

Formalization of Hyper Hoare Logic: A Logic to (Dis-)Prove Program Hyperproperties

Thibault Dardinier
Department of Computer Science
ETH Zurich, Switzerland

June 6, 2026

Abstract

Hoare logics [6, 7] are proof systems that allow one to formally establish properties of computer programs. Traditional Hoare logics prove properties of individual program executions (such as functional correctness). On the one hand, Hoare logic has been generalized to prove properties of multiple executions of a program (so-called hyperproperties [1], such as determinism or non-interference). These program logics prove the absence of (bad combinations of) executions. On the other hand, program logics similar to Hoare logic have been proposed to disprove program properties (e.g., Incorrectness Logic [9]), by proving the existence of (bad combinations of) executions. All of these logics have in common that they specify program properties using assertions over a fixed number of states, for instance, a single pre- and post-state for functional properties or pairs of pre- and post-states for non-interference.

In this entry, we formalize Hyper Hoare Logic [2, 3], a generalization of Hoare logic that lifts assertions to properties of arbitrary sets of states. The resulting logic is simple yet expressive: its judgments can express arbitrary program hyperproperties, a particular class of hyperproperties over the set of terminating executions of a program (including properties of individual program executions). By allowing assertions to reason about sets of states, Hyper Hoare Logic can reason about both the absence and the existence of (combinations of) executions, and, thereby, supports both proving and disproving program (hyper-)properties within the same logic, including hyperproperties that no existing Hoare logic can express. We prove that Hyper Hoare Logic is sound and complete, and demonstrate that it captures important proof principles naturally.

Contents

1	Language and Semantics	4
1.1	Language	4
1.2	Semantics	4
1.3	Extended States and Extended Semantics	5
2	Hyper Hoare Logic	11
2.1	Rules of the Logic	15
2.2	Soundness	15
2.3	Completeness	19
2.4	Disproving Hyper-Triples	25
2.5	Synchronized Rule for Branching	26
3	Expressivity of Hyper Hoare Logic	29
3.1	Program Hyperproperties	29
3.2	Hoare Logic (HL) [7]	36
3.3	Cartesian Hoare Logic (CHL) [10]	38
3.4	Incorrectness Logic [9] or Reverse Hoare Logic [4] (IL)	44
3.5	k-Incorrectness Logic [8] (k-IL)	47
3.6	Forward Underapproximation (FU)	56
3.7	k-Forward Underapproximate logic	59
3.8	k-Universal Existential (RUE) [5]	63
3.9	Program Refinement	73
4	Rules for Loops	77
5	Compositionality Rules	99
5.1	Linking rule	100
5.2	Frame rules	100
5.3	Logical Updates	103
5.4	Filters	106
5.5	Other Compositionality Rules	108
5.6	Synchronous Reasoning (Proposition 14, Appendix H).	115
6	Syntactic Assertions	119
6.1	Preliminaries: Types, expressions, 'a assertions	119
6.2	Assume rule	121
6.2.1	Program expressions (values)	123
6.2.2	Program expressions (booleans)	124
6.2.3	Syntactic rule for assume	125
6.3	Havoc rule	125
6.3.1	Shifting variables	125
6.3.2	Expressions (Boolean and values)	129
6.3.3	Assertions	130

