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. Strong security from [2] formalizes noninterference for concurrent systems.

We present an Isabelle/HOL formalization of strong security for arbitrary security lattices ([2] uses a two-element security lattice). The formalization includes compositionality proofs for strong security and a soundness proof for a security type system that checks strong security 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.

Contents

1 Preliminary definitions 2
1.1 Type synonyms ........................................... 2

2 Strong security 4
2.1 Definition of strong security ............................ 4
2.2 Proof technique for compositionality results ........... 8
2.3 Proof of parallel compositionality ..................... 10

3 Example language and compositionality proofs 13
3.1 Example language with dynamic thread creation ...... 13
3.2 Proofs of atomic compositionality results ............ 15
3.3 Proofs of non-atomic compositionality results ........ 18
1 Preliminary definitions

1.1 Type synonyms

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

For better readability, we use the following type synonyms in our formalization:

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
\textbf{type-synonym} \('id, 'val, 'com\) \(PSConfig = \) 'com ProgramState \(\times\) ('id, 'val) State

— type for labels with a list of spawned threads
\textbf{type-synonym} \('com Label = 'com list\)

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

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

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

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

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

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

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

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

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

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

— type for local escape hatches
\textbf{type-synonym} \(('d, 'exp) lHatch = 'd \times 'exp \times nat\)
— type for sets of local escape hatches

**type-synonym**

\[
\text{IHatches} = ((\text{'d}, \text{'exp}) \text{lHatch}) \text{ set}
\]

end

## 2 Strong security

### 2.1 Definition of strong security

We define strong security such that it is parametric in a security lattice (\text{'d}). The definition of strong security by itself is language-independent, therefore the definition is parametric in a programming language (\text{'com}) in addition.

```plaintext
theory Strong-Security
imports Types
begin
locale Strong-Security = 
fixes SR :: (\text{'exp}, \text{'id}, \text{'val}, \text{'com}) \text{TSteps}
and DA :: (\text{'id}, \text{'d::order}) \text{DomainAssignment}
begin

— define when two states are indistinguishable for an observer on domain d

**definition**

\[
d\text{-equal} \; : \; \text{'d::order} \Rightarrow (\text{'id}, \text{'val}) \text{ State} \\
s \Rightarrow (\text{'id}, \text{'val}) \text{ State} \Rightarrow \text{ bool}
\]

where

\[
d\text{-equal} \; d \; m \; m' \equiv \forall x. \; ((\text{DA} \; x) \leq d \longrightarrow (m \; x) = (m' \; x))
\]

**abbreviation**

\[
d\text{-equal}' :: (\text{'id}, \text{'val}) \text{ State} \\
s \Rightarrow (\text{'d::order} \Rightarrow (\text{'id}, \text{'val}) \text{ State} \Rightarrow \text{ bool}
\]

where

\[
m \; =_d \; m' \equiv \; d\text{-equal} \; d \; m \; m'
\]

— transitivity of d-equality

**lemma**

\[
d\text{-equal-trans}: \\
\langle \; m \; =_d \; m'; \; m' \; =_d \; m'' \rangle \Rightarrow m \; =_d \; m''
\]

by (simp add: d-equal-def)

**abbreviation**

\[
\text{SRabbr} :: (\text{'exp}, \text{'id}, \text{'val}, \text{'com}) \text{TSteps-curry}
\]

(\text{\langle 1\langle,\rangle \rangle \rightarrow/} (1\langle,\rangle) \; [0,0,0] \; 81)

where

\[
\langle c,m \rangle \rightarrow \langle c',m' \rangle \equiv ((c,m),(c',m')) \in \text{SR}
\]

```
— predicate for strong d-bisimulation

definition Strong-d-Bisimulation :: 'd ⇒ 'com Bisimulation-type ⇒ bool
where
Strong-d-Bisimulation d R ≡
(sym R) ∧
(∀ (V,V') ∈ R. length V = length V') ∧
(∀ (V,V') ∈ R. ∀ i < length V. ∀ m1 m1' m2 W.
  ⟨V!i,m1⟩ → ⟨W,m2⟩ ∧ m1 =_d m1' →
  (∃ W' m2'. ⟨V!i,m1⟩ → ⟨W',m2'⟩ ∧ (W,W') ∈ R ∧ m2 =_d m2'))

— union of all strong d-bisimulations

definition USdB :: 'd ⇒ 'com Bisimulation-type
(≈. 65)
where
≈_d ≡ ∪ {r. (Strong-d-Bisimulation d r)}

abbreviation relatedbyUSdB :: 'com list ⇒ 'd ⇒ 'com list ⇒ bool
(infixr ≈. 65)
where V ≈_d V' ≡ (V,V') ∈ USdB d

— predicate to define when a program is strongly secure

definition Strongly-Secure :: 'com list ⇒ bool
where
Strongly-Secure V ≡ (∀ d. V ≈_d V)

— auxiliary lemma to obtain central strong d-Bisimulation property as Lemma in meta logic (allows instantiating all the variables manually if necessary)

lemma strongdB-aux: \[ \forall V \forall V' \forall m1 \forall m1' \forall m2 \forall W. \langle V!i,m1 \rangle \to \langle W,m2 \rangle \wedge m1 =_d m1' \to \exists W' m2'. \langle V!i,m1 \rangle \to \langle W',m2' \rangle \wedge (W,W') \in R \wedge m2 =_d m2' \]
by (simp add: Strong-d-Bisimulation-def, fastforce)

lemma trivialpair-in-USdB:
[[] ≈_d []]
by (simp add: USdB-def Strong-d-Bisimulation-def,
  rule-tac x={([[[],]])} in exI, simp add: sym-def)

lemma USdBsym: sym (≈_d)
by (simp add: USdB-def Strong-d-Bisimulation-def sym-def, auto)

lemma USdBseqlen:
V ≈_d V' \implies \text{length } V = \text{length } V'
by (simp add: USdB-def Strong-d-Bisimulation-def, auto)

lemma USdB-Strong-d-Bisimulation:
Strong-d-Bisimulation d (≈_d)
proof (simp add: Strong-d-Bisimulation-def, auto)
  show sym (≈_d) by (rule USdBsym)
next
fix V V'
show \( V \approx_d V' \implies \text{length} \ V = \text{length} \ V' \) by (rule USdBqeqlen, auto)

next
fix V V' m1 m1' m2 W i
assume inUSdB: \( V \approx_d V' \)
assume stepV: \( (V \! i, mI) \to (W, m2) \)
assume irange: \( i < \text{length} \ V \)
assume dequal: \( m1 =_d m1' \)

from inUSdB obtain R where someR:
  Strong-d-Bisimulation d R \( \land (V, V') \in R \)
  by (simp add: USdB-def, auto)

with strongdB-aux stepV irange dequal show
  \( \exists W', m2'. \ (V \! i, mI) \rightsquigarrow (W', m2') \land W \approx_d W' \land m2 =_d m2' \)
  by (simp add: USdB-def, fastforce)

qed

text{**lemma** USdBtrans; **trans** \( (\approx_d) \)
**proof** (simp add: trans-def, auto)
fix V V' V''
assume p1: \( V \approx_d V' \)
assume p2: \( V' \approx_d V'' \)
let \( ?R = \{(V, V''). \ \exists V'. \ V \approx_d V' \land V' \approx_d V'' \} \)
from p1 p2 have inRest: \( (V, V'') \in ?R \) by auto

have SdB-rest: Strong-d-Bisimulation d ?R
**proof** (simp add: Strong-d-Bisimulation-def sym-def, auto)
fix V V' V''
assume p1: \( V \approx_d V' \)
moreover
assume p2: \( V' \approx_d V'' \)
moreover
from p1 USdBsym have \( V' \approx_d V \)
  by (simp add: sym-def)
moreover
from p2 USdBsym have \( V'' \approx_d V' \)
  by (simp add: sym-def)
ultimately show
  \( \exists V'. \ V'' \approx_d V' \land V' \approx_d V \)
  by (rule_tac x=V'\ in extI, auto)

next
fix V V' V''
assume p1: \( V \approx_d V' \)
moreover 
assume p2: \( V' \approx_d V'' \) 
moreover 
from p1 USdB-eqlen[af \( V, V' \)] have \( \textnormal{length } V = \textnormal{length } V' \) 
by auto 
moreover 
from p2 USdB-eqlen[af \( V', V'' \)] have \( \textnormal{length } V' = \textnormal{length } V'' \) 
by auto 
ultimately show \( \textnormal{eqlen}: \textnormal{length } V = \textnormal{length } V'' \) by auto 

next 
fix \( V, V', V'' \) im1 m1 m1' W m2 
assume step: \( \langle V!i,m1 \rangle \rightarrow \langle W,m2 \rangle \) 
assume dequal: \( m1 =_d m1' \) 
assume p1: \( V \approx_d V' \) 
assume p2: \( V' \approx_d V'' \) 
assume irange: \( i < \textnormal{length } V \) 
from p1 USdB-eqlen[af \( V, V' \)] have \( \textnormal{leq}: \textnormal{length } V = \textnormal{length } V' \) 
by force 

have \( \textnormal{deq-same}: m1' =_d m1 \) by (simp add: \( d\)-equal-def) 

from irange step dequal p1 USdB-Strong-d-Bisimulation 
strongdB-aux[af \( d \approx_d i \ V V', m1 W m2 m1' \)] 
obtain \( W', m2' \) where p1concl: 
\( \langle V!i,m1 \rangle \rightarrow \langle W', m2' \rangle \land W \approx_d W' \land m2 =_d m2' \) 
by auto 

with \( \textnormal{deq-same \ leq \ USdB-Strong-d-Bisimulation} \) 
strongdB-aux[af \( d \approx_d i \ V V', V'', W' m1 W' m2' m1' \)] 
irange p2 dequal obtain \( W'', m2'' \) where p2concl: 
\( W' \approx_d W'' \land (\langle V''!i, m1 \rangle \rightarrow \langle W'', m2'' \rangle \land m2' =_d m2'' \) 
by auto 

from p1concl p2concl d-equal-trans have \( \textnormal{tt}'': m2 =_d m2'' \) 
by blast 

from p1concl p2concl have \( \langle W, W'' \rangle \in ?R \) 
by auto 

with p2concl \( \textnormal{tt}'' \) show \( \exists W'' m2''. \langle V''!i, m1 \rangle \rightarrow \langle W'', m2'' \rangle \land (\exists V'. W \approx_d V' \land V' \approx_d W'') \land m2 =_d m2'' \) 
by auto 

qed 

hence liftup: \( ?R \subseteq (\approx_d) \) 
by (simp add: USdB-def, auto) 

with \( \textnormal{inRest} \) show \( V \approx_d V'' \)
2.2 Proof technique for compositionality results

