An Isabelle Correctness Proof for the Volpano/Smith Security Typing System

Gregor Snelting and Daniel Wasserrab
IPD Snelting
Universität Karlsruhe (TH)

October 13, 2025

Abstract

The Volpano/Smith/Irvine security type systems [2] requires that variables are annotated as high (secret) or low (public), and provides typing rules which guarantee that secret values cannot leak to public output ports. This property of a program is called confidentiality.

For a simple while-language without threads, our proof shows that typeability in the Volpano/Smith system guarantees noninterference. Noninterference means that if two initial states for program execution are low-equivalent, then the final states are low-equivalent as well. This indeed implies that secret values cannot leak to public ports. For more details on noninterference and security typing systems, see [1].

The proof defines an abstract syntax and operational semantics for programs, formalizes noninterference, and then proceeds by rule induction on the operational semantics. The mathematically most intricate part is the treatment of implicit flows. Note that the Volpano/Smith system is not flow-sensitive and thus quite unprecise, resulting in false alarms. However, due to the correctness property, all potential breaks of confidentiality are discovered.

Contents

theory *Semantics*
imports *Main*
begin

1 The Language

1.1 Variables and Values

type-synonym *vname* = *string* — names for variables

datatype *val*
  = *Bool bool* — Boolean value
  | *Intg int* — integer value

abbreviation *true* == *Bool True*
abbreviation *false* == *Bool False*

1.2 Expressions and Commands

datatype *bop* = *Eq* | *And* | *Less* | *Add* | *Sub* — names of binary operations

datatype *expr*
  = *Val val* — value
  | *Var vname* — local variable
  | *BinOp expr bop expr* (‹- «-» -› [*80*,*0*,*81*] *80*) — binary operation

Note: we assume that only type correct expressions are regarded as later proofs fail if expressions evaluate to None due to type errors. However there is [yet] no typing system

fun *binop* :: *bop* ⇒ *val* ⇒ *val* ⇒ *val option*
where *binop Eq* v_1 v_2 = *Some*(*Bool*($v_1 = v_2$))
  | *binop And* (*Bool* b_1) (*Bool* b_2) = *Some*(*Bool*($b_1 \wedge b_2$))
  | *binop Less* (*Intg* i_1) (*Intg* i_2) = *Some*(*Bool*($i_1 < i_2$))
  | *binop Add* (*Intg* i_1) (*Intg* i_2) = *Some*(*Intg*($i_1 + i_2$))
  | *binop Sub* (*Intg* i_1) (*Intg* i_2) = *Some*(*Intg*($i_1 - i_2$))

  | *binop bop* v_1 v_2 = *Some*(*Intg*(*0*))

datatype *com*
  = *Skip*
  | *LAss vname expr* (‹-:=-› [*70*,*70*] *70*) — local assignment
  | *Seq com com* (‹-;;/ -› [*61*,*60*] *60*)
  | *Cond expr com com* (‹*if* ′(-′) -/ *else* -› [*80*,*79*,*79*] *70*)
  | *While expr com* (‹*while* ′(-′) -› [*80*,*79*] *70*)

fun *fv* :: *expr* ⇒ *vname set* — free variables in an expression
where

FVc: *fv* (*Val V*) = {}
| *FVv*: *fv* (*Var V*) = {*V*}
| *FVe*: *fv* (*e1* «*bop*» *e2*) = *fv e1* ∪ *fv e2*

1.3 State

type-synonym *state* = *vname* ⇀ *val*

interpret silently assumes type correct expressions, i.e. no expression evaluates to None

fun *interpret* :: *expr* ⇒ *state* ⇒ *val option* (‹⟦-⟧-›)
where *Val*: ⟦*Val v*⟧ *s* = *Some v*
| *Var*: ⟦*Var V*⟧ *s* = *s V*
| *BinOp*: $[\![e_1«bop»e_2]\!]$ *s* = (*case* $[\![e_1]\!]$ *s of None* ⇒ *None*
| *Some* v_1 ⇒ (*case* $[\![e_2]\!]$ *s of None* ⇒ *None*
| *Some* v_2 ⇒ *binop bop* v_1 v_2))

1.4 Small Step Semantics

inductive *red* :: *com* ∗ *state* ⇒ *com* ∗ *state* ⇒ *bool*
and *red′* :: *com* ⇒ *state* ⇒ *com* ⇒ *state* ⇒ *bool*
(‹((1⟨-,/-⟩) →/ (1⟨-,/-⟩))› [0,0,0,0] 81)
where
⟨*c1*,*s1*⟩ → ⟨*c2*,*s2*⟩ == *red* (*c1*,*s1*) (*c2*,*s2*) |
RedLAss:
⟨*V*:=*e*,*s*⟩ → ⟨*Skip*,*s*(*V*:=(⟦*e*⟧ *s*))⟩

| *SeqRed*:
$\langle c_1,s\rangle \to \langle c_1',s'\rangle \Longrightarrow \langle c_1;;c_2,s\rangle \to \langle c_1';;c_2,s'\rangle$

| *RedSeq*:
$\langle Skip;;c_2,s\rangle \to \langle c_2,s\rangle$

| *RedCondTrue*:
$[\![b]\!]$ *s* = *Some true* ⟹ $\langle if\ (b)\ c_1\ else\ c_2,s\rangle \to \langle c_1,s\rangle$

| *RedCondFalse*:
$[\![b]\!]$ *s* = *Some false* ⟹ $\langle if\ (b)\ c_1\ else\ c_2,s\rangle \to \langle c_2,s\rangle$

| *RedWhileTrue*:
⟦*b*⟧ *s* = *Some true* ⟹ ⟨*while* (*b*) *c*,*s*⟩ → ⟨*c*;;*while* (*b*) *c*,*s*⟩

| *RedWhileFalse*:
⟦*b*⟧ *s* = *Some false* ⟹ ⟨*while* (*b*) *c*,*s*⟩ → ⟨*Skip*,*s*⟩

lemmas *red-induct* = *red.induct*[*split-format* (*complete*)]

abbreviation *reds* ::*com* ⇒ *state* ⇒ *com* ⇒ *state* ⇒ *bool*
(‹((1⟨-,/-⟩) →∗/ (1⟨-,/-⟩))› [0,0,0,0] 81) **where**
⟨*c*,*s*⟩ →∗ ⟨*c′*,*s′*⟩ == red^{**} (*c*,*s*) (*c′*,*s′*)

lemma *Skip-reds*:
  $\langle Skip,s\rangle \rightarrow* \langle c',s'\rangle \Longrightarrow s = s' \wedge c' = Skip$
by(*blast elim*:*converse-rtranclpE red.cases*)

lemma *LAss-reds*:
  $\langle V{:=}e,s\rangle \rightarrow* \langle Skip,s'\rangle \Longrightarrow s' = s(V{:=}[\![e]\!]\ s)$
proof(*induct V*:=*e s rule*:*converse-rtranclp-induct2*)
  case (*step s c'' s''*)
  hence $c'' = Skip$ **and** $s'' = s(V{:=}([\![e]\!]\ s))$ **by**(*auto elim*:*red.cases*)
  with ‹$\langle c'',s''\rangle \rightarrow* \langle Skip,s'\rangle$› **show** *?case* **by**(*auto dest*:*Skip-reds*)
qed

lemma *Seq2-reds*:
  $\langle Skip;;c_2,s\rangle \rightarrow* \langle Skip,s'\rangle \Longrightarrow \langle c_2,s\rangle \rightarrow* \langle Skip,s'\rangle$
by(*induct c*==*Skip*;;c_2 *s rule*:*converse-rtranclp-induct2*)(*auto elim*:*red.cases*)

lemma *Seq-reds*:
  assumes $\langle c_1;;c_2,s\rangle \rightarrow* \langle Skip,s'\rangle$
  obtains s'' **where** $\langle c_1,s\rangle \rightarrow* \langle Skip,s''\rangle$ **and** $\langle c_2,s''\rangle \rightarrow* \langle Skip,s'\rangle$
proof −
  have $\exists s''.\ \langle c_1,s\rangle \rightarrow* \langle Skip,s''\rangle \wedge \langle c_2,s''\rangle \rightarrow* \langle Skip,s'\rangle$
  proof −
    { fix *c c'*
      assume $\langle c,s\rangle \rightarrow* \langle c',s'\rangle$ **and** $c = c_1;;c_2$ **and** $c' = Skip$
      hence $\exists s''.\ \langle c_1,s\rangle \rightarrow* \langle Skip,s''\rangle \wedge \langle c_2,s''\rangle \rightarrow* \langle Skip,s'\rangle$
      proof(*induct arbitrary*:c_1 *rule*:*converse-rtranclp-induct2*)
        case *refl* **thus** *?case* **by** *simp*
      next
        case (*step c s c'' s''*)
        note *IH* = ‹$\bigwedge c_1.\ [\![c'' = c_1;;c_2;\ c' = Skip]\!]$
          $\Longrightarrow \exists sx.\ \langle c_1,s''\rangle \rightarrow* \langle Skip,sx\rangle \wedge \langle c_2,sx\rangle \rightarrow* \langle Skip,s'\rangle$›
        from *step*
        have $\langle c_1;;c_2,s\rangle \rightarrow \langle c'',s''\rangle$ **by** *simp*
        hence $(c_1 = Skip \wedge c'' = c_2 \wedge s = s'') \vee$
          $(\exists c_1'.\ \langle c_1,s\rangle \rightarrow \langle c_1',s''\rangle \wedge c'' = c_1';;c_2)$
          by(*auto elim*:*red.cases*)
        thus *?case*
        proof
          assume $c_1 = Skip \wedge c'' = c_2 \wedge s = s''$
          with ‹$\langle c'',s''\rangle \rightarrow* \langle c',s'\rangle$› ‹$c' = Skip$›
          show *?thesis* **by** *auto*
        next
          assume $\exists c_1'.\ \langle c_1,s\rangle \rightarrow \langle c_1',s''\rangle \wedge c'' = c_1';;c_2$
          then obtain c_1' **where** $\langle c_1,s\rangle \rightarrow \langle c_1',s''\rangle$ **and** $c'' = c_1';;c_2$ **by** *blast*
          from *IH*[*OF* ‹$c'' = c_1';;c_2$› ‹$c' = Skip$›]
          obtain *sx* **where** $\langle c_1',s''\rangle \rightarrow* \langle Skip,sx\rangle$ **and** $\langle c_2,sx\rangle \rightarrow* \langle Skip,s'\rangle$
            by *blast*
          from ‹$\langle c_1,s\rangle \rightarrow \langle c_1',s''\rangle$› ‹$\langle c_1',s''\rangle \rightarrow* \langle Skip,sx\rangle$›