6.3.4	Transformation for havoc	134
6.3.5	Syntactic rule for havoc	137
6.4	Assignment rule	137
6.4.1	Program expressions	137
6.4.2	Expressions (Boolean and values)	138
6.4.3	Assertions	139
6.4.4	Syntactic rule for assignments	141
6.5	Loop rules	142
6.6	Rewrite rules for 'a assertions	149
6.7	Free variables and safe frame rule	151
7	Terminating Hyper-Triples	156
7.1	Specialize rule	160
7.2	Total version of core rules	177
8	Examples	182
8.1	Examples using the core rules.	182
8.2	Examples using the compositionality rules	182
8.3	Other examples	189
9	Summary of the Results from the Paper	193
9.1	3: Hyper Hoare Logic	194
9.1.1	3.1: Language and Semantics	194
9.1.2	3.2: Hyper-Triples, Formally	194
9.1.3	3.3: Core Rules	194
9.1.4	3.4: Soundness and Completeness	195
9.1.5	3.5: Expressivity of Hyper-Triples	195
9.2	4: Syntactic Rules	196
9.2.1	4.1: Syntactic Hyper-Assertions	196
9.2.2	4.2: Syntactic Rules for Deterministic and Non-Deterministic Assignments.	196
9.2.3	4.3: Syntactic Rules for Assume Statements	196
9.3	5: Proof Principles for Loops	197
9.4	Appendix A: Technical Definitions Omitted from the Paper	199
9.5	Appendix C: Expressing Judgments of Hoare Logics as Hyper- Triples	199
9.5.1	Appendix C.1: Overapproximate Hoare Logics	199
9.5.2	Appendix C.2: Underapproximate Hoare Logics	200
9.5.3	Appendix C.3: Beyond Over- and Underapproximation	201
9.6	Appendix D: Compositionality	202
9.6.1	Appendix D.1: Compositionality Rules	202
9.6.2	Appendix D.2: Examples	204
9.7	Appendix E: Termination-Based Reasoning	205
9.8	Appendix H: Synchronous Reasoning over Different Branches	206

1 Language and Semantics

In this file, we formalize concepts from section 3: - Program states and programming language (definition 1) - Big-step semantics (figure 2) - Extended states (definition 2) - Extended semantics (definition 4) and some useful properties (lemma 1)

```
theory Language
  imports Main
begin
```

1.1 Language

Definition 1

```
type-synonym ('var, 'val) pstate = 'var  $\Rightarrow$  'val
```

```
type-synonym ('var, 'val) bexp = ('var, 'val) pstate  $\Rightarrow$  bool
```

```
type-synonym ('var, 'val) exp = ('var, 'val) pstate  $\Rightarrow$  'val
```

```
datatype ('var, 'val) stmt =
  Assign 'var ('var, 'val) exp
  | Seq ('var, 'val) stmt ('var, 'val) stmt (infixl <;> 60)
  | If ('var, 'val) stmt ('var, 'val) stmt — Non-deterministic choice
  | Skip
  | Havoc 'var — Non-deterministic
assignment
  | Assume ('var, 'val) bexp
  | While ('var, 'val) stmt — Non-deterministic loop
```

1.2 Semantics

Figure 2

```
inductive single-sem :: ('var, 'val) stmt  $\Rightarrow$  ('var, 'val) pstate  $\Rightarrow$  ('var, 'val) pstate
 $\Rightarrow$  bool
```

```
(<<->, -)  $\rightarrow$  -> [51,0] 81)
```

```
where
```

```
SemSkip: <Skip,  $\sigma$ >  $\rightarrow$   $\sigma$ 
```

```
| SemAssign: <Assign var e,  $\sigma$ >  $\rightarrow$   $\sigma$ (var := (e  $\sigma$ ))
```

```
| SemSeq:  $\llbracket$  <C1,  $\sigma$ >  $\rightarrow$   $\sigma$ 1; <C2,  $\sigma$ 1>  $\rightarrow$   $\sigma$ 2  $\rrbracket \Longrightarrow$  <Seq C1 C2,  $\sigma$ >  $\rightarrow$   $\sigma$ 2
```

```
| SemIf1: <C1,  $\sigma$ >  $\rightarrow$   $\sigma$ 1  $\Longrightarrow$  <If C1 C2,  $\sigma$ >  $\rightarrow$   $\sigma$ 1
```

```
| SemIf2: <C2,  $\sigma$ >  $\rightarrow$   $\sigma$ 2  $\Longrightarrow$  <If C1 C2,  $\sigma$ >  $\rightarrow$   $\sigma$ 2
```

```
| SemHavoc: <Havoc var,  $\sigma$ >  $\rightarrow$   $\sigma$ (var := v)
```

```
| SemAssume: b  $\sigma \Longrightarrow$  <Assume b,  $\sigma$ >  $\rightarrow$   $\sigma$ 
```

```
| SemWhileIter:  $\llbracket$  <C,  $\sigma$ >  $\rightarrow$   $\sigma'$ ; <While C,  $\sigma'$ >  $\rightarrow$   $\sigma''$   $\rrbracket \Longrightarrow$  <While C,  $\sigma$ >  $\rightarrow$   $\sigma''$ 
```

```
| SemWhileExit: <While C,  $\sigma$ >  $\rightarrow$   $\sigma$ 
```

```
inductive-cases single-sem-Seq-elim[elim!]: <Seq C1 C2,  $\sigma$ >  $\rightarrow$   $\sigma'$ 
```