For proving compositionality results for strong security, we formalize the following “up-to technique” and prove it sound:

theory Up-To-Technique
imports Strong-Security
begin

context Strong-Security
begin

— define d-bisimulation 'up to' union of strong d-Bisimulations

definition d-Bisimulation-Up-To-USdB :: 
'('d ⇒ com Bisimulation-type ⇒ bool
where

d-Bisimulation-Up-To-USdB d R ≡ 
(sym R) ∧ (∀(V,V') ∈ R. length V = length V') ∧ 
(∀(V,V') ∈ R. ∀i < length V. ∀m1 m1' W m2. 
⟨V!i,m1⟩ → ⟨W,m2⟩ ∧ (m1 =_d m1') 
→ (∃W' m2'. ⟨V!i,m1'⟩ → ⟨W',m2'⟩ 
∧ (W,W') ∈ (R ∪ (≈_d)) ∧ (m2 =_d m2'))) 

lemma UpTo-aux: ∀ V V' m1 m1' m2 W i. [ d-Bisimulation-Up-To-USdB d R; 
i < length V; (V,V') ∈ R; ⟨V!i,m1⟩ → ⟨W,m2⟩; m1 =_d m1' ] 
⇒ (∃W' m2'. ⟨V!i,m1'⟩ → ⟨W',m2'⟩ 
∧ (W,W') ∈ (R ∪ (≈_d)) ∧ (m2 =_d m2'))
by (simp add: d-Bisimulation-Up-To-USdB-def, fastforce)

lemma RuUSdBeqlen:
[ d-Bisimulation-Up-To-USdB d R; 
(V,V') ∈ (R ∪ (≈_d)) ] 
⇒ length V = length V'
by (auto, simp add: d-Bisimulation-Up-To-USdB-def, auto, 
rule USdBeqlen, auto)

lemma Up-To-Technique:
assumes upToR: d-Bisimulation-Up-To-USdB d R
shows R ⊆ ≈_d
proof – 

by auto

qed
\[
\begin{align*}
def S & \equiv R \cup (\approx_d) \\
\text{from } S\text{-def have } & R \subseteq S \\
\text{by auto} \\
\text{moreover have } & S \subseteq (\approx_d) \\
\text{proof (simp add: USdB-def, auto, rule-tac x=S in exI, auto,} \\
\text{simp add: Strong-d-Bisimulation-def, auto)}
\end{align*}
\]

— show symmetry
\[
\text{show symS: sym } S
\]
\[
\text{proof - from upToR}
\]
\[
\text{have Rsym: sym } R
\]
\[
\text{by (simp add: d-Bisimulation-Up-To-USdB-def)}
\]
\[
\text{with USdBsym have Usym:}
\]
\[
\text{sym } (R \cup (\approx_d))
\]
\[
\text{by (metis sym-Un)}
\]
\[
\text{with S-def show } \text{thesis}
\]
\[
\text{by simp}
\]
qed

next
\[
\text{fix } V V'
\]
\[
\text{assume inS: } (V, V') \in S
\]
— show equal length (by definition)
\[
\text{from inS S-def upToR RuUSdBEqlen}
\]
\[
\text{show eqlen: length } V = \text{length } V'
\]
by simp

next
— show general bisimulation property
\[
\text{fix } V V' W m1 m1' m2 i
\]
\[
\text{assume inS: } (V, V') \in S
\]
\[
\text{assume irange: } i < \text{length } V
\]
\[
\text{assume stepV: } \langle V!i, m1 \rangle \rightarrow \langle W, m2 \rangle
\]
\[
\text{assume dequal: } m1 =_d m1'
\]
\[
\text{from inS show } \exists W' m2'. \langle V!i, m1' \rangle \rightarrow \langle W', m2' \rangle \land (W, W') \in S \land m2 =_d m2'
\]
\[
\text{proof (simp add: S-def, auto)}
\]
\[
\text{assume firstcase: } (V, V') \in R
\]
\[
\text{with upToR dequal irange stepV}
\]
\[
\text{UpTo-aux[of } d R i V V' m1 W m2 m1']
\]
\[
\text{show } \exists W' m2'. \langle V!i, m1' \rangle \rightarrow \langle W', m2' \rangle \land (W, W') \in S \land m2 =_d m2'
\]
by (auto simp add: S-def)

next
\[
\text{assume secondcase: } V \approx_d V'
\]
\[
\text{from USdB-Strong-d-Bisimulation upToR}
\]
\[
\text{secondcase dequal irange stepV}
\]
2.3 Proof of parallel compositionality

We prove that strong security is preserved under composition of strongly secure threads.

theory Parallel-Composition
imports Up-To-Technique
begin

context Strong-Security
begin

theorem parallel-composition:
assumes eqlen: length V = length V'
assumes partsrelated: ∀ i < length V. [V!i] ≈_d [V'!i]
shows V ≈_d V'
proof –
def R ≡ {(V,V'), length V = length V'
∧ (∀ i < length V. [V!i] ≈_d [V'!i])}
from eqlen partsrelated have inR: (V,V') ∈ R
by (simp add: R-def)

have d-Bisimulation-Up-To-USdB d R
proof (simp add: d-Bisimulation-Up-To-USdB-def, auto)
from USdBsym show sym R
by (simp add: R-def sym-def)
next
fix V V'
assume (V,V') ∈ R
with USdBeqlen show length V = length V'
by (simp add: R-def)
next
fix V V' i m1 m1' RS m2
assume inR: (V,V') ∈ R
assume irange: i < length V
assume **step**: \((V!i,m1) \rightarrow (RS,m2)\)
assume **dequal**: \(m1 =_d m1'\)

from **inR** have **Vassump**:
  \[\text{length } V = \text{length } V' \land (\forall i < \text{length } V. \ [V!i] \approx_d [V'^!i])\]
  by (simp add: **R-def**) with **step** **dequal** **USdB-Strong-d-Bisimulation irange**
  strongdB-**aux** \([d \approx d] \ [V'^!i] \ [V'^!i] \ m1 \ RS \ m2 \ m1'\]
  \[\text{show } \exists RS', m2'. \ (V'^!i,m1') \rightarrow (RS',m2') \land ((RS,RS') \in R \lor RS \approx_d RS') \land m2 =_d m2'\]
  by (simp, fastforce)

**qed**

**hence** \(R \subseteq (\approx_d)\) by (rule **Up-To-Technique**)

with **inR** show ?thesis by auto
**qed**

**lemma** **parallel-decomposition**:
assumes **related**: \(V \approx_d V'\)
shows \(\forall i < \text{length } V. \ [V!i] \approx_d [V'^!i]\)
**proof** –
  def \(R \equiv \{(C,C'). \exists i W W'. \ W \approx_d W' \land i < \text{length } W \land C = [W!i] \land C' = [W'^!i]\}\)
  with **related** have **inR**: \(\forall i < \text{length } V. \ ([V!i],[V'^!i]) \in R\) by auto
  **proof** (simp add: \(d\)-**Bisimulation-Up-To-USdB d R\)
  from **USdBsym** **USdBEqlen show** **sym** **R**
  by (simp add: **sym-def R-def**, metis)
next
  fix \(C C'\)
  assume \((C,C') \in R\)
  with **USdBEqlen show** **length** \(C = \text{length } C'\)
  by (simp add: **R-def**, auto)
next
  fix \(C C' i m1 m1' RS m2\)
  assume **inR**: \((C,C') \in R\)
  assume **irange**: \(i < \text{length } C\)
  assume **step**: \((C!i,m1) \rightarrow (RS,m2)\)
  assume **dequal**: \(m1 =_d m1'\)
from **inR** obtain \(j W W'\) where **Rassump**:
  \(W \approx_d W' \land j < \text{length } W \land C = [W'^!j] \land C' = [W'^!j]\)
by (simp add: R-def, auto)

with irange have i0: i = 0 by auto

from Rassump i0 strongdB-aux[of d ≈ j W W′
  m1 RS m2 m1′]
  USdB-Strong-d-Bisimulation step dequal
show ∃ RS′ m2′. (C′!i, m1′) → (RS′, m2′)
  ∧ ((RS, RS′) ∈ R ∨ RS ≈ d RS′) ∧ m2 = d m2′
  by auto
qed

