos}) y \]
(proof)

lemma assign-annos-eval:
\[ ((x \leftarrow e) \otimes \text{annos}, mds, \text{mem}) \rightsquigarrow (\text{Stop} \otimes \text{annos}, \text{mem} (x := \text{ev}_A \text{ mem e})) \]
(proof)

lemma assign-annos-eval-elim:
\[ \{ \langle x \leftarrow e \rangle \otimes \text{annos}, mds, \text{mem} \rangle \rightsquigarrow \langle \text{c}', mds', \text{mem}' \rangle \} \implies c' = \text{Stop} \land mds' = mds \oplus \text{annos} \]
(proof)
lemma mem-upd-commute:
\[ \{ x \neq y \} \implies \text{mem} (x := v_1, y := v_2) = \text{mem} (y := v_2, x := v_1) \]
(proof)

lemma assign-doesnt-read:
\[ \{ y \neq x \land \text{aexp-vars } e \land \text{C-vars } y \land \text{aexp-vars } e = \{ \} \} \implies \text{doesnt-read-or-modify } ((x ← e) \otimes \text{annos}) y \]
(proof)

lemma assign-loc-reach:
\[ \{ \langle c', mds', \text{mem}' \rangle \in \text{loc-reach } ((x ← e) \otimes \text{annos}, mds, \text{mem}) \} \implies \langle c' = \text{Stop} \land mds' = (mds \oplus \text{annos}) \rangle \lor \langle c' = (x ← e) \otimes \text{annos} \land mds' = mds \rangle \]
(proof)

lemma if-doesnt-modify:
\[ \text{doesnt-modify } (\text{If } e \ c_1 \ c_2 \otimes \text{annos}) x \]
(proof)

lemma vars-eval_B:
\[ x \notin \text{bexp-vars } e \implies \text{ev}_B \text{mem } e = \text{ev}_B \text{mem } (x := v) \]
(proof)

lemma if-doesnt-read:
\[ x \notin \text{bexp-vars } e \implies \text{C-vars } x \land \text{bexp-vars } e = \{ \} \implies \text{doesnt-read-or-modify } (\text{If } e \ c_1 \ c_2 \otimes \text{annos}) x \]
(proof)

lemma if-eval-true:
\[ \{ \text{ev}_B \text{mem } e \} \implies \langle e_1, mds \oplus \text{annos}, \text{mem} \rangle \]
(proof)

lemma if-eval-false:
\[ \{ \lnot \text{ev}_B \text{mem } e \} \implies \langle e_2, mds \oplus \text{annos}, \text{mem} \rangle \]
(proof)

lemma if-eval-elim:
\[ \{ \langle e_1, mds, \text{mem} \rangle \lnot \text{ev}_B \text{mem } e \} \implies \langle c_1 \land \text{ev}_B \text{mem } e \rangle 
\lor \langle c_2 \land \lnot \text{ev}_B \text{mem } e \rangle 
\land mds' = mds \land \text{mem}' = \text{mem} \]
(proof)

lemma if-eval-elim':
\[ \{ \langle e_1, mds, \text{mem} \rangle \implies \langle c_1 \land \text{ev}_B \text{mem } e \rangle 
\lor \langle c_2 \land \lnot \text{ev}_B \text{mem } e \rangle 
\land mds' = mds \land \text{mem}' = \text{mem} \}
\]
(proof)
**lemma** $\text{loc-reach-refl}':$

\[
\langle c', mds', \text{mem}' \rangle \in \text{loc-reach} \langle c, mds, \text{mem} \rangle \\
\langle \text{proof} \rangle
\]

**lemma** $\text{if-loc-reach}:$

\[
\langle \langle c', mds', \text{mem}' \rangle \in \text{loc-reach} \langle \text{If e c_1 c_2 \otimes \text{annos}, mds, \text{mem} \rangle \rangle \rightarrow \\
\langle c' = \text{If e c_1 c_2 \otimes \text{annos} \land mds' = mds \rangle \rangle \\
\langle \exists \text{mem''}. \langle c', mds', \text{mem}' \rangle \in \text{loc-reach} \langle c_1, mds \oplus \text{annos, mem''} \rangle \rangle \\
\langle \exists \text{mem''}. \langle c', mds', \text{mem}' \rangle \in \text{loc-reach} \langle c_2, mds \oplus \text{annos, mem''} \rangle \rangle \\
\langle \text{proof} \rangle
\]

**lemma** $\text{if-loc-reach}':$

\[
\langle \langle c', mds', \text{mem}' \rangle \in \text{loc-reach} \langle \text{If e c_1 c_2, mds, \text{mem} \rangle \rangle \rightarrow \\
\langle c' = \text{If e c_1 c_2 \land mds' = mds \rangle \rangle \\
\langle \exists \text{mem''}. \langle c', mds', \text{mem}' \rangle \in \text{loc-reach} \langle c_1, mds, \text{mem''} \rangle \rangle \\
\langle \exists \text{mem''}. \langle c', mds', \text{mem}' \rangle \in \text{loc-reach} \langle c_2, mds, \text{mem''} \rangle \rangle \\
\langle \text{proof} \rangle
\]