have $\langle c_1,s\rangle \rightarrow* \langle Skip,sx\rangle$ **by**(*auto intro*:*converse-rtranclp-into-rtranclp*)
with ‹$\langle c_2,sx\rangle \rightarrow* \langle Skip,s'\rangle$› **show** *?thesis* **by** *auto*
qed
qed }
with ‹$\langle c_1;;c_2,s\rangle \rightarrow* \langle Skip,s'\rangle$› **show** *?thesis* **by** *simp*
qed
with *that* **show** *?thesis* **by** *blast*
qed

lemma *Cond-True-or-False*:
$\langle if\ (b)\ c_1\ else\ c_2,s\rangle \rightarrow* \langle Skip,s'\rangle \Longrightarrow [\![b]\!]\ s = Some\ true \vee [\![b]\!]\ s = Some\ false$
by(*induct c==if* (*b*) c_1 *else* c_2 *s rule*:*converse-rtranclp-induct2*)(*auto elim*:*red.cases*)

lemma *CondTrue-reds*:
$\langle if\ (b)\ c_1\ else\ c_2,s\rangle \rightarrow* \langle Skip,s'\rangle \Longrightarrow [\![b]\!]\ s = Some\ true \Longrightarrow \langle c_1,s\rangle \rightarrow* \langle Skip,s'\rangle$
by(*induct c==if* (*b*) c_1 *else* c_2 *s rule*:*converse-rtranclp-induct2*)(*auto elim*:*red.cases*)

lemma *CondFalse-reds*:
$\langle if\ (b)\ c_1\ else\ c_2,s\rangle \rightarrow* \langle Skip,s'\rangle \Longrightarrow [\![b]\!]\ s = Some\ false \Longrightarrow \langle c_2,s\rangle \rightarrow* \langle Skip,s'\rangle$
by(*induct c==if* (*b*) c_1 *else* c_2 *s rule*:*converse-rtranclp-induct2*)(*auto elim*:*red.cases*)

lemma *WhileFalse-reds*:
$\langle while\ (b)\ cx,s\rangle \rightarrow* \langle Skip,s'\rangle \Longrightarrow [\![b]\!]\ s = Some\ false \Longrightarrow s = s'$
proof(*induct while* (*b*) *cx s rule*:*converse-rtranclp-induct2*)
case *step* **thus** *?case* **by**(*auto elim*:*red.cases dest*: *Skip-reds*)
qed

lemma *WhileTrue-reds*:
$\langle while\ (b)\ cx,s\rangle \rightarrow* \langle Skip,s'\rangle \Longrightarrow [\![b]\!]\ s = Some\ true$
$\Longrightarrow \exists\, sx.\ \langle cx,s\rangle \rightarrow* \langle Skip,sx\rangle \wedge \langle while\ (b)\ cx,sx\rangle \rightarrow* \langle Skip,s'\rangle$
proof(*induct while* (*b*) *cx s rule*:*converse-rtranclp-induct2*)
case (*step s* c'' s'')
hence $c'' = cx;;while\ (b)\ cx \wedge s'' = s$ **by**(*auto elim*:*red.cases*)
with ‹$\langle c'',s''\rangle \rightarrow* \langle Skip,s'\rangle$› **show** *?case* **by**(*auto dest*:*Seq-reds*)
qed

lemma *While-True-or-False*:
$\langle while\ (b)\ com,s\rangle \rightarrow* \langle Skip,s'\rangle \Longrightarrow [\![b]\!]\ s = Some\ true \vee [\![b]\!]\ s = Some\ false$
by(*induct c==while* (*b*) *com s rule*:*converse-rtranclp-induct2*)(*auto elim*:*red.cases*)

inductive *red-n* :: *com* ⇒ *state* ⇒ *nat* ⇒ *com* ⇒ *state* ⇒ *bool*
(‹((*1*⟨-,/-⟩) →^- (*1*⟨-,/-⟩))› [*0*,*0*,*0*,*0*,*0*] *81*)
where *red-n-Base*: $\langle c,s\rangle \rightarrow^{0} \langle c,s\rangle$
| *red-n-Rec*: $[\![\langle c,s\rangle \rightarrow \langle c'',s''\rangle; \langle c'',s''\rangle \rightarrow^{n} \langle c',s'\rangle]\!] \Longrightarrow \langle c,s\rangle \rightarrow^{Suc\ n} \langle c',s'\rangle$

lemma *Seq-red-nE*: **assumes** $\langle c_1;;c_2,s\rangle \rightarrow^{n} \langle Skip,s'\rangle$

obtains $i\ j\ s''$ **where** $\langle c_1,s\rangle \rightarrow^{i} \langle Skip,s''\rangle$ **and** $\langle c_2,s''\rangle \rightarrow^{j} \langle Skip,s'\rangle$
and $n = i + j + 1$
proof $-$
from ‹$\langle c_1;;c_2,s\rangle \rightarrow^{n} \langle Skip,s'\rangle$›
have $\exists\, i\ j\ s''.\ \langle c_1,s\rangle \rightarrow^{i} \langle Skip,s''\rangle \wedge \langle c_2,s''\rangle \rightarrow^{j} \langle Skip,s'\rangle \wedge n = i + j + 1$
proof(*induct* $c_1;;c_2\ s\ n\ Skip\ s'$ *arbitrary*:c_1 *rule*:*red-n.induct*)
case (*red-n-Rec* $s\ c''\ s''\ n\ s'$)
note IH = ‹$\bigwedge c_1.\ c'' = c_1;;c_2$
$\Longrightarrow \exists\, i\ j\ sx.\ \langle c_1,s''\rangle \rightarrow^{i} \langle Skip,sx\rangle \wedge \langle c_2,sx\rangle \rightarrow^{j} \langle Skip,s'\rangle \wedge n = i + j + 1$›
from ‹$\langle c_1;;c_2,s\rangle \rightarrow \langle c'',s''\rangle$›
have $(c_1 = Skip \wedge c'' = c_2 \wedge s = s'') \vee$
$(\exists\, c_1'.\ c'' = c_1';;c_2 \wedge \langle c_1,s\rangle \rightarrow \langle c_1',s''\rangle)$
by(*induct* $c_1;;c_2$ - - - *rule*:*red-induct*) *auto*
thus *?case*
proof
assume $c_1 = Skip \wedge c'' = c_2 \wedge s = s''$
hence $c_1 = Skip$ **and** $c'' = c_2$ **and** $s = s''$ **by** *simp-all*
from ‹$c_1 = Skip$› **have** $\langle c_1,s\rangle \rightarrow^{0} \langle Skip,s\rangle$ **by**(*fastforce intro*:*red-n-Base*)
with ‹$\langle c'',s''\rangle \rightarrow^{n} \langle Skip,s'\rangle$› ‹$c'' = c_2$› ‹$s = s''$›
show *?thesis* **by**(*rule-tac x=0* **in** *exI*) *auto*
next
assume $\exists\, c_1'.\ c'' = c_1';;c_2 \wedge \langle c_1,s\rangle \rightarrow \langle c_1',s''\rangle$
then obtain c_1' **where** $c'' = c_1';;c_2$ **and** $\langle c_1,s\rangle \rightarrow \langle c_1',s''\rangle$ **by** *blast*
from $IH[OF$ ‹$c'' = c_1';;c_2$›$]$ **obtain** $i\ j\ sx$
where $\langle c_1',s''\rangle \rightarrow^{i} \langle Skip,sx\rangle$ **and** $\langle c_2,sx\rangle \rightarrow^{j} \langle Skip,s'\rangle$
and $n = i + j + 1$ **by** *blast*
from ‹$\langle c_1,s\rangle \rightarrow \langle c_1',s''\rangle$› ‹$\langle c_1',s''\rangle \rightarrow^{i} \langle Skip,sx\rangle$›
have $\langle c_1,s\rangle \rightarrow^{Suc\ i} \langle Skip,sx\rangle$ **by**(*rule red-n.red-n-Rec*)
with ‹$\langle c_2,sx\rangle \rightarrow^{j} \langle Skip,s'\rangle$› ‹$n = i + j + 1$› **show** *?thesis*
by(*rule-tac x=Suc i* **in** *exI*) *auto*
qed
qed
with *that* **show** *?thesis* **by** *blast*
qed

lemma *while-red-nE*:
$\langle while\ (b)\ cx,s\rangle \rightarrow^{n} \langle Skip,s'\rangle$
$\Longrightarrow ([\![b]\!]\ s = Some\ false \wedge s = s' \wedge n = 1) \vee$
$(\exists\, i\ j\ s''.\ [\![b]\!]\ s = Some\ true \wedge \langle cx,s\rangle \rightarrow^{i} \langle Skip,s''\rangle \wedge$
$\langle while\ (b)\ cx,s''\rangle \rightarrow^{j} \langle Skip,s'\rangle \wedge n = i + j + 2)$
proof(*induct while* $(b)\ cx\ s\ n\ Skip\ s'$ *rule*:*red-n.induct*)
case (*red-n-Rec* $s\ c''\ s''\ n\ s'$)
from ‹$\langle while\ (b)\ cx,s\rangle \rightarrow \langle c'',s''\rangle$›
have $([\![b]\!]\ s = Some\ false \wedge c'' = Skip \wedge s'' = s) \vee$
$([\![b]\!]\ s = Some\ true \wedge c'' = cx;;while\ (b)\ cx \wedge s'' = s)$
by(*induct while* (b) cx - - - *rule*:*red-induct*) *auto*
thus *?case*
proof

assume $[\![b]\!]\ s = Some\ false \wedge c'' = Skip \wedge s'' = s$
hence $[\![b]\!]\ s = Some\ false$ **and** $c'' = Skip$ **and** $s'' = s$ **by** *simp-all*
with ‹$\langle c'',s'\rangle \rightarrow^{n} \langle Skip,s'\rangle$› **have** $s = s'$ **and** $n = 0$
by(*induct - - Skip - rule*:*red-n.induct*,*auto elim*:*red.cases*)
with ‹$[\![b]\!]\ s = Some\ false$› **show** *?thesis* **by** *fastforce*
next
assume $[\![b]\!]\ s = Some\ true \wedge c'' = cx;;while\ (b)\ cx \wedge s'' = s$
hence $[\![b]\!]\ s = Some\ true$ **and** $c'' = cx;;while\ (b)\ cx$
and $s'' = s$ **by** *simp-all*
with ‹$\langle c'',s'\rangle \rightarrow^{n} \langle Skip,s'\rangle$›
obtain *i j sx* **where** $\langle cx,s\rangle \rightarrow^{i} \langle Skip,sx\rangle$ **and** $\langle while\ (b)\ cx,sx\rangle \rightarrow^{j} \langle Skip,s'\rangle$
and $n = i + j + 1$ **by**(*fastforce elim*:*Seq-red-nE*)
with ‹$[\![b]\!]\ s = Some\ true$› **show** *?thesis* **by** *fastforce*
qed
qed

