

A Formalization of Declassification with WHAT&WHERE-Security

Sylvia Grewe, Alexander Lux, Heiko Mantel, Jens Sauer

April 11, 2026

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	2
1.1	Type synonyms	2
2	WHAT&WHERE-security	4
2.1	Definition of WHAT&WHERE-security	4
2.2	Proof technique for compositionality results	7
2.3	Proof of parallel compositionality	8

3	Example language and compositionality proofs	9
3.1	Example language with dynamic thread creation	9
3.2	Proofs of atomic compositionality results	13
3.3	Proofs of non-atomic compositionality results	14
4	Security type system	15
4.1	Abstract security type system with soundness proof	15
4.2	Example language for Boolean and arithmetic expressions . .	17
4.3	Example interpretation of abstract security type system . . .	18

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 =
```

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

— type for program states (including the set of commands and a symbol for terminating - None)

type-synonym $\text{'com ProgramState} = \text{'com option}$

— type for configurations with program states

type-synonym $(\text{'id, 'val, 'com}) \text{ PConfig} =$
 $\text{'com ProgramState} \times (\text{'id, 'val}) \text{ State}$

— type for labels with a list of spawned threads

type-synonym $\text{'com Label} = \text{'com list}$

— type for step relations from single commands to a program state, with a label

type-synonym $(\text{'exp, 'id, 'val, 'com}) \text{ TLSteps} =$
 $((\text{'id, 'val, 'com}) \text{ TConfig} \times \text{'com Label}$
 $\times (\text{'id, 'val, 'com}) \text{ PConfig}) \text{ set}$

— curried version of previously defined type

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

— type for step relations from thread pools to thread pools

type-synonym $(\text{'exp, 'id, 'val, 'com}) \text{ TPSteps} =$
 $((\text{'id, 'val, 'com}) \text{ TPCConfig} \times (\text{'id, 'val, 'com}) \text{ TPCConfig}) \text{ set}$

— curried version of previously defined type

type-synonym $(\text{'exp, 'id, 'val, 'com}) \text{ TPSteps-curry} =$
 $\text{'com list} \Rightarrow (\text{'id, 'val}) \text{ State} \Rightarrow \text{'com list} \Rightarrow (\text{'id, 'val}) \text{ State} \Rightarrow \text{bool}$

— define type of step relations for single threads to thread pools

type-synonym $(\text{'exp, 'id, 'val, 'com}) \text{ TSteps} =$
 $((\text{'id, 'val, 'com}) \text{ TConfig} \times (\text{'id, 'val, 'com}) \text{ TPCConfig}) \text{ set}$

— define the same type as TSteps, but in a curried version (allowing syntax abbreviations)

type-synonym $(\text{'exp, 'id, 'val, 'com}) \text{ TSteps-curry} =$
 $\text{'com} \Rightarrow (\text{'id, 'val}) \text{ State} \Rightarrow \text{'com list} \Rightarrow (\text{'id, 'val}) \text{ State} \Rightarrow \text{bool}$

— type for simple domain assignments; 'd has to be an instance of order (partial order)

type-synonym $(\text{'id, 'd}) \text{ DomainAssignment} = \text{'id} \Rightarrow \text{'d::order}$

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

— type for escape hatches

type-synonym $(\text{'d, 'exp}) \text{ Hatch} = \text{'d} \times \text{'exp}$

— 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 × 'exp × *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 ⇒ *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-equal* :: 'd::order ⇒ ('id, 'val) *State*
⇒ ('id, 'val) *State* ⇒ *bool*

where

d-equal d m m' ≡ ∀ x. ((*DA* x) ≤ d ⟶ (m x) = (m' x))

abbreviation *d-equal'* :: ('id, 'val) *State*
