An Isabelle/HOL formalization of Strong Security

Sylvia Grewe, Alexander Lux, Heiko Mantel, Jens Sauer

May 28, 2015

Abstract

Research in information-flow security aims at developing methods to identify undesired information leaks within programs from private sources to public sinks. Noninterference captures this intuition. 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 .......... 6
  2.3 Proof of parallel compositionality .................... 7

3 Example language and compositionality proofs 8
  3.1 Example language with dynamic thread creation ....... 8
  3.2 Proofs of atomic compositionality results .......... 9
  3.3 Proofs of non-atomic compositionality results ...... 11
1 Preliminary definitions

1.1 Type synonyms

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

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

```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) \texttt{PSConfig} = 'com ProgramState × ('id, 'val) State

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

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

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

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

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

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

— define the same type as \texttt{TSteps}, but in a curried version (allowing syntax abbreviations)
\textbf{type-synonym} ('exp, 'id, 'val, 'com) \texttt{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)
\textbf{type-synonym} ('id, 'd) \texttt{DomainAssignment} = 'id ⇒ 'd::order

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

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

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

— type for local escape hatches
\textbf{type-synonym} ('d, 'exp) \texttt{lHatch} = 'd × 'exp × 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 ⇒ (′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
```

4
— 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
(≈ d)

where
≈ d ≡ \{ r. (Strong-d-Bisimulation d r) \}

abbreviation relatedbyUSdB :: 'com list ⇒ 'd ⇒ 'com list ⇒ bool
(infixr ≈ d)

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: \[ \land V V' m1 m1' m2 W i. [ Strong-d-Bisimulation d R; i < length V ; (V,V') ∈ R; (V!i,m1) → (W,m2); m1 =_d m1' ] \] \[ ⇒ \[ \exists W' m2'. (V!i,m1') → (W',m2') ∧ (W,W') ∈ R ∧ m2 =_d m2' \] \]

(thesis)

lemma trivialpair-in-USdB:
\[] ≈ d \]
(thesis)

lemma USdBsym:
\sym (≈ d)
(thesis)

lemma USdBneglen:
\ V ≈ d V' ⇒ length V = length V'
(thesis)

lemma USdB-Strong-d-Bisimulation:
Strong-d-Bisimulation d (≈ d)
(thesis)
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 ::
\((v, v') \in R \Rightarrow \text{com Bisimulation-type} \Rightarrow \text{bool}\)
where
\(d\)-Bisimulation-Up-To-USdB \(d\) \(R\)
\(\equiv\)
\((\text{sym } R) \land (\forall (v, v') \in R. \text{length } v = \text{length } v') \land\)
\((\forall (v, v') \in R. \forall i < \text{length } v. \forall m1 m1' W m2.\)
\((v!i, m1) \rightarrow (W, m2) \land (m1 =_d m1')\)
\(\rightarrow (\exists W' m2'. (v'!i, m1') \rightarrow (W', m2'))\)
\(\land (W, W') \in (R \cup (\approx_d)) \land (m2 =_d m2'))\)

lemma UpTo-aux:
\((i < \text{length } v; (v, v') \in R; (v!i, m1) \rightarrow (W, m2); m1 =_d m1')\)
\(\Rightarrow\)
\((\exists W' m2'. (v'!i, m1') \rightarrow (W', m2'))\)
\(\land (W, W') \in (R \cup (\approx_d)) \land (m2 =_d m2'))\)
(proof)

lemma RuUSdBeqlen:
\[(d\)-Bisimulation-Up-To-USdB \(d\) \(R\);\]
\((v, v') \in (R \cup (\approx_d))\)]
\(\Rightarrow\)
\(\text{length } v = \text{length } v'\)
(proof)

lemma Up-To-Technique:
assumes upToR:
\(d\)-Bisimulation-Up-To-USdB \(d\) \(R\)
shows \(R \subseteq \approx_d\)
(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: ∀ i < length V. [V!i] ≈_d [V'!i]
  shows V ≈_d V'
⟨proof⟩

lemma parallel-decomposition:
  assumes related: V ≈_d V'
  shows ∀ i < length V. [V!i] ≈_d [V'!i]
⟨proof⟩

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

lemma USdB-decomp-head-tail:
  assumes relatedlist: (c#V) ≈_d (c'#V')
  shows [c] ≈_d [c'] ∧ V ≈_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 sup-
ports 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 (\textquote{exp}).

theory MWLf
imports Types
begin
— SYNTAX
— Commands for the multi-threaded while language with fork (to instantiate \textquote{com})

datatype ('exp, 'id) MWLfCom
  = Skip (skip)
  | Assign 'id 'exp
      (\texttt{\texttt{:=}} \texttt{[70,70] 70})
  | Seq ('exp, 'id) MWLfCom ('exp, 'id) MWLfCom
      (\texttt{::} \texttt{[61,60] 60})
  | If-Else 'exp ('exp, 'id) MWLfCom ('exp, 'id) MWLfCom
      (\texttt{if - then - else - fi} \texttt{[80,79,79] 70})
  | While-Do 'exp ('exp, 'id) MWLfCom
      (\texttt{while - do - od} \texttt{[80,79] 70})
  | Fork ('exp, 'id) MWLfCom (('exp, 'id) MWLfCom list
      (\texttt{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(\texttt{\texttt{-/-}})) \Rightarrow (1(\texttt{\texttt{-/-}})) \texttt{[0,0,0,0]} 81)
where
\texttt{\{c1,m1\} \rightarrow \{c2,m2\} \equiv ((c1,m1),(c2,m2)) \in MWLfSteps-det |}
skip: \langle \text{skip}, m \rangle \rightarrow \langle [], m \rangle | 
assign: \langle E \ c \ m \rangle = 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'\#V), m' \rangle | 
iftrue: BMap (E \ b \ m) = True \Rightarrow \langle if \ b \ then \ c1 \ else \ c2 \ fi, m \rangle \rightarrow \langle [c1], m \rangle | 
iffalse: BMap (E \ b \ m) = False \Rightarrow \langle if \ b \ then \ c1 \ else \ c2 \ fi, m \rangle \rightarrow \langle [c2], m \rangle | 
whiletrue: BMap (E \ b \ m) = True \Rightarrow \langle while \ b \ do \ c \ od, m \rangle \rightarrow \langle [c1; while \ b \ do \ c \ od], m \rangle | 
whilefalse: BMap (E \ b \ m) = False \Rightarrow \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 MWL\textbf{f}Steps-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 c\#V, m \rangle