hence R ⊆ (≈ d)
  by (rule Up-To-Technique)

with inR show ?thesis
  by auto
qed

lemma USdB-comp-head-tail:
  assumes relatedhead: [c] ≈_d [c′]
  assumes relatedtail: V ≈_d V′
  shows (c#V) ≈_d (c′#V′)
proof
  from relatedtail USdBeqlen have eqlen: length (c#V) = length (c′#V′)
    by force

  from relatedtail parallel-decomposition have singleV:
    ∀ i < length V. [V!i] ≈_d [V′!i]
    by force

  with relatedhead have intermediate:
    ∀ i < length (c#V). [(c#V)!i] ≈_d [[c′#V′]!i]
    by (auto, case-tac i, auto)

  with eqlen parallel-composition
  show ?thesis
    by blast
qed

lemma USdB-decomp-head-tail:
  assumes relatedlist: (c#V) ≈_d (c′#V′)
  shows |c| ≈_d |c′| ∧ V ≈_d V′
proof auto
  from relatedlist USdBeqlen[of c#V c′#V′]
  have eqlen: length V = length V′
    by auto
from relatedlist parallel-decomposition[of c#V c'#V' d]
have intermediate:
∀i < length (c#V). [(c#V)!i] ≈_d [(c'#V')!i]
  by auto
thus [c] ≈_d [c']
  by force

from intermediate eqlen show V ≈_d V'
proof (case-tac V)
  assume Vcase1: V = []
  with eqlen have V' = [] by auto
  with Vcase1 trivialpair-in-USdB show V ≈_d V'
    by auto
next
  fix c1 W
  assume Vcase2: V = c1#W
  hence Vlen: length V > 0 by auto

from intermediate have intermediate-aux:
  \(\forall i. i < \text{length } V\)
  \(\Rightarrow [V!i] \approx_d [V'!i]\)
  by force

  with parallel-composition[of V V'] eqlen
  show V ≈_d V'
    by blast

qed
qed

end

end

3 Example language and compositionality proofs

3.1 Example language with dynamic thread creation

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

theory MWLf
imports Types
begin
— SYNTAX

— Commands for the multi-threaded while language with fork (to instantiate 'com)

datatype ('exp, 'id) MWLfCom
  = Skip (skip)
  | Assign 'id 'exp
      (\::= (70,70) 70)

  | Seq ('exp, 'id) MWLfCom ('exp, 'id) MWLfCom
      (\::= [61,60] 60)

  | If-Else 'exp ('exp, 'id) MWLfCom ('exp, 'id) MWLfCom
      (if - then - else - fi [80,79,79] 70)

  | While-Do 'exp ('exp, 'id) MWLfCom
      (while - do - od [80,79] 70)

  | Fork ('exp, 'id) MWLfCom ('exp, 'id) MWLfCom) list
      (fork - - [70,70] 70)

— SEMANTICS

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

— steps semantics, set of deterministic steps from single threads to either single threads or thread pools

inductive-set
MWLfSteps-det :: ('exp, 'id, 'val, ('exp, 'id) MWLfCom) TSteps
and MWLfSteps-det' :: ('exp, 'id, 'val, ('exp, 'id) MWLfCom) TSteps-curry
     ((1(-/-)) → (1(-/-)) [0,0,0,0] 81)

where
  ⟨c1,m1⟩ → ⟨c2,m2⟩ ≡ ((c1,m1),(c2,m2)) ∈ MWLfSteps-det |
skip: ⟨skip,m⟩ → ([],m) |
assign: (E e m) = v ⇒ ⟨x := e,m⟩ → ([],m(x := v)) |
seq1: ⟨c1,m⟩ → ([],m') ⇒ ⟨c1;c2,m⟩ → ([c2,m']) |
seq2: ⟨c1,m⟩ → (c1'=V,m') ⇒ ⟨c1;c2,m⟩ → ((c1';c2)\#V,m') |
iftrue: BMap (E b m) = True ⇒
  ⟨if b then c1 else c2 fi,m⟩ → ([],m) |
iffalse: BMap (E b m) = False ⇒
  ⟨if b then c1 else c2 fi,m⟩ → ([c2],m) |
whiletrue: BMap (E b m) = True ⇒
  ⟨while b do c od,m⟩ → ([c(while b do c od)],m) |
whilefalse: BMap (E b m) = False ⇒
  ⟨while b do c od,m⟩ → ([],m) |
fork: \langle \text{fork } c \ V, m \rangle \rightarrow \langle c\#V, m \rangle

\text{inductive-cases} \quad \text{MWLfSteps-det-cases}:
\langle \text{skip}, m \rangle \rightarrow \langle W, m' \rangle
\langle x := e, m \rangle \rightarrow \langle W, m' \rangle
\langle c_1; c_2, m \rangle \rightarrow \langle W, m' \rangle
\langle \text{if } b \ \text{then } c_1 \ \text{else } c_2 \ fi, m \rangle \rightarrow \langle W, m' \rangle
\langle \text{while } b \ \text{do } c \ od, m \rangle \rightarrow \langle W, m' \rangle
\langle \text{fork } c \ V, m \rangle \rightarrow \langle W, m' \rangle

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

\text{inductive-set} \quad \text{MWLfSteps-ndet} :: \langle \exp, \id, \val, (\exp, \id) \text{MWLfCom} \rangle \text{TPSteps}
\text{and} \quad \text{MWLfSteps-ndet'} :: \langle \exp, \id, \val, (\exp, \id) \text{MWLfCom} \rangle \text{TPSteps-curry}
((1\langle\cdot,\cdot\rangle) \Rightarrow (1\langle\cdot,\cdot\rangle) [0,0,0,0] 81)
\text{where}
\langle V1, m1 \rangle \Rightarrow \langle V2, m2 \rangle \equiv ((V1, m1),(V2, m2)) \in \text{MWLfSteps-ndet} |
\langle ci, m \rangle \rightarrow \langle c, m' \rangle \Rightarrow \langle Vf \ @ [ci] \ @ Va, m \rangle \Rightarrow \langle Vf \ @ c \ @ Va, m' \rangle

\text{end}

\text{end}

3.2 Proofs of atomic compositionality results

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

\text{theory} \quad \text{Strongly-Secure-Skip-Assign}
\text{imports} \quad \text{MWLf Parallel-Composition}
\begin{verbatim}
begin

locale \text{Strongly-Secure-Programs} =
L : MWLf-semantics E BMap
+ SS: Strong-Security MWLfSteps-det DA
for E :: \langle \exp, \id, \val \rangle \text{Evalfunction}
and BMap :: \val \Rightarrow \text{bool}
and DA :: \langle \id, \d::\text{order} \rangle \text{DomainAssignment}
begin

abbreviation USdBname ::\d \Rightarrow \langle \exp, \id \rangle \text{MWLfCom Bisimulation-type} (\approx)
\text{where} \approx_d \equiv USdB \ d

end

\end{verbatim}

15
abbreviation relatedbyUSdB :: ('exp,'id) MWLfCom list ⇒ 'd
⇒ ('exp,'id) MWLfCom list ⇒ bool (infixr ≈ 65)
where V ≈_d V' ≡ (V,V') ∈ USdB d
— define when two expressions are indistinguishable with respect to a domain d

definition d-indistinguishable :: 'd::order ⇒ 'exp ⇒ 'exp ⇒ bool
where
d-indistinguishable d e1 e2 ≡
  ∀ m m'. ((m ≈_d m') −→ ((E e1 m) = (E e2 m')))

abbreviation d-indistinguishable' :: 'exp ⇒ 'd::order ⇒ 'exp ⇒ bool
  ( ( - ≈_- ) )
where
e1 ≡_d e2 ≡ d-indistinguishable d e1 e2
— symmetry of d-indistinguishable

lemma d-indistinguishable-sym: e1 ≡_d e1' ≡_d e2' ≡_d e2
by (simp add: d-indistinguishable-def d-equal-def, metis)

