A Formalization of Declassification with
WHAT&WHERE-Security

Sylvia Grewe, Alexander Lux, Heiko Mantel, Jens Sauer

May 28, 2015

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 11
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 for memory states - map ids to values
type-synonym ('id, 'val) State = 'id ⇒ 'val

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

— define configurations with threads as pair of commands and states
type-synonym ('id, 'val, 'com) TConfig = 'com × ('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) × (‘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 × (‘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 × ‘com Label
× (‘id, ‘val, ‘com) PSConfig) set

— curried version of previously defined type
type-synonym (‘exp, ‘id, ‘val, ‘com) TLSteps-curry =
‘com ⇒ (‘id, ‘val) State ⇒ ‘com Label ⇒ ‘com ProgramState
⇒ (‘id, ‘val) State ⇒ bool

— type for step relations from thread pools to thread pools
type-synonym (‘exp, ‘id, ‘val, ‘com) TPSteps =
((‘id, ‘val, ‘com) TPConfig × (‘id, ‘val, ‘com) TPConfig) set

— curried version of previously defined type
type-synonym (‘exp, ‘id, ‘val, ‘com) TPSteps-curry =
‘com list ⇒ (‘id, ‘val) State ⇒ ‘com list ⇒ (‘id, ‘val) State ⇒ bool

— define type of step relations for single threads to thread pools
type-synonym (‘exp, ‘id, ‘val, ‘com) TSteps =
((‘id, ‘val, ‘com) TConfig × (‘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-curry =
‘com ⇒ (‘id, ‘val) State ⇒ ‘com list ⇒ (‘id, ‘val) State ⇒ bool

— type for simple domain assignments; ’d has to be an instance of order (partial order)
type-synonym (‘id, ‘d) DomainAssignment = ‘id ⇒ ‘d::order

type-synonym ‘com Bisimulation-type = ((‘com list) × (‘com list)) set

— type for escape hatches
type-synonym (‘d, ‘exp) Hatch = ‘d × ‘exp
— type for sets of escape hatches
\textbf{type-synonym} (′d, ′exp) \textit{Hatches} = ((′d, ′exp) \textit{Hatch}) set

— type for local escape hatches
\textbf{type-synonym} (′d, ′exp) \textit{lHatch} = ′d × ′exp × \textit{nat}

— type for sets of local escape hatches
\textbf{type-synonym} (′d, ′exp) \textit{lHatches} = ((′d, ′exp) \textit{lHatch}) set

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).

\textit{theory} \textit{WHATWHERE-Security} \textit{imports} ../\textit{Strong-Security}/\textit{Types} \textit{begin}

\textit{locale} \textit{WHATWHERE} =
\textit{fixes} \textit{SR} :: (′exp, ′id, ′val, ′com) \textit{TLSteps}
\textit{and} \textit{E} :: (′exp, ′id, ′val) \textit{Evalfunction}
\textit{and} \textit{pp} :: ′com ⇒ \textit{nat}
\textit{and} \textit{DA} :: (′id, ′d::order) \textit{DomainAssignment}
\textit{and} \textit{IH} :: (′d::order, ′exp) \textit{lHatches}

\textit{begin}

— define when two states are indistinguishable for an observer on domain d
\textit{definition} \textit{d-equal} :: ′d::order ⇒ (′id, ′val) \textit{State} ⇒ bool
\textit{where}
\textit{d-equal} \textit{d m m′} ≡ ∀x. ((\textit{DA} x) ≤ \textit{d} → (m x) = (m′ x))

\textit{abbreviation} \textit{d-equal′} :: (′id, ′val) \textit{State} ⇒ ′d::order ⇒ (′id, ′val) \textit{State} ⇒ bool
\textit{where}
\textit{d-equal′} \textit{d m m′} ≡ \textit{d-equal} \textit{d m m′}

— transitivity of d-equality
\textit{lemma} \textit{d-equal-trans}:
\textit{[ [ m =_d m′ ; m′ =_d m″ ]] ⇒ m =_d m″}
by \((\text{simp add: } d\text{-equal-def})\)

abbreviation SRabbr :: \((\exp, \id, \val, \com) \Rightarrow \TLSteps\text{-curry} ((1\langle -,- \rangle) \rightarrow \lhd / (1\langle -,- \rangle) [0,0,0,0] 81)\) where 
\(\langle c,m \rangle \rightarrow \lhd \alpha \lds \langle p,m' \rangle \equiv ((c,m),\alpha,(p,m')) \in SR\)

— function for obtaining the unique memory (state) after one step for a command and a memory (state)
definition NextMem :: \('\com \Rightarrow ('\id, '\val) \Rightarrow ('\id, '\val)\) where 
\([c](m) \equiv (\THE \alpha, (\exists p \alpha. (c,m) \rightarrow \lhd \alpha \lds \langle p,m' \rangle))\)

— function getting all escape hatches for some location
definition htchLoc :: \('nat \Rightarrow ('d, 'exp) \Rightarrow \Hatches\) where 
\(htchLoc \iota \equiv \{(d,e). (d,e,i) \in \IH\}\)

— function for getting all escape hatches for some set of locations
definition htchLocSet :: \('nat set \Rightarrow ('d, 'exp) \Rightarrow \Hatches\) where 
\(htchLocSet PP \equiv \bigcup \{ h. (\exists \iota \in PP. h = htchLoc \iota) \}\)

— predicate for \((d,H)\)-equality
definition dH-equal :: \('d \Rightarrow ('d, 'exp) \Rightarrow \Hatches\Rightarrow \Hatches \Rightarrow bool\) where 
\(dH\text{-equal } d H m m' \equiv (m=d m' \land \forall (d',e) \in H. (d' \leq d \rightarrow (E e m = E e m'))))\)

abbreviation dH-equal' :: \('\id, '\val) \Rightarrow ('d, 'exp) \Rightarrow bool\) where 
\(m \sim_{d,H} m' \equiv \textit{dH-equal } d H m m'\)

— predicate indicating that a command is not a \(d\text{-declassification} command\)
definition NDC :: \('d \Rightarrow 'com \Rightarrow bool\) where 
\(NDC d c \equiv (\forall m m'. m =_d m' \rightarrow [c](m) =_d [c](m'))\)

— predicate indicating an ‘immediate \(d\text{-declassification} command’ for a set of escape hatches
definition IDC :: \('d \Rightarrow 'com \Rightarrow ('d, 'exp) \Rightarrow \Hatches \Rightarrow bool\) where 
\(IDC d c H \equiv (\exists m m'. m =_d m' \land \forall (d',e) \in H. (d' \leq d \rightarrow (E e m = E e m'))))\)
Lemma in meta logic (allows instantiating all the variables manually if necessary)

\[
(\neg [c](m) =_{d} [c](m')) \\
\land (\forall m', m \sim_d H m' \rightarrow [c](m) =_{d} [c](m'))
\]

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

**where**

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

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

**where**

\[
\text{dhequality-alternative} d PP i m m' \equiv m \sim_d (\text{htchLocSet PP}) m' \lor \\
(\neg (\text{htchLoc} i) \subseteq (\text{htchLocSet PP}))
\]

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

**where**

\[
\text{Strong-dlHPP-Bisimulation} d PP R \equiv \\
\text{(sym R)} \land (\text{trans R}) \land \\
(\forall (V,V') \in R. \, \text{length} V = \text{length} V') \land \\
(\forall (V,V') \in R. \, \forall i < \text{length} V. \\
\text{(IDC} d (V!i)) \lor \\
(\forall (V,V') \in R. \, \forall i < \text{length} V. \, \forall m1 m1' m2 \alpha, \alpha. \\
(V!i,m1) \rightarrow_{\langle \alpha \rangle} (p,m2) \land m1 \sim_d (\text{htchLocSet PP}) m1' \\
\rightarrow (\exists p' \alpha' m2'. (V!i,m1') \rightarrow_{\langle \alpha' \rangle} (p',m2') \land \\
\text{stepResultsinR} p p' R \land (\alpha,\alpha') \in R \land \\
(\text{dhequality-alternative} d PP (pp (V!i)) m2 m2'))) )
\]

— predicate to define when a program is strongly secure

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

**where**

\[
\text{WHATWHERE-Secure} V \equiv (\forall d PP. \\
(\exists l. \text{Strong-dlHPP-Bisimulation} d PP R \land (V,V) \in R))
\]

— 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:

\[
\forall V V' m1 m1' m2 p i \alpha. [\text{Strong-dlHPP-Bisimulation} d PP R; \\
i < \text{length} V; (V,V') \in R; \\
(V!i,m1) \rightarrow_{\langle \alpha \rangle} (p,m2); m1 \sim_d (\text{htchLocSet PP}) m1' ] \\
\Rightarrow (\exists p' \alpha' m2'. (V!i,m1') \rightarrow_{\langle \alpha' \rangle} (p',m2') \\
\land \text{stepResultsinR} p p' R \land (\alpha,\alpha') \in R \land \\
(\text{dhequality-alternative} d PP (pp (V!i)) m2 m2'))
\]

**by** (simp add: Strong-dlHPP-Bisimulation-def, fastforce)
— auxiliary lemma to obtain 'NDC or IDC' from strong (d,lH,PP)-Bisimulation as
lemma in meta logic allowing instantiation of the variables

**lemma** strongdlHPPB-NDCIDCaux:
\[ \forall V V' \ i. \ \text{[Strong-dlHPP-Bisimulation d PP R;}
\]
\[ (V, V') \in R; \ i < \text{length } V \]
\[ \implies (\text{NDC d (V!i)} \lor \text{IDC d (V!i)} (\text{hchLoc (pp (V!i)))}) \]
\[ \text{by (simp add: Strong-dlHPP-Bisimulation-def, auto)} \]

**lemma** WHATWHERE-empty:
WHATWHERE-Secure []
\[ \text{by (simp add: WHATWHERE-Secure-def, auto),}
\]
\[ \text{rule-tac x} = \{([],[])} \text{ in ext},
\]
\[ \text{simp add: Strong-dlHPP-Bisimulation-def sym-def trans-def)} \]

---

**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** Arefl :: \( \langle \text{a} \times \text{a} \rangle \text{ set} \Rightarrow \text{a set} \)
where
Arefl R = \{ e. \ (e,e) \in R \}

**lemma** commonArefl-subset-commonDomain:
Arefl R1 \( \cap \) Arefl R2 \( \subseteq \) (Domain R1 \( \cap \) Domain R2)
\[ \text{by (simp add: Arefl-def, auto)} \]

— define disjoint strong (d,lH,PP)-bisimulation up-to-R’ for a relation R
**definition** disj-dlHPP-Bisimulation-Up-To-R’ ::
\( \langle d \Rightarrow \text{nat set} \Rightarrow \text{com Bisimulation-type} \)
\[ \Rightarrow \langle \text{com Bisimulation-type} \Rightarrow \text{bool} \)
where
disj-dlHPP-Bisimulation-Up-To-R' d PP R' R Í
SdlHPPB d PP R' ∧ (sym R) ∧ (trans R)
∧ (∀(V, V') ∈ R. length V = length V') ∧
(∀(V, V') ∈ R. ∀i < length V.

{(NDC d (V!i)) ∨
 (IDC d (V!i) (htchLoc (pp (V!i))))) ∧
(∀(V, V') ∈ R. ∀i < length V. ∀m1 m1' m2 α p.

{(V!i,m1) →<α> (p,m2) ∧ m1 ~d,(htchLocSet PP) m1'}
→ (∃p' α' m2'. ⟨V!i,m1⟩ →<α'> ⟨p',m2⟩' ∧
(stepResultsinR p p' (R ∪ R')) ∧ (α,α') ∈ (R ∪ R') ∧
(dequality-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: ∀(V, V') ∈ R. length V = length V'
assumes eqlenR': ∀(V, V') ∈ R'. length V = length V'
assumes Areflassump: (Arefl R ∩ Arefl R') ⊆ {[]}
shows trans (R ∪ R')
proof –

{ fix V V' V''
 assume p1: (V, V') ∈ (R ∪ R')
 assume p2: (V', V'') ∈ (R ∪ R')

 from p1 p2 have (V,V'') ∈ (R ∪ R')
 proof (auto)

 assume inR1: (V,V') ∈ R
 assume inR2: (V',V'') ∈ R
 from inR1 inR2 transR show (V,V'') ∈ R
 unfolding trans-def
 by blast

 next

 assume inR1': (V,V') ∈ R'
 assume inR2': (V',V'') ∈ R'
 assume notinR': (V,V'') /∈ R'
 from inR1' inR2' transR' have inR': (V,V'') ∈ R'
 unfolding trans-def
 by blast

 from notinR' inR' have False
 by auto
 thus (V,V'') ∈ R ..

 next

 assume inR1: (V,V') ∈ R
 assume inR2': (V',V'') ∈ R'

from inR1 symR transR have \((V, V) \in R \land (V', V') \in R\)
  unfolding sym-def trans-def
  by blast
hence AreflR: \(\{V, V'\} \subseteq Arefl R\) by (simp add: Arefl-def)
from inR'2 symR' transR' have \((V', V') \in R' \land (V'', V'') \in R'\)
  unfolding sym-def trans-def
  by blast
hence AreflR': \(\{V', V''\} \subseteq Arefl R'\) by (simp add: Arefl-def)

from AreflR AreflR' Areflassump have V'empt: \(V' = []\)
  by (simp add: Arefl-def, blast)
with inR'2 eqlenR' have \(V' = V''\) by auto
with inR1 show \((V, V'') \in R\) by auto
next
assume inR'1: \((V, V') \in R'\)
assume inR2: \((V', V'') \in R\)
from inR'1 symR' transR' have \((V, V) \in R' \land (V', V') \in R'\)
  unfolding sym-def trans-def
  by blast
hence AreflR': \(\{V, V'\} \subseteq Arefl R'\) by (simp add: Arefl-def)
from inR2 symR transR have \((V', V') \in R \land (V'', V'') \in R\)
  unfolding sym-def trans-def
  by blast
hence AreflR: \(\{V', V''\} \subseteq Arefl R\) by (simp add: Arefl-def)

from AreflR AreflR' Areflassump have V'empt: \(V' = []\)
  by (simp add: Arefl-def, blast)
with inR'1 eqlenR' have \(V' = V\) by auto
with inR2 show \((V, V'') \in R\) by auto
qed

thus thesis unfolding trans-def by force

qed

lemma Up-To-Technique:

\[
\begin{align*}
& \ [[ \text{disj-dlHPP-Bisimulation-Up-To-R'} \ d PP'R' R; \\
& \ Arefl R \cap Arefl R' \subseteq \{[]\} ]] \\
& \implies SdlPPB d PP (R \cup R')
\end{align*}
\]
proof (simp add: disj-dlHPP-Bisimulation-Up-To-R'-def
  Strong-dlHPP-Bisimulation-def, auto)
assume symR': sym R'
assume symR: sym R
with symR' show sym \((R \cup R')\)
  by (simp add: sym-def)
next
assume symR': sym R'
assume symR: sym R
assume transR': trans R'}
assume \( \text{trans} R \cdot \text{trans} R \)
assume \( \text{eglen} R \cdot \forall (V, V') \in R', \text{length } V = \text{length } V' \)
assume \( \text{eglen} R \cdot \forall (V, V') \in R, \text{length } V = \text{length } V' \)
assume \( \text{arefl} R \cdot \text{arefl} R' \subseteq \{()\} \)
from \( \text{sym} R' \cdot \text{sym} R \cdot \text{trans} R' \cdot \text{trans} R \cdot \text{eglen} R' \cdot \text{eglen} R \cdot \text{arefl} R \cdot \text{arefl} R' \)
show \( \text{trans} (R \cup R') \)
  by blast
— condition about IDC and NDC and equal length already proven above by auto tactic!

