

Secure information flow and program logics — Isabelle/HOL sources

Lennart Beringer and Martin Hofmann

February 6, 2026

Abstract

We present interpretations of type systems for secure information flow in Hoare logic, complementing previous encodings in relational program logics. We first treat the imperative language **IMP**, extended by a simple procedure call mechanism. For this language we consider base-line non-interference in the style of Volpano et al. [8] and the flow-sensitive type system by Hunt and Sands [4]. In both cases, we show how typing derivations may be used to automatically generate proofs in the program logic that certify the absence of illicit flows. We then add instructions for object creation and manipulation, and derive appropriate proof rules for base-line non-interference. As a consequence of our work, standard verification technology may be used for verifying that a concrete program satisfies the non-interference property.

The present proof development represents an update of the formalisation underlying our paper [2] and is intended to resolve any ambiguities that may be present in the paper.

Contents

1	The language IMP	2
1.1	Syntax	3
1.2	Dynamic semantics	3
2	Program logic	5
2.1	Assertions and their semantic validity	5
2.2	Proof system	6
2.3	Soundness	7
2.4	Admissible rules	8
2.5	Completeness	9
3	Base-line noninterference	9
3.1	Basic definitions	10
3.2	Derivation of the LOW rules	10

3.3	Derivation of the HIGH rules	12
3.4	The type system of Volpano, Smith and Irvine	13
3.5	Contextual closure	16
4	Lattices	18
5	Flow-sensitivity a la Hunt and Sands	19
5.1	General $A; R \Rightarrow S$ -security	19
5.2	Basic definitions	19
5.3	Type system	20
5.4	Derived proof rules	22
5.5	Soundness results	26
6	Base-line non-interference with objects	28
6.1	Syntax and operational semantics	28
6.2	Program logic	32
6.2.1	Assertions and their semantic validity	32
6.2.2	Proof system	32
6.2.3	Soundness	34
6.2.4	Derived rules	34
6.2.5	Completeness	35
6.3	Partial bijections	35
6.4	Non-interference	37
6.4.1	Indistinguishability relations	37
6.4.2	Definition and characterisation of security	39
6.5	Derivation of proof rules	40
6.5.1	Low proof rules	40
6.5.2	High proof rules	42
6.6	Type system	44
6.7	Contextual closure	47

theory *IMP* imports *Main* begin

1 The language IMP

In this section we define a simple imperative programming language. Syntax and operational semantics are as in [9], except that we enrich the language with a single unnamed, parameterless procedure. Both, this section and the following one merely set the basis for the development described in the later sections and largely follow the approach to formalize program logics advocated by Kleymann, Nipkow, and others - see for example [5, 6, 7].

1.1 Syntax

We start from unspecified categories of program variables and values.

typedec1 *Var*

typedec1 *Val*

Arithmetic expressions are inductively built up from variables, values, and binary operators which are modeled as meta-logical functions over values. Similarly, boolean expressions are built up from arithmetic expressions using binary boolean operators which are modeled as functions of the ambient logic HOL.

datatype *Expr* =

varE *Var*

| *valE* *Val*

| *opE* *Val* \Rightarrow *Val* \Rightarrow *Val Expr Expr*

datatype *BExpr* = *compB* *Val* \Rightarrow *Val* \Rightarrow *bool Expr Expr*

Commands are the usual ones for an imperative language, plus the command *Call* which stands for the invocation of a single (unnamed, parameterless) procedure.

datatype *IMP* =

Skip

| *Assign* *Var Expr*

| *Comp* *IMP IMP*

| *While* *BExpr IMP*

| *Iff* *BExpr IMP IMP*

| *Call*

The body of this procedure is identified by the following constant.

consts *body* :: *IMP*

1.2 Dynamic semantics

States are given by stores - in our case, HOL functions mapping program variables to values.

type-synonym *State* = *Var* \Rightarrow *Val*

definition *update* :: *State* \Rightarrow *Var* \Rightarrow *Val* \Rightarrow *State*

where *update* *s* *x* *v* = (λ *y* . if *x*=*y* then *v* else *s* *y*)

The evaluation of expressions is defined inductively, as standard.

primrec *evalE*::*Expr* \Rightarrow *State* \Rightarrow *Val*

where

evalE (*varE* *x*) *s* = *s* *x* |

evalE (*valE* *v*) *s* = *v* |

evalE (*opE* *f* *e1* *e2*) *s* = *f* (*evalE* *e1* *s*) (*evalE* *e2* *s*)

primrec $evalB :: BExpr \Rightarrow State \Rightarrow bool$

where

$evalB (compB f e1 e2) s = f (evalE e1 s) (evalE e2 s)$

The operational semantics is a standard big-step relation, with a height index that facilitates the Kleymann-Nipkow-style [5, 6] soundness proof of the program logic.

inductive-set $Semn :: (State \times IMP \times nat \times State) \text{ set}$ **where**

$SemSkip: (s, Skip, 1, s) : Semn$

| $SemAssign:$

$\llbracket t = update\ s\ x\ (evalE\ e\ s) \rrbracket \Longrightarrow (s, Assign\ x\ e, 1, t) : Semn$

| $SemComp:$

$\llbracket (s, c1, n, r) : Semn; (r, c2, m, t) : Semn; k = (max\ n\ m) + 1 \rrbracket$
 $\Longrightarrow (s, Comp\ c1\ c2, k, t) : Semn$

| $SemWhileT:$

$\llbracket evalB\ b\ s; (s, c, n, r) : Semn; (r, While\ b\ c, m, t) : Semn;$
 $k = ((max\ n\ m) + 1) \rrbracket$
 $\Longrightarrow (s, While\ b\ c, k, t) : Semn$

| $SemWhileF: \llbracket \neg (evalB\ b\ s); t = s \rrbracket \Longrightarrow (s, While\ b\ c, 1, t) : Semn$

| $SemTrue:$

$\llbracket evalB\ b\ s; (s, c1, n, t) : Semn \rrbracket \Longrightarrow (s, Iff\ b\ c1\ c2, n+1, t) : Semn$

| $SemFalse:$

$\llbracket \neg (evalB\ b\ s); (s, c2, n, t) : Semn \rrbracket \Longrightarrow (s, Iff\ b\ c1\ c2, n+1, t) : Semn$

| $SemCall: (s, body, n, t) : Semn \Longrightarrow (s, Call, n+1, t) : Semn$

abbreviation

$SemN :: [State, IMP, nat, State] \Rightarrow bool$ ($\langle - , - \rightarrow - \rangle$)

where

$s, c \rightarrow_n t == (s, c, n, t) : Semn$

Often, the height index does not matter, so we define a notion hiding it.

definition $Sem :: [State, IMP, State] \Rightarrow bool$ ($\langle - , - \Downarrow - \rangle 1000$)

where $s, c \Downarrow t = (\exists n. s, c \rightarrow_n t)$

Inductive elimination rules for the (indexed) dynamic semantics:

inductive-cases $Sem\text{-eval-cases}:$

$s, Skip \rightarrow_n t$

$s, (Assign\ x\ e) \rightarrow_n t$

$s, (Comp\ c1\ c2) \rightarrow_n t$

$s, (While\ b\ c) \rightarrow_n t$

$s, (Iff\ b\ c1\ c2) \rightarrow_n t$

$s, \text{Call} \rightarrow_n t$

An induction on c shows that no derivations of height 0 exist.

lemma *Sem-no-zero-height-derivs*: $(s, c \rightarrow_0 t) \implies \text{False}$

The proof of determinism is by induction on the (indexed) operational semantics.

lemma *SemDeterm*: $\llbracket s, c \Downarrow t; s, c \Downarrow r \rrbracket \implies r=t$

End of theory IMP

end

theory *VDM* **imports** *IMP* **begin**

2 Program logic

The program logic is a partial correctness logic in (precondition-less) VDM style. This means that assertions are binary predicates over states and relate the initial and final states of a terminating execution.

2.1 Assertions and their semantic validity

Assertions are binary predicates over states, i.e. are of type

type-synonym *VDMAssn* = $\text{State} \Rightarrow \text{State} \Rightarrow \text{bool}$

Command c satisfies assertion A if all (terminating) operational behaviours are covered by the assertion.

definition *VDM-valid* :: $\text{IMP} \Rightarrow \text{VDMAssn} \Rightarrow \text{bool}$

$(\langle \models - : - \rangle [100,100] 100)$

where $\models c : A = (\forall s t . (s, c \Downarrow t) \longrightarrow A s t)$

A variation of this property for the height-indexed operational semantics,...

definition *VDM-validn* :: $\text{nat} \Rightarrow \text{IMP} \Rightarrow \text{VDMAssn} \Rightarrow \text{bool}$

$(\langle \models - : - \rangle [100,100,100] 100)$

where $\models_n c : A = (\forall m . m \leq n \longrightarrow (\forall s t . (s, c \rightarrow_m t) \longrightarrow A s t))$

...plus the obvious relationships.

lemma *VDM-valid-validn*: $\models c:A \implies \models_n c:A$

lemma *VDM-validn-valid*: $(\forall n . \models_n c:A) \implies \models c:A$

lemma *VDM-lowerm*: $\llbracket \models_n c:A; m \leq n \rrbracket \implies \models_m c:A$

Proof contexts are simply sets of assertions – each entry represents an assumption for the unnamed procedure. In particular, a context is valid if each entry is satisfied by the method call instruction.

definition *Ctxt-valid* :: $VDMAssn\ set \Rightarrow bool$ ($\langle \vdash - \rangle [100] 100$)
where $\vdash G = (\forall A . A \in G \longrightarrow (\vdash Call : A))$

Again, a relativised sibling ...

definition *Ctxt-validn* :: $nat \Rightarrow (VDMAssn\ set) \Rightarrow bool$
 $(\langle \vdash_n - \rangle [100,100] 100)$
where $\vdash_n G = (\forall m . m \leq n \longrightarrow (\forall A . A \in G \longrightarrow (\vdash_m Call : A)))$

satisfies the obvious properties.

lemma *Ctxt-valid-validn*: $\vdash G \Longrightarrow \vdash_n G$
lemma *Ctxt-validn-valid*: $(\forall n . \vdash_n G) \Longrightarrow \vdash G$
lemma *Ctxt-lowerm*: $\llbracket \vdash_n G; m < n \rrbracket \Longrightarrow \vdash_m G$

A judgement is valid if the validity of the context implies that of the command-assertion pair.

definition *valid* :: $(VDMAssn\ set) \Rightarrow IMP \Rightarrow VDMAssn \Rightarrow bool$
 $(\langle \vdash - : - \rangle [100,100,100] 100)$
where $G \vdash c : A = (\vdash G \longrightarrow \vdash c : A)$

And, again, a related notion of judgement validity.

definition *validn* ::
 $(VDMAssn\ set) \Rightarrow nat \Rightarrow IMP \Rightarrow VDMAssn \Rightarrow bool$
 $(\langle \vdash_n - : - \rangle [100,100,100,100] 100)$
where $G \vdash_n c : A = (\vdash_n G \longrightarrow \vdash_n c : A)$

lemma *validn-valid*: $(\forall n . G \vdash_n c : A) \Longrightarrow G \vdash c : A$
lemma *ctxt-consn*: $\llbracket \vdash_n G; \vdash_n Call:A \rrbracket \Longrightarrow \vdash_n (\{A\} \cup G)$

2.2 Proof system

inductive-set

VDM-proof :: $(VDMAssn\ set \times IMP \times VDMAssn)\ set$
where

VDMSkip: $(G, Skip, \lambda s\ t . t=s) : VDM-proof$

| *VDMAssign*:
 $(G, Assign\ x\ e, \lambda s\ t . t = (update\ s\ x\ (evalE\ e\ s))) : VDM-proof$

| *VDMComp*:
 $\llbracket (G, c1, A1) : VDM-proof; (G, c2, A2) : VDM-proof \rrbracket \Longrightarrow$
 $(G, Comp\ c1\ c2, \lambda s\ t . \exists r . A1\ s\ r \wedge A2\ r\ t) : VDM-proof$

| *VDMIff*:
 $\llbracket (G, c1, A) : VDM-proof; (G, c2, B) : VDM-proof \rrbracket \Longrightarrow$
 $(G, Iff\ b\ c1\ c2, \lambda s\ t . (((evalB\ b\ s) \longrightarrow A\ s\ t) \wedge$
 $((\neg (evalB\ b\ s)) \longrightarrow B\ s\ t))) : VDM-proof$

| *VDMWhile*:

$$\llbracket (G, c, B):VDM\text{-proof}; \forall s. (\neg \text{eval}B\ b\ s) \longrightarrow A\ s\ s;$$

$$\forall s\ r\ t. \text{eval}B\ b\ s \longrightarrow B\ s\ r \longrightarrow A\ r\ t \longrightarrow A\ s\ t \rrbracket \Longrightarrow$$

$$(G, \text{While}\ b\ c, \lambda s\ t. A\ s\ t \wedge \neg (\text{eval}B\ b\ t)) : VDM\text{-proof}$$

| *VDMCall*:

$$(\{A\} \cup G, \text{body}, A):VDM\text{-proof} \Longrightarrow (G, \text{Call}, A):VDM\text{-proof}$$

| *VDMAx*: $A \in G \Longrightarrow (G, \text{Call}, A):VDM\text{-proof}$

| *VDMConseq*:

$$\llbracket (G, c, A):VDM\text{-proof}; \forall s\ t. A\ s\ t \longrightarrow B\ s\ t \rrbracket \Longrightarrow$$

$$(G, c, B):VDM\text{-proof}$$

abbreviation

$VDM\text{-deriv} :: [VDM\text{Assn}\ \text{set}, IMP, VDM\text{Assn}] \Rightarrow \text{bool}$
 $(\triangleleft \triangleright - : - \rightarrow [100,100,100]\ 100)$

where $G \triangleright c : A == (G, c, A) \in VDM\text{-proof}$

The while-rule is in fact inter-derivable with the following rule.

lemma Hoare-While:

$G \triangleright c : (\lambda s\ s'. \forall r. \text{eval}B\ b\ s \longrightarrow I\ s\ r \longrightarrow I\ s'\ r) \Longrightarrow$

$G \triangleright \text{While}\ b\ c : (\lambda s\ s'. \forall r. I\ s\ r \longrightarrow (I\ s'\ r \wedge \neg \text{eval}B\ b\ s'))$

apply (*subgoal-tac* $G \triangleright (\text{While}\ b\ c)$:

$(\lambda s\ t. (\lambda s\ t. \forall r. I\ s\ r \longrightarrow I\ t\ r)\ s\ t \wedge \neg (\text{eval}B\ b\ t)))$

apply (*erule* *VDMConseq*)

apply *simp*

apply (*rule* *VDMWhile*) **apply** *assumption* **apply** *simp* **apply** *simp*

done

Here's the proof in the opposite direction.

lemma VDMWhile-derivable:

$\llbracket G \triangleright c : B; \forall s. (\neg \text{eval}B\ b\ s) \longrightarrow A\ s\ s;$

$\forall s\ r\ t. \text{eval}B\ b\ s \longrightarrow B\ s\ r \longrightarrow A\ r\ t \longrightarrow A\ s\ t \rrbracket$

$\Longrightarrow G \triangleright (\text{While}\ b\ c) : (\lambda s\ t. A\ s\ t \wedge \neg (\text{eval}B\ b\ t))$

apply (*rule* *VDMConseq*)

apply (*rule* *Hoare-While* [*of* $G\ c\ b\ \lambda s\ r. \forall t. A\ s\ t \longrightarrow A\ r\ t$])

apply (*erule* *VDMConseq*) **apply** *clarsimp*

apply *fast*

done

2.3 Soundness

An auxiliary lemma stating the soundness of the while rule. Its proof is by induction on n .

lemma SoundWhile[rule-format]:

$(\forall m. G \models_m c : B) \longrightarrow (\forall s. (\neg \text{eval}B\ b\ s) \longrightarrow A\ s\ s) \longrightarrow$

$(\forall s. \text{eval}B\ b\ s \longrightarrow (\forall r. B\ s\ r \longrightarrow (\forall t. A\ r\ t \longrightarrow A\ s\ t))) \longrightarrow$

$G \models_n (\text{While } b \ c) : (\lambda s \ t. \ A \ s \ t \wedge \neg \text{evalB } b \ t)$

Similarly, an auxiliary lemma for procedure invocations. Again, the proof proceeds by induction on n .

lemma *SoundCall[rule-format]*:

$\llbracket \forall n. \models_n (\{A\} \cup G) \longrightarrow \models_n \text{body} : A \rrbracket \implies \models_n G \longrightarrow \models_n \text{Call} : A$

The heart of the soundness proof is the following lemma which is proven by induction on the judgement $G \triangleright c : A$.

lemma *VDM-Sound-n*: $G \triangleright c : A \implies (\forall n. G \models_n c:A)$

Combining this result with lemma *validn-valid*, we obtain soundness in contexts,...

theorem *VDM-Sound*: $G \triangleright c : A \implies G \models c:A$

...and consequently soundness w.r.t. empty contexts.

lemma *VDM-Sound-emptyCtxt*: $\{\} \triangleright c : A \implies \models c : A$

2.4 Admissible rules

A weakening rule and some cut rules are easily derived.

lemma *WEAK[rule-format]*:

$G \triangleright c : A \implies (\forall H. G \subseteq H \longrightarrow H \triangleright c : A)$

lemma *CutAux*:

$\llbracket H \triangleright c : A; H = \text{insert } P \ D; G \triangleright \text{Call} : P; G \subseteq D \rrbracket \implies D \triangleright c : A$

lemma *Cut*: $\llbracket G \triangleright \text{Call} : P; (\text{insert } P \ G) \triangleright c : A \rrbracket \implies G \triangleright c : A$

We call context G verified if all entries are justified by derivations for the procedure body.

definition *verified*: $\text{VDMAssn set} \Rightarrow \text{bool}$

where *verified* $G = (\forall A. A : G \longrightarrow G \triangleright \text{body} : A)$

The property is preserved by sub-contexts

lemma *verified-preserved*: $\llbracket \text{verified } G; A : G \rrbracket \implies \text{verified } (G - \{A\})$

The *Mutrec* rule allows us to eliminate verified (finite) contexts. Its proof proceeds by induction on n .

theorem *Mutrec*:

$\llbracket \text{finite } G; \text{card } G = n; \text{verified } G; A : G \rrbracket \implies \{\} \triangleright \text{Call} : A$

In particular, *Mutrec* may be used to show that verified finite contexts are valid.

lemma *Ctxt-verified-valid*: $\llbracket \text{verified } G; \text{finite } G \rrbracket \implies \models G$

2.5 Completeness

Strongest specifications, given precisely by the operational behaviour.

definition $SSpec :: IMP \Rightarrow VDMAssn$

where $SSpec\ c\ s\ t = s, c \Downarrow t$

Strongest specifications are valid ...

lemma *SSpec-valid*: $\models c : (SSpec\ c)$

and imply any other valid assertion for the same program (hence their name).

lemma *SSpec-strong*: $\models c : A \implies \forall\ s\ t. SSpec\ c\ s\ t \longrightarrow A\ s\ t$

By induction on c we show the following.

lemma *SSpec-derivable*: $G \triangleright Call : SSpec\ Call \implies G \triangleright c : SSpec\ c$

The (singleton) strong context contains the strongest specification of the procedure.

definition *StrongG* :: $VDMAssn\ set$

where $StrongG = \{SSpec\ Call\}$

By construction, the strongest specification of the procedure's body can be verified with respect to this context.

lemma *StrongG-Body*: $StrongG \triangleright body : SSpec\ Call$

Thus, the strong context is verified.

lemma *StrongG-verified*: $verified\ StrongG$

Using this result and the rules *Cut* and *Mutrec*, we show that arbitrary commands satisfy their strongest specification with respect to the empty context.

lemma *SSpec-derivable-empty*: $\{\} \triangleright c : SSpec\ c$

From this, we easily obtain (relative) completeness.

theorem *VDM-Complete*: $\models c : A \implies \{\} \triangleright c : A$

Finally, it is easy to show that valid contexts are verified.

lemma *Ctxt-valid-verified*: $\models G \implies verified\ G$

End of theory VDM

end

theory *VS* imports *VDM* begin

3 Base-line noninterference

We now show how to interpret the type system of Volpano, Smith and Irvine [8], as described in Section 3 of our paper [2].

