An Isabelle/HOL formalization of Strong Security

Sylvia Grewe, Alexander Lux, Heiko Mantel, Jens Sauer

September 19, 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. 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:

```plaintext
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) TPCConfig = ('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
```

2
— type for configurations with program states

```haskell
type-synonym ('id, 'val, 'com) PSConfig =
  'com ProgramState × ('id, 'val) State
```

— type for labels with a list of spawned threads

```haskell
type-synonym 'com Label = 'com list
```

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

```haskell
type-synonym ('exp, 'id, 'val, 'com) TLSteps =
  (('id, 'val, 'com) TConfig × 'com Label
  × ('id, 'val, 'com) PSConfig) set
```

— curried version of previously defined type

```haskell
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

```haskell
type-synonym ('exp, 'id, 'val, 'com) TPSteps =
  ((('id, 'val, 'com) TPConfig × ('id, 'val, 'com) TPConfig) set
```

— curried version of previously defined type

```haskell
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

```haskell
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)

```haskell
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

```haskell
type-synonym ('id, 'd) DomainAssignment = 'id ⇒ 'd::order
```

```haskell
type-synonym 'com Bisimulation-type = (('com list) × ('com list)) set
```

— type for escape hatches

```haskell
type-synonym ('d, 'exp) Hatch = 'd × 'exp
```

— type for sets of escape hatches

```haskell
type-synonym ('d, 'exp) Hatches = (('d, 'exp) Hatch) set
```

— type for local escape hatches

```haskell
type-synonym ('d, 'exp) lHatch = 'd × 'exp × nat
```
2 Strong security

2.1 Definition of strong security

We define strong security such that it is parametric in a security lattice (′d).

theory Strong-Security
imports Types
begin

locale Strong-Security =
fixes SR :: (′exp, ′id, ′val, ′com) TSteps
and DA :: (′id, ′d::order) DomainAssignment
begin

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

abbreviation d-equal′ :: (′id, ′val) State ⇒ ′d::order ⇒ (′id, ′val) State ⇒ bool
( ′- ′- )
where
m =d m′ ≡ d-equal d m m′

— transitivity of d-equality
lemma d-equal-trans:
[ [ m =d m'; m'' =d m'' ] ] ⇒ m =d m''
by (simp add: d-equal-def)

abbreviation SRabbr :: (′exp, ′id, ′val, ′com) TSteps-curry
((1⟨-/-⟩) →/ (1⟨-/-⟩) [0,0,0] 81)
where
⟨c,m⟩ → ⟨c',m'⟩ ≡ ((c,m),(c',m')) ∈ 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 =rå m1' 
  ― (∃ W' m2'. ⟨V!i,m1⟩ → ⟨W',m2⟩ ∧ (W,W') ∈ R ∧ m2 =rå 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: \( \forall V V' m1 m1' m2 W. \) [ Strong-d-Bisimulation d R; 
  i < length V ; (V,V') ∈ R; (V!i,m1) → (W,m2); m1 =rå m1' ] 
  ⇒ (∃ W' m2'. (V!i,m1') → (W',m2') ∧ (W,W') ∈ R ∧ m2 =rå 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 USdBeglen: 
  V ≈ d V' ⇒ length V = 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 \( V \approx_d V' \Rightarrow \text{length } V = \text{length } V' \) by (rule USdBlen, auto)
next
 fix \( V V' m1 m1' m2 W i \)
 assume inUSdB: \( V \approx_d V' \)
 assume stepV: \( \langle V'i, m1 \rangle \rightarrow \langle W, m2 \rangle \)
 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-axt stepV irange dequal show
 \( \exists W' m2'. \langle V'i, m1 \rangle \rightarrow \langle W', m2' \rangle \land W \approx_d W' \land m2 =_d m2' \)
 by (simp add: USdB-def, fastforce)

qed

**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 exI, auto)
next
 fix \( V V' V'' \)
 assume p1: \( V \approx_d V' \)
moreover
assume p2: $V' \approx_d V''$
moreover
from p1 USdBeglen[of V V'] have length $V = \text{length } V'$
  by auto
moreover
from p2 USdBeglen[of V' V''] have length $V' = \text{length } V''$
  by auto
ultimately show eqlen: length $V = \text{length } V''$ by auto
next
fix $V V' V'' i m1 m1' W m2$
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: $i < \text{length } V$
from p1 USdBeglen[of V V']
have leq: length $V = \text{length } V'$
  by force
have deq-same: $m1' =_d m1'$ by (simp add: d-equal-def)
from irange step dequal p1 USdB-Strong-d-Bisimulation
  strongdB-aux[of $d \approx_d i V V' m1 W m2 m1'$]
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 deq-same leq USdB-Strong-d-Bisimulation
  strongdB-aux[of $d \approx_d i V' V'' m1' W' m2' m1'$]
irange p2 dequal obtain $W'' m2''$ where p2concl:
  $W' \approx_d W'' \land (V''!i,m1) \rightarrow \langle W'',m2'' \rangle \land m2' =_d m2''$
  by auto
from p1concl p2concl d-equal-trans have tt'': $m2 =_d m2''$
  by blast
from p1concl p2concl have $(W,W'') \in ?R$
  by auto
with p2concl tt'' show $\exists V''. \langle V''!i,m1 \rangle \rightarrow \langle W'',m2'' \rangle \land$
  $(\exists V'. W \approx_d V' \land V' \approx_d W'') \land m2 =_d m2''$
  by auto
qed

hence liftup: $?R \subseteq (\approx_d)$
  by (simp add: USdB-def, auto)
with inRest show $V \approx_d V''$