⇒ 'd::order ⇒ ('id, 'val) *State* ⇒ *bool*
(⟨(- =. -)⟩)

where

m =_d *m'* ≡ *d-equal* d m m'

— transitivity of d-equality

lemma *d-equal-trans*:

[[*m* =_d *m'*; *m'* =_d *m''*]] ⟹ *m* =_d *m''*
⟨*proof*⟩

abbreviation $SRabbr :: ('exp, 'id, 'val, 'com) TLSteps\text{-}curry$
 $(\langle (1\langle -,/- \rangle) \rightarrow \triangleleft \triangleright / (1\langle -,/- \rangle) \rangle [0,0,0,0,0] \delta 1)$
where
 $\langle c, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m' \rangle \equiv ((c, m), \alpha, (p, m')) \in SR$

— function for obtaining the unique memory (state) after one step for a command and a memory (state)

definition $NextMem :: 'com \Rightarrow ('id, 'val) State \Rightarrow ('id, 'val) State$
 $(\langle \llbracket - \rrbracket'(-) \rangle)$
where
 $\llbracket c \rrbracket(m) \equiv (THE\ m'. (\exists p\ \alpha. \langle c, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m' \rangle))$

— function getting all escape hatches for some location

definition $htchLoc :: nat \Rightarrow ('d, 'exp) Hatches$
where
 $htchLoc\ \iota \equiv \{(d, e). (d, e, \iota) \in lH\}$

— function for getting all escape hatches for some set of locations

definition $htchLocSet :: nat\ set \Rightarrow ('d, 'exp) Hatches$
where
 $htchLocSet\ PP \equiv \bigcup \{h. (\exists \iota \in PP. h = htchLoc\ \iota)\}$

— predicate for (d,H)-equality

definition $dH\text{-}equal :: 'd \Rightarrow ('d, 'exp) Hatches$
 $\Rightarrow ('id, 'val) State \Rightarrow ('id, 'val) State \Rightarrow bool$
where
 $dH\text{-}equal\ d\ H\ m\ m' \equiv (m =_d m' \wedge$
 $(\forall (d', e) \in H. (d' \leq d \longrightarrow (E\ e\ m = E\ e\ m'))))$

abbreviation $dH\text{-}equal' :: ('id, 'val) State \Rightarrow 'd \Rightarrow ('d, 'exp) Hatches$
 $\Rightarrow ('id, 'val) State \Rightarrow bool$
 $(\langle (-\ \sim_{d,H}\ -) \rangle)$

where
 $m \sim_{d,H} m' \equiv dH\text{-}equal\ d\ H\ m\ m'$

— predicate indicating that a command is not a d-declassification command

definition $NDC :: 'd \Rightarrow 'com \Rightarrow bool$
where
 $NDC\ d\ c \equiv (\forall m\ m'. m =_d m' \longrightarrow \llbracket c \rrbracket(m) =_d \llbracket c \rrbracket(m'))$

— predicate indicating an 'immediate d-declassification command' for a set of escape hatches

definition $IDC :: 'd \Rightarrow 'com \Rightarrow ('d, 'exp) Hatches \Rightarrow bool$
where
 $IDC\ d\ c\ H \equiv (\exists m\ m'. m =_d m' \wedge$
 $(\neg \llbracket c \rrbracket(m) =_d \llbracket c \rrbracket(m')))$

$$\wedge (\forall m m'. m \sim_{d,H} m' \longrightarrow \llbracket c \rrbracket(m) =_d \llbracket c \rrbracket(m'))$$

definition *stepResultsinR* :: 'com ProgramState \Rightarrow 'com ProgramState
 \Rightarrow 'com Bisimulation-type \Rightarrow bool

where

$$\text{stepResultsinR } p \ p' \ R \equiv (p = \text{None} \wedge p' = \text{None}) \vee \\ (\exists c \ c'. (p = \text{Some } c \wedge p' = \text{Some } c' \wedge ([c],[c']) \in R))$$

definition *dhequality-alternative* :: 'd \Rightarrow nat set \Rightarrow nat
 \Rightarrow ('id, 'val) State \Rightarrow ('id, 'val) State \Rightarrow bool

where

$$\text{dhequality-alternative } d \ PP \ \iota \ m \ m' \equiv m \sim_{d,(\text{htchLocSet } PP)} m' \vee \\ (\neg (\text{htchLoc } \iota) \subseteq (\text{htchLocSet } PP))$$

definition *Strong-dlHPP-Bisimulation* :: 'd \Rightarrow nat set
 \Rightarrow 'com Bisimulation-type \Rightarrow bool

where

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

