A Formalization of Declassification with WHAT&WHERE-Security

Sylvia Grewe, Alexander Lux, Heiko Mantel, Jens Sauer

May 26, 2024

Abstract

Research in information-flow security aims at developing methods to identify undesired information leaks within programs from private sources to public sinks. Noninterference captures this intuition by requiring that no information whatsoever flows from private sources to public sinks. However, in practice this definition is often too strict: Depending on the intuitive desired security policy, the controlled declassification of certain private information (WHAT) at certain points in the program (WHERE) might not result in an undesired information leak.

We present an Isabelle/HOL formalization of such a security property for controlled declassification, namely WHAT&WHERE-security from [2]. The formalization includes compositionality proofs for and a soundness proof for a security type system that checks for programs in a simple while language with dynamic thread creation.

Our formalization of the security type system is abstract in the language for expressions and in the semantic side conditions for expressions. It can easily be instantiated with different syntactic approximations for these side conditions. The soundness proof of such an instantiation boils down to showing that these syntactic approximations imply the semantic side conditions.

This Isabelle/HOL formalization uses theories from the entry Strong-Security (see proof document for details).

Contents

1 Preliminary definitions

1.1 Type synonyms

The formalization is parametric in different aspects. Notably, it is parametric in the security lattice it supports.

For better readability, we use the following type synonyms in our formalization (from the entry Strong-Security):

theory *Types*
imports *Main*
begin

— type parameters:
— 'exp: expressions (arithmetic, boolean...)
— 'val: values
— 'id: identifier names
— 'com: commands
— 'd: domains

This is a collection of type synonyms. Note that not all of these type synonyms are used within Strong-Security - some are used in WHATandWHERE-Security.

type-synonym $('id, 'val)\ State = 'id \Rightarrow 'val$

— type for evaluation functions mapping expressions to a values depending on a state
type-synonym $('exp, 'id, 'val)\ Evalfunction =$
$'exp \Rightarrow ('id, 'val)\ State \Rightarrow 'val$

— define configurations with threads as pair of commands and states
type-synonym $('id, 'val, 'com)\ TConfig = 'com \times ('id, 'val)\ State$

— define configurations with thread pools as pair of command lists (thread pool) and states
type-synonym $('id, 'val, 'com)\ TPConfig =$

$('com\ list) \times ('id, 'val)\ State$

— type for program states (including the set of commands and a symbol for terminating - None)
type-synonym $'com\ ProgramState = 'com\ option$

— type for configurations with program states
type-synonym $('id, 'val, 'com)\ PSConfig =$
$'com\ ProgramState \times ('id, 'val)\ State$

— type for labels with a list of spawned threads
type-synonym $'com\ Label = 'com\ list$

— type for step relations from single commands to a program state, with a label
type-synonym $('exp, 'id, 'val, 'com)\ TLSteps =$
$(('id, 'val, 'com)\ TConfig \times 'com\ Label$
$\times ('id, 'val, 'com)\ PSConfig)\ set$

— curried version of previously defined type
type-synonym $('exp, 'id, 'val, 'com)\ TLSteps\text{-}curry =$
$'com \Rightarrow ('id, 'val)\ State \Rightarrow 'com\ Label \Rightarrow 'com\ ProgramState$
$\Rightarrow ('id, 'val)\ State \Rightarrow bool$

— type for step relations from thread pools to thread pools
type-synonym $('exp, 'id, 'val, 'com)\ TPSteps =$
$(('id, 'val, 'com)\ TPConfig \times ('id, 'val, 'com)\ TPConfig)\ set$

— curried version of previously defined type
type-synonym $('exp, 'id, 'val, 'com)\ TPSteps\text{-}curry =$
$'com\ list \Rightarrow ('id, 'val)\ State \Rightarrow 'com\ list \Rightarrow ('id, 'val)\ State \Rightarrow bool$

— define type of step relations for single threads to thread pools
type-synonym $('exp, 'id, 'val, 'com)\ TSteps =$
$(('id, 'val, 'com)\ TConfig \times ('id, 'val, 'com)\ TPConfig)\ set$

— define the same type as TSteps, but in a curried version (allowing syntax abbreviations)
type-synonym $('exp, 'id, 'val, 'com)\ TSteps\text{-}curry =$
$'com \Rightarrow ('id, 'val)\ State \Rightarrow 'com\ list \Rightarrow ('id, 'val)\ State \Rightarrow bool$

— type for simple domain assignments; 'd has to be an instance of order (partial order
type-synonym $('id, 'd)\ DomainAssignment = 'id \Rightarrow 'd::order$

type-synonym $'com\ Bisimulation\text{-}type = (('com\ list) \times ('com\ list))\ set$

— type for escape hatches
type-synonym $('d, 'exp)\ Hatch = 'd \times 'exp$

— type for sets of escape hatches
type-synonym $('d, 'exp)\ Hatches = (('d, 'exp)\ Hatch)\ set$

— type for local escape hatches
type-synonym $('d, 'exp)\ lHatch = 'd \times 'exp \times nat$

— type for sets of local escape hatches
type-synonym $('d, 'exp)\ lHatches = (('d, 'exp)\ lHatch)\ set$

end

2 WHAT&WHERE-security

2.1 Definition of WHAT&WHERE-security

The definition of WHAT&WHERE-security is parametric in a security lattice (*'d*) and in a programming language (*'com*).

theory *WHATWHERE-Security*
imports *Strong-Security.Types*
begin

locale *WHATWHERE* =
fixes $SR :: ('exp, 'id, 'val, 'com)\ TLSteps$
and $E :: ('exp, 'id, 'val)\ Evalfunction$
and $pp :: 'com \Rightarrow nat$
and $DA :: ('id, 'd::order)\ DomainAssignment$
and $lH :: ('d::order, 'exp)\ lHatches$

begin

— define when two states are indistinguishable for an observer on domain d
definition $d\text{-}equal :: 'd::order \Rightarrow ('id, 'val)\ State$
$\Rightarrow ('id, 'val)\ State \Rightarrow bool$
where
$d\text{-}equal\ d\ m\ m' \equiv \forall x.\ ((DA\ x) \leq d \longrightarrow (m\ x) = (m'\ x))$

abbreviation $d\text{-}equal' :: ('id, 'val)\ State$
$\Rightarrow 'd::order \Rightarrow ('id, 'val)\ State \Rightarrow bool$
((- =- -))
where
$m =_d m' \equiv d\text{-}equal\ d\ m\ m'$

