

A Dependent Security Type System for Concurrent Imperative Programs

Toby Murray, Robert Sison, Edward Pierzchalski and Christine Rizkallah

May 26, 2024

Abstract

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

The formalization includes the following parts:

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

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

Contents

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	28
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  $\vee$  d2 = High) then High else
Low)
```

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

```
definition bot-Sec-def: bot = Low
```

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

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

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

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

```
instance
  <proof>
```

end

Memories are mappings from variables to values

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

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

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

Local configurations:

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

Global configurations:

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

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

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

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

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

lemma C-vars-subset-C:
 C-vars x \subseteq C
 <proof>

lemma dma-C:

$\forall x \in \mathcal{C}. \text{mem}_1 x = \text{mem}_2 x \implies \text{dma mem}_1 = \text{dma mem}_2$
 ⟨proof⟩

end

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

$\llbracket (a, b) \in r^+;$
 $P a;$
 $\bigwedge x y. \llbracket (x, y) \in r; P x \rrbracket \implies P y \rrbracket \implies P b$
 ⟨proof⟩

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

$\llbracket (a, b) \in r^+;$
 $\bigwedge x y. (x, y) \in r \implies P x y;$
 $\bigwedge x y z. P x y \implies (y, z) \in r \implies P x z \rrbracket \implies P a b$
 ⟨proof⟩

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

$\llbracket (a, b) \in r^+;$
 $\bigwedge x y. (x, y) \in r \implies P x y;$
 $\bigwedge x y z. (x, y) \in r^+ \implies P x y \implies (y, z) \in r \implies P y z \implies P x z \rrbracket \implies P a b$
 ⟨proof⟩

lemmas *my-trancl-step-induct3* =

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

lemmas *my-trancl-big-step-induct3* =

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

end

2 Definition of the SIFUM-Security Property

theory *Security*

imports *Preliminaries*

begin

type-synonym $('var, 'val)$ *adaptation* = $'var \rightarrow ('val \times 'val)$

definition *apply-adaptation* ::

$bool \Rightarrow ('Var, 'Val) Mem \Rightarrow ('Var, 'Val) adaptation \Rightarrow ('Var, 'Val) Mem$

where *apply-adaptation first mem* $A =$

$(\lambda x. \text{case } (A x) \text{ of}$
 $\quad \text{Some } (v_1, v_2) \Rightarrow \text{if first then } v_1 \text{ else } v_2$
 $\quad | \text{None} \Rightarrow \text{mem } x)$

abbreviation $apply\text{-}adaptation_1 ::$

$(\text{'Var}, \text{'Val}) \text{Mem} \Rightarrow (\text{'Var}, \text{'Val}) \text{adaptation} \Rightarrow (\text{'Var}, \text{'Val}) \text{Mem}$
 $(- \llbracket_1 - \rrbracket [900, 0] 1000)$
where $mem \llbracket_1 A \rrbracket \equiv apply\text{-}adaptation \text{ True } mem \ A$

abbreviation $apply\text{-}adaptation_2 ::$

$(\text{'Var}, \text{'Val}) \text{Mem} \Rightarrow (\text{'Var}, \text{'Val}) \text{adaptation} \Rightarrow (\text{'Var}, \text{'Val}) \text{Mem}$
 $(- \llbracket_2 - \rrbracket [900, 0] 1000)$
where $mem \llbracket_2 A \rrbracket \equiv apply\text{-}adaptation \text{ False } mem \ A$

definition

$var\text{-}asm\text{-}not\text{-}written :: \text{'Var} \text{Mds} \Rightarrow \text{'Var} \Rightarrow \text{bool}$

where

$var\text{-}asm\text{-}not\text{-}written \ mds \ x \equiv x \in mds \ AsmNoWrite \vee x \in mds \ AsmNoReadOrWrite$

context $sifum\text{-}security\text{-}init \ \mathbf{begin}$

2.1 Evaluation of Concurrent Programs

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

$(\mathbf{infixl} \rightsquigarrow 70)$

where

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

abbreviation $conf\text{-}abv :: \text{'Com} \Rightarrow \text{'Var} \text{Mds} \Rightarrow (\text{'Var}, \text{'Val}) \text{Mem} \Rightarrow (-, -, -) \text{LocalConf}$

$((-, -, -) [0, 0, 0] 1000)$

where

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

inductive-set $meval :: ((-, -, -) \text{GlobalConf} \times \text{nat} \times (-, -, -) \text{GlobalConf}) \text{set}$

and $meval\text{-}abv :: - \Rightarrow - \Rightarrow - \Rightarrow \text{bool} \ (- \rightsquigarrow_ - \ 70)$

where

$conf \rightsquigarrow_k conf' \equiv (conf, k, conf') \in meval \mid$

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

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

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

inductive $neval :: (\text{'Com}, \text{'Var}, \text{'Val}) \text{LocalConf} \Rightarrow \text{nat} \Rightarrow (-, -, -) \text{LocalConf} \Rightarrow \text{bool}$

$(\mathbf{infixl} \rightsquigarrow^ - \ 70)$

where

$neval\text{-}0: x = y \Longrightarrow x \rightsquigarrow^0 y \mid$

$neval\text{-}S\text{-}n: x \rightsquigarrow y \Longrightarrow y \rightsquigarrow^n z \Longrightarrow x \rightsquigarrow^{Suc \ n} z$

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

lemma *neval-det*:
 $x \rightsquigarrow^n y \implies x \rightsquigarrow^n y' \implies y = y'$
 $\langle \text{proof} \rangle$

lemma *neval-Suc-simp* [*simp*]:
 $\text{neval } x \text{ (Suc } 0) y = x \rightsquigarrow y$
 $\langle \text{proof} \rangle$

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

fun
 $meval\text{-sched} :: \text{nat list} \Rightarrow ('Com, 'Var, 'Val) \text{ GlobalConf} \Rightarrow (-, -, -) \text{ GlobalConf} \Rightarrow \text{bool}$
where
 $meval\text{-sched} [] c c' = (c = c') \mid$
 $meval\text{-sched} (n\#\text{ns}) c c' = (\exists c''. c \rightsquigarrow_n c'' \wedge meval\text{-sched } ns c'' c')$

abbreviation
 $meval\text{-sched-}abv :: (-, -, -) \text{ GlobalConf} \Rightarrow \text{nat list} \Rightarrow (-, -, -) \text{ GlobalConf} \Rightarrow \text{bool} (- \rightarrow_- - \ 70)$
where
 $c \rightarrow_{ns} c' \equiv meval\text{-sched } ns c c'$

lemma *meval-sched-det*:
 $meval\text{-sched } ns c c' \implies meval\text{-sched } ns c c'' \implies c' = c''$
 $\langle \text{proof} \rangle$

2.2 Low-equivalence and Strong Low Bisimulations

definition
 $low\text{-eq} :: ('Var, 'Val) \text{ Mem} \Rightarrow (-, -) \text{ Mem} \Rightarrow \text{bool} (\text{infixl } =^l \ 80)$
where
 $mem_1 =^l mem_2 \equiv (\forall x. dma \ mem_1 \ x = Low \longrightarrow mem_1 \ x = mem_2 \ x)$

definition
 $low\text{-mds-}eq :: 'Var \text{ Mds} \Rightarrow ('Var, 'Val) \text{ Mem} \Rightarrow (-, -) \text{ Mem} \Rightarrow \text{bool}$
 $(- =^l - [100, 100] \ 80)$
where
 $(mem_1 =_{mds}^l mem_2) \equiv (\forall x. dma \ mem_1 \ x = Low \wedge (x \in \mathcal{C} \vee x \notin mds \ \text{Asm-NoReadOrWrite}) \longrightarrow mem_1 \ x = mem_2 \ x)$

lemma *low-eq-low-mds-eq*:
 $(mem_1 =^l mem_2) = (mem_1 =_{(\lambda m. \{\})} mem_2)$
 $\langle proof \rangle$

lemma *low-mds-eq-dma*:
 $(mem_1 =_{mds} mem_2) \implies dma\ mem_1 = dma\ mem_2$
 $\langle proof \rangle$

lemma *low-mds-eq-sym*:
 $(mem_1 =_{mds} mem_2) \implies (mem_2 =_{mds} mem_1)$
 $\langle proof \rangle$

lemma *low-eq-sym*:
 $(mem_1 =^l mem_2) \implies (mem_2 =^l mem_1)$
 $\langle proof \rangle$

lemma [*simp*]: $mem =^l mem' \implies mem =_{mds} mem'$
 $\langle proof \rangle$

lemma [*simp*]: $(\forall mds. mem =_{mds} mem') \implies mem =^l mem'$
 $\langle proof \rangle$

lemma *High-not-in-C* [*simp*]:
 $dma\ mem_1\ x = High \implies x \notin C$
 $\langle proof \rangle$

definition

closed-glob-consistent :: $((Com, Var, Val)\ LocalConf)\ rel \implies bool$
where
closed-glob-consistent $\mathcal{R} =$
 $(\forall c_1\ mds\ mem_1\ c_2\ mem_2. (\langle c_1, mds, mem_1 \rangle, \langle c_2, mds, mem_2 \rangle) \in \mathcal{R} \implies$
 $(\forall A. ((\forall x. case\ A\ x\ of\ Some\ (v, v') \implies (mem_1\ x \neq v \vee mem_2\ x \neq v')) \implies \neg$
 $var\ asm\ not\ written\ mds\ x \mid - \implies True) \wedge$
 $(\forall x. dma\ (mem_1\ [\![\!_1\ A])\ x \neq dma\ mem_1\ x \implies \neg\ var\ asm\ not\ written\ mds$
 $x) \wedge$
 $(\forall x. dma\ (mem_1\ [\![\!_1\ A])\ x = Low \wedge (x \notin mds\ AsmNoReadOrWrite \vee x \in$
 $C) \implies (mem_1\ [\![\!_1\ A])\ x = (mem_2\ [\![\!_2\ A])\ x)) \implies$
 $(\langle c_1, mds, mem_1[\![\!_1\ A] \rangle], \langle c_2, mds, mem_2[\![\!_2\ A] \rangle]) \in \mathcal{R}))$

definition

strong-low-bisim-mm :: $((Com, Var, Val)\ LocalConf)\ rel \implies bool$
where
strong-low-bisim-mm $\mathcal{R} \equiv$
 $sym\ \mathcal{R} \wedge$
closed-glob-consistent $\mathcal{R} \wedge$
 $(\forall c_1\ mds\ mem_1\ c_2\ mem_2. (\langle c_1, mds, mem_1 \rangle, \langle c_2, mds, mem_2 \rangle) \in \mathcal{R} \implies$

$$\begin{aligned}
& (mem_1 =_{m_{ds}^l} mem_2) \wedge \\
& (\forall c_1' m_{ds}' mem_1'. \langle c_1, m_{ds}, mem_1 \rangle \rightsquigarrow \langle c_1', m_{ds}', mem_1' \rangle \longrightarrow \\
& (\exists c_2' mem_2'. \langle c_2, m_{ds}, mem_2 \rangle \rightsquigarrow \langle c_2', m_{ds}', mem_2' \rangle \wedge \\
& (\langle c_1', m_{ds}', mem_1' \rangle, \langle c_2', m_{ds}', mem_2' \rangle) \in \mathcal{R}))
\end{aligned}$$

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

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

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

where

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

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

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

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

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

where

$c_1 \sim_{m_{ds}} c_2 = (\forall mem_1 mem_2. mem_1 =_{m_{ds}^l} mem_2 \longrightarrow \langle c_1, m_{ds}, mem_1 \rangle \approx \langle c_2, m_{ds}, mem_2 \rangle)$

2.3 SIFUM-Security

definition

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

where

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

definition

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

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

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

$(\forall sched\ cms_1' mem_1'.$

$(cmds, mem_1) \rightarrow_{sched} (cms_1', mem_1') \longrightarrow$

$(\exists cms_2' mem_2'. (cmds, mem_2) \rightarrow_{sched} (cms_2', mem_2') \wedge$

$map\ snd\ cms_1' = map\ snd\ cms_2' \wedge$

$length\ cms_2' = length\ cms_1' \wedge$

$(\forall x. dma\ mem_1' x = Low \wedge (x \in \mathcal{C} \vee (\forall i < length\ cms_1'.$

$x \notin snd\ (cms_1' ! i)\ AsmNoReadOrWrite)) \longrightarrow mem_1' x =$

$mem_2' x))))$

lemma $prog\text{-sifum-secure-cont-def2}:$

$prog\text{-sifum-secure-cont } cmds \equiv$

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

$(\forall sched\ cms_1' mem_1'.$

$(cmds, mem_1) \rightarrow_{sched} (cms_1', mem_1') \longrightarrow$

$(\exists cms_2' mem_2'. (cmds, mem_2) \rightarrow_{sched} (cms_2', mem_2') \wedge$

$$\begin{aligned}
& (\forall \text{ cms}_2' \text{ mem}_2'. (\text{cmds}, \text{mem}_2) \rightarrow_{\text{sched}} (\text{cms}_2', \text{mem}_2') \longrightarrow \\
& \quad \text{map snd cms}_1' = \text{map snd cms}_2' \wedge \\
& \quad \text{length cms}_2' = \text{length cms}_1' \wedge \\
& \quad (\forall x. \text{dma mem}_1' x = \text{Low} \wedge (x \in \mathcal{C} \vee (\forall i < \text{length cms}_1'. \\
& \quad x \notin \text{snd} (\text{cms}_1' ! i) \text{AsmNoReadOrWrite})) \longrightarrow \text{mem}_1' x = \\
& \text{mem}_2' x))) \\
& \langle \text{proof} \rangle
\end{aligned}$$

2.4 Sound Mode Use

definition

$$\text{subst} :: ('a \rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$$

where

$$\begin{aligned}
\text{subst } f \text{ mem} &= (\lambda x. \text{case } f \text{ } x \text{ of} \\
& \quad \text{None} \Rightarrow \text{mem } x \mid \\
& \quad \text{Some } v \Rightarrow v)
\end{aligned}$$

abbreviation

$$\text{subst-abv} :: ('a \Rightarrow 'b) \Rightarrow ('a \rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \text{ (- } [\mapsto] [900, 0] 1000)$$

where

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

lemma *subst-not-in-dom* : $\llbracket x \notin \text{dom } \sigma \rrbracket \Longrightarrow \text{mem} [\mapsto \sigma] x = \text{mem } x$
 $\langle \text{proof} \rangle$

definition

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

where

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

definition

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

where

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

lemma *vars-C-subset-C*:

$$\text{vars-C } X \subseteq \mathcal{C}$$

$$\langle \text{proof} \rangle$$

definition

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

where

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

definition

doesn't-modify :: 'Com \Rightarrow 'Var \Rightarrow bool

where

doesn't-modify $c\ x = (\forall\ mds\ mem\ c'\ mds'\ mem'. (\langle c, mds, mem \rangle \rightsquigarrow \langle c', mds', mem' \rangle) \longrightarrow mem\ x = mem'\ x \wedge dma\ mem\ x = dma\ mem'\ x)$

lemma *noread-nowrite*:

assumes *step*: $\langle c, mds, mem \rangle \rightsquigarrow \langle c', mds', mem' \rangle$

assumes *noread*: $(\bigwedge v. \langle c, mds, mem(x := v) \rangle \rightsquigarrow \langle c', mds', mem'(x := v) \rangle)$

shows $mem\ x = mem'\ x$

$\langle proof \rangle$

lemma *doesn't-read-or-modify-doesn't-modify*:

doesn't-read-or-modify $c\ x \implies$ *doesn't-modify* $c\ x$

$\langle proof \rangle$

inductive-set

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

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

where

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

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

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

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

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

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

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

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

lemma *neval-loc-reach*:

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

$\langle proof \rangle$

definition

locally-sound-mode-use :: (-, -, -) LocalConf \Rightarrow bool

where

locally-sound-mode-use $lc =$

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

$(\forall\ x. (x \in mds'\ GuarNoReadOrWrite \longrightarrow doesn't\text{-}read\text{-}or\text{-}modify\ c'\ x) \wedge$

$(x \in mds'\ GuarNoWrite \longrightarrow doesn't\text{-}modify\ c'\ x)))$

definition

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

where

respects-own-guarantees $cm \equiv$

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

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

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

lemma *locally-sound-respects-guarantees:*

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

definition

$\text{compatible-modes} :: ('Var \text{ Mds}) \text{ list} \Rightarrow \text{ bool}$

where

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

definition

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

where

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

definition

$\text{globally-sound-mode-use} :: ('Com, 'Var, 'Val) \text{ GlobalConf} \Rightarrow \text{ bool}$

where

$\text{globally-sound-mode-use } gc \equiv$
 $(\forall mdss. mdss \in \text{reachable-mode-states } gc \longrightarrow \text{compatible-modes } mdss)$

primrec

$\text{sound-mode-use} :: (-, -, -) \text{ GlobalConf} \Rightarrow \text{ bool}$

where

$\text{sound-mode-use } (cms, \text{ mem}) =$
 $(\text{list-all } (\lambda cm. \text{locally-sound-mode-use } (cm, \text{ mem})) cms \wedge$
 $\text{globally-sound-mode-use } (cms, \text{ mem}))$

lemma *mm-equiv-sym:*

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

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *mm-equiv-glob-consistent: closed-glob-consistent mm-equiv*
⟨proof⟩

lemma *mm-equiv-strong-low-bisim: strong-low-bisim-mm mm-equiv*
⟨proof⟩

end

end

3 Compositionality Proof for SIFUM-Security Property

theory *Compositionality*
imports *Security*
begin

context *sifum-security-init*
begin

definition

differing-vars :: ('Var, 'Val) Mem ⇒ (-, -) Mem ⇒ 'Var set

where

differing-vars mem₁ mem₂ ≡ {x. mem₁ x ≠ mem₂ x}

definition

differing-vars-lists :: ('Var, 'Val) Mem ⇒ (-, -) Mem ⇒
((-, -) Mem × (-, -) Mem) list ⇒ nat ⇒ 'Var set

where

differing-vars-lists mem₁ mem₂ mems i ≡
(*differing-vars mem₁ (fst (mems ! i))*) ∪ *differing-vars mem₂ (snd (mems ! i))*)

lemma *differing-finite: finite (differing-vars mem₁ mem₂)*
⟨proof⟩

lemma *differing-lists-finite: finite (differing-vars-lists mem₁ mem₂ mems i)*
⟨proof⟩

fun *makes-compatible* ::

('Com, 'Var, 'Val) GlobalConf ⇒
('Com, 'Var, 'Val) GlobalConf ⇒
((-, -) Mem × (-, -) Mem) list ⇒
bool

where

makes-compatible (cms₁, mem₁) (cms₂, mem₂) mems =

$$\begin{aligned}
& (\text{length } cms_1 = \text{length } cms_2 \wedge \text{length } cms_1 = \text{length } mems \wedge \\
& (\forall i. i < \text{length } cms_1 \longrightarrow \\
& \quad (\forall \sigma. \text{dom } \sigma = \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i \longrightarrow \\
& \quad \quad (cms_1 ! i, (\text{fst } (mems ! i)) [\mapsto \sigma]) \approx (cms_2 ! i, (\text{snd } (mems ! i)) [\mapsto \sigma])) \wedge \\
& \quad (\forall x. (\text{mem}_1 \ x = \text{mem}_2 \ x \vee \text{dma } \text{mem}_1 \ x = \text{High} \vee x \in \mathcal{C}) \longrightarrow \\
& \quad \quad x \notin \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i)) \wedge \\
& ((\text{length } cms_1 = 0 \wedge \text{mem}_1 =^l \text{mem}_2) \vee (\forall x. \exists i. i < \text{length } cms_1 \wedge \\
& \quad x \notin \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i)))
\end{aligned}$$

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

$$\begin{aligned}
& \llbracket \text{length } cms_1 = \text{length } cms_2 \wedge \text{length } cms_1 = \text{length } mems; \\
& \quad (\wedge i \ \sigma. \llbracket i < \text{length } cms_1; \text{dom } \sigma = \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i \rrbracket \\
\implies & \quad (cms_1 ! i, (\text{fst } (mems ! i)) [\mapsto \sigma]) \approx (cms_2 ! i, (\text{snd } (mems ! i)) [\mapsto \sigma]); \\
& \quad (\wedge i \ x. \llbracket i < \text{length } cms_1; \text{mem}_1 \ x = \text{mem}_2 \ x \vee \text{dma } \text{mem}_1 \ x = \text{High} \vee x \in \\
\mathcal{C} \rrbracket \implies & \quad x \notin \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i); \\
& \quad (\text{length } cms_1 = 0 \wedge \text{mem}_1 =^l \text{mem}_2) \vee \\
& \quad (\forall x. \exists i. i < \text{length } cms_1 \wedge x \notin \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i) \rrbracket \\
\implies & \quad \text{makes-compatible } (cms_1, \text{mem}_1) \ (cms_2, \text{mem}_2) \ mems \\
& \quad \langle \text{proof} \rangle
\end{aligned}$$

lemma *compat-low*:

$$\begin{aligned}
& \llbracket \text{makes-compatible } (cms_1, \text{mem}_1) \ (cms_2, \text{mem}_2) \ mems; \\
& \quad i < \text{length } cms_1; \\
& \quad x \in \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i \rrbracket \implies \text{dma } \text{mem}_1 \ x = \text{Low} \\
& \quad \langle \text{proof} \rangle
\end{aligned}$$

lemma *compat-different*:

$$\begin{aligned}
& \llbracket \text{makes-compatible } (cms_1, \text{mem}_1) \ (cms_2, \text{mem}_2) \ mems; \\
& \quad i < \text{length } cms_1; \\
& \quad x \in \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i \rrbracket \implies \text{mem}_1 \ x \neq \text{mem}_2 \ x \wedge \text{dma} \\
& \text{mem}_1 \ x = \text{Low} \wedge x \notin \mathcal{C} \\
& \quad \langle \text{proof} \rangle
\end{aligned}$$

lemma *sound-modes-no-read* :

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

lemma *differing-vars-neg*: $x \notin \text{differing-vars-lists } mem_1 \ mem_2 \ mems \ i \implies$

$$\begin{aligned}
& (\text{fst } (mems ! i) \ x = \text{mem}_1 \ x \wedge \text{snd } (mems ! i) \ x = \text{mem}_2 \ x) \\
& \quad \langle \text{proof} \rangle
\end{aligned}$$

lemma *differing-vars-neg-intro*:

where

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

fun *change-respecting* ::

$$\begin{aligned} &('Com, 'Var, 'Val) LocalConf \Rightarrow \\ &('Com, 'Var, 'Val) LocalConf \Rightarrow \\ &'Var set \Rightarrow bool \end{aligned}$$

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

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

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

<proof>

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

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

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

<proof>

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

assumes *eval*: $(\text{cms}, \text{mem}) \rightsquigarrow (\text{cms}', \text{mem}')$

assumes *cr*: $\forall f. \text{dom } f = Y \longrightarrow (\text{cms}, \text{mem} [\mapsto f]) \rightsquigarrow (\text{cms}', \text{mem}' [\mapsto f])$

assumes *x-in-dom*: $x \in Y$

shows $\text{mem } x = \text{mem}' x$

<proof>

lemma *change-respecting-subset'*:

assumes *step*: $(\text{cms}, \text{mem}) \rightsquigarrow (\text{cms}', \text{mem}')$

assumes *noread*: $(\forall \sigma. \text{dom } \sigma = X \longrightarrow (\text{cms}, \text{mem} [\mapsto \sigma]) \rightsquigarrow (\text{cms}', \text{mem}' [\mapsto \sigma]))$

assumes *dom-subset*: $\text{dom } \sigma \subseteq X$

shows $(\text{cms}, \text{mem} [\mapsto \sigma]) \rightsquigarrow (\text{cms}', \text{mem}' [\mapsto \sigma])$

<proof>

lemma *change-respecting-subst*:

change-respecting (*cms*, *mem*) (*cms'*, *mem'*) *X* \implies

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

<proof>

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

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

$\bigwedge f. \text{dom } f = \text{vars-and-}\mathcal{C} X \implies$

$(\langle c, \text{mds}, \text{mem} [\mapsto f] \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' [\mapsto f] \rangle) \rrbracket$

$\implies \text{change-respecting } \langle c, \text{mds}, \text{mem} \rangle \langle c', \text{mds}', \text{mem}' \rangle X$

<proof>

lemma *vars-C-mono*:

$$X \subseteq Y \implies \text{vars-C } X \subseteq \text{vars-C } Y$$

<proof>

lemma *vars-C-Un*:

$$\text{vars-C } (X \cup Y) = (\text{vars-C } X \cup \text{vars-C } Y)$$

<proof>

lemma *vars-C-insert*:

$$\text{vars-C } (\text{insert } x \ Y) = (\text{vars-C } \{x\}) \cup (\text{vars-C } Y)$$

<proof>

lemma *vars-C-empty[simp]*:

$$\text{vars-C } \{\} = \{\}$$

<proof>

lemma *C-vars-of-C-vars-empty*:

$$x \in \text{C-vars } y \implies \text{C-vars } x = \{\}$$

<proof>

lemma *vars-and-C-mono*:

$$X \subseteq X' \implies \text{vars-and-C } X \subseteq \text{vars-and-C } X'$$

<proof>

lemma *C-vars-finite[simp]*:

$$\text{finite } (\text{C-vars } x)$$

<proof>

lemma *finite-dom*:

$$\text{finite } (\text{dom } (\sigma :: 'Var \Rightarrow 'Val \text{ option}))$$

<proof>

lemma *doesnt-read-or-modify-subst*:

assumes *noread*: *doesnt-read-or-modify* $c \ x$

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

assumes *subset*: $X \subseteq \{x\} \cup \text{C-vars } x$

shows $\bigwedge \sigma. \text{dom } \sigma = X \implies \langle c, \text{mds}, \text{mem}[\mapsto \sigma] \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}'[\mapsto \sigma] \rangle$

<proof>

lemma *subst-restrict-twice*:

$$\text{dom } \sigma = A \cup B \implies$$

$$\text{mem } [\mapsto (\sigma \mid' A)] [\mapsto (\sigma \mid' B)] = \text{mem } [\mapsto \sigma]$$

<proof>

lemma *noread-exists-change-respecting*:

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

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

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

shows *change-respecting* $\langle c, mds, mem \rangle \langle c', mds', mem' \rangle X$
 $\langle proof \rangle$

lemma *update-nth-eq*:

$\llbracket xs = ys; n < length\ xs \rrbracket \implies xs = ys [n := xs ! n]$
 $\langle proof \rangle$

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

lemma *mm-equiv-step*:

assumes *bisim*: $(cms_1, mem_1) \approx (cms_2, mem_2)$
assumes *modes-eq*: $snd\ cms_1 = snd\ cms_2$
assumes *step*: $(cms_1, mem_1) \rightsquigarrow (cms_1', mem_1')$
shows $\exists c_2' mem_2'. (cms_2, mem_2) \rightsquigarrow \langle c_2', snd\ cms_1', mem_2' \rangle \wedge$
 $(cms_1', mem_1') \approx \langle c_2', snd\ cms_1', mem_2' \rangle$
 $\langle proof \rangle$

lemma *change-respecting-doesnt-modify*:

assumes *cr*: *change-respecting* $(cms, mem) (cms', mem') X$
assumes *eval*: $(cms, mem) \rightsquigarrow (cms', mem')$
assumes *x-in-dom*: $x \in X \cup vars\mathcal{C}\ X$
shows $mem\ x = mem'\ x$
 $\langle proof \rangle$

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

assumes *cr*: *change-respecting* $(cms, mem) (cms', mem') X$
assumes *eval*: $(cms, mem) \rightsquigarrow (cms', mem')$
assumes *x-in-dom*: $x \in X$
shows $dma\ mem\ x = dma\ mem'\ x$
 $\langle proof \rangle$

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

lemma *differing-empty-eq*:

$\llbracket differing\ vars\ mem\ mem' = \{\} \rrbracket \implies mem = mem'$
 $\langle proof \rangle$

lemma *adaptation-finite*:

finite $(dom\ (A::('Var, 'Val)\ adaptation))$
 $\langle proof \rangle$

definition

globally-consistent :: $('Var, 'Val)\ adaptation \Rightarrow 'Var\ Mds \Rightarrow ('Var, 'Val)\ Mem$
 $\Rightarrow ('Var, 'Val)\ Mem \Rightarrow bool$

where *globally-consistent* $A\ mds\ mem_1\ mem_2 \equiv$

$(\forall x. case\ A\ x\ of\ Some\ (v, v') \Rightarrow (mem_1\ x \neq v \vee mem_2\ x \neq v') \longrightarrow \neg\ var\ asm\ not\ written$
 $mds\ x\ | - \Rightarrow True) \wedge$

$(\forall x. dma\ mem_1\ [\llbracket_1\ A\rrbracket]\ x \neq dma\ mem_1\ x \longrightarrow \neg\ var\text{-asm}\text{-not}\text{-written}\ mds\ x)$
 \wedge
 $(\forall x. dma\ (mem_1\ [\llbracket_1\ A\rrbracket])\ x = Low \wedge (x \notin mds\ AsmNoReadOrWrite \vee x \in \mathcal{C}) \longrightarrow (mem_1\ [\llbracket_1\ A\rrbracket])\ x = (mem_2\ [\llbracket_2\ A\rrbracket])\ x)$

lemma *globally-consistent-adapt-bisim*:

assumes *bisim*: $\langle c_1, mds, mem_1 \rangle \approx \langle c_2, mds, mem_2 \rangle$
assumes *globally-consistent*: *globally-consistent* $A\ mds\ mem_1\ mem_2$
shows $\langle c_1, mds, mem_1\ [\llbracket_1\ A\rrbracket] \rangle \approx \langle c_2, mds, mem_2\ [\llbracket_2\ A\rrbracket] \rangle$
 $\langle proof \rangle$

lemma *mm-equiv-C-eq*:

$(a, b) \approx (a', b') \implies snd\ a = snd\ a' \implies$
 $\forall x \in \mathcal{C}. b\ x = b'\ x$
 $\langle proof \rangle$

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

$x \notin dom\ A \implies apply\text{-adaptation}\ b\ blah\ A\ x = blah\ x$
 $\langle proof \rangle$

lemma *makes-compatible-invariant*:

assumes *sound-modes*: *sound-mode-use* (cms_1, mem_1)
 $sound\text{-mode}\text{-use}\ (cms_2, mem_2)$
assumes *compat*: *makes-compatible* $(cms_1, mem_1)\ (cms_2, mem_2)\ mems$
assumes *modes-eq*: $map\ snd\ cms_1 = map\ snd\ cms_2$
assumes *eval*: $(cms_1, mem_1) \rightsquigarrow_k (cms_1', mem_1')$
obtains $cms_2'\ mem_2'\ mems'$ **where**
 $map\ snd\ cms_1' = map\ snd\ cms_2' \wedge$
 $(cms_2, mem_2) \rightsquigarrow_k (cms_2', mem_2') \wedge$
 $makes\text{-compatible}\ (cms_1', mem_1')\ (cms_2', mem_2')\ mems'$
 $\langle proof \rangle$

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* $(cms_1, mem_1)\ (cms_2, mem_2)\ mems$
assumes *modes-eq*: $map\ snd\ cms_1 = map\ snd\ cms_2$
assumes *x-low*: $dma\ mem_1\ x = Low$
assumes *x-readable*: $x \in \mathcal{C} \vee (\forall\ i < length\ cms_1. x \notin snd\ (cms_1\ !\ i)\ AsmNoReadOrWrite)$
shows $mem_1\ x = mem_2\ x$
 $\langle proof \rangle$

lemma *loc-reach-subset*:

assumes *eval*: $\langle c, mds, mem \rangle \rightsquigarrow \langle c', mds', mem' \rangle$
shows $loc\text{-reach}\ \langle c', mds', mem' \rangle \subseteq loc\text{-reach}\ \langle c, mds, mem \rangle$
 $\langle proof \rangle$

lemma *locally-sound-modes-invariant*:

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

assumes *eval*: $\langle c, mds, mem \rangle \rightsquigarrow_k \langle c', mds', mem' \rangle$

shows *locally-sound-mode-use* $\langle c', mds', mem' \rangle$

$\langle proof \rangle$

lemma *meval-sched-one*:

$(cms, mem) \rightsquigarrow_k (cms', mem') \implies$

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

$\langle proof \rangle$

lemma *meval-sched-ConsI*:

$(cms, mem) \rightsquigarrow_k (cms', mem') \implies$

$(cms', mem') \rightarrow_{\text{sched}} (cms'', mem'') \implies$

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

$\langle proof \rangle$

lemma *reachable-modes-subset*:

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

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

$\langle proof \rangle$

lemma *globally-sound-modes-invariant*:

assumes *globally-sound: globally-sound-mode-use* (cms, mem)

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

shows *globally-sound-mode-use* (cms', mem')

$\langle proof \rangle$

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

assumes *mem-diff*: $\forall x. \text{var-asm-not-written } mds\ x \longrightarrow \text{mem}_1\ x = \text{mem}_2\ x \wedge$
 $\text{dma } \text{mem}_1\ x = \text{dma } \text{mem}_2\ x$

shows $\langle c', mds', mem' \rangle \in \text{loc-reach } \langle c, mds, mem_1 \rangle \implies \langle c', mds', mem' \rangle \in$
 $\text{loc-reach } \langle c, mds, mem_2 \rangle$

$\langle proof \rangle$

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

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

shows $\text{loc-reach } \langle c, mds, mem \rangle = \text{loc-reach } \langle c, mds, mem' \rangle$

$\langle proof \rangle$

lemma *sound-modes-invariant*:

assumes *sound-modes: sound-mode-use* (cms, mem)

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

shows *sound-mode-use* (cms', mem')

$\langle proof \rangle$

lemma *app-Cons-rewrite*:

$ns @ (a \# ms) = ((ns @ [a]) @ ms)$
 ⟨proof⟩

lemma *meval-sched-app-iff*:

$(cms_1, mem_1) \rightarrow_{ns@ms} (cms_1', mem_1') =$
 $(\exists cms_1'' mem_1''. (cms_1, mem_1) \rightarrow_{ns} (cms_1'', mem_1'') \wedge (cms_1'', mem_1'') \rightarrow_{ms}$
 $(cms_1', mem_1'))$
 ⟨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*:

$(cms_1, mem_1) \rightarrow_{ns@[n]} (cms_1', mem_1') \implies$
 $\exists cms_1'' mem_1''. (cms_1, mem_1) \rightarrow_{ns} (cms_1'', mem_1'') \wedge (cms_1'', mem_1'') \rightsquigarrow_n$
 (cms_1', mem_1')
 ⟨proof⟩

lemma *meval-sched-snocI*:

$(cms_1, mem_1) \rightarrow_{ns} (cms_1'', mem_1'') \wedge (cms_1'', mem_1'') \rightsquigarrow_n (cms_1', mem_1') \implies$
 $(cms_1, mem_1) \rightarrow_{ns@[n]} (cms_1', mem_1')$
 ⟨proof⟩

lemma *makes-compatible-eval-sched*:

assumes *compat*: *makes-compatible* (cms_1, mem_1) (cms_2, mem_2) *mems*

assumes *modes-eq*: *map snd* $cms_1 = \text{map snd } cms_2$

assumes *sound-modes*: *sound-mode-use* (cms_1, mem_1) *sound-mode-use* (cms_2, mem_2)

assumes *eval*: $(cms_1, mem_1) \rightarrow_{sched} (cms_1', mem_1')$

shows $\exists cms_2' mem_2' mems'. \text{sound-mode-use } (cms_1', mem_1') \wedge$

$\text{sound-mode-use } (cms_2', mem_2') \wedge$

$\text{map snd } cms_1' = \text{map snd } cms_2' \wedge$

$(cms_2, mem_2) \rightarrow_{sched} (cms_2', mem_2') \wedge$

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

⟨proof⟩

lemma *differing-vars-initially-empty*:

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

⟨proof⟩

lemma *compatible-refl*:

assumes *coms-secure*: *list-all com-sifum-secure* *cmds*

assumes *low-eq*: $mem_1 =^l mem_2$

shows *makes-compatible* $(cmds, mem_1)$

$(cmds, mem_2)$

$(\text{replicate } (\text{length } cmds) (mem_1, mem_2))$

⟨proof⟩