7
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 ::
\(\upsilon\) ⇒ Bisimulation-type ⇒ bool
where
\(d\text{-Bisimulation-Up-To-USdB} \ d \ R \equiv (\text{sym } R) \land (\forall (V, V') \in R. \ length \ V = length \ V') \land (\forall (V, V') \in R. \ \forall i < \ length \ V. \ \forall m1 \ m1' W \ m2. \ (V!i, m1) \rightarrow (W, m2) \land (m1 =_d m1') \rightarrow (\exists W' m2'. \ (V'!i, m1') \rightarrow (W', m2') \land (W, W') \in (R \cup (\approx_d)) \land (m2 =_d m2')))\)

lemma UpTo-aux: \(\forall V V' m1 m1' m2 W i. \ (d\text{-Bisimulation-Up-To-USdB} \ d \ R; \ i < \ length \ V; \ (V, V') \in R; \ (V!i, m1) \rightarrow (W, m2); \ m1 =_d m1') \rightarrow (\exists W' m2'. \ (V'!i, m1') \rightarrow (W', m2') \land (W, W') \in (R \cup (\approx_d)) \land (m2 =_d m2'))\)
by (simp add: d-Bisimulation-Up-To-USdB-def, fastforce)

lemma RuUSdBeqlen:
\(\forall V V' m1 m1' m2 W i. \ (d\text{-Bisimulation-Up-To-USdB} \ d \ R; \ (V, V') \in R; \ (V!i, m1) \rightarrow (W, m2); \ m1 =_d m1') \rightarrow (\exists W' m2'. \ (V'!i, m1') \rightarrow (W', m2') \land (W, W') \in (R \cup (\approx_d)) \land (m2 =_d m2'))\)
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 \(R \subseteq \approx_d\)
proof –

by auto

qed
\[ \text{def } S \equiv R \cup (\approx_d) \]

from \( S \)-def have \( R \subseteq S \)
  by auto

moreover have \( S \subseteq (\approx_d) \)
proof (simp add: USdB-def, auto, rule-tac \( x=S \) in exI, auto,
  simp add: Strong-d-Bisimulation-def, auto)
  — show symmetry
show \( \text{sym}_S: \text{sym } S \)
proof –
  from upToR
  have \( R_{sym} \): \( \text{sym } R \)
  by (simp add: d-Bisimulation-Up-To-USdB-def)
with \( USdB_{sym} \) have \( Usym: \text{sym } (R \cup (\approx_d)) \)
  by (metis sym-Un)
with \( S \)-def show \(?thesis\)
  by simp
qed
next
fix \( V V' \)
assume inS: \((V, V') \in S \)
  — show equal length (by definition)
from inS \( S \)-def upToR RuUSdBIn
show eqlen: \( \text{length } V = \text{length } V' \)
  by simp
next
  — show general bisimulation property
fix \( V V' W m1 m1' m2 i \)
assume inS: \((V, V') \in S \)
assume irange: \( i < \text{length } V \)
assume stepV: \( \langle V ! i, m1 \rangle \rightarrow \langle W, m2 \rangle \)
assume dequal: \( m1 =_d m1' \)
from inS show \( \exists W' m2'. \langle V ! i, m1' \rangle \rightarrow \langle W', m2' \rangle \land \langle W, W' \rangle \in S \land m2 =_d m2' \)
proof (simp add: \( S \)-def, auto)
  assume firstcase: \((V, V') \in R \)
    with upToR dequal irange stepV
    UpTo-aux[of \( R \) \( i \) \( V V' m1 W m2 m1' \)]
show \( \exists W' m2'. \langle V ! i, m1' \rangle \rightarrow \langle W', m2' \rangle \land \langle W, W' \rangle \in S \land m2 =_d m2' \)
  by (auto simp add: \( S \)-def)
next
  assume secondcase: \( V \approx_d V' \)
from USdB-Strong-d-Bisimulation upToR
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: \( \forall i < \text{length } V. [V!i] \approx_d [V'!i] \)
  shows V \approx_d V'
proof
  def R \equiv \{(V,V'), \text{length } V = \text{length } V' \\
  \land (\forall i < \text{length } V. [V!i] \approx_d [V'!i])\}
  from eqlen partsrelated have inR: (V,V') \in 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') \in 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') \in R 
      assume irange: i < length V

qed
qed end
end
assume step: \((V!i,m1) \rightarrow (RS,m2)\)
assume dequal: \(m1 =_d m1'\)

from inR have Vassump:
length \(V = length V'\) \(\land (\forall i < length V. [V!i] \approx_d [V'!i])\)
by (simp add: \(R\)-def)

with step dequal USdB-Strong-d-Bisimulation irange
strongdB-aux[of \(d \approx_d 0\) \([V!i]\) \([V'!i]\) m1 RS m2 m1']
show \(\exists RS' m2'. \langle V'!i, m1' \rangle \rightarrow \langle RS', m2' \rangle \land \langle (RS, RS') \in R \lor RS \approx_d RS' \rangle \land m2 =_d m2'\)
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 \(\forall i < length V. [V!i] \approx_d [V'!i]\)
proof –
def \(R \equiv \{(C,C'). \exists i W W'. W \approx_d W' \land i < length W \land C = [W!i] \land C' = [W'!i]\}\)

with related have inR: \(\forall i < length V. ([V!i],[V'!i]) \in R\)
by auto
have d-Bisimulation-Up-To-USdB d R
proof (simp add: d-Bisimulation-Up-To-USdB-def , auto)
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 \(C = length C'\)
by (simp add: R-def , auto)
next
fix C C' i m1 m1' RS m2
assume inR: \(\langle C,C' \rangle \in R\)
assume irange: \(i < length C\)
assume step: \(\langle C!i,m1 \rangle \rightarrow \langle RS,m2 \rangle\)
assume dequal: \(m1 =_d m1'\)

from inR obtain \(j W W'\) where Rassump:
\(W \approx_d W'\) \(\land j < length W \land C = [W'j] \land C' = [W'j]\)
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′i1,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:
\forall i < length (c# V). [(c# V)!i] \approx_d [(c'# V')!i]
by auto
thus [c] \approx_d [c']
by force

from intermediate eqlen show V \approx_d V'
proof (case-tac V)
  assume Vcase1: V = []
  with eqlen have V' = [] by auto
  with Vcase1 trivialpair-in-USdB show V \approx_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 < length V \Rightarrow [V!i] \approx_d [V'!i] \)
  by force

  with parallel-composition[of V V'] eqlen
  show V \approx_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 lis (fork - - [70,70] 70)

