Secure information flow and program logics — Isabelle/HOL sources

Lennart Beringer and Martin Hofmann

March 19, 2025

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 . . . 8

3 Base-line noninterference **9**
3.1 Basic definitions . . . 9
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 12
3.5 Contextual closure 16

4 Lattices 18

5 Flow-sensitivity a la Hunt and Sands 18
5.1 General $A; R \Rightarrow S$-security 19
5.2 Basic definitions 19
5.3 Type system 20
5.4 Derived proof rules 21
5.5 Soundness results 26

6 Base-line non-interference with objects 28
6.1 Syntax and operational semantics 28
6.2 Program logic 31
6.2.1 Assertions and their semantic validity 31
6.2.2 Proof system 32
6.2.3 Soundness 34
6.2.4 Derived rules 34
6.2.5 Completeness 34
6.3 Partial bijections 35
6.4 Non-interference 36
6.4.1 Indistinguishability relations 36
6.4.2 Definition and characterisation of security 38
6.5 Derivation of proof rules 39
6.5.1 Low proof rules 39
6.5.2 High proof rules 41
6.6 Type system 44
6.7 Contextual closure 46

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.

typedecl *Var*
typedecl *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* ⇒ *Val* ⇒ *Val Expr Expr*

datatype *BExpr* = *compB Val* ⇒ *Val* ⇒ *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* ⇒ *Val*

definition *update* :: *State* ⇒ *Var* ⇒ *Val* ⇒ *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* ⇒ *State* ⇒ *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* ⇒ *State* ⇒ *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* × *IMP* × *nat* × *State*) *set* **where**
SemSkip: (*s*,*Skip*,*1*,*s*) : *Semn*

| *SemAssign*:
⟦ *t* = *update s x* (*evalE e s*)⟧ ⟹ (*s*,*Assign x e*,*1*,*t*):*Semn*

| *SemComp*:
⟦ (*s*,*c1*,*n*,*r*):*Semn*; (*r*,*c2*,*m*,*t*):*Semn*; *k*=(*max n m*)+*1*⟧
⟹ (*s*,*Comp c1 c2*,*k*,*t*):*Semn*

| *SemWhileT*:
⟦*evalB b s*; (*s*,*c*,*n*,*r*):*Semn*; (*r*,*While b c*,*m*,*t*):*Semn*;
k=((*max n m*)+*1*)⟧
⟹ (*s*,*While b c*,*k*,*t*):*Semn*

| *SemWhileF*: ⟦¬ (*evalB b s*); *t*=*s*⟧ ⟹ (*s*,*While b c*,*1*,*t*):*Semn*

| *SemTrue*:
⟦*evalB b s*; (*s*,*c1*,*n*,*t*):*Semn* ⟧ ⟹ (*s*,*Iff b c1 c2*,*n*+*1*,*t*):*Semn*

| *SemFalse*:
⟦¬ (*evalB b s*); (*s*,*c2*,*n*,*t*):*Semn*⟧ ⟹ (*s*,*Iff b c1 c2*,*n*+*1*,*t*):*Semn*

| *SemCall*: (*s*,*body*,*n*,*t*):*Semn* ⟹ (*s*,*Call*,*n*+*1*,*t*):*Semn*

abbreviation
SemN :: [*State*, *IMP*, *nat*, *State*] ⇒ *bool* (‹ - , - $\rightarrow_{-}$ - ›)
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*] ⇒ *bool* (‹- , - ⇓ - › *1000*)
where *s*,*c* ⇓ *t* = (∃ *n*. *s*,*c* $\rightarrow_n$ *t*)

Inductive elimination rules for the (indexed) dynamic semantics:

inductive-cases *Sem-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*

$s,\ \mathit{Call} \rightarrow_n\ t$
$\langle \mathit{proof} \rangle$

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

lemma *Sem-no-zero-height-derivs*: $(s,\ c \rightarrow_0\ t)$ ==> $\mathit{False}\langle \mathit{proof}\rangle\langle \mathit{proof}\rangle$

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

lemma *SemDeterm*: $[\![s,\ c \Downarrow t;\ s,\ c \Downarrow r]\!] \Longrightarrow r{=}t\langle \mathit{proof}\rangle$

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* = *State* ⇒ *State* ⇒ *bool*

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

definition *VDM-valid* :: *IMP* ⇒ *VDMAssn* ⇒ *bool*
(‹ ⊨ - : - › [*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* :: *nat* ⇒ *IMP* ⇒ *VDMAssn* ⇒ *bool*
(‹ $\models_{\text{-}}$ - : - › [*100*,*100*,*100*] *100*)
where $\models_n\ c : A = (\forall\ m\ .\ m \leq n$ --> $(\forall\ s\ t\ .\ (s{,}c \rightarrow_m t)$ --> $A\ s\ t))$

. . . plus the obvious relationships.

lemma *VDM-valid-validn*: $\models c{:}A \Longrightarrow \models_n c{:}A\langle \mathit{proof}\rangle$
lemma *VDM-validn-valid*: $(\forall\ n\ .\ \models_n c{:}A) \Longrightarrow \models c{:}A\langle \mathit{proof}\rangle$
lemma *VDM-lowerm*: $[\![\ \models_n c{:}A;\ m \leq n\ ]\!] \Longrightarrow \models_m c{:}A\langle \mathit{proof}\rangle$

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* ⇒ *bool* (‹ ⊨ - › [*100*] *100*)
where $\models G = (\forall\ A\ .\ A \in G \longrightarrow (\models Call : A))$

Again, a relativised sibling ...

definition *Ctxt-validn* :: *nat* ⇒ (*VDMAssn set*) ⇒ *bool*
(‹ ⊨_ - › [*100*,*100*] *100*)
where $\models_n G = (\forall\ m\ .\ m \le n \longrightarrow (\forall\ A.\ A \in G \longrightarrow (\models_m Call : A)))$

satisfies the obvious properties.

lemma *Ctxt-valid-validn*: $\models G \Longrightarrow \models_n G$⟨*proof*⟩
lemma *Ctxt-validn-valid*: $(\forall\ n\ .\ \models_n G) \Longrightarrow \models G$⟨*proof*⟩
lemma *Ctxt-lowerm*: $[\![\models_n G;\ m < n]\!] \Longrightarrow \models_m G$⟨*proof*⟩

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

definition *valid* :: (*VDMAssn set*) ⇒ *IMP* ⇒ *VDMAssn* ⇒ *bool*
(‹- ⊨ - : - › [*100*,*100*,*100*] *100*)
where $G \models c : A = (\models G \longrightarrow \models c : A)$

And, again, a relatived notion of judgement validity.

definition *validn* ::
(*VDMAssn set*) ⇒ *nat* ⇒ *IMP* ⇒ *VDMAssn* ⇒ *bool*
(‹- ⊨_ - : -› [*100*,*100*,*100*,*100*] *100*)
where $G \models_n c : A = (\models_n G \longrightarrow \models_n c : A)$

lemma *validn-valid*: $(\forall\ n\ .\ G \models_n c : A) \Longrightarrow G \models c : A$⟨*proof*⟩
lemma *ctxt-consn*: $[\![\models_n G;\ \models_n Call{:}A]\!] \Longrightarrow \models_n (\{A\} \cup G)$⟨*proof*⟩

2.2 Proof system

inductive-set
VDM-proof :: (*VDMAssn set* × *IMP* × *VDMAssn*) *set*
where

VDMSkip: (*G*, *Skip*, λ *s t* . *t*=*s*) : *VDM-proof*

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

| *VDMComp*:
⟦ (*G*, *c1*, *A1*) : *VDM-proof*; (*G*, *c2*, *A2*) : *VDM-proof*⟧ ⟹
(*G*, *Comp c1 c2*, λ *s t* . ∃ *r* . *A1 s r* ∧ *A2 r t*) : *VDM-proof*

| *VDMIff*:
⟦ (*G*, *c1*, *A*):*VDM-proof*; (*G*, *c2*, *B*):*VDM-proof*⟧ ⟹
(*G*, *Iff b c1 c2*, λ *s t* . (((*evalB b s*) ⟶ *A s t*) ∧
((¬ (*evalB b s*)) ⟶ *B s t*))) : *VDM-proof*

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

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

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

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

abbreviation
VDM-deriv :: [*VDMAssn set*, *IMP*, *VDMAssn*] $\Rightarrow$ *bool*
(‹- $\rhd$ - : -› [*100*,*100*,*100*] *100*)
where $G \rhd c : A == (G,c,A) \in$ *VDM-proof*

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

lemma *Hoare-While*:
$G \rhd c : (\lambda\ s\ s'\ .\ \forall\ r\ .\ evalB\ b\ s \longrightarrow I\ s\ r \longrightarrow I\ s'\ r) \Longrightarrow$
$G \rhd While\ b\ c : (\lambda\ s\ s'.\ \forall\ r\ .\ I\ s\ r \longrightarrow (I\ s'\ r \wedge \neg\ evalB\ b\ s'))$
⟨*proof*⟩

Here's the proof in the opposite direction.

lemma *VDMWhile-derivable*:
$\llbracket$ $G \rhd c : B$; $\forall\ s\ .\ (\neg\ evalB\ b\ s) \longrightarrow A\ s\ s$;
$\forall\ s\ r\ t.\ evalB\ b\ s \longrightarrow B\ s\ r \longrightarrow A\ r\ t \longrightarrow A\ s\ t$ $\rrbracket$
$\Longrightarrow G \rhd (While\ b\ c) : (\lambda\ s\ t\ .\ A\ s\ t \wedge \neg\ (evalB\ b\ t))$
⟨*proof*⟩

2.3 Soundness

⟨*proof*⟩⟨*proof*⟩

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

lemma *SoundWhile*[*rule-format*]:
$(\forall m.\ G \models_m c : B) \longrightarrow (\forall s.\ (\neg\ evalB\ b\ s) \longrightarrow A\ s\ s) \longrightarrow$
$(\forall s.\ evalB\ b\ s \longrightarrow (\forall r.\ B\ s\ r \longrightarrow (\forall t.\ A\ r\ t \longrightarrow A\ s\ t))) \longrightarrow$
$G \models_n (While\ b\ c) : (\lambda s\ t.\ A\ s\ t \wedge \neg\ evalB\ b\ t)$⟨*proof*⟩

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

lemma *SoundCall*[*rule-format*]:
$\llbracket\forall n.\ \models_n (\{A\} \cup G) \longrightarrow\ \models_n body : A\rrbracket \Longrightarrow\ \models_n G \longrightarrow\ \models_n Call : A$⟨*proof*⟩

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

lemma *VDM-Sound-n*: $G \rhd c\text{: } A \Longrightarrow (\forall\ n\ .\ G \models_n c\text{:}A)\langle proof\rangle$

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

theorem *VDM-Sound*: $G \rhd c\text{: } A \Longrightarrow G \models c\text{:}A\langle proof\rangle$

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

lemma *VDM-Sound-emptyCtxt*: $\{\} \rhd c : A \Longrightarrow \models c : A\langle proof\rangle$

2.4 Admissible rules

A weakening rule and some cut rules are easily derived.

lemma *WEAK*[*rule-format*]:
$G \rhd c : A \Longrightarrow (\forall\ H\ .\ G \subseteq H \longrightarrow H \rhd c : A)\langle proof\rangle\langle proof\rangle$
lemma *CutAux*:
$[\![H \rhd c : A;\ H = insert\ P\ D;\ G \rhd Call : P;\ G \subseteq D]\!] \Longrightarrow D \rhd c\text{:}A\langle proof\rangle$
lemma *Cut*: $[\![\ G \rhd Call : P\ ;\ (insert\ P\ G) \rhd c : A\ ]\!] \Longrightarrow G \rhd c : A\langle proof\rangle$

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

definition *verified*:: *VDMAssn set* ⇒ *bool*
where $verified\ G = (\forall\ A\ .\ A\text{:}G \longrightarrow G \rhd body : A)$

The property is preserved by sub-contexts

lemma *verified-preserved*: $[\![verified\ G;\ A\text{:}G]\!] \Longrightarrow verified\ (G - \{A\})\langle proof\rangle\langle proof\rangle\langle proof\rangle$

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

theorem *Mutrec*:
$[\![\ finite\ G;\ card\ G = n;\ verified\ G;\ A : G\ ]\!] \Longrightarrow \{\} \rhd Call\text{:}A\langle proof\rangle$

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

lemma *Ctxt-verified-valid*: $[\![verified\ G;\ finite\ G]\!] \Longrightarrow \models G\langle proof\rangle$

2.5 Completeness

Strongest specifications, given precisely by the operational behaviour.

definition *SSpec*::*IMP* ⇒ *VDMAssn*
where $SSpec\ c\ s\ t = s,c \Downarrow t$

Strongest specifications are valid . . .