```

theorem sifum-compositionality-cont:
  assumes com-secure: list-all com-sifum-secure cmds
  assumes sound-modes:  $\forall$  mem. INIT mem  $\longrightarrow$  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 'var ModeUpd = Acq 'var Mode (infix +=m 75)
  | Rel 'var Mode (infix -=m 75)

```

```

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

```

```

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

```

```

locale sifum-lang-no-dma =
  fixes evalA :: ('Var, 'Val) Mem  $\Rightarrow$  'AExp  $\Rightarrow$  'Val
  fixes evalB :: ('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-detA :  $\llbracket \forall x \in \text{aexp-vars } e. \text{mem}_1 x = \text{mem}_2 x \rrbracket \Longrightarrow \text{eval}_A$ 
    mem1 e = evalA mem2 e
  assumes eval-vars-detB :  $\llbracket \forall x \in \text{bexp-vars } b. \text{mem}_1 x = \text{mem}_2 x \rrbracket \Longrightarrow \text{eval}_B$ 
    mem1 b = evalB mem2 b

```

```

locale sifum-lang = sifum-lang-no-dma evalA evalB aexp-vars bexp-vars
  for evalA :: ('Var, 'Val) Mem  $\Rightarrow$  'AExp  $\Rightarrow$  'Val
  and evalB :: ('Var, 'Val) Mem  $\Rightarrow$  'BExp  $\Rightarrow$  bool
  and aexp-vars :: 'AExp  $\Rightarrow$  'Var set

```

```

and bexp-vars :: 'BExp ⇒ 'Var set+
fixes dma :: 'Var ⇒ Sec

context sifum-lang-no-dma
begin

notation (latex output)
  Seq (-; - 60)

notation (Rule output)
  Seq (- ; - 60)