3.1 Basic definitions

Multi-level security being treated in Section 5, we restrict our attention in the present section to the two-point security lattice.

datatype $TP = low \mid high$

A global context assigns a security type to each program variable.

consts $CONTEXT :: Var \Rightarrow TP$

Next, we define when two states are considered (low) equivalent.

definition $twiddle :: State \Rightarrow State \Rightarrow bool$ ($\langle - \approx - \rangle [100,100] 100$)
where $s \approx ss = (\forall x. CONTEXT\ x = low \longrightarrow s\ x = ss\ x)$

A command c is *secure* if the low equivalence of any two initial states entails the equivalence of the corresponding final states.

definition $secure :: IMP \Rightarrow bool$
where $secure\ c = (\forall s\ t\ ss\ tt. s \approx t \longrightarrow (s, c \Downarrow ss) \longrightarrow (t, c \Downarrow tt) \longrightarrow ss \approx tt)$

Here is the definition of the assertion transformer that is called *Sec* in the paper ...

definition $Sec :: ((State \times State) \Rightarrow bool) \Rightarrow VDMAssn$
where $Sec\ \Phi\ s\ t = ((\forall r. s \approx r \longrightarrow \Phi(t, r)) \wedge (\forall r. \Phi(r, s) \longrightarrow r \approx t))$

... and the proofs of two directions of its characteristic property, Proposition 1.

lemma $Prop1A: \models c : (Sec\ \Phi) \Longrightarrow secure\ c$
lemma $Prop1B:$
 $secure\ c \Longrightarrow \models c : Sec\ (\lambda (r, t). \exists s. (s, c \Downarrow r) \wedge s \approx t)$
lemma $Prop1BB: secure\ c \Longrightarrow \exists \Phi. \models c : Sec\ \Phi$
lemma $Prop1:$
 $(secure\ c) = (\models c : Sec\ (\lambda (r, t). \exists s. (s, c \Downarrow r) \wedge s \approx t))$

3.2 Derivation of the LOW rules

We now derive the interpretation of the LOW rules of Volpano et al's paper according to the constructions given in the paper. (The rules themselves are given later, since they are not yet needed).

lemma $CAST[rule-format]:$
 $G \triangleright c : twiddle \longrightarrow G \triangleright c : Sec\ (\lambda (s, t). s \approx t)$
lemma $SKIP: G \triangleright Skip : Sec\ (\lambda (s, t). s \approx t)$
lemma $ASSIGN:$
 $(\forall s\ ss. s \approx ss \longrightarrow evalE\ e\ s = evalE\ e\ ss) \Longrightarrow$
 $G \triangleright (Assign\ x\ e)$
 $: (Sec\ (\lambda (s, t). s \approx (update\ t\ x\ (evalE\ e\ t))))$
lemma $COMP:$
 $\llbracket G \triangleright c1 : (Sec\ \Phi); G \triangleright c2 : (Sec\ \Psi) \rrbracket \Longrightarrow$
 $G \triangleright (Comp\ c1\ c2) : (Sec\ (\lambda (s, t). \exists r. \Phi(r, t) \wedge$

$$(\forall w . (r \approx w \longrightarrow \Psi(s, w))))$$

lemma *IFF*:

$$\begin{aligned} & \llbracket (\forall s \ ss. s \approx ss \longrightarrow \text{evalB } b \ s = \text{evalB } b \ ss); \\ & G \triangleright c1 : (\text{Sec } \Phi); G \triangleright c2 : (\text{Sec } \Psi) \rrbracket \Longrightarrow \\ & G \triangleright (\text{Iff } b \ c1 \ c2) : \text{Sec } (\lambda (s, t) . (\text{evalB } b \ t \longrightarrow \Phi(s, t)) \wedge \\ & \quad ((\neg \text{evalB } b \ t) \longrightarrow \Psi(s, t))) \end{aligned}$$

We introduce an explicit fixed point construction over the type TT of the invariants Φ .

type-synonym $TT = (\text{State} \times \text{State}) \Rightarrow \text{bool}$

We deliberately introduce a new type here since the agreement with *VDMAssn* (modulo currying) is purely coincidental. In particular, in the generalisation for objects in Section 6 the type of invariants will differ from the type of program logic assertions.

definition *FIX*:: $(TT \Rightarrow TT) \Rightarrow TT$

where $\text{FIX } \varphi = (\lambda (s, t). \forall \Phi. (\forall ss \ tt . \varphi \ \Phi \ (ss, tt) \longrightarrow \Phi \ (ss, tt)) \longrightarrow \Phi \ (s, t))$

definition *Monotone*:: $(TT \Rightarrow TT) \Rightarrow \text{bool}$

where $\text{Monotone } \varphi = (\forall \Phi \ \Psi . (\forall s \ t . \Phi(s, t) \longrightarrow \Psi(s, t)) \longrightarrow (\forall s \ t . \varphi \ \Phi \ (s, t) \longrightarrow \varphi \ \Psi \ (s, t)))$

For monotone invariant transformers φ , the construction indeed yields a fixed point.

lemma *Fix-lemma*: $\text{Monotone } \varphi \Longrightarrow \varphi \ (\text{FIX } \varphi) = \text{FIX } \varphi$

In order to derive the while rule we define the following transformer.

definition *PhiWhileOp*:: $BExpr \Rightarrow TT \Rightarrow TT \Rightarrow TT$

where $\text{PhiWhileOp } b \ \Phi = (\lambda \Psi . (\lambda (s, t). (\text{evalB } b \ t \longrightarrow (\exists r. \Phi(r, t) \wedge (\forall w. r \approx w \longrightarrow \Psi(s, w)))) \wedge (\neg \text{evalB } b \ t \longrightarrow s \approx t)))$

Since this operator is monotone, ...

lemma *PhiWhileOp-Monotone*: $\text{Monotone } (\text{PhiWhileOp } b \ \Phi)$

we may define its fixed point,

definition *PhiWhile*:: $BExpr \Rightarrow TT \Rightarrow TT$

where $\text{PhiWhile } b \ \Phi = \text{FIX } (\text{PhiWhileOp } b \ \Phi)$

which we can use to derive the following rule.

lemma *WHILE*:

$$\begin{aligned} & \llbracket (\forall s \ t. s \approx t \longrightarrow \text{evalB } b \ s = \text{evalB } b \ t); G \triangleright c : (\text{Sec } \Phi) \rrbracket \Longrightarrow \\ & G \triangleright (\text{While } b \ c) : (\text{Sec } (\text{PhiWhile } b \ \Phi)) \end{aligned}$$

The operator that given Φ returns the invariant occurring in the conclusion of the rule is itself monotone - this is the property required for the rule for procedure invocations.

lemma *PhiWhileMonotone*: *Monotone* $(\lambda \Phi . \text{PhiWhile } b \Phi)$

We now derive an alternative while rule that employs an inductive formulation of a variant that replaces the fixed point construction. This version is given in the paper.

First, the inductive definition of the *var* relation.

inductive-set *var*::(*BExpr* \times *TT* \times *State* \times *State*) *set*

where

varFalse: $\llbracket \neg \text{evalB } b \ t; \ s \approx t \rrbracket \implies (b, \Phi, s, t) : \text{var}$
 \mid *varTrue*: $\llbracket \text{evalB } b \ t; \ \Phi(r, t); \ \forall w . r \approx w \longrightarrow (b, \Phi, s, w) : \text{var} \rrbracket$
 $\implies (b, \Phi, s, t) : \text{var}$

It is easy to prove the equivalence of *var* and the fixed point:

lemma *FIXvarFIX*: $(\text{PhiWhile } b) = (\lambda \Phi . (\lambda (s, t) . (b, \Phi, s, t) : \text{var}))$

From this rule and the rule WHILE above, one may derive the while rule we gave in the paper.

lemma *WHILE-IND*:

$\llbracket (\forall s \ t. \ s \approx t \longrightarrow \text{evalB } b \ s = \text{evalB } b \ t); \ G \triangleright c : (\text{Sec } \Phi) \rrbracket \implies$
 $G \triangleright (\text{While } b \ c) : (\text{Sec } (\lambda (s, t) . (b, \Phi, s, t) : \text{var}))$

Not suprisingly, the construction *var* can be shown to be monotone in Φ .

lemma *var-Monotone*: *Monotone* $(\lambda \Phi . (\lambda (s, t) . (b, \Phi, s, t) : \text{var}))$

The call rule is formulated for an arbitrary fixed point of a monotone transformer.

lemma *CALL*:

$\llbracket (\{\text{Sec}(\text{FIX } \Phi)\} \cup G) \triangleright \text{body} : \text{Sec}(\Phi (\text{FIX } \Phi)); \ \text{Monotone } \Phi \rrbracket \implies$
 $G \triangleright \text{Call} : \text{Sec}(\text{FIX } \Phi)$

3.3 Derivation of the HIGH rules

The HIGH rules are easy.

lemma *HIGH-SKIP*: $G \triangleright \text{Skip} : \text{twiddle}$

lemma *HIGH-ASSIGN*:

CONTEXT $x = \text{high} \implies G \triangleright (\text{Assign } x \ e) : \text{twiddle}$

lemma *HIGH-COMP*:

$\llbracket G \triangleright c1 : \text{twiddle}; \ G \triangleright c2 : \text{twiddle} \rrbracket$
 $\implies G \triangleright (\text{Comp } c1 \ c2) : \text{twiddle}$

lemma *HIGH-IFF*:

$\llbracket G \triangleright c1 : \text{twiddle}; \ G \triangleright c2 : \text{twiddle} \rrbracket$

$\implies G \triangleright (\text{Iff } b \ c1 \ c2) : \text{twiddle}$

lemma HIGH-WHILE:

$\llbracket G \triangleright c : \text{twiddle} \rrbracket \implies G \triangleright (\text{While } b \ c) : \text{twiddle}$

lemma HIGH-CALL:

$(\{\text{twiddle}\} \cup G) \triangleright \text{body} : \text{twiddle} \implies G \triangleright \text{Call} : \text{twiddle}$

3.4 The type system of Volpano, Smith and Irvine

We now give the type system of Volpano et al. and then prove its embedding into the system of derived rules. First, type systems for expressions and boolean expressions.

inductive-set $VS\text{-expr} :: (\text{Expr} \times \text{TP}) \text{ set}$

where

$VS\text{-exprVar}: \text{CONTEXT } x = t \implies (\text{varE } x, t) : VS\text{-expr}$

| $VS\text{-exprVal}: (\text{valE } v, \text{low}) : VS\text{-expr}$

| $VS\text{-exprOp}: \llbracket (e1, t) : VS\text{-expr}; (e2, t) : VS\text{-expr} \rrbracket$

$\implies (\text{opE } f \ e1 \ e2, t) : VS\text{-expr}$

| $VS\text{-exprHigh}: (e, \text{high}) : VS\text{-expr}$

inductive-set $VS\text{-Bexpr} :: (\text{BExpr} \times \text{TP}) \text{ set}$

where

$VS\text{-BexprOp}: \llbracket (e1, t) : VS\text{-expr}; (e2, t) : VS\text{-expr} \rrbracket$

$\implies (\text{compB } f \ e1 \ e2, t) : VS\text{-Bexpr}$

| $VS\text{-BexprHigh}: (e, \text{high}) : VS\text{-Bexpr}$

Next, the core of the type system, the rules for commands.

inductive-set $VS\text{-com} :: (\text{TP} \times \text{IMP}) \text{ set}$

where

$VS\text{-comSkip}: (\text{pc}, \text{Skip}) : VS\text{-com}$

| $VS\text{-comAssHigh}:$

$\text{CONTEXT } x = \text{high} \implies (\text{pc}, \text{Assign } x \ e) : VS\text{-com}$

| $VS\text{-comAssLow}:$

$\llbracket \text{CONTEXT } x = \text{low}; \text{pc} = \text{low}; (e, \text{low}) : VS\text{-expr} \rrbracket \implies$
 $(\text{pc}, \text{Assign } x \ e) : VS\text{-com}$

| $VS\text{-comComp}:$

$\llbracket (\text{pc}, c1) : VS\text{-com}; (\text{pc}, c2) : VS\text{-com} \rrbracket \implies$
 $(\text{pc}, \text{Comp } c1 \ c2) : VS\text{-com}$

| $VS\text{-comIf}:$

$\llbracket (b, \text{pc}) : VS\text{-Bexpr}; (\text{pc}, c1) : VS\text{-com}; (\text{pc}, c2) : VS\text{-com} \rrbracket \implies$
 $(\text{pc}, \text{Iff } b \ c1 \ c2) : VS\text{-com}$

| $VS\text{-comWhile}:$

$\llbracket (b, \text{pc}) : VS\text{-Bexpr}; (\text{pc}, c) : VS\text{-com} \rrbracket \implies (\text{pc}, \text{While } b \ c) : VS\text{-com}$

| $VS\text{-comSub}: (high, c) : VS\text{-com} \implies (low, c) : VS\text{-com}$

We define the interpretation of expression typings...

primrec $SemExpr::Expr \Rightarrow TP \Rightarrow bool$

where

$SemExpr\ e\ low = (\forall\ s\ ss.\ s \approx ss \longrightarrow evalE\ e\ s = evalE\ e\ ss) \mid$
 $SemExpr\ e\ high = True$

... and show the soundness of the typing rules.

lemma $ExprSound: (e, tp) : VS\text{-expr} \implies SemExpr\ e\ tp$

Likewise for the boolean expressions.

primrec $SemBExpr::BExpr \Rightarrow TP \Rightarrow bool$

where

$SemBExpr\ b\ low = (\forall\ s\ ss.\ s \approx ss \longrightarrow evalB\ b\ s = evalB\ b\ ss) \mid$
 $SemBExpr\ b\ high = True$

lemma $BExprSound: (e, tp) : VS\text{-Bexpr} \implies SemBExpr\ e\ tp$

The proof of the main theorem (called Theorem 2 in our paper) proceeds by induction on $(t, c) : VS_com$.

theorem $VS\text{-com-VDM}[\text{rule-format}]$:

$(t, c) : VS\text{-com} \implies (t=high \longrightarrow G \triangleright c : twiddle) \wedge$
 $(t=low \longrightarrow (\exists\ A . G \triangleright c : Sec\ A))$

The semantic of typing judgements for commands is now the expected one: HIGH commands require initial and final state be low equivalent (i.e. the low variables in the final state can't depend on the high variables of the initial state), while LOW commands must respect the above mentioned security property.

primrec $SemCom::TP \Rightarrow IMP \Rightarrow bool$

where

$SemCom\ low\ c = (\forall\ s\ ss\ t\ tt.\ s \approx ss \longrightarrow (s, c \Downarrow t) \longrightarrow$
 $(ss, c \Downarrow tt) \longrightarrow t \approx tt) \mid$
 $SemCom\ high\ c = (\forall\ s\ t . (s, c \Downarrow t) \longrightarrow s \approx t)$

Combining theorem $VS\text{-com-VDM}$ with the soundness result of the program logic and the definition of validity yields the soundness of Volpano et al.'s type system.

theorem $VS\text{-SOUND}: (t, c) : VS\text{-com} \implies SemCom\ t\ c$

As a further minor result, we prove that all judgements interpreting the low rules indeed yield assertions A of the form $A = Sec(\Phi(FIX\Phi))$ for some monotone Φ .

inductive-set $Deriv :: (VDMAssn\ set \times IMP \times VDMAssn)\ set$

where

$D\text{-CAST}$:

$(G, c, twiddle) : Deriv \implies (G, c, Sec(\lambda(s, t) . s \approx t)) : Deriv$

| *D-SKIP*: $(G, \text{Skip}, \text{Sec } (\lambda (s,t) . s \approx t)) : \text{Deriv}$

| *D-ASSIGN*:
 $(\forall s \text{ ss}. s \approx \text{ss} \longrightarrow \text{evalE } e \ s = \text{evalE } e \ \text{ss}) \implies$
 $(G, \text{Assign } x \ e, \text{Sec } (\lambda (s, t) . s \approx (\text{update } t \ x \ (\text{evalE } e \ t)))) : \text{Deriv}$

| *D-COMP*:
 $\llbracket (G, c1, \text{Sec } \Phi) : \text{Deriv}; (G, c2, \text{Sec } \Psi) : \text{Deriv} \rrbracket \implies$
 $(G, \text{Comp } c1 \ c2, \text{Sec } (\lambda (s,t) . \exists r . \Phi(r, t) \wedge$
 $(\forall w . (r \approx w \longrightarrow \Psi(s, w)))) : \text{Deriv}$

| *C-IFF*:
 $\llbracket (\forall s \text{ ss}. s \approx \text{ss} \longrightarrow \text{evalB } b \ s = \text{evalB } b \ \text{ss});$
 $(G, c1, \text{Sec } \Phi) : \text{Deriv}; (G, c2, \text{Sec } \Psi) : \text{Deriv} \rrbracket \implies$
 $(G, \text{Iff } b \ c1 \ c2, \text{Sec } (\lambda (s, t) . (\text{evalB } b \ t \longrightarrow \Phi(s,t)) \wedge$
 $((\neg \text{evalB } b \ t) \longrightarrow \Psi(s,t)))) : \text{Deriv}$

| *D-WHILE*:
 $\llbracket (\forall s \text{ ss}. s \approx \text{ss} \longrightarrow \text{evalB } b \ s = \text{evalB } b \ \text{ss});$
 $(G, c, \text{Sec } \Phi) : \text{Deriv} \rrbracket \implies$
 $(G, \text{While } b \ c, \text{Sec } (\text{PhiWhile } b \ \Phi)) : \text{Deriv}$

| *D-CALL*:
 $\llbracket (\{\text{Sec}(\text{FIX } \Phi)\} \cup G, \text{body}, \text{Sec}(\Phi(\text{FIX } \Phi))) : \text{Deriv};$
 $\text{Monotone } \Phi \rrbracket \implies$
 $(G, \text{Call}, \text{Sec}(\text{FIX } \Phi)) : \text{Deriv}$

| *D-HighSKIP*: $(G, \text{Skip}, \text{twiddle}) : \text{Deriv}$

| *D-HighASSIGN*:
 $\text{CONTEXT } x = \text{high} \implies (G, \text{Assign } x \ e, \text{twiddle}) : \text{Deriv}$

| *D-HighCOMP*:
 $\llbracket (G, c1, \text{twiddle}) : \text{Deriv}; (G, c2, \text{twiddle}) : \text{Deriv} \rrbracket \implies$
 $(G, \text{Comp } c1 \ c2, \text{twiddle}) : \text{Deriv}$

| *D-HighIFF*:
 $\llbracket (G, c1, \text{twiddle}) : \text{Deriv}; (G, c2, \text{twiddle}) : \text{Deriv} \rrbracket \implies$
 $(G, \text{Iff } b \ c1 \ c2, \text{twiddle}) : \text{Deriv}$

| *D-HighWHILE*:
 $(G, c, \text{twiddle}) : \text{Deriv} \implies (G, \text{While } b \ c, \text{twiddle}) : \text{Deriv}$

| *D-HighCALL*:
 $(\{\text{twiddle}\} \cup G, \text{body}, \text{twiddle}) : \text{Deriv} \implies (G, \text{Call}, \text{twiddle}) : \text{Deriv}$

lemma *DerivMono*:

$(X, c, A) : \text{Deriv} \implies \exists \Phi . A = \text{Sec } (\Phi (\text{FIX } \Phi)) \wedge \text{Monotone } \Phi$

Also, all rules in the *Deriv* relation are indeed derivable in the program logic.

lemma *Deriv-derivable*: $(G, c, A): \text{Deriv} \implies G \triangleright c: A$

End of theory VS

end

theory *ContextVS* **imports** *VS* **begin**

3.5 Contextual closure

We show that the notion of security is closed w.r.t. low attacking contexts, i.e. contextual programs into which a secure program can be substituted and which itself employs only *obviously* low variables.

Contexts are **IMP** programs with (multiple) designated holes (represented by constructor *Ctxt_Here*).

datatype *CtxtProg* =
Ctxt-Hole
| *Ctxt-Skip*
| *Ctxt-Assign* *Var Expr*
| *Ctxt-Comp* *CtxtProg CtxtProg*
| *Ctxt-If* *BExpr CtxtProg CtxtProg*
| *Ctxt-While* *BExpr CtxtProg*
| *Ctxt-Call*

We let *C*, *D* range over contextual programs. The substitution operation is defined by structural recursion.

primrec *Fill*::*CtxtProg* \Rightarrow *IMP* \Rightarrow *IMP*
where
Fill Ctxt-Hole $c = c$ |
Fill Ctxt-Skip $c = \text{Skip}$ |
Fill (Ctxt-Assign x e) $c = \text{Assign } x \ e$ |
Fill (Ctxt-Comp C1 C2) $c = \text{Comp } (\text{Fill } C1 \ c) \ (\text{Fill } C2 \ c)$ |
Fill (Ctxt-If b C1 C2) $c = \text{Iff } b \ (\text{Fill } C1 \ c) \ (\text{Fill } C2 \ c)$ |
Fill (Ctxt-While b C) $c = \text{While } b \ (\text{Fill } C \ c)$ |
Fill Ctxt-Call $c = \text{Call}$

Equally obvious are the definitions of the (syntactically) mentioned variables of arithmetic and boolean expressions.

primrec *EVars*::*Expr* \Rightarrow *Var set*
where
EVars (varE x) = $\{x\}$ |
EVars (valE v) = $\{\}$ |
EVars (opE f e1 e2) = $\text{EVars } e1 \cup \text{EVars } e2$

lemma *low-Eval*[*rule-format*]:

$(\forall x . x \in EVars\ e \longrightarrow CONTEXT\ x = low) \longrightarrow$
 $(\forall s\ t . s \approx t \longrightarrow evalE\ e\ s = evalE\ e\ t)$

primrec $BVars::BExpr \Rightarrow Var\ set$

where

$BVars\ (compB\ f\ e1\ e2) = EVars\ e1 \cup EVars\ e2$

lemma $low-EvalB[rule-format]$:

$(\forall x . x \in BVars\ b \longrightarrow CONTEXT\ x = low) \longrightarrow$
 $(\forall s\ t . s \approx t \longrightarrow evalB\ b\ s = evalB\ b\ t)$

The variables possibly read from during the evaluation of c are denoted by $Vars\ c$. Note that in the clause for assignments the variable that is assigned to is not included in the set.

primrec $Vars::IMP \Rightarrow Var\ set$

where

$Vars\ Skip = \{\}$ |

$Vars\ (Assign\ x\ e) = EVars\ e$ |

$Vars\ (Comp\ c\ d) = Vars\ c \cup Vars\ d$ |

$Vars\ (While\ b\ c) = BVars\ b \cup Vars\ c$ |

$Vars\ (Iff\ b\ c\ d) = BVars\ b \cup Vars\ c \cup Vars\ d$ |

$Vars\ Call = \{\}$

For contexts, we define when a set X of variables is an upper bound for the variables read from.

primrec $CtxtVars::Var\ set \Rightarrow CtxtProg \Rightarrow bool$

where

$CtxtVars\ X\ Ctxt-Hole = True$ |

$CtxtVars\ X\ Ctxt-Skip = True$ |

$CtxtVars\ X\ (Ctxt-Assign\ x\ e) = (EVars\ e \subseteq X)$ |

$CtxtVars\ X\ (Ctxt-Comp\ C1\ C2) = (CtxtVars\ X\ C1 \wedge CtxtVars\ X\ C2)$ |

$CtxtVars\ X\ (Ctxt-If\ b\ C1\ C2) = (BVars\ b \subseteq X \wedge CtxtVars\ X\ C1$
 $\quad \wedge\ CtxtVars\ X\ C2)$ |

$CtxtVars\ X\ (Ctxt-While\ b\ C) = (BVars\ b \subseteq X \wedge CtxtVars\ X\ C)$ |

$CtxtVars\ X\ Ctxt-Call = True$

A constant representing the procedure body with holes.

consts $Ctxt-Body::CtxtProg$

The following predicate expresses that all variables read from by a command c are contained in the set X of low variables.

definition $LOW::Var\ set \Rightarrow CtxtProg \Rightarrow bool$

where $LOW\ X\ C = (CtxtVars\ X\ C \wedge (\forall x . x : X \longrightarrow CONTEXT\ x = low))$

By induction on the maximal height of the operational judgement (hidden in the definition of *secure*) we can prove that the security of c implies that of $Fill\ C\ c$, provided that the context and the procedure-context satisfy the LOW predicate for some X , and that the "real" body is obtained by substituting c into the procedure context.

lemma *secureI-secureFill*:

$\llbracket \text{secure } c; \text{LOW } X \ C; \text{LOW } X \ \text{Ctxt-Body}; \text{body} = \text{Fill } \text{Ctxt-Body } c \rrbracket$
 $\implies \text{secure } (\text{Fill } C \ c)$

Consequently, a (low) variable representing the result of the attacking context does not leak any unwanted information.

consts *res*::*Var*

theorem

$\llbracket \text{secure } c; \text{LOW } X \ C; \text{LOW } X \ \text{Ctxt-Body}; s \approx ss; s, (\text{Fill } C \ c) \Downarrow t;$
 $ss, (\text{Fill } C \ c) \Downarrow tt; \text{body} = \text{Fill } \text{Ctxt-Body } c; \text{CONTEXT } \text{res} = \text{low} \rrbracket$
 $\implies t \ \text{res} = tt \ \text{res}$

End of theory ContextVS

end

theory *Lattice* **imports** *Main* **begin**

4 Lattices

In preparation of the encoding of the type system of Hunt and Sands, we define some abstract type of lattices, together with the operations \perp , \sqsubseteq and \sqcup , and some obvious axioms.

typedecl *L*

axiomatization

bottom :: *L* **and**
LEQ :: *L* \Rightarrow *L* \Rightarrow *bool* **and**
LUB :: *L* \Rightarrow *L* \Rightarrow *L*

where

LAT1: *LEQ bottom p* **and**
LAT2: *LEQ p1 p2* \implies *LEQ p2 p3* \implies *LEQ p1 p3* **and**
LAT3: *LEQ p (LUB p q)* **and**
LAT4: *LUB p q = LUB q p* **and**
LAT5: *LUB p (LUB q r) = LUB (LUB p q) r* **and**
LAT6: *LEQ x x* **and**
LAT7: *p = LUB p p*

End of theory Lattice

end

theory *HuntSands* **imports** *VDM Lattice* **begin**

5 Flow-sensitivity a la Hunt and Sands

¹ The paper [4] by Hunt and Sands presents a generalisation of the type system of Volpano et al. to flow-sensitivity. Thus, programs such as $l := h; l := 5$ are not rejected any longer by the type system. Following the description in Section 4 of our paper [2], we embed Hunt and Sands' type system into the program logic given in Section 2.

5.1 General $A; R \Rightarrow S$ -security

Again, we define the type TT of intermediate formulae Φ , and an assertion operator Sec . The latter is now parametrised not only by the intermediate formulae but also by the (possibly differing) pre- and post-relations R and S (both instantiated to \approx in Section 3), and by a specification A that directly links pre- and post-states.

type-synonym $TT = (State \times State) \Rightarrow bool$

definition $RSsecure :: (State \Rightarrow State \Rightarrow bool) \Rightarrow (State \Rightarrow State \Rightarrow bool) \Rightarrow IMP \Rightarrow bool$
where $RSsecure R S c = (\forall s t ss tt . R s t \longrightarrow (s, c \Downarrow ss) \longrightarrow (t, c \Downarrow tt) \longrightarrow S ss tt)$

definition $ARSsecure :: VDMAssn \Rightarrow (State \Rightarrow State \Rightarrow bool) \Rightarrow (State \Rightarrow State \Rightarrow bool) \Rightarrow IMP \Rightarrow bool$
where $ARSsecure A R S c = ((\models c : A) \wedge RSsecure R S c)$

Definition 3 of our paper follows.

definition $Sec :: VDMAssn \Rightarrow (State \Rightarrow State \Rightarrow bool) \Rightarrow (State \Rightarrow State \Rightarrow bool) \Rightarrow TT \Rightarrow VDMAssn$
where $Sec A R S \Phi s t = (A s t \wedge (\forall r . R s r \longrightarrow \Phi(t, r)) \wedge (\forall r . \Phi(r, s) \longrightarrow S r t))$

With these definitions, we can prove Proposition 4 of our paper.

lemma $Prop4A : \models c : Sec A R S \Phi \Longrightarrow ARSsecure A R S c$

lemma $Prop4B : ARSsecure A R S c \Longrightarrow \models c : Sec A R S (\lambda (r, t) . \exists s . (s, c \Downarrow r) \wedge R s t)$

5.2 Basic definitions

Contexts map program variables to lattice elements.

type-synonym $CONTEXT = Var \Rightarrow L$

definition $upd :: CONTEXT \Rightarrow Var \Rightarrow L \Rightarrow CONTEXT$
where $upd G x p = (\lambda y . \text{if } x=y \text{ then } p \text{ else } G y)$

¹As the Isabelle theory representing this section is dependent only on VDM.thy and Lattice.thy, name conflicts with notions defined in Section 3 are avoided.

We also define the predicate EQ which expresses when two states agree on all variables whose entry in a given context is below a certain security level.

definition $EQ::CONTEXT \Rightarrow L \Rightarrow State \Rightarrow State \Rightarrow bool$
where $EQ\ G\ p = (\lambda\ s\ t . \forall\ x . LEQ\ (G\ x)\ p \longrightarrow s\ x = t\ x)$

lemma $EQ\text{-}LEQ: \llbracket EQ\ G\ p\ s\ t; LEQ\ pp\ p \rrbracket \Longrightarrow EQ\ G\ pp\ s\ t$

The assertion called Q in our paper:

definition $Q::L \Rightarrow CONTEXT \Rightarrow VDMAssn$
where $Q\ p\ H = (\lambda\ s\ t . \forall\ x . (\neg\ LEQ\ p\ (H\ x)) \longrightarrow t\ x = s\ x)$

Q expresses the preservation of values in a single execution, and corresponds to the first clause of Definition 3.2 in [4]. In accordance with this, the following definition of security instantiates the A position of $A; R \Rightarrow S$ -security with Q , while the context-dependent binary state relations are plugged in as the R and S components.

definition $secure::L \Rightarrow CONTEXT \Rightarrow IMP \Rightarrow CONTEXT \Rightarrow bool$
where $secure\ p\ G\ c\ H = (\forall\ q . ARSsecure\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ c)$

Indeed, one may show that this notion of security amounts to the conjunction of a unary (i.e. one-execution-)property and a binary (i.e. two-execution-) property, as expressed in Hunt & Sands' Definition 3.2.

definition $secure1::L \Rightarrow CONTEXT \Rightarrow IMP \Rightarrow CONTEXT \Rightarrow bool$
where $secure1\ p\ G\ c\ H = (\forall\ s\ t . (s,c \Downarrow t) \longrightarrow Q\ p\ H\ s\ t)$

definition $secure2::L \Rightarrow CONTEXT \Rightarrow IMP \Rightarrow CONTEXT \Rightarrow bool$
where $secure2\ p\ G\ c\ H = ((\forall\ s\ t\ ss\ tt . (s,c \Downarrow t) \longrightarrow (ss,c \Downarrow tt) \longrightarrow EQ\ G\ p\ s\ ss \longrightarrow EQ\ H\ p\ t\ tt))$

lemma $secureEQUIV:$
 $secure\ p\ G\ c\ H = (\forall\ q . secure1\ p\ G\ c\ H \wedge secure2\ q\ G\ c\ H)$

5.3 Type system

The type system of Hunt and Sands – our language formalisation uses a concrete datatype of expressions, so we add the obvious typing rules for expressions and prove the expected evaluation lemmas.

inductive-set $HS\text{-}E::(CONTEXT \times Expr \times L)\ set$
where
 $HS\text{-}E\text{-}var: (G, varE\ x, G\ x) : HS\text{-}E$
 $| HS\text{-}E\text{-}val: (G, valE\ c, bottom) : HS\text{-}E$
 $| HS\text{-}E\text{-}op: \llbracket (G, e1, p1):HS\text{-}E; (G, e2, p2):HS\text{-}E; p = LUB\ p1\ p2 \rrbracket$
 $\implies (G, opE\ f\ e1\ e2, p) : HS\text{-}E$
 $| HS\text{-}E\text{-}sup: \llbracket (G, e, p):HS\text{-}E; LEQ\ p\ q \rrbracket \implies (G, e, q):HS\text{-}E$

lemma $HS\text{-}E\text{-}eval[rule\text{-}format]:$

$$(G, e, t) \in HS-E \implies \\ \forall r s q. EQ G q r s \longrightarrow LEQ t q \longrightarrow evalE e r = evalE e s$$

Likewise for boolean expressions:

inductive-set $HS-B:: (CONTEXT \times BExpr \times L) \text{ set}$

where

$$HS-B-compB: \llbracket (G, e1, p1):HS-E; (G, e2, p2):HS-E; p = LUB p1 p2 \rrbracket \\ \implies (G, compB f e1 e2, p) : HS-B$$

$$| HS-B-sup: \llbracket (G, b, p):HS-B; LEQ p q \rrbracket \implies (G, b, q):HS-B$$

lemma $HS-B-eval[rule-format]$:

$$(G, b, t) \in HS-B \implies$$

$$\forall r s pp. EQ G pp r s \longrightarrow LEQ t pp \longrightarrow evalB b r = evalB b s$$

The typing rules for commands follow.