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

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

lemma *SSpec-strong*: $\models c : A \Longrightarrow \forall\ s\ t\ .\ SSpec\ c\ s\ t \longrightarrow A\ s\ t\langle proof\rangle$

By induction on c we show the following.

lemma *SSpec-derivable*: $G \rhd Call : SSpec\ Call \Longrightarrow G \rhd c : SSpec\ c \langle proof \rangle$

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*: $StrongG \rhd body : SSpec\ Call \langle proof \rangle$

Thus, the strong context is verified.

lemma *StrongG-verified*: *verified StrongG*⟨*proof*⟩

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*: $\{\} \rhd c : SSpec\ c \langle proof \rangle$

From this, we easily obtain (relative) completeness.

theorem *VDM-Complete*: $\models c : A \Longrightarrow \{\} \rhd c : A \langle proof \rangle$

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

lemma *Ctxt-valid-verified*: $\models G \Longrightarrow$ *verified G*⟨*proof*⟩

End of theory VDM

end

theory *VS* **imports** *VDM* **begin**

3 Base-line noninterference

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

3.1 Basic definitions

Muli-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* (‹ - $\approx$ - › [*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*: $\models c : (Sec\ \Phi) \Longrightarrow secure\ c\langle proof\rangle$
lemma *Prop1B*:
$secure\ c \Longrightarrow\ \models c : Sec\ (\lambda\ (r,\ t)\ .\ \exists\ s\ .\ (s\ ,\ c \Downarrow r) \wedge s \approx t)\langle proof\rangle$
lemma *Prop1BB*: $secure\ c \Longrightarrow \exists\ \Phi\ .\ \models c : Sec\ \Phi\langle proof\rangle$
lemma *Prop1*:
$(secure\ c) = (\models c : Sec\ (\lambda\ (r,\ t)\ .\ \exists\ s\ .\ (s\ ,\ c \Downarrow r) \wedge s \approx t))\langle proof\rangle$

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*]:
$G \rhd c : twiddle \longrightarrow G \rhd c : Sec\ (\lambda\ (s,t)\ .\ s \approx t)\langle proof\rangle$
lemma *SKIP*: $G \rhd Skip : Sec\ (\lambda\ (s,t)\ .\ s \approx t)\langle proof\rangle$
lemma *ASSIGN*:
$(\forall\ s\ ss.\ s \approx ss \longrightarrow evalE\ e\ s = evalE\ e\ ss) \Longrightarrow$
$G \rhd (Assign\ x\ e)$
$: (Sec\ (\lambda\ (s,\ t)\ .\ s \approx (update\ t\ x\ (evalE\ e\ t))))\langle proof\rangle$
lemma *COMP*:
$[\![\ G \rhd c1 : (Sec\ \Phi);\ G \rhd c2 : (Sec\ \Psi)]\!] \Longrightarrow$
$G \rhd (Comp\ c1\ c2) : (Sec\ (\lambda\ (s,t)\ .\ \exists\ r\ .\ \Phi(r,\ t) \wedge$
$(\forall\ w\ .\ (r \approx w \longrightarrow \Psi(s,\ w)))))\langle proof\rangle$
lemma *IFF*:
$[\![\ (\forall\ s\ ss.\ s \approx ss \longrightarrow evalB\ b\ s = evalB\ b\ ss);$
$G \rhd c1 : (Sec\ \Phi);\ G \rhd c2 : (Sec\ \Psi)]\!] \Longrightarrow$
$G \rhd (Iff\ b\ c1\ c2) : Sec\ (\lambda\ (s,\ t)\ .\ (evalB\ b\ t \longrightarrow \Phi(s,t)) \wedge$
$((\neg\ evalB\ b\ t) \longrightarrow \Psi(s,t)))\langle proof\rangle$

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

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

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

definition $Monotone{::}(TT \Rightarrow TT) \Rightarrow bool$
where $Monotone\ \varphi = (\forall\ \Phi\ \Psi\ .\ (\forall\ s\ t\ .\ \Phi(s,t) \longrightarrow \Psi(s,t)) \longrightarrow (\forall\ s\ t\ .\ \varphi\ \Phi\ (s,t) \longrightarrow \varphi\ \Psi\ (s,t)))$
⟨*proof*⟩⟨*proof*⟩

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

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

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

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

Since this operator is monotone, ...

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

we may define its fixed point,

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

which we can use to derive the following rule.

lemma *WHILE*:
$[\![\ (\forall\ s\ t.\ s \approx t \longrightarrow evalB\ b\ s = evalB\ b\ t);\ G \rhd c : (Sec\ \Phi)]\!] \Longrightarrow$
$G \rhd (While\ b\ c) : (Sec\ (PhiWhile\ b\ \Phi))$⟨*proof*⟩

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*: $Monotone\ (\lambda\ \Phi\ .\ PhiWhile\ b\ \Phi)$⟨*proof*⟩

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: $[\![\neg$ *evalB b t*; $s \approx t]\!] \Longrightarrow (b,\Phi,s,t)$:*var*
| *varTrue*:$[\![$*evalB b t*; $\Phi(r,t)$; $\forall$ w . $r \approx w \longrightarrow (b,\Phi,s,w)$: *var*$]\!]$
$\Longrightarrow (b,\Phi,s,t)$:*var*

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

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩
lemma *FIXvarFIX*: (*PhiWhile b*) = (λ Φ . (λ (s,t) . (b,Φ,s,t):*var*))⟨*proof*⟩

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

lemma *WHILE-IND*:
$[\![$ ($\forall$ *s t*. $s \approx t \longrightarrow$ *evalB b s* = *evalB b t*); $G \rhd c$: (*Sec* Φ)$]\!] \Longrightarrow$
$G \rhd$ (*While b c*) : (*Sec* (λ (s,t) . (b,Φ,s,t):*var*))⟨*proof*⟩

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

⟨*proof*⟩
lemma *var-Monotone*: *Monotone* (λ Φ . (λ (s,t) .(b,Φ,s,t):*var*))⟨*proof*⟩⟨*proof*⟩

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

lemma *CALL*:
$[\![$ ({*Sec*(*FIX* Φ)} $\cup$ G) $\rhd$ *body* : *Sec*(Φ (*FIX* Φ)); *Monotone* $\Phi]\!] \Longrightarrow$
$G \rhd$ *Call* : *Sec*(*FIX* Φ)⟨*proof*⟩

3.3 Derivation of the HIGH rules

The HIGH rules are easy.

lemma *HIGH-SKIP*: $G \rhd$ *Skip* : *twiddle*⟨*proof*⟩
lemma *HIGH-ASSIGN*:
CONTEXT x = *high* $\Longrightarrow$ $G \rhd$ (*Assign x e*) : *twiddle*⟨*proof*⟩
lemma *HIGH-COMP*:
$[\![$ $G \rhd$ *c1* : *twiddle*; $G \rhd$ *c2* : *twiddle*$]\!]$
$\Longrightarrow$ $G \rhd$ (*Comp c1 c2*): *twiddle*⟨*proof*⟩
lemma *HIGH-IFF*:
$[\![$ $G \rhd$ *c1* : *twiddle*; $G \rhd$ *c2* : *twiddle* $]\!]$
$\Longrightarrow$ $G \rhd$ (*Iff b c1 c2*) : *twiddle*⟨*proof*⟩
lemma *HIGH-WHILE*:
$[\![$ $G \rhd c$: *twiddle*$]\!] \Longrightarrow$ $G \rhd$ (*While b c*) : *twiddle*⟨*proof*⟩
lemma *HIGH-CALL*:
({*twiddle*} $\cup$ G) $\rhd$ *body* : *twiddle* $\Longrightarrow$ $G \rhd$ *Call* : *twiddle*⟨*proof*⟩

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-expr* :: (*Expr* × *TP*) *set*
where
VS-exprVar: *CONTEXT x* = *t* ⟹ (*varE x*, *t*) : *VS-expr*
| *VS-exprVal*: (*valE v*, *low*) : *VS-expr*
| *VS-exprOp*: ⟦(*e1*,*t*) : *VS-expr*; (*e2*,*t*):*VS-expr*⟧
⟹ (*opE f e1 e2*,*t*) : *VS-expr*
| *VS-exprHigh*: (*e*, *high*) : *VS-expr*

inductive-set *VS-Bexpr* :: (*BExpr* × *TP*) *set*
where
VS-BexprOp: ⟦(*e1*,*t*) : *VS-expr*; (*e2*,*t*):*VS-expr*⟧
⟹ (*compB f e1 e2*,*t*) : *VS-Bexpr*
| *VS-BexprHigh*: (*e*,*high*) : *VS-Bexpr*

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

inductive-set *VS-com* :: (*TP* × *IMP*) *set*
where

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

| *VS-comAssHigh*:
CONTEXT x = *high* ⟹ (*pc*,*Assign x e*) : *VS-com*

| *VS-comAssLow*:
⟦*CONTEXT x* = *low*; *pc* = *low*; (*e*,*low*):*VS-expr*⟧ ⟹
(*pc*,*Assign x e*) : *VS-com*

| *VS-comComp*:
⟦ (*pc*,*c1*):*VS-com*; (*pc*,*c2*):*VS-com*⟧ ⟹
(*pc*,*Comp c1 c2*) : *VS-com*

| *VS-comIf*:
⟦ (*b*,*pc*):*VS-Bexpr*; (*pc*,*c1*):*VS-com*; (*pc*,*c2*):*VS-com*⟧ ⟹
(*pc*,*Iff b c1 c2*):*VS-com*

| *VS-comWhile*:
⟦(*b*,*pc*):*VS-Bexpr*; (*pc*,*c*):*VS-com*⟧ ⟹ (*pc*,*While b c*):*VS-com*

| *VS-comSub*: (*high*,*c*) : *VS-com* ⟹ (*low*,*c*):*VS-com*

We define the interpretation of expression typings...

primrec *SemExpr*::*Expr* ⇒ *TP* ⇒ *bool*
where
SemExpr e low = (∀ *s ss*. *s* ≈ *ss* ⟶ *evalE e s* = *evalE e ss*) |
SemExpr e high = *True*

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

lemma *ExprSound*: (*e*,*tp*):*VS-expr* ⟹ *SemExpr e tp*⟨*proof*⟩

Likewise for the boolean expressions.

primrec *SemBExpr*::*BExpr* ⇒ *TP* ⇒ *bool*
where
SemBExpr b low = (∀ *s ss*. *s* ≈ *ss* ⟶ *evalB b s* = *evalB b ss*) |
SemBExpr b high = *True*

lemma *BExprSound*: (*e*,*tp*):*VS-Bexpr* ⟹ *SemBExpr e tp*⟨*proof*⟩

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

theorem *VS-com-VDM*[*rule-format*]:
(*t*,*c*):*VS-com* ⟹ (*t*=*high* ⟶ *G* ▷ *c* : *twiddle*) ∧
(*t*=*low* ⟶ (∃ *A* . *G* ▷ *c* : *Sec A*))⟨*proof*⟩

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* ⇒ *IMP* ⇒ *bool*
where
SemCom low c = (∀ *s ss t tt*. *s* ≈ *ss* ⟶ (*s*,*c* ⇓ *t*) ⟶
(*ss*,*c* ⇓ *tt*) ⟶ *t* ≈ *tt*) |
SemCom high c = (∀ *s t* . (*s*,*c* ⇓ *t*) ⟶ *s* ≈ *t*)

Combining theorem *VS-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-SOUND*: (*t*,*c*):*VS-com* ⟹ *SemCom t c*⟨*proof*⟩

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* × *IMP* × *VDMAssn*) *set*
where
D-CAST:
(*G*,*c*,*twiddle*):*Deriv* ⟹ (*G*, *c*, *Sec* (λ (*s*,*t*) . *s* ≈ *t*)) : *Deriv*

| *D-SKIP*: (*G*, *Skip*, *Sec* (λ (*s*,*t*) . *s* ≈ *t*)) : *Deriv*

| *D-ASSIGN*:
(∀ *s ss*. *s* ≈ *ss* ⟶ *evalE e s* = *evalE e ss*) ⟹
(*G*, *Assign x e*, *Sec* (λ (*s*, *t*) . *s* ≈ (*update t x* (*evalE e t*)))):*Deriv*

| *D-COMP*:
⟦ (*G*, *c1*, *Sec* Φ):*Deriv*; (*G*, *c2*, *Sec* Ψ):*Deriv*⟧ ⟹
(*G*, *Comp c1 c2*, *Sec* (λ (*s*,*t*) . ∃ *r* . Φ(*r*, *t*) ∧
(∀ *w* . (*r* ≈ *w* ⟶ Ψ(*s*, *w*))))):*Deriv*

| *C-IFF*:
⟦ (∀ *s ss*. *s* ≈ *ss* ⟶ *evalB b s* = *evalB b ss*);
(*G*, *c1*, *Sec* Φ):*Deriv*; (*G*,*c2*, *Sec* Ψ):*Deriv*⟧ ⟹
(*G*, *Iff b c1 c2*, *Sec* (λ (*s*, *t*) . (*evalB b t* ⟶ Φ(*s*,*t*)) ∧
((¬ *evalB b t*) ⟶ Ψ(*s*,*t*)))):*Deriv*

| *D-WHILE*:
⟦ (∀ *s ss*. *s* ≈ *ss* ⟶ *evalB b s* = *evalB b ss*);
(*G*, *c*, *Sec* Φ):*Deriv*⟧ ⟹
(*G*, *While b c*, *Sec* (*PhiWhile b* Φ)):*Deriv*

| *D-CALL*:
⟦ ({*Sec*(*FIX* Φ)} ∪ *G*, *body*, *Sec*(Φ(*FIX* Φ))):*Deriv*;
Monotone Φ⟧ ⟹
(*G*, *Call*, *Sec*(*FIX* Φ)):*Deriv*

| *D-HighSKIP*:(*G*, *Skip*, *twiddle*):*Deriv*

| *D-HighASSIGN*:
CONTEXT x = *high* ⟹ (*G*,*Assign x e*, *twiddle*):*Deriv*

| *D-HighCOMP*:
⟦ (*G*,*c1*,*twiddle*):*Deriv*; (*G*,*c2*,*twiddle*):*Deriv*⟧ ⟹
(*G*, *Comp c1 c2*, *twiddle*):*Deriv*

| *D-HighIFF*:
⟦ (*G*,*c1*,*twiddle*):*Deriv*; (*G*,*c2*,*twiddle*):*Deriv*⟧ ⟹
(*G*, *Iff b c1 c2*, *twiddle*):*Deriv*

| *D-HighWHILE*:
(*G*, *c*, *twiddle*):*Deriv* ⟹ (*G*, *While b c*, *twiddle*):*Deriv*

| *D-HighCALL*:
({*twiddle*} ∪ *G*, *body*, *twiddle*):*Deriv* ⟹ (*G*, *Call*, *twiddle*):*Deriv*
⟨*proof*⟩
lemma *DerivMono*:
(*X*,*c*,*A*):*Deriv* ⟹ ∃ Φ . *A* = *Sec* (Φ (*FIX* Φ)) ∧ *Monotone* Φ⟨*proof*⟩

