A Dependent Security Type System for Concurrent Imperative Programs

Toby Murray, Robert Sison, Edward Pierzchalski and Christine Rizkallah

December 14, 2021

Abstract

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

The formalization includes the following parts:

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

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

Contents

1 Preliminaries

theory *Preliminaries*
imports *Main*
begin

Possible modes for variables:

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

We consider a two-element security lattice:

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

Sec forms a (complete) lattice:

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

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

instance
⟨*proof*⟩

end

Memories are mappings from variables to values

type-synonym (*'var*, *'val*) *Mem* = *'var* ⇒ *'val*

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

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

Local configurations:

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

Global configurations:

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

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

locale *sifum-security-init* =
fixes *dma* :: (*'Var*,*'Val*) *Mem* ⇒ *'Var* ⇒ *Sec*
fixes $\mathcal{C}$*-vars* :: *'Var* ⇒ *'Var set*
fixes $\mathcal{C}$ :: *'Var set*
fixes *eval* :: (*'Com*, *'Var*, *'Val*) *LocalConf rel*
fixes *some-val* :: *'Val*
fixes *INIT* :: (*'Var*,*'Val*) *Mem* ⇒ *bool*
assumes *deterministic*: ⟦ (*lc*, *lc'*) ∈ *eval*; (*lc*, *lc''*) ∈ *eval* ⟧ ⟹ *lc'* = *lc''*
assumes *finite-memory*: *finite* {(*x*::*'Var*). *True*}
defines $\mathcal{C} \equiv \bigcup x.\ \mathcal{C}$*-vars x*
assumes $\mathcal{C}$*-vars-*$\mathcal{C}$: *x* ∈ $\mathcal{C}$ ⟹ $\mathcal{C}$*-vars x* = {}
assumes *dma-*$\mathcal{C}$*-vars*: ∀ *x*∈$\mathcal{C}$*-vars y*. *mem*$_1$ *x* = *mem*$_2$ *x* ⟹ *dma mem*$_1$ *y* = *dma mem*$_2$ *y*
assumes $\mathcal{C}$*-Low*: ∀ *x*∈$\mathcal{C}$. *dma mem x* = *Low*

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

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

lemma $\mathcal{C}$*-vars-subset-*$\mathcal{C}$:
$\mathcal{C}$*-vars x* ⊆ $\mathcal{C}$
⟨*proof*⟩

lemma *dma-*$\mathcal{C}$:

$\forall x{\in}\mathcal{C}.\ mem_1\ x = mem_2\ x \Longrightarrow dma\ mem_1 = dma\ mem_2$
⟨*proof*⟩

end

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

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

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

definition *apply-adaptation* ::
bool ⇒ (*'Var*, *'Val*) *Mem* ⇒ (*'Var*, *'Val*) *adaptation* ⇒ (*'Var*, *'Val*) *Mem*
where *apply-adaptation first mem A* =
(λ *x*. *case* (*A x*) *of*
Some (*v*$_1$, *v*$_2$) ⇒ *if first then v*$_1$ *else v*$_2$
| *None* ⇒ *mem x*)

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

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

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

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

2.1 Evaluation of Concurrent Programs

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

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

inductive-set *meval* :: ((-,-,-) *GlobalConf* × *nat* × (-,-,-) *GlobalConf*) *set*
and *meval-abv* :: - ⇒ - ⇒ - ⇒ *bool* (- ⇝$_-$ - *70*)
where
conf ⇝$_k$ *conf'* ≡ (*conf*, *k*, *conf'*) ∈ *meval* |
meval-intro [*iff*]: ⟦ (*cms* ! *n*, *mem*) ⇝ (*cm'*, *mem'*); *n* < *length cms* ⟧ ⟹
((*cms*, *mem*), *n*, (*cms* [*n* := *cm'*], *mem'*)) ∈ *meval*

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

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

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

lemma *neval-det*:
x $\rightsquigarrow^n$ *y* $\Longrightarrow$ *x* $\rightsquigarrow^n$ *y'* $\Longrightarrow$ *y* = *y'*
⟨*proof*⟩

lemma *neval-Suc-simp* [*simp*]:
neval x (Suc 0) y = *x* $\rightsquigarrow$ *y*
⟨*proof*⟩

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

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

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

lemma *meval-sched-det*:
meval-sched ns c c' $\Longrightarrow$ *meval-sched ns c c''* $\Longrightarrow$ *c'* = *c''*
⟨*proof*⟩

2.2 Low-equivalence and Strong Low Bisimulations

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

definition
low-mds-eq :: *'Var Mds* ⇒ (*'Var*, *'Val*) *Mem* ⇒ (-, -) *Mem* ⇒ *bool*
(- $=^l$ - [*100*, *100*] *80*)
where
(*mem*$_1$ $=_{mds}{}^l$ *mem*$_2$) ≡ (∀ *x*. *dma mem*$_1$ *x* = *Low* ∧ (*x* ∈ $\mathcal{C}$ ∨ *x* ∉ *mds Asm-NoReadOrWrite*) ⟶ *mem*$_1$ *x* = *mem*$_2$ *x*)

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

lemma *low-mds-eq-dma*:
$(mem_1 =_{mds}{}^{l} mem_2) \Longrightarrow dma\ mem_1 = dma\ mem_2$
⟨*proof*⟩

lemma *low-mds-eq-sym*:
$(mem_1 =_{mds}{}^{l} mem_2) \Longrightarrow (mem_2 =_{mds}{}^{l} mem_1)$
⟨*proof*⟩

lemma *low-eq-sym*:
$(mem_1 =^{l} mem_2) \Longrightarrow (mem_2 =^{l} mem_1)$
⟨*proof*⟩

lemma [*simp*]: $mem =^{l} mem' \Longrightarrow mem =_{mds}{}^{l} mem'$
⟨*proof*⟩

lemma [*simp*]: $(\forall\ mds.\ mem =_{mds}{}^{l} mem') \Longrightarrow mem =^{l} mem'$
⟨*proof*⟩

lemma *High-not-in-*$\mathcal{C}$ [*simp*]:
$dma\ mem_1\ x = High \Longrightarrow x \notin \mathcal{C}$
⟨*proof*⟩

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

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

$(mem_1 =_{mds}{}^{l} mem_2) \wedge$
$(\forall\ c_1{}'\ mds'\ mem_1{}'.\ \langle\ c_1,\ mds,\ mem_1\ \rangle \rightsquigarrow \langle\ c_1{}',\ mds',\ mem_1{}'\ \rangle \longrightarrow$
$(\exists\ c_2{}'\ mem_2{}'.\ \langle\ c_2,\ mds,\ mem_2\ \rangle \rightsquigarrow \langle\ c_2{}',\ mds',\ mem_2{}'\ \rangle \wedge$
$(\langle\ c_1{}',\ mds',\ mem_1{}'\ \rangle, \langle\ c_2{}',\ mds',\ mem_2{}'\ \rangle) \in \mathcal{R})))$

inductive-set *mm-equiv* :: $(('Com, 'Var, 'Val)\ LocalConf)\ rel$
and *mm-equiv-abv* :: $('Com, 'Var, 'Val)\ LocalConf \Rightarrow$
$('Com, 'Var, 'Val)\ LocalConf \Rightarrow bool$ (**infix** $\approx$ *60*)
where
mm-equiv-abv x y $\equiv (x, y) \in$ *mm-equiv* |
mm-equiv-intro [*iff*]: $[\![$ *strong-low-bisim-mm* $\mathcal{R}$; $(lc_1, lc_2) \in \mathcal{R}\ ]\!] \Longrightarrow (lc_1, lc_2) \in$ *mm-equiv*

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

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

2.3 SIFUM-Security

definition
com-sifum-secure :: *'Com* $\times$ *'Var Mds* $\Rightarrow$ *bool*
where
com-sifum-secure cmd $\equiv$ *case cmd of* $(c,mds_s) \Rightarrow c \sim_{mds_s} c$

definition
prog-sifum-secure-cont :: $('Com \times 'Var\ Mds)\ list \Rightarrow bool$
where *prog-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-sifum-secure-cont-def2*:
prog-sifum-secure-cont cmds $\equiv$
$(\forall\ mem_1\ mem_2.\ INIT\ mem_1 \wedge INIT\ mem_2 \wedge mem_1 =^{l} mem_2 \longrightarrow$
$(\forall\ sched\ cms_1{}'\ mem_1{}'.$
$(cmds, mem_1) \rightarrow_{sched} (cms_1{}', mem_1{}') \longrightarrow$
$(\exists\ cms_2{}'\ mem_2{}'.\ (cmds, mem_2) \rightarrow_{sched} (cms_2{}', mem_2{}')) \wedge$

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

2.4 Sound Mode Use

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

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

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

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

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

lemma $vars\text{-}\mathcal{C}\text{-}subset\text{-}\mathcal{C}$:
$vars\text{-}\mathcal{C}\ X \subseteq \mathcal{C}$
$\langle proof \rangle$

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

definition

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

lemma *noread-nowrite*:
assumes *step*: ⟨*c*, *mds*, *mem*⟩ ⇝ ⟨*c'*, *mds'*, *mem'*⟩
assumes *noread*: (⋀*v*. ⟨*c*, *mds*, *mem*(*x* := *v*)⟩ ⇝ ⟨*c'*, *mds'*, *mem'*(*x* := *v*)⟩)
shows *mem x* = *mem' x*
⟨*proof*⟩

lemma *doesnt-read-or-modify-doesnt-modify*:
doesnt-read-or-modify c x ⟹ *doesnt-modify c x*
⟨*proof*⟩

inductive-set
loc-reach :: (*'Com*, *'Var*, *'Val*) *LocalConf* ⇒ (*'Com*, *'Var*, *'Val*) *LocalConf set*
for *lc* :: (-, -, -) *LocalConf*
where
refl : ⟨*fst* (*fst lc*), *snd* (*fst lc*), *snd lc*⟩ ∈ *loc-reach lc* |
step : ⟦ ⟨*c'*, *mds'*, *mem'*⟩ ∈ *loc-reach lc*;
⟨*c'*, *mds'*, *mem'*⟩ ⇝ ⟨*c''*, *mds''*, *mem''*⟩ ⟧ ⟹
⟨*c''*, *mds''*, *mem''*⟩ ∈ *loc-reach lc* |
mem-diff : ⟦ ⟨ *c'*, *mds'*, *mem'* ⟩ ∈ *loc-reach lc*;
(∀ *x*. *var-asm-not-written mds' x* ⟶ *mem' x* = *mem'' x* ∧ *dma mem' x* = *dma mem'' x*) ⟧ ⟹
⟨ *c'*, *mds'*, *mem''* ⟩ ∈ *loc-reach lc*

lemma *neval-loc-reach*:
neval lc' n lc'' ⟹ *lc'* ∈ *loc-reach lc* ⟹ *lc''* ∈ *loc-reach lc*
⟨*proof*⟩

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

definition
respects-own-guarantees :: (*'Com* × *'Var Mds*) ⇒ *bool*
where
respects-own-guarantees cm ≡

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

lemma *locally-sound-mode-use-def2*:
locally-sound-mode-use lc $\equiv \forall\, lc' \in$ *loc-reach lc. respects-own-guarantees* (*fst lc'*)
⟨*proof*⟩

lemma *locally-sound-respects-guarantees*:
locally-sound-mode-use (*cm*, *mem*) $\Longrightarrow$ *respects-own-guarantees cm*
⟨*proof*⟩

definition
compatible-modes :: ('*Var Mds*) *list* ⇒ *bool*
where
compatible-modes mdss = $(\forall\ (i :: nat)\ x.\ i < length\ mdss \longrightarrow$
$(x \in (mdss\ !\ i)\ AsmNoReadOrWrite \longrightarrow$
$(\forall\ j < length\ mdss.\ j \neq i \longrightarrow x \in (mdss\ !\ j)\ GuarNoReadOrWrite)) \wedge$
$(x \in (mdss\ !\ i)\ AsmNoWrite \longrightarrow$
$(\forall\ j < length\ mdss.\ j \neq i \longrightarrow x \in (mdss\ !\ j)\ GuarNoWrite)))$

definition
reachable-mode-states :: ('*Com*, '*Var*, '*Val*) *GlobalConf* ⇒ (('*Var Mds*) *list*) *set*
where
reachable-mode-states gc ≡
$\{mdss.\ (\exists\ cms'\ mem'\ sched.\ gc \rightarrow_{sched} (cms', mem') \wedge map\ snd\ cms' = mdss)\}$

definition
globally-sound-mode-use :: ('*Com*, '*Var*, '*Val*) *GlobalConf* ⇒ *bool*
where
globally-sound-mode-use gc ≡
$(\forall\ mdss.\ mdss \in reachable\text{-}mode\text{-}states\ gc \longrightarrow compatible\text{-}modes\ mdss)$

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

lemma *mm-equiv-sym*:
assumes *equivalent*: $\langle c_1, mds_1, mem_1\rangle \approx \langle c_2, mds_2, mem_2\rangle$
shows $\langle c_2, mds_2, mem_2\rangle \approx \langle c_1, mds_1, mem_1\rangle$
⟨*proof*⟩

lemma *low-indistinguishable-sym*: $lc \sim_{mds} lc' \Longrightarrow lc' \sim_{mds} lc$
⟨*proof*⟩