inductive-set $HS:: (L \times CONTEXT \times IMP \times CONTEXT) \text{ set}$

where

$$HS-Skip: (p, G, Skip, G):HS$$

| $HS-Assign$:

$$(G, e, t):HS-E \implies (p, G, Assign x e, upd G x (LUB p t)):HS$$

| $HS-Seq$:

$$\llbracket (p, G, c, K):HS; (p, K, d, H):HS \rrbracket \implies (p, G, Comp c d, H):HS$$

| $HS-If$:

$$\llbracket (G, b, t):HS-B; (LUB p t, G, c, H):HS; (LUB p t, G, d, H):HS \rrbracket \implies \\ (p, G, Iff b c d, H):HS$$

| $HS-If-alg$:

$$\llbracket (G, b, p):HS-B; (p, G, c, H):HS; (p, G, d, H):HS \rrbracket \implies \\ (p, G, Iff b c d, H):HS$$

| $HS-While$:

$$\llbracket (G, b, t):HS-B; (LUB p t, G, c, H):HS; H = G \rrbracket \implies \\ (p, G, While b c, H):HS$$

| $HS-Sub$:

$$\llbracket (pp, GG, c, HH):HS; LEQ p pp; \forall x. LEQ (G x) (GG x); \\ \forall x. LEQ (HH x) (H x) \rrbracket \implies \\ (p, G, c, H):HS$$

Using $HS-Sub$, rules If and $If-alg$ are inter-derivable.

lemma $IF-derivable-from-If-alg$:

$$\llbracket (G, b, t):HS-B; (LUB p t, G, c1, H):HS; (LUB p t, G, c2, H):HS \rrbracket \\ \implies (p, G, Iff b c1 c2, H):HS$$

apply ($subgoal-tac (LUB p t, G, Iff b c1 c2, H):HS$)

apply ($erule HS-Sub$) **apply** ($rule LAT3$)

apply ($clarsimp, rule LAT6$) **apply** ($clarsimp, rule LAT6$)

apply (rule *HS-If-alg*) **apply** (erule *HS-B-sup*)
apply (subgoal-tac *LEQ t (LUB t p), simp add: LAT4*)
apply (rule *LAT3*) **apply** *assumption+*
done

lemma *IF-alg-derivable-from-If*:
 $\llbracket (G, b, p) : HS-B; (p, G, c1, H) : HS; (p, G, c2, H) : HS \rrbracket$
 $\implies (p, G, \text{Iff } b \ c1 \ c2, H) : HS$
apply (erule *HS-If*) **apply** (subgoal-tac *LUB p p = p, clarsimp*)
apply (subgoal-tac *p = LUB p p, fastforce*) **apply** (rule *LAT7*)
apply (subgoal-tac *LUB p p = p, clarsimp*)
apply (subgoal-tac *p = LUB p p, fastforce*) **apply** (rule *LAT7*)
done

An easy induction on typing derivations shows the following property.

lemma *HS-Aux1*:
 $(p, G, c, H) : HS \implies \forall x. LEQ (G x) (H x) \vee LEQ p (H x)$

5.4 Derived proof rules

In order to show the derivability of the properties given in Theorem 3.3 of Hunt and Sands' paper, we give the following derived proof rules. By including the *Q* property in the *A* position of *Sec*, we prove both parts of theorem in one proof, and can exploit the first property (*Q*) in the proof of the second.

lemma *SKIP*:
 $X \triangleright \text{Skip} : \text{Sec} (Q p H) (EQ G q) (EQ G q)$
 $(\lambda (s, t) . EQ G q s t)$

lemma *ASSIGN*:
 $\llbracket H = \text{upd } G x (LUB p t);$
 $\forall s ss . EQ G t s ss \longrightarrow \text{evalE } e s = \text{evalE } e ss \rrbracket$
 $\implies X \triangleright \text{Assign } x e : \text{Sec} (Q p H) (EQ G q) (EQ H q)$
 $(\lambda (s, t) . \exists r . s = \text{update } r x (\text{evalE } e r) \wedge EQ G q r t)$

lemma *COMP*:
 $\llbracket X \triangleright c1 : \text{Sec} (Q p K) (EQ G q) (EQ K q) \Phi;$
 $X \triangleright c2 : \text{Sec} (Q p H) (EQ K q) (EQ H q) \Psi;$
 $\forall x . LEQ (G x) (K x) \vee LEQ p (K x);$
 $\forall x . LEQ (K x) (H x) \vee LEQ p (H x) \rrbracket$
 $\implies X \triangleright \text{Comp } c1 \ c2 : \text{Sec} (Q p H) (EQ G q) (EQ H q)$
 $(\lambda (x, y) . \exists z . \Phi (z, y) \wedge$
 $(\forall w . EQ K q z w \longrightarrow \Psi (x, w)))$

We distinguish, for any given *q*, *parallel* conditionals from *diagonal* ones. Speaking operationally (i.e. in terms of two executions), conditionals of the former kind evaluate the branch condition identically in both executions. The following rule expresses this condition explicitly, in the first side condition. The formula inside the *Sec*-operator of the conclusion resembles the conclusion of the VDM rule for conditionals in that the formula chosen

depends on the outcome of the branch.

lemma IF-PARALLEL:

$$\begin{aligned}
& \llbracket \forall s \text{ ss} . EQ \ G \ p \ s \ \text{ss} \longrightarrow evalB \ b \ s = evalB \ b \ \text{ss}; \\
& \quad \forall x . LEQ \ (G \ x) \ (H \ x) \vee LEQ \ p \ (H \ x); \\
& \quad \exists x . LEQ \ p \ (H \ x) \wedge LEQ \ (H \ x) \ q; \\
& \quad X \triangleright c1 : Sec \ (Q \ p \ H) \ (EQ \ G \ q) \ (EQ \ H \ q) \ \Phi; \\
& \quad X \triangleright c2 : Sec \ (Q \ p \ H) \ (EQ \ G \ q) \ (EQ \ H \ q) \ \Psi \rrbracket \\
\implies & X \triangleright Iff \ b \ c1 \ c2 : Sec \ (Q \ p \ H) \ (EQ \ G \ q) \ (EQ \ H \ q) \\
& \quad (\lambda \ (r, u) . (evalB \ b \ u \longrightarrow \Phi \ (r, u)) \wedge \\
& \quad \quad (\neg \ evalB \ b \ u \longrightarrow \Psi \ (r, u)))
\end{aligned}$$

An alternative formulation replaces the first side condition with a typing hypothesis on the branch condition, thus exploiting lemma HS_B_eval.

lemma IF-PARALLEL-tp:

$$\begin{aligned}
& \llbracket (G, b, p) \in HS\text{-}B; (p, G, c1, H) \in HS; (p, G, c2, H) \in HS; \\
& \quad \exists x . LEQ \ p \ (H \ x) \wedge LEQ \ (H \ x) \ q; \\
& \quad X \triangleright c1 : Sec \ (Q \ p \ H) \ (EQ \ G \ q) \ (EQ \ H \ q) \ \Phi; \\
& \quad X \triangleright c2 : Sec \ (Q \ p \ H) \ (EQ \ G \ q) \ (EQ \ H \ q) \ \Psi \rrbracket \\
\implies & X \triangleright Iff \ b \ c1 \ c2 : Sec \ (Q \ p \ H) \ (EQ \ G \ q) \ (EQ \ H \ q) \\
& \quad (\lambda \ (r, u) . (evalB \ b \ u \longrightarrow \Phi \ (r, u)) \wedge \\
& \quad \quad (\neg \ evalB \ b \ u \longrightarrow \Psi \ (r, u)))
\end{aligned}$$

Diagonal conditionals, in contrast, capture cases where (from the perspective of an observer at level q) the two executions may evaluate the branch condition differently. In this case, the formula inside the *Sec*-operator in the conclusion cannot depend upon the branch outcome, so the least common denominator of the two branches must be taken, which is given by the equality condition w.r.t. the post-context H . A side condition (the first one given in the rule) ensures that indeed no information leaks during the execution of either branch, by relating G and H .

lemma IF-DIAGONAL:

$$\begin{aligned}
& \llbracket \forall x . LEQ \ (G \ x) \ (H \ x) \vee LEQ \ p \ (H \ x); \\
& \quad \neg (\exists x . LEQ \ p \ (H \ x) \wedge LEQ \ (H \ x) \ q); \\
& \quad X \triangleright c1 : Sec \ (Q \ p \ H) \ (EQ \ G \ q) \ (EQ \ H \ q) \ \Phi; \\
& \quad X \triangleright c2 : Sec \ (Q \ p \ H) \ (EQ \ G \ q) \ (EQ \ H \ q) \ \Psi \rrbracket \\
\implies & X \triangleright Iff \ b \ c1 \ c2 : Sec \ (Q \ p \ H) \ (EQ \ G \ q) \ (EQ \ H \ q) \\
& \quad (\lambda \ (s, t) . EQ \ H \ q \ s \ t)
\end{aligned}$$

Again, the first side condition of the rule may be replaced by a typing condition, but now this condition is on the commands (instead of the branch condition) – in fact, a derivation for either branch suffices.

lemma IF-DIAGONAL-tp:

$$\begin{aligned}
& \llbracket (p, G, c1, H) \in HS \vee (p, G, c2, H) \in HS; \\
& \quad \neg (\exists x . LEQ \ p \ (H \ x) \wedge LEQ \ (H \ x) \ q); \\
& \quad X \triangleright c1 : Sec \ (Q \ p \ H) \ (EQ \ G \ q) \ (EQ \ H \ q) \ \Phi; \\
& \quad X \triangleright c2 : Sec \ (Q \ p \ H) \ (EQ \ G \ q) \ (EQ \ H \ q) \ \Psi \rrbracket \\
\implies & X \triangleright Iff \ b \ c1 \ c2 : Sec \ (Q \ p \ H) \ (EQ \ G \ q) \ (EQ \ H \ q) \\
& \quad (\lambda \ (s, t) . EQ \ H \ q \ s \ t)
\end{aligned}$$

Obviously, given q , any conditional is either parallel or diagonal as the second side conditions of the diagonal rules and the parallel rules are exclusive.

lemma *if-algorithmic*:

$$\begin{aligned} & \llbracket \exists x . LEQ p (H x) \wedge LEQ (H x) q; \\ & \quad \neg (\exists x . LEQ p (H x) \wedge LEQ (H x) q) \rrbracket \\ & \implies False \end{aligned}$$

As in Section 3 we define a fixed point construction, useful for the (parallel) while rule.

definition *FIX*:: $(TT \Rightarrow TT) \Rightarrow TT$

where *FIX* $\varphi = (\lambda (s,t) . \forall \Phi . (\forall ss tt . \varphi \Phi (ss, tt) \longrightarrow \Phi (ss, tt)) \longrightarrow \Phi (s, t))$

For monotone invariant transformers, the construction indeed yields a fixed point.

definition *Monotone*:: $(TT \Rightarrow TT) \Rightarrow bool$

where *Monotone* $\varphi = (\forall \Phi \Psi . (\forall s t . \Phi(s,t) \longrightarrow \Psi(s,t)) \longrightarrow (\forall s t . \varphi \Phi (s,t) \longrightarrow \varphi \Psi (s,t)))$

lemma *Fix-lemma*: $Monotone \varphi \implies \varphi (FIX \varphi) = FIX \varphi$

Next, the definition of a while-operator.

definition *PhiWhilePOp*::

$$VDMAssn \Rightarrow BExpr \Rightarrow TT \Rightarrow TT \Rightarrow TT$$

where *PhiWhilePOp* $A b \Phi =$

$$\begin{aligned} & (\lambda \Psi . (\lambda (r, u) . (evalB b u \longrightarrow (\exists z . \Phi (z, u) \wedge \\ & \quad (\forall w . A z w \longrightarrow \Psi (r, w)))))) \wedge \\ & ((\neg evalB b u) \longrightarrow A r u)) \end{aligned}$$

This operator is monotone in Φ .

lemma *PhiWhilePOp-Monotone*: $Monotone (PhiWhilePOp A b \Phi)$

Therefore, we can define the following fixed point.

definition *PhiWhileP*:: $VDMAssn \Rightarrow BExpr \Rightarrow TT \Rightarrow TT$

where *PhiWhileP* $A b \Phi = FIX (PhiWhilePOp A b \Phi)$

As as a function on ϕ , this PhiWhileP is itself monotone in ϕ :

lemma *PhiWhilePMonotone*: $Monotone (\lambda \Phi . PhiWhileP A b \Phi)$

Now the rule for parallel while loops, i.e. loops where the branch condition evaluates identically in both executions.

lemma *WHILE-PARALLEL*:

$$\begin{aligned} & \llbracket X \triangleright c : Sec (Q p G) (EQ G q) (EQ G q) \Phi; \\ & \quad \forall s ss . EQ G p s ss \longrightarrow evalB b s = evalB b ss; LEQ p q \rrbracket \\ & \implies X \triangleright While b c : Sec (Q p G) (EQ G q) (EQ G q) \\ & \quad (PhiWhileP (EQ G q) b \Phi) \end{aligned}$$

The side condition regarding the evaluation of the branch condition may be replaced by a typing hypothesis, thanks to lemma *HS-B-eval*.

lemma *WHILE-PARALLEL-tp*:

$$\begin{aligned} & \llbracket X \triangleright c : \text{Sec } (Q \ p \ G) \ (EQ \ G \ q) \ (EQ \ G \ q) \ \Phi; \\ & \quad (G, b, p) \in \text{HS-B}; \text{LEQ } p \ q \rrbracket \\ \implies & X \triangleright \text{While } b \ c : \text{Sec } (Q \ p \ G) \ (EQ \ G \ q) \ (EQ \ G \ q) \\ & \quad (\text{PhiWhileP } (EQ \ G \ q) \ b \ \Phi) \end{aligned}$$

One may also give an inductive formulation of FIX:

inductive-set *var::(BExpr × VDMAssn × TT × State × State) set*

where

varFalse:

$$\llbracket \neg \text{evalB } b \ t; \ A \ s \ t \rrbracket \implies (b, A, \Phi, s, t):var$$

| *varTrue*:

$$\begin{aligned} & \llbracket \text{evalB } b \ t; \ \Phi(r, t); \ (\forall w . A \ r \ w \longrightarrow \\ & \quad (b, A, \Phi, s, w): var) \rrbracket \implies (b, A, \Phi, s, t):var \end{aligned}$$

The inductive formulation and the fixed point formulation are equivalent.

lemma *FIXvarFIX*:

$$\text{PhiWhileP } A \ b = (\lambda \Phi . (\lambda (s, t) . (b, A, \Phi, s, t):var))$$

Thus, the above while rule may also be written using the inductive formulation.

lemma *WHILE-PARALLEL-IND*:

$$\begin{aligned} & \llbracket X \triangleright c : \text{Sec } (Q \ p \ G) \ (EQ \ G \ q) \ (EQ \ G \ q) \ \Phi; \\ & \quad \forall s \ ss . EQ \ G \ p \ s \ ss \longrightarrow \text{evalB } b \ s = \text{evalB } b \ ss; \text{LEQ } p \ q \rrbracket \implies \\ & X \triangleright \text{While } b \ c : (\text{Sec } (Q \ p \ G) \ (EQ \ G \ q) \ (EQ \ G \ q) \\ & \quad (\lambda (s, t) . (b, EQ \ G \ q, \Phi, s, t):var)) \end{aligned}$$

Again, we may replace the side condition regarding the branch condition by a typing hypothesis.

lemma *WHILE-PARALLEL-IND-tp*:

$$\begin{aligned} & \llbracket X \triangleright c : \text{Sec } (Q \ p \ G) \ (EQ \ G \ q) \ (EQ \ G \ q) \ \Phi; \\ & \quad (G, b, p) \in \text{HS-B}; \text{LEQ } p \ q \rrbracket \implies \\ & X \triangleright (\text{While } b \ c) : \\ & \quad (\text{Sec } (Q \ p \ G) \ (EQ \ G \ q) \ (EQ \ G \ q) \ (\lambda (s, t) . (b, EQ \ G \ q, \Phi, s, t):var)) \end{aligned}$$

Of course, the inductive formulation is also monotone:

lemma *var-MonotoneInPhi*:

$$\text{Monotone } (\lambda \Phi . (\lambda (s, t) . (b, A, \Phi, s, t):var))$$

In order to derive a diagonal while rule, we directly define an inductive relation that calculates the transitive closure of relation A , such that all but the last state evaluate b to *True*.

inductive-set *varD::(BExpr × VDMAssn × State × State) set*

where

$$\text{varDFalse: } \llbracket \neg \text{evalB } b \ s; \ A \ s \ t \rrbracket \implies (b, A, s, t):varD$$

$$\text{varDTrue: } \llbracket \text{evalB } b \ s; \ A \ s \ w; \ (b, A, w, t): varD \rrbracket \implies (b, A, s, t):varD$$

Here is the obvious definition of transitivity for assertions.

definition *transitive*:: $VDMAssn \Rightarrow bool$
where *transitive* $P = (\forall x y z . P x y \longrightarrow P y z \longrightarrow P x z)$

The inductive relation satisfies the following property.

lemma *varD-transitive*[*rule-format*]:
 $(b, A, s, t):varD \Longrightarrow transitive A \longrightarrow A s t$

On the other hand, the assertion Q defined above is transitive,

lemma *Q-transitive*: $transitive (Q q G)$

and is hence respected by the inductive closure:

lemma *varDQ*: $(b, Q q G, s, t):varD \Longrightarrow Q q G s t$

The diagonal while rule has a conclusion that is independent of ϕ .

lemma *WHILE-DIAGONAL*:
 $\llbracket X \triangleright c : Sec (Q p G) (EQ G q) (EQ G q) \Phi; \neg LEQ p q \rrbracket$
 $\Longrightarrow X \triangleright While b c : Sec (Q p G) (EQ G q) (EQ G q)$
 $(\lambda (s, t). EQ G q s t)$

varD is monotone in the assertion position.

lemma *varDMonotoneInAssertion*[*rule-format*]:
 $(b, A, s, t) \in varD \Longrightarrow$
 $(\forall s t. A s t \longrightarrow B s t) \longrightarrow (b, B, s, t) \in varD$

Finally, the subsumption rule.

lemma *SUB*:
 $\llbracket LEQ p pp; \forall x. LEQ (G x) (GG x); \forall x. LEQ (HH x) (H x);$
 $X \triangleright c : Sec (Q pp HH) (EQ GG q) (EQ HH q) \Phi \rrbracket$
 $\Longrightarrow X \triangleright c : Sec (Q p H) (EQ G q) (EQ H q) \Phi$