— transitivity of d-indistinguishable

lemma d-indistinguishable-trans:
  [| e ≡_d e', e' ≡_d e'' |] −→ e ≡_d e''
byp (simp add: d-indistinguishable-def d-equal-def, metis)

theorem Strongly-Secure-Skip:
  [skip] ≈_d [skip]
proof −
def R0 ≡ { (V::('exp,'id) MWLfCom list,V'::('exp,'id) MWLfCom list).
  V = [skip] ∧ V' = [skip] }

have uptoR0: d-Bisimulation-Up-To-USdB d R0
by (simp add: d-Bisimulation-Up-To-USdB-def, auto)

next
fix V V'
assume (V,V') ∈ R0
thus length V = length V'
bysimp add: R0-def

next
fix V V' i m1 m1' W m2
assume inR0: (V,V') ∈ R0
assume irange: i < length V
assume step: ⟨V!i,m1⟩ −→ ⟨W,m2⟩
assume dequal: m1 ≡_d m1'

from inR0 have Vassump:
  V = [skip] ∧ V' = [skip]
bysimp add: R0-def
with step range have step1:
  \( W = [] \land m2 = m1 \)
  by (simp, metis MWLf-semantics.MWLjSteps-det-cases(1))

from Vassamp range obtain m2' where step2:
  \( \langle V \uparrow ! i, m1 \rangle \rightarrow (\[\], m2') \land m2' = m1' \)
  by (simp, metis MWLfSteps-det.skip)

with step1 dequal trivialpair-in-USdB show \( \exists W' \) m2'.
  \( \langle V \uparrow ! i, m1 \rangle \rightarrow (W', m2') \land (W, W') \in R0 \lor W \approx_d W' \land m2 = m2' \)
  by auto

qed

hence \( R0 \subseteq (\approx_d) \)
  by (rule Up-To-Technique)

thus \(?thesis\)
  by (simp add: R0-def)

qed

theorem Strongly-Secure-Assign:
assumes d-indistinguishable-exp: \( e \equiv_{DA} x e' \)
sows \( [x := e] \approx_d [x := e'] \)
proof -
def R0 \( \equiv \{ (V, V'), \exists x e e'. V = [x := e] \land V' = [x := e'] \land e \equiv_{DA} x e' \} \)

from d-indistinguishable-exp have inR0: \( ([x := e], [x := e']) \in R0 \)
  by (simp add: R0-def)

have d-Bisimulation-Up-To-USdB d R0
proof (simp add: d-Bisimulation-Up-To-USdB-def, auto)
  from d-indistinguishable-sym show sym R0
    by (simp add: R0-def sym-def, fastforce)
next
  fix V V'
  assume (V, V') \( \in R0 \)
  thus length V = length V'
    by (simp add: R0-def, auto)
next
  fix V V' i m1 m1' W m2
  assume inR0: \( (V, V') \in R0 \)
  assume irange: \( i < \text{length } V \)
  assume step: \( \langle V \uparrow ! i, m1 \rangle \rightarrow (W, m2) \)
  assume dequal: \( m1 = m1' \)


from inR0 obtain x e e' where Vassump:
V = [x := e] \land V' = [x := e'] \land 
eq DA x e'
by (simp add: R0-def, auto)

with step irange obtain v where step1:
E e m1 = v \land W = [] \land m2 = m1(x := v)
by (auto, metis MWLf-semantics.MWLSteps-det-cases(2))

from Vassump irange obtain m2' v' where step2:
E e' m1' = v' \land \langle V'!i, m1' \rangle \rightarrow \langle [], m2' \rangle \land m2' = m1'(x := v')
by (auto, metis MWLfSteps-det-assign)

with Vassump dequal step step1
have dequalnext: m1(x := v) =_d m1'(x := v')
by (simp add: d-equal-def d-indistinguishable-def, auto)

with step1 step2 trivialpair-in-USdB show \exists W' m2'.
\langle V'!i, m1' \rangle \rightarrow \langle W', m2' \rangle \land (\langle W, m2' \rangle \in R0 \lor W \approx_d W')
\land m2 =_d m2'
by auto

qed

hence R0 \subseteq (\approx_d)
by (rule Up-To-Technique)

with inR0 show ?thesis
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 Strongly-Secure-Skip-Assign
begin

context Strongly-Secure-Programs


begin

**Theorem** Compositionality-Seq:

**Assumes** relatedpart1: \([c_1] \approx_d [c_1']\)

**Assumes** relatedpart2: \([c_2] \approx_d [c_2']\)

**Shows** \([c_1; c_2] \approx_d [c_1'; c_2']\)

**Proof**

\[\text{def } R_0 \equiv \{(S_1, S_2) \mid \exists c_1 c_1' c_2 c_2' W W'.\] 
\[S_1 = (c_1; c_2)\#W \land S_2 = (c_1'; c_2')\#W' \land\] 
\[c_1] \approx_d [c_1'] \land [c_2] \approx_d [c_2'] \land W \approx_d W'\] 

**from** relatedpart1 relatedpart2 trivialpair-in-USdB 
**have** inR0: \(([c_1; c_2], [c_1'; c_2']) \in R_0\)

**by** (simp add: R0-def)

**have** uptoR0: d-Bisimulation-Up-To-USdB d R0

**proof** (simp add: d-Bisimulation-Up-To-USdB-def, auto)

**from** USdBsym
**show** sym R0

**by** (simp add: sym-def R0-def, fastforce)

**next**

**fix** S1 S2 
**assume** inR0: \((S_1, S_2) \in R_0\)

**with** USdBlen 
**show** length S1 = length S2

**by** (auto simp add: R0-def)

**next**

**fix** S1 S2 RS m1 m2 m1' i 
**assume** inR0: \((S_1, S_2) \in R_0\)

**assume** irange: \(i < \text{length } S_1\)

**assume** S1step: \(\langle S_1!, m_1 \rangle \rightarrow \langle RS, m_2 \rangle\)

**assume** dequal: \(m_1 =_d m_1'\)

**from** inR0 obtain c1 c1' c2 c2' V V'

**where** R0def': \(S_1 = (c_1; c_2)\#V \land S_2 = (c_1'; c_2')\#V' \land\] 
\[[c_1] \approx_d [c_1'] \land [c_2] \approx_d [c_2'] \land V \approx_d V'\] 

**by** (simp add: R0-def, force)

**with** irange 
**have** case-distinction1:

\(i = 0 \lor (V \neq \[] \land i \neq 0)\)

**by** auto

**moreover**

**have** case1: \(i = 0 \implies\) 
\(\exists RS' m_2'. \langle S_2!i, m_1' \rangle \rightarrow \langle RS', m_2' \rangle \land\] 
\(((RS, RS') \in R_0 \lor RS \approx_d RS') \land m_2 =_d m_2'\)

**proof**

**assume** i0: \(i = 0\)

**— get the two different sub-cases:**

**with** R0def' S1step obtain c3 W 
**where** case-distinction:
\[ RS = [c2] \land (c1,m1) \rightarrow (\emptyset,m2) \]
\[ \lor RS = (c3;c2)#W \land (c1,m1) \rightarrow (c3#W,m2) \]
by (simp, metis MWLfSteps-det-cases(3))

moreover
— Case 1: first command terminates

\{  
assume RSassump: RS = [c2]  
assume StepAssump: (c1,m1) \rightarrow (\emptyset,m2)  
\}

from USdBBeqlen[of \emptyset] StepAssump R0def’
USdB-Strong-d-Bisimulation dequal
strongdB-aux[of d \approx_d i  
\[c1\] \[c1\] \[m1\] \[m2\] \[m1\] \[i0\]  
\]

\begin{align*}
& \text{obtain } W’ \text{ where } c1c1’\text{’reason:} \\
& \langle c1’,m1’ \rangle \rightarrow \langle W’,m2’ \rangle \land W’ = \emptyset \\
& \land \[c1\] \approx_d W’ \land m2 =_d m2’ \\
& \text{by } fastforce  
\end{align*}

with c1c1’\text{’reason have conclpart:} 
\langle c1’,c2’,m1’ \rangle \rightarrow (\{c2\},m2’) \land m2 =_d m2’  
by (simp add: MWLfSteps-det.seq1)

with RSassump R0def’ i0 have case1-concl: 
\exists RS’ \text{’m2’}. \langle S2\|i,m1’ \rangle \rightarrow (RS’,m2’) \land  
\langle RS,RS’ \rangle \in R0 \lor RS \approx_d RS’ \land m2 =_d m2’  
by (simp, rule-tac x=[c2’] in exI, auto)

moreover
— Case 2: first command does not terminate

\{  
assume RSassump: RS = (c3;c2)#W  
assume StepAssump: (c1,m1) \rightarrow (c3#W,m2)  
\}

from StepAssump R0def’ USdB-Strong-d-Bisimulation dequal
strongdB-aux[of d \approx_d i [c1] \[c1\] \[m1\] \[m1\] \[i0\]  
\]

\begin{align*}
& \text{obtain } V’’ \text{’m2’ where } c1c1’\text{’reason:} \\
& \langle c1’,m1’ \rangle \rightarrow \langle V’,m2’ \rangle \\
& \land (c3#W) \approx_d V’’ \land m2 =_d m2’ \\
& \text{by } fastforce  
\end{align*}

with USdBBeqlen[of c3#W V’’] obtain c3’ W’  
where V’’\text{’reason:} 
V’’ = c3’#W’ \land length W = length W’  
by (cases V’’, force, force)

with c1c1’\text{’reason have conclpart1:} 
\langle c1’,c2’,m1’ \rangle \rightarrow (\{c3’,c2’\},m2’) \land m2 =_d m2’  
by (simp add: MWLfSteps-det.seq2)
from V''reason c1c1'reason
USdB-decomp-head-tail[of c3 W]
USdB-Strong-d-Bisimulation
have c3aWinUSDB:
[c3] ≈_d [c3'] ∧ W ≈_d W'
by blast

with R0def' have conclpart2:
((c3;c2) #W,(c3';c2') #W') ∈ R0
by (auto simp add: R0-def)

with i0 RSassump R0def' V''reason conclpart1
have case2-concl:
∃RS' m2'. ⟨S2!i,m1⟩ → ⟨RS',m2⟩ ∧
((RS,RS') ∈ R0 ∨ RS ≈_d RS') ∧ m2 ≈_d m2'
by (rule-tac x=(c3';c2') #W' in exI, auto)

ultimately
show ∃RS' m2'. ⟨S2!i,m1⟩ → ⟨RS',m2⟩ ∧
((RS,RS') ∈ R0 ∨ RS ≈_d RS') ∧ m2 =_d m2'
by blast
qed