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

lemma *Deriv-derivable*: (*G*,*c*,*A*):*Deriv* ⟹ *G* ▷ *c*: *A*⟨*proof*⟩

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* ⇒ *IMP* ⇒ *IMP*
where
Fill Ctxt-Hole c = *c* |
Fill Ctxt-Skip c = *Skip* |
Fill (*Ctxt-Assign x e*) *c* = *Assign x e* |
Fill (*Ctxt-Comp C1 C2*) *c* = *Comp* (*Fill C1 c*) (*Fill C2 c*) |
Fill (*Ctxt-If b C1 C2*) *c* = *Iff b* (*Fill C1 c*) (*Fill C2 c*) |
Fill (*Ctxt-While b C*) *c* = *While b* (*Fill C c*) |
Fill Ctxt-Call c = *Call*

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

primrec *EVars*::*Expr* ⇒ *Var set*
where
EVars (*varE x*) = {*x*} |
EVars (*valE v*) = {} |
EVars (*opE f e1 e2*) = *EVars e1* ∪ *EVars e2*

lemma *low-Eval*[*rule-format*]:
 (∀ *x* . *x* ∈ *EVars e* ⟶ *CONTEXT x* = *low*) ⟶
  (∀ *s t* . *s* ≈ *t* ⟶ *evalE e s* = *evalE e t*)⟨*proof*⟩
primrec *BVars*::*BExpr* ⇒ *Var set*
where
BVars (*compB f e1 e2*) = *EVars e1* ∪ *EVars e2*

lemma *low-EvalB*[*rule-format*]:
 (∀ *x* . *x* ∈ *BVars b* ⟶ *CONTEXT x* = *low*) ⟶
  (∀ *s t* . *s* ≈ *t* ⟶ *evalB b s* = *evalB b t*)⟨*proof*⟩

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* ⇒ *Var set*
where
Vars Skip = {} |
Vars (*Assign x e*) = *EVars e* |
Vars (*Comp c d*) = *Vars c* ∪ *Vars d* |
Vars (*While b c*) = *BVars b* ∪ *Vars c* |
Vars (*Iff b c d*) = *BVars b* ∪ *Vars c* ∪ *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* ⇒ *CtxtProg* ⇒ *bool*
where
CtxtVars X Ctxt-Hole = *True* |
CtxtVars X Ctxt-Skip = *True* |
CtxtVars X (*Ctxt-Assign x e*) = (*EVars e* ⊆ *X*) |
CtxtVars X (*Ctxt-Comp C1 C2*) = (*CtxtVars X C1* ∧ *CtxtVars X C2*) |
CtxtVars X (*Ctxt-If b C1 C2*) = (*BVars b* ⊆ *X* ∧ *CtxtVars X C1*
∧ *CtxtVars X C2*) |
CtxtVars X (*Ctxt-While b C*) = (*BVars b* ⊆ *X* ∧ *CtxtVars X C*) |
CtxtVars X Ctxt-Call = *True*
⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

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* ⇒ *CtxtProg* ⇒ *bool*
where *LOW X C* = (*CtxtVars X C* ∧ (∀ *x* . *x* : *X* ⟶ *CONTEXT x* = *low*))
⟨*proof*⟩

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-secureFillI*:
⟦*secure c*; *LOW X C*; *LOW X Ctxt-Body*; *body* = *Fill Ctxt-Body c*⟧
⟹ *secure* (*Fill C c*)⟨*proof*⟩

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

consts *res*::*Var*

theorem

$[\![$ *secure c*; *LOW X C*; *LOW X Ctxt-Body*; $s \approx ss$; s,(*Fill C c*)$\Downarrow t$;
ss,(*Fill C c*)$\Downarrow tt$; *body* = *Fill Ctxt-Body c*; *CONTEXT res* = *low*$]\!]$
$\Longrightarrow$ *t res* = *tt res*⟨*proof*⟩

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 $\bot$, $\sqsubseteq$ 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* $\Longrightarrow$ *LEQ p2 p3* $\Longrightarrow$ *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.

[1] As 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.

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 $Prop4A$: $\models c : Sec\ A\ R\ S\ \Phi \Longrightarrow ARSsecure\ A\ R\ S\ c\langle proof\rangle$
lemma $Prop4B$: $ARSsecure\ A\ R\ S\ c \Longrightarrow \models c : Sec\ A\ R\ S\ (\lambda\ (r,t)\ .\ \exists\ s\ .\ (s\ ,\ c \Downarrow r) \wedge R\ s\ t)\langle proof\rangle$

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 $upd\ G\ x\ p = (\lambda\ y\ .\ if\ x{=}y\ then\ p\ else\ G\ y)$

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 $EQ\text{-}LEQ$: $[\![EQ\ G\ p\ s\ t;\ LEQ\ pp\ p]\!] \Longrightarrow EQ\ G\ pp\ s\ t\langle proof\rangle$

The assertion called $\mathcal{Q}$ 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 amounds 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*)⟨*proof*⟩

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-E*:: (*CONTEXT* $\times$ *Expr* $\times$ *L*) *set*
where
HS-E-var: (*G*, *varE x*, *G x*) : *HS-E*
| *HS-E-val*: (*G*, *valE c*, *bottom*) : *HS-E*
| *HS-E-op*: ⟦(*G*, *e1*,*p1*):*HS-E*; (*G*, *e2*,*p2*):*HS-E*; *p*= *LUB p1 p2*⟧
$\Longrightarrow$ (*G*,*opE f e1 e2*,*p*) : *HS-E*
| *HS-E-sup*: ⟦(*G*,*e*,*p*):*HS-E*; *LEQ p q*⟧ $\Longrightarrow$ (*G*,*e*,*q*):*HS-E*

lemma *HS-E-eval*[*rule-format*]:
(*G*, *e*, *t*) $\in$ *HS-E* $\Longrightarrow$
$\forall$ *r s q*. *EQ G q r s* $\longrightarrow$ *LEQ t q* $\longrightarrow$ *evalE e r* = *evalE e s*⟨*proof*⟩

Likewise for boolean expressions:

inductive-set *HS-B*:: (*CONTEXT* $\times$ *BExpr* $\times$ *L*) *set*
where
HS-B-compB: ⟦(*G*, *e1*,*p1*):*HS-E*; (*G*, *e2*,*p2*):*HS-E*; *p*= *LUB p1 p2*⟧
$\Longrightarrow$ (*G*,*compB f e1 e2*,*p*) : *HS-B*
| *HS-B-sup*: ⟦(*G*,*b*,*p*):*HS-B*; *LEQ p q*⟧ $\Longrightarrow$ (*G*,*b*,*q*):*HS-B*

lemma *HS-B-eval*[*rule-format*]:
(*G*, *b*, *t*) $\in$ *HS-B* $\Longrightarrow$

$\forall$ *r s pp* . *EQ G pp r s* $\longrightarrow$ *LEQ t pp* $\longrightarrow$ *evalB b r* = *evalB b s*⟨*proof*⟩

The typing rules for commands follow.

inductive-set *HS*::(*L* × *CONTEXT* × *IMP* × *CONTEXT*) *set*
where
HS-Skip: (*p*,*G*,*Skip*,*G*):*HS*

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

| *HS-Seq*:
⟦(*p*,*G*,*c*,*K*):*HS*; (*p*,*K*,*d*,*H*):*HS*⟧ $\Longrightarrow$ (*p*,*G*, *Comp c d*,*H*):*HS*

| *HS-If*:
⟦(*G*,*b*,*t*):*HS-B*; (*LUB p t*,*G*,*c*,*H*):*HS*; (*LUB p t*,*G*,*d*,*H*):*HS*⟧ $\Longrightarrow$
(*p*,*G*,*Iff b c d*,*H*):*HS*

| *HS-If-alg*:
⟦(*G*,*b*,*p*):*HS-B*; (*p*,*G*,*c*,*H*):*HS*; (*p*,*G*,*d*,*H*):*HS*⟧ $\Longrightarrow$
(*p*,*G*,*Iff b c d*,*H*):*HS*

| *HS-While*:
⟦(*G*,*b*,*t*):*HS-B*; (*LUB p t*,*G*,*c*,*H*):*HS*;*H*=*G*⟧ $\Longrightarrow$
(*p*,*G*,*While b c*,*H*):*HS*

| *HS-Sub*:
⟦ (*pp*,*GG*,*c*,*HH*):*HS*; *LEQ p pp*; $\forall$ *x* . *LEQ* (*G x*) (*GG x*);
$\forall$ *x* . *LEQ* (*HH x*) (*H x*)⟧ $\Longrightarrow$
(*p*,*G*,*c*,*H*):*HS*

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

lemma *IF-derivable-from-If-alg*:
⟦(*G*,*b*,*t*):*HS-B*; (*LUB p t*,*G*,*c1*,*H*):*HS*; (*LUB p t*,*G*,*c2*,*H*):*HS*⟧
$\Longrightarrow$ (*p*,*G*,*Iff b c1 c2*,*H*):*HS*
⟨*proof*⟩

lemma *IF-alg-derivable-from-If*:
⟦(*G*,*b*,*p*):*HS-B*; (*p*,*G*,*c1*,*H*):*HS*; (*p*,*G*,*c2*,*H*):*HS*⟧
$\Longrightarrow$ (*p*,*G*,*Iff b c1 c2*,*H*):*HS*
⟨*proof*⟩

An easy induction on typing derivations shows the following property.

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

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*:
$X \rhd Skip : Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ G\ q)$
$(\lambda\ (s,t)\ .\ EQ\ G\ q\ s\ t)\langle proof\rangle$

lemma *ASSIGN*:
$[\![H = upd\ G\ x\ (LUB\ p\ t);$
$\forall\ s\ ss\ .\ EQ\ G\ t\ s\ ss \longrightarrow evalE\ e\ s = evalE\ e\ ss]\!]$
$\Longrightarrow X \rhd Assign\ x\ e : Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)$
$(\lambda\ (s,t)\ .\ \exists\ r\ .\ s = update\ r\ x\ (evalE\ e\ r) \wedge EQ\ G\ q\ r\ t)\langle proof\rangle$

lemma *COMP*:
$[\![\ X \rhd c1\ : Sec\ (Q\ p\ K)\ (EQ\ G\ q)\ (EQ\ K\ q)\ \Phi;$
$X \rhd c2\ : Sec\ (Q\ p\ H)\ (EQ\ K\ q)\ (EQ\ H\ q)\ \Psi;$
$\forall\ x\ .\ LEQ\ (G\ x)\ (K\ x) \vee LEQ\ p\ (K\ x);$
$\forall\ x\ .\ LEQ\ (K\ x)\ (H\ x) \vee LEQ\ p\ (H\ x)]\!]$
$\Longrightarrow X \rhd Comp\ c1\ c2 : Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)$
$(\lambda\ (x,\ y)\ .\ \exists\ z\ .\ \Phi\ (z,\ y) \wedge$
$(\forall\ w\ .\ EQ\ K\ q\ z\ w \longrightarrow \Psi\ (x,\ w)))\langle proof\rangle$

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*:
$[\![\ \forall\ s\ ss\ .\ EQ\ G\ p\ s\ ss \longrightarrow evalB\ b\ s = evalB\ b\ ss;$
$\forall\ x.\ LEQ\ (G\ x)\ (H\ x) \vee LEQ\ p\ (H\ x);$
$\exists\ x\ .\ LEQ\ p\ (H\ x) \wedge LEQ\ (H\ x)\ q;$
$X \rhd c1\ : Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ \Phi;$
$X \rhd c2\ : Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ \Psi]\!]$
$\Longrightarrow X \rhd Iff\ b\ c1\ c2 : Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)$
$(\lambda\ (r,\ u)\ .\ (evalB\ b\ u \longrightarrow \Phi\ (r,\ u)) \wedge$
$(\ (\neg\ evalB\ b\ u) \longrightarrow \Psi\ (r,\ u)))\langle proof\rangle$