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 mem$_1$ *mem*$_2$ ≡ {*x*. *mem*$_1$ *x* ≠ *mem*$_2$ *x*}

definition
differing-vars-lists :: ('*Var*, '*Val*) *Mem* ⇒ (-, -) *Mem* ⇒
((-, -) *Mem* × (-, -) *Mem*) *list* ⇒ *nat* ⇒ '*Var set*
where
differing-vars-lists mem$_1$ *mem*$_2$ *mems i* ≡
(*differing-vars mem*$_1$ (*fst* (*mems* ! *i*)) ∪ *differing-vars mem*$_2$ (*snd* (*mems* ! *i*)))

lemma *differing-finite*: *finite* (*differing-vars mem*$_1$ *mem*$_2$)
⟨*proof*⟩

lemma *differing-lists-finite*: *finite* (*differing-vars-lists mem*$_1$ *mem*$_2$ *mems i*)
⟨*proof*⟩

fun *makes-compatible* ::
('*Com*, '*Var*, '*Val*) *GlobalConf* ⇒
('*Com*, '*Var*, '*Val*) *GlobalConf* ⇒
((-, -) *Mem* × (-, -) *Mem*) *list* ⇒
bool
where
makes-compatible (*cms*$_1$, *mem*$_1$) (*cms*$_2$, *mem*$_2$) *mems* =

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

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

lemma *compat-low*:
$[\![$ *makes-compatible* $(cms_1, mem_1)\ (cms_2, mem_2)\ mems;$
$i < length\ cms_1;$
$x \in differing\text{-}vars\text{-}lists\ mem_1\ mem_2\ mems\ i\ ]\!] \Longrightarrow dma\ mem_1\ x = Low$
⟨*proof*⟩

lemma *compat-different*:
$[\![$ *makes-compatible* $(cms_1, mem_1)\ (cms_2, mem_2)\ mems;$
$i < length\ cms_1;$
$x \in differing\text{-}vars\text{-}lists\ mem_1\ mem_2\ mems\ i\ ]\!] \Longrightarrow mem_1\ x \neq mem_2\ x \wedge dma\ mem_1\ x = Low \wedge x \notin \mathcal{C}$
⟨*proof*⟩

lemma *sound-modes-no-read* :
$[\![$ *sound-mode-use* $(cms, mem);\ x \in (map\ snd\ cms\ !\ i)\ GuarNoReadOrWrite;\ i < length\ cms\ ]\!] \Longrightarrow$
doesnt-read-or-modify $(fst\ (cms\ !\ i))\ x$
⟨*proof*⟩

lemma *differing-vars-neg*: $x \notin differing\text{-}vars\text{-}lists\ mem1\ mem2\ mems\ i \Longrightarrow$
$(fst\ (mems\ !\ i)\ x = mem1\ x \wedge snd\ (mems\ !\ i)\ x = mem2\ x)$
⟨*proof*⟩

lemma *differing-vars-neg-intro*:

⟦ $mem_1\ x = fst\ (mems\ !\ i)\ x$;
$mem_2\ x = snd\ (mems\ !\ i)\ x$ ⟧ ⟹ $x \notin$ *differing-vars-lists* $mem_1\ mem_2\ mems\ i$
⟨*proof*⟩

lemma *differing-vars-elim* [*elim*]:
$x \in$ *differing-vars-lists* $mem_1\ mem_2\ mems\ i$ ⟹
(*fst* $(mems\ !\ i)\ x \neq mem_1\ x$) ∨ (*snd* $(mems\ !\ i)\ x \neq mem_2\ x$)
⟨*proof*⟩

lemma *makes-compatible-dma-eq*:
assumes *compat*: *makes-compatible* $(cms_1, mem_1)\ (cms_2, mem_2)\ mems$
assumes *ile*: $i <$ *length* cms_1
assumes *domσ*: *dom* $\sigma =$ *differing-vars-lists* $mem_1\ mem_2\ mems\ i$
shows *dma* ((*fst* $(mems\ !\ i)$) [↦ σ]) = *dma* mem_1
⟨*proof*⟩

lemma *compat-different-vars*:
⟦ *fst* $(mems\ !\ i)\ x =$ *snd* $(mems\ !\ i)\ x$;
$x \notin$ *differing-vars-lists* $mem_1\ mem_2\ mems\ i$ ⟧ ⟹
$mem_1\ x = mem_2\ x$
⟨*proof*⟩

lemma *differing-vars-subst* [*rule-format*]:
assumes *domσ*: *dom* $\sigma \supseteq$ *differing-vars* $mem_1\ mem_2$
shows mem_1 [↦ σ] = mem_2 [↦ σ]
⟨*proof*⟩

lemma *mm-equiv-low-eq*:
⟦ ⟨ c_1, *mds*, mem_1 ⟩ ≈ ⟨ c_2, *mds*, mem_2 ⟩ ⟧ ⟹ $mem_1 =_{mds}{}^{l}\ mem_2$
⟨*proof*⟩

lemma *globally-sound-modes-compatible*:
⟦ *globally-sound-mode-use* (*cms*, *mem*) ⟧ ⟹ *compatible-modes* (*map snd cms*)
⟨*proof*⟩

lemma *compatible-different-no-read* :
assumes *sound-modes*: *sound-mode-use* (cms_1, mem_1)
sound-mode-use (cms_2, mem_2)
assumes *compat*: *makes-compatible* $(cms_1, mem_1)\ (cms_2, mem_2)\ mems$
assumes *modes-eq*: *map snd* $cms_1 =$ *map snd* cms_2
assumes *ile*: $i <$ *length* cms_1
assumes x: $x \in$ *differing-vars-lists* $mem_1\ mem_2\ mems\ i$
shows *doesnt-read-or-modify* (*fst* $(cms_1\ !\ i)$) x ∧ *doesnt-read-or-modify* (*fst* (cms_2 ! i)) x
⟨*proof*⟩

definition
vars-and-C :: *'Var set* ⇒ *'Var set*

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

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

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

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

lemma *dom-restrict-total*: *dom* (*to-partial f* |` *X*) = *X*
⟨*proof*⟩

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

lemma *change-respecting-subset'*:
assumes *step*: (*cms*, *mem*) ⇝ (*cms'*, *mem'*)
assumes *noread*: (∀ σ. *dom* σ = *X* ⟶ (*cms*, *mem* [↦ σ]) ⇝ (*cms'*, *mem'* [↦ σ]))
assumes *dom-subset*: *dom* σ ⊆ *X*
shows (*cms*, *mem* [↦ σ]) ⇝ (*cms'*, *mem'* [↦ σ])
⟨*proof*⟩

lemma *change-respecting-subst*:
change-respecting (*cms*, *mem*) (*cms'*, *mem'*) *X* ⟹
(∀ σ. *dom* σ = *X* ⟶ (*cms*, *mem* [↦ σ]) ⇝ (*cms'*, *mem'* [↦ σ]))
⟨*proof*⟩

lemma *change-respecting-intro* [*iff*]:
⟦ ⟨ *c*, *mds*, *mem* ⟩ ⇝ ⟨ *c'*, *mds'*, *mem'* ⟩;
⋀ *f*. *dom f* = *vars-and-$\mathcal{C}$* *X* ⟹
(⟨ *c*, *mds*, *mem* [↦ *f*] ⟩ ⇝ ⟨ *c'*, *mds'*, *mem'* [↦ *f*] ⟩) ⟧
⟹ *change-respecting* ⟨*c*, *mds*, *mem*⟩ ⟨*c'*, *mds'*, *mem'*⟩ *X*
⟨*proof*⟩

lemma *vars-$\mathcal{C}$-mono*:
$X \subseteq Y \Longrightarrow$ *vars-$\mathcal{C}$* $X \subseteq$ *vars-$\mathcal{C}$* Y
⟨*proof*⟩

lemma *vars-$\mathcal{C}$-Un*:
vars-$\mathcal{C}$ $(X \cup Y) =$ (*vars-$\mathcal{C}$* $X \cup$ *vars-$\mathcal{C}$* Y)
⟨*proof*⟩

lemma *vars-$\mathcal{C}$-insert*:
vars-$\mathcal{C}$ (*insert* x Y) = (*vars-$\mathcal{C}$* $\{x\}$) $\cup$ (*vars-$\mathcal{C}$* Y)
⟨*proof*⟩

lemma *vars-$\mathcal{C}$-empty*[*simp*]:
vars-$\mathcal{C}$ $\{\} = \{\}$
⟨*proof*⟩

lemma *$\mathcal{C}$-vars-of-$\mathcal{C}$-vars-empty*:
$x \in$ *$\mathcal{C}$-vars* $y \Longrightarrow$ *$\mathcal{C}$-vars* $x = \{\}$
⟨*proof*⟩

lemma *vars-and-$\mathcal{C}$-mono*:
$X \subseteq X' \Longrightarrow$ *vars-and-$\mathcal{C}$* $X \subseteq$ *vars-and-$\mathcal{C}$* X'
⟨*proof*⟩

lemma *$\mathcal{C}$-vars-finite*[*simp*]:
finite (*$\mathcal{C}$-vars* x)
⟨*proof*⟩

lemma *finite-dom*:
finite (*dom* (σ::*'Var* ⇒ *'Val option*))
⟨*proof*⟩

lemma *doesnt-read-or-modify-subst*:
assumes *noread*: *doesnt-read-or-modify* c x
assumes *step*: $\langle c,\ mds,\ mem\rangle \rightsquigarrow \langle c',\ mds',\ mem'\rangle$
assumes *subset*: $X \subseteq \{x\} \cup$ *$\mathcal{C}$-vars* x
shows $\bigwedge \sigma.\ dom\ \sigma = X \Longrightarrow \langle c,\ mds,\ mem[\mapsto \sigma]\rangle \rightsquigarrow \langle c',\ mds',\ mem'[\mapsto \sigma]\rangle$
⟨*proof*⟩

lemma *subst-restrict-twice*:
dom $\sigma = A \cup B \Longrightarrow$
mem $[\mapsto (\sigma\ |`\ A)]\ [\mapsto (\sigma\ |`\ B)] =$ *mem* $[\mapsto \sigma]$
⟨*proof*⟩

lemma *noread-exists-change-respecting*:
assumes *fin*: *finite* (X :: *'Var set*)
assumes *eval*: ⟨ c, *mds*, *mem* ⟩ ⇝ ⟨ c', *mds'*, *mem'* ⟩
assumes *noread*: ∀ $x \in X$. *doesnt-read-or-modify* c x

shows *change-respecting* ⟨*c*, *mds*, *mem*⟩ ⟨*c'*, *mds'*, *mem'*⟩ *X*
⟨*proof*⟩

lemma *update-nth-eq*:
⟦ *xs* = *ys*; *n* < *length xs* ⟧ ⟹ *xs* = *ys* [*n* := *xs* ! *n*]
⟨*proof*⟩

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

lemma *mm-equiv-step*:
assumes *bisim*: (*cms*$_1$, *mem*$_1$) ≈ (*cms*$_2$, *mem*$_2$)
assumes *modes-eq*: *snd cms*$_1$ = *snd cms*$_2$
assumes *step*: (*cms*$_1$, *mem*$_1$) ⇝ (*cms*$_1$', *mem*$_1$')
shows ∃ *c*$_2$' *mem*$_2$'. (*cms*$_2$, *mem*$_2$) ⇝ ⟨ *c*$_2$', *snd cms*$_1$', *mem*$_2$' ⟩ ∧
(*cms*$_1$', *mem*$_1$') ≈ ⟨ *c*$_2$', *snd cms*$_1$', *mem*$_2$' ⟩
⟨*proof*⟩

lemma *change-respecting-doesnt-modify*:
assumes *cr*: *change-respecting* (*cms*, *mem*) (*cms'*, *mem'*) *X*
assumes *eval*: (*cms*, *mem*) ⇝ (*cms'*, *mem'*)
assumes *x-in-dom*: *x* ∈ *X* ∪ *vars-C X*
shows *mem x* = *mem' x*
⟨*proof*⟩

lemma *change-respecting-doesnt-modify-dma*:
assumes *cr*: *change-respecting* (*cms*, *mem*) (*cms'*, *mem'*) *X*
assumes *eval*: (*cms*, *mem*) ⇝ (*cms'*, *mem'*)
assumes *x-in-dom*: *x* ∈ *X*
shows *dma mem x* = *dma mem' x*
⟨*proof*⟩

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

lemma *differing-empty-eq*:
⟦ *differing-vars mem mem'* = {} ⟧ ⟹ *mem* = *mem'*
⟨*proof*⟩

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

definition
globally-consistent :: ('*Var*, '*Val*) *adaptation* ⇒ '*Var Mds* ⇒ ('*Var*,'*Val*) *Mem*
⇒ ('*Var*,'*Val*) *Mem* ⇒ *bool*
where *globally-consistent A mds mem*$_1$ *mem*$_2$ ≡
(∀ *x*. *case A x of Some* (*v*,*v'*) ⇒ (*mem*$_1$ *x* ≠ *v* ∨ *mem*$_2$ *x* ≠ *v'*) ⟶ ¬ *var-asm-not-written mds x* | - ⇒ *True*) ∧