lemma *while-red-n-induct* [*consumes 1*, *case-names false true*]:
assumes *major*: $\langle while\ (b)\ cx,s\rangle \rightarrow^{n} \langle Skip,s'\rangle$
and *IHfalse*:$\bigwedge s.\ [\![b]\!]\ s = Some\ false \Longrightarrow P\ s\ s$
and *IHtrue*:$\bigwedge s\ i\ j\ s''.\ [\![\ [\![b]\!]\ s = Some\ true;\ \langle cx,s\rangle \rightarrow^{i} \langle Skip,s''\rangle;$
$\langle while\ (b)\ cx,s''\rangle \rightarrow^{j} \langle Skip,s'\rangle;\ P\ s''\ s']\!] \Longrightarrow P\ s\ s'$
shows $P\ s\ s'$
using *major*
proof(*induct n arbitrary*:*s rule*:*nat-less-induct*)
fix *n s*
assume *IHall*:$\forall m<n.\ \forall x.\ \langle while\ (b)\ cx,x\rangle \rightarrow^{m} \langle Skip,s'\rangle \longrightarrow P\ x\ s'$
and $\langle while\ (b)\ cx,s\rangle \rightarrow^{n} \langle Skip,s'\rangle$
from ‹$\langle while\ (b)\ cx,s\rangle \rightarrow^{n} \langle Skip,s'\rangle$›
have $([\![b]\!]\ s = Some\ false \wedge s = s' \wedge n = 1) \vee$
$(\exists\, i\ j\ s''.\ [\![b]\!]\ s = Some\ true \wedge \langle cx,s\rangle \rightarrow^{i} \langle Skip,s''\rangle \wedge$
$\langle while\ (b)\ cx,s''\rangle \rightarrow^{j} \langle Skip,s'\rangle \wedge n = i + j + 2)$
by(*rule while-red-nE*)
thus $P\ s\ s'$
proof
assume $[\![b]\!]\ s = Some\ false \wedge s = s' \wedge n = 1$
hence $[\![b]\!]\ s = Some\ false$ **and** $s = s'$ **by** *auto*
from *IHfalse*[*OF* ‹$[\![b]\!]\ s = Some\ false$›] **have** $P\ s\ s$ **.**
with ‹$s = s'$› **show** *?thesis* **by** *simp*
next
assume $\exists\, i\ j\ s''.\ [\![b]\!]\ s = Some\ true \wedge \langle cx,s\rangle \rightarrow^{i} \langle Skip,s''\rangle \wedge$
$\langle while\ (b)\ cx,s''\rangle \rightarrow^{j} \langle Skip,s'\rangle \wedge n = i + j + 2$
then obtain $i\ j\ s''$ **where** $[\![b]\!]\ s = Some\ true$
and $\langle cx,s\rangle \rightarrow^{i} \langle Skip,s''\rangle$ **and** $\langle while\ (b)\ cx,s''\rangle \rightarrow^{j} \langle Skip,s'\rangle$
and $n = i + j + 2$ **by** *blast*
with *IHall* **have** $P\ s''\ s'$
apply(*erule-tac x*=*j* **in** *allE*) **apply** *clarsimp* **done**
from *IHtrue*[*OF* ‹$[\![b]\!]\ s = Some\ true$› ‹$\langle cx,s\rangle \rightarrow^{i} \langle Skip,s''\rangle$›
‹$\langle while\ (b)\ cx,s''\rangle \rightarrow^{j} \langle Skip,s'\rangle$› *this*] **show** *?thesis* **.**

qed
qed

lemma *reds-to-red-n*:$\langle c,s\rangle \rightarrow* \langle c',s'\rangle \Longrightarrow \exists\, n.\ \langle c,s\rangle \rightarrow^n \langle c',s'\rangle$
by(*induct rule*:*converse-rtranclp-induct2*,*auto intro*:*red-n.intros*)

lemma *red-n-to-reds*:$\langle c,s\rangle \rightarrow^n \langle c',s'\rangle \Longrightarrow \langle c,s\rangle \rightarrow* \langle c',s'\rangle$
by(*induct rule*:*red-n.induct*,*auto intro*:*converse-rtranclp-into-rtranclp*)

lemma *while-reds-induct*[*consumes 1*, *case-names false true*]:
$[\![\langle while\ (b)\ cx,s\rangle \rightarrow* \langle Skip,s'\rangle;\ \bigwedge s.\ [\![b]\!]\ s = Some\ false \Longrightarrow P\ s\ s;$
$\bigwedge s\ s''.\ [\![\ [\![b]\!]\ s = Some\ true;\ \langle cx,s\rangle \rightarrow* \langle Skip,s''\rangle;$
$\langle while\ (b)\ cx,s''\rangle \rightarrow* \langle Skip,s'\rangle;\ P\ s''\ s'\,]\!] \Longrightarrow P\ s\ s'\,]\!]$
$\Longrightarrow P\ s\ s'$
apply(*drule reds-to-red-n*,*clarsimp*)
apply(*erule while-red-n-induct*,*clarsimp*)
by(*auto dest*:*red-n-to-reds*)

lemma *red-det*:
$[\![\langle c,s\rangle \rightarrow \langle c_1,s_1\rangle;\ \langle c,s\rangle \rightarrow \langle c_2,s_2\rangle]\!] \Longrightarrow c_1 = c_2 \wedge s_1 = s_2$
proof(*induct arbitrary*:c_2 *rule*:*red-induct*)
case ($SeqRed\ c_1\ s\ c_1{}'\ s'\ c_2{}'$)
note $IH = ‹\bigwedge c_2.\ \langle c_1,s\rangle \rightarrow \langle c_2,s_2\rangle \Longrightarrow c_1{}' = c_2 \wedge s' = s_2›$
from $‹\langle c_1;;c_2{}',s\rangle \rightarrow \langle c_2,s_2\rangle›$ **have** $c_1 = Skip \vee (\exists\, cx.\ c_2 = cx;;c_2{}' \wedge \langle c_1,s\rangle \rightarrow \langle cx,s_2\rangle)$
by(*fastforce elim*:*red.cases*)
thus *?case*
proof
assume $c_1 = Skip$
with $‹\langle c_1,s\rangle \rightarrow \langle c_1{}',s'\rangle›$ **have** *False* **by**(*fastforce elim*:*red.cases*)
thus *?thesis* **by** *simp*
next
assume $\exists\, cx.\ c_2 = cx;;c_2{}' \wedge \langle c_1,s\rangle \rightarrow \langle cx,s_2\rangle$
then obtain cx **where** $c_2 = cx;;c_2{}'$ **and** $\langle c_1,s\rangle \rightarrow \langle cx,s_2\rangle$ **by** *blast*
from $IH[OF\ ‹\langle c_1,s\rangle \rightarrow \langle cx,s_2\rangle›]$ **have** $c_1{}' = cx \wedge s' = s_2$ **.**
with $‹c_2 = cx;;c_2{}'›$ **show** *?thesis* **by** *simp*
qed
qed (*fastforce elim*:*red.cases*)+

theorem *reds-det*:
$[\![\langle c,s\rangle \rightarrow* \langle Skip,s_1\rangle;\ \langle c,s\rangle \rightarrow* \langle Skip,s_2\rangle]\!] \Longrightarrow s_1 = s_2$
proof(*induct rule*:*converse-rtranclp-induct2*)
case *refl*
from $‹\langle Skip,s_1\rangle \rightarrow* \langle Skip,s_2\rangle›$ **show** *?case*

by $-$(*erule converse-rtranclpE*,*auto elim*:*red.cases*)
next
case (*step* c'' s'' c' s')
note *IH* = ‹$\langle c',s'\rangle \rightarrow* \langle Skip,s_2\rangle \Longrightarrow s_1 = s_2$›
from *step* **have** $\langle c'',s''\rangle \rightarrow \langle c',s'\rangle$
by *simp*
from ‹$\langle c'',s''\rangle \rightarrow* \langle Skip,s_2\rangle$› *this* **have** $\langle c',s'\rangle \rightarrow* \langle Skip,s_2\rangle$
by $-$(*erule converse-rtranclpE*,*auto elim*:*red.cases dest*:*red-det*)
from *IH*[*OF this*] **show** *?thesis* **.**
qed

end
theory *secTypes*
imports *Semantics*
begin

2 Security types

2.1 Security definitions

datatype *secLevel* = *Low* | *High*

type-synonym *typeEnv* = *vname* ⇀ *secLevel*

inductive *secExprTyping* :: *typeEnv* ⇒ *expr* ⇒ *secLevel* ⇒ *bool* (‹- ⊢ - : -›)
where *typeVal*: $\Gamma \vdash Val\ V : lev$

| *typeVar*: $\Gamma\ Vn = Some\ lev \Longrightarrow \Gamma \vdash Var\ Vn : lev$

| *typeBinOp1*: $[\![\Gamma \vdash e1 : Low; \Gamma \vdash e2 : Low]\!] \Longrightarrow \Gamma \vdash e1$ «*bop*» $e2 : Low$

| *typeBinOp2*: $[\![\Gamma \vdash e1 : High; \Gamma \vdash e2 : lev]\!] \Longrightarrow \Gamma \vdash e1$ «*bop*» $e2 : High$

| *typeBinOp3*: $[\![\Gamma \vdash e1 : lev; \Gamma \vdash e2 : High]\!] \Longrightarrow \Gamma \vdash e1$ «*bop*» $e2 : High$

inductive *secComTyping* :: *typeEnv* ⇒ *secLevel* ⇒ *com* ⇒ *bool* (‹-,- ⊢ -›)
where *typeSkip*: $\Gamma,T \vdash Skip$

| *typeAssH*: $\Gamma\ V = Some\ High \Longrightarrow \Gamma,T \vdash V := e$

| *typeAssL*: $[\![\Gamma \vdash e : Low; \Gamma\ V = Some\ Low]\!] \Longrightarrow \Gamma,Low \vdash V := e$

| *typeSeq*: $[\![\Gamma,T \vdash c1; \Gamma,T \vdash c2]\!] \Longrightarrow \Gamma,T \vdash c1;;c2$

| *typeWhile*: $[\![\Gamma \vdash b : T; \Gamma,T \vdash c]\!] \Longrightarrow \Gamma,T \vdash while\ (b)\ c$

| *typeIf*: $[\![\Gamma \vdash b : T; \Gamma,T \vdash c1; \Gamma,T \vdash c2]\!] \Longrightarrow \Gamma,T \vdash \mathit{if}\ (b)\ c1\ \mathit{else}\ c2$

| *typeConvert*: $\Gamma,\mathit{High} \vdash c \Longrightarrow \Gamma,\mathit{Low} \vdash c$