— transitivity of d-equality
lemma *d-equal-trans*:
$[\![\ m =_d m';\ m' =_d m''\ ]\!] \Longrightarrow m =_d m''$
⟨*proof*⟩

abbreviation *SRabbr* :: (*'exp*, *'id*, *'val*, *'com*) *TLSteps-curry*
((*1*⟨-,/-⟩) →◁-▷/ (*1*⟨-,/-⟩) [*0*,*0*,*0*,*0*,*0*] *81*)
where
⟨*c*,*m*⟩ →◁α▷ ⟨*p*,*m'*⟩ ≡ ((*c*,*m*),α,(*p*,*m'*)) ∈ *SR*

— function for obtaining the unique memory (state) after one step for a command and a memory (state)
definition *NextMem* :: *'com* ⇒ (*'id*, *'val*) *State* ⇒ (*'id*, *'val*) *State*
(⟦-⟧'(-'))
where
⟦*c*⟧(*m*) ≡ (*THE m'*. (∃ *p* α. ⟨*c*,*m*⟩ →◁α▷ ⟨*p*,*m'*⟩))

— function getting all escape hatches for some location
definition *htchLoc* :: *nat* ⇒ (*'d*, *'exp*) *Hatches*
where
htchLoc ι ≡ {(*d*,*e*). (*d*,*e*,ι) ∈ *lH*}

— function for getting all escape hatches for some set of locations
definition *htchLocSet* :: *nat set* ⇒ (*'d*, *'exp*) *Hatches*
where
htchLocSet PP ≡ ⋃{*h*. (∃ ι ∈ *PP*. *h* = *htchLoc* ι)}

— predicate for (d,H)-equality
definition *dH-equal* :: *'d* ⇒ (*'d*, *'exp*) *Hatches*
⇒ (*'id*, *'val*) *State* ⇒ (*'id*, *'val*) *State* ⇒ *bool*
where
dH-equal d H m m' ≡ (*m* =$_d$ *m'* ∧
(∀ (*d'*,*e*) ∈ *H*. (*d'* ≤ *d* ⟶ (*E e m* = *E e m'*))))

abbreviation *dH-equal'* :: (*'id*, *'val*) *State* ⇒ *'d* ⇒ (*'d*, *'exp*) *Hatches*
⇒ (*'id*, *'val*) *State* ⇒ *bool*
((- ∼-,- -))
where
m ∼$_{d,H}$ *m'* ≡ *dH-equal d H m m'*

— predicate indicating that a command is not a d-declassification command
definition *NDC* :: *'d* ⇒ *'com* ⇒ *bool*
where
NDC d c ≡ (∀ *m m'*. *m* =$_d$ *m'* ⟶ ⟦*c*⟧(*m*) =$_d$ ⟦*c*⟧(*m'*))

— predicate indicating an 'immediate d-declassification command' for a set of escape hatches
definition *IDC* :: *'d* ⇒ *'com* ⇒ (*'d*, *'exp*) *Hatches* ⇒ *bool*
where
IDC d c H ≡ (∃ *m m'*. *m* =$_d$ *m'* ∧
(¬ ⟦*c*⟧(*m*) =$_d$ ⟦*c*⟧(*m'*)))

$\wedge$ ($\forall$ *m m'*. *m* $\sim_{d,H}$ *m'* $\longrightarrow$ $[\![c]\!](m) =_d [\![c]\!](m')$)

definition *stepResultsinR* :: *'com ProgramState* $\Rightarrow$ *'com ProgramState*
$\Rightarrow$ *'com Bisimulation-type* $\Rightarrow$ *bool*
where
stepResultsinR p p' R $\equiv$ (*p* = *None* $\wedge$ *p'* = *None*) $\vee$
($\exists$ *c c'*. (*p* = *Some c* $\wedge$ *p'* = *Some c'* $\wedge$ ([*c*],[*c'*]) $\in$ *R*))

definition *dhequality-alternative* :: *'d* $\Rightarrow$ *nat set* $\Rightarrow$ *nat*
$\Rightarrow$ (*'id*, *'val*) *State* $\Rightarrow$ (*'id*, *'val*) *State* $\Rightarrow$ *bool*
where
dhequality-alternative d PP ι *m m'* $\equiv$ *m* $\sim_{d,(htchLocSet\ PP)}$ *m'* $\vee$
($\neg$ (*htchLoc* ι) $\subseteq$ (*htchLocSet PP*))

definition *Strong-dlHPP-Bisimulation* :: *'d* $\Rightarrow$ *nat set*
$\Rightarrow$ *'com Bisimulation-type* $\Rightarrow$ *bool*
where
Strong-dlHPP-Bisimulation d PP R $\equiv$
(*sym R*) $\wedge$ (*trans R*) $\wedge$
($\forall$ (*V*,*V'*) $\in$ *R*. *length V* = *length V'*) $\wedge$
($\forall$ (*V*,*V'*) $\in$ *R*. $\forall$ *i* < *length V*.
((*NDC d* (*V*!*i*)) $\vee$
(*IDC d* (*V*!*i*) (*htchLoc* (*pp* (*V*!*i*)))))) $\wedge$
($\forall$ (*V*,*V'*) $\in$ *R*. $\forall$ *i* < *length V*. $\forall$ *m1 m1' m2* α *p*.
($\langle$ *V*!*i*,*m1* $\rangle$ $\rightarrow\triangleleft\alpha\triangleright$ $\langle$ *p*,*m2* $\rangle$ $\wedge$ *m1* $\sim_{d,(htchLocSet\ PP)}$ *m1'*)
$\longrightarrow$ ($\exists$ *p'* α' *m2'*. ($\langle$ *V'*!*i*,*m1'* $\rangle$ $\rightarrow\triangleleft\alpha'\triangleright$ $\langle$ *p'*,*m2'* $\rangle$ $\wedge$
(*stepResultsinR p p' R*) $\wedge$ (α,α') $\in$ *R* $\wedge$
(*dhequality-alternative d PP* (*pp* (*V*!*i*)) *m2 m2'*))))

— predicate to define when a program is strongly secure
definition *WHATWHERE-Secure* :: *'com list* $\Rightarrow$ *bool*
where
WHATWHERE-Secure V $\equiv$ ($\forall$ *d PP*.
($\exists$ *R*. *Strong-dlHPP-Bisimulation d PP R* $\wedge$ (*V*,*V*) $\in$ *R*))