$(\forall\, x.\ dma\ mem_1\ [\|_1\ A]\ x \neq dma\ mem_1\ x \longrightarrow \neg\ var\text{-}asm\text{-}not\text{-}written\ mds\ x)$
$\wedge$
$(\forall\, x.\ dma\ (mem_1\ [\|_1\ A])\ x = Low \wedge (x \notin mds\ AsmNoReadOrWrite \vee x \in \mathcal{C}) \longrightarrow (mem_1\ [\|_1\ A])\ x = (mem_2\ [\|_2\ A])\ x)$

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 $\langle c_1,\ mds,\ mem_1\ [\|_1\ A]\rangle \approx \langle c_2,\ mds,\ mem_2\ [\|_2\ A]\rangle$
⟨*proof*⟩

lemma *mm-equiv-*$\mathcal{C}$*-eq*:
$(a,b) \approx (a',b') \Longrightarrow snd\ a = snd\ a' \Longrightarrow$
$\forall\, x{\in}\mathcal{C}.\ b\ x = b'\ x$
⟨*proof*⟩

lemma *apply-adaptation-not-in-dom*:
$x \notin dom\ A \Longrightarrow apply\text{-}adaptation\ b\ blah\ A\ x = blah\ x$
⟨*proof*⟩

lemma *makes-compatible-invariant*:
assumes *sound-modes*: *sound-mode-use* $(cms_1,\ mem_1)$
sound-mode-use $(cms_2,\ mem_2)$
assumes *compat*: *makes-compatible* $(cms_1,\ mem_1)\ (cms_2,\ mem_2)\ mems$
assumes *modes-eq*: *map snd* $cms_1 =$ *map snd* cms_2
assumes *eval*: $(cms_1,\ mem_1) \rightsquigarrow_k (cms_1',\ mem_1')$
obtains $cms_2'\ mem_2'\ mems'$ **where**
map snd $cms_1' =$ *map snd* $cms_2' \wedge$
$(cms_2,\ mem_2) \rightsquigarrow_k (cms_2',\ mem_2') \wedge$
makes-compatible $(cms_1',\ mem_1')\ (cms_2',\ mem_2')\ mems'$
⟨*proof*⟩

The Isar proof language provides a readable way of specifying assumptions while also giving them names for subsequent usage.

lemma *compat-low-eq*:
assumes *compat*: *makes-compatible* $(cms_1,\ mem_1)\ (cms_2,\ mem_2)\ mems$
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 $mem_1\ x = mem_2\ x$
⟨*proof*⟩

lemma *loc-reach-subset*:
assumes *eval*: $\langle c,\ mds,\ mem\rangle \rightsquigarrow \langle c',\ mds',\ mem'\rangle$
shows *loc-reach* $\langle c',\ mds',\ mem'\rangle \subseteq$ *loc-reach* $\langle c,\ mds,\ mem\rangle$
⟨*proof*⟩

lemma *locally-sound-modes-invariant*:
assumes *sound-modes*: *locally-sound-mode-use* $\langle c,\ mds,\ mem\rangle$
assumes *eval*: $\langle c,\ mds,\ mem\rangle \rightsquigarrow \langle c',\ mds',\ mem'\rangle$
shows *locally-sound-mode-use* $\langle c',\ mds',\ mem'\rangle$
⟨*proof*⟩

lemma *meval-sched-one*:
$(cms,\ mem) \rightsquigarrow_k (cms',\ mem') \Longrightarrow$
$(cms,\ mem) \rightarrow_{[k]} (cms',\ mem')$
⟨*proof*⟩

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

lemma *reachable-modes-subset*:
assumes *eval*: $(cms,\ mem) \rightsquigarrow_k (cms',\ mem')$
shows *reachable-mode-states* $(cms',\ mem') \subseteq$ *reachable-mode-states* $(cms,\ mem)$
⟨*proof*⟩

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* $(cms',\ mem')$
⟨*proof*⟩

lemma *loc-reach-mem-diff-subset*:
assumes *mem-diff*: $\forall\ x.$ *var-asm-not-written mds* $x \longrightarrow mem_1\ x = mem_2\ x \wedge$ *dma* $mem_1\ x =$ *dma* $mem_2\ x$
shows $\langle c',\ mds',\ mem'\rangle \in$ *loc-reach* $\langle c,\ mds,\ mem_1\rangle \Longrightarrow \langle c',\ mds',\ mem'\rangle \in$ *loc-reach* $\langle c,\ mds,\ mem_2\rangle$
⟨*proof*⟩

lemma *loc-reach-mem-diff-eq*:
assumes *mem-diff*: $\forall\ x.$ *var-asm-not-written mds* $x \longrightarrow mem'\ x = mem\ x \wedge$ *dma* $mem'\ x =$ *dma mem* x
shows *loc-reach* $\langle c,\ mds,\ mem\rangle =$ *loc-reach* $\langle c,\ mds,\ mem'\rangle$
⟨*proof*⟩

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

lemma *app-Cons-rewrite*:

$ns \mathbin{@} (a \# ms) = ((ns \mathbin{@} [a]) \mathbin{@} ms)$
⟨*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*:
$(cms_1, mem_1) \rightarrow_{ns@[n]} (cms_1', mem_1') \Longrightarrow$
$\exists\, cms_1''\ mem_1''.\ (cms_1, mem_1) \rightarrow_{ns} (cms_1'', mem_1'') \wedge (cms_1'', mem_1'') \rightsquigarrow_n (cms_1', mem_1')$
⟨*proof*⟩

lemma *meval-sched-snocI*:
$(cms_1, mem_1) \rightarrow_{ns} (cms_1'', mem_1'') \wedge (cms_1'', mem_1'') \rightsquigarrow_n (cms_1', mem_1') \Longrightarrow$
$(cms_1, mem_1) \rightarrow_{ns@[n]} (cms_1', mem_1')$
⟨*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 = *map snd* cms_2
assumes *sound-modes*: *sound-mode-use* (cms_1, mem_1) *sound-mode-use* (cms_2, mem_2)
assumes *eval*: $(cms_1, mem_1) \rightarrow_{sched} (cms_1', mem_1')$
shows $\exists$ $cms_2'\ mem_2'\ mems'$. *sound-mode-use* $(cms_1', mem_1') \wedge$
sound-mode-use $(cms_2', mem_2') \wedge$
map snd $cms_1' =$ *map snd* $cms_2' \wedge$
$(cms_2, mem_2) \rightarrow_{sched} (cms_2', mem_2') \wedge$
makes-compatible (cms_1', mem_1') (cms_2', mem_2') $mems'$
⟨*proof*⟩

lemma *differing-vars-initially-empty*:
$i < n \Longrightarrow x \notin$ *differing-vars-lists* mem_1 mem_2 (*zip* (*replicate* n mem_1) (*replicate* n mem_2)) i
⟨*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)
(*replicate* (*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 $eval_A$ :: (*'Var*, *'Val*) *Mem* ⇒ *'AExp* ⇒ *'Val*
fixes $eval_B$ :: (*'Var*, *'Val*) *Mem* ⇒ *'BExp* ⇒ *bool*
fixes *aexp-vars* :: *'AExp* ⇒ *'Var set*
fixes *bexp-vars* :: *'BExp* ⇒ *'Var set*
assumes *Var-finite* : *finite* {(*x* :: *'Var*). *True*}
assumes *eval-vars-det*$_A$: ⟦ $\forall$ *x* ∈ *aexp-vars e*. $mem_1\ x = mem_2\ x$ ⟧ ⟹ $eval_A$ $mem_1\ e = eval_A\ mem_2\ e$
assumes *eval-vars-det*$_B$: ⟦ $\forall$ *x* ∈ *bexp-vars b*. $mem_1\ x = mem_2\ x$ ⟧ ⟹ $eval_B$ $mem_1\ b = eval_B\ mem_2\ b$

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

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

context *sifum-lang-no-dma*
begin

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

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

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

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

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

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

4.2 Semantics

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

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

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

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

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

lemma *cond*:
((*If b t e*, *mem*), (*if* $eval_B$ *mem b then t else e*, *mem*)) ∈ $eval_w$*-simple*
⟨*proof*⟩

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

lemma *rtrancl-mono-proof*[*mono*]:
(⋀*a b*. *x a b* ⟶ *y a b*) ⟹ *rtranclp x a b* ⟶ *rtranclp y a b*
⟨*proof*⟩

lemma *trancl-mono-proof*[*mono*]:
(⋀*a b*. *x a b* ⟶ *y a b*) ⟹ *tranclp x a b* ⟶ *tranclp y a b*
⟨*proof*⟩

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

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

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

(**infixr** $\rightsquigarrow_w$ *60*)
where
$c \rightsquigarrow_w c' \equiv (c, c') \in eval_w$ |
unannotated: $[\![(c, mem) \rightsquigarrow_s (c', mem')]\!]$
$\Longrightarrow (\langle cxt\text{-}to\text{-}stmt\ E\ c, mds, mem\rangle_w, \langle cxt\text{-}to\text{-}stmt\ E\ c', mds, mem'\rangle_w) \in eval_w$ |
seq: $[\![\langle c_1, mds, mem\rangle_w \rightsquigarrow_w \langle c_1', mds', mem'\rangle_w]\!] \Longrightarrow (\langle (c_1\ ;;\ c_2), mds, mem\rangle_w, \langle (c_1'\ ;;\ c_2), mds', mem'\rangle_w) \in eval_w$ |
decl: $[\![\langle c, update\text{-}modes\ mu\ mds, mem\rangle_w \rightsquigarrow_w \langle c', mds', mem'\rangle_w]\!] \Longrightarrow$
$(\langle cxt\text{-}to\text{-}stmt\ E\ (ModeDecl\ c\ mu), mds, mem\rangle_w, \langle cxt\text{-}to\text{-}stmt\ E\ c', mds', mem'\rangle_w) \in eval_w$ |

await: $[\![eval_B\ mem\ b; no\text{-}await\ c_1;$
$(\langle c_1, mds, mem\rangle_w, \langle c_2, mds', mem'\rangle_w) \in eval_w{}^+;$
$is\text{-}final\ c_2]\!] \Longrightarrow$
$(\langle Await\ b\ c_1, mds, mem\rangle_w, \langle c_2, mds', mem'\rangle_w) \in eval_w$

abbreviation $eval_w\text{-}plus$ ::
$(('Var, 'AExp, 'BExp)\ Stmt, 'Var, 'Val)\ LocalConf \Rightarrow$
$(('Var, 'AExp, 'BExp)\ Stmt, 'Var, 'Val)\ LocalConf \Rightarrow bool$ (- $\rightsquigarrow_w{}^+$ -) **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 *cxt-inv*:
$[\![cxt\text{-}to\text{-}stmt\ E\ c = c'\ ; \bigwedge p\ q.\ c' \neq Seq\ p\ q]\!] \Longrightarrow E = [] \wedge c' = c$
⟨*proof*⟩

lemma *cxt-inv-assign*:
$[\![cxt\text{-}to\text{-}stmt\ E\ c = x \leftarrow e]\!] \Longrightarrow c = x \leftarrow e \wedge E = []$
⟨*proof*⟩

lemma *cxt-inv-skip*:
$[\![cxt\text{-}to\text{-}stmt\ E\ c = Skip]\!] \Longrightarrow c = Skip \wedge E = []$
⟨*proof*⟩

lemma *cxt-inv-stop*:
$cxt\text{-}to\text{-}stmt\ E\ c = Stop \Longrightarrow c = Stop \wedge E = []$
⟨*proof*⟩

lemma *cxt-inv-if*:
$cxt\text{-}to\text{-}stmt\ E\ c = If\ e\ p\ q \Longrightarrow c = If\ e\ p\ q \wedge E = []$

⟨*proof*⟩

lemma *ctx-inv-anno*:
cxt-to-stmt E c = *c'*@[*mu*] ⟹ *c* = *c'*@[*mu*] ∧ *E* = []
⟨*proof*⟩

lemma *cxt-inv-await*:
cxt-to-stmt E c = *Await e p* ⟹ *c* = *Await e p* ∧ *E* = []
⟨*proof*⟩

lemma *cxt-inv-while*:
cxt-to-stmt E c = *While e p* ⟹ *c* = *While e p* ∧ *E* = []
⟨*proof*⟩

lemma *skip-elim* [*elim*]:
⟨*Skip*, *mds*, *mem*⟩$_w$ ⇝$_w$ ⟨*c'*, *mds'*, *mem'*⟩$_w$ ⟹ *c'* = *Stop* ∧ *mds* = *mds'* ∧ *mem* = *mem'*
⟨*proof*⟩

lemma *assign-elim* [*elim*]:
⟨*x* ← *e*, *mds*, *mem*⟩$_w$ ⇝$_w$ ⟨*c'*, *mds'*, *mem'*⟩$_w$ ⟹ *c'* = *Stop* ∧ *mds* = *mds'* ∧ *mem'* = *mem* (*x* := *eval*$_A$ *mem e*)
⟨*proof*⟩

inductive-cases *if-elim'* [*elim*!]: (*If b p q*, *mem*) ⇝$_s$ (*c'*, *mem'*)

lemma *if-elim* [*elim*]:
⋀ *P*.
⟦ ⟨*If b p q*, *mds*, *mem*⟩$_w$ ⇝$_w$ ⟨*c'*, *mds'*, *mem'*⟩$_w$;
⟦ *c'* = *p*; *mem'* = *mem* ; *mds'* = *mds* ; *eval*$_B$ *mem b* ⟧ ⟹ *P* ;
⟦ *c'* = *q*; *mem'* = *mem* ; *mds'* = *mds* ; ¬ *eval*$_B$ *mem b* ⟧ ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

inductive-cases *await-elim'* [*elim*!]: ⟨*Await b p*, *mds*, *mem*⟩$_w$ ⇝$_w$ ⟨*c'*,*mds'*, *mem'*⟩$_w$