— SEMANTICS

locale MWLf-semantics = fixes E :: ('exp, 'id, 'val) Evalfunction and BMap :: 'val \(\Rightarrow\) 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<\(-\/-\)>)) \(\rightarrow\) (1<\(-\/-\)>) [0,0,0] 81) where \(\langle c1,m1\rangle \rightarrow \langle c2,m2\rangle\) \(\equiv\) \((c1,m1),(c2,m2)\) \(\in\) MWLfSteps-det | skip: \(\langle\text{skip},m\rangle \rightarrow \langle [],m\rangle\) | assign: \((E;c,m) = v\) \(\rightarrow\) \(\langle x := e,m\rangle \rightarrow \langle [],m(x := v)\rangle\) | seq1: \(\langle c1,m\rangle \rightarrow \langle [],m'\rangle\) \(\rightarrow\) \(\langle c1;c2,m\rangle \rightarrow \langle [c2],m'\rangle\) | seq2: \(\langle c1,m\rangle \rightarrow \langle c1';V,m'\rangle\) \(\rightarrow\) \(\langle c1;c2,m\rangle \rightarrow \langle (c1';c2)#V,m'\rangle\) | iftrue: BMap (E b m) = True \(\rightarrow\) \(\langle\text{if }b\text{ then }c1\text{ else }c2\text{ fi},m\rangle \rightarrow \langle [c1],m\rangle\) | iffalse: BMap (E b m) = False \(\rightarrow\) \(\langle\text{if }b\text{ then }c1\text{ else }c2\text{ fi},m\rangle \rightarrow \langle [c2],m\rangle\) | whiletrue: BMap (E b m) = True \(\rightarrow\) \(\langle\text{while }b\text{ do }c\text{ od},m\rangle \rightarrow \langle [c;(\text{while }b\text{ do }c\text{ od})],m\rangle\) | whilefalse: BMap (E b m) = False \(\rightarrow\) \(\langle\text{while }b\text{ do }c\text{ od},m\rangle \rightarrow \langle [],m\rangle\)
fork: \( (\text{fork} \ c \ V, m) \rightarrow (\text{\#} \ V, m) \)

**inductive-cases** \( \text{MWLfSteps-det-cases} \):

\[
\begin{align*}
\langle \text{skip}, m \rangle & \rightarrow \langle W, m' \rangle \\
\langle x := e, m \rangle & \rightarrow \langle W, m' \rangle \\
\langle e_1; e_2, m \rangle & \rightarrow \langle W, m' \rangle \\
\langle \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ fi}, m \rangle & \rightarrow \langle W, m' \rangle \\
\langle \text{while } b \text{ do } c \text{ od}, m \rangle & \rightarrow \langle W, m' \rangle \\
\langle \text{fork } c \ V, m \rangle & \rightarrow \langle W, m' \rangle
\end{align*}
\]

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

**inductive-set**

\( \text{MWLfSteps-ndet} :: (\exp, \id, \val, (\exp, \id) \text{ MWLfCom}) \text{ TPSteps} \)

and \( \text{MWLfSteps-ndet}' :: (\exp, \id, \val, (\exp, \id) \text{ MWLfCom}) \text{ TPSteps-curry} \)

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

where

\[
\begin{align*}
\langle V_1, m_1 \rangle & \Rightarrow (V_2, m_2) \equiv ((V_1, m_1), (V_2, m_2)) \in \text{MWLfSteps-ndet} \mid \\
\langle c, m \rangle & \Rightarrow \langle c, m' \rangle \Rightarrow (Vf @ [c] @ \text{Va}, m) \Rightarrow (Vf @ c @ \text{Va}, m')
\end{align*}
\]

end

end

### 3.2 Proofs of atomic compositionality results

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

**theory** \textit{Strongly-Secure-Skip-Assign}

**imports** \textit{MWLf Parallel-Composition}

**begin**

**locale** \textit{Strongly-Secure-Programs} =

\( L : \text{MWLf-semantics E BMap} \)

+ \( \text{SS: Strong-Security MWLfSteps-det DA} \)

for \( E :: (\exp, \id, \val) \text{ Evalfunction} \)

and \( \text{BMap :: 'val} \Rightarrow \text{bool} \)

and \( \text{DA :: ('id, 'd::order) DomainAssignment} \)

**begin**

**abbreviation** \textit{USdBname} ::= \textit{d} \Rightarrow (\exp, \id) \textit{MWLfCom Bisimulation-type} 

\( (\approx_d) \)

where \( \approx_d \equiv \text{USdB} \ d \)
abbreviation relatedbyUSdB :: \( ('exp', 'id) \) MWLfCom list \( \Rightarrow \) 'd
\( \Rightarrow \) \( ('exp', 'id) \) MWLfCom list \( \Rightarrow \) bool \( \text{infixr} \approx 65 \)
where \( V \approx_d V' \equiv (V, V') \in \text{USdB} \)

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

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

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

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

theorem Strongly-Secure-Skip:
\([\text{skip}] \approx_d [\text{skip}] \)
proof 
  def \( R0 \equiv \{(V::('exp', 'id) \) MWLfCom list, V'::('exp', 'id) \) MWLfCom list). \)
  \( V = [\text{skip}] \land V' = [\text{skip}] \)
  have uptoR0: \( \text{d-Bisimulation-Up-To-USdB} \ d \ R0 \)
  proof \( \text{simp add: d-Bisimulation-Up-To-USdB-def, auto} \)
    show sym R0 by \( \text{simp add: R0-def sym-def} \)
  next 
    fix \( V \ V' \)
    assume \( (V, V') \in R0 \)
    thus \( \text{length} \ V = \text{length} \ V' \)
    by \( \text{simp add: R0-def} \)
  next 
    fix \( V \ V' i m1 m1' \ W m2 \)
    assume inR0: \( (V, V') \in R0 \)
    assume irange: \( i < \text{length} \ V \)
    assume step: \( \langle V!i, m1 \rangle \rightarrow \langle W, m2 \rangle \)
    assume dequal: \( m1 \approx_d m1' \)
  from inR0 have Vassump:
    \( V = [\text{skip}] \land V' = [\text{skip}] \)
    by \( \text{simp add: R0-def} \)

16
with step irange have step1:
\[ W = \text{[]} \land m2 = m1 \]
by (simp, metis MWLf-semantics.MWLfSteps-det-cases(1))