— auxiliary lemma to obtain central strong (d,lH,PP)-Bisimulation property as Lemma in meta logic (allows instantiating all the variables manually if necessary)
lemma *strongdlHPPB-aux*:
$\bigwedge$*V V' m1 m1' m2 p i* α. $[\![$ *Strong-dlHPP-Bisimulation d PP R*;
i < *length V*; (*V*,*V'*) $\in$ *R*;
$\langle$ *V*!*i*,*m1* $\rangle$ $\rightarrow\triangleleft\alpha\triangleright$ $\langle$ *p*,*m2* $\rangle$; *m1* $\sim_{d,(htchLocSet\ PP)}$ *m1'* $]\!]$
$\Longrightarrow$ ($\exists$ *p'* α' *m2'*. $\langle$ *V'*!*i*,*m1'* $\rangle$ $\rightarrow\triangleleft\alpha'\triangleright$ $\langle$ *p'*,*m2'* $\rangle$
$\wedge$ *stepResultsinR p p' R* $\wedge$ (α,α') $\in$ *R* $\wedge$
(*dhequality-alternative d PP* (*pp* (*V*!*i*)) *m2 m2'*))
$\langle$*proof*$\rangle$
lemma *strongdlHPPB-NDCIDCaux*:

$\bigwedge V\ V'\ i.\ [\![\mathit{Strong\text{-}dlHPP\text{-}Bisimulation}\ d\ PP\ R;$
$(V,V') \in R;\ i < \mathit{length}\ V\ ]\!]$
$\Longrightarrow (\mathit{NDC}\ d\ (V!i) \vee \mathit{IDC}\ d\ (V!i)\ (\mathit{htchLoc}\ (\mathit{pp}\ (V!i))))$
$\langle proof \rangle$

lemma *WHATWHERE-empty*:
WHATWHERE-Secure []
$\langle proof \rangle$

end

end

2.2 Proof technique for compositionality results

For proving compositionality results for WHAT&WHERE-security, we formalize the following "up-to technique" and prove it sound:

theory *Up-To-Technique*
imports *WHATWHERE-Security*
begin

context *WHATWHERE*
begin

abbreviation *SdlHPPB* **where** $\mathit{SdlHPPB} \equiv \mathit{Strong\text{-}dlHPP\text{-}Bisimulation}$

— define the 'reflexive part' of a relation (sets of elements which are related with themselves by the given relation)
definition $\mathit{Refl} :: ('a \times 'a)\ \mathit{set} \Rightarrow 'a\ \mathit{set}$
where
$\mathit{Refl}\ R = \{e.\ (e,e) \in R\}$

lemma *commonRefl-subset-commonDomain*:
$(\mathit{Refl}\ R1 \cap \mathit{Refl}\ R2) \subseteq (\mathit{Domain}\ R1 \cap \mathit{Domain}\ R2)$
$\langle proof \rangle$
definition *disj-dlHPP-Bisimulation-Up-To-R'* ::
$'d \Rightarrow \mathit{nat\ set} \Rightarrow 'com\ \mathit{Bisimulation\text{-}type}$
$\Rightarrow 'com\ \mathit{Bisimulation\text{-}type} \Rightarrow \mathit{bool}$
where
$\mathit{disj\text{-}dlHPP\text{-}Bisimulation\text{-}Up\text{-}To\text{-}R'}\ d\ PP\ R'\ R \equiv$
$\mathit{SdlHPPB}\ d\ PP\ R' \wedge (\mathit{sym}\ R) \wedge (\mathit{trans}\ R)$
$\wedge\ (\forall (V,V') \in R.\ \mathit{length}\ V = \mathit{length}\ V') \wedge$
$(\forall (V,V') \in R.\ \forall i < \mathit{length}\ V.$
$((\mathit{NDC}\ d\ (V!i)) \vee$
$(\mathit{IDC}\ d\ (V!i)\ (\mathit{htchLoc}\ (\mathit{pp}\ (V!i))))))\ \wedge$
$(\forall (V,V') \in R.\ \forall i < \mathit{length}\ V.\ \forall\ m1\ m1'\ m2\ \alpha\ p.$
$(\langle V!i,m1\rangle \rightarrow\!\triangleleft\alpha\triangleright \langle p,m2\rangle \wedge m1 \sim_{d,(\mathit{htchLocSet}\ PP)} m1')$

$\longrightarrow (\exists\, p'\ \alpha'\ m2'.\ \langle V'!i,m1'\rangle \rightarrow\!\triangleleft\alpha'\!\triangleright\ \langle p',m2'\rangle \wedge$
$(stepResultsinR\ p\ p'\ (R \cup R')) \wedge (\alpha,\alpha') \in (R \cup R') \wedge$
$(dhequality\text{-}alternative\ d\ PP\ (pp\ (V!i))\ m2\ m2'))$

— lemma about the transitivity of the union of symmetric and transitive relations under certain circumstances
lemma *trans-RuR'*:
assumes *transR*: *trans R*
assumes *symR*: *sym R*
assumes *transR'*: *trans R'*
assumes *symR'*: *sym R'*
assumes *eqlenR*: $\forall (V,V') \in R.\ length\ V = length\ V'$
assumes *eqlenR'*: $\forall (V,V') \in R'.\ length\ V = length\ V'$
assumes *Areflassump*: $(Arefl\ R \cap Arefl\ R') \subseteq \{[]\}$
shows *trans* $(R \cup R')$
⟨*proof*⟩

lemma *Up-To-Technique*:
⟦ *disj-dlHPP-Bisimulation-Up-To-R' d PP R' R*;
$Arefl\ R \cap Arefl\ R' \subseteq \{[]\}$ ⟧
$\Longrightarrow SdlHPPB\ d\ PP\ (R \cup R')$
⟨*proof*⟩

lemma *Union-Strong-dlHPP-Bisim*:
⟦ *SdlHPPB d PP R*; *SdlHPPB d PP R'*;
$Arefl\ R \cap Arefl\ R' \subseteq \{[]\}$ ⟧
$\Longrightarrow SdlHPPB\ d\ PP\ (R \cup R')$
⟨*proof*⟩

lemma *adding-emptypair-keeps-SdlHPPB*:
assumes *SdlHPP*: *SdlHPPB d PP R*
shows $SdlHPPB\ d\ PP\ (R \cup \{([],[])\})$
⟨*proof*⟩

end

end

2.3 Proof of parallel compositionality

We prove that WHAT&WHERE-security is preserved under composition of WHAT&WHERE-secure threads.