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

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

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

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

4.2 Semantics

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

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

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

inductive-set evalw-simple :: (('Var, 'AExp, 'BExp) Stmt × ('Var, 'Val) Mem)
  rel
and evalw-simple-abv :: (('Var, 'AExp, 'BExp) Stmt × ('Var, 'Val) Mem) ⇒

```

$(\text{'Var}, \text{'AExp}, \text{'BExp}) \text{ Stmt} \times (\text{'Var}, \text{'Val}) \text{ Mem} \Rightarrow \text{bool}$
(infixr \rightsquigarrow_s 60)
where
 $c \rightsquigarrow_s c' \equiv (c, c') \in \text{eval}_w\text{-simple} \mid$
 $\text{assign}: ((x \leftarrow e, \text{mem}), (\text{Stop}, \text{mem} (x := \text{eval}_A \text{ mem } e))) \in \text{eval}_w\text{-simple} \mid$
 $\text{skip}: ((\text{Skip}, \text{mem}), (\text{Stop}, \text{mem})) \in \text{eval}_w\text{-simple} \mid$
 $\text{seq-stop}: ((\text{Seq Stop } c, \text{mem}), (c, \text{mem})) \in \text{eval}_w\text{-simple} \mid$
 $\text{if-true}: [\text{eval}_B \text{ mem } b] \Longrightarrow ((\text{If } b \text{ t } e, \text{mem}), (t, \text{mem})) \in \text{eval}_w\text{-simple} \mid$
 $\text{if-false}: [\neg \text{eval}_B \text{ mem } b] \Longrightarrow ((\text{If } b \text{ t } e, \text{mem}), (e, \text{mem})) \in \text{eval}_w\text{-simple} \mid$
 $\text{while}: ((\text{While } b \text{ c}, \text{mem}), (\text{If } b (c ;; \text{While } b \text{ c}) \text{ Stop}, \text{mem})) \in \text{eval}_w\text{-simple}$

lemma cond:

$((\text{If } b \text{ t } e, \text{mem}), (\text{if eval}_B \text{ mem } b \text{ then } t \text{ else } e, \text{mem})) \in \text{eval}_w\text{-simple}$
 $\langle \text{proof} \rangle$

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

$\Rightarrow (\text{'Var}, \text{'AExp}, \text{'BExp}) \text{ Stmt}$

where

$\text{cxt-to-stmt} [] \ c = c \mid$

$\text{cxt-to-stmt} (c \# \text{cs}) \ c' = \text{Seq } c' (\text{cxt-to-stmt } \text{cs } c)$

lemma rtrancl-mono-proof[mono]:

$(\bigwedge a \ b. \ x \ a \ b \longrightarrow y \ a \ b) \Longrightarrow \text{rtranclp } x \ a \ b \longrightarrow \text{rtranclp } y \ a \ b$
 $\langle \text{proof} \rangle$

lemma trancl-mono-proof[mono]:

$(\bigwedge a \ b. \ x \ a \ b \longrightarrow y \ a \ b) \Longrightarrow \text{tranclp } x \ a \ b \longrightarrow \text{tranclp } y \ a \ b$
 $\langle \text{proof} \rangle$

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

$\text{no-await } (x \leftarrow e) \mid$

$\text{no-await } c1 \Longrightarrow \text{no-await } c2 \Longrightarrow \text{no-await } (c1 ;; c2) \mid$

$\text{no-await } c1 \Longrightarrow \text{no-await } c2 \Longrightarrow \text{no-await } (\text{If } b \ c1 \ c2) \mid$

$\text{no-await } c \Longrightarrow \text{no-await } (\text{While } b \ c) \mid$

$\text{no-await } \text{Skip} \mid$

$\text{no-await } \text{Stop} \mid$

$\text{no-await } c \Longrightarrow \text{no-await } (c@[m])$

inductive is-final :: $(\text{'Var}, \text{'AExp}, \text{'BExp}) \text{ Stmt} \Rightarrow \text{bool}$ **where**

$\text{is-final } \text{Stop} \mid$

$\text{is-final } c \Longrightarrow \text{is-final } (c@[m])$

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

and eval_w-abv :: $(\text{'Var}, \text{'AExp}, \text{'BExp}) \text{ Stmt}, \text{'Var}, \text{'Val}) \text{ LocalConf} \Rightarrow$

$(\text{'Var}, \text{'AExp}, \text{'BExp}) \text{ Stmt}, \text{'Var}, \text{'Val}) \text{ LocalConf} \Rightarrow \text{bool}$

(**infixr** \rightsquigarrow_w 60)

where

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

await: $\llbracket eval_B \ mem \ b; no\text{-await } c_1;$
 $\langle c_1, mds, mem \rangle_w, \langle c_2, mds', mem' \rangle_w \in eval_w^+;$
is-final $c_2 \rrbracket \implies$
 $\langle Await \ b \ c_1, mds, mem \rangle_w, \langle c_2, mds', mem' \rangle_w \in eval_w$

abbreviation *eval_w-plus* ::

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

-) **where**

$ctx \rightsquigarrow_w^+ ctx' \equiv (ctx, ctx') \in eval_w^+$

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

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

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

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

lemma *ctx-inv*:

$\llbracket cxt\text{-to}\text{-stmt } E \ c = c' ; \wedge \ p \ q. \ c' \neq Seq \ p \ q \rrbracket \implies E = [] \wedge c' = c$
 $\langle proof \rangle$

lemma *ctx-inv-assign*:

$\llbracket cxt\text{-to}\text{-stmt } E \ c = x \leftarrow e \rrbracket \implies c = x \leftarrow e \wedge E = []$
 $\langle proof \rangle$

lemma *ctx-inv-skip*:

$\llbracket cxt\text{-to}\text{-stmt } E \ c = Skip \rrbracket \implies c = Skip \wedge E = []$
 $\langle proof \rangle$

lemma *ctx-inv-stop*:

$cxt\text{-to}\text{-stmt } E \ c = Stop \implies c = Stop \wedge E = []$
 $\langle proof \rangle$

lemma *ctx-inv-if*:

$cxt\text{-to}\text{-stmt } E \ c = If \ e \ p \ q \implies c = If \ e \ p \ q \wedge E = []$

$\langle \text{proof} \rangle$

lemma *ctx-inv-anno*:

$$\text{ctx-to-stmt } E \ c = c'@[mu] \Longrightarrow c = c'@[mu] \wedge E = []$$

$\langle \text{proof} \rangle$

lemma *ctx-inv-await*:

$$\text{ctx-to-stmt } E \ c = \text{Await } e \ p \Longrightarrow c = \text{Await } e \ p \wedge E = []$$

$\langle \text{proof} \rangle$

lemma *ctx-inv-while*:

$$\text{ctx-to-stmt } E \ c = \text{While } e \ p \Longrightarrow c = \text{While } e \ p \wedge E = []$$

$\langle \text{proof} \rangle$

lemma *skip-elim* [elim]:

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

$\langle \text{proof} \rangle$

lemma *assign-elim* [elim]:

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

$\langle \text{proof} \rangle$

inductive-cases *if-elim'* [elim!]: $\langle \text{If } b \ p \ q, \text{mem} \rangle \rightsquigarrow_s \langle c', \text{mem}' \rangle$

lemma *if-elim* [elim]:

$\bigwedge P.$

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

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

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

$\langle \text{proof} \rangle$

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

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

lemma *while-elim* [elim]:

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

$\langle \text{proof} \rangle$

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

lemma *upd-elim* [elim]:

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

$\langle \text{proof} \rangle$

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

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

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

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

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

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

lemma *cxt-stmt-seq*:

$c ;; \text{cxt-to-stmt } E \ c' = \text{cxt-to-stmt } (c' \# E) \ c$
 ⟨proof⟩

lemma *seq-elim* [elim]:

$\llbracket \langle c_1 ;; c_2, \text{mds}, \text{mem} \rangle_w \rightsquigarrow_w \langle c', \text{mds}', \text{mem}' \rangle_w ; c_1 \neq \text{Stop} \rrbracket \implies (\exists \ c_1'. \langle c_1, \text{mds}, \text{mem} \rangle_w \rightsquigarrow_w \langle c_1', \text{mds}', \text{mem}' \rangle_w \wedge c' = c_1' ;; c_2)$
 ⟨proof⟩

lemma *stop-cxt*: $\text{Stop} = \text{cxt-to-stmt } E \ c \implies c = \text{Stop}$

⟨proof⟩

lemmas *decl-eval_w* = *decl*[*OF unannotated*, *OF skip*, **where** $E=[]$, *simplified*, **where** $E1=[]$, *simplified*]

lemmas *seq-stop-eval_w* = *unannotated*[*OF seq-stop*, **where** $E=[]$, *simplified*]

lemmas *assign-eval_w* = *unannotated*[*OF assign*, **where** $E=[]$, *simplified*]

lemmas *if-eval_w* = *unannotated*[*OF cond*, **where** $E=[]$, *simplified*]

lemmas *if-false-eval_w* = *unannotated*[*OF if-false*, **where** $E=[]$, *simplified*]

lemmas *skip-eval_w* = *unannotated*[*OF skip*, **where** $E=[]$, *simplified*]

lemmas *while-eval_w* = *unannotated*[*OF while*, **where** $E=[]$, *simplified*]

lemma *decl-eval_w'*:

assumes *mem-unchanged*: $\text{mem}' = \text{mem}$

assumes *upd*: $\text{mds}' = \text{update-modes } \text{upd} \ \text{mds}$

shows $(\langle \text{Skip}@[\text{upd}], \text{mds}, \text{mem} \rangle_w, \langle \text{Stop}, \text{mds}', \text{mem}' \rangle_w) \in \text{eval}_w$

⟨proof⟩

lemma *assign-eval_w'*:

$\llbracket \text{mds} = \text{mds}'; \text{mem}' = \text{mem}(x := \text{eval}_A \ \text{mem} \ e) \rrbracket \implies$

$\langle x \leftarrow e, \text{mds}, \text{mem} \rangle_w \rightsquigarrow_w \langle \text{Stop}, \text{mds}', \text{mem}' \rangle_w$

⟨proof⟩

lemma *seq-decl-elim*:

$\langle \langle \text{Skip}@[\text{upd}] \rangle \rangle ; c, \text{mds}, \text{mem} \rangle_w \rightsquigarrow_w \langle c', \text{mds}', \text{mem}' \rangle_w \implies$
 $c' = \text{Stop} ; c \wedge \text{mem}' = \text{mem} \wedge \text{mds}' = \text{update-modes upd mds}$
<proof>

lemma *seq-assign-elim*:

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

lemma *no-await-trans*:

$\llbracket \text{no-await } c ; \langle c, \text{mds}, \text{mem} \rangle_w \rightsquigarrow_w \langle c', \text{mds}', \text{mem}' \rangle_w \rrbracket \implies \text{no-await } c'$
<proof>

lemma *no-await-no-await[elim]*: $\llbracket \text{no-await } c \rrbracket \implies c \neq \text{Await } b \ c'$

<proof>

lemma *no-await-trancl-impl*:

$\llbracket \text{ctx} \rightsquigarrow_w^+ \text{ctx}' \rrbracket \implies \text{no-await } (\text{fst } (\text{fst } \text{ctx})) \longrightarrow \text{no-await } (\text{fst } (\text{fst } \text{ctx}'))$
<proof>

lemma *no-await-trancl*:

$\llbracket \text{ctx} \rightsquigarrow_w^+ \text{ctx}'; \text{no-await } (\text{fst } (\text{fst } \text{ctx})) \rrbracket \implies \text{no-await } (\text{fst } (\text{fst } \text{ctx}'))$
<proof>

lemma *await-elim*:

$\llbracket \langle \text{Await } b \ c_1, \text{mds}, \text{mem} \rangle_w \rightsquigarrow_w \langle c_2, \text{mds}', \text{mem}' \rangle_w \rrbracket \implies$
 $\text{eval}_B \text{ mem } b \wedge \text{no-await } c_1 \wedge \text{is-final } c_2 \wedge$
 $\langle c_1, \text{mds}, \text{mem} \rangle_w \rightsquigarrow_w^+ \langle c_2, \text{mds}', \text{mem}' \rangle_w$
<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 *AsmNoReadOrWrite* assumption is assigned the value in variable *input*, but where *input*'s classification depends on some control variable. Then the new type of x depends on memory. If we were to just take the upper bound of *input*'s classification, this would likely give us *High* as x 's type, but that would prevent us from treating x as *Low* if we later learn *input*'s original classification.

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

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

type-synonym $'BExp\ Type = 'BExp\ set$

We require Γ to track all stable (i.e. *AsmNoWrite* or *AsmNoReadOrWrite*), non- \mathcal{C} variables.

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

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

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

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

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

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

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

We track a set of predicates on memories as we execute. If we evaluate a boolean expression all of whose variables are stable, then we enrich this set predicate with that one. If we assign to a stable variable, then we enrich

this predicate also. If we release an assumption making a variable unstable, we need to remove all predicates that pertain to it from this set.

This needs to be deeply embedded (i.e. it cannot be stored as a predicate of type $(\text{'Var}, \text{'Val}) \text{Mem} \Rightarrow \text{bool}$ or even $(\text{'Var}, \text{'Val}) \text{Mem set}$), 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-synonym $\text{'bexp preds} = \text{'bexp set}$

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

definition

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

where

$\text{pred } P \equiv \lambda \text{mem}. (\forall p \in P. \text{eval}_B \text{ mem } p)$

end

locale $\text{sifum-types} =$

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

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

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

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

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

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

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

and $\text{C} :: \text{'Var set} +$

fixes $\text{bexp-neg} :: \text{'BExp} \Rightarrow \text{'BExp}$

assumes $\text{bexp-neg-negates}: \bigwedge \text{mem } e. (\text{ev}_B \text{ mem } (\text{bexp-neg } e)) = (\neg (\text{ev}_B \text{ mem } e))$

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

assumes $\text{assign-post-valid}: \bigwedge \text{mem}. \text{pred } P \text{ mem} \Longrightarrow \text{pred } (\text{assign-post } P \text{ } x \text{ } e)$
 $(\text{mem}(x := \text{ev}_A \text{ mem } e))$

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

assumes $\text{dma-correct}:$

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

assumes $\text{C-vars-correct}:$

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

fixes $\text{pred-False} :: \text{'BExp}$

assumes $\text{pred-False-is-False}: \neg \text{ev}_B \text{ mem } \text{pred-False}$

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

locale $\text{sifum-types-assign} =$

sifum-lang-no-dma ev_A ev_B $aexp\text{-vars}$ $bexp\text{-vars}$ + *sifum-security* dma $C\text{-vars}$ C