next
fix \( V V' i m1 m1' m2 p \)
assume \( \text{in} R' \cdot (V, V') \in R' \)
assume \( \text{irange} \cdot i < \text{length } V \)
assume \( \text{step} \cdot \langle V!i, m1 \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m2 \rangle \)
assume \( \text{dhequal} \cdot m1 \sim d \cdot \langle \text{htchLocSet } PP \rangle m1' \)
assume \( \text{disjBisimUpTo} \cdot \forall (V, V') \in R'. \forall i < \text{length } V. \forall m1 m1' m2 p.
  \langle V!i, m1 \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m2 \rangle \wedge m1 \sim d \cdot \langle \text{htchLocSet } PP \rangle m1' \rightarrow
  \exists p' \alpha' m2'. \langle V!i, m1' \rangle \rightarrow \triangleleft \alpha' \triangleright \langle p', m2' \rangle \wedge
  \text{stepResultsinR } p p' R' \wedge (\alpha, \alpha') \in R' \wedge
  \text{dhequality-alternative } d \cdot \langle \text{htchLocSet } PP \rangle (pp (V!i)) m2 m2' \)
from \( \text{in} R' \cdot \text{irange} \cdot \text{step} \cdot \text{dhequal} \cdot \text{disjBisimUpTo} \) show \( \exists p' \alpha' m2'.
  \langle V!i, m1' \rangle \rightarrow \triangleleft \alpha' \triangleright \langle p', m2' \rangle \wedge \text{stepResultsinR } p p' (R \cup R') \wedge
  ((\alpha, \alpha') \in R \vee (\alpha, \alpha') \in R') \wedge
  \text{dhequality-alternative } d \cdot \langle \text{htchLocSet } PP \rangle (pp (V!i)) m2 m2' \)
  by (simp add: \( \text{stepResultsinR-def} \), fastforce)
qed

lemma \text{Union-Strong-dlHPP-Bisim}:
\[
\begin{align*}
\text{SdlHPPB } d \cdot PP \cdot R & ; \\
\text{SdlHPPB } d \cdot PP \cdot R' ; \\
\text{arefl } R & \cap \text{arefl } R' \subseteq \{()\} \\
\implies \text{SdlHPPB } d \cdot PP \cdot (R \cup R') & ;
\end{align*}
\]
by (rule \text{Up-To-Technique}, simp-all add:
\text{disj-dlHPP-Bisimulation-Up-To-R'-def} \text{Strong-dlHPP-Bisimulation-def} \text{stepResultsinR-def} \text{, fastforce})

lemma \text{adding-emptypair-keeps-SdlHPPB}:
assumes \( \text{SdlHPP} \cdot \text{SdlHPPB } d \cdot PP \cdot R \)
shows \( \text{SdlHPPB } d \cdot PP \cdot (R \cup \{(\cdot, \cdot)\}) \)
proof
  have \( \text{SdlHPPemp} \cdot \text{SdlHPPB } d \cdot PP \cdot \{(\cdot, \cdot)\} \)
    by (simp add: \text{Strong-dlHPP-Bisimulation-def} \text{sym-def} \text{trans-def})
  have \( \text{commonDom} \cdot \text{Domain } R \cap \text{Domain } \{(\cdot, \cdot)\} \subseteq \{()\} \)
    by auto
  with \( \text{commonArefl-subset-commonDomain} \) have \( \text{Arefl} \cdot \text{Arefl } \{(\cdot, \cdot)\} \subseteq \{()\} \)
qed
by force

with SdlHPP SdlHPP Emp Union-Strong-dlHPP-Bisim show
SdlHPP d PP (R ∪ \{[[],[]]\})
by force
qed
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) ∈ R'
assumes Rdef: R = \{(V',V''), (V',V'') ∈ R' ∧ set (PPV V') ⊆ set (PPV V) ∧ set (PPV V'') ⊆ set (PPV V)\}
shows SdlHPPB d PP R
proof (simp add: Strong-dlHPP-Bisimulation-def, auto)
from SdlHPPB have sym R'
  by (simp add: Strong-dlHPP-Bisimulation-def)
with Rdef show sym R
  by (simp add: sym-def)
next
from SdlHPPB have trans R'
  by (simp add: Strong-dlHPP-Bisimulation-def)
with Rdef show trans R
  by (simp add: trans-def, auto)
next
fix V' V''
assume inR-part: (V',V'') ∈ R

11
with SdlHPPB Rdef show length $V' = length V''$
   by (simp add: Strong-dlHPP-Bisimulation-def, auto)

next
fix $V' V'' i$
assume inR-part: $(V', V'') \in R$
assume irange: $i < length V'$
assume notIDC:

\[ \neg IDC d \ (V''!i) \ (htchLoc (pp (V''!i))) \]

with SdlHPPB inR-part irange Rdef
show NDC d $(V''!i)$
   by (simp add: Strong-dlHPP-Bisimulation-def, auto)

next
fix $V' V'' i \alpha p m1 m1' m2$
assume inR-part: $(V', V'') \in R$
assume irange: $i < length V'$
assume step: \( \langle V''!i, m1 \rangle \rightarrow_{\alpha} \langle p, m2 \rangle \)
assume dhequal: $m1 \sim_d (htchLocSet PP) m1'$

from inR-part SdlHPPB Rdef have eqlen: $length V' = length V''$
   by (simp add: Strong-dlHPP-Bisimulation-def, auto)

from inR-part Rdef
have set $(PPV V') \subseteq set (PPV V) \land set (PPV V'') \subseteq set (PPV V)$
   by auto

with irange PPc-in-PPV-version eqlen
have PPc-Vs-at-i:
set $(PPc (V''!i)) \subseteq set (PPV V) \land set (PPc (V''!i)) \subseteq set (PPV V)$
   by (metis subset-trans)

from SdlHPPB inR-part Rdef irange step dhequal
strongdlHPPB-aux[of d PP R' i]
$V' V'' i \alpha p m1 m1' m2$

obtain $p' \alpha' m2'$ where stepreq: $(V''!i, m1') \rightarrow_{\alpha'} \langle p', m2' \rangle \land$
stepResultsinR p $p' R' \land (\alpha, \alpha') \in R' \land$
dhequality-alternative d PP $(pp (V''!i)) m2 m2'$
   by auto

have Rpp': stepResultsinR p $p' R$
proof -
{
  fix $c c'$
  assume step1: $(V''!i, m1) \rightarrow_{\alpha} \langle Some c, m2 \rangle$
  assume step2: $(V''!i, m1') \rightarrow_{\alpha'} \langle Some c', m2' \rangle$
  assume inR'-res: $([c], [c']) \in R'$

  from PPc-Vs-at-i step1 step2 PPsc-of-step
  have set $(PPc c) \subseteq set (PPV V) \land set (PPc c') \subseteq set (PPV V)$
     by (metis (no-types) option.sel zt1(6))
with inR'-res Rdef have \((c],[c']) \in R\)
  by auto
}
thus thesis
  by (metis step stepResultsinR-def stepreq)
qed

have Rαα': \((α,α') \in R\)
proof
  from PPc-Vs-at-i step stepreq PPs
    have \(\text{set}(PPV\ \alpha) \subseteq \text{set}(PPV\ V) \land \text{set}(PPV\ \alpha') \subseteq \text{set}(PPV\ V)\)
    by (metis (no-types) xt1\((6)\))
  with stepreq Rdef
    show thesis
      by auto
qed

from stepreq Rpp' Rαα show
  \(∃p'\ \alpha^\prime \ m2'. \langle V''!i,m1' \rangle \rightarrow\langle α',m2' \rangle \land\)
  stepResultsinR p p' R \land \(α,α') \in R\land\)
  dhequality-alternative d PP \(pp\ (V^!\ i))\ m2 m2'\)
  by auto
qed

theorem parallel-composition:
  \[ ∀i < \text{length} V. \ \text{WHATWHERE-Secure} [V^!i]; \ \text{unique-PPV} V \]
  \(⇒ \ \text{WHATWHERE-Secure} V\)
proof (simp add: WHATWHERE-Secure-def, induct V, auto)
  fix d PP
  from WHATWHERE-empty
    show \(∃R. \ \text{SdlHPPB} \ d PP \ \land ([],[]) \in R\)
      by (simp add: WHATWHERE-Secure-def)
next
  fix c V d PP
  assume IH: \(\forall i < \text{length} V. \\)
  \(\forall d PP, \exists R. \ \text{SdlHPPB} \ d PP \ \land ([V^!i],[V^!i]) \in R;\)
  unique-PPV V \]
    \(⇒ \ \forall d PP, \exists R. \ \text{SdlHPPB} \ d PP \ \land (V,V) \in R\)
  assume ISassump: \(∀i < \text{Suc} (\text{length} V).\)
    \(∀ d PP, \exists R. \ \text{SdlHPPB} \ d PP \ \land ((c\# V)^!i],[c\# V]^!i]) \in R\)
  assume uniPPcV: unique-PPV (c#V)

  hence IHassump1: unique-PPV V
    by (simp add: unique-PPV-def)

  from uniPPcV have nocommonPP: set (Ppc c) \cap set (PPV V) = \{
    by (simp add: unique-PPV-def)

  from ISassump have IHassump2: \(∀ i < \text{length} V.\)
\[\forall d \text{ PP}, \exists R. \text{SdlHPPB} d \text{ PP} R \land ([V\!i],[V\!i]) \in R\]

by auto

with IHassump1 IH obtain RV' where RV'assump:
\[\text{SdlHPPB} d \text{ PP} RV' \land (V,V) \in RV'\]
by blast

def RV \equiv \{(V',V''), (V',V'') \in RV' \land \text{set (PPV V')} \subseteq \text{set (PPV V)} \land \text{set (PPV V'')} \subseteq \text{set (PPV V)}\}

with RV'assump RV-def SdlHPPB-restricted-on-PP-is-SdlHPPB
have SdlHPPRV: SdlHPPB d PP RV
by force

from ISassump obtain Re' where Re'assump:
\[\text{SdlHPPB} d \text{ PP} Re' \land ([c],[c]) \in Re'\]
by (metis append-Nil drop-Nil neq0-conv not-Cons-self nth-append-length Cons-nth-drop-Suc zero-less-Suc)

def Re \equiv \{(V',V'), (V',V'') \in Re' \land \text{set (PPV V')} \subseteq \text{set (PPc c)} \land \text{set (PPV V'')} \subseteq \text{set (PPc c)}\}

with Re'assump Re-def SdlHPPB-restricted-on-PP-is-SdlHPPB
have SdlHPPRe: SdlHPPB d PP Re
by force

from nocommonPP have Domain RV \cap Domain Re \subseteq {[]} 
by (simp add: RV-def Re-def, auto, 
  metis Inf-mono Inf-commute Inf-idem le-bot nocommonPP unique-V-uneq)

with commonArefl-subset-commonDomain
have Areflassump1: Arefl RV \cap Arefl Re \subseteq {[]} 
by force

def R \equiv \{(V',V''), \exists c c' W W'. V' = c#W \land V'' = c'#W' \land W \neq [] \land W' \neq [] \land ([c],[c']) \in Re \land (W,W') \in RV\}

with RV-def RV'assump Re-def Re'assump inR:
\[V \neq [] \Rightarrow (c\#V,c\#V) \in R\]
by auto

from R-def Re-def RV-def nocommonPP
have Domain R \cap Domain (Re \cup RV) = {}
by (simp add: R-def Re-def RV-def, auto,
  metis Inf-bot-right Inf-inf-iff subset-empty unique-V-uneq,
  metis (hide-lams, no-types) Inf-absorb1 Inf-bot-right Inf-inf-iff unique-c-uneq)

with commonArefl-subset-commonDomain
have Areflassump2: Arefl R \cap Arefl (Re \cup RV) \subseteq {[]}

14
by force

have disjuptoR:
\[ \text{disj-dlHPP-Bisimulation-Up-To-R' d PP (Rc \cup RV) R} \]
proof (simp add: disj-dlHPP-Bisimulation-Up-To-R'-def, auto)
  from Areflassump1 SdlHPPRc SdlHPPRV Union-Strong-dlHPP-Bisim
  show SdlHPPB d PP (Rc \cup RV)
  by force
next
  from SdlHPPRV have symRV: sym RV
  by (simp add: Strong-dlHPP-Bisimulation-def)
  from SdlHPPRc have symRc: sym Rc
  by (simp add: Strong-dlHPP-Bisimulation-def)
  with symRV Rc-def show sym R
  by (simp add: sym-def, auto)
next
  from SdlHPPRV have transRV: trans RV
  by (simp add: Strong-dlHPP-Bisimulation-def)
  from SdlHPPRc have transRc: trans Rc
  by (simp add: Strong-dlHPP-Bisimulation-def)
  show trans R
  proof –
  { fix \( V, V', V'' \)
  assume p1: \((V,V') \in R\)
  assume p2: \((V',V'') \in R\)
  have \((V,V'') \in R\)
  proof –
  from p1 R-def obtain \( c \ c' W W' \) where p1assump:
  \( V = c \# W \land V' = c' \# W' \land W \neq \[] \land W' \neq \[] \land \([c],[c'] \) \in Rc \land (W,W') \in RV \)
  by auto
  with p2 R-def obtain \( c'' W'' \) where p2assump:
  \( V'' = c'' \# W'' \land W'' \neq \[] \land \([c],[c''] \) \in Rc \land (W',W'') \in RV \)
  by auto
  with p1assump transRc transRV have
  trans-assump: \([c],[c''] \) \in Rc \land (W,W'') \in RV
  by (simp add: trans-def, blast)
  with p1assump p2assump R-def show \( ?\)thesis
  by auto
  qed
  } thus \( ?\)thesis unfolding trans-def by blast
  qed
next
  fix \( V, V' \)
  assume \((V,V') \in R\)
  with R-def SdlHPPRV show length V = length V'
by (simp add: Strong-dlHPP-Bisimulation-def, auto)

next
fix \( V \) \( V' \) \( i \)
assume \( \text{inR}: (V,V') \in R \)
assume \( \text{irange}: i < \text{length } V \)
assume \( \text{notIDC}: \sim \text{ IDC } d \left( V!i \right) \)
\((\text{htchLoc} \ (pp \ (V!i)))\)
from \( \text{inR} \) \( \text{R-def} \) obtain \( c \) \( c' \) \( W \) \( W' \) where \( VV'\text{assump}: \)
\( V = c \# W \land V' = c' \# W' \land W \neq [] \land W' \neq [] \land \)
\( ([c],[c']) \in Rc \land (W,W') \in RV \)
by auto
— Case separation for \( i \)
from \( VV'\text{assump SdlHPPRc} \) have Case-i0:
\( i = 0 \implies (\text{NDC } d \ (V!i)) \lor \)
\( \text{IDC } d \ (V!i) \ (\text{htchLoc} \ (pp \ (V!i))) \)
by (simp add: Strong-dlHPP-Bisimulation-def, auto)

from \( VV'\text{assump SdlHPPRv} \) have \( \forall i < \text{length } W. \)
\( (\text{NDC } d \ (W!i)) \lor \)
\( \text{IDC } d \ (W!i) \ (\text{htchLoc} \ (pp \ (W!i))) \)
by (simp add: Strong-dlHPP-Bisimulation-def, auto)

with \( \text{irange} \) \( VV'\text{assump have Case-in0}: \)
\( i > 0 \implies (\text{NDC } d \ (V!i)) \lor \)
\( \text{IDC } d \ (V!i) \ (\text{htchLoc} \ (pp \ (V!i))) \)
by simp
from \( \text{notIDC Case-i0 Case-in0} \)
show \( \text{NDC } d \ (V!i) \)
by auto