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

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

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

lemma *upd-elim* [*elim*]:
⟨*c*@[*upd*], *mds*, *mem*⟩$_w$ ⇝$_w$ ⟨*c'*, *mds'*, *mem'*⟩$_w$ ⟹ ⟨*c*, *update-modes upd mds*, *mem*⟩$_w$ ⇝$_w$ ⟨*c'*, *mds'*, *mem'*⟩$_w$
⟨*proof*⟩

lemma *cxt-seq-elim* [*elim*]:
c_1 ;; c_2 = *cxt-to-stmt* E c $\Longrightarrow$ (E = [] $\wedge$ c = c_1 ;; c_2) $\vee$ ($\exists$ c' *cs*. E = c' # *cs* $\wedge$ c = c_1 $\wedge$ c_2 = *cxt-to-stmt cs* c')
⟨*proof*⟩

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

lemma *stop-no-eval*: ¬ (⟨*Stop*, *mds*, *mem*⟩$_w$ $\rightsquigarrow_w$ ⟨c', *mds'*, *mem'*⟩$_w$)
⟨*proof*⟩

lemma *seq-stop-elim* [*elim*]:
⟨*Stop* ;; c, *mds*, *mem*⟩$_w$ $\rightsquigarrow_w$ ⟨c', *mds'*, *mem'*⟩$_w$ $\Longrightarrow$ c' = c $\wedge$ *mds'* = *mds* $\wedge$ *mem'* = *mem*
⟨*proof*⟩

lemma *cxt-stmt-seq*:
c ;; *cxt-to-stmt* E c' = *cxt-to-stmt* (c' # E) c
⟨*proof*⟩

lemma *seq-elim* [*elim*]:
⟦ ⟨c_1 ;; c_2, *mds*, *mem*⟩$_w$ $\rightsquigarrow_w$ ⟨c', *mds'*, *mem'*⟩$_w$; c_1 ≠ *Stop* ⟧ $\Longrightarrow$
($\exists$ c_1'. ⟨c_1, *mds*, *mem*⟩$_w$ $\rightsquigarrow_w$ ⟨c_1', *mds'*, *mem'*⟩$_w$ $\wedge$ c' = c_1' ;; c_2)
⟨*proof*⟩

lemma *stop-cxt*: *Stop* = *cxt-to-stmt* E c $\Longrightarrow$ c = *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*: *mem'* = *mem*
assumes *upd*: *mds'* = *update-modes upd mds*
shows (⟨*Skip*@[*upd*], *mds*, *mem*⟩$_w$, ⟨*Stop*, *mds'*, *mem'*⟩$_w$) ∈ *eval*$_w$
⟨*proof*⟩

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

lemma *seq-decl-elim*:
$\langle (Skip@[upd]) ;; c, mds, mem\rangle_w \rightsquigarrow_w \langle c', mds', mem'\rangle_w \Longrightarrow$
$c' = Stop ;; c \wedge mem' = mem \wedge mds' = update\text{-}modes\ upd\ mds$
⟨*proof*⟩

lemma *seq-assign-elim*:
$\langle (x \leftarrow e) ;; c, mds, mem\rangle_w \rightsquigarrow_w \langle c', mds', mem'\rangle_w \Longrightarrow$
$c' = Stop ;; c \wedge mds' = mds \wedge mem' = mem(x := eval_A\ mem\ e)$
⟨*proof*⟩

lemma *no-await-trans*:
$[\![no\text{-}await\ c; \langle c, mds, mem\rangle_w \rightsquigarrow_w \langle c', mds', mem'\rangle_w]\!] \Longrightarrow no\text{-}await\ c'$
⟨*proof*⟩

lemma *no-await-no-await*[*elim*]: $[\![no\text{-}await\ c]\!] \Longrightarrow c \neq Await\ b\ c'$
⟨*proof*⟩

lemma *no-await-trancl-impl*:
$[\![ctx \rightsquigarrow_w{}^+\ ctx']\!] \Longrightarrow no\text{-}await\ (fst\ (fst\ ctx)) \longrightarrow no\text{-}await\ (fst\ (fst\ ctx'))$
⟨*proof*⟩

lemma *no-await-trancl*:
$[\![ctx \rightsquigarrow_w{}^+\ ctx'; no\text{-}await\ (fst\ (fst\ ctx))]\!] \Longrightarrow no\text{-}await\ (fst\ (fst\ ctx'))$
⟨*proof*⟩

lemma *await-elim*:
$[\![\langle Await\ b\ c_1, mds, mem\rangle_w \rightsquigarrow_w \langle c_2, mds', mem'\rangle_w]\!] \Longrightarrow$
$eval_B\ mem\ b \wedge no\text{-}await\ c_1 \wedge is\text{-}final\ c_2\ \wedge$
$\langle c_1, mds, mem\rangle_w \rightsquigarrow_w{}^+ \langle c_2, mds', mem'\rangle_w$
⟨*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 ⇀ '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 × 'Var set)*

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

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

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

type-synonym *'bexp preds* = *'bexp set*

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

definition
pred :: *'BExp preds* ⇒ (*'Var*,*'Val*) *Mem* ⇒ *bool*
where
pred P ≡ λ*mem*. (∀ *p*∈*P*. $eval_B$ *mem p*)

end

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

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

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

locale *sifum-types-assign* =

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

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

fixes *bexp-assign* :: *'Var* ⇒ *'AExp* ⇒ *'BExp*
assumes *bexp-assign-eval*: ⋀*mem e x*. (ev_B *mem* (*bexp-assign x e*)) = (*mem x* = (ev_A *mem e*))
assumes *bexp-assign-vars*: ⋀*e x*. (*bexp-vars* (*bexp-assign x e*)) = *aexp-vars e* ∪ {*x*}

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

definition
stable :: *'Var Stable* ⇒ *'Var* ⇒ *bool*
where
stable $\mathcal{S}$ *x* ≡ *x* ∈ (*fst* $\mathcal{S}$ ∪ *snd* $\mathcal{S}$)

definition
add-pred :: *'BExp preds* ⇒ *'Var Stable* ⇒ *'BExp* ⇒ *'BExp preds* (- +- - [*120*, *120*, *120*] *1000*)
where
P +$_{\mathcal{S}}$ *e* ≡ (*if* (∀ *x*∈*bexp-vars e*. *stable* $\mathcal{S}$ *x*) *then P* ∪ {*e*} *else P*)

lemma *add-pred-subset*:
P ⊆ *P* +$_{\mathcal{S}}$ *p*
⟨*proof*⟩

definition

restrict-preds-to-vars :: *'BExp preds* ⇒ *'Var set* ⇒ *'BExp preds* (- |‘ - [*120*, *120*] *1000*)
where
P |‘ *V* ≡ {*e*. *e* ∈ *P* ∧ *bexp-vars e* ⊆ *V*}

end

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

the most simple assignment postcondition transformer

definition
assign-post :: *'BExp preds* ⇒ *'Var* ⇒ *'AExp* ⇒ *'BExp preds*
where
assign-post P x e ≡
(*if x* ∈ (*aexp-vars e*) *then*
(*restrict-preds-to-vars P* (−{*x*}))
else
(*restrict-preds-to-vars P* (−{*x*})) ∪ {*bexp-assign x e*})

end

sublocale *sifum-types-assign* ⊆ *sifum-types* - - - - - - - - *assign-post*
⟨*proof*⟩

context *sifum-types*
begin

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

abbreviation
eval-abv2 :: (-, *'Var*, *'Val*) *LocalConf* ⇒ (-, -, -) *LocalConf* ⇒ *bool*
(**infixl** ⇝ *70*)
where
x ⇝ *y* ≡ (*x*, *y*) ∈ $eval_w$

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

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

where
$x \rightsquigarrow \bot \equiv \forall\ y.\ (x,\ y) \notin eval_w$

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

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

definition
type-wellformed :: *'BExp Type* ⇒ *bool*
where
type-wellformed t ≡ *vars-of-type t* ⊆ $\mathcal{C}$

lemma *dma-type-wellformed* [*simp*]:
type-wellformed (*dma-type x*)
⟨*proof*⟩

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

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

lemma *to-total-type-wellformed*:
types-wellformed Γ ⟹
type-wellformed (*to-total* Γ *x*)
⟨*proof*⟩

lemma *Un-type-wellformed*:
∀ *t*∈*ts*. *type-wellformed t* ⟹ *type-wellformed* (⋃ *ts*)
⟨*proof*⟩

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

lemma *type-aexprI*:
$t = \bigcup (image\ (\lambda\ x.\ to\text{-}total\ \Gamma\ x)\ (aexp\text{-}vars\ e)) \Longrightarrow \Gamma \vdash_a e \in t$
⟨*proof*⟩

lemma *type-aexpr-type-wellformed*:
types-wellformed $\Gamma \Longrightarrow \Gamma \vdash_a e \in t \Longrightarrow$ *type-wellformed* t
⟨*proof*⟩

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

inductive
type-bexpr :: (*'Var*,*'BExp*) *TyEnv* ⇒ *'BExp* ⇒ *'BExp Type* ⇒ *bool* (- ⊢$_b$ - ∈ - [*120*, *120*, *120*] *1000*)
where
type-bexpr [*intro*!]: $\Gamma \vdash_b e \in \bigcup (image\ (\lambda\ x.\ to\text{-}total\ \Gamma\ x)\ (bexp\text{-}vars\ e))$

lemma *type-bexprI*:
$t = \bigcup (image\ (\lambda\ x.\ to\text{-}total\ \Gamma\ x)\ (bexp\text{-}vars\ e)) \Longrightarrow \Gamma \vdash_b e \in t$
⟨*proof*⟩

lemma *type-bexpr-type-wellformed*:
types-wellformed $\Gamma \Longrightarrow \Gamma \vdash_b e \in t \Longrightarrow$ *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
pred-stable :: *'Var Stable* ⇒ *'BExp* ⇒ *bool*
where
pred-stable $\mathcal{S}$ $p \equiv \forall\, x{\in}bexp\text{-}vars\ p.\ stable\ \mathcal{S}\ x$

abbreviation
type-stable :: *'Var Stable* ⇒ *'BExp Type* ⇒ *bool*
where
type-stable $\mathcal{S}$ $t \equiv (\forall\, p{\in}t.\ pred\text{-}stable\ \mathcal{S}\ p)$

lemma *type-stable-is-sufficient*:
⟦*type-stable* $\mathcal{S}$ t⟧ ⟹

$\forall\, mem\ mem'.\ (\forall\, x.\ stable\ \mathcal{S}\ x \longrightarrow mem\ x = mem'\ x) \longrightarrow (ev_B\ mem)\ `\ t = (ev_B\ mem')\ `\ t$
⟨*proof*⟩

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

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

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

lemma *add-anno-acq-AsmNoReadOrWrite-idemp* [*simp*]:
v ∈ *dom* Γ ∨ *v* ∈ $\mathcal{C}$ ⟹ Γ ⊕$_{\mathcal{S}}$ (*Acq v AsmNoReadOrWrite*) = Γ
⟨*proof*⟩

lemma *add-anno-rel-AsmNoReadOrWrite-idemp* [*simp*]:
⟦*v* ∉ *dom* Γ; *v* ∉ *fst* $\mathcal{S}$⟧ ⟹ Γ ⊕$_{\mathcal{S}}$ (*Rel v AsmNoReadOrWrite*) = Γ
⟨*proof*⟩

lemma *add-anno-acq-AsmNoReadOrWrite* [*simp*]:
assumes *notin* [*simp*]: *v* ∉ *dom* Γ
shows *v* ∉ $\mathcal{C}$ ⟹ Γ ⊕$_{\mathcal{S}}$ (*Acq v AsmNoReadOrWrite*) = (Γ(*v* ↦ *dma-type v*))
⟨*proof*⟩

lemma *add-anno-rel-AsmNoReadOrWrite* [*simp*]:
assumes *isin* [*simp*]: *v* ∈ *dom* Γ