theory *Parallel-Composition*
imports *Up-To-Technique MWLs*
begin

locale *WHATWHERE-Secure-Programs* =
L? : *MWLs-semantics E BMap*
+ *WWs?* : *WHATWHERE MWLsSteps-det E pp DA lH*
for *E* :: (*'exp*, *'id*, *'val*) *Evalfunction*
and *BMap* :: *'val* ⇒ *bool*
and *DA* :: (*'id*, *'d*::*order*) *DomainAssignment*
and *lH* :: (*'d*, *'exp*) *lHatches*
begin

lemma *SdlHPPB-restricted-on-PP-is-SdlHPPB*:
assumes *SdlHPPB*: *SdlHPPB d PP R'*
assumes *inR'*: $(V,V) \in R'$
assumes *Rdef*: $R = \{(V',V''). (V',V'') \in R'$
$\wedge$ *set* (*PPV V'*) $\subseteq$ *set* (*PPV V*)
$\wedge$ *set* (*PPV V''*) $\subseteq$ *set* (*PPV V*)$\}$
shows *SdlHPPB d PP R*
⟨*proof*⟩

theorem *parallel-composition*:
⟦ $\forall i <$ *length V*. *WHATWHERE-Secure* [*V*!*i*]; *unique-PPV V* ⟧
⟹ *WHATWHERE-Secure V*
⟨*proof*⟩

end

end

3 Example language and compositionality proofs

3.1 Example language with dynamic thread creation

As in [2], we instantiate the language with a simple while language that supports dynamic thread creation via a spawn command (Multi-threaded While Language with spawn, MWLs). Note that the language is still parametric in the language used for Boolean and arithmetic expressions (*'exp*).

theory *MWLs*
imports *Strong-Security.Types*
begin

— type parameters not instantiated:
— 'exp: expressions (arithmetic, boolean...)
— 'val: numbers, boolean constants....
— 'id: identifier names

— SYNTAX

datatype (*'exp*, *'id*) *MWLsCom*
= *Skip nat* (*skip*$_-$ [*50*] *70*)
| *Assign 'id nat 'exp*
(-:=$_-$ - [*70*,*50*,*70*] *70*)

| *Seq* (*'exp*, *'id*) *MWLsCom*
(*'exp*, *'id*) *MWLsCom*
(-;- [*61*,*60*] *60*)

| *If-Else nat 'exp* (*'exp*, *'id*) *MWLsCom*
(*'exp*, *'id*) *MWLsCom*
(*if*$_-$ - *then* - *else* - *fi* [*50*,*80*,*79*,*79*] *70*)

| *While-Do nat 'exp* (*'exp*, *'id*) *MWLsCom*
(*while*$_-$ - *do* - *od* [*50*,*80*,*79*] *70*)

| *Spawn nat* ((*'exp*, *'id*) *MWLsCom*) *list*
(*spawn*$_-$ - [*50*,*70*] *70*)

— function for obtaining the program point of some MWLsloc command
primrec *pp* ::(*'exp*, *'id*) *MWLsCom* ⇒ *nat*
where
pp (*skip*$_\iota$) = ι |
pp (*x* :=$_\iota$ *e*) = ι |
pp (*c1*;*c2*) = *pp c1* |
pp (*if*$_\iota$ *b then c1 else c2 fi*) = ι |
pp (*while*$_\iota$ *b do c od*) = ι |
pp (*spawn*$_\iota$ *V*) = ι

— mutually recursive functions to collect program points of commands and thread pools
primrec *PPc* :: (*'exp*,*'id*) *MWLsCom* ⇒ *nat list*
and *PPV* :: (*'exp*,*'id*) *MWLsCom list* ⇒ *nat list*
where
PPc (*skip*$_\iota$) = [ι] |
PPc (*x* :=$_\iota$ *e*) = [ι] |
PPc (*c1*;*c2*) = (*PPc c1*) @ (*PPc c2*) |
PPc (*if*$_\iota$ *b then c1 else c2 fi*) = [ι] @ (*PPc c1*) @ (*PPc c2*) |
PPc (*while*$_\iota$ *b do c od*) = [ι] @ (*PPc c*) |
PPc (*spawn*$_\iota$ *V*) = [ι] @ (*PPV V*) |

PPV [] = [] |
PPV (*c*#*V*) = (*PPc c*) @ (*PPV V*)

— predicate indicating that a command only contains unique program points
definition *unique-PPc* :: (*'exp*, *'id*) *MWLsCom* ⇒ *bool*
where
unique-PPc c = *distinct* (*PPc c*)

— predicate indicating that a thread pool only contains unique program points
definition *unique-PPV* :: (*'exp*, *'id*) *MWLsCom list* ⇒ *bool*
where
unique-PPV V = *distinct* (*PPV V*)

lemma *PPc-nonempt*: *PPc c* ≠ []
⟨*proof*⟩

lemma *unique-c-uneq*: *set* (*PPc c*) ∩ *set* (*PPc c'*) = {} ⟹ *c* ≠ *c'*
⟨*proof*⟩

lemma *V-nonempt-PPV-nonempt*: *V* ≠ [] ⟹ *PPV V* ≠ []
⟨*proof*⟩

lemma *unique-V-uneq*:
⟦*V* ≠ []; *V'* ≠ []; *set* (*PPV V*) ∩ *set* (*PPV V'*) = {}⟧ ⟹ *V* ≠ *V'*
⟨*proof*⟩

lemma *PPc-in-PPV*: *c* ∈ *set V* ⟹ *set* (*PPc c*) ⊆ *set* (*PPV V*)
⟨*proof*⟩

lemma *listindices-aux*: *i* < *length V* ⟹ (*V*!*i*) ∈ *set V*
⟨*proof*⟩

lemma *PPc-in-PPV-version*:
i < *length V* ⟹ *set* (*PPc* (*V*!*i*)) ⊆ *set* (*PPV V*)
⟨*proof*⟩

lemma *uniPPV-uniPPc*: *unique-PPV V* ⟹ (∀ *i* < *length V*. *unique-PPc* (*V*!*i*))
⟨*proof*⟩

locale *MWLs-semantics* =
fixes *E* :: (*'exp*, *'id*, *'val*) *Evalfunction*
and *BMap* :: *'val* ⇒ *bool*
begin

— steps semantics, set of deterministic steps from commands to program states
inductive-set
MWLsSteps-det ::
(*'exp*, *'id*, *'val*, (*'exp*, *'id*) *MWLsCom*) *TLSteps*
and *MWLslocSteps-det'* ::
(*'exp*, *'id*, *'val*, (*'exp*, *'id*) *MWLsCom*) *TLSteps-curry*
((*1*⟨-,/-⟩) →◁-▷/ (*1*⟨-,/-⟩) [*0*,*0*,*0*,*0*,*0*] *81*)
where
⟨*c1*,*m1*⟩ →◁α▷ ⟨*c2*,*m2*⟩ ≡ ((*c1*,*m1*),α,(*c2*,*m2*)) ∈ *MWLsSteps-det* |
skip: ⟨*skip*$_\iota$,*m*⟩ →◁[]▷ ⟨*None*,*m*⟩ |
assign: (*E e m*) = *v* ⟹
⟨*x* :=$_\iota$ *e*,*m*⟩ →◁[]▷ ⟨*None*,*m*(*x* := *v*)⟩ |
seq1: ⟨*c1*,*m*⟩ →◁α▷ ⟨*None*,*m'*⟩ ⟹