— predicate to define when a program is strongly secure

definition *WHATWHERE-Secure* :: 'com list \Rightarrow bool

where

$$\text{WHATWHERE-Secure } V \equiv (\forall d \ PP. \\ (\exists R. \text{Strong-dlHPP-Bisimulation } d \ PP \ R \wedge (V, V) \in R))$$

— 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*:

$$\wedge V \ V' \ m1 \ m1' \ m2 \ p \ i \ \alpha. \llbracket \text{Strong-dlHPP-Bisimulation } d \ PP \ R; \\ i < \text{length } V; (V, V') \in R; \\ \langle V!i, m1 \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m2 \rangle; m1 \sim_{d,(\text{htchLocSet } PP)} m1' \rrbracket \\ \Longrightarrow (\exists p' \ \alpha' \ m2'. \langle V!i, m1' \rangle \rightarrow \triangleleft \alpha' \triangleright \langle p', m2' \rangle \\ \wedge \text{stepResultsinR } p \ p' \ R \wedge (\alpha, \alpha') \in R \wedge \\ (\text{dhequality-alternative } d \ PP \ (pp \ (V!i)) \ m2 \ m2')) \\ \langle \text{proof} \rangle$$

lemma *strongdlHPPB-NDCIDCaux*:

$\wedge V V' i. \llbracket \text{Strong-dlHPP-Bisimulation } d \text{ PP } R; \text{ (} V, V' \text{)} \in R; i < \text{length } V \rrbracket$
 $\implies (\text{NDC } d (V!i) \vee \text{IDC } d (V!i) (\text{htchLoc } (pp (V!i))))$
 <proof>

lemma *WHATWHERE-empty*:
WHATWHERE-Secure \square
 <proof>

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** *SdlHPPB* \equiv *Strong-dlHPP-Bisimulation*

— define the ‘reflexive part’ of a relation (sets of elements which are related with themselves by the given relation)

definition *Areft* $:: ('a \times 'a) \text{ set} \Rightarrow 'a \text{ set}$
where
Areft $R = \{e. (e, e) \in R\}$

lemma *commonAreft-subset-commonDomain*:
 $(\text{Areft } R1 \cap \text{Areft } R2) \subseteq (\text{Domain } R1 \cap \text{Domain } R2)$
 <proof>

definition *disj-dlHPP-Bisimulation-Up-To-R'* $::$
 $'d \Rightarrow \text{nat set} \Rightarrow 'com \text{ Bisimulation-type}$
 $\Rightarrow 'com \text{ Bisimulation-type} \Rightarrow \text{bool}$

where

disj-dlHPP-Bisimulation-Up-To-R' $d \text{ PP } R' R \equiv$
 $\text{SdlHPPB } d \text{ PP } R' \wedge (\text{sym } R) \wedge (\text{trans } R)$
 $\wedge (\forall (V, V') \in R. \text{length } V = \text{length } V') \wedge$
 $(\forall (V, V') \in R. \forall i < \text{length } V.$
 $((\text{NDC } d (V!i)) \vee$
 $(\text{IDC } d (V!i) (\text{htchLoc } (pp (V!i)))))) \wedge$
 $(\forall (V, V') \in R. \forall i < \text{length } V. \forall m1 m1' m2 \alpha p.$
 $(\langle V!i, m1 \rangle \rightarrow \langle \alpha \rangle \langle p, m2 \rangle \wedge m1 \sim_{d, (\text{htchLocSet } PP)} m1')$

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

— 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*:

$\llbracket disj-dlHPP-Bisimulation-Up-To-R'\ d\ PP\ R'\ R;$
 $Arefl\ R \cap Arefl\ R' \subseteq \{\}\} \rrbracket$
 $\implies SdlHPPB\ d\ PP\ (R \cup R')$

<proof>

lemma *Union-Strong-dlHPP-Bisim*:

$\llbracket SdlHPPB\ d\ PP\ R; SdlHPPB\ d\ PP\ R';$
 $Arefl\ R \cap Arefl\ R' \subseteq \{\}\} \rrbracket$
 $\implies SdlHPPB\ d\ PP\ (R \cup R')$

<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  $\Rightarrow$  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:
   $\llbracket \forall i < \text{length } V. \text{WHATWHERE-Secure } [V!i]; \text{unique-PPV } V \rrbracket$ 
   $\implies$  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 (skipl) =  $\iota$  |
  pp (x :=l e) =  $\iota$  |
  pp (c1;c2) = pp c1 |
  pp (ifl b then c1 else c2 fi) =  $\iota$  |
  pp (whilel b do c od) =  $\iota$  |
  pp (spawnl V) =  $\iota$ 