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

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

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

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

qed

theorem Strongly-Secure-Assign:
assumes \( d\)-indistinguishable-exp: \( e \equiv_{DA} x \ e' \)
shows \( [x := e] \approx_d [x := e'] \)
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 \( \text{length } V = \text{length } V' \)
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: \( \langle V!i,m1 \rangle \rightarrow \langle W,m2 \rangle \)
assume dequal: \( m1 =_d m1' \)
from \(\text{in}\, R_0\) obtain \(x\ e\ e'\) where Vassump:
\(V = [x := e] \land V' = [x := e'] \land e \equiv \text{DA}\ x\ e'
\)
by (simp add: \(R_0\)-def, auto)

with step irange obtain \(v\) where step1:
\(E\ e\ m_1 = v\ \land W = [] \land m_2 = m_1(x := v)
\)
by (auto, metis MWLf-semantics.MWLfSteps-det-cases(2))

from Vassump irange obtain \(m_2'\ v'\) where step2:
\(E\ e'\ m_1' = v'\ \land \langle V^n\!\!, m_1' \rangle \rightarrow \langle [], m_2' \rangle \land m_2' = m_1'(x := v')
\)
by (auto, metis MWLfSteps-det-assign)

with Vassump dequal step step1

have dequalnext: \(m_1(x := v) =_{d} m_1'(x := v')\)
by (simp add: \(d\)-equal-def \(d\)-indistinguishable-def, auto)

with step1 step2 trivialpair-in-USdB show \(\exists W'\ m_2'.\langle V^n\!\!, m_1' \rangle \rightarrow \langle W', m_2' \rangle \land (W, W') \in R_0 \lor W \approx_{d} W'\land m_2 =_{d} m_2'
\)
by auto

qed

hence \(R_0 \subseteq (\approx_{d})\)
by (rule Up-To-Technique)

with \(\text{in}\, R_0\) show \(?\)thesis
by auto

qed

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: [c1] ≈₁ [c1']
  assumes relatedpart2: [c2] ≈₁ [c2']
  shows [c1;c2] ≈₁ [c1';c2']

proof –

  def R0 ≡ {(S1,S2). ∃ c1 c1' c2 c2' W W'.
            S1 = (c1;c2)#W ∧ S2 = (c1';c2')#W' ∧
            [c1] ≈₁ [c1'] ∧ [c2] ≈₁ [c2'] ∧ W ≈₁ W'}

from relatedpart1 relatedpart2 trivialpair-in-USdB
have inR0: ([c1;c2],[c1';c2']) ∈ R0
  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: (S1,S2) ∈ R0
with USdBlen
  show length S1 = length S2
  by (auto simp add: R0-def)
next
fix S1 S2 RS m1 m2 m1' i
  assume inR0: (S1,S2) ∈ R0
  assume irange: i < length S1
  assume S1step: ⟨S1!,m1⟩ → ⟨RS,m2⟩
  assume dequal: m1 =₁ m1'
  from inR0 obtain c1 c1' c2 c2' V V'
    where R0def1: S1 = (c1;c2)#V ∧ S2 = (c1';c2')#V' ∧
       [c1] ≈₁ [c1'] ∧ [c2] ≈₁ [c2'] ∧ V ≈₁ V'
    by (simp add: R0-def, fastforce)
  with irange
  have case-distinction1:
    i = 0 ∨ (V ≠ [] ∧ i ≠ 0)
    by auto
  moreover
  have case1: i = 0 →
     ∃ RS' m2'. ⟨S2!i,m1'⟩ → ⟨RS',m2'⟩ ∧
     (⟨RS,RS'⟩ ∈ R0 ∨ RS ≈₁ RS') ∧ m2 =₁ m2'
  proof –
    assume i0: i = 0
    — get the two different sub-cases:
    with R0def' S1step
    obtain c3 W
    where case-distinction:
\[ RS = [c2] \land (c1,m1) \rightarrow ([],m2) \]
\[ \lor RS = (c3;c2)#W \land (c1,m1) \rightarrow (c3#W,m2) \]
by (simp, metis MWLjSteps-det-cases(3))

moreover

— Case 1: first command terminates
{}
assume RSassump: \( RS = [c2] \)
assume StepAssump: \( (c1,m1) \rightarrow ([],m2) \)

from USdBBeqlen[of \( [] \)] StepAssump R0def'
USdB-Strong-d-Bisimulation dequal
strongdB-aux[of \( d \approx_d i \) ]
\[ [c1] [c1]^! m1 \] m2 m1'] i0
obtain \( W' \ m2' \) where c1c1’reason:
\( (c1',m1') \rightarrow (W',m2') \land W' = [] \)
\( \land \) \( d \) \( W' \land m2 =_d m2' \)
by fastforce

with c1c1’reason have conclpart:
\( (c1';c2',m1') \rightarrow ([c2]',m2') \land m2 =_d m2' \)
by (simp add: MWLjSteps-det.seq1)

with RSassump R0def' i0 have case1-concl:
\( \exists RS' \ m2'. (S2!i,m1') \rightarrow (RS',m2') \land \)
\( ((RS,RS') \in R0 \lor RS \approx_d RS') \land m2 =_d m2' \)
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 \( d \approx_d i \) ]
\[ c3#W \ m2 m1'] i0
obtain \( V'' m2' \) where c1c1’reason:
\( (c1',m1') \rightarrow (V'',m2') \)
\( \land (c3#W) \approx_d V'' \land m2 =_d m2' \)
by fastforce

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

with c1c1’reason have conclpart1:
\( (c1';c2',m1') \rightarrow ((c3';c2')#W',m2') \land m2 =_d m2' \)
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'
bv 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'
bv (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'
bv 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 S1seq: S1!i = V!(i−Suc 0)
by auto

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

with S1seq 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'
bv force

qed
ultimately show $\exists RS' m2'. (S2\{i,m1\} \rightarrow (RS',m2') \land (RS,RS') \in R0 \lor RS \approx_d RS' \land m2 =_d m2')$
by auto

qed

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

with $inR0$ show $\exists$thesis
by auto

qed

\textbf{theorem Compositionality-Fork:}

\textbf{fixes} $V::(\exp, id)$ MWL JCom list
\textbf{assumes} relatedmain: $[c] \approx_d [c']$
\textbf{assumes} relatedthreads: $V \approx_d V'$
\textbf{shows} $[fork c V] \approx_d [fork c' V']$

\textbf{proof}

\textbf{def} $R0 \equiv \{(F1,F2). \exists c1 c1' W W'. F1 = [fork c1 W] \land F2 = [fork c1' W'] \land [c1] \approx_d [c1'] \land W \approx_d W'\}$

\textbf{from} relatedmain relatedthreads
\textbf{have} $inR0: ([fork c V],[fork c' V']) \in R0$
\textbf{by} (simp add: $R0$-def)

\textbf{have} $uptoR0$: $d$-Bisimulation-Up-To-USdB $d R0$
\textbf{proof} (simp add: $d$-Bisimulation-Up-To-USdB-def, auto)
\textbf{from} USdBsym show sym $R0$
\textbf{by} (simp add: $R0$-def sym-def, auto)