2.2 Lemmas concerning expressions

lemma *exprTypeable*:
  assumes *fv e* ⊆ *dom* Γ **obtains** *T* **where** Γ ⊢ *e* : *T*
proof −
  from ‹*fv e* ⊆ *dom* Γ› **have** ∃ *T*. Γ ⊢ *e* : *T*
  proof(*induct e*)
    case (*Val V*)
    have Γ ⊢ *Val V* : *Low* **by**(*rule typeVal*)
    thus *?case* **by** (*rule exI*)
  next
    case (*Var V*)
    have *V* ∈ *fv* (*Var V*) **by** *simp*
    with ‹*fv* (*Var V*) ⊆ *dom* Γ› **have** *V* ∈ *dom* Γ **by** *simp*
    then obtain *T* **where** Γ *V* = *Some T* **by** *auto*
    hence Γ ⊢ *Var V* : *T* **by** (*rule typeVar*)
    thus *?case* **by** (*rule exI*)
  next
    case (*BinOp e1 bop e2*)
    note *IH1* = ‹*fv e1* ⊆ *dom* Γ ⟹ ∃ *T*. Γ ⊢ *e1* : *T*›
    note *IH2* = ‹*fv e2* ⊆ *dom* Γ ⟹ ∃ *T*. Γ ⊢ *e2* : *T*›
    from ‹*fv* (*e1* «*bop*» *e2*) ⊆ *dom* Γ›
    have *fv e1* ⊆ *dom* Γ **and** *fv e2* ⊆ *dom* Γ **by** *auto*
    from *IH1*[*OF* ‹*fv e1* ⊆ *dom* Γ›] **obtain** *T1* **where** Γ ⊢ *e1* : *T1* **by** *auto*
    from *IH2*[*OF* ‹*fv e2* ⊆ *dom* Γ›] **obtain** *T2* **where** Γ ⊢ *e2* : *T2* **by** *auto*
    show *?case*
    proof (*cases T1*)
      case *Low*
      show *?thesis*
      proof (*cases T2*)
        case *Low*
        with ‹Γ ⊢ *e1* : *T1*› ‹Γ ⊢ *e2* : *T2*› ‹*T1* = *Low*›
        have Γ ⊢ *e1* «*bop*» *e2* : *Low* **by**(*simp add*:*typeBinOp1*)
        thus *?thesis* **by**(*rule exI*)
      next
        case *High*
        with ‹Γ ⊢ *e1* : *T1*› ‹Γ ⊢ *e2* : *T2*› ‹*T1* = *Low*›
        have Γ ⊢ *e1* «*bop*» *e2* : *High* **by**(*simp add*:*typeBinOp3*)
        thus *?thesis* **by**(*rule exI*)
      qed
    next
      case *High*
      with ‹Γ ⊢ *e1* : *T1*› ‹Γ ⊢ *e2* : *T2*›
      have Γ ⊢ *e1* «*bop*» *e2* : *High* **by** (*simp add*: *typeBinOp2*)
      thus *?thesis* **by** (*rule exI*)

```
    qed
  qed
  with that show ?thesis by blast
qed


lemma exprBinopTypeable:
  assumes Γ ⊢ e1 «bop» e2 : T
  shows (∃ T1. Γ ⊢ e1 : T1) ∧ (∃ T2. Γ ⊢ e2 : T2)
using assms by(auto elim:secExprTyping.cases)



lemma exprTypingHigh:
  assumes Γ ⊢ e : T and x ∈ fv e and Γ x = Some High
  shows Γ ⊢ e : High
using assms
proof (induct e arbitrary:T)
  case (Val V) show ?case by (rule typeVal)
next
  case (Var V)
  from ‹x ∈ fv (Var V)› have x = V by simp
  with ‹Γ x = Some High› show ?case by(simp add:typeVar)
next
  case (BinOp e1 bop e2)
  note IH1 = ‹⋀T. ⟦Γ ⊢ e1 : T; x ∈ fv e1; Γ x = Some High⟧ ⟹ Γ ⊢ e1 : High›
  note IH2 = ‹⋀T. ⟦Γ ⊢ e2 : T; x ∈ fv e2; Γ x = Some High⟧ ⟹ Γ ⊢ e2 : High›
  from ‹Γ ⊢ e1 «bop» e2 : T›
 have T:(∃ T1. Γ ⊢ e1 : T1) ∧ (∃ T2. Γ ⊢ e2 : T2) by (auto intro!:exprBinopTypeable)
  then obtain T1 where Γ ⊢ e1 : T1 by auto
  from T obtain T2 where Γ ⊢ e2 : T2 by auto
  from ‹x ∈ fv (e1 «bop» e2)› have x ∈ (fv e1 ∪ fv e2) by simp
  hence x ∈ fv e1 ∨ x ∈ fv e2 by auto
  thus ?case
  proof
    assume x ∈ fv e1
    from IH1[OF ‹Γ ⊢ e1 : T1› this ‹Γ x = Some High›] have Γ ⊢ e1 : High .
    with ‹Γ ⊢ e2 : T2› show ?thesis by(simp add:typeBinOp2)
  next
    assume x ∈ fv e2
    from IH2[OF ‹Γ ⊢ e2 : T2› this ‹Γ x = Some High›] have Γ ⊢ e2 : High .
    with ‹Γ ⊢ e1 : T1› show ?thesis by(simp add:typeBinOp3)
  qed
qed


lemma exprTypingLow:
  assumes Γ ⊢ e : Low and x ∈ fv e shows Γ x = Some Low
using assms
```

```
proof (induct e)
  case (Val V)
  have fv (Val V) = {} by (rule FVc)
  with ‹x ∈ fv (Val V)› have False by auto
  thus ?thesis by simp
next
  case (Var V)
  from ‹x ∈ fv (Var V)› have xV: x = V by simp
 from ‹Γ ⊢ Var V : Low› have Γ V = Some Low by (auto elim:secExprTyping.cases)
  with xV show ?thesis by simp
next
  case (BinOp e1 bop e2)
  note IH1 = ‹⟦Γ ⊢ e1 : Low; x ∈ fv e1⟧ ⟹ Γ x = Some Low›
  note IH2=‹⟦Γ ⊢ e2 : Low; x ∈ fv e2⟧ ⟹ Γ x = Some Low›
  from ‹Γ ⊢ e1 «bop» e2 : Low› have Γ ⊢ e1 : Low and Γ ⊢ e2 : Low
    by(auto elim:secExprTyping.cases)
  from ‹x ∈ fv (e1 «bop» e2)› have x ∈ fv e1 ∪ fv e2 by (simp add:FVe)
  hence x ∈ fv e1 ∨ x ∈ fv e2 by auto
  thus ?case
  proof
    assume x ∈ fv e1
    with IH1[OF ‹Γ ⊢ e1 : Low›] show ?thesis by auto
  next
    assume x ∈ fv e2
    with IH2[OF ‹Γ ⊢ e2 : Low›] show ?thesis by auto
  qed
qed


lemma typeableFreevars:
  assumes Γ ⊢ e : T shows fv e ⊆ dom Γ
using assms
proof(induct e arbitrary:T)
  case (Val V)
  have fv (Val V) = {} by (rule FVc)
  thus ?case by simp
next
  case (Var V)
  show ?case
  proof
    fix x assume x ∈ fv (Var V)
    hence x = V by simp
   from ‹Γ ⊢ Var V : T› have Γ V = Some T by (auto elim:secExprTyping.cases)
    with ‹x = V› show x ∈ dom Γ by auto
  qed
next
  case (BinOp e1 bop e2)
  note IH1 = ‹⋀T. Γ ⊢ e1 : T ⟹ fv e1 ⊆ dom Γ›
  note IH2 = ‹⋀T. Γ ⊢ e2 : T ⟹ fv e2 ⊆ dom Γ›
```

```
  show ?case
  proof
    fix x assume x ∈ fv (e1 «bop» e2)
    from ‹Γ ⊢ e1 «bop» e2 : T›
    have Q:(∃ T1. Γ ⊢ e1 : T1) ∧ (∃ T2. Γ ⊢ e2 : T2)
      by(rule exprBinopTypeable)
    then obtain T1 where Γ ⊢ e1 : T1 by blast
    from Q obtain T2 where Γ ⊢ e2 : T2 by blast
    from IH1[OF ‹Γ ⊢ e1 : T1›] have fv e1 ⊆ dom Γ .
    moreover
    from IH2[OF ‹Γ ⊢ e2 : T2›] have fv e2 ⊆ dom Γ .
    ultimately have (fv e1) ∪ (fv e2) ⊆ dom Γ by auto
    hence fv (e1 «bop» e2) ⊆ dom Γ by(simp add:FVe)
    with ‹x ∈ fv (e1 «bop» e2)› show x ∈ dom Γ by auto
  qed
qed


lemma exprNotNone:
assumes Γ ⊢ e : T and fv e ⊆ dom s
shows ⟦e⟧ s ≠ None
using assms
proof (induct e arbitrary: Γ T s)
  case (Val v)
  show ?case by(simp add: Val)
next
  case (Var V)
  have ⟦Var V⟧ s = s V by (simp add: Var)
  have V ∈ fv (Var V) by (auto simp add:FVv)
  with ‹fv (Var V) ⊆ dom s› have V ∈ dom s by simp
  thus ?case by auto
next
  case (BinOp e1 bop e2)
  note IH1 = ‹⋀T. ⟦Γ ⊢ e1 : T; fv e1 ⊆ dom s ⟧ ⟹ ⟦e1⟧ s ≠ None›
  note IH2 = ‹⋀T. ⟦Γ ⊢ e2 : T; fv e2 ⊆ dom s ⟧ ⟹ ⟦e2⟧ s ≠ None›
  from ‹Γ ⊢ e1 «bop» e2 : T› have (∃ T1. Γ ⊢ e1 : T1) ∧ (∃ T2. Γ ⊢ e2 : T2)
    by(rule exprBinopTypeable)
  then obtain T1 T2 where Γ ⊢ e1 : T1 and Γ ⊢ e2 : T2 by blast
  from ‹fv (e1 «bop» e2) ⊆ dom s› have fv e1 ∪ fv e2 ⊆ dom s by(simp add:FVe)
  hence fv e1 ⊆ dom s and fv e2 ⊆ dom s by auto
  from IH1[OF ‹Γ ⊢ e1 : T1› ‹fv e1 ⊆ dom s›] have ⟦e1⟧s ≠ None .
  moreover from IH2[OF ‹Γ ⊢ e2 : T2› ‹fv e2 ⊆ dom s›] have ⟦e2⟧s ≠ None .
  ultimately show ?case
    apply(cases bop) apply auto
    apply(case-tac y,auto,case-tac ya,auto)+
    done
qed
```

2.3 Noninterference definitions

2.3.1 Low Equivalence

Low Equivalence is reflexive even if the involved states are undefined. But in non-reflexive situations low variables must be initialized (i.e. $\in$ *dom state*), otherwise the proof will not work. This is not a restriction, but a natural requirement, and could be formalized as part of a standard type system.