```

— 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 (skipl) = [l] |
  PPc (x :=l e) = [l] |
  PPc (c1;c2) = (PPc c1) @ (PPc c2) |
  PPc (ifl b then c1 else c2 fi) = [l] @ (PPc c1) @ (PPc c2) |
  PPc (whilel b do c od) = [l] @ (PPc c) |
  PPc (spawnl V) = [l] @ (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 \Rightarrow bool

where

unique-PPV V = *distinct* (PPV V)

lemma *PPc-nonempty*: PPc c \neq []

<proof>

lemma *unique-c-uneq*: set (PPc c) \cap set (PPc c') = {} \implies c \neq c'

<proof>

lemma *V-nonempty-PPV-nonempty*: V \neq [] \implies PPV V \neq []

<proof>

lemma *unique-V-uneq*:

$\llbracket V \neq []; V' \neq []; \text{set} (PPV V) \cap \text{set} (PPV V') = \{\} \rrbracket \implies V \neq V'$

<proof>

lemma *PPc-in-PPV*: c \in set V \implies set (PPc c) \subseteq set (PPV V)

<proof>

lemma *listindices-aux*: i < length V \implies (V!i) \in set V

<proof>

lemma *PPc-in-PPV-version*:

i < length V \implies set (PPc (V!i)) \subseteq set (PPV V)

<proof>

lemma *uniPPV-uniPPc*: *unique-PPV* V \implies (\forall i < length V. *unique-PPc* (V!i))

<proof>

locale *MWLS-semantic* =

fixes E :: ('exp, 'id, 'val) Evalfunction

and BMap :: 'val \Rightarrow 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

($\langle 1 \langle -, - \rangle \rightarrow \triangleleft \triangleright / (1 \langle -, - \rangle) \rangle [0, 0, 0, 0, 0]$ 81)

where

$\langle c1, m1 \rangle \rightarrow \triangleleft \alpha \triangleright \langle c2, m2 \rangle \equiv ((c1, m1), \alpha, (c2, m2)) \in \text{MWLSSteps-det} \mid$

skip: $\langle \text{skip}_\iota, m \rangle \rightarrow \triangleleft [] \triangleright \langle \text{None}, m \rangle \mid$

assign: (E e m) = v \implies

$\langle x :=_\iota e, m \rangle \rightarrow \triangleleft [] \triangleright \langle \text{None}, m(x := v) \rangle \mid$

seq1: $\langle c1, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle \text{None}, m' \rangle \implies$

$\langle c1;c2,m \rangle \rightarrow \triangleleft \alpha \triangleright \langle \text{Some } c2,m^\wedge \rangle \mid$
seq2: $\langle c1,m \rangle \rightarrow \triangleleft \alpha \triangleright \langle \text{Some } c1',m^\wedge \rangle \implies$
 $\langle c1;c2,m \rangle \rightarrow \triangleleft \alpha \triangleright \langle \text{Some } (c1';c2),m^\wedge \rangle \mid$
iftrue: $BMap (E b m) = True \implies$
 $\langle \text{if}_\iota b \text{ then } c1 \text{ else } c2 \text{ fi},m \rangle \rightarrow \triangleleft \square \triangleright \langle \text{Some } c1,m \rangle \mid$
iffalse: $BMap (E b m) = False \implies$
 $\langle \text{if}_\iota b \text{ then } c1 \text{ else } c2 \text{ fi},m \rangle \rightarrow \triangleleft \square \triangleright \langle \text{Some } c2,m \rangle \mid$
whiletrue: $BMap (E b m) = True \implies$
 $\langle \text{while}_\iota b \text{ do } c \text{ od},m \rangle \rightarrow \triangleleft \square \triangleright \langle \text{Some } (c;(\text{while}_\iota b \text{ do } c \text{ od})),m \rangle \mid$
whilefalse: $BMap (E b m) = False \implies$
 $\langle \text{while}_\iota b \text{ do } c \text{ od},m \rangle \rightarrow \triangleleft \square \triangleright \langle \text{None},m \rangle \mid$
spawn: $\langle \text{spawn}_\iota V,m \rangle \rightarrow \triangleleft V \triangleright \langle \text{None},m \rangle$

inductive-cases *MWLSSteps-det-cases*:

$\langle \text{skip}_\iota,m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p,m^\wedge \rangle$
 $\langle x :=_\iota e,m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p,m^\wedge \rangle$
 $\langle c1;c2,m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p,m^\wedge \rangle$
 $\langle \text{if}_\iota b \text{ then } c1 \text{ else } c2 \text{ fi},m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p,m^\wedge \rangle$
 $\langle \text{while}_\iota b \text{ do } c \text{ od},m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p,m^\wedge \rangle$
 $\langle \text{spawn}_\iota V,m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p,m^\wedge \rangle$