 $eval_w$ *undefined*

for $ev_A :: ('Var, 'Val) Mem \Rightarrow 'AExp \Rightarrow 'Val$
and $ev_B :: ('Var, 'Val) Mem \Rightarrow 'BExp \Rightarrow bool$
and $aexp\text{-vars} :: 'AExp \Rightarrow 'Var\ set$
and $bexp\text{-vars} :: 'BExp \Rightarrow 'Var\ set$
and $dma :: ('Var, 'Val) Mem \Rightarrow 'Var \Rightarrow Sec$
and $C\text{-vars} :: 'Var \Rightarrow 'Var\ set$
and $C :: 'Var\ set$ +

fixes $bexp\text{-neg} :: 'BExp \Rightarrow 'BExp$
assumes $bexp\text{-neg-negates}: \bigwedge mem\ e. (ev_B\ mem\ (bexp\text{-neg}\ e)) = (\neg (ev_B\ mem\ e))$

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

assumes $dma\text{-correct}$:

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

assumes $C\text{-vars-correct}$:

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

fixes $pred\text{-False} :: 'BExp$

assumes $pred\text{-False-is-False}: \neg ev_B\ mem\ pred\text{-False}$

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

fixes $bexp\text{-assign} :: 'Var \Rightarrow 'AExp \Rightarrow 'BExp$

assumes $bexp\text{-assign-eval}: \bigwedge mem\ e\ x. (ev_B\ mem\ (bexp\text{-assign}\ x\ e)) = (mem\ x = (ev_A\ mem\ e))$

assumes $bexp\text{-assign-vars}: \bigwedge e\ x. (bexp\text{-vars}\ (bexp\text{-assign}\ x\ e)) = aexp\text{-vars}\ e \cup \{x\}$

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

definition

$stable :: 'Var\ Stable \Rightarrow 'Var \Rightarrow bool$

where

$stable\ \mathcal{S}\ x \equiv x \in (fst\ \mathcal{S} \cup snd\ \mathcal{S})$

definition

$add\text{-pred} :: 'BExp\ preds \Rightarrow 'Var\ Stable \Rightarrow 'BExp \Rightarrow 'BExp\ preds\ (-\ +_S\ -\ [120, 120, 120] 1000)$

where

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

lemma $add\text{-pred-subset}$:

$P \subseteq P\ +_S\ p$

<proof>

definition

restrict-preds-to-vars :: 'BExp preds ⇒ 'Var set ⇒ 'BExp preds (- | ' - [120, 120] 1000)

where

$P \mid ' V \equiv \{e. e \in P \wedge \text{bexp-vars } e \subseteq V\}$

end

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

the most simple assignment postcondition transformer

definition

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

where

assign-post $P \ x \ e \equiv$
 (if $x \in (\text{aexp-vars } e)$ then
 (*restrict-preds-to-vars* $P \ (-\{x\})$)
 else
 (*restrict-preds-to-vars* $P \ (-\{x\}) \cup \{\text{bexp-assign } x \ e\}$)

end

sublocale *sifum-types-assign* \subseteq *sifum-types* - - - - - *assign-post*
 ⟨*proof*⟩

context *sifum-types*

begin

abbreviation

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

where

mm-equiv-abv2 $c \ c' \equiv \text{mm-equiv-abv } c \ c'$

abbreviation

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

where

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

abbreviation

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

where

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

abbreviation

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

where

$$x \rightsquigarrow \perp \equiv \forall y. (x, y) \notin \text{eval}_w$$

abbreviation

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

where

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

abbreviation

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

where

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

definition

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

where

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

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

$$\begin{aligned} &\text{type-wellformed } (\text{dma-type } x) \\ &\langle \text{proof} \rangle \end{aligned}$$

definition

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

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

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

where

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

lemma *to-total-type-wellformed*:

$$\begin{aligned} &\text{types-wellformed } \Gamma \implies \\ &\text{type-wellformed } (\text{to-total } \Gamma\ x) \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *Un-type-wellformed*:

$$\begin{aligned} &\forall t \in \text{ts}. \text{type-wellformed } t \implies \text{type-wellformed } (\bigcup\ \text{ts}) \\ &\langle \text{proof} \rangle \end{aligned}$$

inductive

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

where

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

lemma *type-aexprI*:

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

<proof>

lemma *type-aexpr-type-wellformed*:

$$\text{types-wellformed } \Gamma \implies \Gamma \vdash_a e \in t \implies \text{type-wellformed } t$$

<proof>

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

inductive

$$\text{type-bexpr} :: ('Var, 'BExp) \text{ TyEnv} \Rightarrow 'BExp \Rightarrow 'BExp \text{ Type} \Rightarrow \text{bool } (- \vdash_b - \in -)$$

[120, 120, 120] 1000)

where

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

lemma *type-bexprI*:

$$t = \bigcup (\text{image } (\lambda x. \text{to-total } \Gamma x) (\text{bexp-vars } e)) \implies \Gamma \vdash_b e \in t$$

<proof>

lemma *type-bexpr-type-wellformed*:

$$\text{types-wellformed } \Gamma \implies \Gamma \vdash_b e \in t \implies \text{type-wellformed } t$$

<proof>

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

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

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

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

abbreviation

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

where

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

abbreviation

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

where

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

lemma *type-stable-is-sufficient*:

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

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

definition

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

where

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

fun

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

where

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

definition

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

where

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

lemma *add-anno-acq-AsmNoReadOrWrite-idemp* [simp]:

$v \in dom\ \Gamma \vee v \in \mathcal{C} \Longrightarrow \Gamma \oplus_{\mathcal{S}}\ (Acq\ v\ AsmNoReadOrWrite) = \Gamma$
 $\langle proof \rangle$

lemma *add-anno-rel-AsmNoReadOrWrite-idemp* [simp]:

$\llbracket v \notin dom\ \Gamma; v \notin fst\ \mathcal{S} \rrbracket \Longrightarrow \Gamma \oplus_{\mathcal{S}}\ (Rel\ v\ AsmNoReadOrWrite) = \Gamma$
 $\langle proof \rangle$

lemma *add-anno-acq-AsmNoReadOrWrite* [simp]:

assumes *notin* [simp]: $v \notin dom\ \Gamma$

shows $v \notin \mathcal{C} \Longrightarrow \Gamma \oplus_{\mathcal{S}}\ (Acq\ v\ AsmNoReadOrWrite) = (\Gamma(v \mapsto dma-type\ v))$
 $\langle proof \rangle$

lemma *add-anno-rel-AsmNoReadOrWrite* [simp]:

assumes *isin* [simp]: $v \in dom\ \Gamma$

shows $v \notin \text{fst } \mathcal{S} \implies \Gamma \oplus_{\mathcal{S}} (\text{Rel } v \text{ AsmNoReadOrWrite}) = \text{restrict-map } \Gamma ((\text{dom } \Gamma) - \{v\})$
 ⟨proof⟩

lemma *add-anno-acq-AsmNoWrite-idemp* [simp]:
 $v \in \text{dom } \Gamma \vee v \in \mathcal{C} \implies \Gamma \oplus_{\mathcal{S}} (\text{Acq } v \text{ AsmNoWrite}) = \Gamma$
 ⟨proof⟩

lemma *add-anno-rel-AsmNoWrite-idemp* [simp]:
 $\llbracket v \notin \text{dom } \Gamma; v \notin \text{snd } \mathcal{S} \rrbracket \implies \Gamma \oplus_{\mathcal{S}} (\text{Rel } v \text{ AsmNoWrite}) = \Gamma$
 ⟨proof⟩

lemma *add-anno-acq-AsmNoWrite* [simp]:
assumes *notin* [simp]: $v \notin \text{dom } \Gamma$
shows $v \notin \mathcal{C} \implies \Gamma \oplus_{\mathcal{S}} (\text{Acq } v \text{ AsmNoWrite}) = (\Gamma(v \mapsto \text{dma-type } v))$
 ⟨proof⟩

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

fun
add-anno-stable :: 'Var Stable \Rightarrow 'Var ModeUpd \Rightarrow 'Var Stable

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

definition
pred-entailment :: 'BExp preds \Rightarrow 'BExp preds \Rightarrow bool (**infix** \vdash 70)

where
 $P \vdash P' \equiv \forall \text{ mem. } \text{pred } P \text{ mem} \longrightarrow \text{pred } P' \text{ mem}$

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

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

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

definition
type-max :: 'BExp Type \Rightarrow ('Var, 'Val) Mem \Rightarrow Sec
where

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

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

$\llbracket type-stable\ \mathcal{S}\ t \rrbracket \implies$
 $\forall mem\ mem'. (\forall x. stable\ \mathcal{S}\ x \longrightarrow mem\ x = mem'\ x) \longrightarrow type-max\ t\ mem =$
 $type-max\ t\ mem'$
 $\langle proof \rangle$

lemma *subtype-sound*:

$t \leq_P t' \implies \forall mem. pred\ P\ mem \longrightarrow type-max\ t\ mem \leq type-max\ t'\ mem$
 $\langle proof \rangle$

lemma *subtype-complete*:

assumes $a: \bigwedge mem. pred\ P\ mem \implies type-max\ t\ mem \leq type-max\ t'\ mem$
shows $t \leq_P t'$
 $\langle proof \rangle$

lemma *subtype-correct*:

$(t \leq_P t') = (\forall mem. pred\ P\ mem \longrightarrow type-max\ t\ mem \leq type-max\ t'\ mem)$
 $\langle proof \rangle$

definition

$type-equiv :: 'BExp\ Type \Rightarrow 'BExp\ preds \Rightarrow 'BExp\ Type \Rightarrow bool$ ($- =: -$ - [120, 120, 120] 1000)

where

$t =:_P t' \equiv t \leq_P t' \wedge t' \leq_P t$

lemma *subtype-refl [simp]*:

$t \leq_P t$
 $\langle proof \rangle$

lemma *type-equiv-refl [simp]*:

$t =:_P t$
 $\langle proof \rangle$

definition

$anno-type-stable :: ('Var, 'BExp)\ TyEnv \Rightarrow 'Var\ Stable \Rightarrow 'Var\ ModeUpd \Rightarrow bool$

where

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

definition

$anno-type-sec :: ('Var, 'BExp)\ TyEnv \Rightarrow 'Var\ Stable \Rightarrow 'BExp\ preds \Rightarrow 'Var$
 $ModeUpd \Rightarrow bool$

where

$anno\text{-}type\text{-}sec \Gamma \mathcal{S} P \text{ upd} \equiv (case \text{ upd of } (Rel \ v \ AsmNoReadOrWrite) \Rightarrow$
 $(v \in add\text{-}anno\text{-}dom \Gamma \mathcal{S} \text{ upd} \longrightarrow (the (\Gamma \ v)) \leq_P (dma\text{-}type$
 $v))$
 $| \ - \Rightarrow True)$

definition

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

where

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

definition

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

where

$tyenv\text{-}wellformed \ mds \Gamma \mathcal{S} P \equiv$
 $mds\text{-}consistent \ mds \Gamma \mathcal{S} P \wedge$
 $types\text{-}wellformed \Gamma \wedge types\text{-}stable \Gamma \mathcal{S}$

lemma subset-entailment:

$P' \subseteq P \Longrightarrow P \vdash P'$
 $\langle proof \rangle$

lemma pred-entailment-refl [simp]:

$P \vdash P$
 $\langle proof \rangle$

lemma pred-entailment-mono:

$P \vdash P' \Longrightarrow P \subseteq P'' \Longrightarrow P'' \vdash P'$
 $\langle proof \rangle$

lemma type-equiv-subset:

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

definition

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

where

$\Gamma \ =: _P \ \Gamma' \equiv dom \Gamma = dom \Gamma' \wedge$
 $(\forall x \in dom \Gamma'. type\text{-}equiv (the (\Gamma \ x)) \ P (the (\Gamma' \ x)))$

lemma context-equiv-refl [simp]:

$context\text{-}equiv \Gamma \ P \ \Gamma$
 $\langle proof \rangle$

lemma context-equiv-subset:

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

lemma *pred-entailment-trans*:

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

<proof>

lemma *pred-un [simp]*:

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

<proof>

lemma *pred-entailment-un*:

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

<proof>

lemma *pred-entailment-mono-un*:

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

<proof>

lemma *subtype-trans*:

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

<proof>

lemma *type-equiv-trans*:

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

<proof>

lemma *context-equiv-trans*:

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

<proof>

lemma *un-pred-entailment-mono*:

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

<proof>

lemma *subtype-entailment*:

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

<proof>

lemma *type-equiv-entailment*:

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

<proof>

lemma *context-equiv-entailment*:

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

<proof>

inductive

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

where

$stop\text{-}type [intro]: \vdash \Gamma, \mathcal{S}, P \{Stop\} \Gamma, \mathcal{S}, P \mid$
 $skip\text{-}type [intro]: \vdash \Gamma, \mathcal{S}, P \{Skip\} \Gamma, \mathcal{S}, P \mid$
 $assign_C :$
 $\llbracket x \in \mathcal{C}; \Gamma \vdash_a e \in t; P \vdash t; (\forall v \in dom \Gamma. x \notin vars\text{-}of\text{-}type (the (\Gamma v)));$
 $P' = restrict\text{-}preds\text{-}to\text{-}vars (assign\text{-}post P x e) \{v. stable \mathcal{S} v\};$
 $\forall v. x \in \mathcal{C}\text{-}vars v \wedge v \notin snd \mathcal{S} \longrightarrow P \vdash (to\text{-}total \Gamma v) \wedge$
 $(to\text{-}total \Gamma v) \leq_{P'} (dma\text{-}type v) \rrbracket \Longrightarrow$
 $\vdash \Gamma, \mathcal{S}, P \{x \leftarrow e\} \Gamma, \mathcal{S}, P' \mid$
 $assign_1 :$
 $\llbracket x \notin dom \Gamma; x \in \mathcal{C}; \Gamma \vdash_a e \in t; t \leq_P (dma\text{-}type x);$
 $P' = restrict\text{-}preds\text{-}to\text{-}vars (assign\text{-}post P x e) \{v. stable \mathcal{S} v\} \rrbracket \Longrightarrow$
 $\vdash \Gamma, \mathcal{S}, P \{x \leftarrow e\} \Gamma, \mathcal{S}, P' \mid$
 $assign_2 :$
 $\llbracket x \in dom \Gamma; \Gamma \vdash_a e \in t; type\text{-}stable \mathcal{S} t; P' = restrict\text{-}preds\text{-}to\text{-}vars (assign\text{-}post$