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*:
$[\![\ (G,\ b,\ p) \in HS\text{-}B;\ (p\ ,\ G,\ c1,\ H) \in HS;\ (p,\ G,\ c2,\ H) \in HS;$
$\exists\ x\ .\ LEQ\ p\ (H\ x) \wedge LEQ\ (H\ x)\ q;$
$X \rhd c1\ : Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ \Phi;$
$X \rhd c2\ : Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ \Psi]\!]$
$\Longrightarrow X \rhd Iff\ b\ c1\ c2 : Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)$
$(\lambda\ (r,\ u)\ .\ (evalB\ b\ u \longrightarrow \Phi\ (r,\ u)) \wedge$
$(\ (\neg\ evalB\ b\ u) \longrightarrow \Psi\ (r,\ u)))\langle proof\rangle$

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*:
$[\![\forall x.\ LEQ\ (G\ x)\ (H\ x) \vee LEQ\ p\ (H\ x);$
$\neg\ (\exists x.\ LEQ\ p\ (H\ x) \wedge LEQ\ (H\ x)\ q);$
$X \rhd c1 : Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ \Phi;$
$X \rhd c2 : Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ \Psi]\!]$
$\Longrightarrow X \rhd Iff\ b\ c1\ c2 : Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)$
$(\lambda\ (s,t).\ EQ\ H\ q\ s\ t)\langle proof\rangle$

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*:
$[\![(p,\ G,\ c1,\ H) \in HS \vee (p,\ G,\ c2,\ H) \in HS;$
$\neg\ (\exists x.\ LEQ\ p\ (H\ x) \wedge LEQ\ (H\ x)\ q);$
$X \rhd c1 : Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ \Phi;$
$X \rhd c2 : Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ \Psi]\!]$
$\Longrightarrow X \rhd Iff\ b\ c1\ c2 : Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)$
$(\lambda\ (s,t).\ EQ\ H\ q\ s\ t)\langle proof\rangle$

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*:
$[\![\exists\ x\ .\ LEQ\ p\ (H\ x) \wedge LEQ\ (H\ x)\ q;$
$\neg\ (\exists x.\ LEQ\ p\ (H\ x) \wedge LEQ\ (H\ x)\ q)]\!]$
$\Longrightarrow False\langle proof\rangle$

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 $Monotone\ \varphi = (\forall\ \Phi\ \Psi\ .\ (\forall\ s\ t\ .\ \Phi(s,t) \longrightarrow \Psi(s,t)) \longrightarrow$
$(\forall\ s\ t\ .\ \varphi\ \Phi\ (s,t) \longrightarrow \varphi\ \Psi\ (s,t)))$
$\langle proof\rangle\langle proof\rangle$**lemma** *Fix-lemma*:$Monotone\ \varphi \Longrightarrow \varphi\ (FIX\ \varphi) = FIX\ \varphi\langle proof\rangle$

Next, the definition of a while-operator.

definition *PhiWhilePOp*::
$VDMAssn \Rightarrow BExpr \Rightarrow TT \Rightarrow TT \Rightarrow TT$
where $PhiWhilePOp\ A\ b\ \Phi =$
$(\lambda\ \Psi\ .\ (\lambda(r,\ u).\ (evalB\ b\ u \longrightarrow (\exists\, z.\ \Phi\ (z,\ u) \wedge$
$(\forall\, w.\ A\ z\ w \longrightarrow \Psi\ (r,\ w)))) \wedge$
$((\neg\ evalB\ b\ u) \longrightarrow A\ r\ u)))$

This operator is monotone in Φ.

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

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*: $Monotone\ (\lambda\ \Phi\ .\ PhiWhileP\ A\ b\ \Phi)\langle proof\rangle$

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

lemma *WHILE-PARALLEL*:
$[\![\ X \rhd c : Sec\ (Q\ p\ G)\ (EQ\ G\ q)\ (EQ\ G\ q)\ \Phi;$
$\forall\ \ s\ ss\ .\ EQ\ G\ p\ s\ ss \longrightarrow evalB\ b\ s = evalB\ b\ ss;\ LEQ\ p\ q]\!]$
$\Longrightarrow X \rhd While\ b\ c : Sec\ (Q\ p\ G)\ (EQ\ G\ q)\ (EQ\ G\ q)$
$(PhiWhileP\ (EQ\ G\ q)\ b\ \Phi)\langle proof\rangle$

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

lemma *WHILE-PARALLEL-tp*:
$[\![\ X \rhd c : Sec\ (Q\ p\ G)\ (EQ\ G\ q)\ (EQ\ G\ q)\ \Phi;$
$(G,\ b,\ p) \in HS\text{-}B;\ LEQ\ p\ q]\!]$
$\Longrightarrow X \rhd While\ b\ c : Sec\ (Q\ p\ G)\ (EQ\ G\ q)\ (EQ\ G\ q)$
$(PhiWhileP\ (EQ\ G\ q)\ b\ \Phi)\langle proof\rangle$

One may also give an inductive formulation of FIX:

inductive-set var::$(BExpr \times VDMAssn \times TT \times State \times State)\ set$
where
varFalse:
$[\![\neg\ evalB\ b\ t;\ A\ s\ t]\!] \Longrightarrow (b,A,\Phi,s,t){:}var$
| *varTrue*:
$[\![evalB\ b\ t;\ \Phi(r,t);\ (\forall\ \ w\ .\ A\ r\ w \longrightarrow$
$(b,A,\Phi,s,w){:}\ var)\ ]\!] \Longrightarrow (b,A,\Phi,s,t){:}var$
$\langle proof\rangle\langle proof\rangle$

The inductive formulation and the fixed point formulation are equivalent.

$\langle proof\rangle\langle proof\rangle$**lemma** *FIXvarFIX*:
$PhiWhileP\ A\ b = (\lambda\ \Phi\ .\ (\lambda\ (s,t)\ .\ (b,A,\Phi,s,t){:}var))\langle proof\rangle$

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

lemma *WHILE-PARALLEL-IND*:
$[\![$ $X \rhd c : Sec\ (Q\ p\ G)\ (EQ\ G\ q)\ (EQ\ G\ q)\ \Phi;$
$\forall\ s\ ss\ .\ EQ\ G\ p\ s\ ss \longrightarrow evalB\ b\ s = evalB\ b\ ss;\ LEQ\ p\ q]\!] \Longrightarrow$
$X \rhd While\ b\ c : (Sec\ (Q\ p\ G)\ (EQ\ G\ q)\ (EQ\ G\ q)$
$(\lambda\ (s,t)\ .\ (b,EQ\ G\ q,\Phi,s,t){:}var))\langle proof\rangle$

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

lemma *WHILE-PARALLEL-IND-tp*:
$[\![$ $X \rhd c : Sec\ (Q\ p\ G)\ (EQ\ G\ q)\ (EQ\ G\ q)\ \Phi;$
$(G,\ b,\ p) \in HS\text{-}B;\ LEQ\ p\ q\ ]\!] \Longrightarrow$
$X \rhd (While\ b\ c) :$
$(Sec\ (Q\ p\ G)\ (EQ\ G\ q)\ (EQ\ G\ q)\ (\lambda\ (s,t)\ .\ (b,EQ\ G\ q,\Phi,s,t){:}var))\langle proof\rangle\langle proof\rangle$

Of course, the inductive formulation is also monotone:

lemma *var-MonotoneInPhi*:
$Monotone\ (\lambda\ \Phi\ .\ (\lambda\ (s,t)\ .(b,A,\ \Phi,s,t){:}var))\langle proof\rangle\langle proof\rangle$

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 $varD{::}(BExpr \times VDMAssn \times State \times State)\ set$
where
$varDFalse$: $[\![\neg\ evalB\ b\ s;\ A\ s\ t]\!] \Longrightarrow (b,A,s,t){:}varD$
$|\ varDTrue$: $[\![evalB\ b\ s;\ A\ s\ w;\ (b,A,w,t){:}\ varD\ ]\!] \Longrightarrow (b,A,s,t){:}varD$

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*]:
$(b,A,s,t){:}varD \Longrightarrow transitive\ A \longrightarrow A\ s\ t\langle proof\rangle$

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

lemma *Q-transitive*:$transitive\ (Q\ q\ G)\langle proof\rangle$

and is hence respected by the inductive closure:

lemma *varDQ*:$(b,Q\ q\ G,s,t){:}varD \Longrightarrow Q\ q\ G\ s\ t\langle proof\rangle$

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

lemma *WHILE-DIAGONAL*:
$[\![X \rhd c : Sec\ (Q\ p\ G)\ (EQ\ G\ q)\ (EQ\ G\ q)\ \Phi;\ \neg\ LEQ\ p\ q]\!]$
$\Longrightarrow X \rhd While\ b\ c : Sec\ (Q\ p\ G)\ (EQ\ G\ q)\ (EQ\ G\ q)$
$(\lambda\ (s,t).\ EQ\ G\ q\ s\ t)\langle proof\rangle$

varD is monotone in the assertion position.

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

Finally, the subsumption rule.

lemma *SUB*:
$[\![\ LEQ\ p\ pp;\ \forall\, x.\ LEQ\ (G\ x)\ (GG\ x);\ \forall\, x.\ LEQ\ (HH\ x)\ (H\ x);$
$X \rhd c : Sec\ (Q\ pp\ HH)\ (EQ\ GG\ q)\ (EQ\ HH\ q)\ \Phi]\!]$
$\Longrightarrow X \rhd c : Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ \Phi$⟨*proof*⟩

5.5 Soundness results

⟨*proof*⟩

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

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

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*: $(p,G,c,H){:}HS \Longrightarrow secure\ p\ G\ c\ H$⟨*proof*⟩

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

lemma *Sec1-deriv*: $(p,G,c,H){:}HS \Longrightarrow X \rhd c : (Q\ p\ H)$⟨*proof*⟩⟨*proof*⟩
theorem *HuntSands33-1*:$(p,G,c,H){:}HS \Longrightarrow secure1\ p\ G\ c\ H$⟨*proof*⟩
lemma *Sec2-deriv*:
$(p,G,c,H){:}HS \Longrightarrow$
$(\exists\ A\ .\ X \rhd c : (Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ A))$⟨*proof*⟩⟨*proof*⟩
theorem *HuntSands33-2*: $(p,G,c,H){:}HS \Longrightarrow secure2\ q\ G\ c\ H$⟨*proof*⟩

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

lemma *CALL*:
$[\![\ (\{B\} \cup X) \rhd body : Sec\ A\ R\ S\ (\varphi(FIX\ \varphi));$
$Monotone\ \varphi;\ B = Sec\ A\ R\ S\ (FIX\ \varphi)\ ]\!]$
$\Longrightarrow X \rhd Call : B$⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

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:

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

$|\ D\text{-}ASSIGN$:
$[\![H = upd\ G\ x\ (LUB\ p\ t);$
$\forall\ s\ ss\ .\ EQ\ G\ t\ s\ ss \longrightarrow evalE\ e\ s = evalE\ e\ ss;$
$\Omega = (\lambda\ (s,\ t)\ .\ \exists\ r\ .\ s = update\ r\ x\ (evalE\ e\ r) \wedge EQ\ G\ q\ r\ t)]\!]$
$\Longrightarrow (X,\ Assign\ x\ e,\ Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ \Omega) : Deriv$

$|\ D\text{-}COMP$:
$[\![\ (X,\ c,\ Sec\ (Q\ p\ K)\ (EQ\ G\ q)\ (EQ\ K\ q)\ \Phi) : Deriv;$
$(X,\ d,\ Sec\ (Q\ p\ H)\ (EQ\ K\ q)\ (EQ\ H\ q)\ \Psi) : Deriv;$
$\forall\ x\ .\ LEQ\ (G\ x)\ (K\ x) \vee LEQ\ p\ (K\ x);$
$\forall\ x\ .\ LEQ\ (K\ x)\ (H\ x) \vee LEQ\ p\ (H\ x);$
$\Omega = (\lambda\ (x,\ y)\ .\ \exists\ z\ .\ \Phi(z,y) \wedge (\forall\ w\ .\ EQ\ K\ q\ z\ w \longrightarrow \Psi(x,w)))]\!]$
$\Longrightarrow (X,\ Comp\ c\ d,\ Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ \Omega) : Deriv$

$|\ D\text{-}IF\text{-}PARALLEL$:
$[\![\ \forall\ s\ ss\ .\ EQ\ G\ p\ s\ ss \longrightarrow evalB\ b\ s = evalB\ b\ ss;$
$\forall\ x.\ LEQ\ (G\ x)\ (H\ x) \vee LEQ\ p\ (H\ x);$
$\exists\ x\ .\ LEQ\ p\ (H\ x) \wedge LEQ\ (H\ x)\ q;$
$(X,\ c,\ Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ \Phi) : Deriv;$
$(X,\ d,\ Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ \Psi) : Deriv;$
$\Omega = (\lambda\ (r,\ u)\ .\ (evalB\ b\ u \longrightarrow \Phi(r,u)) \wedge$
$(\ (\neg\ evalB\ b\ u) \longrightarrow \Psi(r,u)))]\!]$
$\Longrightarrow (X,\ Iff\ b\ c\ d,\ Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ \Omega) : Deriv$

