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.
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
```
— type for configurations with program states
\textbf{type-synonym} \((id, val, com)\) PSConfig =
\(\text{com ProgramState} \times (id, val)\) State

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

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

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

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

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

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

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

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

\textbf{type-synonym} \((com\ \text{Bisimulation-type})\ =\ ((\text{com\ list}) \times (\text{com\ list}))\) 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 = \(d \times exp \times \text{nat}\)
— type for sets of local escape hatches

**type-synonym** `('d, 'exp) lHatches = (('d, 'exp) lHatch) set`

end

2 Strong security

2.1 Definition of strong security

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

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 ⇒ bool`

where

`d-equal d m m' ≡ ∀ x. ((DA x) ≤ d −→ (m x) = (m' x))`

**abbreviation**

`d-equal' :: ('id, 'val) State ⇒ 'd::order ⇒ ('id, 'val) State ⇒ bool`

( (· =· ·) )

where

`m =_d m' ≡ d-equal d m m'`

— transitivity of d-equality

**lemma** `d-equal-trans`:

`[ m =_d m'; m' =_d m'' ] −→ m =_d m''`

(proof)

**abbreviation**

`SRabbr :: ('exp, 'id, 'val, 'com) 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 =_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
((≈. -) [66,66] 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: (∀ V V' m1 m1' m2 W i. [ Strong-d-Bisimulation 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 ∧ m2 =_d m2'))
⟨proof⟩

lemma trivialpair-in-USdB:
[ ] ≈_d [ ]
⟨proof⟩

lemma USdBsym: sym (≈_d)
⟨proof⟩

lemma USdBeglen:
V ≈_d V' ⇒ length V = length V'
⟨proof⟩

lemma USdB-Strong-d-Bisimulation:
  Strong-d-Bisimulation d (≈_d)
⟨proof⟩
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: exp 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'))
⟨proof⟩

lemma RaUSdBeglen:
  [ d-Bisimulation-Up-To-USdB d R;
    (V, V') ∈ (R ∪ (≈_d)) ]
  ⇒ length V = length V'
⟨proof⟩

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

lemma parallel-decomposition:
  assumes related: V ≈_d V' 
  shows \( \forall i < \text{length } V. [V!i] \approx_d [V'!i] \)
⟨proof⟩

lemma USdB-comp-head-tail:
  assumes relatedhead: \([c] \approx_d [c']\)
  assumes relatedtail: V ≈_d V' 
  shows \( (c\#V) \approx_d (c'\#V') \)
⟨proof⟩

lemma USdB-decomp-head-tail:
  assumes relatedlist: \((c\#V) \approx_d (c'\#V')\)
  shows \([c] \approx_d [c'] \land V \approx_d V'\)
⟨proof⟩

end

end
3 Example language and compositionality proofs

3.1 Example language with dynamic thread creation

As in [2], we instantiate the language with a simple while language that supports dynamic thread creation via a 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
 ((λ (·,·)) ⇒ (λ (·,·)) [0,0,0,0] 81)
where
⟨c1,m1⟩ → ⟨c2,m2⟩ ⇔ ((c1,m1),(c2,m2)) ∈ MWLfSteps-det | skip: ⟨skip,m⟩ → ⟨[],m⟩ |
assign: \((E \ e \ m) = v \implies (x := e,m) \rightarrow ([],m(x := v))\) |
seq1: \((c1,m) \rightarrow ([],m') \implies (c1;c2,m) \rightarrow ([c2],m')\) |
seq2: \((c1,m) \rightarrow (c1'\#V,m') \implies (c1;c2,m) \rightarrow ([c1'\;c2]\#V,m')\) |
iftrue: BMap \((E \ b \ m) = True \implies \langle if \ b \ then \ c1 \ else \ c2 \ fi, m \rangle \rightarrow \langle [\;], m \rangle |
iffalse: BMap \((E \ b \ m) = False \implies \langle if \ b \ then \ c1 \ else \ c2 \ fi, m \rangle \rightarrow \langle [\;], m \rangle |
whiletrue: BMap \((E \ b \ m) = True \implies \langle while \ b \ do \ c \ od, m \rangle \rightarrow \langle [\;], m \rangle |