 $P x e) \{v. stable \mathcal{S} v\};$
 $x \notin snd \mathcal{S} \longrightarrow t \leq_{P'} (dma\text{-}type x) \rrbracket \Longrightarrow$
 $has\text{-}type \Gamma \mathcal{S} P (x \leftarrow e) (\Gamma (x := Some t)) \mathcal{S} P' \mid$
 $if\text{-}type [intro]:$
 $\llbracket \Gamma \vdash_b e \in t; P \vdash t;$
 $\vdash \Gamma, \mathcal{S}, (P +_{\mathcal{S}} e) \{c_1\} \Gamma', \mathcal{S}', P'; \vdash \Gamma, \mathcal{S}, (P +_{\mathcal{S}} (bexp\text{-}neg e)) \{c_2\} \Gamma'', \mathcal{S}', P'';$
 $context\text{-}equiv \Gamma' P' \Gamma''; context\text{-}equiv \Gamma'' P'' \Gamma''; P' \vdash P''; P'' \vdash P'';$
 $\forall mds. tyenv\text{-}wellformed mds \Gamma' \mathcal{S}' P' \longrightarrow tyenv\text{-}wellformed mds \Gamma'' \mathcal{S}' P'';$
 $\forall mds. tyenv\text{-}wellformed mds \Gamma'' \mathcal{S}' P'' \longrightarrow tyenv\text{-}wellformed mds \Gamma'' \mathcal{S}' P''$
 $\rrbracket \Longrightarrow$
 $\vdash \Gamma, \mathcal{S}, P \{If e c_1 c_2\} \Gamma''', \mathcal{S}', P''' \mid$
 $while\text{-}type [intro]: \llbracket \Gamma \vdash_b e \in t; P \vdash t; \vdash \Gamma, \mathcal{S}, (P +_{\mathcal{S}} e) \{c\} \Gamma, \mathcal{S}, P \rrbracket \Longrightarrow \vdash$
 $\Gamma, \mathcal{S}, P \{While e c\} \Gamma, \mathcal{S}, P \mid$
 $anno\text{-}type [intro]: \llbracket \Gamma' = \Gamma \oplus_{\mathcal{S}} upd; \mathcal{S}' = add\text{-}anno\text{-}stable \mathcal{S} upd; P' = re\text{-}$
 $strict\text{-}preds\text{-}to\text{-}vars P \{v. stable \mathcal{S}' v\};$
 $\vdash \Gamma', \mathcal{S}', P' \{c\} \Gamma'', \mathcal{S}'', P''; c \neq Stop;$
 $(\bigwedge x. (to\text{-}total \Gamma x) \leq_{P'} (to\text{-}total \Gamma' x));$
 $anno\text{-}type\text{-}stable \Gamma \mathcal{S} upd; anno\text{-}type\text{-}sec \Gamma \mathcal{S} P upd \rrbracket \Longrightarrow \vdash \Gamma, \mathcal{S}, P \{$
 $c@[upd]\} \Gamma'', \mathcal{S}'', P'' \mid$
 $seq\text{-}type [intro]: \llbracket \vdash \Gamma, \mathcal{S}, P \{c_1\} \Gamma', \mathcal{S}', P'; \vdash \Gamma', \mathcal{S}', P' \{c_2\} \Gamma'', \mathcal{S}'', P'' \rrbracket \Longrightarrow \vdash$
 $\Gamma, \mathcal{S}, P \{c_1 ;; c_2\} \Gamma'', \mathcal{S}'', P'' \mid$
 $sub : \llbracket \vdash \Gamma_1, \mathcal{S}, P_1 \{c\} \Gamma_1', \mathcal{S}', P_1'; context\text{-}equiv \Gamma_2 P_2 \Gamma_1; (\forall mds. tyenv\text{-}wellformed$
 $mds \Gamma_2 \mathcal{S} P_2 \longrightarrow tyenv\text{-}wellformed mds \Gamma_1 \mathcal{S} P_1);$
 $(\forall mds. tyenv\text{-}wellformed mds \Gamma_1' \mathcal{S}' P_1' \longrightarrow tyenv\text{-}wellformed mds \Gamma_2'$
 $\mathcal{S}' P_2'); context\text{-}equiv \Gamma_1' P_1' \Gamma_2'; P_2 \vdash P_1; P_1' \vdash P_2' \rrbracket \Longrightarrow \vdash \Gamma_2, \mathcal{S}, P_2 \{c\}$
 $\Gamma_2', \mathcal{S}', P_2' \mid$
 $await\text{-}type [intro]:$
 $\llbracket \Gamma \vdash_b e \in t; P \vdash t;$
 $\vdash \Gamma, \mathcal{S}, (P +_{\mathcal{S}} e) \{c\} \Gamma', \mathcal{S}', P' \rrbracket \Longrightarrow$

$\vdash \Gamma, \mathcal{S}, P \{ \text{Await } e \ c \} \Gamma', \mathcal{S}', P'$

lemma *sub'*:

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

lemma *assign₂-helper*:

$\llbracket \Gamma \ x = \text{Some } t; \text{ has-type } \Gamma \ \mathcal{S} \ P \ (x \leftarrow e) \ (\Gamma(x \mapsto t)) \ \mathcal{S} \ P \rrbracket \Longrightarrow \text{has-type } \Gamma \ \mathcal{S} \ P$
 $(x \leftarrow e) \ \Gamma \ \mathcal{S} \ P'$
 $\langle \text{proof} \rangle$

lemma *conc'*:

$\llbracket \vdash \Gamma_1, \mathcal{S}, P \{ c \} \Gamma', \mathcal{S}', P';$
 $\Gamma_1 = (\Gamma_2(x \mapsto t));$
 $x \in \text{dom } \Gamma_2;$
 $\text{type-equiv (the } (\Gamma_2 \ x)) \ P \ t;$
 $\text{type-wellformed } t;$
 $\text{type-stable } \mathcal{S} \ t \rrbracket \Longrightarrow$
 $\vdash \Gamma_2, \mathcal{S}, P \{ c \} \Gamma', \mathcal{S}', P'$
 $\langle \text{proof} \rangle$

lemma *tyenv-wellformed-subset*:

$\text{tyenv-wellformed mds } \Gamma \ \mathcal{S} \ P \Longrightarrow P' \subseteq P \Longrightarrow \text{tyenv-wellformed mds } \Gamma \ \mathcal{S} \ P'$
 $\langle \text{proof} \rangle$

lemma *if-type'*:

$\llbracket \Gamma \vdash_b e \in t;$
 $P \vdash t;$
 $\vdash \Gamma, \mathcal{S}, (P +_{\mathcal{S}} e) \{ c_1 \} \Gamma', \mathcal{S}', P';$
 $\vdash \Gamma, \mathcal{S}, (P +_{\mathcal{S}} (\text{bexp-neg } e)) \{ c_2 \} \Gamma', \mathcal{S}', P'';$
 $P''' \subseteq P' \cap P'' \rrbracket \Longrightarrow$
 $\vdash \Gamma, \mathcal{S}, P \{ \text{If } e \ c_1 \ c_2 \} \Gamma', \mathcal{S}', P'''$
 $\langle \text{proof} \rangle$

lemma *skip-type'*:

$\llbracket \Gamma = \Gamma'; \mathcal{S} = \mathcal{S}'; P = P' \rrbracket \Longrightarrow \vdash \Gamma, \mathcal{S}, P \{ \text{Skip} \} \Gamma', \mathcal{S}', P'$
 $\langle \text{proof} \rangle$

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

$?\Gamma, ?\mathcal{S}, ?P \{ ?c@[?upd] \} ?\Gamma'', ?\mathcal{S}'', ?P''$ rule.

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

$(to\text{-total } \Gamma \ x) \leq_P (to\text{-total } (add\text{-anno } \Gamma \ \mathcal{S} \ (buffer \ +=_m \ AsmNoWrite)) \ x)$
 $(to\text{-total } \Gamma \ x) \leq_P (to\text{-total } (add\text{-anno } \Gamma \ \mathcal{S} \ (buffer \ +=_m \ AsmNoReadOrWrite)) \ x)$
 $\langle proof \rangle$

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.

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

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

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

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

$type\text{-max } (dma\text{-type } x) \ mem = dma \ mem \ x$
 $\langle proof \rangle$

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

lemma *tyenv-eq-sym'*:

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

lemma *tyenv-eq-sym*:

$tyenv\text{-wellformed } mds \ \Gamma \ \mathcal{S} \ P \Longrightarrow mem_1 =_{\Gamma} mem_2 \Longrightarrow mem_2 =_{\Gamma} mem_1$
 $\langle proof \rangle$

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

and $\mathcal{R}_1\text{-abv}$::

$((\text{'Var}, \text{'AExp}, \text{'BExp}) \text{ Stmt}, \text{'Var}, \text{'Val}) \text{ LocalConf} \Rightarrow$
 $(\text{'Var}, \text{'BExp}) \text{ TyEnv} \Rightarrow \text{'Var} \text{ Stable} \Rightarrow \text{'BExp} \text{ preds} \Rightarrow$
 $((\text{'Var}, \text{'AExp}, \text{'BExp}) \text{ Stmt}, \text{'Var}, \text{'Val}) \text{ LocalConf} \Rightarrow$
 $bool \ (- \ \mathcal{R}_1\text{-,,-} \ - [120, 120, 120, 120, 120] \ 1000)$

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

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

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

where

$x \ \mathcal{R}_1\text{-}\Gamma, \mathcal{S}, P \ y \equiv (x, y) \in \mathcal{R}_1 \ \Gamma \ \mathcal{S} \ P \ |$

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

inductive $\mathcal{R}_3\text{-}aux :: ((Var, 'AExp, 'BExp) Stmt, 'Var, 'Val) LocalConf \Rightarrow$
 $(Var, 'BExp) TyEnv \Rightarrow 'Var Stable \Rightarrow 'BExp preds \Rightarrow ((Var, 'AExp,$
 $'BExp) Stmt, 'Var, 'Val) LocalConf \Rightarrow$
 $bool\ (-\ \mathcal{R}^3_{-, -, -}\ -\ [120, 120, 120, 120, 120]\ 1000)$
and $\mathcal{R}_3 :: (Var, 'BExp) TyEnv \Rightarrow 'Var Stable \Rightarrow 'BExp preds \Rightarrow ((Var, 'AExp,$
 $'BExp) Stmt, 'Var, 'Val) LocalConf\ rel$

where
 $\mathcal{R}_3\ \Gamma'\ \mathcal{S}'\ P' \equiv \{(lc_1, lc_2). \mathcal{R}_3\text{-}aux\ lc_1\ \Gamma'\ \mathcal{S}'\ P'\ lc_2\} |$
 $intro_1 [intro] : \llbracket \langle c_1, mds, mem_1 \rangle \mathcal{R}^1_{\Gamma, \mathcal{S}, P} \langle c_2, mds, mem_2 \rangle ; \vdash \Gamma, \mathcal{S}, P \{ c \}$
 $\Gamma', \mathcal{S}', P' \rrbracket \Longrightarrow$
 $\langle Seq\ c_1\ c, mds, mem_1 \rangle \mathcal{R}^3_{\Gamma', \mathcal{S}', P'} \langle Seq\ c_2\ c, mds, mem_2 \rangle |$
 $intro_3 [intro] : \llbracket \langle c_1, mds, mem_1 \rangle \mathcal{R}^3_{\Gamma, \mathcal{S}, P} \langle c_2, mds, mem_2 \rangle ; \vdash \Gamma, \mathcal{S}, P \{ c \}$
 $\Gamma', \mathcal{S}', P' \rrbracket \Longrightarrow$
 $\langle Seq\ c_1\ c, mds, mem_1 \rangle \mathcal{R}^3_{\Gamma', \mathcal{S}', P'} \langle Seq\ c_2\ c, mds, mem_2 \rangle$

definition

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

where

$weak\text{-}bisim\ \mathcal{T}_1\ \mathcal{T} \equiv \forall\ c_1\ c_2\ mds\ mem_1\ mem_2\ c_1'\ mds'\ mem_1'.$
 $((\langle c_1, mds, mem_1 \rangle, \langle c_2, mds, mem_2 \rangle) \in \mathcal{T}_1 \wedge$
 $(\langle c_1, mds, mem_1 \rangle \rightsquigarrow \langle c_1', mds', mem_1' \rangle)) \longrightarrow$
 $(\exists\ c_2'\ mem_2'. \langle c_2, mds, mem_2 \rangle \rightsquigarrow \langle c_2', mds', mem_2' \rangle \wedge$
 $(\langle c_1', mds', mem_1' \rangle, \langle c_2', mds', mem_2' \rangle) \in \mathcal{T})$

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

where

$x\ \mathcal{R}^u_{\Gamma, \mathcal{S}, P}\ y \equiv (x, y) \in \mathcal{R}\ \Gamma\ \mathcal{S}\ P |$
 $intro_1 : lc\ \mathcal{R}^1_{\Gamma, \mathcal{S}, P}\ lc' \Longrightarrow (lc, lc') \in \mathcal{R}\ \Gamma\ \mathcal{S}\ P |$
 $intro_3 : lc\ \mathcal{R}^3_{\Gamma, \mathcal{S}, P}\ lc' \Longrightarrow (lc, lc') \in \mathcal{R}\ \Gamma\ \mathcal{S}\ P$

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

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

inductive-cases $\mathcal{R}\text{-elim}$ [*elim*]: $(\langle c_1, mds, mem_1 \rangle, \langle c_2, mds, mem_2 \rangle) \in \mathcal{R} \Gamma \mathcal{S} P$

inductive-cases $\mathcal{R}\text{-elim}'$: $(\langle c_1, mds, mem_1 \rangle, \langle c_2, mds_2, mem_2 \rangle) \in \mathcal{R} \Gamma \mathcal{S} P$

inductive-cases $\mathcal{R}_1\text{-elim}'$: $\langle c_1, mds, mem_1 \rangle \mathcal{R}^1_{\Gamma, \mathcal{S}, P} \langle c_2, mds_2, mem_2 \rangle$

inductive-cases $\mathcal{R}_3\text{-elim}'$: $\langle c_1, mds, mem_1 \rangle \mathcal{R}^3_{\Gamma, \mathcal{S}, P} \langle c_2, mds_2, mem_2 \rangle$

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

$=_{mds} mem_2$

<proof>

lemma $\mathcal{R}_1\text{-dma-eq}$:

$\langle c_1, mds, mem_1 \rangle \mathcal{R}^1_{\Gamma', \mathcal{S}', P'} \langle c_2, mds, mem_2 \rangle \implies dma mem_1 = dma mem_2$

<proof>

lemma *bisim-simple- \mathcal{R}_1* :

$\langle c, mds, mem \rangle \mathcal{R}^1_{\Gamma, \mathcal{S}, P} \langle c', mds', mem' \rangle \implies c = c'$

<proof>

lemma *bisim-simple- \mathcal{R}_3* :

$lc \mathcal{R}^3_{\Gamma, \mathcal{S}, P} lc' \implies (fst (fst lc)) = (fst (fst lc'))$

<proof>

lemma *bisim-simple- \mathcal{R}_u* :

$lc \mathcal{R}^u_{\Gamma, \mathcal{S}, P} lc' \implies (fst (fst lc)) = (fst (fst lc'))$

<proof>

lemma *C-eq-type-max-eq*:

assumes *wf*: *type-wellformed t*

assumes *C-eq*: $\forall x \in \mathcal{C}. mem_1 x = mem_2 x$

shows *type-max t mem₁ = type-max t mem₂*

<proof>

lemma *vars-of-type-eq-type-max-eq*:

assumes *mem-eq*: $\forall x \in \text{vars-of-type } t. mem_1 x = mem_2 x$

shows *type-max t mem₁ = type-max t mem₂*

<proof>

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

<proof>

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

<proof>

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

lemma \mathcal{R} -sym: $sym (\mathcal{R} \Gamma \mathcal{S} P)$
 $\langle proof \rangle$

lemma \mathcal{R}_1 -closed-glob-consistent: $closed-glob-consistent (\mathcal{R}_1 \Gamma' \mathcal{S}' P')$
 $\langle proof \rangle$

lemma \mathcal{R}_3 -closed-glob-consistent:
 $closed-glob-consistent (\mathcal{R}_3 \Gamma' \mathcal{S}' P')$
 $\langle proof \rangle$

lemma \mathcal{R} -closed-glob-consistent: $closed-glob-consistent (\mathcal{R} \Gamma' \mathcal{S}' P')$
 $\langle proof \rangle$

lemma mode-update-add-anno:
 $mds-consistent mds \Gamma \mathcal{S} P \implies$
 $mds-consistent (update-modes upd mds)$
 $(\Gamma \oplus_{\mathcal{S}} upd)$
 $(add-anno-stable \mathcal{S} upd)$
 $(P \mid \{v. stable (add-anno-stable \mathcal{S} upd) v\})$
 $\langle proof \rangle$

lemma add-anno-acq-GuarNoReadOrWrite [simp]:
 $\Gamma \oplus_{\mathcal{S}} (v \ +=_m \text{GuarNoReadOrWrite}) = \Gamma$
 $\langle proof \rangle$

lemma add-anno-rel-GuarNoReadOrWrite [simp]:
 $\Gamma \oplus_{\mathcal{S}} (v \ -=_m \text{GuarNoReadOrWrite}) = \Gamma$
 $\langle proof \rangle$

lemma add-anno-acq-GuarNoWrite [simp]:
 $\Gamma \oplus_{\mathcal{S}} (v \ +=_m \text{GuarNoWrite}) = \Gamma$
 $\langle proof \rangle$

lemma add-anno-rel-GuarNoWrite [simp]:
 $\Gamma \oplus_{\mathcal{S}} (v \ -=_m \text{GuarNoWrite}) = \Gamma$
 $\langle proof \rangle$

lemma *dom-add-anno-rel*:

$$\forall x \in \text{dom} (\Gamma \oplus_{\mathcal{S}} (v \text{ --}_m m)). (\Gamma \oplus_{\mathcal{S}} (v \text{ --}_m m)) x = \Gamma x$$

<proof>

lemma *types-wellformed-mode-update*:

$$\begin{aligned} & \text{types-wellformed } \Gamma \implies \\ & \text{types-wellformed } (\Gamma \oplus_{\mathcal{S}} \text{upd}) \end{aligned}$$

<proof>

lemma *types-stable-mode-update*:

$$\begin{aligned} & \text{types-stable } \Gamma \mathcal{S} \implies \text{types-wellformed } \Gamma \implies \text{anno-type-stable } \Gamma \mathcal{S} \text{ upd} \\ & \implies \text{types-stable } (\Gamma \oplus_{\mathcal{S}} \text{upd}) \text{ (add-anno-stable } \mathcal{S} \text{ upd)} \end{aligned}$$

<proof>

lemma *tyenv-wellformed-mode-update*:

$$\begin{aligned} & \text{tyenv-wellformed mds } \Gamma \mathcal{S} P \implies \text{anno-type-stable } \Gamma \mathcal{S} \text{ upd} \implies \\ & \text{tyenv-wellformed (update-modes upd mds)} \\ & \quad (\Gamma \oplus_{\mathcal{S}} \text{upd}) \\ & \quad (\text{add-anno-stable } \mathcal{S} \text{ upd}) \\ & \quad (P \mid \{v. \text{stable (add-anno-stable } \mathcal{S} \text{ upd) } v\}) \end{aligned}$$

<proof>

lemma *stop-cxt* :