\textbf{next}
\textbf{fix} $F1 F2$
\textbf{assume} $inR0$: $(F1,F2) \in R0$
\textbf{with} $R0$-def USdBlen show $length F1 = length F2$
\textbf{by} auto

\textbf{next}
\textbf{fix} $F1 F2 c1V m1 m2 m1' i$
\textbf{assume} $inR0$: $(F1,F2) \in R0$
\textbf{assume} irange: $i < length F1$
\textbf{assume} $F1$step: $\{F1\{i,m1\} \rightarrow \{c1V,m2\}\}$
\textbf{assume} dequal: $m1 =_d m1'$

\textbf{from} $inR0$ \textbf{obtain} $c1 c1' V V'$
\textbf{where} $R0$def': $F1 = [fork c1 V] \land F2 = [fork c1' V'] \land [c1] \approx_d [c1'] \land V \approx_d V'$
\textbf{by} (simp add: $R0$-def, force)

\textbf{from} irange $R0$def' $F1$step
\textbf{have} rew: $c1V = c1\#V \land m2 = m1$

22
by (simp, metis MWLf-semantics.MWLSteps-det-cases(6))

from range R0def’ MWLfSteps-det.fork have F2step:  
  ⟨F2!i,m1’⟩ → ⟨c1’#V’,m1’⟩  
by force

from R0def’ USdB-comp-head-tail have conclpart:  
  ((c1#V,c1’#V’) ∈ R0 ∨ (c1#V) ≈d (c1’#V’))  
by auto

with range rew inR0 F1step dequal R0def’ F2step  
show ∃c1V’ m2’. ⟨F2!i,m1’⟩ → ⟨c1V’,m2’⟩ ∧  
  ((c1V,c1V’) ∈ R0 ∨ c1V ≈d c1V’) ∧ m2 =d m2’  
by fastforce

qed

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

with inR0 show ?thesis  
by auto

qed

theorem Compositionality-If:
  assumes dind-or-branchesrelated:
    b ≈d b’ ∨ [c1] ≈d [c2] ∨ [c1’] ≈d [c2’]  
  assumes branch1related: [c1] ≈d [c1’]  
  assumes branch2related: [c2] ≈d [c2’]  
  shows [if b then c1 else c2 fi] ≈d [if b’ then c1’ else c2’ fi]
proof –
  def R1 ≡ {(I1,I2). ∃c1 c1’ c2 c2’ b b’.
    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’}

  def R2 ≡ {(I1,I2). ∃c1 c1’ c2 c2’ b b’.
    I1 = [if b then c1 else c2 fi] ∧ I2 = [if b’ then c1’ else c2’ fi] ∧  
    [c1] ≈d [c1’] ∧ [c2] ≈d [c2’] ∧  
    ([c1] ≈d [c2] ∨ [c1’] ≈d [c2’])}

  def R0 ≡ R1 ∪ R2

from dind-or-branchesrelated branch1related branch2related  
have inR0: ([if b then c1 else c2 fi],[if b’ then c1’ else c2’ fi]) ∈ 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 \( \text{symR1}: \text{sym R1} \)
by (simp add: sym-def R1-def, fastforce)
from USdBsym
have \( \text{symR2}: \text{sym R2} \)
by (simp add: sym-def R2-def, fastforce)

from symR1 symR2 show \( \text{sym R0} \)
by (simp add: sym-def R0-def)

next
fix \( I_1, I_2 \)
assume inR0: \((I_1, I_2) \in R0\)
thus \( \text{length } I_1 = \text{length } I_2 \)
by (simp add: R0-def R1-def R2-def, auto)

next
fix \( I_1, I_2, R, m_1, m_1', m_2, i \)
assume inR0: \((I_1, I_2) \in R0\)
assume irange: \( i < \text{length } I_1 \)
assume \( I_1 \text{step} \): \( \langle I_1! i, m_1 \rangle \rightarrow \langle R, m_2 \rangle \)
assume dequal: \( m_1 =_d m_1' \)

have \( \text{inR1case}: (I_1, I_2) \in R1 \)
\( \implies \exists R', m_2'. \ (I_1, i, m_1') \rightarrow \langle R', m_2' \rangle \wedge \langle (R, R'), R0 \rangle \vee R \approx_d R \wedge m_2 =_d m_2' \)

proof
assume inR1: \((I_1, I_2) \in R1\)

then obtain \( c_1, c_1', c_2, c_2', b, b' \) where \( R1\text{def}' \):
\( I_1 = \langle \text{if } b \text{ then } c_1 \text{ else } c_2 \rangle \) \wedge
\( I_2 = \langle \text{if } b' \text{ then } c_1' \text{ else } c_2' \rangle \) \wedge
\( [c_1] =_d [c_1'] \wedge [c_2] =_d [c_2'] \wedge b =_d b' \)
by (simp add: R1-def, force)

moreover
— get the two different cases True and False from semantics:
from irange \( R1\text{def}' \) \( I_1 \text{step} \) have case-distinction:
\( R = [c_1] \wedge \text{BMap (E b m_1) = True} \vee \)
\( R = [c_2] \wedge \text{BMap (E b m_1) = False} \)
by (simp, metis MWLf-semantics.MWLSteps-det-cases(4))

moreover
— Case 1: \( b \) evaluates to True
\{ 
assume bevalT: \( \text{BMap (E b m_1) = True} \)
assume RSassump: \( RS = [c_1] \)
from irange bevalT \( I_1 \text{step} \) \( R1\text{def}' \) \( RS\text{assump} \) have memeq:
\( m_2 = m_1 \)
by (simp, metis MWLf-semantics.MWLSteps-det-cases(4))

from bevalT \( R1\text{def}' \) dequal have \( b'\text{evalT} \):
\( \text{BMap (E b' m_1') = True} \)
by (simp add: d-indistinguishable-def)

24
hence \texttt{I2step-case1}: 
\[(\text{if } b' \text{ then } c1' \text{ else } c2' \text{ fi}, m1') \rightarrow \langle[c1'], m1'\rangle\]
by (simp add: \texttt{MWLSteps-det.iftrue})

with \texttt{irange dequal RSassump memeq R1def'}
have case1-concl:
\[
\exists RS' m2'. \langle i!i,m1' \rangle \rightarrow \langle RS', m2' \rangle \wedge 
((RS.RS') \in R0 \lor RS \approx d RS') \wedge m2 = d m2'
\]
by auto
}
moreover
— Case 2: \(b\) evaluates to False