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

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

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

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

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

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

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

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

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

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

type-max t mem ≡ *if* (∀ *p*∈*t*. *ev*$_B$ *mem p*) *then Low else High*

lemma *type-stable-is-sufficient'*:
⟦*type-stable* $\mathcal{S}$ *t*⟧ ⟹
∀ *mem mem'*. (∀ *x*. *stable* $\mathcal{S}$ *x* ⟶ *mem x* = *mem' x*) ⟶ *type-max t mem* = *type-max t mem'*
⟨*proof*⟩

lemma *subtype-sound*:
t ≤:$_P$ *t'* ⟹ ∀ *mem*. *pred P mem* ⟶ *type-max t mem* ≤ *type-max t' mem*
⟨*proof*⟩

lemma *subtype-complete*:
assumes *a*: ⋀*mem*. *pred P mem* ⟹ *type-max t mem* ≤ *type-max t' mem*
shows *t* ≤:$_P$ *t'*
⟨*proof*⟩

lemma *subtype-correct*:
(*t* ≤:$_P$ *t'*) = (∀ *mem*. *pred P mem* ⟶ *type-max t mem* ≤ *type-max t' mem*)
⟨*proof*⟩

definition
type-equiv :: *'BExp Type* ⇒ *'BExp preds* ⇒ *'BExp Type* ⇒ *bool* (- =:_ - [*120*, *120*, *120*] *1000*)
where
t =:$_P$ *t'* ≡ *t* ≤:$_P$ *t'* ∧ *t'* ≤:$_P$ *t*

lemma *subtype-refl* [*simp*]:
t ≤:$_P$ *t*
⟨*proof*⟩

lemma *type-equiv-refl* [*simp*]:
t =:$_P$ *t*
⟨*proof*⟩

definition
anno-type-stable :: (*'Var*,*'BExp*) *TyEnv* ⇒ *'Var Stable* ⇒ *'Var ModeUpd* ⇒ *bool*
where
anno-type-stable Γ $\mathcal{S}$ *upd* ≡ (*case upd of* (*Rel v m*) ⇒
(*v* ∈ $\mathcal{C}$ ∧ *v* ∉ *add-anno-dom* Γ $\mathcal{S}$ *upd*) ⟶
(∀ *x*∈*dom* Γ. *v* ∉ *vars-of-type* (*the* (Γ *x*)))
| (*Acq v m*) ⇒
(*v* ∉ $\mathcal{C}$ ∧ *v*∈*add-anno-dom* Γ $\mathcal{S}$ *upd* − *dom* Γ) ⟶
(∀ *x*∈$\mathcal{C}$*-vars v*. *stable* $\mathcal{S}$ *x*))

definition
anno-type-sec :: (*'Var*,*'BExp*) *TyEnv* ⇒ *'Var Stable* ⇒ *'BExp preds* ⇒ *'Var ModeUpd* ⇒ *bool*

where
anno-type-sec Γ $\mathcal{S}$ P *upd* $\equiv$ (*case upd of* (*Rel v AsmNoReadOrWrite*) $\Rightarrow$
($v \in$ *add-anno-dom* Γ $\mathcal{S}$ *upd* $\longrightarrow$ (*the* (Γ v)) $\leq:_P$ (*dma-type* v))
| - $\Rightarrow$ *True*)

definition
types-stable :: (*'Var*,*'BExp*) *TyEnv* $\Rightarrow$ *'Var Stable* $\Rightarrow$ *bool*
where
types-stable Γ $\mathcal{S} \equiv \forall\, x{\in}dom\ \Gamma.$ *type-stable* $\mathcal{S}$ (*the* (Γ x))

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

lemma *subset-entailment*:
$P' \subseteq P \Longrightarrow P \vdash P'$
⟨*proof*⟩

lemma *pred-entailment-refl* [*simp*]:
$P \vdash P$
⟨*proof*⟩

lemma *pred-entailment-mono*:
$P \vdash P' \Longrightarrow P \subseteq P'' \Longrightarrow P'' \vdash P'$
⟨*proof*⟩

lemma *type-equiv-subset*:
type-equiv t P t' $\Longrightarrow P \subseteq P' \Longrightarrow$ *type-equiv t P' t'*
⟨*proof*⟩

definition
context-equiv :: (*'Var*,*'BExp*) *TyEnv* $\Rightarrow$ *'BExp preds* $\Rightarrow$ (*'Var*,*'BExp*) *TyEnv* $\Rightarrow$ *bool* (- =:_ - [*120*, *120*, *120*] *1000*)
where
$\Gamma =:_P \Gamma' \equiv$ *dom* Γ = *dom* $\Gamma' \wedge$
($\forall\, x{\in}dom\ \Gamma'.$ *type-equiv* (*the* (Γ x)) P (*the* (Γ' x)))

lemma *context-equiv-refl*[*simp*]:
context-equiv Γ P Γ
⟨*proof*⟩

lemma *context-equiv-subset*:
context-equiv Γ P $\Gamma' \Longrightarrow P \subseteq P' \Longrightarrow$ *context-equiv* Γ P' Γ'
⟨*proof*⟩

lemma *pred-entailment-trans*:
$P \vdash P' \Longrightarrow P' \vdash P'' \Longrightarrow P \vdash P''$
⟨*proof*⟩

lemma *pred-un* [*simp*]:
pred $(P \cup P')$ *mem* = (*pred* P *mem* $\wedge$ *pred* P' *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*:
type-equiv t P $t' \Longrightarrow$ *type-equiv* t' P' $t'' \Longrightarrow P \vdash P' \Longrightarrow$ *type-equiv* t P t''
⟨*proof*⟩

lemma *context-equiv-trans*:
context-equiv Γ P $\Gamma' \Longrightarrow$ *context-equiv* Γ' P' $\Gamma'' \Longrightarrow P \vdash P' \Longrightarrow$ *context-equiv* Γ P Γ''
⟨*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*:
type-equiv t P $t' \Longrightarrow P' \vdash P \Longrightarrow$ *type-equiv* t P' t'
⟨*proof*⟩

lemma *context-equiv-entailment*:
context-equiv Γ P $\Gamma' \Longrightarrow P' \vdash P \Longrightarrow$ *context-equiv* Γ P' Γ'
⟨*proof*⟩

inductive
has-type :: (*'Var*,*'BExp*) *TyEnv* ⇒ *'Var Stable* ⇒ *'BExp preds* ⇒ (*'Var*, *'AExp*, *'BExp*) *Stmt* ⇒ (*'Var*,*'BExp*) *TyEnv* ⇒ *'Var Stable* ⇒ *'BExp preds* ⇒ *bool*
(⊢ -,-,- {-} -,-,- [*120*, *120*, *120*, *120*, *120*, *120*, *120*] *1000*)
where
stop-type [*intro*]: ⊢ Γ,$\mathcal{S}$,*P* {*Stop*} Γ,$\mathcal{S}$,*P* |
skip-type [*intro*] : ⊢ Γ,$\mathcal{S}$,*P* {*Skip*} Γ,$\mathcal{S}$,*P* |
$assign_{\mathcal{C}}$:
⟦$x \in \mathcal{C}$; Γ ⊢$_a$ *e* ∈ *t*; *P* ⊢ *t*; (∀ *v*∈*dom* Γ. *x* ∉ *vars-of-type* (*the* (Γ *v*)));
P' = *restrict-preds-to-vars* (*assign-post P x e*) {*v*. *stable* $\mathcal{S}$ *v*};
∀ *v*. *x* ∈ $\mathcal{C}$-*vars v* ∧ *v* ∉ *snd* $\mathcal{S}$ ⟶ *P* ⊢ (*to-total* Γ *v*) ∧
(*to-total* Γ *v*) ≤:$_P$ *'* (*dma-type v*) ⟧ ⟹
⊢ Γ,$\mathcal{S}$,*P* {*x* ← *e*} Γ,$\mathcal{S}$,*P'* |
$assign_1$:
⟦ *x* ∉ *dom* Γ ; *x* ∉ $\mathcal{C}$; Γ ⊢$_a$ *e* ∈ *t*; *t* ≤:$_P$ (*dma-type x*);
P' = *restrict-preds-to-vars* (*assign-post P x e*) {*v*. *stable* $\mathcal{S}$ *v*} ⟧ ⟹
⊢ Γ,$\mathcal{S}$,*P* {*x* ← *e*} Γ,$\mathcal{S}$,*P'* |
$assign_2$:
⟦ *x* ∈ *dom* Γ ; Γ ⊢$_a$ *e* ∈ *t*; *type-stable* $\mathcal{S}$ *t*; *P'* = *restrict-preds-to-vars* (*assign-post P x e*) {*v*. *stable* $\mathcal{S}$ *v*};
x ∉ *snd* $\mathcal{S}$ ⟶ *t* ≤:$_P$ *'* (*dma-type x*) ⟧ ⟹
has-type Γ $\mathcal{S}$ *P* (*x* ← *e*) (Γ (*x* := *Some t*)) $\mathcal{S}$ *P'* |
if-type [*intro*]:
⟦ Γ ⊢$_b$ *e* ∈ *t*; *P* ⊢ *t*;
⊢ Γ,$\mathcal{S}$,(*P* +$_{\mathcal{S}}$ *e*) { *c*$_1$ } Γ',$\mathcal{S}$',*P'*; ⊢ Γ,$\mathcal{S}$,(*P* +$_{\mathcal{S}}$ (*bexp-neg e*)) { *c*$_2$ } Γ'',$\mathcal{S}$',*P''*;
context-equiv Γ' *P'* Γ'''; *context-equiv* Γ'' *P''* Γ'''; *P'* ⊢ *P'''*; *P''* ⊢ *P'''*;
∀ *mds*. *tyenv-wellformed mds* Γ' $\mathcal{S}$' *P'* ⟶ *tyenv-wellformed mds* Γ''' $\mathcal{S}$' *P'''*;
∀ *mds*. *tyenv-wellformed mds* Γ'' $\mathcal{S}$' *P''* ⟶ *tyenv-wellformed mds* Γ''' $\mathcal{S}$' *P'''* ⟧ ⟹
⊢ Γ,$\mathcal{S}$,*P* { *If e c*$_1$ *c*$_2$ } Γ''',$\mathcal{S}$',*P'''* |
while-type [*intro*]: ⟦ Γ ⊢$_b$ *e* ∈ *t* ; *P* ⊢ *t*; ⊢ Γ,$\mathcal{S}$,(*P* +$_{\mathcal{S}}$ *e*) { *c* } Γ,$\mathcal{S}$,*P* ⟧ ⟹ ⊢ Γ,$\mathcal{S}$,*P* { *While e c* } Γ,$\mathcal{S}$,*P* |
anno-type [*intro*]: ⟦ Γ' = Γ ⊕$_{\mathcal{S}}$ *upd* ; $\mathcal{S}$' = *add-anno-stable* $\mathcal{S}$ *upd*; *P'* = *restrict-preds-to-vars P* {*v*. *stable* $\mathcal{S}$' *v*};
⊢ Γ',$\mathcal{S}$',*P'* { *c* } Γ'',$\mathcal{S}$'',*P''* ; *c* ≠ *Stop* ;
(⋀ *x*. (*to-total* Γ *x*) ≤:$_P$ *'* (*to-total* Γ' *x*));
anno-type-stable Γ $\mathcal{S}$ *upd*; *anno-type-sec* Γ $\mathcal{S}$ *P upd*⟧ ⟹ ⊢ Γ,$\mathcal{S}$,*P* { *c*@[*upd*] } Γ'',$\mathcal{S}$'',*P''* |
seq-type [*intro*]: ⟦ ⊢ Γ,$\mathcal{S}$,*P* { *c*$_1$ } Γ',$\mathcal{S}$',*P'* ; ⊢ Γ',$\mathcal{S}$',*P'* { *c*$_2$ } Γ'',$\mathcal{S}$'',*P''* ⟧ ⟹ ⊢ Γ,$\mathcal{S}$,*P* { *c*$_1$;; *c*$_2$ } Γ'',$\mathcal{S}$'',*P''* |
sub : ⟦ ⊢ Γ$_1$,$\mathcal{S}$,*P*$_1$ { *c* } Γ$_1$',$\mathcal{S}$',*P*$_1$' ; *context-equiv* Γ$_2$ *P*$_2$ Γ$_1$; (∀ *mds*. *tyenv-wellformed mds* Γ$_2$ $\mathcal{S}$ *P*$_2$ ⟶ *tyenv-wellformed mds* Γ$_1$ $\mathcal{S}$ *P*$_1$);
(∀ *mds*. *tyenv-wellformed mds* Γ$_1$' $\mathcal{S}$' *P*$_1$' ⟶ *tyenv-wellformed mds* Γ$_2$' $\mathcal{S}$' *P*$_2$'); *context-equiv* Γ$_1$' *P*$_1$' Γ$_2$'; *P*$_2$ ⊢ *P*$_1$; *P*$_1$' ⊢ *P*$_2$' ⟧ ⟹ ⊢ Γ$_2$,$\mathcal{S}$,*P*$_2$ { *c* } Γ$_2$',$\mathcal{S}$',*P*$_2$' |
await-type [*intro*]:
⟦ Γ ⊢$_b$ *e* ∈ *t*; *P* ⊢ *t*;
⊢ Γ,$\mathcal{S}$,(*P* +$_{\mathcal{S}}$ *e*) { *c* } Γ',$\mathcal{S}$',*P'* ⟧ ⟹