next
fix \( V \) \( V' \) \( m1 \) \( m1' \) \( m2 \) \( \alpha \) \( p \) \( i \)
assume \( \text{inR}: (V,V') \in R \)
assume \( \text{irange}: i < \text{length } V \)
assume \( \text{step}: \langle V!i,m1 \rangle \xrightarrow{<\alpha>} \langle p,m2 \rangle \)
assume \( \text{dhequal}: m1 \sim_d (\text{htchLocSet PP}) m1' \)
from \( \text{inR} \) \( \text{R-def} \) obtain \( c \) \( c' \) \( W \) \( W' \) where \( VV'\text{assump}: \)
\( V = c \# W \land V' = c' \# W' \land W \neq [] \land W' \neq [] \land \)
\( ([c],[c']) \in Rc \land (W,W') \in RV \)
by auto
— Case separation for \( i \)
from \( VV'\text{assump SdlHPPRc} \) \( \text{strongdlHPPB-aux}[of \ d \ PP} \)
\( Rc \ 0 [c] [c'] \) \( \text{step dhequal} \)
have Case-i0:
\( i = 0 \implies \exists p' \alpha' m2', \)
\( (V!i,m1') \xrightarrow{<\alpha'}} \langle p',m2' \rangle \land \)
\( \text{stepRes} \ (\text{inR}) \ p \ p' \ (R \cup (Rc \cup RV)) \land \)
\( ([\alpha,\alpha'] \in R \lor (\alpha,\alpha') \in Rc \lor (\alpha,\alpha') \in RV) \land \)
\( \text{dhequality-alternative} \ d \ PP \ (pp \ (V!i)) m2 m2' \)
by (simp add: stepResultsinR-def, blast)

from step VV' assume irange have rewV:
  \( i > 0 \Rightarrow (i - \text{Suc} 0) < \text{length} \ W \land \forall i. W!(i - \text{Suc} 0) \)
  by simp

with irange VV' assume step dhequal SdlHPPR V
  strongdHPPB-aux[of d PP RV - W W]

have Case-in0:
  \( i > 0 \Rightarrow \exists p' \alpha' m2'. \)
  \( (V!i, m1') \rightarrow<\alpha'> (p', m2') \land \)
  stepResultsinR p p' (R ∪ (Rc ∪ RV)) ∧
  \( ((\alpha, \alpha') \in R \lor (\alpha, \alpha') \in Rc \lor (\alpha, \alpha') \in RV) \land \)
  dhequality-alternative d PP (pp (V!i)) m2 m2'
  by (simp add: stepResultsinR-def, blast)

from Case-i0 Case-in0

show \( \exists p' \alpha' m2'. \)
  \( (V!i, m1') \rightarrow<\alpha'> (p', m2') \land \)
  stepResultsinR p p' (R ∪ (Rc ∪ RV)) ∧
  \( ((\alpha, \alpha') \in R \lor (\alpha, \alpha') \in Rc \lor (\alpha, \alpha') \in RV) \land \)
  dhequality-alternative d PP (pp (V!i)) m2 m2'
  by auto

qed

with Arrflassump2 Rc'assume Up-To-Technique

show \( \exists R. \ SdlHPPB d PP R \land (c\#V, c\#V) \in R \)
  by (metis UnCI inR)

qed

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

— function for obtaining the program point of some MWLsloc command

primrec pp :: ('exp', 'id') MWLsCom ⇒ nat
where
pp (skipι) = ι |
pp (x :=ι e) = ι |
pp (c1;c2) = pp c1 |
pp (ifι b then c1 else c2 fi) = ι |
pp (whileι b do c od) = ι |
pp (spawnι 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 (skipι) = [ι] |
PPc (x :=ι e) = [ι] |
PPc (c1;c2) = (PPc c1) @ (PPc c2) |
PPc (ifι b then c1 else c2 fi) = [ι] @ (PPc c1) @ (PPc c2) |
PPc (whileι b do c od) = [ι] @ (PPc c) |
PPc (spawnι 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 ≠ []
by (induct c) auto

lemma unique-c-uneq: set (PPc c) ∩ set (PPc c') = {} ⇒ c ≠ c'
by (insert PPc-nonempt, force)

lemma V-nonempt-PPV-nonempt: V ≠ [] ⇒ PPV V ≠ []
by (auto, induct V, simp-all, insert PPc-nonempt, force)

lemma unique-V-uneq:
[[V ≠ []]; [V' ≠ []]; set (PPV V) ∩ set (PPV V') = {}] ⇒ V ≠ V'
by (auto, induct V, simp-all, insert V-nonempt-PPV-nonempt, auto)

lemma PPc-in-PPV: c ∈ set V ⇒ set (PPc c) ⊆ set (PPV V)
by (induct V, auto)

lemma listindices-aux: i < length V ⇒ (V!i) ∈ set V
by (metis nth-mem)

lemma PPc-in-PPV-version:
i < length V ⇒ set (PPc (V!i)) ⊆ set (PPV V)
by (rule PPc-in-PPV, erule listindices-aux)

lemma uniPPV-uniPPc: unique-PPV V ⇒ (∀ i < length V. unique-PPc (V!i))
by (auto, simp add: unique-PPV-def, induct V,
  auto simp add: unique-PPc-def,
  metis in-set-conv-nth length-Suc-conv set-ConsD)

— SEMANTICS

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-ndet ::
(′exp, ′id, ′val, (′exp, ′id) MWLsCom) TLSteps-curry
((1 {−,−}) →⟨⟩/ (1 {−,−}) [0,0,0,0] 81)

where
⟨e1,m1⟩ →⟨α⟩ ⟨e2,m2⟩ ≡ ((e1,m1),α,(e2,m2)) ∈ MWLsSteps-det |
skip: ⟨skip,m⟩ →⟨⟩ ⟨None,m⟩ |
assign: (E e m) = v ↔
⟨x := e,m⟩ →⟨⟩ ⟨None,m(x := v)⟩ |
seq1: ⟨e1,m⟩ →⟨α⟩ ⟨None,m⟩ →
⟨c1;c2,m⟩ →⟨α⟩ ⟨Some c2,m⟩ |
seq2: ⟨e1,m⟩ →⟨α⟩ ⟨Some c1,m⟩ →
⟨c1;c2,m⟩ →⟨α⟩ ⟨Some (c1′;c2),m⟩ |
iftrue: BMap (E b m) = True ↔
⟨if b then c1 else c2,f,m⟩ →⟨⟩ ⟨Some c1,m⟩ |
iffalse: BMap (E b m) = False ↔
⟨if b then c1 else c2,f,m⟩ →⟨⟩ ⟨Some c2,m⟩ |
whiletrue: BMap (E b m) = True ↔
⟨while b do c od,m⟩ →⟨⟩ ⟨Some (c;while b do c od),m⟩ |
whilefalse: BMap (E b m) = False ↔
⟨while b do c od,m⟩ →⟨⟩ ⟨None,m⟩ |
spawn: ⟨spawn e V,m⟩ →⟨⟩ ⟨None,m⟩ |

inductive-cases MWLsSteps-det-cases:
⟨skip,m⟩ →⟨α⟩ ⟨p,m⟩
⟨x := e,m⟩ →⟨α⟩ ⟨p,m⟩
⟨e1;c2,m⟩ →⟨α⟩ ⟨p,m⟩
⟨if b then c1 else c2,f,m⟩ →⟨α⟩ ⟨p,m⟩
⟨while b do c od,m⟩ →⟨⟩ ⟨p,m⟩
⟨spawn e V,m⟩ →⟨⟩ ⟨p,m⟩

— 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-curry
((1 {−,−}) ⇒/ (1 {−,−}) [0,0,0,0] 81)

where
⟨V,m⟩ ⇒ ⟨V’,m’⟩ ≡ ((V,m),(V’,m’)) ∈ MWLsSteps-ndet |
stepreadi1: ⟨ci,m⟩ →⟨α⟩ ⟨None,m⟩ ⇒
⟨cf @ [ci] @ ca,m⟩ ⇒ ⟨cf @ α @ ca,m’⟩ |
stepreadi2: ⟨ci,m⟩ →⟨α⟩ ⟨Some c’,m’⟩ ⇒
⟨cf @ [ci] @ ca,m⟩ ⇒ ⟨cf @ [c’] @ α @ ca,m⟩

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

lemma nextmem-exists-and-unique:
∃m′ p α. ⟨c, m⟩ →◁ α ⊲ ⟨p, m′⟩
∧ (∀m″. ∃p α. ⟨c, m⟩ →◁ α ⊲ ⟨p, m″⟩) → m″ = m′
by (induct c, auto, metis MWLsSteps-det.skip MWLsSteps-det-cases(1),
metis MWLsSteps-det-cases(2) MWLsSteps-det.assign,
metis (no-types) MWLsSteps-det-seq1 MWLsSteps-det-seq2
MWLsSteps-det-cases(3) not-­Some-eq,
metis MWLsSteps-det-iffalse MWLsSteps-det-iftrue
MWLsSteps-det-cases(4),
metis MWLsSteps-det-whilefalse MWLsSteps-det-whiletrue
MWLsSteps-det-cases(5),
metis MWLsSteps-det-spawn MWLsSteps-det-cases(6))

lemma PPsc-of-step:
[[⟨c, m⟩ →◁ α ⊲ ⟨p, m′⟩; ∃c′. p = Some c′]]
⇒ set (PPc (the p)) ⊆ set (PPc c)
by (induct rule: MWLsSteps-det.induct, auto)

lemma PPso-of-step:
⟨c, m⟩ →◁ α ⊲ ⟨p, m′⟩
⇒ set (PPV α) ⊆ set (PPc c)
by (induct rule: MWLsSteps-det.induct, auto)

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'
([‐]′(‐))
where
[‐](m) ≡ NextMem ′ c m
— define when two expressions are indistinguishable with respect to a domain d
definition d-indistinguishable :: 'd::order ⇒ 'exp ⇒ 'exp ⇒ bool
where
d-indistinguishable $d \ e_1 \ e_2 \equiv \forall \ m \ m'. ((m =_d m')) 
\rightarrow ((E \ e_1 \ m) = (E \ e_2 \ m'))$

abbreviation d-indistinguishable' :: 'exp \Rightarrow 'd::order \Rightarrow 'exp \Rightarrow bool
( (\equiv_-) )

where
e_1 \equiv_d e_2 \equiv d\text{-indistinguishable} \ d \ e_1 \ e_2

— symmetry of d\text{-indistinguishable}
lemma d-indistinguishable-sym:
e \equiv_d e' \Rightarrow e' \equiv_d e
by (simp add: d\text{-indistinguishable-def d-equal-def,metis})

— transitivity of d\text{-indistinguishable}
lemma d-indistinguishable-trans:
\[( e \equiv_d e'; e' \equiv_d e'' \] \Rightarrow e \equiv_d e''\nby (simp add: d\text{-indistinguishable-def d-equal-def,metis})

— predicate for dH\text{-indistinguishable}
definition dH\text{-indistinguishable} :: 'd \Rightarrow ('d, 'exp) Hatches
\Rightarrow 'exp \Rightarrow 'exp \Rightarrow bool

where
dH\text{-indistinguishable} \ d \ H \ e_1 \ e_2 \equiv (\forall \ m \ m'. m \sim_{d,H} m') 
\rightarrow ((E \ e_1 \ m) = (E \ e_2 \ m'))

abbreviation dH\text{-indistinguishable'} :: 'exp \Rightarrow 'd
\Rightarrow ('d, 'exp) Hatches \Rightarrow 'exp \Rightarrow bool
( (\equiv_-,-) )

where
e_1 \equiv_{d,H} e_2 \equiv dH\text{-indistinguishable} \ d \ H \ e_1 \ e_2

lemma empH-implies-dHindistinguishable-eq-dindistinguishable:
\((e \equiv_{d,\{\}} e') = (e \equiv_d e')\)
by (simp add: d\text{-indistinguishable-def dH\text{-indistinguishable-def}
dH\text{-equal-def d\text{-equal-def}})

theorem WHATWHERE-Secure-Skip:
WHATWHERE-Secure [skip]
proof (simp add: WHATWHERE-Secure-def, auto)
fix \ d \ PP
def R \equiv {{(V::('exp,'id) MWLsCom list,V'::('exp,'id) MWLsCom list). \ V = V' \land (V = [] \lor V = [skip])}}

have inR: ([skip],[skip]) \in R
by (simp add: R-def)
have \( SdlHPPB \ d \ PP \ R \)

proof (simp add: Strong-dlHPP-Bisimulation-def R-def sym-def trans-def NDC-def NextMem-def, auto)

fix \( m1 \) \( m1' \)

assume dequal: \( m1 =_d m1' \)

have nextm1: \((\text{THE } m2. (\exists \alpha. \langle \text{skip}, m1 \rangle \rightarrow <\alpha> \ (p, m2))) = m1\)

by (insert MWLsSteps-det.simps[of skip_m1], force)

have nextm1':
\((\text{THE } m2'. (\exists \alpha. \langle \text{skip}, m1' \rangle \rightarrow <\alpha> \ (p, m2')) = m1'\)

by (insert MWLsSteps-det.simps[of skip_m1'], force)

with dequal nextm1 show
\( \text{THE } m2. \exists \alpha. \langle \text{skip}, m1 \rangle \rightarrow <\alpha> \ (p, m2) =_d \)
\( \text{THE } m2'. \exists \alpha. \langle \text{skip}, m1' \rangle \rightarrow <\alpha> \ (p, m2') \)

by auto

next

fix \( p \) \( m1 \) \( m1' \) \( m2 \) \( m2' \)

assume dequal: \( m1 \sim_d (\text{htchLocSet PP}) \ m1' \)

assume skipstep: \( \langle \text{skip}, m1 \rangle \rightarrow <\alpha> \ (p, m2) \)

with MWLsSteps-det.simps[of skip_m1 \( \alpha \) \( p \) \( m2 \)]

have aux: \( p = \text{None} \land m2 = m1 \land \alpha = [] \)

by auto

with dequal skipstep MWLsSteps-det.skip

show \( \exists p' \ m2'. \langle \text{skip}, m1' \rangle \rightarrow <\alpha> \ (p', m2') \land \)
\( \text{stepResultsinR } p \ p' \{ (V, V') \land (V = [] \lor V = [\text{skip}]) \} \land \)
\( (\alpha = [] \lor \alpha = [\text{skip}]) \land \)
\( \text{dhequality-alternative } d \ PP \ i \ m2 m2' \)

by (simp add: stepResultsinR-def dhequality-alternative-def, fastforce)

qed

with inR show \( \exists R. \ SdlHPPB \ d \ PP \ R \land ([\text{skip}], [\text{skip}]) \in R \)

by auto

qed

— auxiliary lemma for assign side condition (lemma 9 in original paper)

lemma semAssignSC-aux:

assumes dhind: \( e \equiv DA x,(\text{htchLoc } i) \ e \)

shows NDC \( d \ (x := e) \lor IDC \ d \ (x := e) \ (\text{htchLoc } (pp \ (x := e))) \)

proof (simp add: IDC-def, auto, simp add: NDC-def)

fix \( m1 \) \( m1' \)

assume dhequal: \( m1 \sim_d (\text{htchLoc } i) \ m1' \)

hence dequal: \( m1 =_d m1' \)