— non-deterministic, possibilistic system step (added for intuition, not used in the proofs)

inductive-set

MWLSSteps-ndet ::

$(\text{'exp, 'id, 'val, ('exp, 'id) MWLSCom) TPSteps}$

and *MWLSSteps-ndet'* ::

$(\text{'exp, 'id, 'val, ('exp, 'id) MWLSCom) TPSteps-curry}$

$(\langle (1 \langle -,/- \rangle) \Rightarrow / (1 \langle -,/- \rangle) \rangle [0,0,0,0] \ 81)$

where

$\langle V,m \rangle \Rightarrow \langle V',m^\wedge \rangle \equiv ((V,m),(V',m^\wedge)) \in \text{MWLSSteps-ndet} \mid$

stepthreadi1: $\langle ci,m \rangle \rightarrow \triangleleft \alpha \triangleright \langle \text{None},m^\wedge \rangle \implies$

$\langle cf @ [ci] @ ca,m \rangle \Rightarrow \langle cf @ \alpha @ ca,m^\wedge \rangle \mid$

stepthreadi2: $\langle ci,m \rangle \rightarrow \triangleleft \alpha \triangleright \langle \text{Some } c',m^\wedge \rangle \implies$

$\langle cf @ [ci] @ ca,m \rangle \Rightarrow \langle cf @ [c'] @ \alpha @ ca,m \rangle$

— lemma about existence and uniqueness of next memory of a step

lemma *nextmem-exists-and-unique*:

$\exists m' p \alpha. \langle c,m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p,m^\wedge \rangle$

$\wedge (\forall m''. (\exists p \alpha. \langle c,m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p,m'' \rangle) \longrightarrow m'' = m')$

<proof>

lemma *PPsc-of-step*:

$\llbracket \langle c,m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p,m^\wedge \rangle; \exists c'. p = \text{Some } c' \rrbracket$

$\implies \text{set } (PPc \text{ (the } p)) \subseteq \text{set } (PPc \ c)$

<proof>

lemma *PPsα-of-step*:

$\langle c, m \rangle \rightarrow \langle \alpha \rangle \langle p, m' \rangle$
 $\implies \text{set} (PPV \alpha) \subseteq \text{set} (PPc c)$
 $\langle \text{proof} \rangle$

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'*
 $\langle \llbracket - \rrbracket' (-) \rangle$
where
 $\llbracket c \rrbracket (m) \equiv \text{NextMem } c \ m$

— define when two expressions are indistinguishable with respect to a domain d

definition *d-indistinguishable* $:: 'd :: \text{order} \Rightarrow 'exp \Rightarrow 'exp \Rightarrow \text{bool}$
where
 $d\text{-indistinguishable } d \ e1 \ e2 \equiv \forall m \ m'. ((m =_d m') \longrightarrow ((E \ e1 \ m) = (E \ e2 \ m')))$

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

— symmetry of d -indistinguishable

lemma *d-indistinguishable-sym*:

$e \equiv_d e' \implies e' \equiv_d e$
 $\langle \text{proof} \rangle$

lemma *d-indistinguishable-trans*:

$\llbracket e \equiv_d e'; e' \equiv_d e'' \rrbracket \implies e \equiv_d e''$
 $\langle \text{proof} \rangle$

definition *dH-indistinguishable* $:: 'd \Rightarrow ('d, 'exp) \text{ Hatches} \Rightarrow 'exp \Rightarrow 'exp \Rightarrow \text{bool}$

where

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

abbreviation $dH\text{-indistinguishable}' :: 'exp \Rightarrow 'd$

$\Rightarrow ('d, 'exp) \text{ Hatches} \Rightarrow 'exp \Rightarrow bool$

$(\langle (- \equiv_{-, -} -) \rangle)$

where

$e1 \equiv_{d,H} e2 \equiv dH\text{-indistinguishable } d \ H \ e1 \ e2$

lemma $empH\text{-implies-}dH\text{indistinguishable-eq-}d\text{indistinguishable}$:

$(e \equiv_{d,\{\}} e') = (e \equiv_d e')$

$\langle proof \rangle$

theorem $WHATWHERE\text{-Secure-Skip}$:

$WHATWHERE\text{-Secure } [skip_\iota]$

$\langle proof \rangle$

lemma $semAssignSC\text{-aux}$:

assumes $dhind: e \equiv_{DA} x, (htchLoc \ \iota) \ e$

shows $NDC \ d \ (x :=_\iota e) \vee IDC \ d \ (x :=_\iota e) \ (htchLoc \ (pp \ (x :=_\iota e)))$

$\langle proof \rangle$

theorem $WHATWHERE\text{-Secure-Assign}$:

assumes $dhind: e \equiv_{DA} x, (htchLoc \ \iota) \ e$

assumes $dheq\text{-imp}: \forall m \ m' \ d \ \iota'. \ (m \sim_{d, (htchLoc \ \iota')} m' \wedge$

$\llbracket x :=_\iota e \rrbracket(m) =_d \llbracket x :=_\iota e \rrbracket(m')$

$\longrightarrow \llbracket x :=_\iota e \rrbracket(m) \sim_{d, (htchLoc \ \iota')} \llbracket x :=_\iota e \rrbracket(m')$

shows $WHATWHERE\text{-Secure } [x :=_\iota e]$

$\langle proof \rangle$

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 $Language\text{-Composition}$

imports $WHATWHERE\text{-Secure-Skip-Assign}$

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_l V)*

shows *WHATWHERE-Secure [spawn_l 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_l b then c1 else c2 fi)*

shows *WHATWHERE-Secure [if_l 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_l b do c od)*

shows *WHATWHERE-Secure [while_l 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  $\Rightarrow$  bool
  and DA :: ('id, 'd::order) DomainAssignment
  and lH :: ('d, 'exp) lHatches
+
fixes
  AssignSideCondition :: 'id  $\Rightarrow$  'exp  $\Rightarrow$  nat  $\Rightarrow$  bool
and WhileSideCondition :: 'exp  $\Rightarrow$  bool
and IfSideCondition ::
  'exp  $\Rightarrow$  ('exp, 'id) MWLSCom  $\Rightarrow$  ('exp, 'id) MWLSCom  $\Rightarrow$  bool
assumes semAssignSC: AssignSideCondition x e  $\iota$   $\Longrightarrow$ 
   $e \equiv_{DA} x, (htchLoc \iota) e \wedge (\forall m m' d \iota'. (m \sim_{d, (htchLoc \iota)} m' \wedge$ 
   $\llbracket x :=_{\iota} e \rrbracket(m) =_d \llbracket x :=_{\iota} e \rrbracket(m'))$ 
   $\longrightarrow \llbracket x :=_{\iota} e \rrbracket(m) \sim_{d, (htchLoc \iota)} \llbracket x :=_{\iota} e \rrbracket(m'))$ 
and semWhileSC: WhileSideCondition e  $\Longrightarrow \forall d. e \equiv_d e$ 
and semIfSC: IfSideCondition e c1 c2  $\Longrightarrow \forall d. e \equiv_d e$ 
begin

```

— Security typing rules for the language commands