$\langle c1;c2,m\rangle \rightarrow\!\triangleleft\alpha\triangleright \langle Some\ c2,m'\rangle$ |
seq2: $\langle c1,m\rangle \rightarrow\!\triangleleft\alpha\triangleright \langle Some\ c1',m'\rangle \Longrightarrow$
$\langle c1;c2,m\rangle \rightarrow\!\triangleleft\alpha\triangleright \langle Some\ (c1';c2),m'\rangle$ |
iftrue: $BMap\ (E\ b\ m) = True \Longrightarrow$
$\langle if_\iota\ b\ then\ c1\ else\ c2\ fi,m\rangle \rightarrow\!\triangleleft[]\triangleright \langle Some\ c1,m\rangle$ |
iffalse: $BMap\ (E\ b\ m) = False \Longrightarrow$
$\langle if_\iota\ b\ then\ c1\ else\ c2\ fi,m\rangle \rightarrow\!\triangleleft[]\triangleright \langle Some\ c2,m\rangle$ |
whiletrue: $BMap\ (E\ b\ m) = True \Longrightarrow$
$\langle while_\iota\ b\ do\ c\ od,m\rangle \rightarrow\!\triangleleft[]\triangleright \langle Some\ (c;(while_\iota\ b\ do\ c\ od)),m\rangle$ |
whilefalse: $BMap\ (E\ b\ m) = False \Longrightarrow$
$\langle while_\iota\ b\ do\ c\ od,m\rangle \rightarrow\!\triangleleft[]\triangleright \langle None,m\rangle$ |
spawn: $\langle spawn_\iota\ V,m\rangle \rightarrow\!\triangleleft V\triangleright \langle None,m\rangle$

inductive-cases *MWLsSteps-det-cases*:
$\langle skip_\iota,m\rangle \rightarrow\!\triangleleft\alpha\triangleright \langle p,m'\rangle$
$\langle x :=_\iota e,m\rangle \rightarrow\!\triangleleft\alpha\triangleright \langle p,m'\rangle$
$\langle c1;c2,m\rangle \rightarrow\!\triangleleft\alpha\triangleright \langle p,m'\rangle$
$\langle if_\iota\ b\ then\ c1\ else\ c2\ fi,m\rangle \rightarrow\!\triangleleft\alpha\triangleright \langle p,m'\rangle$
$\langle while_\iota\ b\ do\ c\ od,m\rangle \rightarrow\!\triangleleft\alpha\triangleright \langle p,m'\rangle$
$\langle spawn_\iota\ V,m\rangle \rightarrow\!\triangleleft\alpha\triangleright \langle p,m'\rangle$

— non-deterministic, possibilistic system step (added for intuition, not used in the proofs)
inductive-set
MWLsSteps-ndet ::
$('exp, 'id, 'val, ('exp, 'id)\ MWLsCom)\ TPSteps$
and *MWLsSteps-ndet'* ::
$('exp, 'id, 'val, ('exp, 'id)\ MWLsCom)\ TPSteps\text{-}curry$
$((1\langle\text{-},/\text{-}\rangle) \Rightarrow/\ (1\langle\text{-},/\text{-}\rangle)\ [0,0,0,0]\ 81)$
where
$\langle V,m\rangle \Rightarrow \langle V',m'\rangle \equiv ((V,m),(V',m')) \in MWLsSteps\text{-}ndet$ |
stepthreadi1: $\langle ci,m\rangle \rightarrow\!\triangleleft\alpha\triangleright \langle None,m'\rangle \Longrightarrow$
$\langle cf\ @\ [ci]\ @\ ca,m\rangle \Rightarrow \langle cf\ @\ \alpha\ @\ ca,m'\rangle$ |
stepthreadi2: $\langle ci,m\rangle \rightarrow\!\triangleleft\alpha\triangleright \langle Some\ c',m'\rangle \Longrightarrow$
$\langle cf\ @\ [ci]\ @\ ca,m\rangle \Rightarrow \langle cf\ @\ [c']\ @\ \alpha\ @\ ca,m\rangle$

— lemma about existence and uniqueness of next memory of a step
lemma *nextmem-exists-and-unique*:
$\exists\, m'\ p\ \alpha.\ \langle c,m\rangle \rightarrow\!\triangleleft\alpha\triangleright \langle p,m'\rangle$
$\wedge\ (\forall\, m''.\ (\exists\, p\ \alpha.\ \langle c,m\rangle \rightarrow\!\triangleleft\alpha\triangleright \langle p,m''\rangle) \longrightarrow m'' = m')$
$\langle proof\rangle$

lemma *PPsc-of-step*:
$[\![\ \langle c,m\rangle \rightarrow\!\triangleleft\alpha\triangleright \langle p,m'\rangle;\ \exists\, c'.\ p = Some\ c'\ ]\!]$
$\Longrightarrow set\ (PPc\ (the\ p)) \subseteq set\ (PPc\ c)$
$\langle proof\rangle$

lemma *PPsα-of-step*:

$\langle c,m\rangle \rightarrow\!\triangleleft\alpha\triangleright \langle p,m'\rangle$
$\Longrightarrow$ *set* $(PPV\ \alpha) \subseteq$ *set* $(PPc\ c)$
$\langle proof\rangle$

end

end

3.2 Proofs of atomic compositionality results

We prove for each atomic command of our example programming language (i.e. a command that is not composed out of other commands) that it is strongly secure if the expressions involved are indistinguishable for an observer on security level d.

theory *WHATWHERE-Secure-Skip-Assign*
imports *Parallel-Composition*
begin

context *WHATWHERE-Secure-Programs*
begin

abbreviation *NextMem'*
$([\![\text{-}]\!]'(\text{-}'))$
where
$[\![c]\!](m) \equiv NextMem\ c\ m$

— define when two expressions are indistinguishable with respect to a domain d
definition *d-indistinguishable* :: $'d$::*order* $\Rightarrow$ $'exp \Rightarrow 'exp \Rightarrow$ *bool*
where
d-indistinguishable d e1 e2 $\equiv \forall\, m\ m'.\ ((m =_d m')$
$\longrightarrow ((E\ e1\ m) = (E\ e2\ m')))$

abbreviation *d-indistinguishable'* :: $'exp \Rightarrow$ $'d$::*order* $\Rightarrow 'exp \Rightarrow$ *bool*
$(\ (\text{-} \equiv_{\text{-}} \text{-})\ )$
where
$e1 \equiv_d e2 \equiv$ *d-indistinguishable d e1 e2*