23
by (simp add: dH-equal-def)

obtain v where v eq: E e m1 = v by auto
hence m2eq: [x :=_e e](m1) = m1(x := v)
  by (simp add: NextMem-def, insert MWLsSteps-det.simps[of x :=_e e m1], force)

obtain v' where v' eq: E e m1' = v' by auto
hence m2' eq: [x :=_e e](m1') = m1'(x := v')
  by (simp add: NextMem-def, insert MWLsSteps-det.simps[of x :=_e e m1'], force)

from dhequal have shiftdomain:
  DA x ≤ d =⇒ m1 ∼ DA x.(htchLoc i) m1'
  by (simp add: dH-equal-def d-equal-def htchLoc-def)

with veq v' eq dhind
have (DA x) ≤ d =⇒ v = v'
  by (simp add: dH-indistinguishable-def)

with dequal m2eq m2' eq
show [x :=_e e](m1) =_d [x :=_e e](m1')
  by (simp add: d-equal-def)
qed

theorem WHATWHERE-Secure-Assign:
  assumes dhind: e ≡ DA x.(htchLoc i) e
  assumes dheq-imp: ∀ m m' d e'. (m ∼ d,(htchLoc i') m' ∧
  [x :=_e e](m) =_d [x :=_e e](m'))
  =⇒ [x :=_e e](m) =_d,(htchLoc i') [x :=_e e](m')
  shows WHATWHERE-Secure [x :=_e e]
proof (simp add: WHATWHERE-Secure-def, auto)
fix d PP
def R ≡ {(V::('exp,'id) MWLsCom list,V'::('exp,'id) MWLsCom list).
  V = V' ∧ (V = [] ∨ V = [x :=_e e])
}
have inR: ([x :=_e e],[x :=_e e]) ∈ R
  by (simp add: R-def)

have SdlHPPB d PP R
proof (simp add: Strong-dlHPP-Bisimulation-def R-def
  sym-def trans-def, auto)
  assume notIDC: ¬ IDC d (x :=_e e) (htchLoc i)
  with dhind semAssignSC-aux
  show NDC d (x :=_e e)
  qed
by force

next

fix \( m_1, m_1', m_2, p, \alpha \)

assume assignstep: \( \langle x := e, m_1 \rangle \rightarrow ^{\alpha} \langle p, m_2 \rangle \)

assume dhequal: \( m_1 \sim_d, \text{hatchLocSet } PP \rightarrow m_1' \)

from assignstep have nextm1:

\[ p = \text{None} \land \alpha = [] \land [x := e](m_1) = m_2 \]

by (simp add: NextMem-def,

insert MWLsSteps-det.simps[of \( x := e.m_1 \), force])

obtain \( m_2' \) where nextm1':

\[ \langle x := e, m_1' \rangle \rightarrow ^{[]} \langle \text{None}, m_2' \rangle \land [x := e](m_1') = m_2' \]

by (simp add: NextMem-def,

insert MWLsSteps-det.simps[of \( x := e.m_1 \), force])

from dhequal have dhequal-\( \lambda t. \text{hatchLoc } t \subseteq \text{hatchLocSet } PP \)

\[ \Rightarrow m_1 \sim_d, \text{hatchLoc } t \rightarrow m_1' \]

by (simp add: dH-equal-def, auto)

with dhind semAssignSC-aux

have hatchLoc \( t \subseteq \text{hatchLocSet } PP \Rightarrow \)

\[ [x := e](m_1) = [x := e](m_1') \]

by (simp add: NDC-def IDC-def dH-equal-def, fastforce)

with dhind dheq-imp dhequal-\( \lambda t. \text{hatchLoc } t \subseteq \text{hatchLocSet } PP \Rightarrow \)

\[ [x := e](m_1) \sim_d, \text{hatchLocSet } PP \rightarrow [x := e](m_1') \]

by (simp add: hatchLocSet-def dh-equal-def, blast)

with nextm1 nextm1' assignstep dhequal

show \( \exists p', m_2'. \)

\[ \langle x := e, m_1 \rangle \rightarrow ^{\alpha} \langle p', m_2 \rangle \land \]

stepResultsinR \( p' \}{(V, V')} \land (V = [] \lor V = [x := e]) \}

\[ \land (\alpha = [] \lor \alpha = [x := e]) \land d\text{equality-alternative } d \text{ PP } \rightarrow m_2 m_2' \]

by (auto simp add: stepResultsinR-def dhequality-alternative-def)

qed

with inR show \( \exists R. \text{SdlHPPB } d \text{ PP } R \land ([x := e], [x := e]) \in R \)

by auto

qed

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 $\llbracket c1 \rrbracket$
assumes WWs-part2: WHATWHERE-Secure $\llbracket c2 \rrbracket$
assumes uniPPc1c2: unique-PPc ($c1; c2$)
shows WHATWHERE-Secure $\llbracket c1; c2 \rrbracket$
proof (simp add: WHATWHERE-Secure-def, auto)
fix $d$ PP
from uniPPc1c2 have nocommonPP: $\{ \}$
  by (simp add: unique-PPV-def, auto)
from WWs-part1 obtain $R1'$ where $R1'$assump:
  SdlHPPB $d$ PP $R1'$ $\llbracket c1 \rrbracket$
  by (simp add: WHATWHERE-Secure-def, auto)
def $R1 \equiv \{ (V, V'), (V, V') \in R1' \land \text{set } (PPV V) \subseteq \text{set } (PPc c1) \land \text{set } (PPV V') \subseteq \text{set } (PPc c1) \}$
from $R1'$assump $R1$-def SdlHPPB-restricted-on-PP-is-SdlHPPB
have SdlHPPR1: SdlHPPB $d$ PP $R1$
  by force
from WWs-part2 obtain $R2'$ where $R2'$assump:
  SdlHPPB $d$ PP $R2'$ $\llbracket c2 \rrbracket$
  by (simp add: WHATWHERE-Secure-def, auto)
def $R2 \equiv \{ (V, V'), (V, V') \in R2' \land \text{set } (PPV V) \subseteq \text{set } (PPc c2) \land \text{set } (PPV V') \subseteq \text{set } (PPc c2) \}$
from $R2'$assump $R2$-def SdlHPPB-restricted-on-PP-is-SdlHPPB
have SdlHPPR2: SdlHPPB $d$ PP $R2$
  by force
from nocommonPP have nocommonDomain: Domain $R1$ $\cap$ Domain $R2$ $\subseteq \{ \}$
by (simp add: R1-def R2-def, auto,
       metis inf-greatest inf-idem le-bot unique-V-uneq)

with commonArefl-subset-commonDomain
have AreflASSump1: Arefl R1 ∩ Arefl R2 ⊆ {[]}
  by force

def R0 ≡ {((s1,s2). ∃ c1 c1’ c2 c2’ s1 = [c1;c2] ∧ s2 = [c1’;c2’] ∧
  ([c1],[c1’]) ∈ R1 ∧ ([c2],[c2’]) ∈ R2)}

with R1-def R1’assump R2-def R2’assump
have inR0: ([c1;c2],[c1;c2]) ∈ R0
  by auto

have Domain R0 ∩ Domain (R1 ∪ R2) = {}
  by (simp add: R0-def R1-def R2-def, auto Int-absorb1 Int-assoc Int-empty-left
         nocommonPP unique-c-uneq, metis Int-absorb1 Int-assoc Int-empty-left nocommonPP unique-c-uneq)

with commonArefl-subset-commonDomain
have AreflASSump2: Arefl R0 ∩ Arefl (R1 ∪ R2) ⊆ {[]}
  by force

have disjuptoR0:
  disj-dlHPP-Bisimulation-Up-To-R’ d PP (R1 ∪ R2) R0
proof (simp add: disj-dlHPP-Bisimulation-Up-To-R’-def, auto)
  from AreflASSump1 SdIHPPR1 SdIHPPR2 Union-Strong-dlHPP-Bsim
  show SdIHPPB d PP (R1 ∪ R2)
    by metis
next
  from SdIHPPR1 have symR1: sym R1
    by (simp add: Strong-dlHPP-Bisimulation-def)
  from SdIHPPR2 have symR2: sym R2
    by (simp add: Strong-dlHPP-Bisimulation-def)
  with symR1 R0-def show sym R0
    by (simp add: sym-def, auto)
next
  from SdIHPPR1 have transR1: trans R1
    by (simp add: Strong-dlHPP-Bisimulation-def)
  from SdIHPPR2 have transR2: trans R2
    by (simp add: Strong-dlHPP-Bisimulation-def)
  show trans R0
    proof —
      { fix V V’ V’’
        assume p1: (V,V’) ∈ R0
        assume p2: (V’,V’’) ∈ R0
        have (V,V’’) ∈ R0
        show}
proof -
  from p1 R0-def obtain c1 c2 c1’ c2’ where p1assump:
  \[ \begin{align*}
  V &= [c1;c2] \land V’ = [c1’;c2’] \land \\
  ([c1],[c1’]) &\in R1 \land ([c2],[c2’]) \in R2 \\
  \end{align*} \]
  by auto
  with p2 R0-def obtain c1'' c2'' where p2assump:
  \[ \begin{align*}
  V'' &= [c1'';c2'’] \land \\
  ([c1’’],[c1’’]) &\in R1 \land ([c2’’],[c2’’]) \in R2 \\
  \end{align*} \]
  by auto
  with p1assump transR1 transR2 have
  trans-assump: ([c1],[c1’’]) \in R1 \land ([c2],[c2’’]) \in R2
  by (simp add: trans-def, blast)
  with p1assump p2assump R0-def show \textit{thesis}
  by auto
  qed

thus \textit{thesis} unfolding trans-def by blast
qed

next
  fix V V’
  assume (V,V’) \in R0
  with R0-def show length V = length V’
  by auto

next
  fix V V’ i
  assume inR0: (V,V’) \in R0
  assume irange: i < length V
  assume notIDC: \neg IDC d (htchLoc (pp (V!i)))
  from inR0 R0-def obtain c1 c2 c1’ c2’ where VV’assump:
  \[ \begin{align*}
  V &= [c1;c2] \land V’ = [c1’;c2’] \land \\
  ([c1],[c1’]) &\in R1 \land ([c2],[c2’]) \in R2 \\
  \end{align*} \]
  by auto
  have eqnextmem: \\( \forall m. [c1;c2](m) = [c1](m) \)
  proof -
    fix m
    from nextmem-exists-and-unique obtain m’ where c1nextmem:
    \[ \exists p \ a. \langle c1,m \rangle \rightarrow <\alpha> \langle p,m’ \rangle \land
    \langle \forall m''. \exists p \ a. \langle c1,m \rangle \rightarrow <\alpha> \langle p,m'' \rangle \rightarrow m'' = m' \rangle \]
    by force
    hence eqdir1: \[ [c1](m) = m’ \]
    by (simp add: NextMem-def, auto)
    from c1nextmem obtain p a where \( \langle c1,m \rangle \rightarrow <\alpha> \langle p,m’ \rangle \)
    by auto
    with c1nextmem have \( \exists p \ a. \langle c1;c2,m \rangle \rightarrow <\alpha> \langle p,m’ \rangle \land
    \langle \forall m''. \exists p \ a. \langle c1;c2,m \rangle \rightarrow <\alpha> \langle p,m'' \rangle \rightarrow m'' = m' \rangle \)
    by (auto, metis MWLsSteps-det.seq1 MWLsSteps-det.seq2
option.exhaust, metis MWLsSteps-det-cases(3))

hence eqdir2: [c1;c2](m) = m'
  by (simp add: NextMem-def, auto)

with eqdir1 show [c1;c2](m) = [c1](m)
  by auto
qed

have eqpp: pp (c1;c2) = pp c1
  by simp
from V'assump SDLHPPR1 have IDC d c1 (htchLoc (pp c1))
  ∨ NDC d c1
  by (simp add: Strong-dlHPP-Bisimulation-def, auto)
with eqnextmem eqpp have IDC d (c1;c2)
  (htchLoc (pp (c1;c2))) ∨ NDC d (c1;c2)
  by (simp add: IDC-def NDC-def)
with inR0 irange notIDC VV'assump
show NDC d (V!i)
  by (simp add: IDC-def, auto)

next
fix V V' m1 m1' m2 α p i
assume inR0: (V,V') ∈ R0
assume irange: i < length V
assume step: ⟨V!i,m1⟩ →<α> ⟨p,m2⟩
assume dhequal: m1 ∼ d,htchLocSet PP m1'

from inR0 R0-def obtain c1 c1' c2 c2'
where R0pair:
  V = [c1;c2] ∧ V' = [c1';c2'] ∧ ([c1],[c1']) ∈ R1
  ∧ ([c2],[c2']) ∈ R2
  by auto

from R0pair irange have i0: i = 0 by simp

have eqpp: pp (c1;c2) = pp c1
  by simp

— get the two different cases:
from R0pair step i0 obtain c3 where case-distinction:
  (p = Some c2 ∧ ⟨c1,m1⟩ →<α> ⟨None,m2⟩)
  ∨ (p = Some (c3;c2) ∧ ⟨c1,m1⟩ →<α> ⟨Some c3,m2⟩)
  by (simp, insert MWLsSteps-det.simps[of c1;c2 m1],
  auto)
moreover
— Case 1: first command terminates
{ assume passump: p = Some c2
  assume StepAssump: ⟨c1,m1⟩ →<α> ⟨None,m2⟩
  hence Vstep-case1:
\langle c_1; c_2, m_1 \rangle \rightarrow <\alpha> \langle \text{Some } c_2, m_2 \rangle 

by (simp add: MWLsSteps-det.seq1)

from SdlHPPR1 StepAssump R0pair dhequal stronglyHPPB-aux[of d PP]
R1 \ 0 [c_1] [c_1'] m_1 \alpha \text{ None } m_2 m_1'

obtain \ p' \alpha' \ m_2' \ where \ c_1c_1' \reason:
\ p' = \text{None} \land \langle c_1', m_1' \rangle \rightarrow <\alpha'> <\alpha' > <\alpha> \land (\alpha, \alpha') \in R1 \land 
dhequality-alternative d PP (pp c_1) m_2 m_2'
by (simp add: stepResultsinR-def, fastforce)

with eqpp c_1c_1' \reason have conclpart:
\langle c_1', c_2', m_1' \rangle \rightarrow <\alpha'> \langle \text{Some } c_2', m_2' \rangle \land 
dhequality-alternative d PP (pp (c_1;c_2)) m_2 m_2'
by (auto, simp add: MWLsSteps-det.seq1)

with passump R0pair c_1c_1' \reason i0
have case1-concl:
\exists \ p' \alpha' \ m_2'.
\langle V?!i, m_1 \rangle \rightarrow <\alpha> <\alpha'> \langle p', m_2' \rangle \land 
stepResultsinR p p' (R0 \cup (R1 \cup R2)) \land 
(((\alpha, \alpha') \in R0 \lor (\alpha, \alpha') \in R1 \lor (\alpha, \alpha') \in R2) \land 
dhequality-alternative d PP (pp (V?!i)) m_2 m_2'
by (simp add: stepResultsinR-def, auto)

moreover
— Case 2: first command does not terminate
{
assume passump: \ p = \text{Some} (c_3; c_2)
assume StepAssump: \langle c_1, m_1 \rangle \rightarrow <\alpha> \langle \text{Some } c_3, m_2 \rangle

hence Vstep-case2: \langle c_1; c_2, m_1 \rangle \rightarrow <\alpha> \langle \text{Some } (c_3;c_2), m_2 \rangle
by (simp add: MWLsSteps-det.seq2)

from SdlHPPR1 StepAssump R0pair dhequal stronglyHPPB-aux[of d PP]
R1 \ 0 [c_1] [c_1'] m_1 \alpha \text{ Some } c_3 m_2 m_1'

obtain \ p' \alpha' \ m_2' \ where \ c_1c_1' \reason:
\ p' = \text{Some } c_3' \land \langle c_1', m_1' \rangle \rightarrow <\alpha'> <\alpha' > <\alpha> \land (\alpha, \alpha') \in R1 \land 
dhequality-alternative d PP (pp c_1) m_2 m_2'
by (simp add: stepResultsinR-def, fastforce)

with eqpp c_1c_1' \reason have conclpart:
\langle c_1'; c_2', m_1' \rangle \rightarrow <\alpha'> \langle \text{Some } (c_3';c_2'), m_2' \rangle \land 
dhequality-alternative d PP (pp (c_1;c_2)) m_2 m_2'
by (auto, simp add: MWLsSteps-det.seq2)

from c_1c_1' \reason R0pair R0-def have
\((\{c3; c2\}, [c3'; c2'] \}) \in R0

by auto

**with** \(R0\)-pair conclpart passump \(c1c1'\)'reason i0

**have** case1-concl:

\[
\exists p' \alpha' m2'. \langle V'_i, m1' \rangle \rightarrow \langle p', m2' \rangle \land
\text{stepResultsinR} p p' (R0 \cup (R1 \cup R2)) \land
((\alpha, \alpha') \in R0 \lor (\alpha, \alpha') \in R1 \lor (\alpha, \alpha') \in R2) \land
\text{dhequality-alternative d PP} (pp (V!i)) m2 m2'
\]

by (simp add: stepResultsinR-def, auto)