```

inductive
  ComSecTyping :: ('exp, 'id) MWLSCom  $\Rightarrow$  bool
  ( $\vdash_{\mathcal{C}}$   $\rightarrow$ )
and ComSecTypingL :: ('exp, 'id) MWLSCom list  $\Rightarrow$  bool
  ( $\vdash_{\mathcal{V}}$   $\rightarrow$ )
where
  Skip:  $\vdash_{\mathcal{C}} skip_{\iota}$  |
  Assign:  $\llbracket AssignSideCondition x e \iota \rrbracket \Longrightarrow \vdash_{\mathcal{C}} x :=_{\iota} e$  |
  Spawn:  $\llbracket \vdash_{\mathcal{V}} V \rrbracket \Longrightarrow \vdash_{\mathcal{C}} spawn_{\iota} V$  |
  Seq:  $\llbracket \vdash_{\mathcal{C}} c1; \vdash_{\mathcal{C}} c2 \rrbracket \Longrightarrow \vdash_{\mathcal{C}} c1; c2$  |
  While:  $\llbracket \vdash_{\mathcal{C}} c; WhileSideCondition b \rrbracket$ 
     $\Longrightarrow \vdash_{\mathcal{C}} while_{\iota} b do c od$  |
  If:  $\llbracket \vdash_{\mathcal{C}} c1; \vdash_{\mathcal{C}} c2; IfSideCondition b c1 c2 \rrbracket$ 
     $\Longrightarrow \vdash_{\mathcal{C}} if_{\iota} b then c1 else c2 fi$  |
  Parallel:  $\llbracket \forall i < length V. \vdash_{\mathcal{C}} V!i \rrbracket \Longrightarrow \vdash_{\mathcal{V}} V$ 

```

inductive-cases *parallel-cases*:

$\vdash_{\mathcal{V}} V$

definition *auxiliary-predicate*

where

auxiliary-predicate $V \equiv \text{unique-PPV } V \longrightarrow \text{WHATWHERE-Secure } V$

— soundness proof of abstract type system

theorem *ComSecTyping-single-is-sound*:

$\llbracket \vdash_{\mathcal{C}} c; \text{unique-PPc } c \rrbracket$

$\implies \text{WHATWHERE-Secure } [c]$

<proof>

theorem *ComSecTyping-list-is-sound*:

$\llbracket \vdash_{\mathcal{V}} V; \text{unique-PPV } V \rrbracket \implies \text{WHATWHERE-Secure } V$

<proof>

end

end

4.2 Example language for Boolean and arithmetic expressions

As an 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* \Rightarrow *'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* \Rightarrow (*'id*, *'val*) *State* \Rightarrow *'val list*

where

$ExprEval (Const v) m = v \mid$
 $ExprEval (Var x) m = (m x) \mid$
 $ExprEval (Op f arglist) m = (f (ExprEvalL arglist m)) \mid$

$ExprEvalL [] m = [] \mid$
 $ExprEvalL (e\#V) m = (ExprEval e m)\#(ExprEvalL V m)$

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: $d1 \leq d2 = (if\ d1 = d2\ then\ True$
 $\quad else\ (if\ d1 = low\ then\ True\ else\ False))$

definition

less-Dom-def: $d1 < d2 = (if\ d1 = d2\ then\ False$
 $\quad else\ (if\ d1 = low\ then\ True\ else\ False))$

instance $\langle proof \rangle$

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 $BMap :: 'val \Rightarrow bool$
consts $lH :: (Dom, ('id, 'val) Expr) lHatches$

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

abbreviation $MWLSStepsdet' ::$
 $(('id, 'val) Expr, 'id, 'val, (('id, 'val) Expr, 'id) MWLsCom) TLSteps-curry$
 $(\langle (1 \langle -, / - \rangle) \rightarrow \langle \cdot \rangle / (1 \langle -, / - \rangle) \rangle [0, 0, 0, 0] 81)$
where
 $\langle c1, m1 \rangle \rightarrow \langle \alpha \rangle \langle c2, m2 \rangle \equiv$
 $((c1, m1), \alpha, (c2, m2)) \in MWLs-semantics.MWLSSteps-det ExprEval BMap$

abbreviation $d-equal' :: ('id, 'val) State$
 $\Rightarrow Dom \Rightarrow ('id, 'val) State \Rightarrow bool$
 $(\langle (- = -) \rangle)$
where
 $m \equiv_d m' \equiv WHATWHERE.d-equal DA d m m'$

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

abbreviation $NextMem' :: (('id, 'val) Expr, 'id) MWLsCom$
 $\Rightarrow ('id, 'val) State \Rightarrow ('id, 'val) State$
 $(\langle \llbracket - \rrbracket'(-) \rangle)$
where
 $\llbracket c \rrbracket(m)$
 $\equiv WHATWHERE.NextMem (MWLs-semantics.MWLSSteps-det ExprEval BMap)$
 $c m$

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