inductive-cases *single-sem-Skip-elim*[*elim!*]: $\langle \text{Skip}, \sigma \rangle \rightarrow \sigma'$
inductive-cases *single-sem-While-elim*: $\langle \text{While } C, \sigma \rangle \rightarrow \sigma'$
inductive-cases *single-sem-If-elim*[*elim!*]: $\langle \text{If } C1 \ C2, \sigma \rangle \rightarrow \sigma'$
inductive-cases *single-sem-Assume-elim*[*elim!*]: $\langle \text{Assume } b, \sigma \rangle \rightarrow \sigma'$
inductive-cases *single-sem-Assign-elim*[*elim!*]: $\langle \text{Assign } x \ e, \sigma \rangle \rightarrow \sigma'$
inductive-cases *single-sem-Havoc-elim*[*elim!*]: $\langle \text{Havoc } x, \sigma \rangle \rightarrow \sigma'$

1.3 Extended States and Extended Semantics

Definition 2: Extended states

type-synonym (*'lvar*, *'lval*, *'pvar*, *'pval*) *state* = (*'lvar* \Rightarrow *'lval*) \times (*'pvar*, *'pval*)
pstate

Definition 4: Extended semantics

definition *sem* :: (*'pvar*, *'pval*) *stmt* \Rightarrow (*'lvar*, *'lval*, *'pvar*, *'pval*) *state set* \Rightarrow
(*'lvar*, *'lval*, *'pvar*, *'pval*) *state set* **where**
sem *C S* = { (*l*, σ') | $\sigma' \ \sigma \ l$. (*l*, σ) $\in S \wedge \langle C, \sigma \rangle \rightarrow \sigma'$ }

lemma *in-sem*:

$\varphi \in \text{sem } C \ S \iff (\exists \sigma. (\text{fst } \varphi, \sigma) \in S \wedge \text{single-sem } C \ \sigma \ (\text{snd } \varphi)) \ (\text{is } ?A \iff ?B)$

proof

assume *?A*

then obtain $\sigma' \ \sigma \ l$ **where** $\varphi = (l, \sigma')$ (*l*, σ) $\in S \wedge \langle C, \sigma \rangle \rightarrow \sigma'$

using *sem-def*[of *C S*] **by** *auto*

then show *?B*

by *auto*

next

show *?B* \implies *?A*

by (*metis* (*mono-tags*, *lifting*) *CollectI prod.collapse sem-def*)

qed

Lemma 1: Useful properties of the extended semantics

lemma *sem-seq*:

$\text{sem } (\text{Seq } C1 \ C2) \ S = \text{sem } C2 \ (\text{sem } C1 \ S) \ (\text{is } ?A = ?B)$

proof

show *?A* \subseteq *?B*

proof

fix *x2* **assume** *x2* \in *?A*

then obtain *x0* **where** (*fst* *x2*, *x0*) $\in S$ *single-sem* (*Seq* *C1* *C2*) *x0* (*snd* *x2*)

by (*metis in-sem*)

then obtain *x1* **where** *single-sem* *C1* *x0* *x1* *single-sem* *C2* *x1* (*snd* *x2*)

using *single-sem-Seq-elim*[of *C1 C2 x0 snd x2*]

by *blast*

then show *x2* \in *?B*

by (*metis* $\langle \text{fst } x2, x0 \rangle \in S \rangle$ *fst-conv in-sem snd-conv*)

qed

show *?B* \subseteq *?A*

```

proof
  fix  $x2$  assume  $x2 \in ?B$ 
  then obtain  $x1$  where  $(fst\ x2, x1) \in sem\ C1\ S\ single-sem\ C2\ x1\ (snd\ x2)$ 
    by  $(metis\ in-sem)$ 
  then obtain  $x0$  where  $(fst\ x2, x0) \in S\ single-sem\ C1\ x0\ x1$ 
    by  $(metis\ fst-conv\ in-sem\ snd-conv)$ 
  then have  $single-sem\ (Seq\ C1\ C2)\ x0\ (snd\ x2)$ 
    by  $(simp\ add:\ SemSeq\ \langle C2, x1 \rangle \rightarrow\ snd\ x2)$ 
  then show  $x2 \in ?A$ 
    by  $(meson\ \langle (fst\ x2, x0) \in S \rangle\ in-sem)$ 
qed
qed

```

```

lemma sem-skip:
   $sem\ Skip\ S = S$ 
  using single-sem-Skip-elim SemSkip in-sem[of - Skip S]
  by fastforce

```

```

lemma sem-union:
   $sem\ C\ (S1 \cup S2) = sem\ C\ S1 \cup sem\ C\ S2$  (is  $?A = ?B$ )

```

```

proof
  show  $?A \subseteq ?B$ 
  proof
    fix  $x$  assume  $x \in ?A$ 
    then obtain  $y$  where  $(fst\ x, y) \in S1 \cup S2\ single-sem\ C\ y\ (snd\ x)$ 
      using in-sem by blast
    then show  $x \in ?B$ 
      by  $(metis\ Un-iff\ in-sem)$ 
  qed
  show  $?B \subseteq ?A$ 
  proof
    fix  $x$  assume  $x \in ?B$ 
    show  $x \in ?A$ 
    proof  $(cases\ x \in sem\ C\ S1)$ 
      case True
      then show  $?thesis$ 
        by  $(metis\ IntD2\ Un-Int-eq(3)\ in-sem)$ 
      next
      case False
      then show  $?thesis$ 
        by  $(metis\ Un-iff\ \langle x \in sem\ C\ S1 \cup sem\ C\ S2 \rangle\ in-sem)$ 
    qed
  qed
qed

```

```

lemma sem-union-general:
   $sem\ C\ (\bigcup x. f\ x) = (\bigcup x. sem\ C\ (f\ x))$  (is  $?A = ?B$ )

```

```

proof
  show  $?A \subseteq ?B$ 

```

proof
fix b **assume** $b \in ?A$
then obtain a **where** $a \in (\bigcup x. f\ x)$ $\text{fst } a = \text{fst } b$ *single-sem* C $(\text{snd } a)$ $(\text{snd } b)$
by *(metis fst-conv in-sem snd-conv)*
then obtain x **where** $a \in f\ x$ **by** *blast*
then have $b \in \text{sem } C\ (f\ x)$
by *(metis <C, snd a> → snd b> <fst a = fst b> in-sem surjective-pairing)*
then show $b \in ?B$
by *blast*
qed
show $?B \subseteq ?A$
proof
fix y **assume** $y \in ?B$
then obtain x **where** $y \in \text{sem } C\ (f\ x)$
by *blast*
then show $y \in ?A$
by *(meson UN-I in-sem iso-tuple-UNIV-I)*
qed
qed

lemma *sem-monotonic*:
assumes $S \subseteq S'$
shows $\text{sem } C\ S \subseteq \text{sem } C\ S'$
by *(metis assms sem-union subset-Un-eq)*