} 

ultimately

**show** \(\exists p' \alpha' m2'. \langle V'_i, m1' \rangle \rightarrow \langle p', m2' \rangle \land
\text{stepResultsinR} p p' (R0 \cup (R1 \cup R2)) \land
((\alpha, \alpha') \in R0 \lor (\alpha, \alpha') \in R1 \lor (\alpha, \alpha') \in R2) \land
\text{dhequality-alternative d PP} (pp (V!i)) m2 m2'
\]

by blast

qed

**with** in\(R0\) Arcflassump2 Up-To-Technique

**have** SdlHPPB d PP (R0 \cup (R1 \cup R2))

by auto

**with** in\(R0\) **show** \(\exists R. \ SdlHPPB d PP R \land (\{c1; c2\}, [c1; c2]) \in R\)

by auto

qed

**theorem** Compositionality-Spawn:

**assumes** WWs-threads: WHATWHERE-Secure V

**assumes** uniPPspawn: unique-PPc (spawn \(\iota\) V)

**shows** WHATWHERE-Secure [spawn \(\iota\) V]

**proof** (simp add: WHATWHERE-Secure-def, auto)

**fix** d PP

from uniPPspawn **have** pp-difference: \(\iota \notin \text{set (PPV V)}\)

by (simp add: unique-PPc-def)

— Step 1

**from** WWs-threads **obtain** R' \where** R'assump:

SdlHPPB d PP R' \land (V, V) \in R'

by (simp add: WHATWHERE-Secure-def, auto)

\(def\ R \equiv \{(V', V''). \ (V', V'') \in R' \land \text{set (PPV V')} \subseteq \text{set (PPV V)} \land \text{set (PPV V'')} \subseteq \text{set (PPV V)}\}\)

**from** R'assump \(R\)-def SdlHPPB-restricted-on-PP-is-SdlHPPB

**have** SdlHPPR: SdlHPPB d PP R

31
by force

— Step 2

**def** $R_0 \equiv \{(sp_1, sp_2) \cdot \exists \iota' \iota'' V' V''.
sp_1 = [spaw_n_i, V'] \land sp_2 = [spaw_n_i, V'']
\land \iota' \notin set (PPV V) \land \iota'' \notin set (PPV V) \land (V', V'') \in R\}

with $R$-def $R'$-assump pp-difference have in$R_0$:

\[
([\text{spawn}_i, V], [\text{spawn}_i, V]) \in R_0
\]
by auto

— Step 3

from $R_0$-def $R$-def $R'$-assump have Domain $R_0 \cap \text{Domain } R = \{}$
by auto

with commonArefl-subset-commonDomain
have AreflAssump: Arefl $R_0 \cap \text{Arefl } R \subseteq \{\}$
by force

— Step 4

have disjupto$R_0$:

**disj-dlHPP-Bisimulation-Up-To-$R'$ d PP $R$ $R_0$**

**proof** (simp add: disj-dlHPP-Bisimulation-Up-To-$R'$-def, auto)

from SdlHPP$R$ show SdlHPP$B$ d PP $R$
by auto

next

from SdlHPP$R$ have sym$R$: sym $R$
by (simp add: Strong-dlHPP-Bisimulation-def)
with $R_0$-def show sym $R_0$
by (simp add: sym-def, auto)

next

from SdlHPP$R$ have trans$R$: trans $R$
by (simp add: Strong-dlHPP-Bisimulation-def)
with $R_0$-def show trans $R_0$

**proof** —

\{ fix $V_1 V_2 V_3$
assume inR1: $(V_1, V_2) \in R_0$
assume inR2: $(V_2, V_3) \in R_0$
from inR1 $R_0$-def obtain $W W' \iota \iota'$ where $p_1$: $V_1 = [spaw_n_i, W]$
\land $V_2 = [spaw_n_i, W'] \land \iota \notin set (PPV V) \land \iota' \notin set (PPV V)$
\land $(W, W') \in R$
by auto
with inR2 $R_0$-def obtain $W'' \iota''$ where $p_2$: $V_3 = [spaw_n_i, W'']$
\land $\iota'' \notin set (PPV V) \land (W', W'') \in R$
by auto
from $p_1$ $p_2$ trans$R$ have $(W, W'') \in R$
by (simp add: trans-def, auto)
with $p_1$ $p_2$ $R_0$-def have $(V_1, V_3) \in R_0$
by auto

} thus \( ?thesis \) unfolding trans-def by blast

qed

next

fix \( V' V'' \)
from SdlHPPR have eqlenR: \( (V', V'') \in R \implies length V' = length V'' \)
  by (simp add: Strong-dlHPP-Bisimulation-def, auto)

with R0-def show \( (V', V'') \in R0 \implies length V' = length V'' \)
  by auto

next

fix \( V' V'' \) i
assume inR0: \( (V', V'') \in R0 \)
assume irange: \( i < length V' \)
from inR0 R0-def obtain \( i' i'' W' W'' \)
  where R0pair: \( V' = [spawn_{i} W'] \land V'' = [spawn_{i''} W''] \)
  by auto

{ fix \( m \)
  from nextmem-exists-and-unique obtain \( m' \) where spawnnextmem:
  \[ \exists p \alpha. \langle spawn_i \ W', m \rangle \rightarrow\langle\alpha\rangle \langle p, m' \rangle \]
  \land (\forall m''. \exists p \alpha. \langle spawn_i \ W', m \rangle \rightarrow\langle\alpha\rangle \langle p, m'' \rangle \implies m'' = m')
  by force

  hence \( m = m' \)
  by (metis MWLsSteps-det.spawn)

  with spawnnextmem have eqnextmem:
  \[ [spawn_i \ W'](m) = m \]
  by (simp add: NextMem-def, auto)
}

with R0pair irange show NDC d (V'!i)
  by (simp add: NDC-def)

next

fix \( V' V'' \) i m1 m1' m2 \( \alpha p \)
assume inR0: \( (V', V'') \in R0 \)
assume irange: \( i < length V' \)
assume step: \( (V'!i, m1) \rightarrow\langle\alpha\rangle \langle p, m2 \rangle \)
assume dhequal: \( m1 \sim_{d, hchLocSet PP} m1' \)
from inR0 R0-def obtain \( i' i'' W' W'' \)
  where R0pair: \( V' = [spawn_{i} \ W'] \land V'' = [spawn_{i''} W''] \land (W', W'') \in R \)
  by auto

with step irange
have conc-step1: \( \alpha = W' \land p = \text{None} \land m2 = m1 \)
  by (simp, metis MWLsSteps-det-cases(6))

33
from R0pair irange
obtain p' α' m2' where conc-step2: (V''!i,m1') → α' m2' ∧ p' = None ∧ α' = W'' ∧ m2' = m1'
by (simp, metis MWLsSteps-def.spawn)

with R0pair conc-step1 dhequal irange
show ∃p' α' m2'. (V''!i,m1') → α' m2' ∧
  stepResultsinR p' (R0 ∪ R) ∧
  ((α,α') ∈ R0 ∨ (α,α') ∈ R) ∧
  dhequality-alternative d PP ((pp (V''!i)) m2 m2')
by (simp add: stepResultsinR-def
dhequality-alternative-def, auto)

qed — Step 5
with ArflAssump Up-To-Technique
have SdlHPPB d PP (R0 ∪ R)
by auto

with inR0 show ∃R. SdlHPPB d PP R ∧
  ([spawn_1 V],[spawn_1 V]) ∈ R
by auto

qed

theorem Compositionality-If:
assumes dind: ∀d. b ≡ d b
assumes WWs-branch1: WHATWHERE-Secure [c1]
assumes WWs-branch2: WHATWHERE-Secure [c2]
assumes uniPPif: unique-PPc (if b then c1 else c2 fi)
shows WHATWHERE-Secure [if b then c1 else c2 fi]
proof (simp add: WHATWHERE-Secure-def, auto)
fix d PP
from uniPPif have nocommonPP: set (PPc c1) ∩ set (PPc c2) = {}
  by (simp add: unique-PPc-def)

from uniPPif have pp-difference: t ∉ set (PPc c1) ∪ set (PPc c2)
  by (simp add: unique-PPc-def)

from WWs-branch1 obtain R1' where R1'assump:
  SdlHPPB d PP R1' ∧ ([c1],[c1]) ∈ R1'
by (simp add: WHATWHERE-Secure-def, auto)

def R1 ≡ {(V,V'). (V,V') ∈ R1' ∧ set (PPV V) ⊆ set (PPc c1)}
  ∧ set (PPV V') ⊆ set (PPc c1)}

from R1'assump R1-def SdlHPPB-restricted-on-PP-is-SdlHPPB
have SdlHPPR1: SdlHPPB d PP R1
by force
from WWs-branch2 obtain R2' where R2'assump:

\[ \text{SdlHPPB d PP R2' } \land ([c2],[c2]) \in R2' \]

by (simp add: WHATWHERE-Secure-def, auto)

def R2 ≡ \{(V,V'). (V,V') \in R2' \land set (PPV V) \subseteq set (PPc c2) \land set (PPV V') \subseteq set (PPc c2)\}

from R2'assump R2-def SdlHPPB-restricted-on-PP-is-SdlHPPB have SdlHPPR2: SdlHPPB d PP R2

by force

from nocommonPP have Domain R1 \cap Domain R2 \subseteq \{[]\}

by (simp add: R1-def R2-def, auto, metis empty-subsetI inf-idem inf-mono set-eq-subset unique-V-uneq)

with commonArefl-subset-commonDomain have Areflassump1: Arefl R1 \cap Arefl R2 \subseteq \{[]\}

by force

with SdlHPPR1 SdlHPPR2 Union-Strong-dlHPP-Bisim have SdlHPPR1R2: SdlHPPB d PP (R1 \cup R2)

by force

def R ≡ (R1 \cup R2) \cup \{([],[])\}

def R0 ≡ \{(i1,i2). \exists i' i'' b' b'' c1' c1'' c2' c2''.
i1 = [if b' then c1' else c2' fi]
i2 = [if b'' then c1'' else c2'' fi]
i' \notin (set (PPc c1) \cup set (PPc c2))
i'' \notin (set (PPc c1) \cup set (PPc c2)) \land ([c1],[c1']) \in R1 \land ([c2],[c2'']) \in R2 \land b' \equiv_d b''\}

with R-def R1-def R2-def R2'assump pp-difference dind have inR0: ([if b then c1 else c2 fi],[if b then c1 else c2 fi]) \in R0

by auto

from R0-def R-def R1-def R2-def have Domain R0 \cap Domain R = \{\}

by auto

with commonArefl-subset-commonDomain have Areflassump2: Arefl R0 \cap Arefl R \subseteq \{\}

by force

have disjuptoR0:

disj-dlHPP-Bisimulation-Up-To-R' d PP R R0

proof (simp add: disj-dlHPP-Bisimulation-Up-To-R'-def, auto)
from SdlHPPR1R2 adding-emptypair-keeps-SdlHPPB
show SdlHPPB d PP R
  by (simp add: R-def)

next
from SdlHPPR1 have symR1: sym R1
  by (simp add: Strong-dlHPP-Bisimulation-def)
from SdlHPPR2 have symR2: sym R2
  by (simp add: Strong-dlHPP-Bisimulation-def)
from symR1 symR2 d-indistinguishable-sym R0-def show sym R0
  by (simp add: sym-def, fastforce)

next
from SdlHPPR1 have transR1: trans R1
  by (simp add: Strong-dlHPP-Bisimulation-def)
from SdlHPPR2 have transR2: trans R2
  by (simp add: Strong-dlHPP-Bisimulation-def)
show trans R0
proof
{  
  fix V' V'' V'''
  assume p1: (V',V'') ∈ R0
  assume p2: (V'',V''') ∈ R0

  from p1 R0-def obtain !' !'' b' b'' c1' c1'' c2' c2'' where
    passump1: V' = [if !_b' then c1' else c2' fi]
    ∧ V'' = [if !_b'' then c1'' else c2'' fi]
    ∧ !_b' /∈ (set (PPc c1) ∪ set (PPc c2))
    ∧ !_b'' /∈ (set (PPc c1) ∪ set (PPc c2))
    ∧ ([c1',c1'']) ∈ R1 ∧ ([c2',c2'']) ∈ R2
    ∧ b' ≡_d b''
    by force

  with p2 R0-def obtain !''' b'''' c1''' c2''' where
    passump2: V''' = [if !_b'''' then c1'''' else c2'''' fi]
    ∧ !_b'''' /∈ (set (PPc c1) ∪ set (PPc c2))
    ∧ ([c1',c1''']) ∈ R1 ∧ ([c2',c2''']) ∈ R2
    ∧ b'' ≡_d b''''
    by force

  with passump1 transR1 transR2 d-indistinguashable-trans
  have ([c1',c1'''']) ∈ R1 ∧ ([c2',c2'''']) ∈ R2
    ∧ b' ≡_d b''''
    by (metis transD)

  with passump1 passump2 R0-def have (V',V''') ∈ R0
    by auto
  }
  thus ?thesis unfolding trans-def by blast
qed

next
fix \( V, V' \)
assume inR0: \((V, V') \in R0\)
with R0-def show length \( V = \) length \( V' \) by auto

next
fix \( V', V'' \)
assume inR0: \((V', V'') \in R0\)
assume irange: \( i < \) length \( V' \)
assume notIDC: \( \neg IDC \) \((V'!i)\) \((htchLoc \( pp \) \((V'!i)\))\)

from inR0 R0-def obtain \( \iota', \iota'' \) \( b', b'' \) \( c1', c1'' \) \( c2', c2'' \)
where R0pair: \( V' = [\text{if} \iota', b' \text{then} c1' \text{else} c2' \text{fi}]\)
\( V'' = [\text{if} \iota'', b'' \text{then} c1'' \text{else} c2'' \text{fi}]\)
\( \iota' \notin \text{set} \((\text{PPc} c1) \cup \text{set} \((\text{PPc} c2)\))\)
\( \iota'' \notin \text{set} \((\text{PPc} c1) \cup \text{set} \((\text{PPc} c2)\))\)
\( ([c1'], [c1'']) \in R1 \land ([c2'], [c2'']) \in R2\)
\( b' \equiv_d b'' \)
by force

have NDC \( d \) \((if_{[i, b} \text{then} c1' \text{else} c2' \text{fi})\)
proof
\{
fix \( m \)
from nextmem-exists-and-unique obtain \( m', p, \alpha \) where ifnextmem:
\( \langle if_{[i, b} \text{then} c1' \text{else} c2' \text{fi}, m \rangle \rightarrow \langle \alpha, \langle p, m' \rangle \rangle \)
\( \forall m''. \exists p, \alpha, \langle if_{[i, b} \text{then} c1' \text{else} c2' \text{fi}, m \rangle \rightarrow \langle \alpha, \langle p, m'' \rangle \rangle \)
\( \rightarrow m'' = m' \)
by blast

hence \( m = m' \)
by (metis MWLsSteps-det.iffalse MWLsSteps-det.iftrue)