$|\ D\text{-}IF\text{-}DIAGONAL$:
$[\![\ \forall x.\ LEQ\ (G\ x)\ (H\ x) \vee LEQ\ p\ (H\ x);$
$\neg\ (\exists x.\ LEQ\ p\ (H\ x) \wedge LEQ\ (H\ x)\ q);$
$(X,\ c,\ Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ \Phi) : Deriv;$
$(X,\ d,\ Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ \Psi) : Deriv;$
$\Omega = (\lambda\ (s,t)\ .\ EQ\ H\ q\ s\ t)]\!]$
$\Longrightarrow (X,\ Iff\ b\ c\ d,\ Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ \Omega) : Deriv$

$|\ D\text{-}WHILE\text{-}PARALLEL$:
$[\![\ (X,\ c,\ Sec\ (Q\ p\ G)\ (EQ\ G\ q)\ (EQ\ G\ q)\ \Phi){:}Deriv;$
$\forall\ s\ ss\ .\ EQ\ G\ p\ s\ ss \longrightarrow evalB\ b\ s = evalB\ b\ ss;\ LEQ\ p\ q;$
$\Omega = (\lambda\ (s,t)\ .\ (b,EQ\ G\ q,\Phi,s,t){:}var)]\!]$
$\Longrightarrow (X,\ While\ b\ c,\ Sec\ (Q\ p\ G)\ (EQ\ G\ q)\ (EQ\ G\ q)\ \Omega){:}Deriv$

$|\ D\text{-}WHILE\text{-}DIAGONAL$:
$[\![(X,\ c,\ Sec\ (Q\ p\ G)\ (EQ\ G\ q)\ (EQ\ G\ q)\ \Phi) : Deriv;\ \neg\ LEQ\ p\ q;$
$\Omega = (\lambda\ (s,t)\ .\ EQ\ G\ q\ s\ t)]\!]$
$\Longrightarrow (X,\ While\ b\ c,\ Sec\ (Q\ p\ G)\ (EQ\ G\ q)\ (EQ\ G\ q)\ \Omega) : Deriv$

$|\ D\text{-}SUB$:
$[\![\ LEQ\ p\ pp;\ \forall x.\ LEQ\ (G\ x)\ (GG\ x);\ \forall x.\ LEQ\ (HH\ x)\ (H\ x);$
$(X,\ c,\ Sec\ (Q\ pp\ HH)\ (EQ\ GG\ q)\ (EQ\ HH\ q)\ \Phi) : Deriv]\!]$

$\Longrightarrow (X,\ c,\ Sec\ (Q\ p\ H)\ (EQ\ G\ q)\ (EQ\ H\ q)\ \Phi)\ :Deriv$

| *D-CALL*:
$(\{A\} \cup X, body, A)$: *Deriv* $\Longrightarrow (X, Call, A)$: *Deriv*
⟨*proof*⟩
lemma *DerivMono*:
(X,c,B):*Deriv* $\Longrightarrow$
$\exists\ A\ R\ S\ \varphi\ .\ B = \ Sec\ A\ R\ S\ (\varphi\ (FIX\ \varphi)) \wedge Monotone\ \varphi$⟨*proof*⟩

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

theorem *Deriv-derivable*: (X,c,A):*Deriv* $\Longrightarrow X \rhd c$:A⟨*proof*⟩

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* $L = \{l\ .\ \exists\ a\ .\ lookup\ L\ l = Some\ a\}$
⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

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* × ((*Field* × *Val*) *list*)
type-synonym *Heap* = (*Location* × *Object*) *list*

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

type-synonym *Store* = *Var* ⇒ *Val*

definition *update* :: *Store* ⇒ *Var* ⇒ *Val* ⇒ *Store*
where *update s x v* = (λ *y* . *if x*=*y then v else s y*)

type-synonym *State* = *Store* × *Heap*

Arithmetic and boolean expressions are as before.

datatype *Expr* =
varE Var
| *valE Val*
| *opE Val* ⇒ *Val* ⇒ *Val Expr Expr*

datatype *BExpr* = *compB Val* ⇒ *Val* ⇒ *bool Expr Expr*

The same applies to their semantics.

primrec *evalE*::*Expr* ⇒ *Store* ⇒ *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* ⇒ *Store* ⇒ *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 *Semn* :: (*State* × *OBJ* × *nat* × *State*) *set*
where
SemSkip: *s*=*t* ⟹ (*s*,*Skip*,*1*, *t*):*Semn*

| *SemAssign*:
⟦ *t* = (*update* (*fst s*) *x* (*evalE e* (*fst s*)), *snd s*)⟧
⟹ (*s*, *Assign x e*, *1*, *t*):*Semn*

| *SemNew*:
⟦*l* ∉ *Dom* (*snd s*);
t = (*update* (*fst s*) *x* (*RVal* (*Loc l*)), (*l*,(*C*,[])) # (*snd s*))⟧
⟹ (*s*, *New x C*, *1*, *t*):*Semn*

| *SemGet*:
⟦(*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*)⟧
⟹ (*s*, *Get x y F*, *1*, *t*):*Semn*

| *SemPut*:
⟦(*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*))⟧
⟹ (*s*, *Put x F e*, *1*, *t*):*Semn*

| *SemComp*:
⟦ (*s*, *c*, *n*, *r*):*Semn*; (*r*,*d*, *m*, *t*):*Semn*; *k*=(*max n m*)+*1*⟧
⟹ (*s*, *Comp c d*, *k*, *t*):*Semn*

| *SemWhileT*:
⟦*evalB b* (*fst s*); (*s*,*c*, *n*, *r*):*Semn*; (*r*, *While b c*, *m*, *t*):*Semn*; *k*=((*max n m*)+*1*)⟧
⟹ (*s*, *While b c*, *k*, *t*):*Semn*

| *SemWhileF*:
⟦¬ (*evalB b* (*fst s*)); *t*=*s*⟧ ⟹ (*s*, *While b c*, *1*, *t*):*Semn*

| *SemTrue*:
⟦*evalB b* (*fst s*); (*s*,*c1*, *n*, *t*):*Semn*⟧
⟹ (*s*, *Iff b c1 c2*, *n*+*1*, *t*):*Semn*

| *SemFalse*:
⟦¬ (*evalB b* (*fst s*)); (*s*,*c2*, *n*, *t*):*Semn*⟧
⟹ (*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*, *OBJ*, *nat*, *State*] ⇒ *bool* (‹ - , - →- - ›)
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*, *OBJ*, *State*] ⇒ *bool* (‹- , - ⇓ - › *1000*)
where $s,c \Downarrow t = (\exists\ n.\ s,c \rightarrow_n t)$

inductive-cases *Sem-eval-cases*:
$s,Skip \rightarrow_n t$
$s,(Assign\ x\ e) \rightarrow_n t$
$s,(New\ x\ C) \rightarrow_n t$
$s,(Get\ x\ y\ F) \rightarrow_n t$
$s,(Put\ x\ F\ e) \rightarrow_n t$
$s,(Comp\ c1\ c2) \rightarrow_n t$
$s,(While\ b\ c) \rightarrow_n t$
$s,(Iff\ b\ c1\ c2) \rightarrow_n t$
$s,\ Call \rightarrow_n t$
⟨*proof*⟩**lemma** *Sem-no-zero-height-derivs*: $(s,\ c \rightarrow_0 t) \Longrightarrow False$⟨*proof*⟩

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* ⇒ *State* ⇒ *bool*

definition *VDM-validn* :: *nat* ⇒ *OBJ* ⇒ *Assn* ⇒ *bool*
(‹ ⊨- - : - › *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* ⇒ *Assn* ⇒ *bool*
(‹ ⊨ - : - › *50*)

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

lemma *VDM-valid-validn*: $\models c{:}A \Longrightarrow \models_n c{:}A$⟨*proof*⟩
lemma *VDM-validn-valid*: $(\forall\ n\ .\ \models_n c{:}A) \Longrightarrow \models c{:}A$⟨*proof*⟩
lemma *VDM-lowerm*: $[\![\ \models_n c{:}A;\ m < n\ ]\!] \Longrightarrow \models_m c{:}A$⟨*proof*⟩
definition *Ctxt-validn* :: *nat* ⇒ (*Assn set*) ⇒ *bool*
(‹ ⊨_ - › *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* ⇒ *bool* (‹ ⊨ - › *50*)
where $(\models G) = (\forall\ A\ .\ A \in G \longrightarrow \models Call : A)$

lemma *Ctxt-valid-validn*: $\models G \Longrightarrow \models_n G$⟨*proof*⟩
lemma *Ctxt-validn-valid*: $(\forall\ n\ .\ \models_n G) \Longrightarrow \models G$⟨*proof*⟩
lemma *Ctxt-lowerm*: $[\![\ \models_n G;\ m < n\ ]\!] \Longrightarrow \models_m G$⟨*proof*⟩
definition *valid* :: (*Assn set*) ⇒ *OBJ* ⇒ *Assn* ⇒ *bool*
(‹- ⊨ - : - › *50*)
where $(G \models c : A) = (Ctxt\text{-}valid\ G \longrightarrow VDM\text{-}valid\ c\ A)$

definition *validn* :: (*Assn set*) ⇒ *nat* ⇒ *OBJ* ⇒ *Assn* ⇒ *bool*
(‹- ⊨_ - : -› *50*)
where $(G \models_n c : A) = (\models_n G \longrightarrow \models_n c : A)$

lemma *validn-valid*: $(\forall\ n\ .\ G \models_n c : A) \Longrightarrow G \models c : A$⟨*proof*⟩
lemma *ctxt-consn*: $[\![\ \models_n G;\ \models_n Call{:}A\ ]\!] \Longrightarrow \models_n \{A\} \cup G$⟨*proof*⟩

6.2.2 Proof system

inductive-set *VDM-proof* :: (*Assn set* × *OBJ* × *Assn*) *set*
where
VDMSkip: $(G,\ Skip,\ \lambda\ s\ t\ .\ t{=}s){:}VDM\text{-}proof$

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

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

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

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

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

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

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

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

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

| *VDMConseq*:
$[\![\ (G,\ c,A):VDM\text{-}proof;\ \forall\ s\ t.\ A\ s\ t \longrightarrow B\ s\ t]\!] \Longrightarrow$
$(G,\ c,\ B):VDM\text{-}proof$

abbreviation $VDM\text{-}deriv :: [Assn\ set,\ OBJ,\ Assn] \Rightarrow bool$
$(\langle\text{-} \rhd \text{-} : \text{-}\rangle\ [100,100]\ 50)$
where $G \rhd c : A == (G,c,A) \in VDM\text{-}proof$

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

lemma *Hoare-While*:
$G \rhd c : (\lambda\ s\ t\ .\ \forall\ r\ .\ evalB\ b\ (fst\ s) \longrightarrow I\ s\ r \longrightarrow I\ t\ r) \Longrightarrow$
$G \rhd While\ b\ c : (\lambda\ s\ t.\ \forall\ r\ .\ I\ s\ r \longrightarrow (I\ t\ r \wedge \neg\ evalB\ b\ (fst\ t)))$
$\langle proof \rangle$

Here's the proof in the opposite direction.

lemma *VDMWhile-derivable*:
$[\![\ G \rhd c : B;\ \forall\ s\ .\ (\neg\ evalB\ b\ (fst\ s)) \longrightarrow A\ s\ s;$
$\forall\ s\ r\ t.\ evalB\ b\ (fst\ s) \longrightarrow B\ s\ r \longrightarrow A\ r\ t \longrightarrow A\ s\ t\ ]\!]$
$\Longrightarrow G \rhd (While\ b\ c) : (\lambda\ s\ t\ .\ A\ s\ t \wedge \neg\ (evalB\ b\ (fst\ t)))$
$\langle proof \rangle$

6.2.3 Soundness

⟨*proof*⟩

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

lemma *SoundWhile*[*rule-format*]:
$(\forall\, n.\ G \models_n c : B)$
$\longrightarrow (\forall\, s.\ (\neg\ evalB\ b\ (fst\ s)) \longrightarrow A\ s\ s)$
$\longrightarrow (\forall\, s.\ evalB\ b\ (fst\ s)$
$\longrightarrow (\forall\, r.\ B\ s\ r \longrightarrow (\forall\, t.\ A\ r\ t \longrightarrow A\ s\ t)))$
$\longrightarrow G \models_n (While\ b\ c) : (\lambda s\ t.\ A\ s\ t \wedge \neg\ evalB\ b\ (fst\ t))$⟨*proof*⟩
lemma *SoundCall*[*rule-format*]:
$[\![\forall\, n.\ \models_n (\{A\} \cup G) \longrightarrow \models_n body : A]\!] \Longrightarrow \models_n G \longrightarrow \models_n Call : A$⟨*proof*⟩
lemma *VDM-Sound-n*: $G \rhd c\!: A \Longrightarrow (\forall\ n.\ G \models_n c\!:\!A)$⟨*proof*⟩
theorem *VDM-Sound*: $G \rhd c\!: A \Longrightarrow G \models c\!:\!A$⟨*proof*⟩