lemma *subsetPairI*:
assumes $\bigwedge l\ \sigma. (l, \sigma) \in A \implies (l, \sigma) \in B$
shows $A \subseteq B$
by *(simp add: assms subrelI)*

lemma *sem-if*:
 $\text{sem } (\text{If } C1\ C2)\ S = \text{sem } C1\ S \cup \text{sem } C2\ S$ **(is** $?A = ?B$ **)**
proof
show $?A \subseteq ?B$
proof *(rule subsetPairI)*
fix $l\ y$ **assume** $(l, y) \in ?A$
then obtain x **where** $(l, x) \in S$ *single-sem* $(\text{If } C1\ C2)$ $x\ y$
by *(metis fst-conv in-sem snd-conv)*
then show $(l, y) \in ?B$ **using** *single-sem-If-elim*
 $UnI1\ UnI2$ *in-sem*
by *(metis fst-conv snd-conv)*
qed
show $?B \subseteq ?A$
using *SemIf1 SemIf2 in-sem*
by *(metis (no-types, lifting) Un-subset-iff subsetI)*
qed

lemma *sem-assume*:

```

sem (Assume b) S = { (l, σ) | l σ. (l, σ) ∈ S ∧ b σ } (is ?A = ?B)
proof
  show ?A ⊆ ?B
  proof (rule subsetPairI)
    fix l y assume (l, y) ∈ ?A then obtain x where (l, x) ∈ S single-sem (Assume
b) x y
      using in-sem
      by (metis fst-conv snd-conv)
      then show (l, y) ∈ ?B using single-sem-Assume-elim by blast
    qed
  show ?B ⊆ ?A
  proof (rule subsetPairI)
    fix l σ assume asm0: (l, σ) ∈ {(l, σ) | l σ. (l, σ) ∈ S ∧ b σ}
    then have (l, σ) ∈ S b σ by simp-all
    then show (l, σ) ∈ sem (Assume b) S
      by (metis SemAssume fst-eqD in-sem snd-eqD)
    qed
  qed

```

```

lemma while-then-reaches:
  assumes (single-sem C)** σ σ''
  shows single-sem (While C) σ σ''
  using assms
proof (induct rule: converse-rtranclp-induct)
  case base
  then show ?case
    by (simp add: SemWhileExit)
  next
  case (step y z)
  then show ?case
    using SemWhileIter by blast
  qed

```

```

lemma in-closure-then-while:
  assumes single-sem C' σ σ''
  shows  $\bigwedge C. C' = \text{While } C \implies (\text{single-sem } C)** \sigma \sigma''$ 
  using assms
proof (induct rule: single-sem.induct)
  case (SemWhileIter σ C' σ' σ'')
  then show ?case
    by (metis (no-types, lifting) rtranclp.rtrancl-into-rtrancl rtranclp.rtrancl-refl
rtranclp-trans stmt.inject(6))
  next
  case (SemWhileExit σ C')
  then show ?case
    by blast
  qed (auto)

```

```

theorem loop-equiv:

```

single-sem (While C) $\sigma \sigma' \longleftrightarrow (\text{single-sem } C)^{**} \sigma \sigma'$
using *in-closure-then-while while-then-reaches* **by** *blast*

fun *iterate-sem* **where**

iterate-sem 0 - S = S

| *iterate-sem* (Suc n) C S = *sem* C (*iterate-sem* n C S)

lemma *in-iterate-then-in-trans*:

assumes $(l, \sigma'') \in \text{iterate-sem } n \ C \ S$

shows $\exists \sigma. (l, \sigma) \in S \wedge (\text{single-sem } C)^{**} \sigma \sigma''$

using *assms*

proof (*induct* n *arbitrary*: $\sigma'' \ S$)

case 0

then show *?case*

using *iterate-sem.simps(1)* **by** *blast*

next

case (Suc n)