Low equivalence is also symmetric and transitiv (see lemmas) hence an equivalence relation.

definition $lowEquiv :: typeEnv \Rightarrow state \Rightarrow state \Rightarrow bool$ ($\langle$- $\vdash$ - $\approx_L$ -$\rangle$)
where $\Gamma \vdash s1 \approx_L s2 \equiv \forall v{\in}dom\ \Gamma.\ \Gamma\ v = Some\ Low \longrightarrow (s1\ v = s2\ v)$

lemma *lowEquivReflexive*: $\Gamma \vdash s1 \approx_L s1$
by(*simp add*:*lowEquiv-def*)

lemma *lowEquivSymmetric*:
$\Gamma \vdash s1 \approx_L s2 \Longrightarrow \Gamma \vdash s2 \approx_L s1$
by(*simp add*:*lowEquiv-def*)

lemma *lowEquivTransitive*:
$[\![\Gamma \vdash s1 \approx_L s2;\ \Gamma \vdash s2 \approx_L s3]\!] \Longrightarrow \Gamma \vdash s1 \approx_L s3$
by(*simp add*:*lowEquiv-def*)

2.3.2 Non Interference

definition $nonInterference :: typeEnv \Rightarrow com \Rightarrow bool$
where $nonInterference\ \Gamma\ c \equiv$
$(\forall s1\ s2\ s1'\ s2'.\ (\Gamma \vdash s1 \approx_L s2 \wedge \langle c,s1\rangle \rightarrow* \langle Skip,s1'\rangle \wedge \langle c,s2\rangle \rightarrow* \langle Skip,s2'\rangle)$
$\longrightarrow \Gamma \vdash s1' \approx_L s2')$

lemma *nonInterferenceI*:
$[\![\bigwedge s1\ s2\ s1'\ s2'.\ [\![\Gamma \vdash s1 \approx_L s2;\ \langle c,s1\rangle \rightarrow* \langle Skip,s1'\rangle;\ \langle c,s2\rangle \rightarrow* \langle Skip,s2'\rangle]\!]$
$\Longrightarrow \Gamma \vdash s1' \approx_L s2']\!] \Longrightarrow nonInterference\ \Gamma\ c$
by(*auto simp*:*nonInterference-def*)

lemma *interpretLow*:
assumes $\Gamma \vdash s1 \approx_L s2$ **and** *all*:$\forall V{\in}fv\ e.\ \Gamma\ V = Some\ Low$
shows $[\![e]\!]\ s1 = [\![e]\!]\ s2$
using *all*
proof (*induct e*)
case (*Val v*)
show *?case* **by** (*simp add*: *Val*)
next
case (*Var V*)
have $[\![Var\ V]\!]\ s1 = s1\ V$ **and** $[\![Var\ V]\!]\ s2 = s2\ V$ **by**(*auto simp*:*Var*)
have $V \in fv\ (Var\ V)$ **by**(*simp add*:*FVv*)

from ‹$V \in fv$ (*Var V*)› ‹$\forall X \in fv$ (*Var V*). Γ X = *Some Low*› **have** Γ V = *Some Low* **by** *simp*
with *assms* **have** *s1 V* = *s2 V* **by**(*auto simp add*:*lowEquiv-def*)
thus *?case* **by** *auto*
next
case (*BinOp e1 bop e2*)
note *IH1* = ‹$\forall V \in fv$ *e1*. Γ V = *Some Low* $\Longrightarrow$ ⟦*e1*⟧*s1* = ⟦*e1*⟧*s2*›
note *IH2* = ‹$\forall V \in fv$ *e2*. Γ V = *Some Low* $\Longrightarrow$ ⟦*e2*⟧*s1* = ⟦*e2*⟧*s2*›
from ‹$\forall V \in fv$ (*e1* «*bop*» *e2*). Γ V = *Some Low*› **have** $\forall V \in fv$ *e1*. Γ V = *Some Low*
and $\forall V \in fv$ *e2*. Γ V = *Some Low* **by** *auto*
from *IH1*[*OF* ‹$\forall V \in fv$ *e1*. Γ V = *Some Low*›] **have** ⟦*e1*⟧ *s1* = ⟦*e1*⟧ *s2* **.**
moreover
from *IH2*[*OF* ‹$\forall V \in fv$ *e2*. Γ V = *Some Low*›] **have** ⟦*e2*⟧ *s1* = ⟦*e2*⟧ *s2* **.**
ultimately show *?case* **by**(*cases* ⟦*e1*⟧ *s2*,*auto*)
qed

lemma *interpretLow2*:
assumes $\Gamma \vdash e : Low$ **and** $\Gamma \vdash s1 \approx_L s2$ **shows** ⟦*e*⟧ *s1* = ⟦*e*⟧ *s2*
proof −
from ‹$\Gamma \vdash e : Low$› **have** *fv e* $\subseteq$ *dom* Γ **by**(*auto dest*:*typeableFreevars*)
have $\forall x \in fv$ *e*. Γ x = *Some Low*
proof
fix x **assume** $x \in fv$ *e*
with ‹$\Gamma \vdash e : Low$› **show** Γ x = *Some Low* **by**(*auto intro*:*exprTypingLow*)
qed
with ‹$\Gamma \vdash s1 \approx_L s2$› **show** *?thesis* **by**(*rule interpretLow*)
qed

lemma *assignNIhighlemma*:
assumes $\Gamma \vdash s1 \approx_L s2$ **and** Γ V = *Some High* **and** *s1'* = *s1*(*V*:= ⟦*e*⟧ *s1*)
and *s2'* = *s2*(*V*:= ⟦*e*⟧ *s2*)
shows $\Gamma \vdash s1' \approx_L s2'$
proof −
{ fix V' **assume** $V' \in dom$ Γ **and** Γ V' = *Some Low*
from ‹$\Gamma \vdash s1 \approx_L s2$› ‹$\Gamma$ V' = *Some Low*› **have** *s1 V'* = *s2 V'*
by(*auto simp add*:*lowEquiv-def*)
have *s1' V'* = *s2' V'*
proof(*cases* V' = V)
case *True*
with ‹Γ V' = *Some Low*› ‹Γ V = *Some High*› **have** *False* **by** *simp*
thus *?thesis* **by** *simp*
next
case *False*
with ‹*s1'* = *s1*(*V*:= ⟦*e*⟧ *s1*)› ‹*s2'* = *s2*(*V*:= ⟦*e*⟧ *s2*)›
have *s1 V'* = *s1' V'* **and** *s2 V'* = *s2' V'* **by** *auto*
with ‹*s1 V'* = *s2 V'*› **show** *?thesis* **by** *simp*

qed
}
thus *?thesis* **by**(*auto simp add*:*lowEquiv-def*)
qed

lemma *assignNIlowlemma*:
assumes $\Gamma \vdash s1 \approx_L s2$ **and** $\Gamma\ V = \textit{Some Low}$ **and** $\Gamma \vdash e : \textit{Low}$
and $s1' = s1(V{:=}\ [\![e]\!]\ s1)$ **and** $s2' = s2(V{:=}\ [\![e]\!]\ s2)$
shows $\Gamma \vdash s1' \approx_L s2'$
proof −
{ **fix** V' **assume** $V' \in \textit{dom}\ \Gamma$ **and** $\Gamma\ V' = \textit{Some Low}$
from ‹$\Gamma \vdash s1 \approx_L s2$› ‹$\Gamma\ V' = \textit{Some Low}$›
have $s1\ V' = s2\ V'$ **by**(*auto simp add*:*lowEquiv-def*)
have $s1'\ V' = s2'\ V'$
proof(*cases* $V' = V$)
case *True*
with ‹$s1' = s1(V{:=}\ [\![e]\!]\ s1)$› ‹$s2' = s2(V{:=}\ [\![e]\!]\ s2)$›
have $s1'\ V' = [\![e]\!]\ s1$ **and** $s2'\ V' = [\![e]\!]\ s2$ **by** *auto*
from ‹$\Gamma \vdash e : \textit{Low}$› ‹$\Gamma \vdash s1 \approx_L s2$› **have** $[\![e]\!]\ s1 = [\![e]\!]\ s2$
by(*auto intro*:*interpretLow2*)
with ‹$s1'\ V' = [\![e]\!]\ s1$› ‹$s2'\ V' = [\![e]\!]\ s2$› **show** *?thesis* **by** *simp*
next
case *False*
with ‹$s1' = s1(V{:=}\ [\![e]\!]\ s1)$› ‹$s2' = s2(V{:=}\ [\![e]\!]\ s2)$›
have $s1'\ V' = s1\ V'$ **and** $s2'\ V' = s2\ V'$ **by** *auto*
with ‹$s1\ V' = s2\ V'$› **have** $s1'\ V' = s2'\ V'$ **by** *simp*
with *False* ‹$s1'\ V' = s1\ V'$› ‹$s2'\ V' = s2\ V'$›
show *?thesis* **by** *auto*
qed
}
thus *?thesis* **by**(*simp add*:*lowEquiv-def*)
qed

Sequential Compositionality is given the status of a theorem because compositionality is no longer valid in case of concurrency

theorem *SeqCompositionality*:
assumes *nonInterference* Γ *c1* **and** *nonInterference* Γ *c2*
shows *nonInterference* Γ $(c1;;c2)$
proof(*rule nonInterferenceI*)
fix $s1\ s2\ s1'\ s2'$
assume $\Gamma \vdash s1 \approx_L s2$ **and** $\langle c1;;c2,s1\rangle \rightarrow\!* \langle \textit{Skip},s1'\rangle$
and $\langle c1;;c2,s2\rangle \rightarrow\!* \langle \textit{Skip},s2'\rangle$
from ‹$\langle c1;;c2,s1\rangle \rightarrow\!* \langle \textit{Skip},s1'\rangle$› **obtain** $s1''$ **where** $\langle c1,s1\rangle \rightarrow\!* \langle \textit{Skip},s1''\rangle$
and $\langle c2,s1''\rangle \rightarrow\!* \langle \textit{Skip},s1'\rangle$ **by**(*auto dest*:*Seq-reds*)
from ‹$\langle c1;;c2,s2\rangle \rightarrow\!* \langle \textit{Skip},s2'\rangle$› **obtain** $s2''$ **where** $\langle c1,s2\rangle \rightarrow\!* \langle \textit{Skip},s2''\rangle$
and $\langle c2,s2''\rangle \rightarrow\!* \langle \textit{Skip},s2'\rangle$ **by**(*auto dest*:*Seq-reds*)
from ‹$\Gamma \vdash s1 \approx_L s2$› ‹$\langle c1,s1\rangle \rightarrow\!* \langle \textit{Skip},s1''\rangle$› ‹$\langle c1,s2\rangle \rightarrow\!* \langle \textit{Skip},s2''\rangle$›