with ifnextmem have eqnextmem:
\( [if_{[i, b} \text{then} c1' \text{else} c2' \text{fi}](m) = m \)
by (simp add: NextMem-def, auto)
\}
thus \(?thesis\)
by (simp add: NDC-def)
qed

with R0pair irange show NDC \( d \) \((V'!i)\)
by simp

next
fix \( V', V'' \) \( i \) \( m1 \) \( m1' \) \( m2 \) \( \alpha \) \( p \)
assume inR0: \((V', V'') \in R0\)
assume irange: \( i < \) length \( V' \)
assume step: \( \langle V'!i, m1 \rangle \rightarrow \langle \alpha, \langle p, m2 \rangle \rangle \)
assume dhequal: \( m1 \sim_d htchLocSet PP m1' \)

from inR0 R0-def obtain \( \iota', \iota'' \) \( b', b'' \) \( c1', c1'' \) \( c2', c2'' \)
where $R0pair$: 
\[
V' = \begin{cases} V' & \text{if } \rho' b' \text{ then } c1' \text{ else } c2' \\ \text{fi} \end{cases}
\]
\[
V'' = \begin{cases} V'' & \text{if } \rho'' b'' \text{ then } c1'' \text{ else } c2'' \\ \text{fi} \end{cases}
\]
\[\cup \rho' \notin \text{set} \ (PPc \ c1) \cup \text{set} \ (PPc \ c2)\]
\[\cup \rho'' \notin \text{set} \ (PPc \ c1) \cup \text{set} \ (PPc \ c2)\]
\[\cup [(c1',c1'')] \in R1 \land [(c2',c2'')] \in R2\]
\[b' \equiv d b''\]

by force

with $R0pair$ irange step have case-distinction:
\[(p = \text{Some } c1' \land BMap (E b' m1) = \text{True}) \lor (p = \text{Some } c2' \land BMap (E b' m1) = \text{False})\]
by (simp, metis MWLsSteps-det-cases(4))

moreover
— Case 1: $b$ evaluates to True

\{ 
assume passump: $p = \text{Some } c1'$
assume eval: $BMap (E b' m1) = \text{True}$

from $R0pair$ step irange have stepconcl: $\alpha = [] \land m2 = m1$
by (simp, metis MWLs-semantics.MWLsSteps-det-cases(4))

from eval $R0pair$ dhequal have eval': $BMap (E b'' m1') = \text{True}$
by (simp add: d-indistinguishable-def dH-equal-def, auto)

hence step': $(\rho'' b'' \text{ then } c1'' \text{ else } c2'' \text{ fi}, m1') \rightarrow<[]>$
$(\text{Some } c1'', m1')$
by (simp add: MWLsSteps-det.iiftrue)

with passump $R0pair$ $R$-def dhequal stepconcl irange
have $\exists p' \alpha' m2'. (V''1, m1') \rightarrow<\alpha'> (p', m2') \land stepResultsinR p p' (R0 \cup R) \land ((\alpha, \alpha') \in R0 \lor (\alpha, \alpha') \in R) \land dhequality-alternative d \ PP \ (pp (V''i)) m2 m2'$
by (simp add: stepResultsinR-def dhequality-alternative-def, auto)

\}
moreover
— Case 2: $b$ evaluates to False

\{ 
assume passump: $p = \text{Some } c2'$
assume eval: $BMap (E b' m1) = \text{False}$

from $R0pair$ step irange have stepconcl: $\alpha = [] \land m2 = m1$
by (simp, metis MWLs-semantics.MWLsSteps-det-cases(4))

from eval $R0pair$ dhequal have eval': $BMap (E b'' m1') = \text{False}$
by (simp add: d-indistinguishable-def dH-equal-def, auto)

hence step': $(\rho'' b'' \text{ then } c1'' \text{ else } c2'' \text{ fi}, m1') \rightarrow<[]>$

38
(Some c2",m1')
by (simp add: MWLsSteps-det.iffalse)

with passump R0pair R-def dhequal stepconcl irange
have \exists p' \alpha' m2'. (V'!i,m1') \rightarrow\langle\alpha',\bot\rangle \langle p',m2'\rangle \wedge
    stepResultsinR p' (R0 \cup R) \wedge
    ((\alpha,\alpha') \in R0 \vee (\alpha,\alpha') \in R) \wedge
    dhequality-alternative d PP (pp (V'!i)) m2 m2'
    by (simp add: stepResultsinR-def dhequality-alternative-def, auto)
}
ultimately
show \exists p' \alpha' m2'. (V'!i,m1') \rightarrow\langle\alpha',\bot\rangle \langle p',m2'\rangle \wedge
    stepResultsinR p' (R0 \cup R) \wedge
    ((\alpha,\alpha') \in R0 \vee (\alpha,\alpha') \in R) \wedge
    dhequality-alternative d PP (pp (V'!i)) m2 m2'
    by auto
qed

with Areflassump2 Up-To-Technique
have SdlHPPB d PP (R0 \cup R)
    by auto

with inR0 show \exists R. SdlHPPB d PP R
    \wedge ([if b then c1 else c2 fi],[if b then c1 else c2 fi]) \in R
    by auto

qed

theorem Compositionality-While:
assumes dind: \forall d. b \equiv d b
assumes WWs-body: WHATWHERE-Secure [c]
assumes uniPPwhile: unique-PPc (while b do c od)
shows WHATWHERE-Secure [while b do c od]
proof (simp add: WHATWHERE-Secure-def, auto)
fix d PP

from uniPPwhile have pp-difference: i \notin set (PPc c)
    by (simp add: unique-PPc-def)

from WWs-body obtain R' where R'assump:
    SdlHPPB d PP R' \wedge ([c],[c]) \in R'
    by (simp add: WHATWHERE-Secure-def, auto)

    — add the empty pair because it is needed later in the proof
    def R \equiv \{(V,V'), (V,V') \in R' \wedge set (PPV V) \subseteq set (PPc c) \wedge
    set (PPV V') \subseteq set (PPc c) \} \cup \{([],[])\}

with R'assump SdlHPPB-restricted-on-PP-is-SdlHPPB

39
adding-emptypair-keeps-SdlHPPB
have SdlHPPR: SdlHPPB d PP R
proof
  from R'assump SdlHPPB-restricted-on-PP-is-SdlHPPB have
  SdlHPPB d PP
  \{ (V,V'). (V,V') \in R' \land set (PPV V) \subseteq set (PPc c) \land
  set (PPV V') \subseteq set (PPc c) \}
  by force

with adding-emptypair-keeps-SdlHPPB have
SdlHPPB d PP
\{ \{ (V,V'). (V,V') \in R' \land set (PPV V) \subseteq set (PPc c) \land
  set (PPV V') \subseteq set (PPc c) \} \cup \{([],[])\} \}
by auto

with R-def show ?thesis
by auto
qed

def R1 \equiv \{ (w1,w2). \exists \iota \iota' b b' c1 c2 c2'.
  w1 = \[c1\](while_\iota' b do c2 od)
  \land w2 = \[c1'\](while_\iota b' do c2' od)) \}
\land \iota \notin set (PPc c) \land \iota' \notin set (PPc c)
\land ([c1],[c1']) \in R \land ([c2],[c2']) \in R
\land b \equiv_d b' \}

def R2 \equiv \{ (w1,w2). \exists \iota \iota' b b' c1 c1'.
  w1 = \[while_\iota b do c1 od\]
  \land w2 = \[while_\iota' b' do c1' od\]
\land \iota \notin set (PPc c) \land \iota' \notin set (PPc c) \land
([c1],[c1']) \in R \land b \equiv_d b' \}

def R0 \equiv R1 \cup R2

from R2-def R-def R'assump pp-difference dind
have inR2: ([while_\iota b do c od],[while_\iota b do c od]) \in R2
by auto

from R0-def R1-def R2-def R-def R'assump have
Domain R0 \cap Domain R = \{}
by auto

with commonArefl-subset-commonDomain
have Areflassump: Arefl R0 \cap Arefl R \subseteq \{[]\}
by force

— show some facts about R1 and R2 needed later in the proof at several positions

from SdlHPPR have symR: sym R
from symR R1-def d-indistinguishable-sym have symR1: sym R1
by (simp add: sym-def, fastforce)

have symR2: sym R2
by (simp add: sym-def, fastforce)

have disjuptoR1R2: disj-dlHPP-Bisimulation-Up-To-R′ d PP R R0
proof (simp add: disj-dlHPP-Bisimulation-Up-To-R'-def, auto)
from SdlHPPR show SdlHPPB d PP R
by auto

next
from symR1 symR2 R0-def show sym R0
by (simp add: sym-def)

next
from SdlHPPR have transR: trans R
by (simp add: Strong-dlHPP-Bisimulation-def)

have transR1: trans R1
proof
{ fix V V' V''
assume p1: (V,V') ∈ R1
assume p2: (V',V'') ∈ R1

from p1 R1-def obtain i i' b b' c1 c1' c2 c2' where passump1:
  V = [c1;while_i b do c2 od]
  ∧ V' = [c1';while_i' b' do c2' od]
  ∧ i /∈ set (PPc c) ∧ i' /∈ set (PPc c)
  ∧ ([c1],[c1']) ∈ R ∧ ([c2],[c2']) ∈ R
  ∧ b ≡_d b'
  by force

with p2 R1-def obtain i'' b'' c1'' c2'' where passump2:
  V'' = [c1'';while_i'' b'' do c2'' od] ∧ i'' /∈ set (PPc c)
  ∧ ([c1''],[c1'']) ∈ R ∧ ([c2''],[c2'']) ∈ R
  ∧ b'' ≡_d b''
  by force

with passump1 transR d-indistinguishable-trans have ([c1],[c1'']) ∈ R ∧ ([c2],[c2'']) ∈ R ∧ b ≡_d b''
  by (metis trans-def)

with passump1 passump2 R1-def have (V,V'') ∈ R1
by auto

} thus ?thesis unfolding trans-def by blast
qed

have transR2: trans R2

41
proof

\{ 
  fix \ V \ V' \ V'' 
  assume p1: \((V, V') \in R2\) 
  assume p2: \((V', V'') \in R2\) 

  from p1 R2-def obtain \(\iota \ \iota' \ b \ b' \ c1 \ c1'\) where passump1: 
  \(V = [\text{while} \ \iota \ b \ \text{do} \ c1 \ \text{ad}]\) 
  \(\land \ V' = [\text{while} \ \iota' \ b' \ \text{do} \ c1' \ \text{ad}]\) 
  \(\land \ i \ \notin \ \text{set} \ (PPc \ c) \ \land \ i' \ \notin \ \text{set} \ (PPc \ c)\) 
  \(\land \ ([c1],[c1']) \in R \ \land \ b \equiv_d b'\) 
  by force 

  with p2 R2-def obtain \(\iota'' \ b'' \ c1''\) where passump2: 
  \(V'' = [\text{while} \ \iota'' \ b'' \ \text{do} \ c1'' \ \text{ad}]\) 
  \(\land \ (\iota' \ \notin \ \text{set} \ (PPc \ c)\) 
  \(\land \ ([c1],[c1'']) \in R \ \land \ b' \equiv_d b''\) 
  by force 

  with passump1 transR d-indistinguishable-trans 
  have \((c1,c1'') \in R \ \land \ b \equiv_d b''\) 
  by (metis trans-def) 

  with passump1 passump2 R2-def have \((V, V'') \in R2\) 
  by auto 

  thus ?thesis unfolding trans-def by blast 
qed 

from SdlHPPR have eqlenR: \(\forall (V, V') \in R. \ \text{length} \ V = \text{length} \ V'\) 
by (simp add: Strong-dlHPP-Bisimulation-def) 
from R1-def eqlenR have eqlenR1: \(\forall (V, V') \in R1. \ \text{length} \ V = \text{length} \ V'\) 
by auto 
from R2-def eqlenR have eqlenR2: \(\forall (V, V') \in R2. \ \text{length} \ V = \text{length} \ V'\) 
by auto 

from R1-def R2-def have Domain R1 \(\cap\) Domain R2 = \(\{\}\) 
by auto 

with commonArefl-subset-commonDomain have Arefl-a: 
Arefl R1 \(\cap\) Arefl R2 = \(\{\}\) 
by force 

with symR1 symR2 transR1 transR2 eqlenR1 eqlenR2 trans-RuR' 
have trans (R1 \(\cup\) R2) 
  by force 
with R0-def show trans R0 by auto 
next 
fix \ V \ V' 
assume inR0: \((V, V') \in R0\) 

42
with R0-def R1-def R2-def show length V = length V' by auto 

fix V V' i 
assume inR0: (V, V') ∈ R0 
assume irange: i < length V 
assume notIDC: ¬ IDC (V!i) 
(hatchLoc (pp (V!i)))

from inR0 R0-def R1-def R2-def obtain i' b' c1' c2' c1 c2 
where R0pair: ((V = [c1; (while b do c2 od)]) 
∧ V' = [c1'; (while b' do c2' od)]) 
∨ (V = [while b do c1 od]; V' = [while b' do c1' od]) 
∧ i /∈ set (PPc c) ∧ i' /∈ set (PPc c) 
∧ ([c1][c1']) ∈ R ∧ ([c2][c2']) ∈ R 
∧ b ≡ d b' 
by force

with irange SdlHPPR strongdlHPPB-NDCIDCaux 
[of d PP R [c1] [c1'] i] 
have c1NDCIDC: 
NDC d c1 ∨ IDC d c1 (hatchLoc (pp c1)) 
by auto

— first alternative for V and V'

have case1: NDC d (c1; (while b do c2 od)) ∨ 
IDC d (c1; (while b do c2 od)) 
(hatchLoc (pp (c1; (while b do c2 od))))

proof − 
have eqnextmem: ∀ m. [c1; (while b do c2 od)](m) = [c1](m) 
proof − 
fix m 
from nextmem-exists-and-unique obtain m' where c1nextmem: 
∃ p α. (c1,m) →<α> ⟨p,m'⟩ 
∧ (∀ m''. (∃ p α. (c1,m) →<α> ⟨p,m''⟩) → m'' = m') 
by force 

hence eqdir1: [c1](m) = m' 
by (simp add: NextMem-def, auto)

from c1nextmem obtain p α where (c1,m) →<α> ⟨p,m'⟩ 
by auto

with c1nextmem have 
∃ p α. (c1; (while b do c2 od),m) →<α> ⟨p,m'⟩ 
∧ (∀ m''. (∃ p α. (c1; (while b do c2 od),m) →<α> ⟨p,m''⟩) 
→ m'' = m') 
by (auto, metis MWLsSteps-det.seq1 MWLsSteps-det.seq2 
option.exhaust, metis MWLsSteps-det-cases(3))
hence eqdir2: $[c1:(\text{while}_b \text{ do } c2 \text{ od})](m) = m'$
  by (simp add: NextMem-def, auto)

with eqdir1 show $[c1:(\text{while}_b \text{ do } c2 \text{ od})](m) = [c1](m)$
  by auto
qed

have ecpp: $\text{pp } (c1:(\text{while}_b \text{ do } c2 \text{ od})) = \text{pp } c1$
  by simp

with c1NDCIDC eqnextmem show
  $\text{NDC } d (c1:(\text{while}_b \text{ do } c2 \text{ od})) \lor$
  $\text{IDC } d (c1:(\text{while}_b \text{ do } c2 \text{ od}))$
  $\text{(htchLoc } (\text{pp } (c1:(\text{while}_b \text{ do } c2 \text{ od}))))$
  by (simp add: IDC-def NDC-def)
qed