abbreviation $htchLoc :: nat \Rightarrow (Dom, ('id, 'val) Expr) Hatches$
where
 $htchLoc \iota \equiv WHATWHERE.htchLoc lH \iota$

— Security typing rules for expressions

inductive

ExprSecTyping :: (*Dom*, ('*id*', '*val*') *Expr*) *Hatches*
 \Rightarrow ('*id*', '*val*') *Expr* \Rightarrow *Dom* \Rightarrow *bool*

($\langle \cdot \vdash_{\mathcal{E}} \cdot \rangle$)

for *H* :: (*Dom*, ('*id*', '*val*') *Expr*) *Hatches*

where

Consts: $H \vdash_{\mathcal{E}} (\text{Const } v) : d \mid$

Vars: $DA \ x = d \implies H \vdash_{\mathcal{E}} (\text{Var } x) : d \mid$

Hatch: $(d, e) \in H \implies H \vdash_{\mathcal{E}} e : d \mid$

Ops: $\llbracket \forall i < \text{length } \text{arglist}. H \vdash_{\mathcal{E}} (\text{arglist}!i) : (d!i) \wedge (d!i) \leq d \rrbracket$
 $\implies H \vdash_{\mathcal{E}} (\text{Op } f \ \text{arglist}) : d$

— function substituting a certain expression with another expression in expressions

primrec *Subst* :: ('*id*', '*val*') *Expr* \Rightarrow ('*id*', '*val*') *Expr*

\Rightarrow ('*id*', '*val*') *Expr* \Rightarrow ('*id*', '*val*') *Expr*

($\langle \cdot \leftarrow \cdot \rightarrow \rangle$)

and *SubstL* :: ('*id*', '*val*') *Expr list* \Rightarrow ('*id*', '*val*') *Expr*

\Rightarrow ('*id*', '*val*') *Expr* \Rightarrow ('*id*', '*val*') *Expr list*

where

$(\text{Const } v) \langle e1 \setminus e2 \rangle = (\text{if } e1 = (\text{Const } v) \text{ then } e2 \text{ else } (\text{Const } v)) \mid$

$(\text{Var } x) \langle e1 \setminus e2 \rangle = (\text{if } e1 = (\text{Var } x) \text{ then } e2 \text{ else } (\text{Var } x)) \mid$

$(\text{Op } f \ \text{arglist}) \langle e1 \setminus e2 \rangle = (\text{if } e1 = (\text{Op } f \ \text{arglist}) \text{ then } e2 \text{ else } (\text{Op } f \ (\text{SubstL } \text{arglist } e1 \ e2))) \mid$

SubstL $\llbracket e1 \ e2 = \llbracket \mid$

SubstL ($e \# V$) $e1 \ e2 = (e \langle e1 \setminus e2 \rangle) \# (\text{SubstL } V \ e1 \ e2)$

definition *SubstClosure* :: '*id*' \Rightarrow ('*id*', '*val*') *Expr* \Rightarrow *bool*

where

SubstClosure $x \ e \equiv \forall (d', e', \iota') \in \text{lh}. (d', e' \langle (\text{Var } x) \setminus e \rangle, \iota') \in \text{lh}$

definition *synAssignSC* :: '*id*' \Rightarrow ('*id*', '*val*') *Expr* \Rightarrow *nat* \Rightarrow *bool*

where

synAssignSC $x \ e \ \iota \equiv \exists d. ((\text{htchLoc } \iota) \vdash_{\mathcal{E}} e : d \wedge d \leq DA \ x)$
 $\wedge (\text{SubstClosure } x \ e)$

definition *synWhileSC* :: ('*id*', '*val*') *Expr* \Rightarrow *bool*

where

synWhileSC $e \equiv (\exists d. (\{\} \vdash_{\mathcal{E}} e : d) \wedge (\forall d'. d \leq d'))$

definition *synIfSC* :: ('*id*', '*val*') *Expr*

\Rightarrow ('*id*', '*val*') *Expr*, '*id*') *MWLSCom*

\Rightarrow ('*id*', '*val*') *Expr*, '*id*') *MWLSCom* \Rightarrow *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*:

$$\llbracket H \vdash_{\mathcal{E}} e : d'; d' \leq d \rrbracket \implies e \equiv_{d,H} e$$

<proof>

lemma *substexp-substmem*:

$$\text{ExprEval } e' \langle \text{Var } x \setminus e \rangle m = \text{ExprEval } e' (m(x := \text{ExprEval } e m))$$

$$\wedge \text{ExprEvalL } (\text{SubstL } \text{elist } (\text{Var } x) e) m$$

$$= \text{ExprEvalL } \text{elist } (m(x := \text{ExprEval } e m))$$

<proof>

lemma *SubstClosure-implications*:

$$\llbracket \text{SubstClosure } x e; m \sim_{d,(\text{htchLoc } \iota)} m' \rrbracket;$$

$$\llbracket x :=_{\iota} e \rrbracket(m) =_d \llbracket x :=_{\iota} e \rrbracket(m')$$

$$\implies \llbracket x :=_{\iota} e \rrbracket(m) \sim_{d,(\text{htchLoc } \iota)} \llbracket x :=_{\iota} e \rrbracket(m')$$

<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.