moreover
have case2: [ V ≠ []; i ≠ 0 ]
⇒ ⇒ ∃RS' m2'. ⟨S2!i,m1⟩ → ⟨RS',m2⟩ ∧
((RS,RS') ∈ R0 ∨ RS ≈_d RS') ∧ m2 ≈_d m2'
proof –
assume Vnonempt: V ≠ []
assume inot0: i ≠ 0

with Vnonempt irange R0def' have i1range:
(i−Suc 0) < length V
by simp

from inot0 R0def' have S1ieq: S1!i = V!(i−Suc 0)
by auto

from inot0 R0def' have S2!i = V'!(i−Suc 0)
by auto

with S1ieq R0def' S1step i1range dequal
USdB-Strong-d-Bisimulation
strongdB-aux[of d USdB d
i−Suc 0 V V' m1 RS m2 m1']
show ∃RS' m2'. ⟨S2!i,m1⟩ → ⟨RS',m2⟩ ∧
((RS,RS') ∈ R0 ∨ RS ≈_d RS') ∧ m2 =_d m2'
by force
qed
ultimately show \( \exists RS' m2'. \langle S2\!i,m1 \rangle \rightarrow \langle RS', m2' \rangle \wedge \\
((RS,RS') \in R0 \lor RS \approx_d RS') \wedge m2 =_d m2' \) 
by auto

qed

hence \( R0 \subseteq (\approx_d) \) 
by (rule Up-To-Technique)

with \( \text{in}R0 \) show \(?\text{thesis}\) 
by auto

qed

theorem Compositionality-Fork:

fixes \( V::(\exp', \id) \) \( \text{MWLJCom list} \) 
asssumes relatedmain: \([c]\approx_d [c']\) 
asssumes relatedthreads: \( V \approx_d V' \) 
shows \([\text{fork } c V] \approx_d [\text{fork } c' V']\) 

proof

def \( R0 \equiv \{(F1,F2). \exists c1 c1' W W'. \) 
\( F1 = [\text{fork } c1 W] \wedge F2 = [\text{fork } c1' W'] \wedge [c1] \approx_d [c1'] \wedge W \approx_d W' \) 
from relatedmain relatedthreads 
have \( \text{in}R0: ([\text{fork } c V], [\text{fork } c' V']) \in R0 \) 
by (simp add: \( R0\text{-def} \))

have \( \text{upto}R0: d\text{-Bisimulation-Up-To-USdB } d \) \( R0 \) 
proof (simp add: \( d\text{-Bisimulation-Up-To-USdB-def} \), auto) 
from \( USdBsym \) show \( \text{sym} R0 \) 
by (simp add: \( R0\text{-def sym-def} \), auto)

next

fix \( F1 F2 \) 
assume \( \text{in}R0: (F1,F2) \in R0 \) 
with \( R0\text{-def USdBeglen} \) show \( \text{length } F1 = \text{length } F2 \) 
by auto

next

fix \( F1 F2 c1V m1 m2 m1' i \) 
assume \( \text{in}R0: (F1,F2) \in R0 \) 
assume irange: \( i < \text{length } F1 \) 
assume \( F1\text{step}: \langle F1!i,m1 \rangle \rightarrow \langle c1V,m2 \rangle \) 
assume dequal: \( m1 =_d m1' \) 

from \( \text{in}R0 \) obtain \( c1 c1' V V' \) 
where \( R0\text{def}: F1 = [\text{fork } c1 V] \wedge F2 = [\text{fork } c1' V'] \wedge [c1] \approx_d [c1'] \wedge V \approx_d V' \) 
by (simp add: \( R0\text{-def} \), force)

from irange \( R0\text{def} \) \( F1\text{step} \) 
have \( \text{rew}: c1V = c1\#V \wedge m2 = m1 \)
by (simp, metis MWLf-semantics.MWLfSteps-det-cases(6))

from irange R0def' MWLfSteps-det.fork have F2step:
\langle F2!i,m1' \rangle \rightarrow \langle c1'#V',m1' \rangle
by force

from R0def' USdB-comp-head-tail have conclpart:
\langle (c1#V,c1'#V') \in R0 \lor (c1#V) \approx_d (c1'#V') \rangle
by auto
with irange rew inR0 F1step dequal R0def' F2step
show \exists c1V m2'. \langle F2!i,m1' \rangle \rightarrow \langle c1V',m2' \rangle \land
\langle (c1V,c1V') \in R0 \lor c1V \approx_d c1V' \rangle \land m2' =_d m2'
by fastforce
qed

hence R0 \subseteq (\approx_d)
by (rule Up-To-Technique)

with inR0 show \?thesis
by auto

qed

theorem Compositionality-If:
assumes dind-or-branchesrelated:
b \equiv_d b' \lor [c1] \equiv_d [c2] \lor [c1'] \equiv_d [c2']
assumes branch1related: [c1] \equiv_d [c1']
assumes branch2related: [c2] \equiv_d [c2']
shows [if b then c1 else c2 fi] \approx_d [if b' then c1' else c2' fi]
proof −
def R1 \equiv \{(I1,I2). \exists c1 c1' c2 c2' b b'.
I1 = [if b then c1 else c2 fi] \land I2 = [if b' then c1' else c2' fi] \land
[c1] \approx_d [c1'] \land [c2] \approx_d [c2'] \land b \equiv_d b'

def R2 \equiv \{(I1,I2). \exists c1 c1' c2 c2' b b'.
I1 = [if b then c1 else c2 fi] \land I2 = [if b' then c1' else c2' fi] \land
[c1] \approx_d [c1'] \land [c2] \approx_d [c2'] \land
([c1] \equiv_d [c2] \lor [c1'] \equiv_d [c2'])\}

def R0 \equiv R1 \cup R2

from dind-or-branchesrelated branch1related branch2related
have inR0: ([if b then c1 else c2 fi],[if b' then c1' else c2' fi]) \in R0
by (simp add: R0-def R1-def R2-def)

have uptoR0: d-Bisimulation-Up-To-USdB d R0
proof (simp add: d-Bisimulation-Up-To-USdB-def, auto)
from USdBsym d-indistinguishable-sym

23
have symR1: sym R1
  by (simp add: sym-def R1-def, fastforce)
from USdBsym
have symR2: sym R2
  by (simp add: sym-def R2-def, fastforce)

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

next
fix I1 I2
assume inR0: (I1, I2) ∈ R0
thus length I1 = length I2
  by (simp add: R0-def R1-def R2-def, auto)

next
fix I1 I2 RS m1 m1' m2 i
assume inR0: (I1, I2) ∈ R0
assume irange: i < length I1
assume I1step: ⟨I1!, i, m1⟩ → ⟨RS, m2⟩
assume dequal: m1 =_d m1'

have inR1case: (I1, I2) ∈ R1
  ⟹ ∃ RS' m2'. ⟨I2!, i, m1'⟩ → ⟨RS', m2'⟩ ∧ ```(RS, RS') ∈ R0 ∨ RS ≈_d RS' ∧ m2 =_d m2'```
proof
  assume inR1: (I1, I2) ∈ R1
  then obtain c1 c1' c2 c2' b b' where R1def':
    ```I1 = [if b then c1 else c2 fi]
    ∧ I2 = [if b' then c1' else c2' fi] ∧
    [c1] ≈_d [c1'] ∧ [c2] ≈_d [c2'] ∧ b ≡_d b'```
  by (simp add: R1-def, force)
moreover
  — get the two different cases True and False from semantics:
  from irange R1def' I1step have case-distinction:
    RS = [c1] ∧ BMap (E b m1) = True ∨
    RS = [c2] ∧ BMap (E b m1) = False
  by (simp, metis MWLf-semantics.MWLfSteps-det-cases(4))
moreover
  — Case 1: b evaluates to True
  { assume bevalT: BMap (E b m1) = True
    assume RSassump: RS = [c1]
    from irange bevalT I1step R1def' RSassump have memeq:
      m2 = m1
    by (simp, metis MWLf-semantics.MWLfSteps-det-cases(4))
  }

from bevalT R1def' dequal have b'evalT:
  BMap (E b' m1') = True
  by (simp add: d-indistinguishable-def)
hence \texttt{I2step-case1}:
\((\text{if } b' \text{ then } c1' \text{ else } c2' \text{ fi}, m1') \rightarrow \langle ]c1', m1' \rangle \)
by (simp add: MWLfSteps-det iftrue)

with \texttt{irange dequal RSassump memeq R1def'}
have case1-concl:
\(\exists RS' m2'. \langle \text{I2}!i, m1' \rangle \rightarrow \langle RS', m2' \rangle \wedge\)
\(((RS, RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'\)
by auto
}
moreover
— Case 2: \(b\) evaluates to False
\{
assume bevalF: \(\text{BMap} (E b \ m1) = \text{False}\)
assume RSassump: \(RS = [c2]\)
from \texttt{irange bevalF I1step R1def'} \texttt{RSassump} have memeq:
\(m1 = m2\)
by (simp, metis MWLf-semantics.MWLfSteps-det-cases(4))

from bevalF \texttt{R1def'} dequal have b'evalF:
\(\text{BMap} (E b' \ m1') = \text{False}\)
by (simp add: d-indistinguishable-def)

hence \texttt{I2step-case1}:
\((\text{if } b' \text{ then } c1' \text{ else } c2' \text{ fi}, m1') \rightarrow \langle ]c2', m1' \rangle \)
by (simp add: MWLfSteps-det.iffalse)