5.5 Soundness results

An induction on the typing rules now proves the main theorem which was called Theorem 4 in [2].

theorem *Theorem4*[*rule-format*]:
 $(p, G, c, H):HS \Longrightarrow$
 $(\exists \Phi . X \triangleright c : (Sec (Q p H) (EQ G q) (EQ H q) \Phi))$

By the construction of the operator *Sec* (lemmas *Prop4A* and *Prop4A* in Section 5.1) we obtain the soundness property with respect to the oprational semantics, i.e. the result stated as Theorem 3.3 in [4].

theorem *HuntSands33*: $(p, G, c, H):HS \Longrightarrow secure p G c H$

Both parts of this theorem may also be shown individually. We factor both proofs by the program logic.

lemma *Sec1-deriv*: $(p, G, c, H):HS \Longrightarrow X \triangleright c : (Q p H)$

theorem *HuntSands33-1*: $(p, G, c, H):HS \Longrightarrow secure1 p G c H$

lemma *Sec2-deriv*:

$(p, G, c, H) : HS \implies$
 $(\exists A . X \triangleright c : (Sec (Q p H) (EQ G q) (EQ H q) A))$
theorem *HuntSands33-2*: $(p, G, c, H) : HS \implies secure2 q G c H$

Again, the call rule is formulated for an arbitrary fixed point of a monotone transformer.

lemma *CALL*:

$\llbracket (\{B\} \cup X) \triangleright body : Sec A R S (\varphi(FIX \varphi));$
 $Monotone \varphi; B = Sec A R S (FIX \varphi) \rrbracket$
 $\implies X \triangleright Call : B$

As in Section 3, we define a formal derivation system comprising all derived rules and show that all derivable judgements are of the for $Sec(\Phi)$ for some monotone Φ .

inductive-set *Deriv*:: $(VDMAssn\ set \times IMP \times VDMAssn)\ set$

where

D-SKIP:

$\Omega = (\lambda (s, t). EQ G q s t)$
 $\implies (X, Skip, Sec (Q p H) (EQ G q) (EQ G q) \Omega) : Deriv$

| *D-ASSIGN*:

$\llbracket H = upd G x (LUB p t);$
 $\forall s\ ss . EQ G t s\ ss \longrightarrow evalE e s = evalE e ss;$
 $\Omega = (\lambda (s, t) . \exists r . s = update r x (evalE e r) \wedge EQ G q r t) \rrbracket$
 $\implies (X, Assign x e, Sec (Q p H) (EQ G q) (EQ H q) \Omega) : Deriv$

| *D-COMP*:

$\llbracket (X, c, Sec (Q p K) (EQ G q) (EQ K q) \Phi) : Deriv;$
 $(X, d, Sec (Q p H) (EQ K q) (EQ H q) \Psi) : Deriv;$
 $\forall x . LEQ (G x) (K x) \vee LEQ p (K x);$
 $\forall x . LEQ (K x) (H x) \vee LEQ p (H x);$
 $\Omega = (\lambda (x, y) . \exists z . \Phi(z, y) \wedge (\forall w . EQ K q z w \longrightarrow \Psi(x, w))) \rrbracket$
 $\implies (X, Comp c d, Sec (Q p H) (EQ G q) (EQ H q) \Omega) : Deriv$

| *D-IF-PARALLEL*:

$\llbracket \forall s\ ss . EQ G p s\ ss \longrightarrow evalB b s = evalB b ss;$
 $\forall x . LEQ (G x) (H x) \vee LEQ p (H x);$
 $\exists x . LEQ p (H x) \wedge LEQ (H x) q;$
 $(X, c, Sec (Q p H) (EQ G q) (EQ H q) \Phi) : Deriv;$
 $(X, d, Sec (Q p H) (EQ G q) (EQ H q) \Psi) : Deriv;$
 $\Omega = (\lambda (r, u) . (evalB b u \longrightarrow \Phi(r, u) \wedge$
 $(\neg evalB b u) \longrightarrow \Psi(r, u))) \rrbracket$
 $\implies (X, Iff b c d, Sec (Q p H) (EQ G q) (EQ H q) \Omega) : Deriv$

| *D-IF-DIAGONAL*:

$\llbracket \forall x . LEQ (G x) (H x) \vee LEQ p (H x);$
 $\neg (\exists x . LEQ p (H x) \wedge LEQ (H x) q);$
 $(X, c, Sec (Q p H) (EQ G q) (EQ H q) \Phi) : Deriv;$
 $(X, d, Sec (Q p H) (EQ G q) (EQ H q) \Psi) : Deriv;$

$$\begin{aligned} \Omega &= (\lambda (s,t) . EQ H q s t) \\ \implies (X, \text{Iff } b c d, \text{Sec } (Q p H) (EQ G q) (EQ H q) \Omega) : \text{Deriv} \end{aligned}$$

| *D-WHILE-PARALLEL*:

$$\begin{aligned} &\llbracket (X, c, \text{Sec } (Q p G) (EQ G q) (EQ G q) \Phi) : \text{Deriv}; \\ &\quad \forall s ss . EQ G p s ss \longrightarrow \text{eval} B b s = \text{eval} B b ss; LEQ p q; \\ &\quad \Omega = (\lambda (s,t) . (b, EQ G q, \Phi, s, t) : \text{var}) \rrbracket \\ \implies (X, \text{While } b c, \text{Sec } (Q p G) (EQ G q) (EQ G q) \Omega) : \text{Deriv} \end{aligned}$$

| *D-WHILE-DIAGONAL*:

$$\begin{aligned} &\llbracket (X, c, \text{Sec } (Q p G) (EQ G q) (EQ G q) \Phi) : \text{Deriv}; \neg LEQ p q; \\ &\quad \Omega = (\lambda (s,t) . EQ G q s t) \rrbracket \\ \implies (X, \text{While } b c, \text{Sec } (Q p G) (EQ G q) (EQ G q) \Omega) : \text{Deriv} \end{aligned}$$

| *D-SUB*:

$$\begin{aligned} &\llbracket LEQ p pp; \forall x . LEQ (G x) (GG x); \forall x . LEQ (HH x) (H x); \\ &\quad (X, c, \text{Sec } (Q pp HH) (EQ GG q) (EQ HH q) \Phi) : \text{Deriv} \rrbracket \\ \implies (X, c, \text{Sec } (Q p H) (EQ G q) (EQ H q) \Phi) : \text{Deriv} \end{aligned}$$

| *D-CALL*:

$$(\{A\} \cup X, \text{body}, A) : \text{Deriv} \implies (X, \text{Call}, A) : \text{Deriv}$$

lemma *DerivMono*:

$$\begin{aligned} &(X, c, B) : \text{Deriv} \implies \\ &\quad \exists A R S \varphi . B = \text{Sec } A R S (\varphi (\text{FIX } \varphi)) \wedge \text{Monotone } \varphi \end{aligned}$$

Also, the *Deriv* is indeed a subsystem of the program logic.

theorem *Deriv-derivable*: $(X, c, A) : \text{Deriv} \implies X \triangleright c : A$

End of theory HuntSands

end

theory *OBJ* **imports** *Main* **begin**

6 Base-line non-interference with objects

We now extend the encoding for base-line non-interference to a language with objects. The development follows the structure of Sections 1 to 3. Syntax and operational semantics are defined in Section 6.1, the axiomatic semantics in Section 6.2. The generalised definition of non-interference is given in 6.4, the derived proof rules in Section 6.5, and a type system in the style of Volpano et al. in Section 6.6. Finally, Section 6.7 concludes with results on contextual closure.

6.1 Syntax and operational semantics

First, some operations for association lists

primrec $lookup :: ('a \times 'b) list \Rightarrow 'a \Rightarrow 'b option$
where
 $lookup [] l = None$ |
 $lookup (h \# t) l = (if (fst h)=l then Some (snd h) else lookup t l)$

definition $Dom :: ('a \times 'b) list \Rightarrow 'a set$
where $Dom L = \{l . \exists a . lookup L l = Some a\}$

Abstract types of variables, class names, field names, and locations.

typedecl Var
typedecl $Class$
typedecl $Field$
typedecl $Location$

References are either null or a location. Values are either integers or references.

datatype $Ref = Nullref | Loc Location$

datatype $Val = RVal Ref | IVal int$

The heap is a finite map from locations to objects. Objects have a dynamic class and a field map.

type-synonym $Object = Class \times ((Field \times Val) list)$
type-synonym $Heap = (Location \times Object) list$

Stores contain values for all variables, and states are pairs of stores and heaps.

type-synonym $Store = Var \Rightarrow Val$

definition $update :: Store \Rightarrow Var \Rightarrow Val \Rightarrow Store$
where $update s x v = (\lambda y . if x=y then v else s y)$

type-synonym $State = Store \times Heap$

Arithmetic and boolean expressions are as before.

datatype $Expr =$
 $varE Var$
 $| valE Val$
 $| opE Val \Rightarrow Val \Rightarrow Val Expr Expr$

datatype $BExpr = compB Val \Rightarrow Val \Rightarrow bool Expr Expr$

The same applies to their semantics.

primrec $evalE :: Expr \Rightarrow Store \Rightarrow Val$
where
 $evalE (varE x) s = s x$ |
 $evalE (valE v) s = v$ |
 $evalE (opE f e1 e2) s = f (evalE e1 s) (evalE e2 s)$

primrec $evalB::BExpr \Rightarrow Store \Rightarrow bool$

where

$evalB (compB f e1 e2) s = f (evalE e1 s) (evalE e2 s)$

The category of commands is extended by instructions for allocating a fresh object, obtaining a value from a field and assigning a value to a field.

datatype $OBJ =$

$Skip$
 $| Assign\ Var\ Expr$
 $| New\ Var\ Class$
 $| Get\ Var\ Var\ Field$
 $| Put\ Var\ Field\ Expr$
 $| Comp\ OBJ\ OBJ$
 $| While\ BExpr\ OBJ$
 $| Iff\ BExpr\ OBJ\ OBJ$
 $| Call$

The body of the procedure is identified by the same constant as before.

consts $body :: OBJ$

The operational semantics is again a standard big-step relation.

inductive-set $Sem_n :: (State \times OBJ \times nat \times State) \text{ set}$

where

$SemSkip: s=t \implies (s, Skip, 1, t):Sem_n$

$| SemAssign:$
 $\llbracket t = (update\ (fst\ s)\ x\ (evalE\ e\ (fst\ s)),\ snd\ s) \rrbracket$
 $\implies (s, Assign\ x\ e, 1, t):Sem_n$

$| SemNew:$
 $\llbracket l \notin Dom\ (snd\ s);$
 $t = (update\ (fst\ s)\ x\ (RVal\ (Loc\ l)),\ (l, (C, [])) \# (snd\ s)) \rrbracket$
 $\implies (s, New\ x\ C, 1, t):Sem_n$

$| SemGet:$
 $\llbracket (fst\ s)\ y = RVal\ (Loc\ l); lookup\ (snd\ s)\ l = Some\ (C, Flds);$
 $lookup\ Flds\ F = Some\ v; t = (update\ (fst\ s)\ x\ v,\ snd\ s) \rrbracket$
 $\implies (s, Get\ x\ y\ F, 1, t):Sem_n$

$| SemPut:$
 $\llbracket (fst\ s)\ x = RVal\ (Loc\ l); lookup\ (snd\ s)\ l = Some\ (C, Flds);$
 $t = (fst\ s,\ (l, (C, (F, evalE\ e\ (fst\ s)) \# Flds)) \# (snd\ s)) \rrbracket$
 $\implies (s, Put\ x\ F\ e, 1, t):Sem_n$

$| SemComp:$
 $\llbracket (s, c, n, r):Sem_n; (r, d, m, t):Sem_n; k=(max\ n\ m)+1 \rrbracket$
 $\implies (s, Comp\ c\ d, k, t):Sem_n$

| *SemWhileT*:
 $\llbracket \text{evalB } b \text{ (fst } s); (s, c, n, r):\text{Semn}; (r, \text{While } b \text{ } c, m, t):\text{Semn}; k = ((\max n \ m) + 1) \rrbracket$
 $\implies (s, \text{While } b \text{ } c, k, t):\text{Semn}$

| *SemWhileF*:
 $\llbracket \neg (\text{evalB } b \text{ (fst } s)); t = s \rrbracket \implies (s, \text{While } b \text{ } c, 1, t):\text{Semn}$

| *SemTrue*:
 $\llbracket \text{evalB } b \text{ (fst } s); (s, c1, n, t):\text{Semn} \rrbracket$
 $\implies (s, \text{Iff } b \text{ } c1 \text{ } c2, n+1, t):\text{Semn}$

| *SemFalse*:
 $\llbracket \neg (\text{evalB } b \text{ (fst } s)); (s, c2, n, t):\text{Semn} \rrbracket$
 $\implies (s, \text{Iff } b \text{ } c1 \text{ } c2, n+1, t):\text{Semn}$

| *SemCall*: $\llbracket (s, \text{body}, n, t):\text{Semn} \rrbracket \implies (s, \text{Call}, n+1, t):\text{Semn}$

abbreviation

$\text{SemN} :: [\text{State}, \text{OBJ}, \text{nat}, \text{State}] \Rightarrow \text{bool} \ (\langle -, - \rightarrow - \rangle)$

where

$s, c \rightarrow_n t == (s, c, n, t) : \text{Semn}$

Often, the height index does not matter, so we define a notion hiding it.

definition

$\text{Sem} :: [\text{State}, \text{OBJ}, \text{State}] \Rightarrow \text{bool} \ (\langle -, - \Downarrow - \rangle 1000)$

where $s, c \Downarrow t = (\exists n. s, c \rightarrow_n t)$

inductive-cases *Sem-eval-cases*:

$s, \text{Skip} \rightarrow_n t$
 $s, (\text{Assign } x \ e) \rightarrow_n t$
 $s, (\text{New } x \ C) \rightarrow_n t$
 $s, (\text{Get } x \ y \ F) \rightarrow_n t$
 $s, (\text{Put } x \ F \ e) \rightarrow_n t$
 $s, (\text{Comp } c1 \ c2) \rightarrow_n t$
 $s, (\text{While } b \ c) \rightarrow_n t$
 $s, (\text{Iff } b \ c1 \ c2) \rightarrow_n t$
 $s, \text{Call} \rightarrow_n t$

lemma *Sem-no-zero-height-derivs*: $(s, c \rightarrow_0 t) \implies \text{False}$

Determinism does not hold as allocation is nondeterministic.

End of theory OBJ

end

theory *VDM-OBJ* **imports** *OBJ* **begin**

6.2 Program logic

Apart from the addition of proof rules for the three new instructions, this section is essentially identical to Section 2.

6.2.1 Assertions and their semantic validity

Assertions are binary state predicates, as before.

type-synonym $Assn = State \Rightarrow State \Rightarrow bool$

definition $VDM-validn :: nat \Rightarrow OBJ \Rightarrow Assn \Rightarrow bool$
 $(\langle \models - : - \rangle 50)$

where $(\models_n c : A) = (\forall m . m \leq n \longrightarrow (\forall s t . (s, c \rightarrow_m t) \longrightarrow A s t))$

definition $VDM-valid :: OBJ \Rightarrow Assn \Rightarrow bool$
 $(\langle \models - : - \rangle 50)$

where $(\models c : A) = (\forall s t . (s, c \Downarrow t) \longrightarrow A s t)$

lemma $VDM-valid-validn: \models c:A \Longrightarrow \models_n c:A$

lemma $VDM-validn-valid: (\forall n . \models_n c:A) \Longrightarrow \models c:A$

lemma $VDM-lowerm: \llbracket \models_n c:A; m < n \rrbracket \Longrightarrow \models_m c:A$

definition $Ctxt-validn :: nat \Rightarrow (Assn set) \Rightarrow bool$
 $(\langle \models - : - \rangle 50)$

where $(\models_n G) = (\forall m . m \leq n \longrightarrow (\forall A . A \in G \longrightarrow \models_n Call : A))$

definition $Ctxt-valid :: (Assn set) \Rightarrow bool$ $(\langle \models - : - \rangle 50)$

where $(\models G) = (\forall A . A \in G \longrightarrow \models Call : A)$

lemma $Ctxt-valid-validn: \models G \Longrightarrow \models_n G$

lemma $Ctxt-validn-valid: (\forall n . \models_n G) \Longrightarrow \models G$

lemma $Ctxt-lowerm: \llbracket \models_n G; m < n \rrbracket \Longrightarrow \models_m G$

definition $valid :: (Assn set) \Rightarrow OBJ \Rightarrow Assn \Rightarrow bool$
 $(\langle \models - : - \rangle 50)$

where $(G \models c : A) = (Ctxt-valid G \longrightarrow VDM-valid c A)$

definition $validn :: (Assn set) \Rightarrow nat \Rightarrow OBJ \Rightarrow Assn \Rightarrow bool$
 $(\langle \models - : - \rangle 50)$

where $(G \models_n c : A) = (\models_n G \longrightarrow \models_n c : A)$

lemma $validn-valid: (\forall n . G \models_n c : A) \Longrightarrow G \models c : A$

lemma $ctxt-consn: \llbracket \models_n G; \models_n Call:A \rrbracket \Longrightarrow \models_n \{A\} \cup G$