— symmetry of d-indistinguishable
lemma *d-indistinguishable-sym*:
$e \equiv_d e' \Longrightarrow e' \equiv_d e$
$\langle proof\rangle$
lemma *d-indistinguishable-trans*:
$[\![\ e \equiv_d e';\ e' \equiv_d e''\ ]\!] \Longrightarrow e \equiv_d e''$
$\langle proof\rangle$
definition *dH-indistinguishable* :: $'d \Rightarrow$ ($'d$, $'exp$) *Hatches*
$\Rightarrow 'exp \Rightarrow 'exp \Rightarrow$ *bool*

where
dH-indistinguishable d H e1 e2 $\equiv (\forall\, m\ m'.\ m \sim_{d,H} m'$
$\longrightarrow (E\ e1\ m = E\ e2\ m'))$

abbreviation *dH-indistinguishable'* :: *'exp* $\Rightarrow$ *'d*
$\Rightarrow$ (*'d*, *'exp*) *Hatches* $\Rightarrow$ *'exp* $\Rightarrow$ *bool*
((- $\equiv_{-,-}$ -))
where
$e1 \equiv_{d,H} e2 \equiv$ *dH-indistinguishable d H e1 e2*

lemma *empH-implies-dHindistinguishable-eq-dindistinguishable*:
$(e \equiv_{d,\{\}} e') = (e \equiv_d e')$
⟨*proof*⟩

theorem *WHATWHERE-Secure-Skip*:
WHATWHERE-Secure [$skip_\iota$]
⟨*proof*⟩
lemma *semAssignSC-aux*:
assumes *dhind*: $e \equiv_{DA\ x,(htchLoc\ \iota)} e$
shows *NDC d* $(x :=_\iota e) \vee$ *IDC d* $(x :=_\iota e)$ (*htchLoc* (*pp* ($x:=_\iota$ *e*)))
⟨*proof*⟩

theorem *WHATWHERE-Secure-Assign*:
assumes *dhind*: $e \equiv_{DA\ x,(htchLoc\ \iota)} e$
assumes *dheq-imp*: $\forall\, m\ m'\ d\ \iota'.\ (m \sim_{d,(htchLoc\ \iota')} m' \wedge$
$[\![x :=_\iota e]\!](m) =_d [\![x :=_\iota e]\!](m'))$
$\longrightarrow [\![x :=_\iota e]\!](m) \sim_{d,(htchLoc\ \iota')} [\![x :=_\iota e]\!](m')$
shows *WHATWHERE-Secure* $[x :=_\iota e]$
⟨*proof*⟩

end

end

3.3 Proofs of non-atomic compositionality results

We prove compositionality results for each non-atomic command of our example programming language (i.e. a command that is composed out of other commands): If the components are strongly secure and the expressions involved indistinguishable for an observer on security level d, then the composed command is also strongly secure.

theory *Language-Composition*
imports *WHATWHERE-Secure-Skip-Assign*
begin

context *WHATWHERE-Secure-Programs*
begin

theorem *Compositionality-Seq*:
assumes *WWs-part1*: *WHATWHERE-Secure* [*c1*]
assumes *WWs-part2*: *WHATWHERE-Secure* [*c2*]
assumes *uniPPc1c2*: *unique-PPc* (*c1*;*c2*)
shows *WHATWHERE-Secure* [*c1*;*c2*]
⟨*proof*⟩

theorem *Compositionality-Spawn*:
assumes *WWs-threads*: *WHATWHERE-Secure V*
assumes *uniPPspawn*: *unique-PPc* (*spawn*$_\iota$ *V*)
shows *WHATWHERE-Secure* [*spawn*$_\iota$ *V*]
⟨*proof*⟩

theorem *Compositionality-If*:
assumes *dind*: $\forall\, d.\ b \equiv_d b$
assumes *WWs-branch1*: *WHATWHERE-Secure* [*c1*]
assumes *WWs-branch2*: *WHATWHERE-Secure* [*c2*]
assumes *uniPPif*: *unique-PPc* (*if*$_\iota$ *b then c1 else c2 fi*)
shows *WHATWHERE-Secure* [*if*$_\iota$ *b then c1 else c2 fi*]
⟨*proof*⟩

theorem *Compositionality-While*:
assumes *dind*: $\forall\, d.\ b \equiv_d b$
assumes *WWs-body*: *WHATWHERE-Secure* [*c*]
assumes *uniPPwhile*: *unique-PPc* (*while*$_\iota$ *b do c od*)
shows *WHATWHERE-Secure* [*while*$_\iota$ *b do c od*]
⟨*proof*⟩

end

end

4 Security type system

4.1 Abstract security type system with soundness proof

We formalize an abstract version of the type system in [2] using locales [1]. Our formalization of the type system is abstract in the sense that the rules specify abstract semantic side conditions on the expressions within a command that satisfy for proving the soundness of the rules. That is, it can be instantiated with different syntactic approximations for these semantic

side conditions in order to achieve a type system for a concrete language for Boolean and arithmetic expressions. Obtaining a soundness proof for such a concrete type system then boils down to proving that the concrete type system interprets the abstract type system.

We prove the soundness of the abstract type system by simply applying the compositionality results proven before.

theory *Type-System*
imports *Language-Composition*
begin

locale *Type-System* =
  WWP?: *WHATWHERE-Secure-Programs E BMap DA lH*
  for *E* :: (*'exp*, *'id*, *'val*) *Evalfunction*
  and *BMap* :: *'val* ⇒ *bool*
  and *DA* :: (*'id*, *'d*::*order*) *DomainAssignment*
  and *lH* :: (*'d*, *'exp*) *lHatches*
+
fixes
AssignSideCondition :: *'id* ⇒ *'exp* ⇒ *nat* ⇒ *bool*
and *WhileSideCondition* :: *'exp* ⇒ *bool*
and *IfSideCondition* ::
  'exp ⇒ (*'exp*,*'id*) *MWLsCom* ⇒ (*'exp*,*'id*) *MWLsCom* ⇒ *bool*
assumes *semAssignSC*: *AssignSideCondition x e ι* ⟹
  $e \equiv_{DA\ x,(htchLoc\ \iota)} e \wedge (\forall\, m\ m'\ d\ \iota'.\ (m \sim_{d,(htchLoc\ \iota')} m' \wedge$
  $[\![x :=_\iota e]\!](m) =_d [\![x :=_\iota e]\!](m'))$
  $\longrightarrow [\![x :=_\iota e]\!](m) \sim_{d,(htchLoc\ \iota')} [\![x :=_\iota e]\!](m'))$
and *semWhileSC*: *WhileSideCondition e* ⟹ ∀ *d*. *e* $\equiv_d$ *e*
and *semIfSC*: *IfSideCondition e c1 c2* ⟹ ∀ *d*. *e* $\equiv_d$ *e*
begin