with \texttt{irange dequal RSassump memeq R1def'}
have case1-concl:
\(\exists RS' m2'. \langle \text{I2}!i, m1' \rangle \rightarrow \langle RS', m2' \rangle \wedge\)
\(((RS, RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'\)
by auto
}
ultimately show
\(\exists RS' m2'. \langle \text{I2}!i, m1' \rangle \rightarrow \langle RS', m2' \rangle \wedge\)
\(((RS, RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'\)
by auto
qed

have \texttt{inR2case}: \((I1, I2) \in R2\)
\(\Rightarrow \exists RS' m2'. \langle \text{I2}!i, m1' \rangle \rightarrow \langle RS', m2' \rangle \wedge\)
\(((RS, RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'\)
proof -
assume \texttt{inR2}: \((I1, I2) \in R2\)
then obtain \(c1 c1' c2 c2' b b'\) where \texttt{R2def'}:
\(I1 = [\text{if } b \text{ then } c1 \text{ else } c2 \text{ fi}]\)
\(\wedge I2 = [\text{if } b' \text{ then } c1' \text{ else } c2' \text{ fi}] \wedge\)
\[ c_1 \approx_d [c_1] \land [c_2] \approx_d [c_2] \land \]
\[ ([c_1] \approx_d [c_2] \lor [c_1] \approx_d [c_2]) \]
\[ \text{by (simp add: R2-def, force)} \]

moreover

--- get the two different cases for the result from semantics:

\[ \text{from } \text{range } R2def' \text{ I1step}\]
\[ \text{have case-distinction-left:} \]
\[ (RS = [c_1] \lor RS = [c_2]) \land m_2 = m_1 \]
\[ \text{by (simp, metis MWLf-semantics.MWLfSteps-det-cases(4))} \]

moreover

\[ \text{from } \text{range } R2def' \text{ dequal obtain } RS' \text{ where I2step:} \]
\[ \langle I2i, m_1 \rangle \rightarrow (RS', m_1) \]
\[ \land (RS' = [c_1] \lor RS' = [c_2]) \land m_1 =_d m_1' \]
\[ \text{by (simp, metis MWLfSteps-det-iffalse MWLfSteps-det-iftrue)} \]

moreover

\[ \text{from } \text{UsdBtrans have } [[] \approx_d [c_1]; [c_2] \approx_d [c_2]] \]
\[ \Rightarrow [c_1] \approx_d [c_2] \]
\[ \text{by (unfold trans-def, blast)} \]

moreover

\[ \text{from } \text{UsdBtrans have } [[] \approx_d [c_1]; [c_1] \approx_d [c_2]] \]
\[ \Rightarrow [c_1] \approx_d [c_2] \]
\[ \text{by (unfold trans-def, blast)} \]

moreover

\[ \text{from } \text{UsdBsym have } [c_1] \approx_d [c_2] \Rightarrow [c_2] \approx_d [c_1] \]
\[ \text{by (simp add: sym-def)} \]

moreover

\[ \text{from } \text{UsdBtrans have } [[] \approx_d [c_2]; [c_1] \approx_d [c_1]] \]
\[ \Rightarrow [c_2] \approx_d [c_1] \]
\[ \text{by (unfold trans-def, blast)} \]

moreover

\[ \text{from } \text{UsdBsym have } [c_1] \approx_d [c_2] \Rightarrow [c_2] \approx_d [c_1] \]
\[ \text{by (simp add: sym-def)} \]

moreover

\[ \text{from } \text{UsdBtrans have } [[] \approx_d [c_2]; [c_2] \approx_d [c_1]] \]
\[ \Rightarrow [c_2] \approx_d [c_1] \]
\[ \text{by (unfold trans-def, blast)} \]

ultimately show

\[ \exists RS'. m_2'. \langle I2i, m_1 \rangle \rightarrow (RS', m_2) \land \]
\[ ((RS, RS') \in R0 \lor RS \approx_d RS') \land m_2 =_d m_2' \]
\[ \text{by auto} \]

qed

\[ \text{from } \text{inR0 inR1case inR2case show} \]
\[ \exists RS'. m_2'. \langle I2i, m_1 \rangle \rightarrow (RS', m_2) \land \]
\[ ((RS, RS') \in R0 \lor RS \approx_d RS') \land m_2 =_d m_2' \]
\[ \text{by (auto simp add: R0-def)} \]

qed

hence \( R0 \subseteq (\approx_d) \)
\[ \text{by (rule Up-To-Technique)} \]
with inR₀ show thesis
  by auto

qed

**Theorem** Compositionality-While:
assumes dind: \( b \equiv_d b' \)
assumes bodyrelated: \([c] \equiv_d [c']\)
shows \([\text{while } b \text{ do } c \text{ od}] \approx_d [\text{while } b' \text{ do } c' \text{ od}]\)

**Proof**

- def \( R₁ \equiv \{(S₁, S₂). \exists c₁ c₁' c₂ c₂' b b' W W'. \]
  \( S₁ = (c₁; (\text{while } b \text{ do } c₂ \text{ od}))\#W \wedge \]
  \( S₂ = (c₁'; (\text{while } b' \text{ do } c₂' \text{ od}))\#W' \wedge \]
  \([c₁] \approx_d [c₁'] \wedge [c₂] \approx_d [c₂'] \wedge W \approx_d W' \wedge b \equiv_d b'\)

- def \( R₂ \equiv \{(W₁, W₂). \exists c₁ c₁' b b'. \]
  \( W₁ = [\text{while } b \text{ do } c₁ \text{ od}] \wedge W₂ = [\text{while } b' \text{ do } c₁' \text{ od}] \wedge \]
  \([c₁] \approx_d [c₁'] \wedge b \equiv_d b'\)

- def \( R₀ \equiv R₁ \cup R₂\)

from dind bodyrelated
have inR₀: \([\text{while } b \text{ do } c \text{ od}], [\text{while } b' \text{ do } c' \text{ od}]\) \(\in R₀\)
  by (simp add: R₀-def R₁-def R₂-def)

have uptoR₀: \(d\)-Bisimulation-Up-To-USdB \( d \in R₀\)
proof (simp add: d-Bisimulation-Up-To-USdB-def, auto)
  from USdBsym d-indistinguishable-sym have symR₁: \(\text{sym } R₁\)
  by (simp add: sym-def R₁-def, fastforce)
  from USdBsym d-indistinguishable-sym have symR₂: \(\text{sym } R₂\)
  by (simp add: sym-def R₂-def, fastforce)
  from symR₁ symR₂ have symR₀
  by (simp add: sym-def R₀-def)
next
fix W₁ W₂
assume inR₀: \((W₁, W₂) \in R₀\)
with USdBEqlen show length W₁ = length W₂
  by (simp add: R₀-def R₁-def R₂-def, force)
next
fix W₁ W₂ i m₁ m₁' RS m₂
assume inR₀: \((W₁, W₂) \in R₀\)
assume irange: \(i < \text{length } W₁\)
assume W₁step: \(\langle W₁!i, m₁ \rangle \rightarrow \langle RS, m₂ \rangle\)
assume dequal: \(m₁ =_d m₁'\)

from inR₀ show \(\exists RS' m₂'. \langle W₂!i, m₁' \rangle \rightarrow \langle RS', m₂' \rangle \wedge\)
  \((RS, RS') \in R₀ \wedge RS \approx_d RS' \wedge m₂ =_d m₂'\)
proof (simp add: R₀-def, auto)
assume \( \text{in } \mathcal{R}_1 : (W_1,W_2) \in \mathcal{R}_1 \)

then obtain \( c_1 c_1' c_2 c_2' b b' V V' \)

where \( \mathcal{R}_1 \text{def}': W_1 = (c_1;\{\text{while } b \text{ do } c_2 \text{ od})\} \# V \wedge W_2 = (c_1';\{\text{while } b' \text{ do } c_2' \text{ od})\} \# V' \wedge [c_1] \approx_d [c_1] \wedge [c_2] \approx_d [c_2'] \wedge V \approx_d V' \wedge b \equiv_d b' \)

by \((\text{simp add: } \mathcal{R}_1\text{-def}, \text{force})\)

with \( \text{irange have case-distinction1: } i = 0 \lor \)

\((V \neq [] \wedge i \neq 0)\)

by \(\text{auto}\)

moreover have \(\text{case1: } i = 0 \implies \)

\(\exists RS' m_2'. (W_2!i,m_1') \rightarrow (RS',m_2') \wedge ((RS,RS') \in \mathcal{R}_1 \lor (RS,RS') \in \mathcal{R}_2 \lor RS \approx_d RS') \wedge m_2 = d m_2'\)

proof —

assume \(i0: i = 0\)
— get the two different sub-cases:

with \(\mathcal{R}_1\text{def}' \text{W1step obtain } c_3 W\text{ where case-distinction:}\)

\(RS = [\text{while } b \text{ do } c_2 \text{ od}] \wedge (c_1,m_1) \rightarrow (][,m_2) \lor RS = (c_3;\{\text{while } b \text{ do } c_2 \text{ od})\} \# W \wedge (c_1,m_1) \rightarrow (c_3 \# W,m_2) \wedge m_2 = d m_2'\)

by \((\text{simp, metis MWLSteps-det-cases}(3))\)

moreover — Case 1: first command terminates

\{
assume \(RS\text{assump}: RS = [\text{while } b \text{ do } c_2 \text{ od}]\)
assume \(Step\text{assump}: (c_1,m_1) \rightarrow (][,m_2)\)

from \(\text{USdB}\text{-Beglen}[][] \text{StepAssump } \mathcal{R}_1\text{def}'\)
\(\text{USdB-Strong-d-Bisimulation dequal strongdB-aux}[d \equiv_d i]
\[c_1] [c_1'] m_1 [ ][m_2 m_1'] i0\)

obtain \(W' m_2'\text{where } c_1c_1'\text{reason:}\)