6.2.2 Proof system

inductive-set $VDM-proof :: (Assn set \times OBJ \times Assn) set$

where

$VDMSkip: (G, Skip, \lambda s t . t=s): VDM-proof$

| $VDMAssign:$

$(G, \text{Assign } x e,$
 $\lambda s t . t = (\text{update } (fst s) x (\text{evalE } e (fst s)), snd s)): \text{VDM-proof}$

| *VDMNew*:
 $(G, \text{New } x C,$
 $\lambda s t . \exists l . l \notin \text{Dom } (snd s) \wedge$
 $t = (\text{update } (fst s) x (RVal (Loc l)),$
 $(l, (C, [])) \# (snd s)): \text{VDM-proof}$

| *VDMGet*:
 $(G, \text{Get } x y F,$
 $\lambda s t . \exists l C Flds v. (fst s) y = RVal(Loc l) \wedge$
 $\text{lookup } (snd s) l = \text{Some}(C, Flds) \wedge$
 $\text{lookup } Flds F = \text{Some } v \wedge$
 $t = (\text{update } (fst s) x v, snd s)): \text{VDM-proof}$

| *VDMPut*:
 $(G, \text{Put } x F e,$
 $\lambda s t . \exists l C Flds. (fst s) x = RVal(Loc l) \wedge$
 $\text{lookup } (snd s) l = \text{Some}(C, Flds) \wedge$
 $t = (fst s,$
 $(l, (C, (F, \text{evalE } e (fst s)) \# Flds))$
 $\# (snd s)): \text{VDM-proof}$

| *VDMComp*:
 $\llbracket (G, c, A): \text{VDM-proof}; (G, d, B): \text{VDM-proof} \rrbracket \implies$
 $(G, \text{Comp } c d, \lambda s t . \exists r . A s r \wedge B r t): \text{VDM-proof}$

| *VDMIff*:
 $\llbracket (G, c, A): \text{VDM-proof}; (G, d, B): \text{VDM-proof} \rrbracket \implies$
 $(G, \text{Iff } b c d,$
 $\lambda s t . (((\text{evalB } b (fst s)) \longrightarrow A s t) \wedge$
 $((\neg (\text{evalB } b (fst s))) \longrightarrow B s t)): \text{VDM-proof}$

| *VDMWhile*:
 $\llbracket (G, c, B): \text{VDM-proof};$
 $\forall s . (\neg \text{evalB } b (fst s)) \longrightarrow A s s;$
 $\forall s r t. \text{evalB } b (fst s) \longrightarrow B s r \longrightarrow A r t \longrightarrow A s t \rrbracket$
 $\implies (G, \text{While } b c, \lambda s t . A s t \wedge \neg (\text{evalB } b (fst t))): \text{VDM-proof}$

| *VDMCall*:
 $(\{A\} \cup G, \text{body}, A): \text{VDM-proof} \implies (G, \text{Call}, A): \text{VDM-proof}$

| *VDMAx*: $A \in G \implies (G, \text{Call}, A): \text{VDM-proof}$

| *VDMConseq*:
 $\llbracket (G, c, A): \text{VDM-proof}; \forall s t. A s t \longrightarrow B s t \rrbracket \implies$
 $(G, c, B): \text{VDM-proof}$

abbreviation $VDM\text{-deriv} :: [Assn\ set, OBJ, Assn] \Rightarrow bool$

$(\langle - \triangleright - : - \rangle [100,100] 50)$

where $G \triangleright c : A == (G, c, A) \in VDM\text{-proof}$

The while-rule is in fact inter-derivable with the following rule.

lemma *Hoare-While*:

$G \triangleright c : (\lambda s t . \forall r . evalB\ b\ (fst\ s) \longrightarrow I\ s\ r \longrightarrow I\ t\ r) \Longrightarrow$

$G \triangleright While\ b\ c : (\lambda s t . \forall r . I\ s\ r \longrightarrow (I\ t\ r \wedge \neg evalB\ b\ (fst\ t)))$

apply (*subgoal-tac* $G \triangleright (While\ b\ c)$:

$(\lambda s t . (\lambda s t . \forall r . I\ s\ r \longrightarrow I\ t\ r)\ s\ t \wedge \neg (evalB\ b\ (fst\ t))))$

apply (*erule* $VDMConseq$)

apply *simp*

apply (*rule* $VDMWhile$) **apply** *assumption* **apply** *simp* **apply** *simp*

done

Here's the proof in the opposite direction.

lemma *VDMWhile-derivable*:

$\llbracket G \triangleright c : B; \forall s . (\neg evalB\ b\ (fst\ s)) \longrightarrow A\ s\ s;$

$\forall s\ r\ t . evalB\ b\ (fst\ s) \longrightarrow B\ s\ r \longrightarrow A\ r\ t \longrightarrow A\ s\ t \rrbracket$

$\Longrightarrow G \triangleright (While\ b\ c) : (\lambda s t . A\ s\ t \wedge \neg (evalB\ b\ (fst\ t)))$

apply (*rule* $VDMConseq$)

apply (*rule* *Hoare-While* [*of* $G\ c\ b\ \lambda s\ r . \forall t . A\ s\ t \longrightarrow A\ r\ t$])

apply (*erule* $VDMConseq$) **apply** *clarsimp*

apply *fast*

done

6.2.3 Soundness

The following auxiliary lemma for loops is proven by induction on n .

lemma *SoundWhile*[*rule-format*]:

$(\forall n . G \models_n c : B)$

$\longrightarrow (\forall s . (\neg evalB\ b\ (fst\ s)) \longrightarrow A\ s\ s)$

$\longrightarrow (\forall s . evalB\ b\ (fst\ s)$

$\longrightarrow (\forall r . B\ s\ r \longrightarrow (\forall t . A\ r\ t \longrightarrow A\ s\ t)))$

$\longrightarrow G \models_n (While\ b\ c) : (\lambda s t . A\ s\ t \wedge \neg evalB\ b\ (fst\ t))$

lemma *SoundCall*[*rule-format*]:

$\llbracket \forall n . \models_n (\{A\} \cup G) \longrightarrow \models_n body : A \rrbracket \Longrightarrow \models_n G \longrightarrow \models_n Call : A$

lemma *VDM-Sound-n*: $G \triangleright c : A \Longrightarrow (\forall n . G \models_n c : A)$

theorem *VDM-Sound*: $G \triangleright c : A \Longrightarrow G \models c : A$

A simple corollary is soundness w.r.t. an empty context.

lemma *VDM-Sound-emptyCtx*: $\{\} \triangleright c : A \Longrightarrow \models c : A$

6.2.4 Derived rules

lemma *WEAK*[*rule-format*]:

$G \triangleright c : A \Longrightarrow (\forall H . G \subseteq H \longrightarrow H \triangleright c : A)$

lemma *CutAux*:

($H \triangleright c : A$) \implies
 $(\forall G P D . (H = (\text{insert } P D) \longrightarrow G \triangleright \text{Call} : P \longrightarrow (G \subseteq D)$
 $\longrightarrow D \triangleright c : A))$

lemma *Cut*: $\llbracket G \triangleright \text{Call} : P ; (\text{insert } P G) \triangleright c : A \rrbracket \implies G \triangleright c : A$

definition *verified*: $\text{Assn set} \Rightarrow \text{bool}$

where *verified* $G = (\forall A . A : G \longrightarrow G \triangleright \text{body} : A)$

lemma *verified-preserved*: $\llbracket \text{verified } G ; A : G \rrbracket \implies \text{verified } (G - \{A\})$

theorem *Mutrec*:

$\llbracket \text{finite } G ; \text{card } G = n ; \text{verified } G ; A : G \rrbracket \implies \{\} \triangleright \text{Call} : A$

6.2.5 Completeness

definition *SSpec*: $\text{OBJ} \Rightarrow \text{Assn}$

where *SSpec* $c \ s \ t = (s, c \Downarrow t)$

lemma *SSpec-valid*: $\models c : (\text{SSpec } c)$

lemma *SSpec-strong*: $\models c : A \implies \forall s \ t . \text{SSpec } c \ s \ t \longrightarrow A \ s \ t$

lemma *SSpec-derivable*: $G \triangleright \text{Call} : \text{SSpec } \text{Call} \implies G \triangleright c : \text{SSpec } c$

definition *StrongG* :: Assn set

where *StrongG* = $\{\text{SSpec } \text{Call}\}$

lemma *StrongG-Body*: $\text{StrongG} \triangleright \text{body} : \text{SSpec } \text{Call}$

lemma *StrongG-verified*: $\text{verified } \text{StrongG}$

lemma *SSpec-derivable-empty*: $\{\} \triangleright c : \text{SSpec } c$

theorem *VDM-Complete*: $\models c : A \implies \{\} \triangleright c : A$

theory *PBIJ* imports *OBJ* begin

6.3 Partial bijections

Partial bijections between locations will be used in the next section to define indistinguishability of objects and heaps. We define such bijections as sets of pairs which satisfy the obvious condition.

type-synonym *PBij* = $(\text{Location} \times \text{Location}) \text{ set}$

definition *Pbij* :: PBij set

where *Pbij* = $\{ \beta . \forall l1 \ l2 \ l3 \ l4 . (l1, l2) : \beta \longrightarrow (l3, l4) : \beta \longrightarrow$
 $((l1 = l3) = (l2 = l4)) \}$

Domain and codomain are defined as expected.

definition *Pbij-Dom*: $\text{PBij} \Rightarrow (\text{Location set})$

where *Pbij-Dom* $\beta = \{l . \exists ll . (l, ll) : \beta\}$

definition *Pbij-Rng*: $\text{PBij} \Rightarrow (\text{Location set})$

where *Pbij-Rng* $\beta = \{ll . \exists l . (l, ll) : \beta\}$

We also define the inverse operation, the composition, and a test deciding when one bijection extends another.

definition *Pbij-inverse*:: $PBij \Rightarrow PBij$
where *Pbij-inverse* $\beta = \{(l, ll) . (ll, l):\beta\}$
definition *Pbij-compose*:: $PBij \Rightarrow PBij \Rightarrow PBij$
where *Pbij-compose* $\beta \gamma = \{(l, ll) . \exists ll' . (l, ll'):\beta \wedge (ll', ll):\gamma\}$
definition *Pbij-extends* :: $PBij \Rightarrow PBij \Rightarrow bool$
where *Pbij-extends* $\gamma \beta = (\beta \subseteq \gamma)$

These definitions satisfy the following properties.

lemma *Pbij-insert*:

$\llbracket \beta \in Pbij; l \notin Pbij\text{-Rng } \beta; ll \notin Pbij\text{-Dom } \beta \rrbracket$
 $\implies \text{insert } (ll, l) \beta \in Pbij$

lemma *Pbij-injective*:

$\beta:Pbij \implies (\forall ll' ll'' . (ll', l):\beta \longrightarrow (ll'', l):\beta \longrightarrow ll' = ll'')$

lemma *Pbij-inverse-DomRng*[rule-format]:

$\gamma = Pbij\text{-inverse } \beta \implies$
 $(Pbij\text{-Dom } \beta = Pbij\text{-Rng } \gamma \wedge Pbij\text{-Dom } \gamma = Pbij\text{-Rng } \beta)$

lemma *Pbij-inverse-Dom*: $Pbij\text{-Dom } \beta = Pbij\text{-Rng } (Pbij\text{-inverse } \beta)$

lemma *Pbij-inverse-Rng*: $Pbij\text{-Rng } (Pbij\text{-inverse } \beta) = Pbij\text{-Rng } \beta$

lemma *Pbij-inverse-Pbij*: $\beta:Pbij \implies (Pbij\text{-inverse } \beta) : Pbij$

lemma *Pbij-inverse-Inverse*[rule-format]:

$\gamma = Pbij\text{-inverse } \beta \implies (\forall ll' ll'' . ((ll', ll):\beta) = ((ll', ll):\gamma))$

lemma *Pbij-compose-Dom*:

$Pbij\text{-Dom } (Pbij\text{-compose } \beta \gamma) \subseteq Pbij\text{-Dom } \beta$

lemma *Pbij-compose-Rng*:

$Pbij\text{-Rng } (Pbij\text{-compose } \beta \gamma) \subseteq Pbij\text{-Rng } \gamma$

lemma *Pbij-compose-Pbij*:

$\llbracket \beta : Pbij; \gamma : Pbij \rrbracket \implies Pbij\text{-compose } \beta \gamma : Pbij$

lemma *Pbij-extends-inverse*:

$Pbij\text{-extends } \gamma (Pbij\text{-inverse } \beta) = Pbij\text{-extends } (Pbij\text{-inverse } \gamma) \beta$

lemma *Pbij-extends-reflexive*: $Pbij\text{-extends } \beta \beta$

lemma *Pbij-extends-transitive*:

$\llbracket Pbij\text{-extends } \beta \gamma; Pbij\text{-extends } \gamma \delta \rrbracket \implies Pbij\text{-extends } \beta \delta$

lemma *Pbij-inverse-extends-twice*:

$Pbij\text{-extends } (Pbij\text{-inverse } (Pbij\text{-inverse } \beta)) \beta$

The identity bijection on a heap associates each element of the heap's domain with itself.

definition *mkId*:: $Heap \Rightarrow (Location \times Location) \text{ set}$

where *mkId* $h = \{(ll, ll) . ll = ll \wedge ll : Dom h\}$

lemma *mkId1*: $(mkId h):Pbij$

lemma *mkId2*: $Pbij\text{-Dom } (mkId h) = Dom h$

lemma *mkId2b*: $Pbij\text{-Rng } (mkId h) = Dom h$

lemma *mkId4*: $l:Dom h \implies (l, l):(mkId h)$

lemma *mkId4b*: $(l, ll):(mkId h) \implies l:Dom h \wedge l = ll$

End of theory PBIJ

end

theory *VS-OBJ* **imports** *VDM-OBJ PBIJ* **begin**

6.4 Non-interference

6.4.1 Indistinguishability relations

We have the usual two security types.

datatype $TP = low \mid high$

Global contexts assigns security types to program variables and field names.

consts $CONTEXT :: Var \Rightarrow TP$

consts $GAMMA :: Field \Rightarrow TP$

Indistinguishability of values depends on a partial bijection β .

inductive-set $twiddleVal :: (PBij \times Val \times Val) \text{ set}$

where

$twiddleVal\text{-}Null: (\beta, RVal\ Nullref, RVal\ Nullref) : twiddleVal$

$| twiddleVal\text{-}Loc: (l1, l2) : \beta \Longrightarrow$

$(\beta, RVal\ (Loc\ l1), RVal\ (Loc\ l2)) : twiddleVal$

$| twiddleVal\text{-}IVal: i1 = i2 \Longrightarrow (\beta, IVal\ i1, IVal\ i2) : twiddleVal$

For stores, indistinguishability is as follows.

definition $twiddleStore :: PBij \Rightarrow Store \Rightarrow Store \Rightarrow bool$

where $twiddleStore\ \beta\ s1\ s2 =$

$(\forall x. CONTEXT\ x = low \longrightarrow (\beta, s1\ x, s2\ x) : twiddleVal)$

abbreviation $twiddleStore\text{-}syntax\ (\leftarrow \approx_{\beta} \rightarrow [100, 100])\ 50$

where $s \approx_{\beta} t == twiddleStore\ \beta\ s\ t$

On objects, we require the values in low fields to be related, and the sets of defined low fields to be equal.

definition $LowDom :: ((Field \times Val) \text{ list}) \Rightarrow Field \text{ set}$

where $LowDom\ F = \{f . \exists v . lookup\ F\ f = Some\ v \wedge GAMMA\ f = low\}$

definition $twiddleObj :: PBij \Rightarrow Object \Rightarrow Object \Rightarrow bool$

where $twiddleObj\ \beta\ o1\ o2 = ((fst\ o1 = fst\ o2) \wedge$

$LowDom\ (snd\ o1) = LowDom\ (snd\ o2) \wedge$

$(\forall f\ v\ w . lookup\ (snd\ o1)\ f = Some\ v \longrightarrow$

$lookup\ (snd\ o2)\ f = Some\ w \longrightarrow$

$GAMMA\ f = low \longrightarrow$

$(\beta, v, w) : twiddleVal))$

On heaps, we require locations related by β to contain indistinguishable objects. Domain and codomain of the bijection should be subsets of the domains of the heaps, of course.

definition $twiddleHeap::PBij \Rightarrow Heap \Rightarrow Heap \Rightarrow bool$
where $twiddleHeap \beta h1 h2 = (\beta:PBij \wedge$
 $Pbij-Dom \beta \subseteq Dom h1 \wedge$
 $Pbij-Rng \beta \subseteq Dom h2 \wedge$
 $(\forall l ll v w . (l,ll):\beta \longrightarrow$
 $lookup h1 l = Some v \longrightarrow$
 $lookup h2 ll = Some w \longrightarrow$
 $twiddleObj \beta v w))$

We also define a predicate which expresses when a state does not contain dangling pointers.

definition $noLowDPs::State \Rightarrow bool$
where $noLowDPs S = (case S of (s,h) \Rightarrow$
 $(\forall x l . CONTEXT x = low \longrightarrow s x = RVal(Loc l) \longrightarrow l:Dom h) \wedge$
 $(\forall ll c F fl . lookup h ll = Some(c,F) \longrightarrow GAMMA f = low \longrightarrow$
 $lookup F f = Some(RVal(Loc l)) \longrightarrow l:Dom h))$

The motivation for introducing this notion stems from the intended interpretation of the proof rule for skip, where the initial and final states should be low equivalent. However, in the presence of dangling pointers, indistinguishability does not hold as such a dangling pointer is not in the expected bijection $mkId$. In contrast, for the notion of indistinguishability we use (see the following definition), reflexivity indeed holds, as proven in lemma $twiddle-mkId$ below. As a small improvement in comparison to our paper, we now allow dangling pointers in high variables and high fields since these are harmless.

definition $twiddle::PBij \Rightarrow State \Rightarrow State \Rightarrow bool$
where $twiddle \beta s t = (noLowDPs s \wedge noLowDPs t \wedge$
 $(fst s) \approx_{\beta} (fst t) \wedge twiddleHeap \beta (snd s) (snd t))$

abbreviation $twiddle-syntax (\leftarrow \equiv \rightarrow [100,100] 50)$
where $s \equiv_{\beta} t == twiddle \beta s t$

The following properties are easily proven by unfolding the definitions.