whilefalse: BMap \((E \ b \ m) = False \implies \langle while \ b \ do \ c \ od, m \rangle \rightarrow \langle [\;], m \rangle |
fork: \langle fork \ c \ V, m \rangle \rightarrow \langle c\#V, m \rangle

inductive-cases MLfSteps-det-cases:
\langle \text{skip,} m \rangle \rightarrow \langle W, m' \rangle
\langle x := e, m \rangle \rightarrow \langle W, m' \rangle
\langle c1;c2, m \rangle \rightarrow \langle W, m' \rangle
\langle if \ b \ then \ c1 \ else \ c2 \ fi, m \rangle \rightarrow \langle W, m' \rangle
\langle while \ b \ do \ c \ od, m \rangle \rightarrow \langle W, m' \rangle
\langle fork \ c \ V, m \rangle \rightarrow \langle W, m' \rangle

— non-deterministic, possibilistic system step (added for intuition, not used in the proofs)
inductive-set
MLfSteps-ndet \:: \((\text{'exp, 'id, 'val, ('exp,'id) MLfCom}) \text{ TPSteps})
and MLfSteps-ndet' \:: \((\text{'exp, 'id, 'val, ('exp,'id) MLfCom}) \text{ TPSteps-curry}
((1 \text{-/-}) \Rightarrow (1 \text{-/-}) \{0,0,0,0\} 81)
where
\langle V1,m1 \rangle \Rightarrow \langle V2,m2 \rangle \equiv ((V1,m1),(V2,m2)) \in MLfSteps-ndet |
\langle ci,m \rangle \Rightarrow \langle c,m' \rangle \Rightarrow (Vf \, @ \, [ci] @ Va,m) \Rightarrow (Vf \, @ \, c @ Va,m')

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 Strongly-Secure-Skip-Assign
imports MLf Parallel-Composition
begin
locale Strongly-Secure-Programs =
L? : MLf-semantics E BMap
+ SS?: Strong-Security MLfSteps-det DA
for E :: ('exp, 'id, 'val) Evalfunction
and BMap :: 'val ⇒ bool
and DA :: ('id, 'd::order) DomainAssignment
begin

abbreviation USdBname ::'d ⇒ ('exp, 'id) MLfCom Bisimulation-type
(≈.)
where ≈d ≡ USdB d

abbreviation relatedbyUSdB :: ('exp,'id) MLfCom list ⇒ 'd
⇒ ('exp,'id) MLfCom 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:
e ≡d e' ⇒ e' ≡d e
⟨proof⟩
lemma d-indistinguishable-trans:
[ [ e ≡d e'; e' ≡d e'' ] ] ⇒ e ≡d e''
⟨proof⟩

theorem Strongly-Secure-Skip:
[skip] ≈d [skip]
⟨proof⟩

theorem Strongly-Secure-Assign:
assumes d-indistinguishable-exp: e ≡DA x e'
shows [x := e] ≈d [x := e']
⟨proof⟩
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: $[c1] \approx_d [c1']$
  assumes relatedpart2: $[c2] \approx_d [c2']$
  shows $[c1;c2] \approx_d [c1'c2']$
  ⟨proof⟩

theorem Compositionality-Fork:
  fixes $V::('exp','id)$ MWlfCom list
  assumes relatedmain: $[c] \approx_d [c']$
  assumes relatedthreads: $V \approx_d V'$
  shows $[fork c V] \approx_d [fork c' V']$
  ⟨proof⟩

theorem Compositionality-If:
  assumes dind-or-branchesrelated:
    $b \equiv_d b' \lor [c1] \approx_d [c1'] \lor [c1] \approx_d [c1']$
  assumes branch1related: $[c1] \approx_d [c1']$
  assumes branch2related: $[c2] \approx_d [c2']$
  shows $[if b then c1 else c2 fi] \approx_d [if b' then c1' else c2' fi]$
  ⟨proof⟩

theorem Compositionality-While:
  assumes dind: $b \equiv_d b'$
  assumes bodrelated: $[c] \approx_d [c']$
  shows $[while b do c od] \approx_d [while b' do c' od]$
  ⟨proof⟩

end

end
4 Security type system

4.1 Abstract security type system with soundness proof

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

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

theory Type-System
imports Language-Composition
begin

locale Type-System =
  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 ⇒ ('exp,'id) MWLfCom ⇒ 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] ≈ₜ [c2]

begin
— Security typing rules for the language commands
inductive
ComSecTyping :: ('exp, 'id) MWLfCom ⇒ bool
(Γ C -)
and ComSecTypingL :: ('exp,'id) MWLfCom list ⇒ bool
(Γ V -)
where
skip: Γ C skip |
Assign: [ AssignSideCondition x e ] ⇒ Γ C x := e |
Fork: Γ C c; Γ V V ⇒ Γ C fork c V |
Seq: Γ C c1; Γ C c2 ⇒ Γ C c1;c2 |
While: Γ C c; Γ WhileSideCondition b |

12
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.