— second alternative for $V \ V'$

have case2: $\text{NDC } d ~ (\text{while}_b \text{ do } c1 \text{ od})$
proof

{}  

fix $m$

from nextmem-exists-and-unique obtain $m' \ p \ a$
  where whilenextmem: $\langle \text{while}_b \text{ do } c1 \text{ od},m \rangle \rightarrow <a> \langle p,m' \rangle$
  $\land (\forall m'' : (\exists \ p \ a. \langle \text{while}_b \text{ do } c1 \text{ od},m \rangle \rightarrow <a> \langle p,m'' \rangle)$
  $\rightarrow m'' = m')$
  by blast

hence $m = m'$
  by (metis MWLsSteps-det.whilefalse MWLsSteps-det.whiletrue)

with whilenextmem have eqnextmem:
  $[\text{while}_b \text{ do } c1 \text{ od}] (m) = m$
  by (simp add: NextMem-def, auto)

}  

thus $\text{NDC } d (\text{while}_b \text{ do } c1 \text{ od})$
  by (simp add: NDC-def)
qed

from $R0pair$ case1 case2 irange notIDC
show $\text{NDC } d (V!i)$
  by force

next

fix $V \ V' \ i \ m1 \ m1' \ m2 \ a \ p$

assume inR0: $(V,V') \in R0$
assume irange: $i < \text{length } V$
assume step: $\langle V!i,m1 \rangle \rightarrow <a> \langle p,m2 \rangle$
assume dhequal: $m1 \sim_d \text{htchLocSet } PP \ m1'$

44
from inR0 R0-def R1-def R2-def obtain \( \forall \iota \iota' b' c1 c1' c2 c2' \)

where R0pair: \((V = [c1;\text{while}, b do c2 od]) Λ V' = [c1'(\text{while}, b' do c2' od)])\)

\( \lor (V = [\text{while}, b do c1 od] Λ V' = [\text{while}, b' do c1' od])\)

\( Λ \iota \not\in \text{set (PPr c)} Λ \iota' \not\in \text{set (PPr c)} \)

\( \land \ (\{c1\},\{c1'\}) \in R \land \ (\{c2\},\{c2'\}) \in R \)

\( \land \ b \equiv d b' \)

by force

— Case 1: \( V \) and \( V' \) are sequential commands

have case1: \[
\begin{align*}
V &= [c1;\text{while}, b do c2 od]; \\
V' &= [c1';\text{while}, b' do c2' od];
\end{align*}
\]

\[
\Rightarrow \exists p' α' m2'. (V∧i,m1') →<α'> (p',m2') \land
\]

\( \text{stepResultsinR} \ p \ p' (R0 ∪ R) \land \ (\alpha,α') \in R0 \lor \ (\alpha,α') \in R \land \)

\( \text{dhequality-alternative d PP} \ (pp \ (V!i)) \ m2 m2' \)

proof —

assume Vassump: \( V = [c1;\text{while}, b do c2 od] \)

assume V'assump: \( V' = [c1';\text{while}, b' do c2' od] \)

have eqpp: \( pp \ (c1;\text{while}, b do c2 od) = pp \ c1 \)

by simp

from Vassump irange step irange obtain c3

where case-distinction:

\( (p = \text{Some} \ (\text{while}, b do c2 od) Λ \{c1,m1\} →<α> (\text{None},m2)) \)

\( \lor (p = \text{Some} \ (c3;\text{while}, b do c2 od)) \)

\( \land \ (c1,m1) →<α> \{\text{Some} c3,m2\}) \)

by (simp, metis MWLseq-det-cases(3))

moreover

— Case 1a: first command terminates

{ }

assume passump: \( p = \text{Some} \ (\text{while}, b do c2 od) \)

assume stepassump: \( \{c1,m1\} →<α> \{\text{None},m2\} \)

with R0pair SdlHPPR dhequal
strongdHPPB-aux[of d PP R]

- \([c1]\ [c1'] m1 \text{ α None m2 m1} \]

obtain \( p' α' m2' \) where c1c1't reason:

\( p' = \text{None} Λ \{c1',m1'\} →<α'> \{p',m2'\} Λ \ (\alpha,α') \in R \land \)

\( \text{dhequality-alternative d PP} \ (pp \ c1) m2 m2' \)

by (simp add: stepResultsinR-def, fastforce)

hence conclpart:

\( \{c1';\text{while}, b' do c2' od,m1'\} \)

\( →<α'> \{\text{Some} \ (\text{while}, b' do c2' od),m2'\} \)

by (auto, simp add: MWLseq-det.seq1)

from R0pair R0-def R2-def have

\( ([\text{while}, b do c2 od],[\text{while}, b' do c2' od]) \in R0 \)
by simp

with passump V’assump Vassump eqpp conclpart
c1c1’reason irange
have \( \exists p' \alpha' m2'. (V' i, m1') \rightarrow_{<\alpha'} \langle p', m2' \rangle \wedge \)
stepResultsinR p p' (R0 \cup R) \wedge ((\alpha, \alpha') \in R0 \vee 
(\alpha, \alpha') \in R) \wedge 
dhequality-alternative d PP (pp (V!i)) m2 m2'
by (simp add: stepResultsinR-def, auto)

moreover
— Case 1b: first command does not terminate
{
assume passump: \( p = \text{Some} (c3; (while \_ b do c2 od)) \)
assume stepassump: \( \langle c1, m1 \rangle \rightarrow_{<\alpha} \langle \text{Some} c3, m2 \rangle \)

with R0pair SdlHPPR dhequal
strongdHPPB-aux[of d PP R]
- [c1] [c1'] m1 \alpha Some c3 m2 m1’

obtain p' c3' \alpha' m2' where c1c1’reason:
p' = Some c3' \wedge \langle c1', m1' \rangle \rightarrow_{<\alpha'} \langle p', m2' \rangle \wedge
([c3'], [c3']) \in R \wedge (\alpha, \alpha') \in R \wedge 
dhequality-alternative d PP (pp c1) m2 m2'
by (simp add: stepResultsinR-def, fastforce)

hence conclpart:
\( \langle c1' (while \_ b' do c2' od), m1' \rangle \rightarrow_{<\alpha'} \langle p', m2' \rangle \wedge \)
\( \langle \text{Some} (c3', while \_ b' do c2' od), m2' \rangle \)
by (auto, simp add: MWLsSteps-det.seq2)

from c1c1’reason R0pair R0-def R1-def have
\( ([c3; while \_ b do c2 od], [c3'; while \_ b' do c2' od]) \in R0 \)
by simp

with passump V’assump Vassump eqpp conclpart c1c1’reason irange
have \( \exists p' \alpha' m2'. (V' i, m1') \rightarrow_{<\alpha'} \langle p', m2' \rangle \wedge \)
stepResultsinR p p' (R0 \cup R) \wedge 
((\alpha, \alpha') \in R0 \vee (\alpha, \alpha') \in R) \wedge 
dhequality-alternative d PP (pp (V!i)) m2 m2'
by (simp add: stepResultsinR-def, auto)

} ultimately show \( \exists p' \alpha' m2'. (V' i, m1') \rightarrow_{<\alpha'} \langle p', m2' \rangle \wedge \)
stepResultsinR p p' (R0 \cup R) \wedge 
((\alpha, \alpha') \in R0 \vee (\alpha, \alpha') \in R) \wedge 
dhequality-alternative d PP (pp (V!i)) m2 m2'
by auto

qed

— Case 2: V and V' are while commands
**have case 2**: \[ V = \{ \{ \text{while}_i \ b \ do \ c1 \ od \} \} \]
\[ V' = \{ \{ \text{while}_i' \ b' \ do \ c1' \ od \} \} \]
\[ \implies \exists p' \alpha' m2'. \langle V'i,m1' \rangle \rightarrow <\alpha' > \langle p',m2' \rangle \land \]
\[ \text{stepResultsinR} \ p p' (R0 \cup R) \land ((\alpha,\alpha') \in R0 \lor (\alpha,\alpha') \in R) \land \]
\[ \text{dhequality-alternative} \ d \ PP \ (\text{pp} (V'i)) \ m2 m2' \]

**proof**

- assume \ Vassump: \ V = \{ \{ \text{while}_i \ b \ do \ c1 \ od \} \}
- assume \ V'assump: \ V' = \{ \{ \text{while}_i' \ b' \ do \ c1' \ od \} \}

**from Vassump irange step have case-distinction**:
\( (p = \text{None} \land \text{BMap} (E \ b \ m1) = \text{False}) \lor p = (\text{Some} (c1; \text{while}_i \ b \ do \ c1 \ od)) \land \text{BMap} (E \ b \ m1) = \text{True} \)

- by \ (\text{simp, metis MWLsSteps-det-cases(5) option.simps(2)})

**moreover**
- Case 2a: \ b evaluates to False

{  
  assume \ passump: \ p = \text{None}  
  assume \ eval: \ \text{BMap} (E \ b \ m1) = \text{False} 

  with \ Vassump step irange have \ stepconcl: \ \alpha = \[ \land m2 = m1 
  by \ (\text{simp, metis (no-types) MWLsSteps-det-cases(5)})

  from \ eval R0pair dhequal have \ eval': \ \text{BMap} (E \ b' \ m1') = \text{False} 
  by \ (\text{simp add: d-indistinguishable-def dH-equal-def, auto})

  hence \ \langle \text{while}_i', \ b' \ do \ c1' \ od, m1' \rangle \rightarrow <\[\] > \langle \text{None}, m1' \rangle 
  by \ (\text{simp add: MWLsSteps-det.whilefalse})

  with \ passump R-def Vassump V'assump stepconcl dhequal irange have \ \exists p' \alpha' m2'. \langle V'i,m1' \rangle \rightarrow <\alpha' > \langle p',m2' \rangle \land 

  stepResultsinR \ p p' (R0 \cup R) \land ((\alpha,\alpha') \in R0 \lor (\alpha,\alpha') \in R) \land 
  \text{dhequality-alternative} \ d \ PP \ (\text{pp} (V'i)) \ m2 m2' 
  by \ (\text{simp add: stepResultsinR-def dhequality-alternative-def, auto})

} 

**moreover**
- Case 2b: \ b evaluates to True

{  
  assume \ passump: \ p = (\text{Some} (c1; \text{while}_i \ b \ do \ c1 \ od))  
  assume \ eval: \ \text{BMap} (E \ b \ m1) = \text{True} 

  with \ Vassump step irange have \ stepconcl: \ \alpha = \[ \land m2 = m1 
  by \ (\text{simp, metis (no-types) MWLsSteps-det-cases(5)})

  from \ eval R0pair dhequal have \ eval': \ \text{BMap} (E \ b' \ m1') = \text{True} 
  by \ (\text{simp add: d-indistinguishable-def dH-equal-def, auto})

  hence \ \langle \text{while}_i', \ b' \ do \ c1' \ od, m1' \rangle \rightarrow <\[\] > \langle \text{Some} (c1', \text{while}_i' \ b' \ do \ c1' \ od), m1' \rangle 

  ...
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

Locale: Type-System

Fixes: AssignSideCondition :: 'id ⇒ 'exp ⇒ nat ⇒ bool

And: WhileSideCondition :: 'exp ⇒ bool

And: IfSideCondition ::

'(exp, 'id) MWLsCom ⇒ (exp, 'id) MWLsCom ⇒ bool

Assumes: semAssignSC: AssignSideCondition x e ι ⇒ e ≡ DA x, (htchLoc ι) e ∧ (∀m m' d ι'. (m ∼ d, (htchLoc ι')) m' ∧

[x := e](m) =d [x := e](m'))

⇒ [x := e](m) ∼d (htchLoc ι) [x := e](m'))

And: semWhileSC: WhileSideCondition e ⇒ ∀d. e ≡d e

And: semIfSC: IfSideCondition e c1 c2 ⇒ ∀d. e ≡d e

begin

— Security typing rules for the language commands

Inductive: ComSecTyping :: (exp, 'id) MWLsCom ⇒ bool

(⊢C -)

And: ComSecTypingL :: (exp,'id) MWLsCom list ⇒ bool

(⊢V -)

Where:

Skip: ⊢C skip |

Assign: [ AssignSideCondition x e ι ] ⇒ ⊢C x := e |

Spawn: [ ⊢V V ] ⇒ ⊢C spawn e V |

Seq: [ ⊢C c1; ⊢C c2 ] ⇒ ⊢C c1;c2 |

While: [ ⊢C c; WhileSideCondition b ]

⇒ ⊢C while b do c od |

If: [ ⊢C c1; ⊢C c2; IfSideCondition b c1 c2 ]

⇒ ⊢C if b then c1 else c2 fi |

Parallel: [ ∀i < length V. ⊢V V!i ] ⇒ ⊢V V
inductive-cases parallel-cases:

⊢ V

definition auxiliary-predicate
where
auxiliary-predicate V ≡ unique-PPV V → WHATWHERE-Secure V

— soundness proof of abstract type system

theorem ComSecTyping-single-is-sound:

\[ \begin{align*}
\| C \; c; \text{unique-PPc} \; c \| & \Rightarrow \text{WHATWHERE-Secure} \; [c] \\
\end{align*} \]

proof (induct rule: ComSecTyping-ComSecTypingL.inducts(1)
[of - auxiliary-predicate], simp-all add: auxiliary-predicate-def)

fix t

show WHATWHERE-Secure [skip_t]

by (metis WHATWHERE-Secure-Skip)

next

fix x e t

assume AssignSideCondition x e t

thus WHATWHERE-Secure [x :=_t e]

by (metis WHATWHERE-Secure-Assign semAssignSC)

next

fix V t

assume IH: unique-PPV V → WHATWHERE-Secure V

assume uniPPspawn: unique-PPc (spawn_t V)

hence unique-PPV V

by (simp add: unique-PPV-def unique-PPc-def)

with IH have WHATWHERE-Secure V

.. with uniPPspawn show WHATWHERE-Secure [spawn_t V]

by (metis Compositionality-Spawn)

next

fix c1 c2

assume IH1: unique-PPc c1 ⇐ WHATWHERE-Secure [c1]

assume IH2: unique-PPc c2 ⇐ WHATWHERE-Secure [c2]

assume uniPPc1c2: unique-PPc (c1;c2)

from uniPPc1c2 have uniPPc1: unique-PPc c1

by (simp add: unique-PPc-def)

with IH1 have IS1: WHATWHERE-Secure [c1]

from uniPPc1c2 have uniPPc2: unique-PPc c2

by (simp add: unique-PPc-def)

with IH2 have IS2: WHATWHERE-Secure [c2]

from IS1 IS2 uniPPc1c2 show WHATWHERE-Secure [c1;c2]

by (metis Compositionality-Seq)

next
fix c b ι.
assume SC: WhileSideCondition b
assume IH: unique-PPc c ⇒ WHATWHERE-Secure [c]
assume uniPPwhile: unique-PPc (while _b do c od)
hence unique-PPc c
  by (simp add: unique-PPc-def)
with IH have WHATWHERE-Secure [c]

. with uniPPwhile SC show WHATWHERE-Secure [while _b do c od]
  by (metis Compositionality-While semWhileSC)

next
fix c1 c2 b ι
assume SC: IfSideCondition b c1 c2
assume IH1: unique-PPc c1 ⇒ WHATWHERE-Secure [c1]
assume IH2: unique-PPc c2 ⇒ WHATWHERE-Secure [c2]
assume uniPPif: unique-PPc (if _b then c1 else c2 fi)
from uniPPif have unique-PPc c1
  by (simp add: unique-PPc-def)
with IH1 have IS1: WHATWHERE-Secure [c1]

. from uniPPif have unique-PPc c2
  by (simp add: unique-PPc-def)
with IH2 have IS2: WHATWHERE-Secure [c2]

. from IS1 IS2 SC uniPPif show
  WHATWHERE-Secure [if _b then c1 else c2 fi]
  by (metis Compositionality-If semIfSC)

next
fix V
assume IH: ∀i < length V. ⊢ C V ! i ∧
  (unique-PPc (V ! i) ⇒ WHATWHERE-Secure [V!i])
have unique-PPV V ⇒ (∀i < length V. unique-PPc (V ! i))
  by (metis uniPPV-uniPPc)
with IH have unique-PPV V ⇒ (∀i < length V. WHATWHERE-Secure [V!i])

  by auto
thus uniPPV: unique-PPV V ⇒ WHATWHERE-Secure V
  by (metis parallel-composition)
qed

theorem ComSecTyping-list-is-sound:
[∀V. unique-PPV V ] ⇒ WHATWHERE-Secure V
by (metis ComSecTyping-single-is-sound parallel-cases
parallel-composition uniPPV-uniPPc)

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 ⇒ ’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 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 (ExprEval 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.