lemma $twiddleHeap-isPbij:twiddleHeap \beta h hh \Longrightarrow \beta:PBij$

lemma $isPbij:s \equiv_{\beta} t \Longrightarrow \beta:PBij$

lemma $twiddleVal-inverse:$

$(\beta, w, v) \in twiddleVal \Longrightarrow (Pbij-inverse \beta, v, w) \in twiddleVal$

lemma $twiddleStore-inverse: s \approx_{\beta} t \Longrightarrow t \approx_{(Pbij-inverse \beta)} s$

lemma $twiddleHeap-inverse:$

$twiddleHeap \beta s t \Longrightarrow twiddleHeap (Pbij-inverse \beta) t s$

lemma $Pbij-inverse-twiddle: \llbracket s \equiv_{\beta} t \rrbracket \Longrightarrow t \equiv_{(Pbij-inverse \beta)} s$

lemma $twiddleVal-betaExtend[rule-format]:$

$(\beta,v,w):twiddleVal \Longrightarrow \forall \gamma. Pbij-extends \gamma \beta \longrightarrow (\gamma,v,w):twiddleVal$

lemma $twiddleObj-betaExtend[rule-format]:$

$\llbracket twiddleObj \beta o1 o2; Pbij-extends \gamma \beta \rrbracket \Longrightarrow twiddleObj \gamma o1 o2$

lemma $twiddleVal-compose:$

$\llbracket (\beta, v, u) \in twiddleVal; (\gamma, u, w) \in twiddleVal \rrbracket$

$\implies (Pbij\text{-compose } \beta \ \gamma, v, w) \in twiddleVal$

lemma *twiddleHeap-compose*:

$\llbracket twiddleHeap \ \beta \ h1 \ h2; twiddleHeap \ \gamma \ h2 \ h3; \beta \in Pbij; \gamma \in Pbij \rrbracket$
 $\implies twiddleHeap (Pbij\text{-compose } \beta \ \gamma) \ h1 \ h3$

lemma *twiddleStore-compose*:

$\llbracket s \approx_\beta r; r \approx_\gamma t \rrbracket \implies s \approx_{(Pbij\text{-compose } \beta \ \gamma)} t$

lemma *twiddle-compose*:

$\llbracket s \equiv_\beta r; r \equiv_\gamma t \rrbracket \implies s \equiv_{(Pbij\text{-compose } \beta \ \gamma)} t$

lemma *twiddle-mkId*: $noLowDPs (s, h) \implies (s, h) \equiv_{(mkId \ h)} (s, h)$

We call expressions (semantically) low if the following predicate is satisfied. In particular, this means that if e evaluates in s (respectively, t) to some location l , then $l \in Pbij_dom(\beta)$ ($l \in Pbij_cod(\beta)$) holds.

definition *Expr-low::Expr* $\Rightarrow bool$

where *Expr-low* $e = (\forall s \ t \ \beta. s \approx_\beta t \longrightarrow (\beta, evalE \ e \ s, evalE \ e \ t):twiddleVal)$

A similar notion is defined for boolean expressions, but the fact that these evaluate to (meta-logical) boolean values allows us to replace indistinguishability by equality.

definition *BExpr-low::BExpr* $\Rightarrow bool$

where *BExpr-low* $b = (\forall s \ t \ \beta. s \approx_\beta t \longrightarrow evalB \ b \ s = evalB \ b \ t)$

6.4.2 Definition and characterisation of security

Now, the notion of security, as defined in the paper. Banerjee and Naumann's paper [1] and the Mobius Deliverable 2.3 [3] contain similar notions.

definition *secure::OBJ* $\Rightarrow bool$

where *secure* $c = (\forall s \ ss \ t \ tt \ \beta.$

$s \equiv_\beta ss \longrightarrow (s, c \Downarrow t) \longrightarrow (ss, c \Downarrow tt) \longrightarrow$
 $(\exists \gamma. t \equiv_\gamma tt \wedge Pbij\text{-extends } \gamma \ \beta))$

The type of invariants Φ includes a component that holds a partial bijection.

type-synonym $TT = (State \times State \times Pbij) \Rightarrow bool$

The operator constructing an assertion from an invariant.

definition *Sec* $:: TT \Rightarrow Assn$

where *Sec* $\Phi \ s \ t =$

$((\forall r \ \beta. s \equiv_\beta r \longrightarrow \Phi(t, r, \beta)) \wedge$
 $(\forall r \ \beta. \Phi(r, s, \beta) \longrightarrow (\exists \gamma. r \equiv_\gamma t \wedge Pbij\text{-extends } \gamma \ \beta)))$

The lemmas proving that the operator ensures security, and that secure programs satisfy an assertion formed by the operator, are proven in a similar way as in Section 3.

lemma *Prop1A*: $\models c : Sec \ \Phi \implies secure \ c$

lemma *Prop1B*:

$secure \ c \implies \models c : Sec (\lambda (r, t, \beta). \exists s. s, c \Downarrow r \wedge s \equiv_\beta t)$

lemma *Prop1BB:secure* $c \implies \exists \Phi . \models c : \text{Sec } \Phi$

lemma *Prop1*:

$\text{secure } c = (\models c : \text{Sec } (\lambda (r, t, \beta) . \exists s . (s, c \Downarrow r) \wedge s \equiv_\beta t))$

6.5 Derivation of proof rules

6.5.1 Low proof rules

lemma *SKIP*: $G \triangleright \text{Skip} : \text{Sec } (\lambda (s, t, \beta) . s \equiv_\beta t)$

lemma *ASSIGN*:

Expr-low e

$\implies G \triangleright \text{Assign } x e : \text{Sec } (\lambda (s, t, \beta) .$

$\exists r . s = (\text{update } (fst r) x (\text{evalE } e (fst r)), snd r)$

$\wedge r \equiv_\beta t)$

lemma *COMP*:

$\llbracket G \triangleright c1 : \text{Sec } \Phi; G \triangleright c2 : \text{Sec } \Psi \rrbracket$

$\implies G \triangleright (\text{Comp } c1 c2) : \text{Sec } (\lambda (s, t, \beta) .$

$\exists r . \Phi(r, t, \beta) \wedge (\forall w \gamma . r \equiv_\gamma w \longrightarrow \Psi(s, w, \gamma)))$

lemma *IFF*:

$\llbracket \text{BExpr-low } b; G \triangleright c1 : \text{Sec } \Phi; G \triangleright c2 : \text{Sec } \Psi \rrbracket$

$\implies G \triangleright (\text{Iff } b c1 c2) : \text{Sec } (\lambda (s, t, \beta) .$

$(\text{evalB } b (fst t) \longrightarrow \Phi(s, t, \beta)) \wedge$

$((\neg \text{evalB } b (fst t) \longrightarrow \Psi(s, t, \beta)))$

lemma *NEW*:

CONTEXT $x = low$

$\implies G \triangleright (\text{New } x C) : \text{Sec } (\lambda (s, t, \beta) .$

$\exists l r . l \notin \text{Dom } (snd r) \wedge r \equiv_\beta t \wedge$

$s = (\text{update } (fst r) x (\text{RVal } (\text{Loc } l)),$

$(l, (C, [])) \# (snd r))$

lemma *GET*:

$\llbracket \text{CONTEXT } y = low; \text{GAMMA } f = low \rrbracket$

$\implies G \triangleright \text{Get } x y f : \text{Sec } (\lambda (s, t, \beta) .$

$\exists r l C \text{Flds } v . (fst r) y = \text{RVal}(\text{Loc } l) \wedge$

$\text{lookup } (snd r) l = \text{Some}(C, \text{Flds}) \wedge$

$\text{lookup } \text{Flds } f = \text{Some } v \wedge r \equiv_\beta t \wedge$

$s = (\text{update } (fst r) x v, snd r)$

lemma *PUT*:

$\llbracket \text{CONTEXT } x = low; \text{GAMMA } f = low; \text{Expr-low } e \rrbracket$

$\implies G \triangleright \text{Put } x f e : \text{Sec } (\lambda (s, t, \beta) .$

$\exists r l C \text{Flds} . (fst r) x = \text{RVal}(\text{Loc } l) \wedge r \equiv_\beta t \wedge$

$\text{lookup } (snd r) l = \text{Some}(C, \text{Flds}) \wedge$

$s = (fst r,$

$(l, (C, (f, \text{evalE } e (fst r)) \# \text{Flds})) \# (snd r))$

Again, we define a fixed point operator over invariants.

definition *FIX*:: $(TT \Rightarrow TT) \Rightarrow TT$

where *FIX* $\varphi = (\lambda (s, t, \beta) .$

$\forall \Phi . (\forall ss tt \gamma . \varphi \Phi (ss, tt, \gamma) \longrightarrow \Phi (ss, tt, \gamma)) \longrightarrow \Phi (s, t, \beta))$

definition *Monotone*:: $(TT \Rightarrow TT) \Rightarrow bool$
where *Monotone* $\varphi =$
 $(\forall \Phi \Psi . (\forall s t \beta. \Phi(s,t,\beta) \longrightarrow \Psi(s,t,\beta)) \longrightarrow$
 $(\forall s t \beta. \varphi \Phi (s,t,\beta) \longrightarrow \varphi \Psi (s,t,\beta)))$

For monotone transformers, the construction indeed yields a fixed point.

lemma *Fix-lemma*: $Monotone \varphi \Longrightarrow \varphi (FIX \varphi) = FIX \varphi$

The operator used in the while rule is defined by

definition *PhiWhileOp*:: $BExpr \Rightarrow TT \Rightarrow TT \Rightarrow TT$
where *PhiWhileOp* $b \Phi = (\lambda \Psi . (\lambda (s, t, \beta).$
 $(evalB b (fst t) \longrightarrow$
 $(\exists r. \Phi (r, t, \beta) \wedge (\forall w \gamma. r \equiv_{\gamma} w \longrightarrow \Psi(s, w, \gamma)))) \wedge$
 $(\neg evalB b (fst t) \longrightarrow s \equiv_{\beta} t)))$

and is monotone:

lemma *PhiWhileOp-Monotone*: $Monotone (PhiWhileOp b \Phi)$

Hence, we can define its fixed point:

definition *PhiWhile*:: $BExpr \Rightarrow TT \Rightarrow TT$
where *PhiWhile* $b \Phi = FIX (PhiWhileOp b \Phi)$

The while rule may now be given as follows:

lemma *WHILE*:
 $\llbracket BExpr\text{-low } b; G \triangleright c : (Sec \Phi) \rrbracket$
 $\Longrightarrow G \triangleright (While b c) : (Sec (PhiWhile b \Phi))$

Operator *PhiWhile* is itself monotone in Φ :

lemma *PhiWhileMonotone*: $Monotone (\lambda \Phi . PhiWhile b \Phi)$

We now give an alternative formulation using an inductive relation equivalent to *FIX*. First, the definition of the variant.

inductive-set *var*:: $(BExpr \times TT \times PBij \times State \times State) \text{ set}$
where
 $varFalse: \llbracket \neg evalB b (fst t); s \equiv_{\beta} t \rrbracket \Longrightarrow (b, \Phi, \beta, s, t):var$

| *varTrue*:
 $\llbracket evalB b (fst t); \Phi(r,t,\beta); \forall w \gamma. r \equiv_{\gamma} w \longrightarrow (b, \Phi, \gamma, s, w): var \rrbracket$
 $\Longrightarrow (b, \Phi, \beta, s, t):var$

The equivalence of the invariant with the fixed point construction.

lemma *varFIXvar*: $(PhiWhile b \Phi (s,t,\beta)) = ((b, \Phi, \beta, s, t):var)$

Using this equivalence we can derive the while rule given in our paper from *WHILE*.

lemma *WHILE-IND*:

$$\begin{aligned} & \llbracket BExpr\text{-low } b; G \triangleright c : (Sec \Phi) \rrbracket \\ & \implies G \triangleright \text{While } b \text{ c: } (Sec (\lambda(s,t,\gamma). (b,\Phi,\gamma,s,t):var)) \end{aligned}$$

We can also show the following property.

lemma *var-Monotone*:

$$\text{Monotone } (\lambda \Phi . (\lambda (s,t,\beta) . (b,\Phi,\beta,s,t):var))$$

The call rule is formulated for an arbitrary fixed point of a monotone transformer.

lemma *CALL*:

$$\begin{aligned} & \llbracket (\{Sec (FIX \varphi)\} \cup X) \triangleright \text{body} : Sec (\varphi (FIX \varphi)); \text{Monotone } \varphi \rrbracket \\ & \implies X \triangleright \text{Call} : Sec (FIX \varphi) \end{aligned}$$

6.5.2 High proof rules

definition *HighSec::Assn*

where $\text{HighSec} = (\lambda s t . \text{noLowDPs } s \longrightarrow s \equiv_{(mkId (snd s))} t)$

lemma *CAST[rule-format]*:

$$G \triangleright c : \text{HighSec} \implies G \triangleright c : Sec (\lambda (s, t, \beta) . s \equiv_{\beta} t)$$

lemma *SkipHigh*: $G \triangleright \text{Skip} : \text{HighSec}$

We define a predicate expressing when locations obtained by evaluating an expression are non-dangling.

definition *Expr-good::Expr \Rightarrow State \Rightarrow bool*

where $\text{Expr-good } e \text{ s} =$

$$(\forall l . \text{evalE } e (fst s) = RVal(Loc l) \longrightarrow l : Dom (snd s))$$

lemma *AssignHigh*:

$$\begin{aligned} & \llbracket \text{CONTEXT } x = \text{high}; \forall s . \text{noLowDPs } s \longrightarrow \text{Expr-good } e \text{ s} \rrbracket \\ & \implies G \triangleright \text{Assign } x \text{ e: } \text{HighSec} \end{aligned}$$

lemma *NewHigh*:

$$\text{CONTEXT } x = \text{high} \implies G \triangleright \text{New } x \text{ C} : \text{HighSec}$$

lemma *GetHigh*:

$$\llbracket \text{CONTEXT } x = \text{high} \rrbracket \implies G \triangleright \text{Get } x \text{ y f} : \text{HighSec}$$

lemma *PutHigh*:

$$\begin{aligned} & \llbracket \text{GAMMA } f = \text{high}; \forall s . \text{noLowDPs } s \longrightarrow \text{Expr-good } e \text{ s} \rrbracket \\ & \implies G \triangleright \text{Put } x \text{ f e: } \text{HighSec} \end{aligned}$$

lemma *CompHigh*:

$$\llbracket G \triangleright c : \text{HighSec}; G \triangleright d : \text{HighSec} \rrbracket \implies G \triangleright \text{Comp } c \text{ d: } \text{HighSec}$$

lemma *IfHigh*:

$$\llbracket G \triangleright c : \text{HighSec}; G \triangleright d : \text{HighSec} \rrbracket \implies G \triangleright \text{Iff } b \text{ c d: } \text{HighSec}$$

lemma *WhileHigh*: $\llbracket G \triangleright c : \text{HighSec} \rrbracket \implies G \triangleright \text{While } b \text{ c: } \text{HighSec}$

lemma *CallHigh*:

$$(\{\text{HighSec}\} \cup G) \triangleright \text{body} : \text{HighSec} \implies G \triangleright \text{Call} : \text{HighSec}$$

We combine all rules to an inductive derivation system.

inductive-set *Deriv::(Assn set \times OBJ \times Assn) set*

where

D-CAST:

$$(G, c, \text{HighSec}):Deriv \implies \\ (G, c, \text{Sec } (\lambda (s, t, \beta). s \equiv_{\beta} t)):Deriv$$

| *D-SKIP*: $(G, \text{Skip}, \text{Sec } (\lambda (s, t, \beta). s \equiv_{\beta} t)) : Deriv$

| *D-ASSIGN*:

$$\text{Expr-low } e \implies \\ (G, \text{Assign } x e, \text{Sec } (\lambda (s, t, \beta). \\ \exists r . s = (\text{update } (fst r) x (\text{evalE } e (fst r)), snd r) \\ \wedge r \equiv_{\beta} t)):Deriv$$

| *D-COMP*:

$$\llbracket (G, c1, \text{Sec } \Phi):Deriv; (G, c2, \text{Sec } \Psi):Deriv \rrbracket \implies \\ (G, \text{Comp } c1 c2, \text{Sec } (\lambda (s, t, \beta). \\ \exists r . \Phi(r, t, \beta) \wedge \\ (\forall w \gamma. r \equiv_{\gamma} w \longrightarrow \Psi(s, w, \gamma))):Deriv$$

| *D-IFF*:

$$\llbracket \text{BExpr-low } b; (G, c1, \text{Sec } \Phi):Deriv; (G, c2, \text{Sec } \Psi):Deriv \rrbracket \implies \\ (G, \text{Iff } b c1 c2, \text{Sec } (\lambda (s, t, \beta). \\ (\text{evalB } b (fst t) \longrightarrow \Phi(s, t, \beta)) \wedge \\ ((\neg \text{evalB } b (fst t)) \longrightarrow \Psi(s, t, \beta))):Deriv$$

| *D-NEW*:

$$\text{CONTEXT } x = low \implies \\ (G, \text{New } x C, \text{Sec } (\lambda (s, t, \beta). \\ \exists l r . l \notin \text{Dom } (snd r) \wedge r \equiv_{\beta} t \wedge \\ s = (\text{update } (fst r) x (\text{RVal } (Loc l)), \\ (l, (C, [])) \# (snd r))):Deriv$$

| *D-GET*:

$$\llbracket \text{CONTEXT } y = low; \text{GAMMA } f = low \rrbracket \implies \\ (G, \text{Get } x y f, \text{Sec } (\lambda (s, t, \beta). \\ \exists r l C \text{ Flds } v. (fst r) y = \text{RVal}(Loc l) \wedge \\ \text{lookup } (snd r) l = \text{Some}(C, \text{Flds}) \wedge \\ \text{lookup } \text{Flds } f = \text{Some } v \wedge r \equiv_{\beta} t \wedge \\ s = (\text{update } (fst r) x v, snd r)):Deriv$$

| *D-PUT*:

$$\llbracket \text{CONTEXT } x = low; \text{GAMMA } f = low; \text{Expr-low } e \rrbracket \implies \\ (G, \text{Put } x f e, \text{Sec } (\lambda (s, t, \beta). \\ \exists r l C F h. (fst r) x = \text{RVal}(Loc l) \wedge r \equiv_{\beta} t \wedge \\ \text{lookup } (snd r) l = \text{Some}(C, F) \wedge \\ h = (l, (C, (f, \text{evalE } e (fst r)) \# F)) \# (snd r) \wedge \\ s = (fst r, h)):Deriv$$

| *D-WHILE*:

$$\begin{aligned}
& \llbracket BExpr\text{-low } b; (G, c, Sec \Phi):Deriv \rrbracket \\
& \implies (G, While \ b \ c, Sec (PhiWhile \ b \ \Phi)):Deriv \\
| \ D\text{-CALL}: \\
& \llbracket (\{Sec (FIX \ \varphi)\} \cup G, body, Sec (\varphi (FIX \ \varphi))):Deriv; Monotone \ \varphi \rrbracket \\
& \implies (G, Call, Sec (FIX \ \varphi)):Deriv \\
| \ D\text{-SKIP-H}: (G, Skip, HighSec):Deriv \\
| \ D\text{-ASSIGN-H}: \\
& \llbracket CONTEXT \ x = high; \forall \ s . noLowDPs \ s \longrightarrow Expr\text{-good } e \ s \rrbracket \\
& \implies (G, Assign \ x \ e, HighSec):Deriv \\
| \ D\text{-NEW-H}: CONTEXT \ x = high \implies (G, New \ x \ C, HighSec):Deriv \\
| \ D\text{-GET-H}: CONTEXT \ x = high \implies (G, Get \ x \ y \ f, HighSec):Deriv \\
| \ D\text{-PUT-H}: \\
& \llbracket GAMMA \ f = high; \forall \ s . noLowDPs \ s \longrightarrow Expr\text{-good } e \ s \rrbracket \\
& \implies (G, Put \ x \ f \ e, HighSec):Deriv \\
| \ D\text{-COMP-H}: \\
& \llbracket (G, c, HighSec):Deriv; (G, d, HighSec):Deriv \rrbracket \\
& \implies (G, Comp \ c \ d, HighSec):Deriv \\
| \ D\text{-IFF-H}: \\
& \llbracket (G, c, HighSec):Deriv; (G, d, HighSec):Deriv \rrbracket \\
& \implies (G, Iff \ b \ c \ d, HighSec):Deriv \\
| \ D\text{-WHILE-H}: \\
& \llbracket (G, c, HighSec):Deriv \rrbracket \implies (G, While \ b \ c, HighSec):Deriv \\
| \ D\text{-CALL-H}: \\
& (\{HighSec\} \cup G, body, HighSec):Deriv \implies (G, Call, HighSec):Deriv
\end{aligned}$$