$$\begin{aligned} & \llbracket \vdash \Gamma, \mathcal{S}, P \{c\} \Gamma', \mathcal{S}', P'; c = \text{Stop} \rrbracket \implies \\ & \text{context-equiv } \Gamma P \Gamma' \wedge \mathcal{S}' = \mathcal{S} \wedge P \vdash P' \wedge (\forall \text{ mds. tyenv-wellformed mds } \Gamma \mathcal{S} P \\ & \longrightarrow \text{tyenv-wellformed mds } \Gamma' \mathcal{S}' P') \end{aligned}$$

<proof>

lemma *tyenv-wellformed-preds-update*:

$$\begin{aligned} & P' = P'' \mid \{v. \text{stable } \mathcal{S} v\} \implies \\ & \text{tyenv-wellformed mds } \Gamma \mathcal{S} P \implies \text{tyenv-wellformed mds } \Gamma \mathcal{S} P' \end{aligned}$$

<proof>

lemma *mds-consistent-preds-tyenv-update*:

$$\begin{aligned} & P' = P'' \mid \{v. \text{stable } \mathcal{S} v\} \implies v \in \text{dom } \Gamma \implies \\ & \text{mds-consistent mds } \Gamma \mathcal{S} P \implies \text{mds-consistent mds } (\Gamma(v \mapsto t)) \mathcal{S} P' \end{aligned}$$

<proof>

lemma *pred-preds-update*:

$$\begin{aligned} & \text{assumes mem'-def: mem}' = \text{mem } (x := \text{ev}_A \text{ mem } e) \\ & \text{assumes P'-def: } P' = (\text{assign-post } P x e) \mid \{v. \text{stable } \mathcal{S} v\} \\ & \text{assumes pred-P: pred } P \text{ mem} \\ & \text{shows pred } P' \text{ mem}' \end{aligned}$$

<proof>

lemma *types-wellformed-update*:

$types\text{-wellformed } \Gamma \implies type\text{-wellformed } t \implies types\text{-wellformed } (\Gamma(x \mapsto t))$
 ⟨proof⟩

lemma *types-stable-update*:

$types\text{-stable } \Gamma \mathcal{S} \implies type\text{-stable } \mathcal{S} t \implies types\text{-stable } (\Gamma(x \mapsto t)) \mathcal{S}$
 ⟨proof⟩

lemma *tyenv-wellformed-sub*:

$\llbracket P_1 \subseteq P_2; \Gamma_2 = \Gamma_1; tyenv\text{-wellformed } mds \Gamma_2 \mathcal{S} P_2 \rrbracket \implies$
 $tyenv\text{-wellformed } mds \Gamma_1 \mathcal{S} P_1$
 ⟨proof⟩

abbreviation

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

where

$tyenv\text{-sec } mds \Gamma \ mem \equiv \forall x \in dom \Gamma. x \notin mds \ AsmNoReadOrWrite \longrightarrow type\text{-max}$
 (the $(\Gamma \ x)$ $mem \leq dma \ mem \ x$)

lemma *tyenv-sec-mode-update*:

$(\forall x. (to\text{-total } \Gamma \ x) \leq_{P''} (to\text{-total } \Gamma'' \ x)) \implies pred \ P'' \ mem \implies \mathcal{S} = (mds$
 $AsmNoWrite, mds \ AsmNoReadOrWrite) \implies$
 $anno\text{-type-sec } \Gamma \ \mathcal{S} \ P \ upd \implies \mathcal{S}'' = add\text{-anno-stable } \mathcal{S} \ upd \implies (\forall p \in P.$
 $\forall v \in bexp\text{-vars } p. stable \ \mathcal{S} \ v) \implies$
 $P'' = P \mid \{v. stable \ \mathcal{S}'' \ v\} \implies$
 $\Gamma'' = \Gamma \oplus_{\mathcal{S}} \ upd \implies tyenv\text{-sec } mds \ \Gamma \ mem \implies tyenv\text{-sec } (update\text{-modes } upd$
 $mds) \ \Gamma'' \ mem$
 ⟨proof⟩

lemma *tyenv-sec-eq*:

$\forall x \in \mathcal{C}. mem \ x = mem' \ x \implies types\text{-wellformed } \Gamma \implies tyenv\text{-sec } mds \ \Gamma \ mem =$
 $tyenv\text{-sec } mds \ \Gamma \ mem'$
 ⟨proof⟩

lemma *context-equiv-tyenv-sec*:

$context\text{-equiv } \Gamma_2 \ P_2 \ \Gamma_1 \implies$
 $pred \ P_2 \ mem \implies tyenv\text{-sec } mds \ \Gamma_2 \ mem \implies tyenv\text{-sec } mds \ \Gamma_1 \ mem$
 ⟨proof⟩

lemma *add-pred-entailment*:

$P +_{\mathcal{S}} p \vdash P$
 ⟨proof⟩

lemma *preservation-no-await*:

$\llbracket \vdash \Gamma, \mathcal{S}, P \{ c \} \Gamma', \mathcal{S}', P' \rrbracket;$

$\langle c, mds, mem \rangle \rightsquigarrow \langle c', mds', mem' \rangle;$
 $\llbracket no\text{-await } c \rrbracket \Longrightarrow$
 $\exists \Gamma'' \mathcal{S}'' P''. (\vdash \Gamma'', \mathcal{S}'', P'' \{ c' \} \Gamma', \mathcal{S}', P') \wedge$
 $(tyenv\text{-wellformed } mds \Gamma \mathcal{S} P \wedge pred P mem \wedge tyenv\text{-sec } mds \Gamma mem \longrightarrow$
 $tyenv\text{-wellformed } mds' \Gamma'' \mathcal{S}'' P'' \wedge$
 $pred P'' mem' \wedge$
 $tyenv\text{-sec } mds' \Gamma'' mem')$
 $\langle proof \rangle$

lemma *preservation-no-await-plus*:

$\llbracket \langle c, mds, mem \rangle \rightsquigarrow^+ \langle c', mds', mem' \rangle \rrbracket$
 $\vdash \Gamma, \mathcal{S}, P \{ c \} \Gamma', \mathcal{S}', P';$
 $\llbracket no\text{-await } c \rrbracket \Longrightarrow$
 $no\text{-await } c' \wedge (\exists \Gamma'' \mathcal{S}'' P''. (\vdash \Gamma'', \mathcal{S}'', P'' \{ c' \} \Gamma', \mathcal{S}', P') \wedge$
 $(tyenv\text{-wellformed } mds \Gamma \mathcal{S} P \wedge pred P mem \wedge tyenv\text{-sec } mds \Gamma mem \longrightarrow$
 $tyenv\text{-wellformed } mds' \Gamma'' \mathcal{S}'' P'' \wedge pred P'' mem' \wedge tyenv\text{-sec } mds' \Gamma'' mem'))$
 $\langle proof \rangle$

lemma *preservation*:

assumes *typed*: $\vdash \Gamma, \mathcal{S}, P \{ c \} \Gamma', \mathcal{S}', P'$
assumes *eval*: $\langle c, mds, mem \rangle \rightsquigarrow \langle c', mds', mem' \rangle$
shows $\exists \Gamma'' \mathcal{S}'' P''. (\vdash \Gamma'', \mathcal{S}'', P'' \{ c' \} \Gamma', \mathcal{S}', P') \wedge$
 $(tyenv\text{-wellformed } mds \Gamma \mathcal{S} P \wedge pred P mem \wedge tyenv\text{-sec } mds \Gamma$
 $mem \longrightarrow$
 $tyenv\text{-wellformed } mds' \Gamma'' \mathcal{S}'' P'' \wedge pred P'' mem' \wedge tyenv\text{-sec}$
 $mds' \Gamma'' mem')$
 $\langle proof \rangle$

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

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

lemma $\mathcal{R}_3\text{-mem-eq}$: $\langle c_1, mds, mem_1 \rangle \mathcal{R}_{\Gamma', \mathcal{S}', P'}^3 \langle c_2, mds, mem_2 \rangle \Longrightarrow mem_1$
 $=_{mds}^l mem_2$
 $\langle proof \rangle$

lemma *ev_A-eq*:

assumes *tyenv-eq*: $mem_1 =_{\Gamma} mem_2$
assumes *pred*: $pred P mem_1$
assumes *e-type*: $\Gamma \vdash_a e \in t$
assumes *subtype*: $t \leq_P (dma\text{-type } v)$
assumes *is-Low*: $dma mem_1 v = Low$

shows $ev_A mem_1 e = ev_A mem_2 e$
 ⟨proof⟩

lemma $ev_A\text{-eq}'$:
assumes $tyenv\text{-eq}: mem_1 =_{\Gamma} mem_2$
assumes $pred: pred P mem_1$
assumes $e\text{-type}: \Gamma \vdash_a e \in t$
assumes $subtype: P \vdash t$
shows $ev_A mem_1 e = ev_A mem_2 e$
 ⟨proof⟩

lemma $ev_B\text{-eq}'$:
assumes $tyenv\text{-eq}: mem_1 =_{\Gamma} mem_2$
assumes $pred: pred P mem_1$
assumes $e\text{-type}: \Gamma \vdash_b e \in t$
assumes $subtype: P \vdash t$
shows $ev_B mem_1 e = ev_B mem_2 e$
 ⟨proof⟩

lemma $R1\text{-equiv-entailment}$:
 ⟨ c, mds, mem ⟩ $\mathcal{R}^1_{\Gamma', \mathcal{S}', P'}$ ⟨ c', mds', mem' ⟩ \implies
 context-equiv $\Gamma' P' \Gamma'' \implies P' \vdash P'' \implies$
 $\forall mds. tyenv\text{-wellformed } mds \Gamma' \mathcal{S}' P' \longrightarrow tyenv\text{-wellformed } mds \Gamma'' \mathcal{S}' P'' \implies$
 ⟨ c, mds, mem ⟩ $\mathcal{R}^1_{\Gamma'', \mathcal{S}', P''}$ ⟨ c', mds', mem' ⟩
 ⟨proof⟩

lemma $R3\text{-equiv-entailment}$:
 ⟨ c, mds, mem ⟩ $\mathcal{R}^3_{\Gamma', \mathcal{S}', P'}$ ⟨ c', mds', mem' ⟩ \implies
 context-equiv $\Gamma' P' \Gamma'' \implies P' \vdash P'' \implies$
 $\forall mds. tyenv\text{-wellformed } mds \Gamma' \mathcal{S}' P' \longrightarrow tyenv\text{-wellformed } mds \Gamma'' \mathcal{S}' P'' \implies$
 ⟨ c, mds, mem ⟩ $\mathcal{R}^3_{\Gamma'', \mathcal{S}', P''}$ ⟨ c', mds', mem' ⟩
 ⟨proof⟩

lemma $R\text{-equiv-entailment}$:
 $lc_1 \mathcal{R}^u_{\Gamma', \mathcal{S}', P'} lc_2 \implies$
 context-equiv $\Gamma' P' \Gamma'' \implies P' \vdash P'' \implies$
 $\forall mds. tyenv\text{-wellformed } mds \Gamma' \mathcal{S}' P' \longrightarrow tyenv\text{-wellformed } mds \Gamma'' \mathcal{S}' P'' \implies$
 $lc_1 \mathcal{R}^u_{\Gamma'', \mathcal{S}', P''} lc_2$
 ⟨proof⟩

lemma $context\text{-equiv}\text{-tyenv}\text{-eq}$:
 $tyenv\text{-eq } \Gamma mem mem' \implies context\text{-equiv } \Gamma P \Gamma' \implies pred P mem \implies tyenv\text{-eq}$
 $\Gamma' mem mem'$
 ⟨proof⟩

lemma $\mathcal{R}\text{-typed}\text{-step}\text{-no}\text{-await}$:
 $\llbracket \vdash \Gamma, \mathcal{S}, P \{ c_1 \} \Gamma', \mathcal{S}', P' ;$

$tyenv\text{-wellformed } mds \Gamma \mathcal{S} P; mem_1 =_{\Gamma} mem_2; pred P mem_1;$
 $pred P mem_2; tyenv\text{-sec } mds \Gamma mem_1;$
 $\langle c_1, mds, mem_1 \rangle \rightsquigarrow \langle c_1', mds', mem_1' \rangle; no\text{-await } c_1 \rrbracket \Longrightarrow$
 $(\exists c_2' mem_2'. \langle c_1, mds, mem_2 \rangle \rightsquigarrow \langle c_2', mds', mem_2' \rangle \wedge$
 $\langle c_1', mds', mem_1' \rangle \mathcal{R}_{\Gamma', \mathcal{S}', P'}^u \langle c_2', mds', mem_2' \rangle)$
 (proof)

lemma *is-final- \mathcal{R}_u -is-final*:

$\langle c_1, mds, mem_1 \rangle \mathcal{R}_{\Gamma, \mathcal{S}, P}^u \langle c_2, mds, mem_2 \rangle \Longrightarrow is\text{-final } c_1 \Longrightarrow is\text{-final } c_2$
 (proof)

lemma *pred-plus-impl*:

$pred P mem \Longrightarrow ev_B mem e \Longrightarrow pred P +_S e mem$
 (proof)

lemma *my- \mathcal{R}_3 -aux-induct* [consumes 1, case-names *intro₁* *intro₃*]:

$\llbracket \langle c_1, mds, mem_1 \rangle \mathcal{R}_{\Gamma, \mathcal{S}, P}^3 \langle c_2, mds, mem_2 \rangle;$
 $\bigwedge c_1 mds mem_1 \Gamma \mathcal{S} P c_2 mem_2 c \Gamma' \mathcal{S}' P'.$
 $\llbracket \langle c_1, mds, mem_1 \rangle \mathcal{R}_{\Gamma, \mathcal{S}, P}^1 \langle c_2, mds, mem_2 \rangle;$
 $\vdash \Gamma, \mathcal{S}, P \{c\} \Gamma', \mathcal{S}', P' \rrbracket \Longrightarrow$
 $Q (c_1 ;; c) mds mem_1 \Gamma' \mathcal{S}' P' (c_2 ;; c) mds mem_2;$
 $\bigwedge c_1 mds mem_1 \Gamma \mathcal{S} P c_2 mem_2 c \Gamma' \mathcal{S}' P'.$
 $\llbracket \langle c_1, mds, mem_1 \rangle \mathcal{R}_{\Gamma, \mathcal{S}, P}^3 \langle c_2, mds, mem_2 \rangle;$
 $Q c_1 mds mem_1 \Gamma \mathcal{S} P c_2 mds mem_2;$
 $\vdash \Gamma, \mathcal{S}, P \{c\} \Gamma', \mathcal{S}', P' \rrbracket \Longrightarrow$
 $Q (c_1 ;; c) mds mem_1 \Gamma' \mathcal{S}' P' (c_2 ;; c) mds mem_2 \rrbracket \Longrightarrow$
 $Q c_1 mds mem_1 \Gamma \mathcal{S} P c_2 mds mem_2$
 (proof)

lemma *\mathcal{R} -typed-step-plus*:

$\llbracket \langle c_1, mds, mem_1 \rangle \rightsquigarrow^+ \langle c_1', mds', mem_1' \rangle;$
 $\vdash \Gamma, \mathcal{S}, P \{c_1\} \Gamma', \mathcal{S}', P';$
 $no\text{-await } c_1;$
 $tyenv\text{-wellformed } mds \Gamma \mathcal{S} P;$
 $mem_1 =_{\Gamma} mem_2;$
 $pred P mem_1;$
 $pred P mem_2;$
 $tyenv\text{-sec } mds \Gamma mem_1 \rrbracket \Longrightarrow$
 $(\exists c_2' mem_2'. \langle c_1, mds, mem_2 \rangle \rightsquigarrow^+ \langle c_2', mds', mem_2' \rangle \wedge$
 $\langle c_1', mds', mem_1' \rangle \mathcal{R}_{\Gamma', \mathcal{S}', P'}^u \langle c_2', mds', mem_2' \rangle)$
 (proof)

lemma *\mathcal{R} -typed-step*:

$\llbracket \vdash \Gamma, \mathcal{S}, P \{c_1\} \Gamma', \mathcal{S}', P';$
 $tyenv\text{-wellformed } mds \Gamma \mathcal{S} P; mem_1 =_{\Gamma} mem_2; pred P mem_1;$
 $pred P mem_2; tyenv\text{-sec } mds \Gamma mem_1;$
 $\langle c_1, mds, mem_1 \rangle \rightsquigarrow \langle c_1', mds', mem_1' \rangle \rrbracket \Longrightarrow$
 $(\exists c_2' mem_2'. \langle c_1, mds, mem_2 \rangle \rightsquigarrow \langle c_2', mds', mem_2' \rangle \wedge$

$\langle c_1', mds', mem_1 \rangle \mathcal{R}_{\Gamma', \mathcal{S}', P'}^u \langle c_2', mds', mem_2 \rangle$
 ⟨proof⟩

lemma \mathcal{R}_1 -weak-bisim:
 weak-bisim $(\mathcal{R}_1 \Gamma' \mathcal{S}' P')$ $(\mathcal{R} \Gamma' \mathcal{S}' P')$
 ⟨proof⟩

lemma \mathcal{R} -to- \mathcal{R}_3 : $\llbracket \langle c_1, mds, mem_1 \rangle \mathcal{R}_{\Gamma, \mathcal{S}, P}^u \langle c_2, mds, mem_2 \rangle ; \vdash \Gamma, \mathcal{S}, P \{ c \} \Gamma', \mathcal{S}', P' \rrbracket \implies$
 $\langle c_1 ;; c, mds, mem_1 \rangle \mathcal{R}_{\Gamma', \mathcal{S}', P'}^3 \langle c_2 ;; c, mds, mem_2 \rangle$
 ⟨proof⟩

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

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

lemma Typed-in- \mathcal{R} :
 assumes typeable: $\vdash \Gamma, \mathcal{S}, P \{ c \} \Gamma', \mathcal{S}', P'$
 assumes wf: tyenv-wellformed mds $\Gamma \mathcal{S} P$
 assumes mem-eq: $\forall x. \text{type-max } (to\text{-total } \Gamma x) \text{ mem}_1 = Low \implies \text{mem}_1 x = \text{mem}_2 x$
 assumes pred₁: pred $P \text{ mem}_1$
 assumes pred₂: pred $P \text{ mem}_2$
 assumes tyenv-sec: tyenv-sec mds $\Gamma \text{ mem}_1$
 shows $\langle c, mds, mem_1 \rangle \mathcal{R}_{\Gamma', \mathcal{S}', P'}^u \langle c, mds, mem_2 \rangle$
 ⟨proof⟩

theorem type-soundness:
 assumes well-typed: $\vdash \Gamma, \mathcal{S}, P \{ c \} \Gamma', \mathcal{S}', P'$
 assumes wf: tyenv-wellformed mds $\Gamma \mathcal{S} P$
 assumes mem-eq: $\forall x. \text{type-max } (to\text{-total } \Gamma x) \text{ mem}_1 = Low \implies \text{mem}_1 x = \text{mem}_2 x$
 assumes pred₁: pred $P \text{ mem}_1$
 assumes pred₂: pred $P \text{ mem}_2$
 assumes tyenv-sec: tyenv-sec mds $\Gamma \text{ mem}_1$
 shows $\langle c, mds, mem_1 \rangle \approx \langle c, mds, mem_2 \rangle$
 ⟨proof⟩

definition
 $\Gamma\text{-of-}mds :: 'Var \ Mds \Rightarrow ('Var, 'BExp) \ TyEnv$
 where

$\Gamma\text{-of-}m\text{-}m\text{-}ds \equiv (\lambda x. \text{if } x \notin \mathcal{C} \wedge x \in m\text{-}ds \text{ AsmNoWrite} \cup m\text{-}ds \text{ AsmNoReadOrWrite}$
then

if $x \in m\text{-}ds \text{ AsmNoReadOrWrite}$ *then*
 Some ($\{\text{pred-False}\}$)
else
 Some (*dma-type* x)
else None)

definition

$\mathcal{S}\text{-of-}m\text{-}ds :: 'Var \ Mds \Rightarrow 'Var \ Stable$

where

$\mathcal{S}\text{-of-}m\text{-}ds \ m\text{-}ds \equiv (m\text{-}ds \ \text{AsmNoWrite}, m\text{-}ds \ \text{AsmNoReadOrWrite})$

definition

$m\text{-}ds\text{-yields-stable-types} :: 'Var \ Mds \Rightarrow bool$

where

$m\text{-}ds\text{-yields-stable-types} \ m\text{-}ds \equiv \forall x. x \in m\text{-}ds \ \text{AsmNoWrite} \cup m\text{-}ds \ \text{AsmNoReadOrWrite}$
 $\text{Write} \longrightarrow$
 $(\forall v \in \mathcal{C}\text{-vars } x. v \in m\text{-}ds \ \text{AsmNoWrite} \cup m\text{-}ds$