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

lemma *VDM-Sound-emptyCtxt*:$\{\} \rhd c : A \Longrightarrow\ \models c : A$⟨*proof*⟩

6.2.4 Derived rules

lemma *WEAK*[*rule-format*]:
$G \rhd c : A \Longrightarrow (\forall\ H\ .\ G \subseteq H \longrightarrow H \rhd c\ \!:\!A)$⟨*proof*⟩
lemma *CutAux*:
$(H \rhd c : A) \Longrightarrow$
$(\forall\ G\ P\ D\ .\ (H = (insert\ P\ D) \longrightarrow G \rhd Call\ \!:\!P \longrightarrow (G \subseteq D)$
$\longrightarrow D \rhd c\!:\!A))$⟨*proof*⟩
lemma *Cut*:$[\![\ G \rhd Call : P\ ;\ (insert\ P\ G) \rhd c : A\ ]\!] \Longrightarrow G \rhd c : A$⟨*proof*⟩
definition *verified*::*Assn set* ⇒ *bool*
where $verified\ G = (\forall\ A\ .\ A\!:\!G \longrightarrow G \rhd body : A)$

lemma *verified-preserved*: $[\![verified\ G;\ A\!:\!G]\!] \Longrightarrow verified\ (G - \{A\})$⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩
theorem *Mutrec*:
$[\![\ finite\ G;\ card\ G = n;\ verified\ G;\ A : G\ ]\!] \Longrightarrow \{\} \rhd Call\!:\!A$⟨*proof*⟩

6.2.5 Completeness

definition *SSpec*::*OBJ* ⇒ *Assn*
where $SSpec\ c\ s\ t = (s,c \Downarrow t)$

lemma *SSpec-valid*: $\models c : (SSpec\ c)$⟨*proof*⟩
lemma *SSpec-strong*: $\models c\ \!:\!A \Longrightarrow \forall\ s\ t\ .\ SSpec\ c\ s\ t \longrightarrow A\ s\ t$⟨*proof*⟩
lemma *SSpec-derivable*:$G \rhd Call : SSpec\ Call \Longrightarrow G \rhd c : SSpec\ c$⟨*proof*⟩
definition *StrongG* :: *Assn set*
where $StrongG = \{SSpec\ Call\}$

lemma *StrongG-Body*: $StrongG \rhd body : SSpec\ Call$⟨*proof*⟩
lemma *StrongG-verified*: *verified StrongG*⟨*proof*⟩
lemma *SSpec-derivable-empty*:$\{\} \rhd c : SSpec\ c$⟨*proof*⟩
theorem *VDM-Complete*: $\models c : A \Longrightarrow \{\} \rhd c : A$⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

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* = (*Location* × *Location*) *set*
definition *Pbij* :: *PBij set*
where *Pbij* = { β . ∀ *l1 l2 l3 l4*. (*l1*,*l2*):β ⟶ (*l3*,*l4*):β ⟶
((*l1* = *l3*) = (*l2* = *l4*))}

Domain and codomain are defined as expected.

definition *Pbij-Dom*::*PBij* ⇒ (*Location set*)
where *Pbij-Dom* β = {*l* . ∃ *ll* .(*l*,*ll*):β}

definition *Pbij-Rng*::*PBij* ⇒ (*Location set*)
where *Pbij-Rng* β = {*ll* . ∃ *l* .(*l*,*ll*):β}

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

definition *Pbij-inverse*::*PBij* ⇒ *PBij*
where *Pbij-inverse* β = {(*l*,*ll*) . (*ll*,*l*):β}⟨*proof*⟩⟨*proof*⟩
definition *Pbij-compose*::*PBij* ⇒ *PBij* ⇒ *PBij*
where *Pbij-compose* β γ = {(*l*,*ll*) . ∃ *l1* . (*l*,*l1*):β ∧ (*l1*,*ll*):γ}⟨*proof*⟩⟨*proof*⟩
definition *Pbij-extends* ::*PBij* ⇒ *PBij* ⇒ *bool*
where *Pbij-extends* γ β = (β ⊆ γ)

These definitions satisfy the following properties.

lemma *Pbij-insert*:
⟦β ∈ *Pbij*;*l* ∉ *Pbij-Rng* β; *ll* ∉ *Pbij-Dom* β⟧
⟹ *insert* (*ll*, *l*) β ∈ *Pbij*⟨*proof*⟩
lemma *Pbij-injective*:
β:*Pbij* ⟹ (∀ *l l1 l2* . (*l1*,*l*):β ⟶ (*l2*,*l*):β ⟶ *l1*=*l2*)⟨*proof*⟩
lemma *Pbij-inverse-DomRng*[*rule-format*]:
γ = *Pbij-inverse* β ⟹
(*Pbij-Dom* β = *Pbij-Rng* γ ∧ *Pbij-Dom* γ = *Pbij-Rng* β)⟨*proof*⟩
lemma *Pbij-inverse-Dom*: *Pbij-Dom* β = *Pbij-Rng* (*Pbij-inverse* β)⟨*proof*⟩
lemma *Pbij-inverse-Rng*: *Pbij-Dom* (*Pbij-inverse* β) = *Pbij-Rng* β⟨*proof*⟩
lemma *Pbij-inverse-Pbij*: β:*Pbij* ⟹ (*Pbij-inverse* β) : *Pbij*⟨*proof*⟩
lemma *Pbij-inverse-Inverse*[*rule-format*]:
γ = *Pbij-inverse* β ⟹ (∀ *l ll* . ((*l*,*ll*):β) = ((*ll*,*l*):γ))⟨*proof*⟩
lemma *Pbij-compose-Dom*:
Pbij-Dom (*Pbij-compose* β γ) ⊆ *Pbij-Dom* β⟨*proof*⟩
lemma *Pbij-compose-Rng*:
Pbij-Rng (*Pbij-compose* β γ) ⊆ *Pbij-Rng* γ⟨*proof*⟩
lemma *Pbij-compose-Pbij*:

$[\![\beta : Pbij; \gamma : Pbij]\!] \Longrightarrow Pbij\text{-}compose\ \beta\ \gamma : Pbij\langle proof\rangle$
lemma *Pbij-extends-inverse*:
$Pbij\text{-}extends\ \gamma\ (Pbij\text{-}inverse\ \beta) = Pbij\text{-}extends\ (Pbij\text{-}inverse\ \gamma)\ \beta\langle proof\rangle$
lemma *Pbij-extends-reflexive*:$Pbij\text{-}extends\ \beta\ \beta\langle proof\rangle$
lemma *Pbij-extends-transitive*:
$[\![Pbij\text{-}extends\ \beta\ \gamma; Pbij\text{-}extends\ \gamma\ \delta]\!] \Longrightarrow Pbij\text{-}extends\ \beta\ \delta\langle proof\rangle\langle proof\rangle$
lemma *Pbij-inverse-extends-twice*:
$Pbij\text{-}extends\ (Pbij\text{-}inverse\ (Pbij\text{-}inverse\ \beta))\ \beta\langle proof\rangle$

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

definition $mkId{::}Heap \Rightarrow (Location \times Location)\ set$
where $mkId\ h = \{(l1,l2)\ .\ l1 = l2 \wedge l1 : Dom\ h\}$

lemma *mkId1*: $(mkId\ h){:}Pbij\langle proof\rangle$
lemma *mkId2*: $Pbij\text{-}Dom\ (mkId\ h) = Dom\ h\langle proof\rangle$
lemma *mkId2b*: $Pbij\text{-}Rng\ (mkId\ h) = Dom\ h\langle proof\rangle$
lemma *mkId4*: $l{:}Dom\ h \Longrightarrow (l,l){:}(mkId\ h)\langle proof\rangle$
lemma *mkId4b*: $(l,ll){:}(mkId\ h) \Longrightarrow l{:}Dom\ h \wedge l = ll\langle proof\rangle$

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 $twiddleVal{::}(PBij \times Val \times Val)\ set$
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* ⇒ *Store* ⇒ *Store* ⇒ *bool*
where *twiddleStore β s1 s2* =
(∀ *x*. *CONTEXT x* = *low* ⟶ (*β*, *s1 x*, *s2 x*) : *twiddleVal*)

abbreviation *twiddleStore-syntax* (‹- ≈₋ -› [*100*,*100*] *50*)
where *s* $\approx_\beta$ *t* == *twiddleStore β 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* × *Val*) *list*) ⇒ *Field set*
where *LowDom F* = {*f* . ∃ *v* . *lookup F f* = *Some v* ∧ *GAMMA f* = *low*}

definition *twiddleObj*::*PBij* ⇒ *Object* ⇒ *Object* ⇒ *bool*
where *twiddleObj β o1 o2* = ((*fst o1* = *fst o2*) ∧
LowDom (*snd o1*) = *LowDom* (*snd o2*) ∧
(∀ *f v w* . *lookup* (*snd o1*) *f* = *Some v* ⟶
lookup (*snd o2*) *f* = *Some w* ⟶
GAMMA f = *low* ⟶
(*β*, *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* ⇒ *Heap* ⇒ *Heap* ⇒ *bool*
where *twiddleHeap β h1 h2* = (*β*:*Pbij* ∧
Pbij-Dom β ⊆ *Dom h1* ∧
Pbij-Rng β ⊆ *Dom h2* ∧
(∀ *l ll v w* . (*l*,*ll*):*β* ⟶
lookup h1 l = *Some v* ⟶
lookup h2 ll = *Some w* ⟶
twiddleObj β v w))

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

definition *noLowDPs*::*State* ⇒ *bool*
where *noLowDPs S* = (*case S of* (*s*,*h*) ⇒
((∀ *x l* . *CONTEXT x* = *low* ⟶ *s x* = *RVal*(*Loc l*) ⟶ *l*:*Dom h*) ∧
(∀ *ll c F f l* . *lookup h ll* = *Some*(*c*,*F*) ⟶ *GAMMA f* = *low* ⟶
lookup F f = *Some*(*RVal*(*Loc l*)) ⟶ *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* ⇒ *State* ⇒ *State* ⇒ *bool*
where *twiddle* β *s t* = (*noLowDPs s* ∧ *noLowDPs t* ∧
(*fst s*) $\approx_\beta$ (*fst t*) ∧ *twiddleHeap* β (*snd s*) (*snd t*))

abbreviation *twiddle-syntax* (‹- ≡- -› [*100*,*100*] *50*)
where *s* $\equiv_\beta$ *t* == *twiddle* β *s t*

The following properties are easily proven by unfolding the definitions.

lemma *twiddleHeap-isPbij*:*twiddleHeap* β *h hh* ⟹ β:*Pbij*⟨*proof*⟩
lemma *isPBij*:*s* $\equiv_\beta$ *t* ⟹ β:*Pbij*⟨*proof*⟩
lemma *twiddleVal-inverse*:
(β, *w*, *v*) ∈ *twiddleVal* ⟹ (*Pbij-inverse* β, *v*, *w*) ∈ *twiddleVal*⟨*proof*⟩
lemma *twiddleStore-inverse*: *s* $\approx_\beta$ *t* ⟹ *t* $\approx_($*Pbij-inverse* β) *s*⟨*proof*⟩
lemma *twiddleHeap-inverse*:
twiddleHeap β *s t* ⟹ *twiddleHeap* (*Pbij-inverse* β) *t s*⟨*proof*⟩
lemma *Pbij-inverse-twiddle*: ⟦*s* $\equiv_\beta$ *t*⟧ ⟹ *t* $\equiv_($*Pbij-inverse* β) *s*⟨*proof*⟩
lemma *twiddleVal-betaExtend*[*rule-format*]:
(β,*v*,*w*):*twiddleVal* ⟹ ∀ γ. *Pbij-extends* γ β ⟶ (γ,*v*,*w*):*twiddleVal*⟨*proof*⟩
lemma *twiddleObj-betaExtend*[*rule-format*]:
⟦*twiddleObj* β *o1 o2*; *Pbij-extends* γ β⟧ ⟹ *twiddleObj* γ *o1 o2*⟨*proof*⟩
lemma *twiddleVal-compose*:
⟦(β, *v*, *u*) ∈ *twiddleVal*; (γ, *u*, *w*) ∈ *twiddleVal*⟧
⟹ (*Pbij-compose* β γ, *v*, *w*) ∈ *twiddleVal*⟨*proof*⟩
lemma *twiddleHeap-compose*:
⟦ *twiddleHeap* β *h1 h2*; *twiddleHeap* γ *h2 h3*; β ∈ *Pbij*; γ ∈ *Pbij*⟧
⟹ *twiddleHeap* (*Pbij-compose* β γ) *h1 h3*⟨*proof*⟩
lemma *twiddleStore-compose*:
⟦*s* $\approx_\beta$ *r*; *r*$\approx_\gamma$ *t*⟧ ⟹ *s* $\approx_($*Pbij-compose* β γ) *t*⟨*proof*⟩
lemma *twiddle-compose*:
⟦*s* $\equiv_\beta$ *r*; *r* $\equiv_\gamma$ *t*⟧ ⟹ *s* $\equiv_($*Pbij-compose* β γ) *t*⟨*proof*⟩
lemma *twiddle-mkId*: *noLowDPs* (*s*,*h*) ⟹ (*s*,*h*) $\equiv_($*mkId h*) (*s*,*h*)⟨*proof*⟩