By construction, all derivations represent legal derivations in the program logic. Here's an explicit lemma to this effect.

lemma *Deriv-derivable*: $(G, c, A):Deriv \implies G \triangleright c: A$

6.6 Type system

We now give a type system in the style of Volpano et al. and then prove its embedding into the system of derived rules. First, type systems for expressions and boolean expressions. These are similar to the ones in Section 3 but require some side conditions regarding the (semantically modelled) operators.

definition *opEGood*:: $(Val \Rightarrow Val \Rightarrow Val) \Rightarrow bool$
where *opEGood* $f = (\forall \beta \ v \ v' \ w \ w' . (\beta, v, v') \in twiddleVal \longrightarrow$

$$(\beta, w, w') \in \text{twiddleVal} \longrightarrow (\beta, f v w, f v' w') \in \text{twiddleVal}$$

definition $\text{compBGood}::(\text{Val} \Rightarrow \text{Val} \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where $\text{compBGood } f = (\forall \beta v v' w w'. (\beta, v, v') \in \text{twiddleVal} \longrightarrow$
 $(\beta, w, w') \in \text{twiddleVal} \longrightarrow f v w = f v' w')$

inductive-set $\text{VS-expr}::(\text{Expr} \times \text{TP}) \text{ set}$

where

$\text{VS-exprVar}: \text{CONTEXT } x = t \Longrightarrow (\text{varE } x, t) : \text{VS-expr}$

|

$\text{VS-exprVal}:$

$\llbracket v = \text{RVal Nullref} \vee (\exists i. v = \text{IVal } i) \rrbracket \Longrightarrow (\text{valE } v, \text{low}) : \text{VS-expr}$

|

$\text{VS-exprOp}:$

$\llbracket (e1, t) : \text{VS-expr}; (e2, t) : \text{VS-expr}; \text{opEGood } f \rrbracket$
 $\Longrightarrow (\text{opE } f e1 e2, t) : \text{VS-expr}$

|

$\text{VS-exprHigh}: (e, \text{high}) : \text{VS-expr}$

inductive-set $\text{VS-Bexpr}::(\text{BExpr} \times \text{TP}) \text{ set}$

where

$\text{VS-BexprOp}:$

$\llbracket (e1, t) : \text{VS-expr}; (e2, t) : \text{VS-expr}; \text{compBGood } f \rrbracket$
 $\Longrightarrow (\text{compB } f e1 e2, t) : \text{VS-Bexpr}$

|

$\text{VS-BexprHigh}: (e, \text{high}) : \text{VS-Bexpr}$

Next, the core of the type system, the rules for commands. The second side conditions of rules VS-comAssH and VS-comPutH could be strengthened to $\forall s. \text{Expr_good } e s$.

inductive-set $\text{VS-com}::(\text{TP} \times \text{OBJ}) \text{ set}$

where

$\text{VS-comGetL}:$

$\llbracket \text{CONTEXT } y = \text{low}; \text{GAMMA } f = \text{low} \rrbracket$
 $\Longrightarrow (\text{low}, \text{Get } x y f) : \text{VS-com}$

| $\text{VS-comGetH}: \text{CONTEXT } x = \text{high} \Longrightarrow (\text{high}, \text{Get } x y f) : \text{VS-com}$

| $\text{VS-comPutL}:$

$\llbracket \text{CONTEXT } x = \text{low}; \text{GAMMA } f = \text{low}; (e, \text{low}) : \text{VS-expr} \rrbracket$
 $\Longrightarrow (\text{low}, \text{Put } x f e) : \text{VS-com}$

| $\text{VS-comPutH}:$

$\llbracket \text{GAMMA } f = \text{high}; \forall s. \text{noLowDPs } s \longrightarrow \text{Expr-good } e s \rrbracket$
 $\Longrightarrow (\text{high}, \text{Put } x f e) : \text{VS-com}$

| $\text{VS-comNewL}:$

$\text{CONTEXT } x = \text{low} \Longrightarrow (\text{low}, \text{New } x c) : \text{VS-com}$

| *VS-comNewH*:
 $CONTEXT\ x = high \implies (high, New\ x\ C):VS-com$

| *VS-comAssL*:
 $\llbracket CONTEXT\ x = low; (e, low):VS-expr \rrbracket$
 $\implies (low, Assign\ x\ e) : VS-com$

| *VS-comAssH*:
 $\llbracket CONTEXT\ x = high; \forall s . noLowDPs\ s \longrightarrow Expr-good\ e\ s \rrbracket$
 $\implies (high, Assign\ x\ e) : VS-com$

| *VS-comSkip*: $(pc, Skip) : VS-com$

| *VS-comComp*:
 $\llbracket (pc, c):VS-com; (pc, d):VS-com \rrbracket \implies (pc, Comp\ c\ d) : VS-com$

| *VS-comIf*:
 $\llbracket (b, pc):VS-Bexpr; (pc, c):VS-com; (pc, d):VS-com \rrbracket$
 $\implies (pc, Iff\ b\ c\ d):VS-com$

| *VS-comWhile*:
 $\llbracket (b, pc):VS-Bexpr; (pc, c):VS-com \rrbracket \implies (pc, While\ b\ c):VS-com$

| *VS-comSub*: $(high, c) : VS-com \implies (low, c):VS-com$

In order to prove the type system sound, we first define the interpretation of expression typings...

primrec *SemExpr*::*Expr* \Rightarrow *TP* \Rightarrow *bool*
where
SemExpr *e* *low* = *Expr-low* *e* |
SemExpr *e* *high* = *True*

... and show the soundness of the typing rules.

lemma *ExprSound*: $(e, tp):VS-expr \implies SemExpr\ e\ tp$

Likewise for the boolean expressions.

primrec *SemBExpr*::*BExpr* \Rightarrow *TP* \Rightarrow *bool*
where
SemBExpr *b* *low* = *BExpr-low* *b* |
SemBExpr *b* *high* = *True*

lemma *BExprSound*: $(e, tp):VS-Bexpr \implies SemBExpr\ e\ tp$

Using these auxiliary lemmas we can prove the embedding of the type system for commands into the system of derived proof rules, by induction on the typing rules.

theorem *VS-com-Deriv*[*rule-format*]:
 $(t, c):VS-com \implies (t=high \longrightarrow (G, c, HighSec):Deriv) \wedge$
 $(t=low \longrightarrow (\exists \Phi . (G, c, Sec\ \Phi):Deriv))$

Combining this result with the derivability of the derived proof system and the soundness theorem of the program logic yields non-interference of programs that are low typeable.

theorem *VS-SOUND*: $(low, c): VS-com \implies secure\ c$

End of theory *VS-OBJ*

end

theory *ContextOBJ* **imports** *VS-OBJ* **begin**

6.7 Contextual closure

We first define contexts with multiple holes.

datatype *CtxtProg* =
Ctxt-Here
| *Ctxt-Skip*
| *Ctxt-Assign* *Var Expr*
| *Ctxt-New* *Var Class*
| *Ctxt-Get* *Var Var Field*
| *Ctxt-Put* *Var Field Expr*
| *Ctxt-Comp* *CtxtProg CtxtProg*
| *Ctxt-If* *BExpr CtxtProg CtxtProg*
| *Ctxt-While* *BExpr CtxtProg*
| *Ctxt-Call*

The definition of a procedure body with holes.

consts *Ctxt-Body*::*CtxtProg*

Next, the substitution of a command into a context:

primrec *Fill*::*CtxtProg* \Rightarrow *OBJ* \Rightarrow *OBJ*
where
Fill Ctxt-Here $J = J$ |
Fill Ctxt-Skip $J = Skip$ |
Fill (Ctxt-Assign x e) $J = Assign\ x\ e$ |
Fill (Ctxt-New x c) $J = New\ x\ c$ |
Fill (Ctxt-Get x y f) $J = Get\ x\ y\ f$ |
Fill (Ctxt-Put x f e) $J = Put\ x\ f\ e$ |
Fill (Ctxt-Comp C D) $J = Comp\ (Fill\ C\ J)\ (Fill\ D\ J)$ |
Fill (Ctxt-If b C D) $J = If\ b\ (Fill\ C\ J)\ (Fill\ D\ J)$ |
Fill (Ctxt-While b C) $J = While\ b\ (Fill\ C\ J)$ |
Fill Ctxt-Call $J = Call$

The variables mentioned by an expression:

primrec *EVars*::*Expr* \Rightarrow *Var set*
where
EVars (varE x) = $\{x\}$ |
EVars (valE v) = $\{\}$ |

$$EVars (opE f e1 e2) = EVars e1 \cup EVars e2$$

primrec $BVars::BExpr \Rightarrow Var\ set$

where

$$BVars (compB f e1 e2) = EVars e1 \cup EVars e2$$

The variables possibly read from during the evaluation of I are denoted by $Vars I$.

primrec $Vars::OBJ \Rightarrow Var\ set$

where

$$\begin{aligned} Vars\ Skip &= \{\} \mid \\ Vars\ (Assign\ x\ e) &= EVars\ e \mid \\ Vars\ (New\ x\ C) &= \{\} \mid \\ Vars\ (Get\ x\ y\ f) &= \{y\} \mid \\ Vars\ (Put\ x\ f\ e) &= EVars\ e \mid \\ Vars\ (Comp\ I\ J) &= Vars\ I \cup Vars\ J \mid \\ Vars\ (While\ b\ I) &= BVars\ b \cup Vars\ I \mid \\ Vars\ (Iff\ b\ I\ J) &= BVars\ b \cup Vars\ I \cup Vars\ J \mid \\ Vars\ Call &= \{\} \end{aligned}$$

An abbreviating definition saying when a value is not a constant location.

definition $ValsNoLoc :: Val \Rightarrow bool$

where $ValsNoLoc\ v = (v = RVal\ Nullref \vee (\exists\ i.\ v = IVal\ i))$

Expressions satisfying the following predicate are guaranteed not to return a state-independent location.

primrec $Expr-noLoc::Expr \Rightarrow bool$

where

$$\begin{aligned} Expr-noLoc\ (varE\ x) &= True \mid \\ Expr-noLoc\ (valE\ v) &= ValsNoLoc\ v \mid \\ Expr-noLoc\ (opE\ f\ e1\ e2) &= \\ & (Expr-noLoc\ e1 \wedge Expr-noLoc\ e2 \wedge opEGood\ f) \end{aligned}$$

primrec $BExpr-noLoc::BExpr \Rightarrow bool$

where

$$\begin{aligned} BExpr-noLoc\ (compB\ f\ e1\ e2) &= \\ & (Expr-noLoc\ e1 \wedge Expr-noLoc\ e2 \wedge compBGood\ f) \end{aligned}$$

By induction on e one may show the following three properties.

lemma $Expr-lemma1[rule-format]:$

$$\begin{aligned} Expr-noLoc\ e \longrightarrow EVars\ e \subseteq X \longrightarrow \\ (\forall x. x \in X \longrightarrow CONTEXT\ x = low) \longrightarrow Expr-low\ e \end{aligned}$$

lemma $Expr-lemma2[rule-format]:$

$$\begin{aligned} noLowDPs\ (s, h) \longrightarrow \\ EVars\ e \subseteq X \longrightarrow Expr-noLoc\ e \longrightarrow \\ (\forall x. x \in X \longrightarrow CONTEXT\ x = low) \longrightarrow \\ s \approx_{\beta} t \longrightarrow twiddleHeap\ \beta\ h\ k \longrightarrow \\ noLowDPs\ (\lambda y. if\ x = y\ then\ evalE\ e\ s\ else\ s\ y, h) \end{aligned}$$

lemma *Expr-lemma3*[*rule-format*]:
 $noLowDPs (s,h) \longrightarrow EVars e \subseteq X \longrightarrow Expr-noLoc e \longrightarrow$
 $lookup h l = Some (C, Flds) \longrightarrow$
 $(\forall x. x \in X \longrightarrow CONTEXT x = low) \longrightarrow$
 $s \approx_{\beta} t \longrightarrow twiddleHeap \beta h k \longrightarrow$
 $noLowDPs (s, (l, C, (f, evalE e s) \# Flds) \# h)$

The first of these can be lifted to boolean expressions.

lemma *BExpr-lemma*:
 $\llbracket BVars b \subseteq X; \forall x. x \in X \longrightarrow CONTEXT x = low; BExpr-noLoc b \rrbracket \implies BExpr-low b$

For contexts, we define when a set X of variables is an upper bound for the variables read from. In addition, the *noLoc* condition is imposed on expressions occurring in assignments and field modifications in order to express that if these expressions evaluate to locations then these must stem from lookup operations in the state.

primrec *CtxtVars*:: *Var set* \Rightarrow *CtxtProg* \Rightarrow *bool*

where

$CtxtVars X Ctxt-Here = True$ |
 $CtxtVars X Ctxt-Skip = True$ |
 $CtxtVars X (Ctxt-Assign x e) = (EVars e \subseteq X \wedge Expr-noLoc e)$ |
 $CtxtVars X (Ctxt-New x c) = True$ |
 $CtxtVars X (Ctxt-Get x y f) = (y : X \wedge GAMMA f = low)$ |
 $CtxtVars X (Ctxt-Put x f e) =$
 $(EVars e \subseteq X \wedge CONTEXT x = low \wedge Expr-noLoc e)$ |
 $CtxtVars X (Ctxt-Comp C D) = (CtxtVars X C \wedge CtxtVars X D)$ |
 $CtxtVars X (Ctxt-If b C D) =$
 $(BVars b \subseteq X \wedge CtxtVars X C \wedge CtxtVars X D \wedge BExpr-noLoc b)$ |
 $CtxtVars X (Ctxt-While b C) =$
 $(BVars b \subseteq X \wedge CtxtVars X C \wedge BExpr-noLoc b)$ |
 $CtxtVars X Ctxt-Call = True$

A context is "obviously" low if all accessed variables are (contained in a set X whose members are) low.

definition *LOW*:: *Var set* \Rightarrow *CtxtProg* \Rightarrow *bool*

where $LOW X C = (CtxtVars X C \wedge (\forall x. x : X \longrightarrow CONTEXT x = low))$

Finally, we obtain the following result by induction on an upper bound on the derivation heights of the two executions of *Fill C I*.

theorem *secureI-secureFillI*:

$\llbracket secure I; LOW X C; LOW X Ctxt-Body; body = Fill Ctxt-Body I \rrbracket$
 $\implies secure (Fill C I)$

For a variable

consts *res*:: *Var*

representing the output of the attacking context, the result specialises to

theorem *SecureForAttackingContext*:

\llbracket *secure I; LOW X C; LOW X Ctxt-Body; s \equiv_{β} ss;*
s,(Fill C I) \Downarrow t; ss,(Fill C I) \Downarrow tt; body = Fill Ctxt-Body I;
*CONTEXT res = low \rrbracket
 $\implies \exists \gamma. (\gamma, (fst\ t)\ res, (fst\ tt)\ res):twiddleVal \wedge Pbij\text{-extends } \gamma\ \beta$*

End of theory ContextObj

end

References

- [1] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005.
- [2] L. Beringer and M. Hofmann. Secure information flow and program logics. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF 2007)*, pages 233–248. IEEE Computer Society, 2007.
- [3] MOBIUS. Consortium. Deliverable 2.3: Report on type systems, 2007. Available online from <http://mobius.inria.fr>.
- [4] S. Hunt and D. Sands. On flow-sensitive security types. In J. G. Morrisett and S. L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*, pages 79–90. ACM Press, 2006.
- [5] T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, School of Informatics, Sept. 1998. Technical Report ECS-LFCS-98-392.
- [6] T. Nipkow. Hoare logics for recursive procedures and unbounded non-determinism. In J. Bradfield, editor, *Computer Science Logic (CSL 2002)*, volume 2471 of *Lecture Notes in Computer Science*, pages 103–119. Springer, 2002.
- [7] T. Nipkow. Abstract Hoare logics. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. <http://isa-afp.org/entries/Abstract-Hoare-Logics.shtml>, June 2008. Formal proof development.
- [8] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

- [9] G. Winskel. *The formal semantics of programming languages, an introduction*. MIT Press, 1993.