\textbf{datatype} \ Dom = low \ | \ high

\textbf{instantiation} \ Dom :: order \\
\textbf{begin} \\
\textbf{definition} \ less-eq-Dom-def: \ d1 \leq d2 = (if \ d1 = d2 \ then \ True \ else \ (if \ d1 = \ low \ then \ True \ else \ False))

\textbf{definition} \ less-Dom-def: \ d1 < d2 = (if \ d1 = d2 \ then \ False \ else \ (if \ d1 = \ low \ then \ True \ else \ False))

\textbf{instance proof} \\
fix \ x \ y \ z :: \ Dom \\
show \ (x < y) = (x \leq y \land \neg \leq x) \\
\textit{unfolding} \ less-eq-Dom-def \ less-Dom-def \ by \ \textit{auto} \\
show \ [x \leq y; \ y \leq z] \Rightarrow \ x \leq z \\
\textit{unfolding} \ less-eq-Dom-def \ by \ ((\textit{split\ split-if-asm})+, \ \textit{auto}) \\
show \ [x \leq y; \ y \leq x] \Rightarrow \ x = y \\
\textit{unfolding} \ less-eq-Dom-def \ by \ ((\textit{split\ split-if-asm})+, \\
\textit{auto}, \ (\textit{split\ split-if-asm})+, \ \textit{auto}) \\
\textbf{qed} \\
\textbf{end} \\
\textbf{end} \\
\textbf{theory} \ Type-System-example \\
\textbf{imports} \ Type-System ../Strong-Security/Expr ../Strong-Security/Domain-example \\
\textbf{begin} \\
— \ When interpreting, we have to instantiate the type for domains. As an example, \\
we take a type containing 'low' and 'high' as domains.

\textbf{consts} \ DA :: ('id,Dom) DomainAssignment \\
\textbf{consts} \ BMap :: 'val \Rightarrow bool \\
\textbf{consts} \ LH :: (Dom,('id,'val) Expr) LHatches \\
— \ redefine all the abbreviations necessary for auxiliary lemmas with the correct 
parameter instantiation

\textbf{abbreviation} \ MWLsStepsdet' :: \\
\textit{(('id,'val) Expr, 'id, 'val, (('id,'val) Expr,'id) MWLsCom) TLSsteps-curry} \\
\textit{((1,-/-)) \rightarrow <\rightarrow/ \ ((1,-/-)) [0,0,0,0,0] 81)} \\
\textbf{where}
\( \langle c_1, m_1 \rangle \rightarrow^{<\alpha>} \langle c_2, m_2 \rangle \equiv \langle (c_1, m_1), \alpha, (c_2, m_2) \rangle \in \text{MWLs-semantics}.\text{MWLsSteps-det} \text{ ExprEval BMap} \)

**abbreviation** \( d\text{-equal}' :: \langle \text{'id, 'val} \rangle \text{ State} \quad \Rightarrow \text{Dom} \Rightarrow \langle \text{'id, 'val} \rangle \text{ State} \Rightarrow \text{bool} \)

\[
\begin{align*}
& (\ (- = \ -) ) \\
& \text{where} \\
& m =_d m' \equiv \text{WHATWHERE}.d\text{-equal} \text{ DA} d m m'
\end{align*}
\]

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

**abbreviation** \( \text{NextMem}' :: \langle \text{('id, 'val)} \text{ Expr}, \text{'id} \rangle \text{ MWLsCom} \\
\Rightarrow \langle \text{('id, 'val)} \text{ State} \Rightarrow \langle \text{('id, 'val)} \text{ State} \\
\begin{bmatrix} \text{'}(-) \end{bmatrix} \text{'}() \text{ State} \\
\text{where} \\
\begin{bmatrix} \text{c} (\text{m}) \end{bmatrix} \equiv \text{WHATWHERE}.\text{NextMem} (\text{MWLs-semantics}.\text{MWLsSteps-det} \text{ ExprEval BMap}) \\
c \quad m \\
\text{abbreviation} \ (dH\text{-indistinguishable}' :: \langle \text{('id, 'val)} \text{ Expr} \Rightarrow \text{Dom} \\
\Rightarrow (\text{Dom}.\langle \text{('id, 'val)} \text{ Expr} \text{ Hatches} \Rightarrow \langle \text{('id, 'val)} \text{ Expr} \Rightarrow \text{bool} \\
\quad (\ (- \equiv, \ -) ) \\
\text{where} \\
\text{e} 1 \equiv_{d,H} \text{e} 2 \\
\equiv \text{WHATWHERE-Secure-Programs}.dH\text{-indistinguishable} \text{ ExprEval DA} d H e 1 e 2 \)

**abbreviation** \( \text{htchLoc} :: \text{nat} \Rightarrow (\text{Dom}, \langle \text{'id, 'val} \rangle \text{ Expr} \text{ Hatches} \\
\text{where} \\
\text{htchLoc} l \equiv \text{WHATWHERE}.\text{htchLoc} lH l \)

---

**Security typing rules for expressions**

**inductive**

\( \text{ExprSecTyping} :: (\text{Dom}, \langle \text{'id, 'val} \rangle \text{ Expr} \text{ Hatches} \Rightarrow (\langle \text{'id, 'val} \rangle \text{ Expr} \Rightarrow \text{Dom} \Rightarrow \text{bool} \\
(\ (- \vdash \cdot \vdash \cdot) \)

**for** \( H :: (\text{Dom}, \langle \text{'id, 'val} \rangle \text{ Expr} \text{ Hatches} \\
\text{where} \\
\text{Consts}: \ H \vdash_{\ell} (\text{Const} \ v) : d \mid \\
\text{Vars}: \ \text{DA} \ x = d \Rightarrow \ H \vdash_{\ell} (\text{Var} \ x) : d \mid \\
\text{Hatch}: \ \text{Hatch} : (d, e) \in H \Rightarrow \ H \vdash_{\ell} e : d \mid \\
\text{Ops}: \ \begin{bmatrix} \forall i < \text{length} \ \text{arglist}. \ H \vdash_{\ell} (\text{arglist}!i) : (d || i) \land (d || i) \leq d \end{bmatrix} \\
\Rightarrow H \vdash_{\ell} (\text{Op} \ f \ \text{arglist}) : d \)
— function substituting a certain expression with another expression in expressions

primrec Subst :: ('id, 'val) Expr ⇒ ('id, 'val) Expr
⇒ ('id, 'val) Expr ⇒ ('id, 'val) Expr
(\<\cdot\>,\<\cdot\>)

and SubstL :: ('id, 'val) Expr list ⇒ ('id, 'val) Expr
⇒ ('id, 'val) Expr ⇒ ('id, 'val) Expr list

where
(\Const\ v) <\{\ Const\ v \} \> = (\if\ Const\ v \then\ Const\ v \else\ \Const\ v \}\) |
(\Var\ x) <\{\ Var\ x \} \> = (\if\ Var\ x \then\ Var\ x \else\ \Var\ x \}\) |
(\Op\ f\ arglist) <\{\ Op\ f\ arglist \} \> = (\if\ Op\ f\ arglist \then\ Op\ f\ arglist \else\ Op\ f\ arglist \}\)
(\Op\ f\ arglist) <\{\ SubstL\ arglist\ e1\ e2 \} \>

SubstL \{\ e1\ e2 \} \= \{\ (\<\{\ Const\ v\ \} \>\) \}
SubstL (\<\{\ e1\ e2 \}\)\#(\SubstL\ V\ e1\ e2)

definition SubstClosure :: 'id ⇒ ('id, 'val) Expr ⇒ bool

where
SubstClosure x e \equiv \forall\ d',e',i' \in\ IH. (d',e'\in\ (\Var\ x)\ \e\ \i' \in\ IH) |

definition synAssignSC :: 'id ⇒ ('id, 'val) Expr ⇒ nat ⇒ bool

where
synAssignSC x e i \equiv \exists\ d. ((\hitchLoc\ i) ⇒ e : \ d \land \ d \leq \ DA\ x)
∧ (SubstClosure\ x\ e)

definition synWhileSC :: ('id, 'val) Expr ⇒ bool

where
synWhileSC e \equiv \exists\ d. (\{\} ⇒ e : \ d) \land (\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. (\{\} ⇒ e : \ d \land (\forall\ d',d \leq \ d'))

— auxiliary lemma for locale interpretation (theorem 7 in original paper)

lemma ExprTypable-with-smallerld-implies-dH-indistinguishable:
[\ H \Rightarrow e : \ d' \land \ d \leq \ d' \Rightarrow e \equiv_d H\ e]

proof (induct rule: ExprSecTyping.induct,
  simp-all add: WHATWHERE-Secure-Programs.dH-indistinguishable-def WHATWHERE.dH-equal-def WHATWHERE.d-equal-def, auto)

fix \ dl\ arglist\ f\ m1\ m2\ d'\ d

assume main: \forall\ i < \ length\ arglist.
(\ H \Rightarrow f\ (\arglist\ i)) \land (\dl!i \leq \ d \Rightarrow
\\forall\ m1\ m2. (\forall\ x. DA\ x \leq \ d \Rightarrow m1\ x = m2\ x) \land
(\forall\ (d',e) \in\ H. d' \leq \ d \Rightarrow ExprEval\ e\ m1 = ExprEval\ e\ m2) \Rightarrow Evaluate\ (arglist!i)\ m1 = Evaluate\ (arglist!i)\ m2) \land \dl!i \leq \ d'
assume smaller: \( d' \leq d \)
assume eeqval: \( \forall (d', e) \in H, d' \leq d \rightarrow \text{ExprEval } e \ m1 = \text{ExprEval } e \ m2 \)
assume eeqstate: \( \forall x. \ DA \ x \leq d \rightarrow m1 \ x = m2 \ x \)

from main smaller have irangesubst:
\[ \forall i < \text{length arglist}, d!i \leq d \]
by (metis order-trans)

with eeqstate eeqval main have
\[ \forall i < \text{length arglist}, (\text{arglist}!i) m1 \]
\[ = \text{ExprEval} (\text{arglist}!i) m2 \]
by force

hence substmap: \( (\text{ExprEvalL arglist } m1) = (\text{ExprEvalL arglist } m2) \)
by (induct arglist, auto, force)

show \( f (\text{ExprEvalL arglist } m1) = f (\text{ExprEvalL arglist } m2) \)
by (subst substmap, auto)

qed

— auxiliary lemma about substitutions in expressions and in memories

lemma substexp-substmem:
\[ \text{ExprEval } e' <\text{Var } x \setminus e > m = \text{ExprEval } e' (m(x := \text{ExprEval } e \ m)) \]
\[ \land \text{ExprEvalL} (\text{SubstL elist (Var } x) \ e ) m \]
\[ = \text{ExprEvalL elist} (m(x := \text{ExprEval } e \ m)) \]
by (induct-tac e' and elist rule: ExprEval, induct ExprEvalL, induct, simp-all)

— another auxiliary lemma for locale interpretation (lemma 8 in original paper)

lemma SubstClosure-implications:
\[ \[ \text{SubstClosure } x \ e; m \sim d,(\text{htchLoc } i') \ m' \]
\[ \Rightarrow [x := _i \ e](m) \sim d,(\text{htchLoc } i') \ [x := _i \ e](m') \]
proof –
fix \( m1, m1' \)
assume substclosure: SubstClosure \( x \ e \)
assume dequalm2: \[ x := _i \ e \](m1) =_d [x := _i \ e \](m1')
assume ddequalm1: \( m1 \sim_d,(\text{htchLoc } i') \ m1' \)

from MWLs-semantics.nextmem-exists-and-unique obtain \( m2 \) where m1step:
\[ (\exists p \ alpha. (x := _i \ e, m1) \rightarrow<\alpha> (p, m2)) \]
\[ \land (\forall m'. (\exists p \ alpha. (x := _i \ e, m1) \rightarrow<\alpha> (p, m'')) \rightarrow m'' = m2) \]
by force
hence m2-is-next: \[ [x := _i \ e](m1) = m2 \]
by (simp add: WHATWHERE.NextMem-def, auto)

from m1step MWLs-semantics.MWLsteps-det.assign
\[ \text{of } \text{ExprEval } e \ m1 - x \ i \ BMap \]
have m2eq: \( m2 = m1(x := (\text{ExprEval } e \ m1)) \)
from MWLs-semantics.nextmem-exists-and-unique obtain \( m' \) where \( m' \):
\[
\begin{align*}
(\exists p. \alpha. \langle x := e, m1' \rangle \rightarrow \langle \alpha \rangle \langle p, m' \rangle) \\
\wedge (\forall m''. (\exists p. \alpha. \langle x := e, m1' \rangle \rightarrow \langle \alpha \rangle \langle p, m'' \rangle) \rightarrow m'' = m')
\end{align*}
\]
by force
hence \( m' \) is next:
\[
[x := e](m1') = m'
\]
by force
from \( m' \) step MWLs-semantics.MWLsSteps-det assign
\[
\text{have } m' \text{eq: } m' = m1'(x := (ExprEval e m1'))
\]
by auto

from \( m' \) substexp-substmem
have substeval1: \( \forall e'. \langle e' < Var x \rangle m1 = ExprEval e' \rangle m2 \)
by force
from \( m' \) substexp-substmem
have substeval2: \( \forall e'. \langle e' < Var x \rangle m1' = ExprEval e' \rangle m2' \)
by force

from substclosure have
\[
\forall (d',e') \in htcLoc \ l'. (d',e' < Var x \rangle \in (htcLoc \ l'))
\]
by (simp add: SubstClosure-def WHATWHERE.htcLoc-def, auto)

with dhequalm1 have
\[
\forall (d',e') \in htcLoc \ l'.
\]
\[
d' \leq d \rightarrow \langle e' < Var x \rangle m1 = ExprEval e' < Var x \rangle m1'
\]
by (simp add: WHATWHERE.dH-equal-def, auto)

with substeval1 substeval2 have
\[
\forall (d',e') \in htcLoc \ l'.
\]
\[
d' \leq d \rightarrow \langle e' \rangle m2 = ExprEval e' \rangle m2'
\]
by auto

with dequalm2 m2-is-next m2'-is-next
show \( [x := e](m1) \sim d, htcLoc \ l', [x := e](m1') \)
by (simp add: WHATWHERE.dH-equal-def)
qed

— interpretation of the abstract type system using the above definitions for the side conditions

**interpretation** Type-System-example: Type-System ExprEval BMap DA lH
synAssignSC synWhileSC synIfSC
by (unfold-locales, auto,
metis ExprTypable-with-smallerd-implies-dH-indistinguishable
synAssignSC-def,
metis SubstClosure-implications synAssignSC-def,
simp add: synWhileSC-def,
metis ExprTypable-with-smallerd-implies-dH-indistinguishable
WHATWHERE-Secure-Programs.empH-implies-dHindistinguishable-eq-dindistinguishable,
simp add: synIfSC-def,
metis ExprTypable-with-smallerd-implies-dH-indistinguishable
WHATWHERE-Secure-Programs.empH-implies-dHindistinguishable-eq-dindistinguishable)
end

References