\(\langle c_1',m_1' \rangle \rightarrow (W',m_2') \wedge W' = [] \wedge [][d W' \wedge m_2 =_d m_2'\]

by \(\text{fastforce}\)

with \(c_1c_1'\text{reason have conclpart1:}\)

\(\langle c_1';\{\text{while } b' \text{ do } c_2' \text{ od})\},m_1' \rangle \rightarrow (][w hile b' \text{ do } c_2' \text{ od}],m_2') \wedge m_2 =_d m_2'\)

by \((\text{simp add: MWLSteps-det.seq1})\)

from \(\mathcal{R}_1\text{def}' \text{have conclpart2:}\)

\(([][\text{while } b \text{ do } c_2 \text{ od}],[][\text{while } b' \text{ do } c_2' \text{ od})] \in \mathcal{R}_2\)

by \((\text{simp add: R2-def})\)

with \(conclpart1 RS\text{assump }i0 \mathcal{R}_1\text{def}'\)

have \(\text{case1-concl:}\)

28
∃RS', m2'. ⟨W2!i, m1'⟩ → ⟨RS', m2'⟩ ∧
((RS, RS') ∈ R1 ∨ (RS, RS') ∈ R2 ∨ RS ≈ d RS')
∧ m2 =_d m2'
by auto
}

moreover
— Case 2: first command does not terminate
{
assume RSassump: RS = (c3;(while b do c2 od))#W
assume StepAssump: ⟨c1, m1⟩ → ⟨c3#W,m2⟩

from StepAssump R1def' USdB-Strong-d-Bisimulation dequal

[1] [c1] [c1'] m1 c3#W m2 m1' i0
obtain V'' m2' where c1c1'reason:
⟨c1', m1⟩ → ⟨V'', m2'⟩
∧ (c3#W) ≈_d V'' ∧ m2 =_d m2'
by fastforce

with USdBBegin[of c3#W V''] obtain c3' W'
where V''reason: V'' = c3'#W'
by (cases V'', force, force)

with c1c1'reason have conclpart1:
⟨c1';(while b' do c2' od), m1'⟩ →
⟨(c3';(while b' do c2' od))#W', m2'⟩
∧ m2 =_d m2'
by (simp add: MWLjSteps-det.seq2)

from V''reason

(c1c1'reason USdB-decomp-head-tail[of c3 W]
USdB-Strong-d-Bisimulation

have c3aWinUSDB:
[C3] ≈_d [c3'] ∧ W ≈_d W'
by blast

with R1def' have conclpart2:
((c3';(while b do c2 od))#W,
(c3';(while b' do c2' od))#W') ∈ R1
by (simp add: R1-def)

with i0 RSassump R1def' V''reason conclpart1
have case2-concl1:
∃RS', m2'. ⟨W2!i,m1'⟩ → ⟨RS', m2'⟩ ∧
((RS, RS') ∈ R1 ∨ (RS, RS') ∈ R2 ∨ RS ≈_d RS')
∧ m2 =_d m2'
by auto

}
ultimately
∃RS′m2′. (W2!i,m1] → (RS′,m2) ∧
((RS,RS′) ∈ R1 ∨ (RS,RS′) ∈ R2 ∨ RS ≈ d RS′)
∧ m2 = d m2′
by blast
qed

moreover
have case2: [ V ≠ []; i ≠ 0 ]
  ⇒ ∃RS′m2′. (W2!i,m1] → (RS′,m2) ∧
  ((RS,RS′) ∈ R1 ∨ (RS,RS′) ∈ R2 ∨ RS ≈ d RS′)
  ∧ m2 = d m2′
proof –
  assume Vnonempt: V ≠ []
  assume inot0: i ≠ 0
  with Vnonempt irange R1def′ have irange:
  (i−Suc 0) < length V
  by simp
  from inot0 R1def′ have W1ieq: W1!i = V!(i−Suc 0)
  by auto
  from inot0 R1def′ have W2!i = V′!(i−Suc 0)
  by auto

  with W1ieq R1def′ W1step irange dequal
  USdB-Strong-d-Bisimulation
  strongB-aux[of d USdB d]
i−Suc 0 V V′ m1 RS m2 m1′

  show ∃RS′m2′. (W2!i,m1] → (RS′,m2) ∧
  ((RS,RS′) ∈ R1 ∨ (RS,RS′) ∈ R2 ∨ RS ≈ d RS′)
  ∧ m2 = d m2′
  by force
qed
ultimately show ∃RS′m2′. (W2!i,m1] → (RS′,m2) ∧
  ((RS,RS′) ∈ R1 ∨ (RS,RS′) ∈ R2 ∨ RS ≈ d RS′)
  ∧ m2 = d m2′
  by auto

next
  assume inR2: (W1,W2) ∈ R2

  then obtain c1 c1′ b b′ where R2def′:
    W1 = [while b do c1 od] ∧ W2 = [while b' do c1' od] ∧
    [c1] ≈ d [c1'] ∧ b ≡ d b'
  by (auto simp add: R2-def)
  — get the two different cases:

  moreover
  from irange R2def′ W1step have case-distinction:
    RS = [c1; while b do c1 od] ∧ BMap (E b m1) = True ∨
    RS = [] ∧ BMap (E b m1) = False
by (simp, metis MWLf-semantics.MWLjSteps-det-cases(5))
moreover
  — Case 1: b evaluates to True
  
  {  
    assume bevalT: BMap (E b m1)
    assume RSassump: RS = [c1:(while b do c1 od)]
    from irange bevalT W1step R2def' RSassump have memeq:
    m2 = m1
    by (simp, metis MWLf-semantics.MWLjSteps-det-cases(5))
  }

from bevalT R2def' dequal b'evalT: BMap (E b' m1')
by (simp add: d-indistinguishable-def)

hence W2step-case1:
  
  ⟨while b' do c1' od,m1'⟩ → ⟨⟨c1';(while b' do c1' od)],m1'⟩
  by (simp add: MWLfSteps-det.whiletrue)

from trivialpair-in-USdB R2def' have inWR2:  
  ([c1:(while b do c1 od)],
   [c1'((while b' do c1' od))]) ∈ R1
by (auto simp add: R1-def)

with irange dequal RSassump memeq W2step-case1 R2def'
have case1-concl:  
  ∃RS' m2'. ⟨W2!i,m1'⟩ → ⟨RS',m2'⟩ ∧
  ((RS,RS') ∈ R1 ∨ (RS,RS') ∈ R2 ∨ RS ≈ d RS')
  ∧ m2 = d m2'
by auto
}

moreover
  — Case 2: b evaluates to False
  
  {  
    assume bevalF: BMap (E b m1) = False
    assume RSassump: RS = []
    from irange bevalF W1step R2def' RSassump have memeq:
    m2 = m1
    by (simp, metis MWLf-semantics.MWLjSteps-det-cases(5))
  }

from bevalF R2def' dequal have b'evalF:
  BMap (E b' m1') = False
by (simp add: d-indistinguishable-def)

hence W2step-case2:
  ⟨while b' do c1' od,m1'⟩ → ⟨[],m1'⟩
by (simp add: MWLfSteps-det.whilefalse)

with trivialpair-in-USdB irange dequal RSassump
memeq R2def'
have case1-concl:
∃RS, m2’. ⟨W2!i.m1⟩ → ⟨RS’,m2’⟩ ∧
((RS,RS’) ∈ R1 ∨ (RS,RS’) ∈ R2 ∨ RS ≈ RS’)
∧ m2 = m2’
by force
}
ultimately
show ∃RS, m2’. ⟨W2!i.m1⟩ → ⟨RS’,m2’⟩ ∧
((RS,RS’) ∈ R1 ∨ (RS,RS’) ∈ R2 ∨ RS ≈ RS’)
∧ m2 = m2’
by auto
qed
qed

hence R0 ⊆ (≈_d)
by (rule Up-To-Technique)

with inR0 show ?thesis
by auto
qed
end
end

4 Security type system

4.1 Abstract security type system with soundness proof

We formalize an abstract version of the type system in [2] using locales [1]. Our formalization of the type system is abstract in the sense that the rules specify abstract semantic side conditions on the expressions within a command that satisfy for proving the soundness of the rules. That is, it can be instantiated with different syntactic approximations for these semantic side conditions in order to achieve a type system for a concrete language for Boolean and arithmetic expressions. Obtaining a soundness proof for such a concrete type system then boils down to proving that the concrete type system interprets the abstract type system.