 $\text{AsmNoReadOrWrite})$

inductive

$type\text{-global} :: (('Var, 'AExp, 'BExp) \ Stmt \times 'Var \ Mds) \ list \Rightarrow bool$
 $(\vdash - [120] \ 1000)$

where

$\llbracket list\text{-all} \ (\lambda (c, m). (\exists \Gamma' \ \mathcal{S}' \ P'. \vdash (\Gamma\text{-of-}m\text{-}ds \ m), (\mathcal{S}\text{-of-}m\text{-}ds \ m), \{\} \ \{ c \} \ \Gamma', \mathcal{S}', P'))$
 $\wedge m\text{-}ds\text{-yields-stable-types} \ m) \ cs ;$
 $\forall mem. \ sound\text{-mode-use} \ (cs, mem)$
 $\rrbracket \Longrightarrow$
 $type\text{-global} \ cs$

inductive-cases $type\text{-global-elim}: \vdash cs$

lemma $of\text{-}m\text{-}ds\text{-tyenv-wellformed}: m\text{-}ds\text{-yields-stable-types} \ m \Longrightarrow tyenv\text{-wellformed}$
 $m \ (\Gamma\text{-of-}m\text{-}ds \ m) \ (\mathcal{S}\text{-of-}m\text{-}ds \ m) \ \{\}$
 $\langle proof \rangle$

lemma $\Gamma\text{-of-}m\text{-}ds\text{-tyenv-sec}:$

$tyenv\text{-sec} \ m \ (\Gamma\text{-of-}m\text{-}ds \ m) \ mem_1$
 $\langle proof \rangle$

lemma $type\text{-max-pred-False} \ [simp]:$

$type\text{-max} \ \{\text{pred-False}\} \ mem = High$
 $\langle proof \rangle$

lemma $typed\text{-secure}:$

$\llbracket \vdash (\Gamma\text{-of-}m\text{-}ds \ m), (\mathcal{S}\text{-of-}m\text{-}ds \ m), \{\} \ \{ c \} \ \Gamma', \mathcal{S}', P'; m\text{-}ds\text{-yields-stable-types} \ m \rrbracket \Longrightarrow$

com-sifum-secure (*c,m*)
⟨*proof*⟩

lemma *list-all-set*: $\forall x \in \text{set } xs. P x \implies \text{list-all } P xs$
⟨*proof*⟩

theorem *type-soundness-global*:
 assumes *typeable*: $\vdash cs$
 shows *prog-sifum-secure-cont* *cs*
 ⟨*proof*⟩

end
end

6 Type System for Ensuring Locally Sound Use of Modes

theory *LocallySoundModeUse*
imports *Security Language*
begin

6.1 Typing Rules

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

context *sifum-modes*
begin

abbreviation
 eval-abv-modes :: (-, 'Var, 'Val) LocalConf \Rightarrow (-, -, -) LocalConf \Rightarrow bool
 (**infixl** \rightsquigarrow 70)

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

fun
 update-annos :: 'Var Mds \Rightarrow 'Var ModeUpd list \Rightarrow 'Var Mds
 (**infix** \oplus 140)

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

$update-annos\ mds\ (a\ \# \ as) = update-annos\ (update-modes\ a\ mds)\ as$

fun

$annotate :: ('Var, 'AExp, 'BExp)\ Stmt \Rightarrow 'Var\ ModeUpd\ list \Rightarrow ('Var, 'AExp, 'BExp)\ Stmt$

(**infix** \otimes 140)

where

$annotate\ c\ [] = c$ |

$annotate\ c\ (a\ \# \ as) = (annotate\ c\ as)@[a]$

inductive

$mode-type :: 'Var\ Mds \Rightarrow ('Var, 'AExp, 'BExp)\ Stmt \Rightarrow 'Var\ Mds \Rightarrow bool\ (\vdash\ -\ \{-\}\ -)$

where

$skip: \vdash\ mds\ \{ Skip\ \otimes\ annos\ }\ (mds\ \oplus\ annos) |$

$assign: \llbracket x \notin mds\ GuarNoWrite \cup mds\ GuarNoReadOrWrite ; aexp-vars\ e \cap mds\ GuarNoReadOrWrite = \{\};$

$\forall v. (x \in \mathcal{C}\text{-vars}\ v \longrightarrow v \notin mds\ GuarNoWrite \cup mds\ GuarNoReadOrWrite)$

\wedge

$(\mathcal{C}\text{-vars}\ v \cap aexp\text{-vars}\ e \neq \{\} \longrightarrow v \notin mds\ GuarNoReadOrWrite)\rrbracket$

\implies

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

$if: \llbracket \vdash\ (mds\ \oplus\ annos)\ \{ c_1\ }\ mds'' ;$

$\vdash\ (mds\ \oplus\ annos)\ \{ c_2\ }\ mds'' ;$

$bexp\text{-vars}\ e \cap mds\ GuarNoReadOrWrite = \{\};$

$\forall v. \mathcal{C}\text{-vars}\ v \cap bexp\text{-vars}\ e \neq \{\} \longrightarrow v \notin mds\ GuarNoReadOrWrite\rrbracket \implies$

$\vdash\ mds\ \{ If\ e\ c_1\ c_2\ \otimes\ annos\ }\ mds'' |$

$while: \llbracket mds' = mds\ \oplus\ annos ; \vdash\ mds'\ \{ c\ }\ mds' ; bexp\text{-vars}\ e \cap mds'\ GuarNoReadOrWrite = \{\};$

$\forall v. \mathcal{C}\text{-vars}\ v \cap bexp\text{-vars}\ e \neq \{\} \longrightarrow v \notin mds'\ GuarNoReadOrWrite\rrbracket$

\implies

$\vdash\ mds\ \{ While\ e\ c\ \otimes\ annos\ }\ mds' |$

$seq: \llbracket \vdash\ mds\ \{ c_1\ }\ mds' ; \vdash\ mds'\ \{ c_2\ }\ mds'' \rrbracket \implies \vdash\ mds\ \{ c_1\ ;\ ;\ c_2\ }\ mds'' |$

$sub: \llbracket \vdash\ mds_2\ \{ c\ }\ mds_2' ; mds_1 \leq mds_2 ; mds_2' \leq mds_1' \rrbracket \implies$

$\vdash\ mds_1\ \{ c\ }\ mds_1'$

6.2 Soundness of the Type System

lemma *cxt-eval*:

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

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

$\langle proof \rangle$

lemma *update-preserves-le*:

$mds_1 \leq mds_2 \implies (mds_1 \oplus annos) \leq (mds_2 \oplus annos)$

$\langle proof \rangle$

lemma *doesnt-read-annos*:

$doesn't-read-or-modify\ c\ x \implies doesn't-read-or-modify\ (c \otimes annos)\ x$
 ⟨proof⟩

lemma *doesn't-modify-annos*:

$doesn't-modify\ c\ x \implies doesn't-modify\ (c \otimes annos)\ x$
 ⟨proof⟩

lemma *stop-loc-reach*:

$\llbracket \langle c', mds', mem' \rangle \in loc-reach\ \langle Stop, mds, mem \rangle \rrbracket \implies$
 $c' = Stop \wedge mds' = mds$
 ⟨proof⟩

lemma *stop-doesnt-access*:

$doesn't-modify\ Stop\ x \wedge doesn't-read-or-modify\ Stop\ x$
 ⟨proof⟩

lemma *skip-eval-step*:

$\langle Skip \otimes annos, mds, mem \rangle \rightsquigarrow \langle Stop, mds \oplus annos, mem \rangle$
 ⟨proof⟩

lemma *skip-eval-elim*:

$\llbracket \langle Skip \otimes annos, mds, mem \rangle \rightsquigarrow \langle c', mds', mem' \rangle \rrbracket \implies c' = Stop \wedge mds' = mds$
 $\oplus annos \wedge mem' = mem$
 ⟨proof⟩

lemma *skip-doesnt-read*:

$doesn't-read-or-modify\ (Skip \otimes annos)\ x$
 ⟨proof⟩

lemma *skip-doesnt-write*:

$doesn't-modify\ (Skip \otimes annos)\ x$
 ⟨proof⟩

lemma *skip-loc-reach*:

$\llbracket \langle c', mds', mem' \rangle \in loc-reach\ \langle Skip \otimes annos, mds, mem \rangle \rrbracket \implies$
 $(c' = Stop \wedge mds' = (mds \oplus annos)) \vee (c' = Skip \otimes annos \wedge mds' = mds)$
 ⟨proof⟩

lemma *skip-doesnt-access*:

$\llbracket lc \in loc-reach\ \langle Skip \otimes annos, mds, mem \rangle ; lc = \langle c', mds', mem' \rangle \rrbracket \implies$
 $doesn't-read-or-modify\ c'\ x \wedge doesn't-modify\ c'\ x$
 ⟨proof⟩

lemma *assign-doesnt-modify*:

$\llbracket x \neq y ; x \notin \mathcal{C}\text{-vars}\ y \rrbracket \implies doesn't-modify\ ((x \leftarrow e) \otimes annos)\ y$
 ⟨proof⟩

lemma *assign-annos-eval*:

$$\langle (x \leftarrow e) \otimes \text{annos}, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle \text{Stop}, \text{mds} \oplus \text{annos}, \text{mem} (x := \text{ev}_A \text{ mem } e) \rangle$$

<proof>

lemma *assign-annos-eval-elim*:

$$\llbracket \langle (x \leftarrow e) \otimes \text{annos}, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \rangle \rrbracket \implies$$

$$c' = \text{Stop} \wedge \text{mds}' = \text{mds} \oplus \text{annos}$$

<proof>

lemma *mem-upd-commute*:

$$\llbracket x \neq y \rrbracket \implies \text{mem} (x := v_1, y := v_2) = \text{mem} (y := v_2, x := v_1)$$

<proof>

lemma *assign-doesnt-read*:

$$\llbracket y \neq x; y \notin \text{aexp-vars } e; x \notin \mathcal{C}\text{-vars } y; \mathcal{C}\text{-vars } y \cap \text{aexp-vars } e = \{\} \rrbracket \implies$$

$$\text{doesnt-read-or-modify} ((x \leftarrow e) \otimes \text{annos}) y$$

<proof>

lemma *assign-loc-reach*:

$$\llbracket \langle c', \text{mds}', \text{mem}' \rangle \in \text{loc-reach} \langle (x \leftarrow e) \otimes \text{annos}, \text{mds}, \text{mem} \rangle \rrbracket \implies$$

$$(c' = \text{Stop} \wedge \text{mds}' = (\text{mds} \oplus \text{annos})) \vee (c' = (x \leftarrow e) \otimes \text{annos} \wedge \text{mds}' = \text{mds})$$

<proof>

lemma *if-doesnt-modify*:

$$\text{doesnt-modify} (\text{If } e \ c_1 \ c_2 \otimes \text{annos}) x$$

<proof>

lemma *vars-eval_B*:

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

<proof>

lemma *if-doesnt-read*:

$$x \notin \text{bexp-vars } e \implies \mathcal{C}\text{-vars } x \cap \text{bexp-vars } e = \{\} \implies \text{doesnt-read-or-modify} (\text{If } e$$

$$c_1 \ c_2 \otimes \text{annos}) x$$

<proof>

lemma *if-eval-true*:

$$\llbracket \text{ev}_B \text{ mem } e \rrbracket \implies$$

$$\langle \text{If } e \ c_1 \ c_2 \otimes \text{annos}, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c_1, \text{mds} \oplus \text{annos}, \text{mem} \rangle$$

<proof>

lemma *if-eval-false*:

$$\llbracket \neg \text{ev}_B \text{ mem } e \rrbracket \implies$$

$$\langle \text{If } e \ c_1 \ c_2 \otimes \text{annos}, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c_2, \text{mds} \oplus \text{annos}, \text{mem} \rangle$$

<proof>

lemma *if-eval-elim*:

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

$$((c' = c_1 \wedge \text{ev}_B \text{ mem } e) \vee (c' = c_2 \wedge \neg \text{ev}_B \text{ mem } e)) \wedge \text{mds}' = \text{mds} \oplus \text{annos} \wedge$$

$mem' = mem$
 ⟨proof⟩

lemma if-eval-elim':

$\llbracket \langle \text{If } e \ c_1 \ c_2, mds, mem \rangle \rightsquigarrow \langle c', mds', mem' \rangle \rrbracket \implies$
 $((c' = c_1 \wedge ev_B \ mem \ e) \vee (c' = c_2 \wedge \neg ev_B \ mem \ e)) \wedge mds' = mds \wedge mem' = mem$
 ⟨proof⟩

lemma loc-reach-refl':

$\langle c, mds, mem \rangle \in loc\text{-reach} \ \langle c, mds, mem \rangle$
 ⟨proof⟩

lemma if-loc-reach:

$\llbracket \langle c', mds', mem' \rangle \in loc\text{-reach} \ \langle \text{If } e \ c_1 \ c_2 \otimes annos, mds, mem \rangle \rrbracket \implies$
 $(c' = \text{If } e \ c_1 \ c_2 \otimes annos \wedge mds' = mds) \vee$
 $(\exists mem''. \langle c', mds', mem' \rangle \in loc\text{-reach} \ \langle c_1, mds \oplus annos, mem'' \rangle) \vee$
 $(\exists mem''. \langle c', mds', mem' \rangle \in loc\text{-reach} \ \langle c_2, mds \oplus annos, mem'' \rangle)$
 ⟨proof⟩

lemma if-loc-reach':

$\llbracket \langle c', mds', mem' \rangle \in loc\text{-reach} \ \langle \text{If } e \ c_1 \ c_2, mds, mem \rangle \rrbracket \implies$
 $(c' = \text{If } e \ c_1 \ c_2 \wedge mds' = mds) \vee$
 $(\exists mem''. \langle c', mds', mem' \rangle \in loc\text{-reach} \ \langle c_1, mds, mem'' \rangle) \vee$
 $(\exists mem''. \langle c', mds', mem' \rangle \in loc\text{-reach} \ \langle c_2, mds, mem'' \rangle)$
 ⟨proof⟩

lemma seq-loc-reach:

$\llbracket \langle c', mds', mem' \rangle \in loc\text{-reach} \ \langle c_1 ;; c_2, mds, mem \rangle \rrbracket \implies$
 $(\exists c''. c' = c'' ;; c_2 \wedge \langle c'', mds', mem' \rangle \in loc\text{-reach} \ \langle c_1, mds, mem \rangle) \vee$
 $(\exists c'' mds'' mem''. \langle \text{Stop}, mds'', mem'' \rangle \in loc\text{-reach} \ \langle c_1, mds, mem \rangle \wedge$
 $\langle c', mds', mem' \rangle \in loc\text{-reach} \ \langle c_2, mds'', mem'' \rangle)$
 ⟨proof⟩

lemma seq-doesnt-read:

$\llbracket \text{doesnt-read-or-modify } c \ x \rrbracket \implies \text{doesnt-read-or-modify } (c ;; c') \ x$
 ⟨proof⟩

lemma seq-doesnt-modify:

$\llbracket \text{doesnt-modify } c \ x \rrbracket \implies \text{doesnt-modify } (c ;; c') \ x$
 ⟨proof⟩

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

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

$c' = c \wedge mds' = mds \wedge mem' = mem$
 ⟨proof⟩

lemma seq-split:

$$\begin{aligned} & \llbracket \langle \text{Stop}, \text{mds}', \text{mem}' \rangle \in \text{loc-reach } \langle c_1 ;; c_2, \text{mds}, \text{mem} \rangle \rrbracket \implies \\ & \exists \text{mds}'' \text{mem}'' . \langle \text{Stop}, \text{mds}'', \text{mem}'' \rangle \in \text{loc-reach } \langle c_1, \text{mds}, \text{mem} \rangle \wedge \\ & \quad \langle \text{Stop}, \text{mds}', \text{mem}' \rangle \in \text{loc-reach } \langle c_2, \text{mds}'', \text{mem}'' \rangle \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *while-eval*:

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

$$\langle \text{proof} \rangle$$

lemma *while-eval'*:

$$\langle \text{While } e \text{ } c, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle \text{If } e (c ;; \text{While } e \text{ } c) \text{ Stop}, \text{mds}, \text{mem} \rangle$$

$$\langle \text{proof} \rangle$$

lemma *while-eval-elim*:

$$\begin{aligned} & \llbracket \langle \text{While } e \text{ } c \otimes \text{annos}, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \rangle \rrbracket \implies \\ & (c' = \text{If } e (c ;; \text{While } e \text{ } c) \text{ Stop} \wedge \text{mds}' = \text{mds} \oplus \text{annos} \wedge \text{mem}' = \text{mem}) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *while-eval-elim'*:

$$\begin{aligned} & \llbracket \langle \text{While } e \text{ } c, \text{mds}, \text{mem} \rangle \rightsquigarrow \langle c', \text{mds}', \text{mem}' \rangle \rrbracket \implies \\ & (c' = \text{If } e (c ;; \text{While } e \text{ } c) \text{ Stop} \wedge \text{mds}' = \text{mds} \wedge \text{mem}' = \text{mem}) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *while-doesnt-read*:

$$\llbracket x \notin \text{bexp-vars } e \rrbracket \implies \text{doesnt-read-or-modify } (\text{While } e \text{ } c \otimes \text{annos}) x$$

$$\langle \text{proof} \rangle$$

lemma *while-doesnt-modify*:

$$\text{doesnt-modify } (\text{While } e \text{ } c \otimes \text{annos}) x$$

$$\langle \text{proof} \rangle$$

lemma *disjE3*:

$$\llbracket A \vee B \vee C ; A \implies P ; B \implies P ; C \implies P \rrbracket \implies P$$

$$\langle \text{proof} \rangle$$

lemma *disjE5*:

$$\begin{aligned} & \llbracket A \vee B \vee C \vee D \vee E ; A \implies P ; B \implies P ; C \implies P ; D \implies P ; E \implies P \rrbracket \\ & \implies P \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *if-doesnt-read'*:

$$x \notin \text{bexp-vars } e \implies \mathcal{C}\text{-vars } x \cap \text{bexp-vars } e = \{\} \implies \text{doesnt-read-or-modify } (\text{If } e \text{ } c_1 \text{ } c_2) x$$

$$\langle \text{proof} \rangle$$

theorem *mode-type-sound*:

$$\text{assumes } \text{typeable} : \vdash \text{mds}_1 \{ c \} \text{mds}_1'$$