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

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

lemma *assign$_2$-helper*:
$[\![\Gamma\ x = \mathit{Some}\ t;\ \mathit{has\text{-}type}\ \Gamma\ \mathcal{S}\ P\ (x \leftarrow e)\ (\Gamma(x \mapsto t))\ \mathcal{S}\ P']\!] \Longrightarrow \mathit{has\text{-}type}\ \Gamma\ \mathcal{S}\ P\ (x \leftarrow e)\ \Gamma\ \mathcal{S}\ P'$
⟨*proof*⟩

lemma *conc'*:
$[\![\vdash \Gamma_1,\mathcal{S},P \{ c \} \Gamma',\mathcal{S}',P';$
$\Gamma_1 = (\Gamma_2(x \mapsto t));$
$x \in \mathit{dom}\ \Gamma_2;$
$\mathit{type\text{-}equiv}\ (\mathit{the}\ (\Gamma_2\ x))\ P\ t;$
$\mathit{type\text{-}wellformed}\ t;$
$\mathit{type\text{-}stable}\ \mathcal{S}\ t\ ]\!] \Longrightarrow$
$\vdash \Gamma_2,\mathcal{S},P \{ c \} \Gamma',\mathcal{S}',P'$
⟨*proof*⟩

lemma *tyenv-wellformed-subset*:
$\mathit{tyenv\text{-}wellformed}\ \mathit{mds}\ \Gamma\ \mathcal{S}\ P \Longrightarrow P' \subseteq P \Longrightarrow \mathit{tyenv\text{-}wellformed}\ \mathit{mds}\ \Gamma\ \mathcal{S}\ P'$
⟨*proof*⟩

lemma *if-type'*:
$[\![\Gamma \vdash_b e \in t;$
$P \vdash t;$
$\vdash \Gamma,\mathcal{S},(P +_{\mathcal{S}} e) \{ c_1 \} \Gamma',\mathcal{S}',P';$
$\vdash \Gamma,\mathcal{S},(P +_{\mathcal{S}} (\mathit{bexp\text{-}neg}\ e)) \{ c_2 \} \Gamma',\mathcal{S}',P'';$
$P''' \subseteq P' \cap P'']\!] \Longrightarrow$
$\vdash \Gamma,\mathcal{S},P \{ \mathit{If}\ e\ c_1\ c_2 \} \Gamma',\mathcal{S}',P'''$
⟨*proof*⟩

lemma *skip-type'*:
$[\![\Gamma = \Gamma';\ \mathcal{S} = \mathcal{S}';\ P = P']\!] \Longrightarrow \vdash \Gamma,\mathcal{S},P \{\mathit{Skip}\} \Gamma',\mathcal{S}',P'$
⟨*proof*⟩

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

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

lemma *anno-type-helpers* [*simp*]:
(*to-total* Γ x) $\leq:_P$ (*to-total* (*add-anno* Γ $\mathcal{S}$ (*buffer* $+=_m$ *AsmNoWrite*)) x)
(*to-total* Γ x) $\leq:_P$ (*to-total* (*add-anno* Γ $\mathcal{S}$ (*buffer* $+=_m$ *AsmNoReadOrWrite*)) x)
⟨*proof*⟩

5.2 Typing Soundness

The following predicate is needed to exclude some pathological cases, that abuse the *Stop* command which is not allowed to occur in actual programs.

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

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

lemma *type-max-dma-type* [*simp*]:
type-max (*dma-type* x) *mem* = *dma mem x*
⟨*proof*⟩

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

lemma *tyenv-eq-sym'*:
dom $\Gamma \cap \mathcal{C} = \{\}$ ⟹ *types-wellformed* Γ ⟹ $mem_1 =_\Gamma mem_2$ ⟹ $mem_2 =_\Gamma mem_1$
⟨*proof*⟩

lemma *tyenv-eq-sym*:
tyenv-wellformed mds Γ $\mathcal{S}$ P ⟹ $mem_1 =_\Gamma mem_2$ ⟹ $mem_2 =_\Gamma mem_1$
⟨*proof*⟩

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

intro [*intro*!] : $[\![\vdash \Gamma,\mathcal{S},P \ \{\ c\ \}\ \Gamma',\mathcal{S}',P'$; *tyenv-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-max* (*the* ($\Gamma\ x$)) $mem_1 \leq dma\ mem_1\ x]\!] \Longrightarrow$
$\langle c, mds, mem_1\rangle\ \mathcal{R}^1{}_{\Gamma',\mathcal{S}',P'}\ \langle c, mds, mem_2\rangle$

inductive $\mathcal{R}_3$-*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*] : $[\![\langle c_1, mds, mem_1\rangle\ \mathcal{R}^1{}_{\Gamma,\mathcal{S},P}\ \langle c_2, mds, mem_2\rangle; \vdash \Gamma,\mathcal{S},P\ \{\ c\ \}\ \Gamma',\mathcal{S}',P']\!] \Longrightarrow$
$\langle Seq\ c_1\ c, mds, mem_1\rangle\ \mathcal{R}^3{}_{\Gamma',\mathcal{S}',P'}\ \langle Seq\ c_2\ c, mds, mem_2\rangle$ |
intro$_3$ [*intro*] : $[\![\langle c_1, mds, mem_1\rangle\ \mathcal{R}^3{}_{\Gamma,\mathcal{S},P}\ \langle c_2, mds, mem_2\rangle; \vdash \Gamma,\mathcal{S},P\ \{\ c\ \}\ \Gamma',\mathcal{S}',P']\!] \Longrightarrow$
$\langle Seq\ c_1\ c, mds, mem_1\rangle\ \mathcal{R}^3{}_{\Gamma',\mathcal{S}',P'}\ \langle Seq\ c_2\ c, mds, mem_2\rangle$

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

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

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

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

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

lemma $\mathcal{R}_1$-*mem-eq*: $\langle c_1,\ mds,\ mem_1\rangle\ {\mathcal{R}^1}_{\Gamma',\mathcal{S}',P'}\ \langle c_2,\ mds,\ mem_2\rangle \Longrightarrow mem_1 =_{mds}{}^{l}\ mem_2$
⟨*proof*⟩

lemma $\mathcal{R}_1$-*dma-eq*:
$\langle c_1,\ mds,\ mem_1\rangle\ {\mathcal{R}^1}_{\Gamma',\mathcal{S}',P'}\ \langle c_2,\ mds,\ mem_2\rangle \Longrightarrow dma\ mem_1 = dma\ mem_2$
⟨*proof*⟩

lemma *bisim-simple-*$\mathcal{R}_1$:
$\langle c,\ mds,\ mem\rangle\ {\mathcal{R}^1}_{\Gamma,\mathcal{S},P}\ \langle c',\ mds',\ mem'\rangle \Longrightarrow c = c'$
⟨*proof*⟩

lemma *bisim-simple-*$\mathcal{R}_3$:
$lc\ {\mathcal{R}^3}_{\Gamma,\mathcal{S},P}\ lc' \Longrightarrow (fst\ (fst\ lc)) = (fst\ (fst\ lc'))$
⟨*proof*⟩

lemma *bisim-simple-*$\mathcal{R}_u$:
$lc\ {\mathcal{R}^u}_{\Gamma,\mathcal{S},P}\ lc' \Longrightarrow (fst\ (fst\ lc)) = (fst\ (fst\ lc'))$
⟨*proof*⟩

lemma $\mathcal{C}$-*eq-type-max-eq*:
assumes *wf*: *type-wellformed t*
assumes $\mathcal{C}$-*eq*: $\forall\, x{\in}\mathcal{C}.\ mem_1\ x = mem_2\ x$
shows *type-max t* mem_1 = *type-max t* mem_2
⟨*proof*⟩

lemma *vars-of-type-eq-type-max-eq*:
assumes *mem-eq*: $\forall\, x{\in}$*vars-of-type t*. $mem_1\ x = mem_2\ x$
shows *type-max t* mem_1 = *type-max t* mem_2
⟨*proof*⟩

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

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

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

lemma $\mathcal{R}$*-sym*: *sym* ($\mathcal{R}\ \Gamma\ \mathcal{S}\ P$)
⟨*proof*⟩

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

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

lemma $\mathcal{R}$*-closed-glob-consistent*: *closed-glob-consistent* ($\mathcal{R}\ \Gamma'\ \mathcal{S}'\ P'$)
⟨*proof*⟩

lemma *mode-update-add-anno*:
mds-consistent mds $\Gamma\ \mathcal{S}\ P \Longrightarrow$
mds-consistent (*update-modes upd mds*)
($\Gamma \oplus_{\mathcal{S}} upd$)
(*add-anno-stable* $\mathcal{S}$ *upd*)
(P |‘ {v. *stable* (*add-anno-stable* $\mathcal{S}$ *upd*) v})
⟨*proof*⟩

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

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

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

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

lemma *dom-add-anno-rel*:
$\forall x{\in}dom\ (\Gamma \oplus_{\mathcal{S}} (v -=_m m)).\ (\Gamma \oplus_{\mathcal{S}} (v -=_m m))\ x = \Gamma\ x$
⟨*proof*⟩

lemma *types-wellformed-mode-update*:
types-wellformed $\Gamma \Longrightarrow$
types-wellformed $(\Gamma \oplus_{\mathcal{S}} upd)$
⟨*proof*⟩

lemma *types-stable-mode-update*:
types-stable $\Gamma\ \mathcal{S} \Longrightarrow$ *types-wellformed* $\Gamma \Longrightarrow$ *anno-type-stable* $\Gamma\ \mathcal{S}\ upd$
$\Longrightarrow$ *types-stable* $(\Gamma \oplus_{\mathcal{S}} upd)$ (*add-anno-stable* $\mathcal{S}\ upd$)
⟨*proof*⟩

lemma *tyenv-wellformed-mode-update*:
tyenv-wellformed mds $\Gamma\ \mathcal{S}\ P \Longrightarrow$ *anno-type-stable* $\Gamma\ \mathcal{S}\ upd \Longrightarrow$
tyenv-wellformed (*update-modes upd mds*)
$(\Gamma \oplus_{\mathcal{S}} upd)$
(*add-anno-stable* $\mathcal{S}\ upd$)
$(P\ |`\ \{v.\ stable\ (add\text{-}anno\text{-}stable\ \mathcal{S}\ upd)\ v\})$
⟨*proof*⟩

lemma *stop-cxt* :
$[\![\vdash \Gamma,\mathcal{S},P\ \{\ c\ \}\ \Gamma',\mathcal{S}',P'\ ;\ c = Stop\ ]\!] \Longrightarrow$
context-equiv $\Gamma\ P\ \Gamma' \wedge \mathcal{S}' = \mathcal{S} \wedge P \vdash P' \wedge (\forall mds.\ tyenv\text{-}wellformed\ mds\ \Gamma\ \mathcal{S}\ P$
$\longrightarrow$ *tyenv-wellformed mds* $\Gamma'\ \mathcal{S}\ P'$)
⟨*proof*⟩

lemma *tyenv-wellformed-preds-update*:
$P' = P''\ |`\ \{v.\ stable\ \mathcal{S}\ v\} \Longrightarrow$
tyenv-wellformed mds $\Gamma\ \mathcal{S}\ P \Longrightarrow$ *tyenv-wellformed mds* $\Gamma\ \mathcal{S}\ P'$
⟨*proof*⟩

lemma *mds-consistent-preds-tyenv-update*:
$P' = P''\ |`\ \{v.\ stable\ \mathcal{S}\ v\} \Longrightarrow v \in dom\ \Gamma \Longrightarrow$
mds-consistent mds $\Gamma\ \mathcal{S}\ P \Longrightarrow$ *mds-consistent mds* $(\Gamma(v \mapsto t))\ \mathcal{S}\ P'$
⟨*proof*⟩

lemma *pred-preds-update*:
assumes *mem′-def*: $mem' = mem\ (x := ev_A\ mem\ e)$
assumes *P′-def*: $P' = (assign\text{-}post\ P\ x\ e)\ |`\ \{v.\ stable\ \mathcal{S}\ v\}$
assumes *pred-P*: *pred P mem*
shows *pred* P' mem'
⟨*proof*⟩

lemma *types-wellformed-update*:
types-wellformed $\Gamma \Longrightarrow$ *type-wellformed* $t \Longrightarrow$ *types-wellformed* $(\Gamma(x \mapsto t))$
⟨*proof*⟩

lemma *types-stable-update*:
types-stable $\Gamma\ \mathcal{S} \Longrightarrow$ *type-stable* $\mathcal{S}\ t \Longrightarrow$ *types-stable* $(\Gamma(x \mapsto t))\ \mathcal{S}$
⟨*proof*⟩

lemma *tyenv-wellformed-sub*:
$[\![P_1 \subseteq P_2;$
$\Gamma_2 = \Gamma_1;$ *tyenv-wellformed mds* $\Gamma_2\ \mathcal{S}\ P_2]\!] \Longrightarrow$
tyenv-wellformed mds $\Gamma_1\ \mathcal{S}\ P_1$
⟨*proof*⟩