**lemma** $\text{seq-loc-reach}:$

\[
\langle \langle c', mds', \text{mem}' \rangle \in \text{loc-reach} \langle c_1 : c_2, mds, \text{mem} \rangle \rangle \rightarrow \\
\langle \exists c''. c' = c'' \land c_2 \land \langle c'', mds', \text{mem}' \rangle \in \text{loc-reach} \langle c_1, mds, \text{mem} \rangle \rangle \\
\langle \exists c''. mds'' \text{mem''}. \langle \text{Stop, mds'', \text{mem''} \rangle \in \text{loc-reach} \langle c_1, mds, \text{mem} \rangle \land \\
\langle c', mds', \text{mem}' \rangle \in \text{loc-reach} \langle c_2, mds'', \text{mem''} \rangle \rangle \\
\langle \text{proof} \rangle
\]

**lemma** $\text{seq-doesnt-read}:$

\[
\langle \text{doesnt-read-or-modify c x \rangle \rightarrow \text{doesnt-read-or-modify (c ;; c') x} \rangle \langle \text{proof} \rangle
\]

**lemma** $\text{seq-doesnt-modify}:$

\[
\langle \text{doesnt-modify c x \rangle \rightarrow \text{doesnt-modify (c ;; c') x} \rangle \langle \text{proof} \rangle
\]

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

**lemma** $\text{seq-stop-elim}: \langle \text{Stop ;; c, mds, \text{mem} \rangle \rightarrow \langle c', mds', \text{mem}' \rangle \rightarrow \\
\langle c' = c \land mds' = mds \land \text{mem'} = \text{mem} \rangle \rightarrow \\
\langle \text{proof} \rangle
\]

**lemma** $\text{seq-split}:$

\[
\langle \langle \text{Stop, mds', \text{mem}' \rangle \in \text{loc-reach} \langle c_1 : c_2, mds, \text{mem} \rangle \rangle \rightarrow \\
\exists mds'' \text{mem'}. \langle \text{Stop, mds'', \text{mem''} \rangle \in \text{loc-reach} \langle c_1, mds, \text{mem} \rangle \land \\
\langle \text{Stop, mds', \text{mem}' \rangle \in \text{loc-reach} \langle c_2, mds'', \text{mem''} \rangle \rangle \\
\langle \text{proof} \rangle
\]

**lemma** $\text{while-eval}:$

\[
\langle \text{While e c \otimes \text{annos, mds, \text{mem} \rangle \rightarrow ((\text{If e (c ;; While e c) \text{Stop}, mds \oplus \text{annos, \text{mem} \rangle})} \rangle
\]

56
lemma while-eval':
\[ \{ \text{While } e \ c, \ mds, \ mem \} \rightsquigarrow \{ \text{If } e \ (e \ ;; \ \text{While } e \ c) \ \text{Stop}, \ mds, \ mem \} \]
(proof)

lemma while-eval-elim:
\[ \{ \langle \text{While } e \ c \ \otimes \ \text{annos}, \ mds, \ mem \rangle \rightsquigarrow \langle e', \ mds', \ mem' \rangle \} \implies \]
\[ (e' = \text{If } e \ (e \ ;; \ \text{While } e \ c) \ \text{Stop} \land mds' = mds \otimes \text{annos} \land mem' = mem) \]
(proof)

lemma while-eval-elim':
\[ \{ \langle \text{While } e \ c, \ mds, \ mem \rangle \rightsquigarrow \langle e', \ mds', \ mem' \rangle \} \implies \]
\[ (e' = \text{If } e \ (e \ ;; \ \text{While } e \ c) \ \text{Stop} \land mds' = mds \land mem' = mem) \]
(proof)

lemma while-doesnt-read:
\[ x \notin bexp-vars e \implies \text{doesn't-read-or-modify} (\text{While } e \ c \ \otimes \ \text{annos}) \ x \]
(proof)

lemma while-doesnt-modify:
\[ \text{doesn't-modify} (\text{While } e \ c \ \otimes \ \text{annos}) \ x \]
(proof)

lemma disjE3:
\[ \{ A \lor B \lor C ; A \implies P ; B \implies P ; C \implies P \} \implies P \]
(proof)

lemma disjE5:
\[ \{ A \lor B \lor C \lor D \lor E ; A \implies P ; B \implies P ; C \implies P ; D \implies P ; E \implies P \} \]
\[ \implies P \]
(proof)

lemma if-doesnt-read':
\[ x \notin bexp-vars e \implies C-vars x \cap bexp-vars e = \{ \} \implies \text{doesn't-read-or-modify} (\text{If } e \ c_1 \ c_2) \ x \]
(proof)

theorem mode-type-sound:
assumes typeable: \( \vdash mds_1 \{ e \} mds_1' \)
assumes mode-le: \( mds_2 \leq mds_1 \)
shows \( \forall \text{ mem. } (\langle \text{Stop}, mds_2', \ mem' \rangle \in \text{loc-reach} \langle c, mds_2, \ mem \rangle \longrightarrow mds_2' \leq mds_1') \land \)
\[ \text{locally-sound-mode-use} \langle c, mds_2, \ mem \rangle \]
(proof)

end
References