\{
assume bevalF: BMap (E b m1) = False
assume RSassump: RS = [c2]
from \texttt{irange bevalF I1step R1def'} \texttt{RSassump} have memeq:
\[
m1 = m2
\]
by (simp, metis \texttt{MWLf-semantics.MWLSteps-det-cases(4)})

from \texttt{bevalF R1def'} \texttt{dequal} have b'evalF:
\[
BMap (E b' m1') = False
\]
by (simp add: \texttt{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: \texttt{MWLSteps-det.iffalse})

with \texttt{irange dequal RSassump memeq R1def'}
have case1-concl:
\[
\exists RS' m2'. \langle i!i,m1' \rangle \rightarrow \langle RS', m2' \rangle \wedge 
((RS.RS') \in R0 \lor RS \approx d RS') \wedge m2 = d m2'
\]
by auto
}
ultimately show
\[
\exists RS' m2'. \langle i!i,m1' \rangle \rightarrow \langle RS', m2' \rangle \wedge 
((RS.RS') \in R0 \lor RS \approx d RS') \wedge m2 = d m2'
\]
by auto
qed

have \texttt{inR2case}: \((I1,I2) \in R2\)
\[
\Rightarrow \exists RS' m2'. \langle i!i,m1' \rangle \rightarrow \langle RS', m2' \rangle \wedge 
((RS.RS') \in R0 \lor RS \approx_d RS') \wedge m2 =_d m2'
\]
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
\[ [c_1] \approx_d [c_1'] \land [c_2] \approx_d [c_2'] \land ( [c_1] \approx_d [c_2] \lor [c_1'] \approx_d [c_2']) \]

by (simp add: R2-def, force)

moreover

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

from range R2def' I1step have case-distinction-left:
\begin{align*}
(RS = [c_1] \lor RS = [c_2]) \land m_2 = m_1
\end{align*}

by (simp, metis MWLf-semantics.MWLfSteps-det-cases(4))

moreover

from range R2def' dequal obtain RS' where I2step:
\begin{align*}
\langle I_2!i, m_1' \rangle \rightarrow (RS', m_1') \\
\land (RS' = [c_1'] \lor RS' = [c_2']) \land m_1 =_d m_1'
\end{align*}

by (simp, metis MWLfSteps-det.iffalse MWLfSteps-det.iftrue)

moreover

from USdBtrans have \([ [c_1] \approx_d [c_2]; [c_2] \approx_d [c_2'] ] \] 
\[ \Longrightarrow [c_1] \approx_d [c_2'] \]

by (unfold trans-def, blast)

moreover

from USdBtrans have \([ [c_1] \approx_d [c_1]; [c_1'] \approx_d [c_2'] ] \] 
\[ \Longrightarrow [c_1] \approx_d [c_2'] \]

by (unfold trans-def, blast)

moreover

from USdBsym have \([ [c_1] \approx_d [c_2] ] \Longrightarrow [c_2] \approx_d [c_1] \]

by (simp add: sym-def)

moreover

from USdBtrans have \([ [c_2] \approx_d [c_1]; [c_1] \approx_d [c_1'] ] \] 
\[ \Longrightarrow [c_2] \approx_d [c_1'] \]

by (unfold trans-def, blast)

moreover

from USdBsym have \([ [c_1'] \approx_d [c_2'] ] \Longrightarrow [c_2'] \approx_d [c_1'] \]

by (simp add: sym-def)

moreover

from USdBtrans have \([ [c_2] \approx_d [c_2]; [c_2'] \approx_d [c_1'] ] \] 
\[ \Longrightarrow [c_2] \approx_d [c_1'] \]

by (unfold trans-def, blast)

ultimately show
\[ \exists RS', m_2'. \langle I_2!i, m_1' \rangle \rightarrow (RS', m_2') \land ((RS, RS') \in R_0 \lor RS \approx_d RS') \land m_2 =_d m_2' \]

by auto

qed

from inR0 inR1case inR2case show
\[ \exists RS', m_2'. \langle I_2!i, m_1' \rangle \rightarrow (RS', m_2') \land ((RS, RS') \in R_0 \lor RS \approx_d RS') \land m_2 =_d m_2' \]

by (auto simp add: R0-def)

qed

hence \( R_0 \subseteq (\approx_d) \)

by (rule Up-To-Technique)

26
with inR0 show ?thesis
  by auto

qed

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

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

  def R2 \( \equiv \{ (W1,W2). \exists c1 c1' b b'.\)
  \( W1 = [\text{while } b \text{ do } c1 \text{ od}] \land W2 = [\text{while } b' \text{ do } c1' \text{ od}] \land \)
  \([c1] \approx_d [c1'] \land 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 R0
proof (simp add: d-Bisimulation-Up-To-USdB-def , auto)
  from USdBsym d-indistinguishable-sym have symR1: sym R1
    by (simp add: sym-def R1-def, fastforce)
  from USdBsym d-indistinguishable-sym 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 W1 W2
  assume inR0: \((W1,W2) \in R0\)
  with USdBEqn 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: \(m1 \equiv_d m1'\)

  from inR0 show \(\exists RS' m2'. \langle W2!i,m1' \rangle \rightarrow \langle RS',m2' \rangle \land \)
    \((RS,RS') \in R0 \lor RS \approx_d RS' \land m2 =_d m2'\)
  proof (simp add: R0-def , auto)
assume inR1: (W1,W2) ∈ R1

then obtain c1 c1’ c2 c2’ b b’ V V’
where R1def’: W1 = (c1;(while b do c2 od))#V ∧ W2 = (c1’;(while b’ do c2’ od))#V’ ∧ [c1] ≈ [c1’] ∧ [c2] ≈ [c2’] ∧ V ≈ V’ ∧ b ≈ b’
by (simp add: R1-def, force)