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

lemma *tyenv-sec-mode-update*:
(∀ *x*. (*to-total* Γ *x*) ≤:$_P$ *''* (*to-total* Γ*''* *x*)) ⟹ *pred P''* *mem* ⟹ $\mathcal{S}$ = (*mds AsmNoWrite*, *mds AsmNoReadOrWrite*) ⟹
anno-type-sec Γ $\mathcal{S}$ *P upd* ⟹ $\mathcal{S}''$ = *add-anno-stable* $\mathcal{S}$ *upd* ⟹ (∀ *p*∈*P*. ∀ *v*∈*bexp-vars p*. *stable* $\mathcal{S}$ *v*) ⟹
P'' = *P* |‘ {*v*. *stable* $\mathcal{S}''$ *v*} ⟹
Γ*''* = Γ ⊕$_{\mathcal{S}}$ *upd* ⟹ *tyenv-sec mds* Γ *mem* ⟹ *tyenv-sec* (*update-modes upd mds*) Γ*''* *mem*
⟨*proof*⟩

lemma *tyenv-sec-eq*:
∀ *x*∈$\mathcal{C}$. *mem x* = *mem' x* ⟹ *types-wellformed* Γ ⟹ *tyenv-sec mds* Γ *mem* = *tyenv-sec mds* Γ *mem'*
⟨*proof*⟩

lemma *context-equiv-tyenv-sec*:
context-equiv $\Gamma_2\ P_2\ \Gamma_1 \Longrightarrow$
pred P_2 *mem* ⟹ *tyenv-sec mds* Γ_2 *mem* ⟹ *tyenv-sec mds* Γ_1 *mem*
⟨*proof*⟩

lemma *add-pred-entailment*:
$P +_{\mathcal{S}} p \vdash P$
⟨*proof*⟩

lemma *preservation-no-await*:
$[\![\vdash \Gamma,\mathcal{S},P\ \{\ c\ \}\ \Gamma',\mathcal{S}',P';$

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

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

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-wellformed mds* $\Gamma\ \mathcal{S}\ P \wedge$ *pred* P *mem* $\wedge$ *tyenv-sec mds* Γ *mem* $\longrightarrow$
tyenv-wellformed mds' $\Gamma''\ \mathcal{S}''\ P'' \wedge$ *pred* P'' *mem'* $\wedge$ *tyenv-sec mds'* Γ'' *mem'*)
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

lemma *context-equiv-tyenv-eq*:
tyenv-eq $\Gamma\ mem\ mem' \Longrightarrow$ *context-equiv* $\Gamma\ P\ \Gamma' \Longrightarrow$ *pred* $P\ mem \Longrightarrow$ *tyenv-eq* $\Gamma'\ mem\ mem'$
⟨*proof*⟩

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

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

⟨*proof*⟩

lemma *is-final-*$\mathcal{R}_u$*-is-final*:
$\langle c_1, mds, mem_1\rangle\ \mathcal{R}^u{}_{\Gamma,\mathcal{S},P}\ \langle c_2, mds, mem_2\rangle \Longrightarrow$ *is-final* $c_1 \Longrightarrow$ *is-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 intro*$_1$ *intro*$_3$]:
$[\![\langle c_1, mds, mem_1\rangle\ \mathcal{R}^3{}_{\Gamma,\mathcal{S},P}\ \langle c_2, mds, mem_2\rangle;$
$\bigwedge c_1\ mds\ mem_1\ \Gamma\ \mathcal{S}\ P\ c_2\ mem_2\ c\ \Gamma'\ \mathcal{S}'\ P'.$
$[\![\langle c_1, mds, mem_1\rangle\ \mathcal{R}^1{}_{\Gamma,\mathcal{S},P}\ \langle c_2, mds, mem_2\rangle;$
$\vdash \Gamma,\mathcal{S},P\ \{c\}\ \Gamma',\mathcal{S}',P']\!] \Longrightarrow$
$Q\ (c_1\ ;;\ c)\ mds\ mem_1\ \Gamma'\ \mathcal{S}'\ P'\ (c_2\ ;;\ c)\ mds\ mem_2;$
$\bigwedge c_1\ mds\ mem_1\ \Gamma\ \mathcal{S}\ P\ c_2\ mem_2\ c\ \Gamma'\ \mathcal{S}'\ P'.$
$[\![\langle c_1, mds, mem_1\rangle\ \mathcal{R}^3{}_{\Gamma,\mathcal{S},P}\ \langle c_2, mds, mem_2\rangle;$
$Q\ c_1\ mds\ mem_1\ \Gamma\ \mathcal{S}\ P\ c_2\ mds\ mem_2;$
$\vdash \Gamma,\mathcal{S},P\ \{c\}\ \Gamma',\mathcal{S}',P']\!] \Longrightarrow$
$Q\ (c_1\ ;;\ c)\ mds\ mem_1\ \Gamma'\ \mathcal{S}'\ P'\ (c_2\ ;;\ c)\ mds\ mem_2]\!] \Longrightarrow$
$Q\ c_1\ mds\ mem_1\ \Gamma\ \mathcal{S}\ P\ c_2\ mds\ mem_2$

⟨*proof*⟩

lemma $\mathcal{R}$*-typed-step-plus*:
$[\![\langle c_1, mds, mem_1\rangle \rightsquigarrow^+ \langle c_1', mds', mem_1'\rangle;$
$\vdash \Gamma,\mathcal{S},P\ \{\ c_1\ \}\ \Gamma',\mathcal{S}',P';$
no-await c_1;
tyenv-wellformed mds Γ $\mathcal{S}$ P;
$mem_1 =_\Gamma mem_2$;
pred P mem_1;
pred P mem_2;
tyenv-sec mds Γ mem_1 $\rrbracket \Longrightarrow$
$(\exists\ c_2'\ mem_2'.\ \langle c_1, mds, mem_2\rangle \rightsquigarrow^+ \langle c_2', mds', mem_2'\rangle \wedge$
$\langle c_1', mds', mem_1'\rangle\ \mathcal{R}^u{}_{\Gamma',\mathcal{S}',P'}\ \langle c_2', mds', mem_2'\rangle)$

⟨*proof*⟩

lemma $\mathcal{R}$*-typed-step*:
$[\![\vdash \Gamma,\mathcal{S},P\ \{\ c_1\ \}\ \Gamma',\mathcal{S}',P'\ ;$
tyenv-wellformed mds Γ $\mathcal{S}$ P; $mem_1 =_\Gamma mem_2$; *pred* P mem_1;
pred P mem_2; *tyenv-sec mds* Γ mem_1;
$\langle c_1, mds, mem_1\rangle \rightsquigarrow \langle c_1', mds', mem_1'\rangle\ \rrbracket \Longrightarrow$
$(\exists\ c_2'\ mem_2'.\ \langle c_1, mds, mem_2\rangle \rightsquigarrow \langle c_2', mds', mem_2'\rangle \wedge$

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

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

lemma $\mathcal{R}$-*to*-$\mathcal{R}_3$: ⟦ $\langle c_1, mds, mem_1 \rangle \; \mathcal{R}^u{}_{\Gamma,\mathcal{S},P} \; \langle c_2, mds, mem_2 \rangle$; ⊢ Γ,$\mathcal{S}$,P { c } Γ',$\mathcal{S}'$,P' ⟧ ⟹
$\langle c_1 \;;;\; c, mds, mem_1 \rangle \; \mathcal{R}^3{}_{\Gamma',\mathcal{S}',P'} \; \langle c_2 \;;;\; c, mds, mem_2 \rangle$
⟨*proof*⟩

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

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

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

theorem *type-soundness*:
assumes *well-typed*: ⊢ Γ,$\mathcal{S}$,P { c } Γ',$\mathcal{S}'$,P'
assumes *wf*: *tyenv-wellformed mds* Γ $\mathcal{S}$ P
assumes *mem-eq*: ∀ x. *type-max* (*to-total* Γ x) mem_1 = *Low* ⟶ mem_1 x = mem_2 x
assumes *pred*$_1$: *pred* P mem_1
assumes *pred*$_2$: *pred* P mem_2
assumes *tyenv-sec*: *tyenv-sec mds* Γ mem_1
shows $\langle c, mds, mem_1 \rangle \approx \langle c, mds, mem_2 \rangle$
⟨*proof*⟩

definition
Γ-*of-mds* :: *'Var Mds* ⇒ (*'Var*,*'BExp*) *TyEnv*
where

Γ-*of-mds mds* $\equiv$ (λx. *if* $x \notin \mathcal{C} \wedge x \in$ *mds AsmNoWrite* $\cup$ *mds AsmNoReadOrWrite* *then*
if $x \in$ *mds AsmNoReadOrWrite then*
Some ({*pred-False*})
else
Some (*dma-type* x)
else None)

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

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

inductive
type-global :: ((*'Var*, *'AExp*, *'BExp*) *Stmt* $\times$ *'Var Mds*) *list* $\Rightarrow$ *bool*
($\vdash$ - [*120*] *1000*)
where
$[\![$ *list-all* (λ (*c*,*m*). ($\exists$ Γ' $\mathcal{S}'$ P'. $\vdash$ (Γ-*of-mds m*),($\mathcal{S}$-*of-mds m*),{} { *c* } Γ',$\mathcal{S}'$,P') $\wedge$ *mds-yields-stable-types m*) *cs* ;
$\forall$ *mem. sound-mode-use* (*cs*, *mem*)
$]\!] \Longrightarrow$
type-global cs

inductive-cases *type-global-elim*: $\vdash$ *cs*

lemma *of-mds-tyenv-wellformed*: *mds-yields-stable-types m* $\Longrightarrow$ *tyenv-wellformed m* (Γ-*of-mds m*) ($\mathcal{S}$-*of-mds m*) {}
⟨*proof*⟩

lemma Γ-*of-mds-tyenv-sec*:
tyenv-sec m (Γ-*of-mds m*) mem_1
⟨*proof*⟩

lemma *type-max-pred-False* [*simp*]:
type-max {*pred-False*} *mem* = *High*
⟨*proof*⟩

lemma *typed-secure*:
$[\![\vdash$ (Γ-*of-mds m*),($\mathcal{S}$-*of-mds m*),{} { *c* } Γ',$\mathcal{S}'$,P'; *mds-yields-stable-types m* $]\!] \Longrightarrow$

com-sifum-secure (*c*,*m*)
⟨*proof*⟩

lemma *list-all-set*: ∀ *x* ∈ *set xs*. *P x* ⟹ *list-all P xs*
⟨*proof*⟩

theorem *type-soundness-global*:
assumes *typeable*: ⊢ *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* $\mathcal{C}$*-vars* $\mathcal{C}$ $eval_w$ *undefined*
for ev_A :: (*'Var*, *'Val*) *Mem* ⇒ *'AExp* ⇒ *'Val*
and ev_B :: (*'Var*, *'Val*) *Mem* ⇒ *'BExp* ⇒ *bool*
and *aexp-vars* :: *'AExp* ⇒ *'Var set*
and *bexp-vars* :: *'BExp* ⇒ *'Var set*
and *dma* :: (*'Var*,*'Val*) *Mem* ⇒ *'Var* ⇒ *Sec*
and $\mathcal{C}$*-vars* :: *'Var* ⇒ *'Var set*
and $\mathcal{C}$ :: *'Var set*

context *sifum-modes*
begin

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

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

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

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

inductive
mode-type :: *'Var Mds ⇒ ('Var, 'AExp, 'BExp) Stmt ⇒ 'Var Mds ⇒ bool* (⊢ - { - } -)
where
skip: ⊢ *mds* { *Skip* ⊗ *annos* } (*mds* ⊕ *annos*) |
assign: ⟦ *x* ∉ *mds GuarNoWrite* ∪ *mds GuarNoReadOrWrite* ; *aexp-vars e* ∩ *mds GuarNoReadOrWrite* = {};
∀ *v*. (*x* ∈ *C-vars v* ⟶ *v* ∉ *mds GuarNoWrite* ∪ *mds GuarNoReadOrWrite*) ∧
(*C-vars v* ∩ *aexp-vars e* ≠ {} ⟶ *v* ∉ *mds GuarNoReadOrWrite*)⟧ ⟹
⊢ *mds* { (*x* ← *e*) ⊗ *annos* } (*mds* ⊕ *annos*) |
if-: ⟦ ⊢ (*mds* ⊕ *annos*) { *c*$_1$ } *mds''* ;
⊢ (*mds* ⊕ *annos*) { *c*$_2$ } *mds''* ;
bexp-vars e ∩ *mds GuarNoReadOrWrite* = {};
∀ *v*. *C-vars v* ∩ *bexp-vars e* ≠ {} ⟶ *v* ∉ *mds GuarNoReadOrWrite*⟧ ⟹
⊢ *mds* { *If e c*$_1$ *c*$_2$ ⊗ *annos* } *mds''* |
while: ⟦ *mds'* = *mds* ⊕ *annos* ; ⊢ *mds'* { *c* } *mds'* ; *bexp-vars e* ∩ *mds' GuarNoReadOrWrite* = {};
∀ *v*. *C-vars v* ∩ *bexp-vars e* ≠ {} ⟶ *v* ∉ *mds' GuarNoReadOrWrite*⟧ ⟹
⊢ *mds* { *While e c* ⊗ *annos* } *mds'* |
seq: ⟦ ⊢ *mds* { *c*$_1$ } *mds'* ; ⊢ *mds'* { *c*$_2$ } *mds''* ⟧ ⟹ ⊢ *mds* { *c*$_1$;; *c*$_2$ } *mds''* |
sub: ⟦ ⊢ *mds*$_2$ { *c* } *mds*$_2$*'* ; *mds*$_1$ ≤ *mds*$_2$; *mds*$_2$*'* ≤ *mds*$_1$*'* ⟧ ⟹
⊢ *mds*$_1$ { *c* } *mds*$_1$*'*