then show *?case*

using *in-sem rtranclp.rtrancl-into-rtrancl*

by (*metis* (*mono-tags*, *lifting*) *fst-conv* *iterate-sem.simps(2)* *snd-conv*)

qed

lemma *reciprocal*:

assumes $(\text{single-sem } C)^{**} \sigma \sigma''$

and $(l, \sigma) \in S$

shows $\exists n. (l, \sigma'') \in \text{iterate-sem } n \ C \ S$

using *assms*

proof (*induct* rule: *rtranclp-induct*)

case *base*

then show *?case*

using *iterate-sem.simps(1)* **by** *blast*

next

case (*step* y z)

then obtain n **where** $(l, y) \in \text{iterate-sem } n \ C \ S$ **by** *blast*

then show *?case*

using *in-sem* *iterate-sem.simps(2)* *step.hyps(2)*

by (*metis* *fst-eqD* *snd-eqD*)

qed

lemma *union-iterate-sem-trans*:

$(l, \sigma'') \in (\bigcup n. \text{iterate-sem } n \ C \ S) \longleftrightarrow (\exists \sigma. (l, \sigma) \in S \wedge (\text{single-sem } C)^{**} \sigma \sigma'')$ (**is** *?A* \longleftrightarrow *?B*)

using *in-iterate-then-in-trans reciprocal* **by** *auto*

lemma *sem-while*:

sem (While C) S = $(\bigcup n. \text{iterate-sem } n \ C \ S)$ (**is** *?A* = *?B*)

proof

show *?A* \subseteq *?B*

```

proof (rule subsetPairI)
  fix  $l\ y$  assume  $(l, y) \in ?A$ 
  then obtain  $x$  where  $x\text{-def}: (l, x) \in S$  (single-sem  $C$ )**  $x\ y$ 
    using in-closure-then-while in-sem
    by (metis fst-eqD snd-conv)
  then have single-sem (While  $C$ )  $x\ y$ 
    using while-then-reaches by blast
  then show  $(l, y) \in ?B$ 
    by (metis x-def union-iterate-sem-trans)
qed
show  $?B \subseteq ?A$ 
proof (rule subsetPairI)
  fix  $l\ y$  assume  $(l, y) \in ?B$ 
  then obtain  $x$  where  $(l, x) \in S$  (single-sem  $C$ )**  $x\ y$ 
    using union-iterate-sem-trans by blast
  then show  $(l, y) \in ?A$ 
    using in-sem while-then-reaches by fastforce
qed
qed

```

lemma *assume-sem*:

```

sem (Assume  $b$ )  $S = \text{Set.filter } (b \circ \text{snd})\ S$  (is  $?A = ?B$ )
by (auto simp add: sem-assume)

```

lemma *sem-split-general*:

```

sem  $C$  ( $\bigcup x. F\ x$ ) = ( $\bigcup x. \text{sem } C\ (F\ x)$ ) (is  $?A = ?B$ )
proof
show  $?A \subseteq ?B$ 
proof (rule subsetPairI)
  fix  $l\ \sigma'$ 
  assume  $\text{asm0}: (l, \sigma') \in \text{sem } C\ (\bigcup (\text{range } F))$ 
  then obtain  $x\ \sigma$  where  $(l, \sigma) \in F\ x$  single-sem  $C\ \sigma\ \sigma'$ 
    by (metis (no-types, lifting) UN-iff fst-conv in-sem snd-conv)
  then show  $(l, \sigma') \in (\bigcup x. \text{sem } C\ (F\ x))$ 
    using  $\text{asm0}$  sem-union-general by blast
qed
show  $?B \subseteq ?A$ 
  by (simp add: SUP-least Sup-upper sem-monotonic)
qed