with irange have case-distinction1: i = 0 ∨ (V ≠ [] ∧ i ≠ 0)
by auto

moreover have case1: i = 0 ⇒ ∃RS’ m2’. ⟨W2!i,m1⟩ → ⟨RS’,m2’⟩ ∧ ((RS,RS’) ∈ R1 ∨ (RS,RS’) ∈ R2 ∨ RS ≈ RS’)
∧ m2 = d m2’
proof −
assume i0: i = 0
— get the two different sub-cases:
with R1def’ W1step obtain c3 W where case-distinction:
RS = [while b do c2 od] ∧ ⟨c1,m1⟩ → ⟨[] , m2⟩
∨ RS = (c3;(while b do c2 od))#W ∧ ⟨c1,m1⟩ → ⟨c3#W , m2⟩
by (simp, metis MWLfSteps-det-cases)

moreover
— Case 1: first command terminates
{ assume RSAssump: RS = [while b do c2 od]
assume StepAssump: ⟨c1,m1⟩ → ⟨[] , m2⟩

from USdBeglen[of []] StepAssump R1def’
USdB-Strong-d-Bisimulation dequal strongdB-aux[of d ≈ d i]
[c1] [c1’] m1 [] m2 m1’ i0
obtain W’ m2’ where c1c1’reason:
⟨c1’,m1’⟩ → ⟨W’,m2’⟩ ∧ W’ = []
∧ [] ≈ d W’ ∧ m2 = d m2’
by fastforce

with c1c1’reason have conclpart1:
⟨c1’;(while b’ do c2’ od),m1’⟩ → ⟨[while b’ do c2’ od],m2’⟩ ∧ m2 = d m2’
by (simp add: MWLfSteps-det-seq1)

from R1def’ have conclpart2:
([while b do c2 od],[while b’ do c2’ od]) ∈ R2
by (simp add: R2-def)

with conclpart1 RSAssump i0 R1def’
have case1-concl:
\[ \exists RS', m2'. \langle W!i, m1' \rangle \rightarrow \langle RS', m2' \rangle \land (RS, RS') \in R1 \lor (RS, RS') \in R2 \lor RS \approx_d RS' \land m2 =_d m2' \]

by auto

\}

moreover

— Case 2: first command does not terminate

{ 
assume RSassump: RS = (c3; (while b do c2 od)) \# W
assume StepAssump: \langle c1, m1 \rangle \rightarrow \langle c3 \# W, m2 \rangle 

from StepAssump R1def' USdB-Strong-d-Bisimulation dequal
strongdB-aux[of d \approx_d i 
\[ c1 \] \[ c1 \# W m2 m1 \] i0
obtain V'' m2' where c1c1'reason:
\langle c1', m1' \rangle \rightarrow \langle V'', m2' \rangle 
\land (c3 \# W) \approx_d V'' \land m2 =_d m2' 
by fastforce

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

with c1c1'reason have conclpart1:
\langle c1'; (while b' do c2' od), m1' \rangle \rightarrow
\langle (c3'; (while b' do c2' od)) \# W', m2' \rangle 
\land m2 =_d m2' 
by (simp add: MWLSteps-det-seq2)

from V'' reason
c1c1'reason USdB-decomp-head-tail[of c3 W]
USdB-Strong-d-Bisimulation
have c3aWinUSDB:
\[ [c3] \approx_d [c3'] \land W \approx_d W' \]
by blast

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

with i0 RSassump R1def' V'' reason conclpart1
have case2-concl:
\exists RS' m2'. \langle W!i, m1' \rangle \rightarrow \langle RS', m2' \rangle \land (RS, RS') \in R1 \lor (RS, RS') \in R2 \lor RS \approx_d RS' 
\land m2 =_d m2' 
by auto

\}

ultimately
show \( \exists RS', m2'. (W2!i, m1) \to (RS', m2') \land \)
\((RS, RS') \in R1 \lor (RS, RS') \in R2 \lor RS \approx_d RS' \land m2 =_d m2' \)
by blast
qed
moreover
have case2: \[
[ V \neq []; i \neq 0 ] \implies \exists RS', m2'. (W2!i, m1) \to (RS', m2') \land
((RS, RS') \in R1 \lor (RS, RS') \in R2 \lor RS \approx_d RS') \land m2 =_d m2'
\]
proof –
assume Vnonempt: \( V \neq [] \)
assume inot0: \( i \neq 0 \)
with Vnonempt irange R1def' have irange:
\((i - Suc 0) < \text{length } V\)
by simp
from inot0 R1def have W1eq: \( W1! = V!(i - Suc 0) \)
by auto
from inot0 R1def have W2i = V!(i - Suc 0)
by auto
with W1eq R1def' W1step irange dequal
USdB-Strong-d-Bisimulation
strongdB-aux[of d USdB d
i - Suc 0 V V' m1 RS m2 m1']
show \( \exists RS', m2'. (W2!i, m1) \to (RS', m2') \land \)
\((RS, RS') \in R1 \lor (RS, RS') \in R2 \lor RS \approx_d RS') \land m2 =_d m2'
by force
qed
ultimately show \( \exists RS', m2'. (W2!i, m1) \to (RS', m2') \land \)
\((RS, RS') \in R1 \lor (RS, RS') \in R2 \lor RS \approx_d RS') \land m2 =_d m2'
by auto
next
assume inR2: \((W1, W2) \in R2 \)
then obtain c1 c1' b b' where R2def':
\( W1 = [\text{while } b \text{ do } c1 \text{ od}] \land W2 = [\text{while } b' \text{ do } c1' \text{ od}] \land \)
\[ [c1] \approx_d [c1'] \land b \equiv_d b' \]
by (auto simp add: R2-def)
— get the two different cases:
moreover
from irange R2def' W1step have case-distinction:
RS = \([c1; (\text{while } b \text{ do } c1 \text{ od})] \land \text{BMap } (E b m1) = \text{True} \lor \]
RS = \([] \land \text{BMap } (E b m1) = \text{False} \)

30
by (simp,metis MWLfl-semantics.MWLflSteps-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 MWLfl-semantics.MWLflSteps-det-cases(5)) 
  }