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

theory Type-System
imports Language-Composition
begin
locale Type-System =
SSP : Strongly-Secure-Programs E BMap DA
for E :: ('exp, 'id, 'val) Evalfunction
and BMap :: 'val ⇒ bool
and DA :: ('id, 'd::order) DomainAssignment
+
fixes
AssignSideCondition :: 'id ⇒ 'exp ⇒ bool
and WhileSideCondition :: 'exp ⇒ bool
and IfSideCondition :: 'exp ⇒ ('exp, 'id) MWFCom ⇒ bool
assumes semAssignSC: AssignSideCondition x e ⇒ e ≡ DA x e
and semWhileSC: WhileSideCondition e ⇒ ∀ d. e ≡_d e
and semIfSC: IfSideCondition e c1 c2 ⇒ ∀ d. e ≡_d e ∨ [c1] ≈_d [c2]
begin

— Security typing rules for the language commands

inductive
ComSecTyping :: ('exp, 'id) MWFCom ⇒ bool
(\vdash_c -)
and ComSecTypingL :: ('exp,'id) MWFCom list ⇒ bool
(\vdash_y -)
where
skip: \vdash_y skip |
Assign: [ AssignSideCondition x e ] ⇒ \vdash_c x := e |
Fork: [ \vdash_c c; \vdash_y V ] ⇒ \vdash_c fork c V |
Seq: [ \vdash_c c1; \vdash_c c2 ] ⇒ \vdash_c c1;c2 |
While: [ \vdash_c c; WhileSideCondition b ]
⇒ \vdash_c while b do c od |
If: [ \vdash_c c1; \vdash_x c2; IfSideCondition b c1 c2 ]
⇒ \vdash_c if b then c1 else c2 fi |
Parallel: [ ∀ i < length V. \vdash_y V!i ] ⇒ \vdash_y V

inductive-cases parallel-cases:
\vdash_y V

— soundness proof of abstract type system

theorem ComSecTyping-single-is-sound:
\vdash_c c e ⇒ Strongly-Secure [c]
by (induct rule: ComSecTyping-ComSecTypingL.inducts(1))
[of - - Strongly-Secure],
auto simp add: Strongly-Secure-def, metis Strongly-Secure-Skip, metis Strongly-Secure-Assign semAssignSC, metis Compositionality-Fork, metis Compositionality-Seq, metis Compositionality-While semWhileSC, metis Compositionality-If semIfSC, metis parallel-composition)
\textbf{theorem} \textit{ComSecTyping-list-is-sound}:
\[ \vdash V \implies \text{Strongly-Secure } V \]
\textbf{by} (metis ComSecTyping-single-is-sound Strongly-Secure-def parallel-composition parallel-cases)

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.

\begin{verbatim}
theory Expr
imports Types
begin

— type parameters:
— \texttt{'val}: numbers, boolean constants....
— \texttt{'id}: identifier names

type-synonym (\texttt{'val}) operation = \texttt{'val list} \Rightarrow \texttt{'val}

datatype (\texttt{dead 'id, dead 'val}) Expr =
  Const \texttt{'val} |
  Var \texttt{'id} |
  Op \texttt{'val operation} ((\texttt{'id, 'val}) Expr list)

— defining a simple recursive evaluation function on this datatype
primrec ExprEval :: ((\texttt{'id, 'val}) Expr, \texttt{'id, 'val}) Evalfunction
and ExprEvalL :: ((\texttt{'id, 'val}) Expr list \Rightarrow \texttt{(id, 'val)} State \Rightarrow \texttt{val list}
where
  ExprEval (Const v) m = v |
  ExprEval (Var x) m = (m x) |
  ExprEval (Op f arglist) \( m = (f \ (ExprEvalL \ f arglist m)) \) |

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.

theory Domain-example
imports Expr
begin
— When interpreting, we have to instantiate the type for domains. As an example, we take a type containing ‘low’ and ‘high’ as domains.
datatype Dom = low | high
instantiation Dom :: order
begin
definition less-eq-Dom-def: d1 ≤ d2 = (if d1 = d2 then True
   else (if d1 = low then True else False))
definition less-Dom-def: d1 < d2 = (if d1 = d2 then False
   else (if d1 = low then True else False))
instance proof
fix x y z :: Dom
   show (x < y) = (x ≤ y ∧ ¬ y ≤ x)
      unfolding less-eq-Dom-def less-Dom-def by auto
   show x ≤ x unfolding less-eq-Dom-def by auto
   show [x ≤ y; y ≤ z] ⇒ x ≤ z
      unfolding less-eq-Dom-def by ((split split-if-asm)+, auto)
   show [x ≤ y; y ≤ x] ⇒ x = y
      unfolding less-eq-Dom-def by ((split split-if-asm)+,
                                    auto, (split split-if-asm)+, auto)
qed
end
end

theory Type-System-example
imports Type-System Expr Domain-example
begin
— When interpreting, we have to instantiate the type for domains.
— As an example, we take a type containing ‘low’ and ‘high’ as domains.
consts DA :: ('id,Dom) DomainAssignment
consts BMap :: 'val ⇒ bool
abbreviation \( d\text{-indistinguishable} \) :: \((\text{'id'},{\text{val}})~\text{Expr} \Rightarrow \text{Dom} \)

\( \Rightarrow (\text{'id',}{\text{val}})\text{Expr} \Rightarrow \text{bool} \)

where
e1 \equiv_d e2

\equiv \text{Strongly-Secure-Programs.d-indistinguishableExprEval DA d e1 e2}

abbreviation relatedby\(USdB\) :: \((\text{'id',}{\text{val}})\text{Expr}, \text{'id'}\) \(\text{MWLjCom list} \Rightarrow \text{Dom} \Rightarrow (\text{'id',}{\text{val}})\text{Expr}, \text{'id'}\) \(\text{MWLjCom list} \Rightarrow \text{bool (infixr \(\approx\) \(\approx\) \(65\)}) \)

where \( V \approx_d V' \equiv (V, V') \in \text{Strong-Security.UsdB} \)

\((\text{MWLj-semantics.MWLjSteps-det ExprEval BMap}) \text{DA d} \)

Security typing rules for expressions - will be part of a side condition

inductive

\(\text{ExprSecTyping} :: (\text{'id',}{\text{val}})\text{Expr} \Rightarrow \text{Dom set} \Rightarrow \text{bool} \)

\((\vdash e : -)\)

where

\(\text{Constrs: } \vdash e \{\text{Const v}\} : \{d\} | \)

\(\text{Vars: } \vdash e \{\text{Var x}\} : \{\text{DA x}\} | \)

\(\text{Ops: } \forall i < \text{length arglist. } \vdash (\arglist !_i) : (\text{dl}!i) \)

\(\Rightarrow (\vdash (\text{Op f arglist}) : (\bigcup \{\text{d. } (\exists i < \text{length arglist. } \text{d} = (\text{dl}!i))\}) \)

definition \(\text{synAssignSC} :: \text{'id'} \Rightarrow (\text{'id',}{\text{val}})\text{Expr} \Rightarrow \text{bool} \)

where

\(\text{synAssignSC} \ x \ e \equiv \exists D. (\vdash e : D \land (\forall d \in D. (d \leq DA x))) \)

definition \(\text{synWhileSC} :: (\text{'id',}{\text{val}})\text{Expr} \Rightarrow \text{bool} \)

where

\(\text{synWhileSC} \ e \ c1 \ c2 \equiv \)

\(\forall d. (\neg (e \equiv_d e) \Rightarrow [c1] \equiv_d [c2]) \)

lemma \(\text{ExprTypable-with-smallerD-implies-d-indistinguishable:} \)

\([(\vdash e : D'; \forall d' \in D'. d' \leq d) \Rightarrow e \equiv_d e \)

proof (induct rule: ExprSecTyping.induct,

\(\text{simp-all add: } \text{Strongly-Secure-Programs.d-indistinguishable-def} \)

\(\text{Strong-Security.d-equal-def}, \text{auto}) \)

fix \(\text{dl}\) and \(\arglist :((\text{'id',}{\text{val}})\text{Expr})\text{list}\) and \(f::{\text{val}}\text{list} \Rightarrow \text{val} \)

and \(\text{m1}::{\text{'id',}{\text{val}}}\text{State}\) and \(\text{m2}::{\text{'id',}{\text{val}}}\text{State} \)

assume main: \(\forall i < \text{length arglist. } \vdash \arglist!i : \text{dl}!i \land \)

\((\forall d' \in (\text{dl}!i). d' \leq d) \Rightarrow (\forall m \ m'. (\forall x. \text{DA x} \leq d \Rightarrow m x = m' x) \)

\Rightarrow \text{ExprEval (arglist!i) m = ExprEval (arglist!i) m'}) \)

assume smaller: \(\forall D. (\exists i < \text{length arglist. } D = (\text{dl}!i)) \)

\)

36
\[ (\forall d' \in D. \, d' \leq d) \]

**assume** eqstate: \( \forall x. \, DA \, x \leq d \rightarrow m1 \, x = m2 \, x \)

**from** smaller **have** irangesubst:
\( \forall i < \text{length} \, \text{arglist}. \, \forall d' \in (dl!i). \, d' \leq d \)
by auto

**with** eqstate main **have**
\( \forall i < \text{length} \, \text{arglist}. \, \text{ExprEval} \, (\text{arglist}!i) \, m1 = \text{ExprEval} \, (\text{arglist}!i) \, m2 \)
by force

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

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

**interpretation** Type-System-example: Type-System ExprEval BMap DA
synAssignSC synWhileSC synIfSC
by (unfold-locales, simp add: synAssignSC-def,
metis ExprTypable-with-smallerD-implies-d-indistinguishable,
simp add: synWhileSC-def,
metis ExprTypable-with-smallerD-implies-d-indistinguishable,
simp add: synIfSC-def, metis)
end

**References**