⟨*nonInterference* Γ *c1*⟩
have $\Gamma \vdash s1'' \approx_L s2''$ **by**(*auto simp*:*nonInterference-def*)
with ⟨⟨*c2*,*s1''*⟩ →∗ ⟨*Skip*,*s1'*⟩⟩ ⟨⟨*c2*,*s2''*⟩ →∗ ⟨*Skip*,*s2'*⟩⟩ ⟨*nonInterference* Γ *c2*⟩

show $\Gamma \vdash s1' \approx_L s2'$ **by**(*auto simp*:*nonInterference-def*)
qed

lemma *WhileStepInduct*:
assumes *while*:⟨*while* (*b*) *c*,*s1*⟩ →∗ ⟨*Skip*,*s2*⟩
and *body*:$\bigwedge$*s1 s2*. ⟨*c*,*s1*⟩ →∗ ⟨*Skip*,*s2*⟩ ⟹ $\Gamma \vdash s1 \approx_L s2$ **and** Γ,*High* ⊢ *c*
shows $\Gamma \vdash s1 \approx_L s2$
using *while*
proof (*induct rule*:*while-reds-induct*)
case (*false s*) **thus** *?case* **by**(*auto simp add*:*lowEquiv-def*)
next
case (*true s1 s3*)
from *body*[*OF* ⟨⟨*c*,*s1*⟩ →∗ ⟨*Skip*,*s3*⟩⟩] **have** $\Gamma \vdash s1 \approx_L s3$ **by** *simp*
with ⟨$\Gamma \vdash s3 \approx_L s2$⟩ **show** *?case* **by**(*auto intro*:*lowEquivTransitive*)
qed

In case control conditions from if/while are high, the body of an if/while must not change low variables in order to prevent implicit flow. That is, start and end state of any if/while body must be low equivalent.

theorem *highBodies*:
assumes Γ,*High* ⊢ *c* **and** ⟨*c*,*s1*⟩ →∗ ⟨*Skip*,*s2*⟩
shows $\Gamma \vdash s1 \approx_L s2$
— all intermediate states must be well formed, otherwise the proof does not work for uninitialized variables. Thus it is propagated through the theorem conclusion
using *assms*
proof(*induct c arbitrary*:*s1 s2 rule*:*com.induct*)
case *Skip*
from ⟨⟨*Skip*,*s1*⟩ →∗ ⟨*Skip*,*s2*⟩⟩ **have** *s1* = *s2* **by** (*auto dest*:*Skip-reds*)
thus *?case* **by**(*simp add*:*lowEquiv-def*)
next
case (*LAss V e*)
from ⟨Γ,*High* ⊢ *V*:=*e*⟩ **have** Γ *V* = *Some High* **by**(*auto elim*:*secComTyping.cases*)
from ⟨⟨*V*:=*e*,*s1*⟩ →∗ ⟨*Skip*,*s2*⟩⟩ **have** *s2* = *s1*(*V*:= ⟦*e*⟧*s1*) **by** (*auto intro*:*LAss-reds*)
{ **fix** *V'* **assume** *V'* ∈ *dom* Γ **and** Γ *V'* = *Some Low*
have *s1 V'* = *s2 V'*
proof(*cases V'* = *V*)
case *True*
with ⟨Γ *V'* = *Some Low*⟩ ⟨Γ *V* = *Some High*⟩ **have** *False* **by** *simp*
thus *?thesis* **by** *simp*
next
case *False*
with ⟨*s2* = *s1*(*V*:= ⟦*e*⟧*s1*)⟩ **show** *?thesis* **by** *simp*
qed

}
thus *?case* **by**(*auto simp add*:*lowEquiv-def*)
next
case (*Seq c1 c2*)
note $IH1 = ‹\bigwedge s1\ s2.\ [\![\Gamma,High \vdash c1;\ \langle c1,s1\rangle \rightarrow* \langle Skip,s2\rangle]\!] \Longrightarrow \Gamma \vdash s1 \approx_L s2›$
note $IH2 = ‹\bigwedge s1\ s2.\ [\![\Gamma,High \vdash c2;\ \langle c2,s1\rangle \rightarrow* \langle Skip,s2\rangle]\!] \Longrightarrow \Gamma \vdash s1 \approx_L s2›$
from $‹\Gamma,High \vdash c1;;c2›$ **have** $\Gamma,High \vdash c1$ **and** $\Gamma,High \vdash c2$
by(*auto elim*:*secComTyping.cases*)
from $‹\langle c1;;c2,s1\rangle \rightarrow* \langle Skip,s2\rangle›$
have $\exists s3.\ \langle c1,s1\rangle \rightarrow* \langle Skip,s3\rangle \wedge \langle c2,s3\rangle \rightarrow* \langle Skip,s2\rangle$ **by**(*auto intro*:*Seq-reds*)
then obtain *s3* **where** $\langle c1,s1\rangle \rightarrow* \langle Skip,s3\rangle$ **and** $\langle c2,s3\rangle \rightarrow* \langle Skip,s2\rangle$ **by** *auto*
from $IH1[OF\ ‹\Gamma,High \vdash c1›\ ‹\langle c1,s1\rangle \rightarrow* \langle Skip,s3\rangle›]$
have $\Gamma \vdash s1 \approx_L s3$ **by** *simp*
from $IH2[OF\ ‹\Gamma,High \vdash c2›\ ‹\langle c2,s3\rangle \rightarrow* \langle Skip,s2\rangle›]$
have $\Gamma \vdash s3 \approx_L s2$ **by** *simp*
from $‹\Gamma \vdash s1 \approx_L s3›\ ‹\Gamma \vdash s3 \approx_L s2›$ **show** *?case*
by(*auto intro*:*lowEquivTransitive*)
next
case (*Cond b c1 c2*)
note $IH1 = ‹\bigwedge s1\ s2.\ [\![\Gamma,High \vdash c1;\ \langle c1,s1\rangle \rightarrow* \langle Skip,s2\rangle]\!] \Longrightarrow \Gamma \vdash s1 \approx_L s2›$
note $IH2 = ‹\bigwedge s1\ s2.\ [\![\Gamma,High \vdash c2;\ \langle c2,s1\rangle \rightarrow* \langle Skip,s2\rangle]\!] \Longrightarrow \Gamma \vdash s1 \approx_L s2›$
from $‹\Gamma,High \vdash if\ (b)\ c1\ else\ c2›$ **have** $\Gamma,High \vdash c1$ **and** $\Gamma,High \vdash c2$
by(*auto elim*:*secComTyping.cases*)
from $‹\langle if\ (b)\ c1\ else\ c2,s1\rangle \rightarrow* \langle Skip,s2\rangle›$
have $[\![b]\!]\ s1 = Some\ true \vee [\![b]\!]\ s1 = Some\ false$ **by**(*auto dest*:*Cond-True-or-False*)
thus *?case*
proof
assume $[\![b]\!]\ s1 = Some\ true$
with $‹\langle if\ (b)\ c1\ else\ c2,s1\rangle \rightarrow* \langle Skip,s2\rangle›$ **have** $\langle c1,s1\rangle \rightarrow* \langle Skip,s2\rangle$
by (*auto intro*:*CondTrue-reds*)
from $IH1[OF\ ‹\Gamma,High \vdash c1›\ this]$ **show** *?thesis* **.**
next
assume $[\![b]\!]\ s1 = Some\ false$
with $‹\langle if\ (b)\ c1\ else\ c2,s1\rangle \rightarrow* \langle Skip,s2\rangle›$ **have** $\langle c2,s1\rangle \rightarrow* \langle Skip,s2\rangle$
by(*auto intro*:*CondFalse-reds*)
from $IH2[OF\ ‹\Gamma,High \vdash c2›\ this]$ **show** *?thesis* **.**
qed
next
case (*While b c'*)
note $IH = ‹\bigwedge s1\ s2.\ [\![\Gamma,High \vdash c';\ \langle c',s1\rangle \rightarrow* \langle Skip,s2\rangle]\!] \Longrightarrow \Gamma \vdash s1 \approx_L s2›$
from $‹\Gamma,High \vdash while\ (b)\ c'›$ **have** $\Gamma,High \vdash c'$ **by**(*auto elim*:*secComTyping.cases*)
from *IH*[*OF this*]
have $\bigwedge s1\ s2.\ [\![\langle c',s1\rangle \rightarrow* \langle Skip,s2\rangle]\!] \Longrightarrow \Gamma \vdash s1 \approx_L s2$ **.**
with $‹\langle while\ (b)\ c',s1\rangle \rightarrow* \langle Skip,s2\rangle›\ ‹\Gamma,High \vdash c'›$
show *?case* **by**(*auto dest*:*WhileStepInduct*)
qed

```
lemma CondHighCompositionality:
  assumes Γ,High ⊢ if (b) c1 else c2
  shows nonInterference Γ (if (b) c1 else c2)
proof(rule nonInterferenceI)
  fix s1 s2 s1' s2'
  assume Γ ⊢ s1 ≈L s2 and ⟨if (b) c1 else c2,s1⟩ →* ⟨Skip,s1'⟩
    and ⟨if (b) c1 else c2,s2⟩ →* ⟨Skip,s2'⟩
  show Γ ⊢ s1' ≈L s2'
  proof -
    from ‹Γ,High ⊢ if (b) c1 else c2› ‹⟨if (b) c1 else c2,s1⟩ →* ⟨Skip,s1'⟩›
    have Γ ⊢ s1 ≈L s1' by(auto dest:highBodies)
    from ‹Γ,High ⊢ if (b) c1 else c2› ‹⟨if (b) c1 else c2,s2⟩ →* ⟨Skip,s2'⟩›
    have Γ ⊢ s2 ≈L s2' by(auto dest:highBodies)
    with ‹Γ ⊢ s1 ≈L s2› have Γ ⊢ s1 ≈L s2' by(auto intro:lowEquivTransitive)
    from ‹Γ ⊢ s1 ≈L s1'› have Γ ⊢ s1' ≈L s1 by(auto intro:lowEquivSymmetric)
    with ‹Γ ⊢ s1 ≈L s2'› show ?thesis by(auto intro:lowEquivTransitive)
  qed
qed
```

```
lemma CondLowCompositionality:
  assumes nonInterference Γ c1 and nonInterference Γ c2 and Γ ⊢ b : Low
  shows nonInterference Γ (if (b) c1 else c2)
proof(rule nonInterferenceI)
  fix s1 s2 s1' s2'
  assume Γ ⊢ s1 ≈L s2 and ⟨if (b) c1 else c2,s1⟩ →* ⟨Skip,s1'⟩
    and ⟨if (b) c1 else c2,s2⟩ →* ⟨Skip,s2'⟩
  from ‹Γ ⊢ b : Low› ‹Γ ⊢ s1 ≈L s2› have ⟦b⟧ s1 = ⟦b⟧ s2
    by(auto intro:interpretLow2)
  from ‹⟨if (b) c1 else c2,s1⟩ →* ⟨Skip,s1'⟩›
  have ⟦b⟧ s1 = Some true ∨ ⟦b⟧ s1 = Some false by(auto dest:Cond-True-or-False)
  thus Γ ⊢ s1' ≈L s2'
  proof
    assume ⟦b⟧ s1 = Some true
    with ‹⟦b⟧ s1 = ⟦b⟧ s2› have ⟦b⟧ s2 = Some true by(auto intro:CondTrue-reds)
    from ‹⟦b⟧ s1 = Some true› ‹⟨if (b) c1 else c2,s1⟩ →* ⟨Skip,s1'⟩›
    have ⟨c1,s1⟩ →* ⟨Skip,s1'⟩ by(auto intro:CondTrue-reds)
    from ‹⟦b⟧ s2 = Some true› ‹⟨if (b) c1 else c2,s2⟩ →* ⟨Skip,s2'⟩›
    have ⟨c1,s2⟩ →* ⟨Skip,s2'⟩ by(auto intro:CondTrue-reds)
    with ‹⟨c1,s1⟩ →* ⟨Skip,s1'⟩› ‹Γ ⊢ s1 ≈L s2› ‹nonInterference Γ c1›
    show ?thesis by(auto simp:nonInterference-def)
  next
    assume ⟦b⟧ s1 = Some false
    with ‹⟦b⟧ s1 = ⟦b⟧ s2› have ⟦b⟧ s2 = Some false by(auto intro:CondTrue-reds)
    from ‹⟦b⟧ s1 = Some false› ‹⟨if (b) c1 else c2,s1⟩ →* ⟨Skip,s1'⟩›
    have ⟨c2,s1⟩ →* ⟨Skip,s1'⟩ by(auto intro:CondFalse-reds)
    from ‹⟦b⟧ s2 = Some false› ‹⟨if (b) c1 else c2,s2⟩ →* ⟨Skip,s2'⟩›
    have ⟨c2,s2⟩ →* ⟨Skip,s2'⟩ by(auto intro:CondFalse-reds)
```