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

definition *Expr-low*::*Expr* ⇒ *bool*
where *Expr-low e* = (∀ *s t* β. *s* $\approx_\beta$ *t* ⟶ (β, *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* ⇒ *bool*
where *BExpr-low b* = (∀ *s t* β . *s* $\approx_\beta$ *t* ⟶ *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 $secure\ c = (\forall\ s\ ss\ t\ tt\ \beta\ .$
$s \equiv_\beta ss \longrightarrow (s,c \Downarrow t) \longrightarrow (ss,c \Downarrow tt) \longrightarrow$
$(\exists\ \gamma\ .\ t \equiv_\gamma tt \wedge Pbij\text{-}extends\ \gamma\ \beta))$
$\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle$

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 $Prop1A{:}\models c : Sec\ \Phi \Longrightarrow secure\ c\langle proof\rangle$
lemma $Prop1B$:
$secure\ c \Longrightarrow\ \models c : Sec\ (\lambda\ (r,\ t,\ \beta).\ \exists\ s.\ s\ ,\ c \Downarrow r \wedge s \equiv_\beta t)\langle proof\rangle$
lemma $Prop1BB{:}secure\ c \Longrightarrow \exists\ \Phi\ .\models c : Sec\ \Phi\langle proof\rangle$
lemma $Prop1$:
$secure\ c = (\models c : Sec\ (\lambda\ (r,\ t,\beta)\ .\ \exists\ s\ .\ (s\ ,\ c \Downarrow r) \wedge s \equiv_\beta t))\langle proof\rangle$

6.5 Derivation of proof rules

6.5.1 Low proof rules

$\langle proof\rangle$
lemma $SKIP$: $G \rhd Skip : Sec\ (\lambda\ (s,\ t,\ \beta)\ .\ s \equiv_\beta t)\langle proof\rangle$
lemma $ASSIGN$:
$Expr\text{-}low\ e$
$\Longrightarrow G \rhd Assign\ x\ e : Sec\ (\lambda\ (s,\ t,\ \beta)\ .$
$\exists\ r\ .\ s = (update\ (fst\ r)\ x\ (evalE\ e\ (fst\ r)),\ snd\ r)$
$\wedge\ r \equiv_\beta t)\langle proof\rangle\langle proof\rangle$
lemma $COMP$:
$[\![\ G \rhd c1 : Sec\ \Phi;\ G \rhd c2 : Sec\ \Psi]\!]$
$\Longrightarrow G \rhd (Comp\ c1\ c2) : Sec\ (\lambda\ (s,\ t,\ \beta)\ .$
$\exists\ r\ .\ \Phi(r,\ t,\ \beta) \wedge (\forall\ w\ \gamma.\ r \equiv_\gamma w \longrightarrow \Psi(s,\ w,\ \gamma)))\langle proof\rangle\langle proof\rangle$
lemma IFF:
$[\![\ BExpr\text{-}low\ b;\ G \rhd c1 : Sec\ \Phi;\ G \rhd c2 : Sec\ \Psi]\!]$
$\Longrightarrow G \rhd (Iff\ b\ c1\ c2) : Sec\ (\lambda\ (s,t,\beta)\ .$
$(evalB\ b\ (fst\ t) \longrightarrow \Phi(s,t,\beta)) \wedge$
$((\neg\ evalB\ b\ (fst\ t)) \longrightarrow \Psi(s,t,\beta)))\langle proof\rangle\langle proof\rangle$
lemma NEW:
$CONTEXT\ x = low$
$\Longrightarrow G \rhd (New\ x\ C) : Sec\ (\lambda\ (s,t,\beta)\ .$

$\exists\ l\ r\ .\ l \notin Dom\ (snd\ r) \land r \equiv_\beta t \land$
$s = (update\ (fst\ r)\ x\ (RVal\ (Loc\ l)),$
$(l,(C,[]))\ \#\ (snd\ r)))\langle proof\rangle$

lemma *GET*:
$[\![\ CONTEXT\ y = low;\ GAMMA\ f = low]\!]$
$\Longrightarrow G \rhd Get\ x\ y\ f : Sec\ (\lambda\ (s,t,\beta)\ .$
$\exists\ r\ l\ C\ Flds\ v.\ (fst\ r)\ y = RVal(Loc\ l) \land$
$lookup\ (snd\ r)\ l = Some(C,Flds) \land$
$lookup\ Flds\ f = Some\ v \land r \equiv_\beta t \land$
$s = (update\ (fst\ r)\ x\ v,\ snd\ r))\langle proof\rangle$

lemma *PUT*:
$[\![\ CONTEXT\ x = low;\ GAMMA\ f = low;\ Expr\text{-}low\ e]\!]$
$\Longrightarrow G \rhd Put\ x\ f\ e: Sec\ (\lambda\ (s,t,\beta)\ .$
$\exists\ r\ l\ C\ Flds.\ (fst\ r)\ x = RVal(Loc\ l) \land r \equiv_\beta t \land$
$lookup\ (snd\ r)\ l = Some(C,Flds) \land$
$s = (fst\ r,$
$(l,(C,(f,evalE\ e\ (fst\ r))\ \#\ Flds))\ \#\ (snd\ r)))\langle proof\rangle$

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 $Monotone\ \varphi =$
$(\forall\ \Phi\ \Psi\ .\ (\forall\ s\ t\ \beta.\ \Phi(s,t,\beta) \longrightarrow \Psi(s,t,\beta)) \longrightarrow$
$(\forall\ s\ t\ \beta.\ \varphi\ \Phi\ (s,t,\beta) \longrightarrow \varphi\ \Psi\ (s,t,\beta)))$
$\langle proof\rangle\langle proof\rangle$

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

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

The operator used in the while rule is defined by

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

and is monotone:

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

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*:
$[\![\ BExpr\text{-}low\ b;\ G \rhd c : (Sec\ \Phi)]\!]$

$\Longrightarrow G \rhd (While\ b\ c) : (Sec\ (PhiWhile\ b\ \Phi))\langle proof\rangle$

Operator *PhiWhile* is itself monotone in Φ:

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

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

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

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

The equivalence of the invariant with the fixed point construction.

$\langle proof\rangle\langle proof\rangle$
lemma *varFIXvar*: $(PhiWhile\ b\ \Phi\ (s,t,\beta)) = ((b,\Phi,\beta,s,t){:}var)\langle proof\rangle\langle proof\rangle\langle proof\rangle$

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

lemma *WHILE-IND*:
$[\![\ BExpr\text{-}low\ b;\ G \rhd c : (Sec\ \Phi)]\!]$
$\Longrightarrow G \rhd While\ b\ c{:}\ (Sec\ (\lambda(s,t,\gamma).\ (b,\Phi,\gamma,s,t){:}var))\langle proof\rangle$

We can also show the following property.

$\langle proof\rangle$
lemma *var-Monotone*:
$Monotone\ (\lambda\ \Phi\ .\ (\lambda\ (s,t,\beta)\ .(b,\Phi,\beta,s,t){:}var))\langle proof\rangle\langle proof\rangle$

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

lemma *CALL*:
$[\![(\{Sec\ (FIX\ \varphi)\} \cup X) \rhd body : Sec\ (\varphi\ (FIX\ \varphi));\ Monotone\ \varphi]\!]$
$\Longrightarrow X \rhd Call : Sec\ (FIX\ \varphi)\langle proof\rangle$

6.5.2 High proof rules

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

lemma $CAST[rule\text{-}format]$:
$G \rhd c : HighSec \Longrightarrow G \rhd c : Sec\ (\lambda\ (s,\ t,\ \beta)\ .\ s \equiv_\beta t)\langle proof\rangle$
lemma *SkipHigh*: $G \rhd Skip{:}\ HighSec\langle proof\rangle$

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

definition $Expr\text{-}good{::}Expr \Rightarrow State \Rightarrow bool$

where *Expr-good e s* =
($\forall$ *l* . *evalE e* (*fst s*) = *RVal*(*Loc l*) $\longrightarrow$ *l* : *Dom* (*snd s*))

lemma *AssignHigh*:
$[\![$ *CONTEXT x* = *high*; $\forall$ *s* . *noLowDPs s* $\longrightarrow$ *Expr-good e s*$]\!]$
$\Longrightarrow$ *G* $\rhd$ *Assign x e*: *HighSec*$\langle$*proof*$\rangle$
lemma *NewHigh*:
CONTEXT x = *high* $\Longrightarrow$ *G* $\rhd$ *New x C* : *HighSec*$\langle$*proof*$\rangle$
lemma *GetHigh*:
$[\![$ *CONTEXT x* = *high* $]\!]$ $\Longrightarrow$ *G* $\rhd$ *Get x y f* : *HighSec*$\langle$*proof*$\rangle$
lemma *PutHigh*:
$[\![$ *GAMMA f* = *high*; $\forall$ *s* . *noLowDPs s* $\longrightarrow$ *Expr-good e s*$]\!]$
$\Longrightarrow$ *G* $\rhd$ *Put x f e*: *HighSec*$\langle$*proof*$\rangle\langle$*proof*$\rangle\langle$*proof*$\rangle\langle$*proof*$\rangle$
lemma *CompHigh*:
$[\![$ *G* $\rhd$ *c*: *HighSec*; *G* $\rhd$ *d*:*HighSec*$]\!]$ $\Longrightarrow$ *G* $\rhd$ *Comp c d*: *HighSec*$\langle$*proof*$\rangle$
lemma *IfHigh*:
$[\![$ *G* $\rhd$ *c*: *HighSec*; *G* $\rhd$ *d*:*HighSec*$]\!]$ $\Longrightarrow$ *G* $\rhd$ *Iff b c d*: *HighSec*$\langle$*proof*$\rangle$
lemma *WhileHigh*: $[\![$ *G* $\rhd$ *c*: *HighSec*$]\!]$ $\Longrightarrow$ *G* $\rhd$ *While b c*: *HighSec*$\langle$*proof*$\rangle$
lemma *CallHigh*:
({*HighSec*} $\cup$ *G*) $\rhd$ *body* : *HighSec* $\Longrightarrow$ *G* $\rhd$ *Call* : *HighSec*$\langle$*proof*$\rangle$

We combine all rules to an inductive derivation system.

inductive-set *Deriv*::(*Assn set* $\times$ *OBJ* $\times$ *Assn*) *set*
where
D-CAST:
(*G*, *c*, *HighSec*):*Deriv* $\Longrightarrow$
(*G*, *c*, *Sec* (λ (*s*, *t*, β). $s \equiv_\beta t$)):*Deriv*

| *D-SKIP*: (*G*, *Skip*, *Sec* (λ (*s*, *t*, β) . $s \equiv_\beta t$)) : *Deriv*

| *D-ASSIGN*:
Expr-low e $\Longrightarrow$
(*G*, *Assign x e*, *Sec* (λ (*s*, *t*, β) .
$\exists$ *r* . *s* = (*update* (*fst r*) *x* (*evalE e* (*fst r*)), *snd r*)
$\wedge$ $r \equiv_\beta t$)):*Deriv*

| *D-COMP*:
$[\![$ (*G*, *c1*, *Sec* Φ):*Deriv*; (*G*, *c2*, *Sec* Ψ):*Deriv*$]\!]$ $\Longrightarrow$
(*G*, *Comp c1 c2*, *Sec* (λ (*s*, *t*, β) .
$\exists$ *r* . Φ(*r*, *t*, β) $\wedge$
($\forall$ *w* γ. $r \equiv_\gamma w$ $\longrightarrow$ Ψ(*s*, *w*, γ)))):*Deriv*

| *D-IFF*:
$[\![$ *BExpr-low b*; (*G*, *c1*, *Sec* Φ):*Deriv*; (*G*, *c2*, *Sec* Ψ):*Deriv*$]\!]$ $\Longrightarrow$
(*G*, *Iff b c1 c2*, *Sec* (λ (*s*,*t*,β) .
(*evalB b* (*fst t*) $\longrightarrow$ Φ(*s*,*t*,β)) $\wedge$
(($\neg$ *evalB b* (*fst t*)) $\longrightarrow$ Ψ(*s*,*t*,β)))):*Deriv*

| *D-NEW*:

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

| *D-GET*:
$[\![\ CONTEXT\ y = low;\ GAMMA\ f = low]\!] \Longrightarrow$
$(G,\ Get\ x\ y\ f,\ Sec\ (\lambda\ (s,t,\beta)\ .$
$\quad \exists\ r\ l\ C\ Flds\ v.\ (fst\ r)\ y = RVal(Loc\ l) \wedge$
$\quad\quad lookup\ (snd\ r)\ l = Some(C,Flds) \wedge$
$\quad\quad lookup\ Flds\ f = Some\ v \wedge r \equiv_\beta t \wedge$
$\quad\quad s = (update\ (fst\ r)\ x\ v,\ snd\ r))):Deriv$

| *D-PUT*:
$[\![\ CONTEXT\ x = low;\ GAMMA\ f = low;\ Expr\text{-}low\ e]\!] \Longrightarrow$
$(G,\ Put\ x\ f\ e,\ Sec\ (\lambda\ (s,t,\beta)\ .$
$\quad \exists\ r\ l\ C\ F\ h.\ (fst\ r)\ x = RVal(Loc\ l) \wedge r \equiv_\beta t \wedge$
$\quad\quad lookup\ (snd\ r)\ l = Some(C,F) \wedge$
$\quad\quad h = (l,(C,(f,evalE\ e\ (fst\ r))\ \#\ F))\ \#\ (snd\ r) \wedge$
$\quad\quad s = (fst\ r,\ h))):Deriv$

| *D-WHILE*:
$[\![\ BExpr\text{-}low\ b;\ (G,\ c,\ Sec\ \Phi):Deriv]\!]$
$\Longrightarrow (G,\ While\ b\ c,\ Sec\ (PhiWhile\ b\ \Phi)):Deriv$

| *D-CALL*:
$[\![(\{Sec\ (FIX\ \varphi)\} \cup G,\ body,\ Sec\ (\varphi\ (FIX\ \varphi))):Deriv;\ Monotone\ \varphi]\!]$
$\Longrightarrow (G,\ Call,\ Sec\ (FIX\ \varphi)):Deriv$

| *D-SKIP-H*: $(G,\ Skip,\ HighSec):Deriv$

| *D-ASSIGN-H*:
$[\![\ CONTEXT\ x = high;\ \forall\ s\ .\ noLowDPs\ s \longrightarrow Expr\text{-}good\ e\ s]\!]$
$\Longrightarrow (G,\ Assign\ x\ e,\ HighSec):Deriv$

| *D-NEW-H*: $CONTEXT\ x = high \Longrightarrow (G,\ New\ x\ C,\ HighSec):Deriv$

| *D-GET-H*: $CONTEXT\ x = high \Longrightarrow (G,\ Get\ x\ y\ f,\ HighSec):Deriv$

| *D-PUT-H*:
$[\![\ GAMMA\ f = high;\ \forall\ s\ .\ noLowDPs\ s \longrightarrow Expr\text{-}good\ e\ s]\!]$
$\Longrightarrow (G,\ Put\ x\ f\ e,\ HighSec):Deriv$

| *D-COMP-H*:
$[\![\ (G,\ c,\ HighSec):Deriv;\ (G,\ d,\ HighSec):Deriv]\!]$
$\Longrightarrow (G,\ Comp\ c\ d,\ HighSec):Deriv$

| *D-IFF-H*:

$[\![$ $(G,\ c,\ HighSec)$:$Deriv$; $(G,\ d,\ HighSec)$:$Deriv]\!]$
$\Longrightarrow (G,\ Iff\ b\ c\ d,\ HighSec)$:$Deriv$

| *D-WHILE-H*:
$[\![$ $(G,\ c,\ HighSec)$:$Deriv]\!] \Longrightarrow (G,\ While\ b\ c,\ HighSec)$:$Deriv$

| *D-CALL-H*:
$(\{HighSec\} \cup G,\ body,\ HighSec)$:$Deriv \Longrightarrow (G,\ Call,\ HighSec)$:$Deriv$

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):$Deriv \Longrightarrow G \rhd c$: $A\langle proof\rangle$

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* $= (\forall\ \beta\ v\ v'\ w\ w'\ .\ (\beta,\ v,\ v') \in twiddleVal \longrightarrow$
$(\beta,\ w,\ w') \in twiddleVal \longrightarrow (\beta,\ f\ v\ w,\ f\ v'\ w') \in twiddleVal)$

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

inductive-set *VS-expr*:: $(Expr \times TP)$ *set*
where
VS-exprVar: $CONTEXT\ x = t \Longrightarrow (varE\ x,\ t) : VS\text{-}expr$
|
VS-exprVal:
$[\![v{=}RVal\ Nullref \vee (\exists\ i\ .\ v{=}IVal\ i)]\!] \Longrightarrow (valE\ v,\ low) : VS\text{-}expr$
|
VS-exprOp:
$[\![(e1,t) : VS\text{-}expr;\ (e2,t){:}VS\text{-}expr;\ opEGood\ f]\!]$
$\Longrightarrow (opE\ f\ e1\ e2,t) : VS\text{-}expr$
|
VS-exprHigh: $(e,\ high) : VS\text{-}expr$

inductive-set *VS-Bexpr*:: $(BExpr \times TP)$ *set*
where
VS-BexprOp:
$[\![(e1,t){:}VS\text{-}expr;\ (e2,t){:}VS\text{-}expr;\ compBGood\ f]\!]$
$\Longrightarrow (compB\ f\ e1\ e2,t){:}VS\text{-}Bexpr$
|
VS-BexprHigh: $(e,high) : VS\text{-}Bexpr$

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 ∀ *s*. *Epxr_good e s*.

inductive-set *VS-com*:: (*TP* × *OBJ*) *set*
where
VS-comGetL:
⟦ *CONTEXT y* = *low*; *GAMMA f* = *low*⟧
⟹ (*low*, *Get x y f*): *VS-com*

| *VS-comGetH*: *CONTEXT x* = *high* ⟹ (*high*, *Get x y f*): *VS-com*

| *VS-comPutL*:
⟦ *CONTEXT x* = *low*; *GAMMA f* = *low*; (*e*, *low*) : *VS-expr*⟧
⟹ (*low*,*Put x f e*): *VS-com*

| *VS-comPutH*:
⟦ *GAMMA f* = *high*; ∀ *s* . *noLowDPs s* ⟶ *Expr-good e s*⟧
⟹ (*high*,*Put x f e*): *VS-com*

| *VS-comNewL*:
CONTEXT x = *low* ⟹ (*low*, *New x c*) : *VS-com*

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

| *VS-comAssL*:
⟦*CONTEXT x* = *low*; (*e*,*low*): *VS-expr*⟧
⟹ (*low*,*Assign x e*) : *VS-com*

| *VS-comAssH*:
⟦*CONTEXT x* = *high*; ∀ *s* . *noLowDPs s* ⟶ *Expr-good e s*⟧
⟹ (*high*,*Assign x e*) : *VS-com*

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

| *VS-comComp*:
⟦ (*pc*,*c*): *VS-com*; (*pc*,*d*): *VS-com*⟧ ⟹ (*pc*,*Comp c d*) : *VS-com*

| *VS-comIf*:
⟦ (*b*,*pc*): *VS-Bexpr*; (*pc*,*c*): *VS-com*; (*pc*,*d*): *VS-com*⟧
⟹ (*pc*,*Iff b c d*): *VS-com*

| *VS-comWhile*:
⟦(*b*,*pc*): *VS-Bexpr*; (*pc*,*c*): *VS-com*⟧ ⟹ (*pc*, *While b c*): *VS-com*

| *VS-comSub*: (*high*,*c*) : *VS-com* ⟹ (*low*,*c*): *VS-com*

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

primrec *SemExpr*::*Expr* ⇒ *TP* ⇒ *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* ⟹ *SemExpr e tp*⟨*proof*⟩

Likewise for the boolean expressions.

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

lemma *BExprSound*: (*e*,*tp*):*VS-Bexpr* ⟹ *SemBExpr e tp*⟨*proof*⟩

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* ⟹ (*t*=*high* ⟶ (*G*, *c*, *HighSec*):*Deriv*) ∧
(*t*=*low* ⟶ (∃ Φ . (*G*, *c*, *Sec* Φ):*Deriv*))⟨*proof*⟩

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* ⟹ *secure c*⟨*proof*⟩

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* = *Iff 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
Vars Skip = {} |
Vars (*Assign x e*) = *EVars e* |
Vars (*New x C*) = {} |
Vars (*Get x y f*) = {*y*} |
Vars (*Put x f e*) = *EVars e* |
Vars (*Comp I J*) = *Vars I* $\cup$ *Vars J* |
Vars (*While b I*) = *BVars b* $\cup$ *Vars I* |
Vars (*Iff b I J*) = *BVars b* $\cup$ *Vars I* $\cup$ *Vars J* |
Vars Call = {}
⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

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

definition *ValIsNoLoc* :: *Val* => *bool*
where *ValIsNoLoc 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* ⇒ *bool*
where
Expr-noLoc (*varE x*) = *True* |
Expr-noLoc (*valE v*) = *ValIsNoLoc v* |
Expr-noLoc (*opE f e1 e2*) =
  (*Expr-noLoc e1* ∧ *Expr-noLoc e2* ∧ *opEGood f*)

primrec *BExpr-noLoc*::*BExpr* ⇒ *bool*
where
BExpr-noLoc (*compB f e1 e2*) =
  (*Expr-noLoc e1* ∧ *Expr-noLoc e2* ∧ *compBGood f*)

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

lemma *Expr-lemma1*[*rule-format*]:
  Expr-noLoc e ⟶ *EVars e* ⊆ *X* ⟶
  (∀ *x*. *x* ∈ *X* ⟶ *CONTEXT x* = *low*) ⟶ *Expr-low e*⟨*proof*⟩
lemma *Expr-lemma2*[*rule-format*]:
  noLowDPs (*s*, *h*) ⟶
    EVars e ⊆ *X* ⟶ *Expr-noLoc e* ⟶
    (∀ *x*. *x* ∈ *X* ⟶ *CONTEXT x* = *low*) ⟶
    $s \approx_\beta t$ ⟶ *twiddleHeap β h k* ⟶
    noLowDPs (λ*y*. *if x* = *y then evalE e s else s y*, *h*)⟨*proof*⟩
lemma *Expr-lemma3*[*rule-format*]:
 noLowDPs (*s*,*h*) ⟶ *EVars e* ⊆ *X* ⟶ *Expr-noLoc e* ⟶
  lookup h l = *Some* (*C*, *Flds*) ⟶
  (∀ *x*. *x* ∈ *X* ⟶ *CONTEXT x* = *low*) ⟶
  $s \approx_\beta t$ ⟶ *twiddleHeap β h k* ⟶
  noLowDPs (*s*, (*l*, *C*, (*f*, *evalE e s*) # *Flds*) # *h*)⟨*proof*⟩

The first of these can be lifted to boolean expressions.

lemma *BExpr-lemma*:
⟦*BVars b* ⊆ *X*; ∀ *x*. *x* ∈ *X* ⟶ *CONTEXT x* = *low*; *BExpr-noLoc b*⟧ ⟹ *BExpr-low b*⟨*proof*⟩

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* ⇒ *CtxtProg* ⇒ *bool*
where
CtxtVars X Ctxt-Here = *True* |
CtxtVars X Ctxt-Skip = *True* |
CtxtVars X (*Ctxt-Assign x e*) = (*EVars e* ⊆ *X* ∧ *Expr-noLoc e*) |
CtxtVars X (*Ctxt-New x c*) = *True* |
CtxtVars X (*Ctxt-Get x y f*) = (*y* : *X* ∧ *GAMMA f* = *low*) |
CtxtVars X (*Ctxt-Put x f e*) =
  (*EVars e* ⊆ *X* ∧ *CONTEXT x* = *low* ∧ *Expr-noLoc e*) |
CtxtVars X (*Ctxt-Comp C D*) = (*CtxtVars X C* ∧ *CtxtVars X D*) |

$CtxtVars\ X\ (Ctxt\text{-}If\ b\ C\ D) =$
$(BVars\ b \subseteq X \wedge CtxtVars\ X\ C \wedge CtxtVars\ X\ D \wedge BExpr\text{-}noLoc\ b)\ |$
$CtxtVars\ X\ (Ctxt\text{-}While\ b\ C) =$
$(BVars\ b \subseteq X \wedge CtxtVars\ X\ C \wedge BExpr\text{-}noLoc\ b)\ |$
$CtxtVars\ X\ Ctxt\text{-}Call = True$

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

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*:
$[\![secure\ I;\ LOW\ X\ C;\ LOW\ X\ Ctxt\text{-}Body;\ body = Fill\ Ctxt\text{-}Body\ I]\!]$
$\Longrightarrow secure\ (Fill\ C\ I)\langle proof \rangle$

For a variable

consts $res :: Var$

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

theorem *SecureForAttackingContext*:
$[\![\ secure\ I;\ LOW\ X\ C;\ \ LOW\ X\ Ctxt\text{-}Body;\ s \equiv_\beta ss;$
$s,(Fill\ C\ I)\Downarrow t;\ ss,(Fill\ C\ I)\Downarrow tt;\ body = Fill\ Ctxt\text{-}Body\ I;$
$CONTEXT\ res = low]\!]$
$\Longrightarrow \exists\ \gamma.\ (\gamma,(fst\ t)\ res,(fst\ tt)\ res){:}twiddleVal \wedge Pbij\text{-}extends\ \gamma\ \beta\langle proof \rangle$

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 nondeterminism. 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.