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

with trivialpair-in-USdB irange dequal RSassump 
memeq R2def'


have case1-concl:
\[ \exists RS', m_2'. (W2!i,m1') \rightarrow (RS',m2') \land \\
((RS,RS') \in R_1 \lor (RS,RS') \in R_2 \lor RS \approx_d RS') \land m_2 =_d m_2' \]
by force
}
ultimately

show \( \exists RS', m_2'. (W2!i,m1') \rightarrow (RS',m2') \land \\
((RS,RS') \in R_1 \lor (RS,RS') \in R_2 \lor RS \approx_d RS') \land m_2 =_d m_2' \)
by auto
qed
qed

hence \( R_0 \subseteq (\approx_d) \)
by (rule Up-To-Technique)
with \( \text{in} R_0 \) show \( \text{thesis} \)
by auto
qed
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.
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) MWLfCom ⇒ bool
assumes semAssignSC: AssignSideCondition x e ⇒ e ≡ DA x e
and semWhileSC: WhileSideCondition e ⇒ ∀d. e ≡ₜ d e
and semIfSC: IfSideCondition e c1 c2 ⇒ ∀d. e ≡ₜ d e ∨ [c1] ≈ₜ [c2]

begin
— Security typing rules for the language commands

inductive
ComSecTyping :: (exp, id) MWLfCom ⇒ bool
(⇒ C -)
and ComSecTypingL :: (exp, id) MWLfCom list ⇒ bool
(⇒ V -)
where
skip: C skip ⇒ C c
Assign: C AssignSideCondition x e ⇒ C x := e
Fork: C c; V ⇒ C fork 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. C V!i ⇒ V

inductive-cases parallel-cases:
⇒ V
— soundness proof of abstract type system

theorem ComSecTyping-single-is-sound:
⇒ C c ⇒ 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} ComSecTyping-list-is-sound:
\[ \forall V. V \Rightarrow \text{Strongly-Secure } V \]
\textbf{by} (metis ComSecTyping-single-is-sound Strongly-Secure-def parallel-composition parallel-cases)

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.

\textbf{theory} \textit{Expr} \textbf{imports} Types \textbf{begin}

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

\textbf{type-synonym} (‘val) operation = ‘val list ⇒ ‘val

\textbf{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
\textbf{and} ExprEvalL :: ((‘id, ‘val) Expr list ⇒ (‘id, ‘val) State ⇒ ‘val list
\textbf{where}
ExprEval (Const v) m = v |
ExprEval (Var x) m = (m x) |
ExprEval (Op f arglist) m = (f (ExprEvalL arglist m)) |

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

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
lemma \[ \text{ExprTypable-with-smallerD-implies-d-indistinguishable} \]
\[ (\text{id}, \text{val}) \text{ Expr} \Rightarrow \text{Dom} \]
\[ (\text{id}, \text{val}) \text{ Expr} \Rightarrow \text{bool} \]

\[ e1 \equiv_d e2 \]
\[ \equiv \text{Strongly-Secure-Programs.d-indistinguishable} \text{ ExprEval DA d e1 e2} \]

abbreviation \[ \text{relatedbyUSdB} \] :: \[ ((\text{id}, \text{val}) \text{ Expr}, \text{'id}) \text{ MWLjCom list} \Rightarrow \text{Dom} \Rightarrow ((\text{id}, \text{val}) \text{ Expr}, \text{'id}) \text{ MWLjCom list} \Rightarrow \text{bool (infixr \approx \_65)} \]

\[ V \approx_d V' \equiv (V, V') \in \text{Strong-Security.USdB} \]
\[ (\text{MWLj-semantics.MWLjSteps-det ExprEval BMap}) \text{ DA d} \]

--- 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
\[ \text{Consts: } \vdash e \{ \text{Const v} \} \in \{ d \} | \]
\[ \text{Vars: } \vdash e \{ \text{Var x} \} \in \{ DA x \} | \]
\[ \text{Ops: } \forall i < \text{length arglist}. \vdash e \{ \text{arglist!i} \} : (d!i) \]
\[ \Rightarrow \vdash e \{ Op f \text{ arglist!i} \} : \bigcup \{ d. (\exists i < \text{length arglist}. d = (d!i)) \} \]

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
\[ \text{synWhileSC e} \equiv \exists D. (\vdash e : D \land (\forall d \in D. (\forall d'. (d \leq d'))) \]

definition \[ \text{synIfSC} :: (\text{id}, \text{val}) \text{ Expr} \Rightarrow ((\text{id}, \text{val}) \text{ Expr}, \text{'id}) \text{ MWLjCom} \Rightarrow ((\text{id}, \text{val}) \text{ Expr}, \text{'id}) \text{ MWLjCom} \Rightarrow \text{bool} \]

where
\[ \text{synIfSC e c1 c2} \equiv \forall d. (\neg (e \equiv_d e) \Rightarrow [c1] \equiv_d [c2]) \]

lemma \[ \text{ExprTypable-with-smallerD-implies-d-indistinguishable}: \]
\[ \llbracket \vdash e : D'; \forall d' \in D'. (d' \leq d) \rrbracket \Rightarrow e \equiv_d e \]

proof (induct rule: \text{ExprSecTyping.induct},
\[ \text{simp-all add: Strongly-Secure-Programs.d-indistinguishable-def, auto}) \]

fix \[\text{dl} \text{ and arglist:}((\text{id}, \text{val}) \text{ Expr}) \text{ list and f::} \text{val list} \Rightarrow \text{val} \]
\[\text{and m1::}((\text{id}, \text{val}) \text{ State and m2::}((\text{id}, \text{val}) \text{ State} \]

assume main: \[ \forall i < \text{length arglist}. \vdash e \{ \text{arglist!i} \} : d!i \land \]
\[ ((\forall d' \in (d!i)). (d' \leq d) \Rightarrow (\forall m' \in (f i). DA x \leq d \Rightarrow m x = m x') \]
\[ \Rightarrow \text{ExprEval (arglist!i) m} = \text{ExprEval (arglist!i) m')} \]

assume smaller: \[ \forall D. (\exists i < \text{length arglist}. D = (d!i)) \]}
\[ \forall d' \in D. \ d' \leq d \]

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

**from** **smaller** **have** **irangesubst:**

\( \forall i < \text{length arglist.} \ \forall d' \in (dl!i). \ d' \leq d \)

**by** **auto**

**with** **eqstate main** **have**

\( \forall i < \text{length arglist.} \ \text{ExprEval (arglist!i) m1 = ExprEval (arglist!i) m2} \)

**by** **force**

**hence** **substmap:** (ExprEvalL arglist m1) = (ExprEvalL arglist m2)

**by** (induct arglist, auto, force)

**show** \( f (\text{ExprEvalL arglist m1}) = f (\text{ExprEvalL arglist m2}) \)

**by** (subst substmap, auto)

**qed**

**interpretation** **Type-System-example:** Type-System ExprEval BMap DA synAssignSC synWhileSC synIfSC


**end**

**References**