with ‹$\langle c2,s1\rangle \rightarrow* \langle Skip,s1'\rangle$› ‹$\Gamma \vdash s1 \approx_L s2$› ‹*nonInterference* Γ *c2*›
show *?thesis* **by**(*auto simp*:*nonInterference-def*)
qed
qed

lemma *WhileHighCompositionality*:
assumes Γ,*High* $\vdash$ *while* (*b*) *c'*
shows *nonInterference* Γ (*while* (*b*) *c'*)
proof(*rule nonInterferenceI*)
fix *s1 s2 s1' s2'*
assume $\Gamma \vdash s1 \approx_L s2$ **and** $\langle$*while* (*b*) *c'*,*s1*$\rangle \rightarrow* \langle Skip,s1'\rangle$
and $\langle$*while* (*b*) *c'*,*s2*$\rangle \rightarrow* \langle Skip,s2'\rangle$
show $\Gamma \vdash s1' \approx_L s2'$
proof −
from ‹Γ,*High* $\vdash$ *while* (*b*) *c'*› ‹$\langle$*while* (*b*) *c'*,*s1*$\rangle \rightarrow* \langle Skip,s1'\rangle$›
have $\Gamma \vdash s1 \approx_L s1'$ **by**(*auto dest*:*highBodies*)
from ‹Γ,*High* $\vdash$ *while* (*b*) *c'*› ‹$\langle$*while* (*b*) *c'*,*s2*$\rangle \rightarrow* \langle Skip,s2'\rangle$›
have $\Gamma \vdash s2 \approx_L s2'$ **by**(*auto dest*:*highBodies*)
with ‹$\Gamma \vdash s1 \approx_L s2$› **have** $\Gamma \vdash s1 \approx_L s2'$ **by**(*auto intro*:*lowEquivTransitive*)
from ‹$\Gamma \vdash s1 \approx_L s1'$› **have** $\Gamma \vdash s1' \approx_L s1$ **by**(*auto intro*:*lowEquivSymmetric*)
with ‹$\Gamma \vdash s1 \approx_L s2'$› **show** *?thesis* **by**(*auto intro*:*lowEquivTransitive*)
qed
qed

lemma *WhileLowStepInduct*:
assumes *while1*: $\langle$*while* (*b*) *c'*,*s1*$\rangle \rightarrow* \langle Skip,s1'\rangle$
and *while2*: $\langle$*while* (*b*) *c'*,*s2*$\rangle\rightarrow*\langle Skip,s2'\rangle$
and $\Gamma \vdash b : Low$
and *body*:$\bigwedge$*s1 s1' s2 s2'*. $[\![\langle c',s1\rangle \rightarrow* \langle Skip,s1'\rangle; \langle c',s2\rangle \rightarrow* \langle Skip,s2'\rangle;$
$\Gamma \vdash s1 \approx_L s2]\!] \Longrightarrow \Gamma \vdash s1' \approx_L s2'$
and *le*: $\Gamma \vdash s1 \approx_L s2$
shows $\Gamma \vdash s1' \approx_L s2'$
using *while1 le while2*
proof (*induct arbitrary*:*s2 rule*:*while-reds-induct*)
case (*false s1*)
from ‹$\Gamma \vdash b : Low$› ‹$\Gamma \vdash s1 \approx_L s2$› **have** $[\![b]\!]\ s1 = [\![b]\!]\ s2$ **by**(*auto intro*:*interpretLow2*)
with ‹$[\![b]\!]\ s1$ = *Some false*› **have** $[\![b]\!]\ s2$ = *Some false* **by** *simp*
with ‹$\langle$*while* (*b*) *c'*,*s2*$\rangle\rightarrow*\langle Skip,s2'\rangle$› **have** $s2 = s2'$ **by**(*auto intro*: *WhileFalse-reds*)
with ‹$\Gamma \vdash s1 \approx_L s2$› **show** *?case* **by** *auto*
next
case (*true s1 s1''*)
note *IH* = ‹$\bigwedge$*s2''*. $[\![\Gamma \vdash s1'' \approx_L s2''; \langle$*while* (*b*) *c'*,*s2''*$\rangle \rightarrow* \langle Skip,s2'\rangle]\!]$
$\Longrightarrow \Gamma \vdash s1' \approx_L s2'$›
from ‹$\Gamma \vdash b : Low$› ‹$\Gamma \vdash s1 \approx_L s2$› **have** $[\![b]\!]\ s1 = [\![b]\!]\ s2$ **by**(*auto intro*:*interpretLow2*)
with ‹$[\![b]\!]\ s1$ = *Some true*› **have** $[\![b]\!]\ s2$ = *Some true* **by** *simp*
with ‹$\langle$*while* (*b*) *c'*,*s2*$\rangle \rightarrow* \langle Skip,s2'\rangle$› **obtain** *s2''* **where** $\langle c',s2\rangle \rightarrow* \langle Skip,s2''\rangle$

and $\langle$*while* (b) c',*s2''*$\rangle \rightarrow\!*$ $\langle$*Skip*,*s2'*$\rangle$ **by**(*auto dest*:*WhileTrue-reds*)
from *body*[*OF* ‹$\langle c'$,*s1*$\rangle \rightarrow\!*$ $\langle$*Skip*,*s1''*$\rangle$› ‹$\langle c'$,*s2*$\rangle \rightarrow\!*$ $\langle$*Skip*,*s2''*$\rangle$› ‹$\Gamma \vdash$ *s1* $\approx_L$ *s2*›]
have $\Gamma \vdash$ *s1''* $\approx_L$ *s2''* **.**
from *IH*[*OF this* ‹$\langle$*while* (b) c',*s2''*$\rangle \rightarrow\!*$ $\langle$*Skip*,*s2'*$\rangle$›] **show** *?case* **.**
qed

lemma *WhileLowCompositionality*:
assumes *nonInterference* Γ c' **and** $\Gamma \vdash b$: *Low* **and** Γ,*Low* $\vdash c'$
shows *nonInterference* Γ (*while* (b) c')
proof(*rule nonInterferenceI*)
fix *s1 s2 s1' s2'*
assume $\Gamma \vdash$ *s1* $\approx_L$ *s2* **and** $\langle$*while* (b) c',*s1*$\rangle \rightarrow\!*$ $\langle$*Skip*,*s1'*$\rangle$
and $\langle$*while* (b) c',*s2*$\rangle \rightarrow\!*$ $\langle$*Skip*,*s2'*$\rangle$
{ fix *s1 s2 s1'' s2''*
assume $\langle c'$,*s1*$\rangle \rightarrow\!*$ $\langle$*Skip*,*s1''*$\rangle$ **and** $\langle c'$,*s2*$\rangle \rightarrow\!*$ $\langle$*Skip*,*s2''*$\rangle$
and $\Gamma \vdash$ *s1* $\approx_L$ *s2*
with ‹*nonInterference* Γ c'› **have** $\Gamma \vdash$ *s1''* $\approx_L$ *s2''*
by(*auto simp*:*nonInterference-def*)
}
hence $\bigwedge$*s1 s1'' s2 s2''*. $[\![\langle c'$,*s1*$\rangle \rightarrow\!*$ $\langle$*Skip*,*s1''*$\rangle$; $\langle c'$,*s2*$\rangle \rightarrow\!*$ $\langle$*Skip*,*s2''*$\rangle$;
$\Gamma \vdash$ *s1* $\approx_L$ *s2*$]\!]$ $\Longrightarrow$ $\Gamma \vdash$ *s1''* $\approx_L$ *s2''* **by** *auto*
with ‹$\langle$*while* (b) c',*s1*$\rangle \rightarrow\!*$ $\langle$*Skip*,*s1'*$\rangle$› ‹$\langle$*while* (b) c',*s2*$\rangle \rightarrow\!*$ $\langle$*Skip*,*s2'*$\rangle$›
‹$\Gamma \vdash b$: *Low*› ‹$\Gamma \vdash$ *s1* $\approx_L$ *s2*› **show** $\Gamma \vdash$ *s1'* $\approx_L$ *s2'*
by(*auto intro*:*WhileLowStepInduct*)
qed

and now: the main theorem:

theorem *secTypeImpliesNonInterference*:
$\Gamma,T \vdash c \Longrightarrow$ *nonInterference* Γ c
proof (*induct c arbitrary*:*T rule*:*com.induct*)
case *Skip*
show *?case*
proof(*rule nonInterferenceI*)
fix *s1 s2 s1' s2'*
assume $\Gamma \vdash$ *s1* $\approx_L$ *s2* **and** $\langle$*Skip*,*s1*$\rangle \rightarrow\!*$ $\langle$*Skip*,*s1'*$\rangle$ **and** $\langle$*Skip*,*s2*$\rangle \rightarrow\!*$ $\langle$*Skip*,*s2'*$\rangle$
from ‹$\langle$*Skip*,*s1*$\rangle \rightarrow\!*$ $\langle$*Skip*,*s1'*$\rangle$› **have** *s1* = *s1'* **by**(*auto dest*:*Skip-reds*)
from ‹$\langle$*Skip*,*s2*$\rangle \rightarrow\!*$ $\langle$*Skip*,*s2'*$\rangle$› **have** *s2* = *s2'* **by**(*auto dest*:*Skip-reds*)
from ‹$\Gamma \vdash$ *s1* $\approx_L$ *s2*› **and** ‹*s1* = *s1'*› **and** ‹*s2* = *s2'*›
show $\Gamma \vdash$ *s1'* $\approx_L$ *s2'* **by** *simp*
qed
next
case (*LAss V e*)
from ‹$\Gamma,T \vdash V$:=e›
have *varprem*:(Γ V = *Some High*) $\vee$ (Γ V = *Some Low* $\wedge$ $\Gamma \vdash e$: *Low* $\wedge$ T = *Low*)
by (*auto elim*:*secComTyping.cases*)
show *?case*

```
  proof(rule nonInterferenceI)
    fix s1 s2 s1' s2'
     assume Γ ⊢ s1 ≈L s2 and ⟨V:=e,s1⟩ →* ⟨Skip,s1'⟩ and ⟨V:=e,s2⟩ →*
⟨Skip,s2'⟩
      from ‹⟨V:=e,s1⟩ →* ⟨Skip,s1'⟩› have s1' = s1(V:=⟦e⟧ s1) by(auto in-
tro:LAss-reds)
      from ‹⟨V:=e,s2⟩ →* ⟨Skip,s2'⟩› have s2' = s2(V:=⟦e⟧ s2) by(auto in-
tro:LAss-reds)
    from varprem show Γ ⊢ s1' ≈L s2'
    proof
      assume Γ V = Some High
      with ‹Γ ⊢ s1 ≈L s2› ‹s1' = s1(V:=⟦e⟧ s1)› ‹s2' = s2(V:=⟦e⟧ s2)›
      show ?thesis by(auto intro:assignNIhighlemma)
    next
      assume Γ V = Some Low ∧ Γ ⊢ e : Low ∧ T = Low
      with ‹Γ ⊢ s1 ≈L s2› ‹s1' = s1(V:=⟦e⟧ s1)› ‹s2' = s2(V:=⟦e⟧ s2)›
      show ?thesis by(auto intro:assignNIlowlemma)
    qed
  qed
next
  case (Seq c1 c2)
  note IH1 = ‹⋀T. Γ,T ⊢ c1 ⟹ nonInterference Γ c1›
  note IH2 = ‹⋀T. Γ,T ⊢ c2 ⟹ nonInterference Γ c2›
  show ?case
  proof (cases T)
    case High
    with ‹Γ,T ⊢ c1;;c2› have Γ,High ⊢ c1 and Γ,High ⊢ c2
      by(auto elim:secComTyping.cases)
    from IH1[OF ‹Γ,High ⊢ c1›] have nonInterference Γ c1 .
    moreover
    from IH2[OF ‹Γ,High ⊢ c2›] have nonInterference Γ c2 .
    ultimately show ?thesis by (auto intro:SeqCompositionality)
  next
    case Low
    with ‹Γ,T ⊢ c1;;c2›
    have (Γ,Low ⊢ c1 ∧ Γ,Low ⊢ c2) ∨ Γ,High ⊢ c1;;c2
      by(auto elim:secComTyping.cases)
    thus ?thesis
    proof
      assume Γ,Low ⊢ c1 ∧ Γ,Low ⊢ c2
      hence Γ,Low ⊢ c1 and Γ,Low ⊢ c2 by simp-all
      from IH1[OF ‹Γ,Low ⊢ c1›] have nonInterference Γ c1 .
      moreover
      from IH2[OF ‹Γ,Low ⊢ c2›] have nonInterference Γ c2 .
      ultimately show ?thesis by(auto intro:SeqCompositionality)
    next
      assume Γ,High ⊢ c1;;c2
      hence Γ,High ⊢ c1 and Γ,High ⊢ c2 by(auto elim:secComTyping.cases)
      from IH1[OF ‹Γ,High ⊢ c1›] have nonInterference Γ c1 .
```

```
      moreover
      from IH2[OF ‹Γ,High ⊢ c2›] have nonInterference Γ c2 .
      ultimately show ?thesis by(auto intro:SeqCompositionality)
    qed
  qed
next
  case (Cond b c1 c2)
  note IH1 = ‹⋀T. Γ,T ⊢ c1 ⟹ nonInterference Γ c1›
  note IH2 = ‹⋀T. Γ,T ⊢ c2 ⟹ nonInterference Γ c2›
  show ?case
  proof (cases T)
    case High
    with ‹Γ,T ⊢ if (b) c1 else c2› show ?thesis
      by(auto intro:CondHighCompositionality)
  next
    case Low
    with ‹Γ,T ⊢ if (b) c1 else c2›
    have (Γ ⊢ b : Low ∧ Γ,Low ⊢ c1 ∧Γ,Low ⊢ c2) ∨ Γ,High ⊢ if (b) c1 else c2
      by(auto elim:secComTyping.cases)
    thus ?thesis
    proof
      assume Γ ⊢ b : Low ∧ Γ,Low ⊢ c1 ∧ Γ,Low ⊢ c2
      hence Γ ⊢ b : Low and Γ,Low ⊢ c1 and Γ,Low ⊢ c2 by simp-all
      from IH1[OF ‹Γ,Low ⊢ c1›] have nonInterference Γ c1 .
      moreover
      from IH2[OF ‹Γ,Low ⊢ c2›] have nonInterference Γ c2 .
      ultimately show ?thesis using ‹Γ ⊢ b : Low›
        by(auto intro:CondLowCompositionality)
    next
      assume Γ,High ⊢ if (b) c1 else c2
      thus ?thesis by(auto intro:CondHighCompositionality)
    qed
  qed
next
  case (While b c′)
  note IH = ‹⋀T. Γ,T ⊢ c′ ⟹ nonInterference Γ c′›
  show ?case
  proof(cases T)
    case High
   with ‹Γ,T ⊢ while (b) c′› show ?thesis by(auto intro:WhileHighCompositionality)
  next
    case Low
    with ‹Γ,T ⊢ while (b) c′›
    have (Γ ⊢ b : Low ∧ Γ,Low ⊢ c′) ∨ Γ,High ⊢ while (b) c′
      by(auto elim:secComTyping.cases)
    thus ?thesis
    proof
      assume Γ ⊢ b : Low ∧ Γ,Low ⊢ c′
      hence Γ ⊢ b : Low and Γ,Low ⊢ c′ by simp-all
```

```
    moreover
    from IH[OF ‹Γ,Low ⊢ c′›] have nonInterference Γ c′ .
    ultimately show ?thesis by(auto intro: WhileLowCompositionality)
  next
    assume Γ,High ⊢ while (b) c′
    thus ?thesis by(auto intro: WhileHighCompositionality)
  qed
 qed
qed

end


theory Execute
imports secTypes
begin
```

3 Executing the small step semantics

```
code-pred (modes: i => o => bool as exec-red, i => i ∗ o => bool, i => o ∗ i
=> bool, i => i => bool) red .
thm red.equation

definition [code]: one-step x = Predicate.the (exec-red x)

lemmas [code-pred-intro] = typeVal[where lev = Low] typeVal[where lev = High]
 typeVar
 typeBinOp1 typeBinOp2[where lev = Low] typeBinOp2[where lev = High] type-
BinOp3[where lev = Low]

code-pred (modes: i => i => o => bool as compute-secExprTyping,
 i => i => i => bool as check-secExprTyping) secExprTyping
proof −
 case secExprTyping
 from secExprTyping.prems show thesis
 proof
  fix Γ V lev assume x = Γ xa = Val V xb = lev
  from secExprTyping(1−2) this show thesis by (cases lev) auto
 next
  fix Γ Vn lev
  assume x = Γ xa = Var Vn xb = lev Γ Vn = Some lev
  from secExprTyping(3) this show thesis by (auto simp add: Predicate.eq-is-eq)
 next
  fix Γ e1 e2 bop
  assume x = Γ xa = e1«bop» e2 xb = Low
   Γ ⊢ e1 : Low Γ ⊢ e2 : Low
  from secExprTyping(4) this show thesis by auto
 next
  fix Γ e1 e2 lev bop
```

```
    assume x = Γ xa = e1«bop» e2 xb = High
    Γ ⊢ e1 : High Γ ⊢ e2 : lev
    from secExprTyping(5−6) this show thesis by (cases lev) (auto)
  next
    fix Γ e1 e2 lev bop
    assume x = Γ xa = e1«bop» e2 xb = High
    Γ ⊢ e1 : lev Γ ⊢ e2 : High
    from secExprTyping(6−7) this show thesis by (cases lev) (auto)
  qed
qed

lemmas [code-pred-intro] = typeSkip[where T=Low] typeSkip[where T=High]
  typeAssH[where T = Low] typeAssH[where T=High]
  typeAssL typeSeq typeWhile typeIf typeConvert

code-pred (modes: i => o => i => bool as compute-secComTyping,
  i => i => i => bool as check-secComTyping) secComTyping
proof −
  case secComTyping
  from secComTyping.prems show thesis
  proof
    fix Γ T assume x = Γ xa = T xb = Skip
    from secComTyping(1−2) this show thesis by (cases T) auto
  next
    fix Γ V T e assume x = Γ xa = T xb = V:=e Γ V = Some High
    from secComTyping(3−4) this show thesis by (cases T) (auto)
  next
    fix Γ e V
    assume x = Γ xa = Low xb = V:=e Γ ⊢ e : Low Γ V = Some Low
    from secComTyping(5) this show thesis by auto
  next
    fix Γ T c1 c2
    assume x = Γ xa = T xb = Seq c1 c2 Γ,T ⊢ c1 Γ,T ⊢ c2
    from secComTyping(6) this show thesis by auto
  next
    fix Γ b T c
    assume x = Γ xa = T xb = while (b) c Γ ⊢ b : T Γ,T ⊢ c
    from secComTyping(7) this show thesis by auto
  next
    fix Γ b T c1 c2
    assume x = Γ xa = T xb = if (b) c1 else c2 Γ ⊢ b : T Γ,T ⊢ c1 Γ,T ⊢ c2
    from secComTyping(8) this show thesis by blast
  next
    fix Γ c
    assume x = Γ xa = Low xb = c Γ,High ⊢ c
    from secComTyping(9) this show thesis by blast
  qed
qed
```

thm *secComTyping.equation*

3.1 An example taken from Volpano, Smith, Irvine

definition *com* = *if* (*Var* ''*x*'' «*Eq*» *Val* (*Intg 1*)) (''*y*'' := *Val* (*Intg 1*)) *else* (''*y*'' := *Val* (*Intg 0*))
definition *Env* = *map-of* [(''*x*'', *High*), (''*y*'', *High*)]

values {*T*. *Env* ⊢ (*Var* ''*x*'' «*Eq*» *Val* (*Intg 1*)): *T*}
value *Env*, *High* ⊢ *com*
value *Env*, *Low* ⊢ *com*
values *1* {*T*. *Env*, *T* ⊢ *com*}

definition *Env'* = *map-of* [(''*x*'', *Low*), (''*y*'', *High*)]

value *Env'*, *Low* ⊢ *com*
value *Env'*, *High* ⊢ *com*
values *1* {*T*. *Env*, *T* ⊢ *com*}

end

References

[1] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[2] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.