6.2 Soundness of the Type System

lemma *cxt-eval*:
⟦ ⟨*cxt-to-stmt* [] *c*, *mds*, *mem*⟩ ⇝ ⟨*cxt-to-stmt* [] *c'*, *mds'*, *mem'*⟩ ⟧ ⟹
⟨*c*, *mds*, *mem*⟩ ⇝ ⟨*c'*, *mds'*, *mem'*⟩
⟨*proof*⟩

lemma *update-preserves-le*:
mds$_1$ ≤ *mds*$_2$ ⟹ (*mds*$_1$ ⊕ *annos*) ≤ (*mds*$_2$ ⊕ *annos*)
⟨*proof*⟩

lemma *doesnt-read-annos*:

doesnt-read-or-modify c x $\Longrightarrow$ *doesnt-read-or-modify* $(c \otimes annos)$ *x*
⟨*proof*⟩

lemma *doesnt-modify-annos*:
doesnt-modify c x $\Longrightarrow$ *doesnt-modify* $(c \otimes annos)$ *x*
⟨*proof*⟩

lemma *stop-loc-reach*:
$[\![\langle c', mds', mem' \rangle \in$ *loc-reach* $\langle Stop, mds, mem \rangle]\!] \Longrightarrow$
$c' = Stop \wedge mds' = mds$
⟨*proof*⟩

lemma *stop-doesnt-access*:
doesnt-modify Stop x $\wedge$ *doesnt-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*:
$[\![\langle Skip \otimes annos, mds, mem \rangle \rightsquigarrow \langle c', mds', mem' \rangle]\!] \Longrightarrow c' = Stop \wedge mds' = mds \oplus annos \wedge mem' = mem$
⟨*proof*⟩

lemma *skip-doesnt-read*:
doesnt-read-or-modify $(Skip \otimes annos)$ *x*
⟨*proof*⟩

lemma *skip-doesnt-write*:
doesnt-modify $(Skip \otimes annos)$ *x*
⟨*proof*⟩

lemma *skip-loc-reach*:
$[\![\langle c', mds', mem' \rangle \in$ *loc-reach* $\langle Skip \otimes annos, mds, mem \rangle]\!] \Longrightarrow$
$(c' = Stop \wedge mds' = (mds \oplus annos)) \vee (c' = Skip \otimes annos \wedge mds' = mds)$
⟨*proof*⟩

lemma *skip-doesnt-access*:
$[\![lc \in$ *loc-reach* $\langle Skip \otimes annos, mds, mem \rangle$; $lc = \langle c', mds', mem' \rangle]\!] \Longrightarrow$ *doesnt-read-or-modify* c' *x* $\wedge$ *doesnt-modify* c' *x*
⟨*proof*⟩

lemma *assign-doesnt-modify*:
$[\![x \neq y; x \notin \mathcal{C}\text{-}vars\ y]\!] \Longrightarrow$ *doesnt-modify* $((x \leftarrow e) \otimes annos)$ *y*
⟨*proof*⟩

lemma *assign-annos-eval*:
$\langle (x \leftarrow e) \otimes annos,\ mds,\ mem \rangle \rightsquigarrow \langle Stop,\ mds \oplus annos,\ mem\ (x := ev_A\ mem\ e) \rangle$
⟨*proof*⟩

lemma *assign-annos-eval-elim*:
$[\![\ \langle (x \leftarrow e) \otimes annos,\ mds,\ mem \rangle \rightsquigarrow \langle c',\ mds',\ mem' \rangle\ ]\!] \Longrightarrow$
$c' = Stop \wedge mds' = mds \oplus annos$
⟨*proof*⟩

lemma *mem-upd-commute*:
$[\![\ x \neq y\ ]\!] \Longrightarrow mem\ (x := v_1,\ y := v_2) = mem\ (y := v_2,\ x := v_1)$
⟨*proof*⟩

lemma *assign-doesnt-read*:
$[\![\ y \neq x;\ y \notin aexp\text{-}vars\ e;\ x \notin \mathcal{C}\text{-}vars\ y;\ \mathcal{C}\text{-}vars\ y \cap aexp\text{-}vars\ e = \{\}\ ]\!] \Longrightarrow$ *doesnt-read-or-modify* $((x \leftarrow e) \otimes annos)\ y$
⟨*proof*⟩

lemma *assign-loc-reach*:
$[\![\ \langle c',\ mds',\ mem' \rangle \in loc\text{-}reach\ \langle (x \leftarrow e) \otimes annos,\ mds,\ mem \rangle\ ]\!] \Longrightarrow$
$(c' = Stop \wedge mds' = (mds \oplus annos)) \vee (c' = (x \leftarrow e) \otimes annos \wedge mds' = mds)$
⟨*proof*⟩

lemma *if-doesnt-modify*:
doesnt-modify $(If\ e\ c_1\ c_2 \otimes annos)\ x$
⟨*proof*⟩

lemma *vars-eval*$_B$:
$x \notin bexp\text{-}vars\ e \Longrightarrow ev_B\ mem\ e = ev_B\ (mem\ (x := v))\ e$
⟨*proof*⟩

lemma *if-doesnt-read*:
$x \notin bexp\text{-}vars\ e \Longrightarrow \mathcal{C}\text{-}vars\ x \cap bexp\text{-}vars\ e = \{\} \Longrightarrow$ *doesnt-read-or-modify* $(If\ e\ c_1\ c_2 \otimes annos)\ x$
⟨*proof*⟩

lemma *if-eval-true*:
$[\![\ ev_B\ mem\ e\ ]\!] \Longrightarrow$
$\langle If\ e\ c_1\ c_2 \otimes annos,\ mds,\ mem \rangle \rightsquigarrow \langle c_1,\ mds \oplus annos,\ mem \rangle$
⟨*proof*⟩

lemma *if-eval-false*:
$[\![\ \neg\ ev_B\ mem\ e\ ]\!] \Longrightarrow$
$\langle If\ e\ c_1\ c_2 \otimes annos,\ mds,\ mem \rangle \rightsquigarrow \langle c_2,\ mds \oplus annos,\ mem \rangle$
⟨*proof*⟩

lemma *if-eval-elim*:
$[\![\ \langle If\ e\ c_1\ c_2 \otimes annos,\ mds,\ mem \rangle \rightsquigarrow \langle c',\ mds',\ mem' \rangle\ ]\!] \Longrightarrow$
$((c' = c_1 \wedge ev_B\ mem\ e) \vee (c' = c_2 \wedge \neg\ ev_B\ mem\ e)) \wedge mds' = mds \oplus annos \wedge$

$mem' = mem$
⟨*proof*⟩

lemma *if-eval-elim'*:
⟦ ⟨*If e* c_1 c_2, *mds*, *mem*⟩ ⇝ ⟨c', *mds'*, *mem'*⟩ ⟧ ⟹
((*c'* = c_1 ∧ ev_B *mem e*) ∨ (*c'* = c_2 ∧ ¬ ev_B *mem e*)) ∧ *mds'* = *mds* ∧ *mem'* = *mem*
⟨*proof*⟩

lemma *loc-reach-refl'*:
⟨*c*, *mds*, *mem*⟩ ∈ *loc-reach* ⟨*c*, *mds*, *mem*⟩
⟨*proof*⟩

lemma *if-loc-reach*:
⟦ ⟨*c'*, *mds'*, *mem'*⟩ ∈ *loc-reach* ⟨*If e* c_1 c_2 ⊗ *annos*, *mds*, *mem*⟩ ⟧ ⟹
(*c'* = *If e* c_1 c_2 ⊗ *annos* ∧ *mds'* = *mds*) ∨
(∃ *mem''*. ⟨*c'*, *mds'*, *mem'*⟩ ∈ *loc-reach* ⟨c_1, *mds* ⊕ *annos*, *mem''*⟩) ∨
(∃ *mem''*. ⟨*c'*, *mds'*, *mem'*⟩ ∈ *loc-reach* ⟨c_2, *mds* ⊕ *annos*, *mem''*⟩)
⟨*proof*⟩

lemma *if-loc-reach'*:
⟦ ⟨*c'*, *mds'*, *mem'*⟩ ∈ *loc-reach* ⟨*If e* c_1 c_2, *mds*, *mem*⟩ ⟧ ⟹
(*c'* = *If e* c_1 c_2 ∧ *mds'* = *mds*) ∨
(∃ *mem''*. ⟨*c'*, *mds'*, *mem'*⟩ ∈ *loc-reach* ⟨c_1, *mds*, *mem''*⟩) ∨
(∃ *mem''*. ⟨*c'*, *mds'*, *mem'*⟩ ∈ *loc-reach* ⟨c_2, *mds*, *mem''*⟩)
⟨*proof*⟩

lemma *seq-loc-reach*:
⟦ ⟨*c'*, *mds'*, *mem'*⟩ ∈ *loc-reach* ⟨c_1 ;; c_2, *mds*, *mem*⟩ ⟧ ⟹
(∃ *c''*. *c'* = *c''* ;; c_2 ∧ ⟨*c''*, *mds'*, *mem'*⟩ ∈ *loc-reach* ⟨c_1, *mds*, *mem*⟩) ∨
(∃ *c'' mds'' mem''*. ⟨*Stop*, *mds''*, *mem''*⟩ ∈ *loc-reach* ⟨c_1, *mds*, *mem*⟩ ∧
⟨*c'*, *mds'*, *mem'*⟩ ∈ *loc-reach* ⟨c_2, *mds''*, *mem''*⟩)
⟨*proof*⟩

lemma *seq-doesnt-read*:
⟦ *doesnt-read-or-modify c x* ⟧ ⟹ *doesnt-read-or-modify* (*c* ;; *c'*) *x*
⟨*proof*⟩

lemma *seq-doesnt-modify*:
⟦ *doesnt-modify c x* ⟧ ⟹ *doesnt-modify* (*c* ;; *c'*) *x*
⟨*proof*⟩

inductive-cases *seq-stop-elim'*: ⟨*Stop* ;; *c*, *mds*, *mem*⟩ ⇝ ⟨*c'*, *mds'*, *mem'*⟩

lemma *seq-stop-elim*: ⟨*Stop* ;; *c*, *mds*, *mem*⟩ ⇝ ⟨*c'*, *mds'*, *mem'*⟩ ⟹
c' = *c* ∧ *mds'* = *mds* ∧ *mem'* = *mem*
⟨*proof*⟩

lemma *seq-split*:

⟦ ⟨*Stop*, *mds′*, *mem′*⟩ ∈ *loc-reach* ⟨c_1 ;; c_2, *mds*, *mem*⟩ ⟧ ⟹
∃ *mds″ mem″*. ⟨*Stop*, *mds″*, *mem″*⟩ ∈ *loc-reach* ⟨c_1, *mds*, *mem*⟩ ∧
⟨*Stop*, *mds′*, *mem′*⟩ ∈ *loc-reach* ⟨c_2, *mds″*, *mem″*⟩
⟨*proof*⟩

lemma *while-eval*:
⟨*While e c* ⊗ *annos*, *mds*, *mem*⟩ ⇝ ⟨(*If e* (*c* ;; *While e c*) *Stop*), *mds* ⊕ *annos*, *mem*⟩
⟨*proof*⟩

lemma *while-eval′*:
⟨*While e c*, *mds*, *mem*⟩ ⇝ ⟨*If e* (*c* ;; *While e c*) *Stop*, *mds*, *mem*⟩
⟨*proof*⟩

lemma *while-eval-elim*:
⟦ ⟨*While e c* ⊗ *annos*, *mds*, *mem*⟩ ⇝ ⟨*c′*, *mds′*, *mem′*⟩ ⟧ ⟹
(*c′* = *If e* (*c* ;; *While e c*) *Stop* ∧ *mds′* = *mds* ⊕ *annos* ∧ *mem′* = *mem*)
⟨*proof*⟩

lemma *while-eval-elim′*:
⟦ ⟨*While e c*, *mds*, *mem*⟩ ⇝ ⟨*c′*, *mds′*, *mem′*⟩ ⟧ ⟹
(*c′* = *If e* (*c* ;; *While e c*) *Stop* ∧ *mds′* = *mds* ∧ *mem′* = *mem*)
⟨*proof*⟩

lemma *while-doesnt-read*:
⟦ *x* ∉ *bexp-vars e* ⟧ ⟹ *doesnt-read-or-modify* (*While e c* ⊗ *annos*) *x*
⟨*proof*⟩

lemma *while-doesnt-modify*:
doesnt-modify (*While e c* ⊗ *annos*) *x*
⟨*proof*⟩

lemma *disjE3*:
⟦ *A* ∨ *B* ∨ *C* ; *A* ⟹ *P* ; *B* ⟹ *P* ; *C* ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *disjE5*:
⟦ *A* ∨ *B* ∨ *C* ∨ *D* ∨ *E* ; *A* ⟹ *P* ; *B* ⟹ *P* ; *C* ⟹ *P* ; *D* ⟹ *P* ; *E* ⟹ *P* ⟧
⟹ *P*
⟨*proof*⟩

lemma *if-doesnt-read′*:
x ∉ *bexp-vars e* ⟹ *C*-*vars x* ∩ *bexp-vars e* = {} ⟹ *doesnt-read-or-modify* (*If e* c_1 c_2) *x*
⟨*proof*⟩

theorem *mode-type-sound*:
assumes *typeable*: ⊢ mds_1 { *c* } $mds_1{}'$

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

end

end

References

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

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

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

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