— Security typing rules for the language commands
inductive
ComSecTyping :: (*'exp*, *'id*) *MWLsCom* ⇒ *bool*
  ($\vdash_{\mathcal{C}}$ -)
and *ComSecTypingL* :: (*'exp*,*'id*) *MWLsCom list* ⇒ *bool*
  ($\vdash_{\mathcal{V}}$ -)
where
Skip: $\vdash_{\mathcal{C}} skip_\iota$ |
Assign: [![*AssignSideCondition x e ι*]!] ⟹ $\vdash_{\mathcal{C}} x :=_\iota e$ |
Spawn: [![$\vdash_{\mathcal{V}} V$]!] ⟹ $\vdash_{\mathcal{C}} spawn_\iota\ V$ |
Seq: [![$\vdash_{\mathcal{C}} c1$; $\vdash_{\mathcal{C}} c2$]!] ⟹ $\vdash_{\mathcal{C}} c1;c2$ |
While: [![$\vdash_{\mathcal{C}} c$; *WhileSideCondition b*]!]
  ⟹ $\vdash_{\mathcal{C}} while_\iota\ b\ do\ c\ od$ |
If: [![$\vdash_{\mathcal{C}} c1$; $\vdash_{\mathcal{C}} c2$; *IfSideCondition b c1 c2*]!]
  ⟹ $\vdash_{\mathcal{C}} if_\iota\ b\ then\ c1\ else\ c2\ fi$ |
Parallel: [![∀ *i* < *length V*. $\vdash_{\mathcal{C}} V!i$]!] ⟹ $\vdash_{\mathcal{V}} V$

inductive-cases *parallel-cases*:
$\vdash_{\mathcal{V}} V$

definition *auxiliary-predicate*
where
auxiliary-predicate $V \equiv$ *unique-PPV* $V \longrightarrow$ *WHATWHERE-Secure* V

— soundness proof of abstract type system
theorem *ComSecTyping-single-is-sound*:
$[\![\vdash_{\mathcal{C}} c$; *unique-PPc* $c]\!]$
$\Longrightarrow$ *WHATWHERE-Secure* $[c]$
⟨*proof*⟩

theorem *ComSecTyping-list-is-sound*:
$[\![\vdash_{\mathcal{V}} V$; *unique-PPV* $V]\!] \Longrightarrow$ *WHATWHERE-Secure* V
⟨*proof*⟩

end

end

4.2 Example language for Boolean and arithmetic expressions

As and example, we provide a simple example language for instantiating the parameter *'exp* for the language for Boolean and arithmetic expressions (from the entry Strong-Security).

theory *Expr*
imports *Types*
begin

— type parameters:
— 'val: numbers, boolean constants....
— 'id: identifier names

type-synonym ('*val*) *operation* = *'val list* ⇒ *'val*

datatype (*dead 'id*, *dead 'val*) *Expr* =
Const 'val |
Var 'id |
Op 'val operation ((*'id*, *'val*) *Expr*) *list*

— defining a simple recursive evaluation function on this datatype
primrec *ExprEval* :: ((*'id*, *'val*) *Expr*, *'id*, *'val*) *Evalfunction*
and *ExprEvalL* :: ((*'id*, *'val*) *Expr*) *list* ⇒ (*'id*, *'val*) *State* ⇒ *'val list*

where
ExprEval (*Const v*) *m* = *v* |
ExprEval (*Var x*) *m* = (*m x*) |
ExprEval (*Op f arglist*) *m* = (*f* (*ExprEvalL arglist m*)) |

ExprEvalL [] *m* = [] |
ExprEvalL (*e*#*V*) *m* = (*ExprEval e m*)#(*ExprEvalL V m*)

end

4.3 Example interpretation of abstract security type system

Using the example instantiation of the language for Boolean and arithmetic expressions, we give an example instantiation of our abstract security type system, instantiating the parameter for domains *'d* with a two-level security lattice (from the entry Strong-Security).

theory *Domain-example*
imports *Expr*
begin

— When interpreting, we have to instantiate the type for domains. As an example, we take a type containing 'low' and 'high' as domains.

datatype *Dom* = *low* | *high*

instantiation *Dom* :: *order*
begin

definition
less-eq-Dom-def: *d1* ≤ *d2* = (*if d1* = *d2 then True*
else (*if d1* = *low then True else False*))

definition
less-Dom-def: *d1* < *d2* = (*if d1* = *d2 then False*
else (*if d1* = *low then True else False*))

instance ⟨*proof*⟩

end

end

theory *Type-System-example*
imports *Type-System Strong-Security.Expr Strong-Security.Domain-example*
begin

— When interpreting, we have to instantiate the type for domains. As an example,

we take a type containing 'low' and 'high' as domains.

consts *DA* :: (*'id*,*Dom*) *DomainAssignment*
consts *BMap* :: *'val* ⇒ *bool*
consts *lH* :: (*Dom*,(*'id*,*'val*) *Expr*) *lHatches*

— redefine all the abbreviations necessary for auxiliary lemmas with the correct parameter instantiation

abbreviation *MWLsStepsdet'* ::
(*('id,'val) Expr*, *'id*, *'val*, (*('id,'val) Expr*,*'id*) *MWLsCom*) *TLSteps-curry*
((*1*⟨-,/-⟩) →◁-▷/ (*1*⟨-,/-⟩) [*0*,*0*,*0*,*0*,*0*] *81*)
where
⟨*c1*,*m1*⟩ →◁α▷ ⟨*c2*,*m2*⟩ ≡
(*(c1,m1)*,α,*(c2,m2)*) ∈ *MWLs-semantics.MWLsSteps-det ExprEval BMap*

abbreviation *d-equal'* :: (*'id*, *'val*) *State*
⇒ *Dom* ⇒ (*'id*, *'val*) *State* ⇒ *bool*
((- =$_-$ -))
where
$m =_d m'$ ≡ *WHATWHERE.d-equal DA d m m'*

abbreviation *dH-equal'* :: (*'id*, *'val*) *State* ⇒ *Dom*
⇒ (*Dom*,(*'id*,*'val*) *Expr*) *Hatches*
⇒ (*'id*, *'val*) *State* ⇒ *bool*
((- $\sim_{-,-}$ -))
where
$m \sim_{d,H} m'$ ≡ *WHATWHERE.dH-equal ExprEval DA d H m m'*