theory Expr
imports Types
begin

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

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

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

— defining a simple recursive evaluation function on this datatype
primrec ExprEval :: (('id, 'val) Expr, 'id, 'val) Evalfunction
and ExprEvalL :: (('id, 'val) Expr) list ⇒ ('id, 'val) State ⇒ 'val list
where
ExprEval (Const v) m = v |
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: \(d_1 \leq d_2\) = (if \(d_1 = d_2\) then True else (if \(d_1 = \text{low}\) then True else False))

definition less-Dom-def: \(d_1 < d_2\) = (if \(d_1 = d_2\) then False else (if \(d_1 = \text{low}\) then True else False))

instance ⟨proof⟩

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-indistinguishable :: (′id,′val) Expr ⇒ Dom
g ⇒ (′id,′val) Expr ⇒ bool
where
e1 ≡_d e2 ⇒ \text{Strongly-Secure-Programs} \cdot \text{d-indistinguishable} \cdot \text{ExprEval} \cdot DA \cdot d \cdot e1 \cdot e2

abbreviation relatedbyUSdB :: ((′id,′val)MWLfCom list ⇒ Dom ⇒ ((′id,′val)MWLfCom list ⇒ bool \text{infixr} ≈ 65)
where \text{V ≈}_d V′ ≡ (V, V′) ∈ \text{Strong-Security. USdB}
(MWLj-semantics. MWLfSteps-det \text{ExprEval} \cdot BMap) \cdot DA \cdot d

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

inductive
ExprSecTyping :: (′id,′val) Expr ⇒ Dom set ⇒ bool
(\vdash \cdot : -)
where
Cons: \vdash \mathcal{E} (\text{Const } v) : \{d\} |
Vars: \vdash \mathcal{E} (\text{Var } x) : \{DA \cdot x\} |
Ops: \forall i < \text{length arglist}. \vdash \mathcal{E} (\text{arglist}!i) : (dl!i)
⇒ \vdash \mathcal{E} (\text{Op } f \text{ arglist}) : (\bigcup \{d. (\exists i < \text{length arglist}. \text{d} = (dl!i))\})

definition synAssignSC :: ′id ⇒ (′id,′val) Expr ⇒ bool
where
synAssignSC x e ≡ ∃ D. (\vdash \mathcal{E} e : D ∧ (\forall d \in D. (d ≤ DA \cdot x)))

definition synWhileSC :: (′id,′val) Expr ⇒ bool
where
synWhileSC e ≡ ∃ D. (\vdash \mathcal{E} e : D ∧ (\forall d \in D. \forall d′. d ≤ d′))

definition synIfSC :: (′id,′val) Expr ⇒ ((′id,′val) Expr, ′id) MWLfCom ⇒ ((′id,′val) Expr, ′id) MWLfCom ⇒ bool
where
synIfSC e c1 c2 ≡ 
\forall d. (\neg (e ≡_d e) \rightarrow [c1] ≈_d [c2])

lemma ExprTypable-with-smallerD-implies-d-indistinguishable:
\[\vdash \mathcal{E} e : D'; \forall d′ \in D'. d′ \leq d \implies e ≡_d e\]
(\text{proof})

interpretation Type-System-example: Type-System \text{ExprEval} \cdot BMap \cdot DA
\text{synAssignSC} \cdot \text{synWhileSC} \cdot \text{synIfSC}
(\text{proof})
References