```

assumes mode-le:  $mds_2 \leq mds_1$ 
shows  $\forall mem. (\langle Stop, mds_2', mem \rangle \in loc\text{-}reach \langle c, mds_2, mem \rangle \longrightarrow mds_2' \leq mds_1') \wedge$ 
    locally-sound-mode-use  $\langle c, mds_2, mem \rangle$ 
   $\langle proof \rangle$ 

end

end

```

References

- [GMS14] Sylvia Grewe, Heiko Mantel, and Daniel Schoepe. A formalisation of assumptions and guarantees for compositional noninterference. *Archive of Formal Proofs*, 2014. http://isa-afp.org/entries/SIFUM_Type_Systems.shtml.
- [MSPR16] Toby Murray, Robert Sison, Edward Pierzhalski, and Christine Rizkallah. Compositional verification and refinement of concurrent value-dependent noninterference. In *IEEE Computer Security Foundations Symposium*, Lisbon, Portugal, June 2016.
- [MSS11] Heiko Mantel, David Sands, and Henning Sudbrock. Assumptions and Guarantees for Compositional Noninterference. In *IEEE Computer Security Foundation Symposium*, pages 218–232. IEEE Computer Society, 2011.
- [Mur15] Toby Murray. On high-assurance information-flow-secure programming languages. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 43–48, Prague, Czech Republic, July 2015.