---

end

---

end

---

3.2 Proofs of atomic compositionality results

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

theory Strongly-Secure-Skip-Assign
imports MWL\textbf{f} Parallel-Composition
begin
locale Strongly-Secure-Programs =

L :: MWLf-semantics E BMap
+ SS: Strong-Security MWLfSteps-det DA
for E :: (‘exp, ‘id, ‘val) Evalfunction
and BMap :: ‘val ⇒ bool
and DA :: (‘id, ‘d::order) DomainAssignment

begin

abbreviation USdBname ::‘d ⇒ (‘exp, ‘id) MWLfCom Bisimulation-type
(≃.)
where ≃d ≡ USdB d

abbreviation relatedbyUSdB :: (‘exp,’id) MWLfCom list ⇒ ‘d
⇒ (‘exp,’id) MWLfCom list ⇒ bool (infix ÷ 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

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 $V::(\exp',\id)\ MWLF\Com\ list$
  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:
    $b \equiv_d b' \lor [c1] \approx_d [c2] \lor [c1'] \approx_d [c2']$
  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 bodyrelated: $[c] \approx_d [c']$
  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) M威尔Com ⇒ ('exp,'id) M威尔Com ⇒ 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) M威尔Com ⇒ bool
  (⊢C -)
and ComSecTypingL :: ('exp,'id) M威尔Com list ⇒ bool
  (⊢V -)
where
  skip: ⊢C skip |
  Assign: [ AssignSideCondition x e ] ⇒ ⊢C x := e |
  Fork: [ ⊢C c; ⊢V V ] ⇒ ⊢C fork c V |
  Seq: [ ⊢C c1; ⊢C c2 ] ⇒ ⊢C c1;c2 |
  While: [ ⊢C c; WhileSideCondition b ]

12
inductive-cases parallel-cases:

\[ \forall \ i < \text{length } V. \ C \ i \implies C \ V \]

— soundness proof of abstract type system

\textbf{theorem} \ \textit{ComSecTyping-single-is-sound}:
\[ \vdash C \ i \implies \text{Strongly-Secure} \ [c] \]
(\textit{proof})

\textbf{theorem} \ \textit{ComSecTyping-list-is-sound}:
\[ \vdash V \implies \text{Strongly-Secure} \ V \]
(\textit{proof})

end
end

4.2 Example language for Boolean and arithmetic expressions

As an example, we provide a simple example language for instantiating the parameter 'exp for the language for Boolean and arithmetic expressions.

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

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

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

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

— defining a simple recursive evaluation function on this datatype
\textbf{primrec} ExprEval :: (('id, 'val) Expr, 'id, 'val) Evalfunction
\textbf{and} ExprEvalL :: (('id, 'val) Expr) list \Rightarrow ('id, 'val) State \Rightarrow '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)

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⟩

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
⇒ (′id,′val) Expr ⇒ bool
where
e1 ≡_d e2
≡ Strongly-Secure-Programs.d-indistinguishable ExprEval DA d e1 e2

abbreviation relatedbyUSdB' :: ((′id,′val) Expr, ′id) MWLfCom list
⇒ Dom ⇒ ((′id,′val) Expr, ′id) MWLfCom list ⇒ bool (infixr ≈_d 65)
where V ≈_d V′ ⇒ (V,V′) ∈ Strong-Security.USdB
(MWLf-semantics.MWLfSteps-det ExprEval BMap) DA d

— Security typing rules for expressions - will be part of a side condition
inductive
ExprSecTyping :: (′id,′val) Expr ⇒ Dom set ⇒ bool
(⊢_E - : -)
where
Consts: ⊢_E (Const v) : {d} |
Vars: ∀ i. ⊢_E (Var x) : {DA x} |
Ops: ∀ i. (Op f arglist). ⊢_E (arglist!i) : (dl!i)
⇒ ⊢_E (Op f arglist) : (∪ (exists i. dl!i))

definition synAssignSC :: ′id ⇒ (′id,′val) Expr ⇒ bool
where
synAssignSC x e ≡∃ D. (⊢_E e : D ∧ (∀ d ∈ D. (d ≤ DA x)))

definition synWhileSC :: (′id,′val) Expr ⇒ bool
where
synWhileSC e ≡∃ D. (⊢_E e : D ∧ (∀ d ∈ D. (d ≤ d′)))

definition synIfSC :: (′id,′val) Expr ⇒ ((′id,′val) Expr, ′id) MWLfCom
⇒ ((′id,′val) Expr, ′id) MWLfCom ⇒ bool
where
synIfSC e c1 c2 ≡
∀ d. (∼ (c1 ≡_d e) −→ [c1] ≈_d [c2])

lemma ExprTypable-with-smallerD-implies-d-indistinguishable:
⊢_E e : D; (∀ d′ ∈ D′. d′ ≤ d) −→ e ≡_d e
⟨proof⟩

interpretation Type-System-example: Type-System ExprEval BMap DA
synAssignSC synWhileSC synIfSC
⟨proof⟩
References