abbreviation *NextMem'* :: ((*'id,'val*) *Expr*, *'id*) *MWLsCom*
⇒ (*'id,'val*) *State* ⇒ (*'id,'val*) *State*
(⟦-⟧'(-'))
where
⟦*c*⟧(*m*)
≡ *WHATWHERE.NextMem* (*MWLs-semantics.MWLsSteps-det ExprEval BMap*)
c m

abbreviation *dH-indistinguishable'* :: (*'id,'val*) *Expr* ⇒ *Dom*
⇒ (*Dom*,(*'id,'val*) *Expr*) *Hatches* ⇒ (*'id,'val*) *Expr* ⇒ *bool*
((- $\equiv_{-,-}$ -))
where
$e1 \equiv_{d,H} e2$
≡ *WHATWHERE-Secure-Programs.dH-indistinguishable ExprEval DA d H e1 e2*

abbreviation *htchLoc* :: *nat* ⇒ (*Dom*, (*'id,'val*) *Expr*) *Hatches*
where
htchLoc ι ≡ *WHATWHERE.htchLoc lH* ι

— Security typing rules for expressions
inductive
ExprSecTyping :: (*Dom*, (*'id*,*'val*) *Expr*) *Hatches*
⇒ (*'id*, *'val*) *Expr* ⇒ *Dom* ⇒ *bool*
(- $\vdash_{\mathcal{E}}$ - : -)
for *H* :: (*Dom*, (*'id*, *'val*) *Expr*) *Hatches*
where
Consts: $H \vdash_{\mathcal{E}} (Const\ v) : d$ |
Vars: $DA\ x = d \Longrightarrow H \vdash_{\mathcal{E}} (Var\ x) : d$ |
Hatch: $(d,e) \in H \Longrightarrow H \vdash_{\mathcal{E}} e : d$ |
Ops: $[\![\forall i < length\ arglist.\ H \vdash_{\mathcal{E}} (arglist!i) : (dl!i) \wedge (dl!i) \leq d]\!]$
$\Longrightarrow H \vdash_{\mathcal{E}} (Op\ f\ arglist) : d$

— function substituting a certain expression with another expression in expressions
primrec *Subst* :: (*'id*, *'val*) *Expr* ⇒ (*'id*, *'val*) *Expr*
⇒ (*'id*, *'val*) *Expr* ⇒ (*'id*, *'val*) *Expr*
(-<-\->)
and *SubstL* :: (*'id*, *'val*) *Expr list* ⇒ (*'id*, *'val*) *Expr*
⇒ (*'id*, *'val*) *Expr* ⇒ (*'id*, *'val*) *Expr list*
where
$(Const\ v)\langle e1 \backslash e2\rangle = (if\ e1=(Const\ v)\ then\ e2\ else\ (Const\ v))$ |
$(Var\ x)\langle e1 \backslash e2\rangle = (if\ e1=(Var\ x)\ then\ e2\ else\ (Var\ x))$ |
$(Op\ f\ arglist)\langle e1 \backslash e2\rangle = (if\ e1=(Op\ f\ arglist)\ then\ e2\ else$
$(Op\ f\ (SubstL\ arglist\ e1\ e2)))$ |

SubstL [] *e1 e2* = [] |
$SubstL\ (e\#V)\ e1\ e2 = (e\langle e1 \backslash e2\rangle)\#(SubstL\ V\ e1\ e2)$

definition *SubstClosure* :: *'id* ⇒ (*'id*, *'val*) *Expr* ⇒ *bool*
where
$SubstClosure\ x\ e \equiv \forall (d',e',\iota') \in lH.\ (d',e'\langle (Var\ x) \backslash e\rangle,\iota') \in lH$

definition *synAssignSC* :: *'id* ⇒ (*'id*, *'val*) *Expr* ⇒ *nat* ⇒ *bool*
where
$synAssignSC\ x\ e\ \iota \equiv \exists d.\ ((htchLoc\ \iota) \vdash_{\mathcal{E}} e : d \wedge d \leq DA\ x)$
$\wedge\ (SubstClosure\ x\ e)$

definition *synWhileSC* :: (*'id*, *'val*) *Expr* ⇒ *bool*
where
$synWhileSC\ e \equiv (\exists d.\ (\{\} \vdash_{\mathcal{E}} e : d) \wedge (\forall d'.\ d \leq d'))$

definition *synIfSC* :: (*'id*, *'val*) *Expr*
⇒ ((*'id*, *'val*) *Expr*, *'id*) *MWLsCom*
⇒ ((*'id*, *'val*) *Expr*, *'id*) *MWLsCom* ⇒ *bool*
where
$synIfSC\ e\ c1\ c2 \equiv \exists d.\ (\{\} \vdash_{\mathcal{E}} e : d \wedge (\forall d'.\ d \leq d'))$

— auxiliary lemma for locale interpretation (theorem 7 in original paper)
lemma *ExprTypable-with-smallerd-implies-dH-indistinguishable*:
$[\![H \vdash_{\mathcal{E}} e : d';\ d' \leq d]\!] \Longrightarrow e \equiv_{d,H} e$
⟨*proof*⟩
lemma *substexp-substmem*:
ExprEval e'<*Var* x\e> m = *ExprEval* e' (m(x := *ExprEval* e m))
$\wedge$ *ExprEvalL* (*SubstL elist* (*Var* x) e) m
= *ExprEvalL elist* (m(x := *ExprEval* e m))
⟨*proof*⟩
lemma *SubstClosure-implications*:
$[\![$ *SubstClosure* x e; $m \sim_{d,(htchLoc\ \iota')} m'$;
$[\![x :=_\iota e]\!](m) =_d [\![x :=_\iota e]\!](m')$ $]\!]$
$\Longrightarrow [\![x :=_\iota e]\!](m) \sim_{d,(htchLoc\ \iota')} [\![x :=_\iota e]\!](m')$
⟨*proof*⟩
interpretation *Type-System-example*: *Type-System ExprEval BMap DA lH*
synAssignSC synWhileSC synIfSC
⟨*proof*⟩

end

References

[1] C. Ballarin. Locales and Locale Expressions in Isabelle/Isar. In S. Berardi, M. Coppo, and F. Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2003.

[2] A. Lux, H. Mantel, and M. Perner. Scheduler-independent declassification. In *Proceedings of the 11th International Conference on Mathematics of Program Construction (MPC)*, LNCS 7342, pages 25–47. Springer, 2012.