```

fun *written-vars* **where**

```

written-vars (Assign  $x\ -$ ) =  $\{x\}$ 
| written-vars (Havoc  $x$ ) =  $\{x\}$ 
| written-vars ( $C1\ ;;\ C2$ ) = written-vars  $C1 \cup$  written-vars  $C2$ 
| written-vars (If  $C1\ C2$ ) = written-vars  $C1 \cup$  written-vars  $C2$ 
| written-vars (While  $C$ ) = written-vars  $C$ 
| written-vars - =  $\{\}$ 

```

```

lemma written-vars-not-modified:
  assumes single-sem C  $\varphi \varphi'$ 
    and  $x \notin \text{written-vars } C$ 
    shows  $\varphi x = \varphi' x$ 
  using assms
  by (induct rule: single-sem.induct) auto

end

```

2 Hyper Hoare Logic

This file contains technical results from sections 3 and 5: - Hyper-assertions (definition 3) - Hyper-triples (definition 5) - Core rules of Hyper Hoare Logic (figure 2) - Soundness of the core rules (theorem 1) - Completeness of the core rules (theorem 2) - Ability to disprove hyper-triples (theorem 5)

```

theory Logic
  imports Language
begin

```

Definition 3

```

type-synonym 'a hyperassertion = ('a set  $\Rightarrow$  bool)

```

```

definition entails where
  entails A B  $\longleftrightarrow (\forall S. A S \longrightarrow B S)$ 

```

```

lemma entails-refl:
  entails A A
  by (simp add: entails-def)

```

```

lemma entailsI:
  assumes  $\bigwedge S. A S \Longrightarrow B S$ 
  shows entails A B
  by (simp add: assms entails-def)

```

```

lemma entailsE:
  assumes entails A B
    and A x
  shows B x
  by (meson assms(1) assms(2) entails-def)

```

```

lemma bientails-equal:
  assumes entails A B
    and entails B A
  shows A = B
proof (rule ext)
  fix S show A S = B S
  by (meson assms(1) assms(2) entailsE)

```

qed

lemma *entails-trans*:

assumes *entails A B*
and *entails B C*
shows *entails A C*
by (*metis assms(1) assms(2) entails-def*)

definition *setify-prop where*

setify-prop b = { (l, σ) | l σ. b σ }

lemma *sem-assume-setify*:

sem (Assume b) S = S ∩ setify-prop b (is ?A = ?B)

proof –

have $\bigwedge l \sigma. (l, \sigma) \in ?A \longleftrightarrow (l, \sigma) \in ?B$

proof –

fix *l σ*

have $(l, \sigma) \in ?A \longleftrightarrow (l, \sigma) \in S \wedge b \sigma$

by (*simp add: assume-sem*)

then show $(l, \sigma) \in ?A \longleftrightarrow (l, \sigma) \in ?B$

by (*simp add: setify-prop-def*)

qed

then show *?thesis*

by *auto*

qed

definition *over-approx :: 'a set ⇒ 'a hyperassertion where*

over-approx P S $\longleftrightarrow S \subseteq P$

definition *lower-closed :: 'a hyperassertion ⇒ bool where*

lower-closed P $\longleftrightarrow (\forall S S'. P S \wedge S' \subseteq S \longrightarrow P S')$

lemma *over-approx-lower-closed*:

lower-closed (over-approx P)

by (*metis (full-types) lower-closed-def order-trans over-approx-def*)

definition *under-approx :: 'a set ⇒ 'a hyperassertion where*

under-approx P S $\longleftrightarrow P \subseteq S$

definition *upper-closed :: 'a hyperassertion ⇒ bool where*

upper-closed P $\longleftrightarrow (\forall S S'. P S \wedge S \subseteq S' \longrightarrow P S')$

lemma *under-approx-upper-closed*:

upper-closed (under-approx P)

by (*metis (no-types, lifting) order.trans under-approx-def upper-closed-def*)

definition *closed-by-union :: 'a hyperassertion ⇒ bool where*

closed-by-union P $\longleftrightarrow (\forall S S'. P S \wedge P S' \longrightarrow P (S \cup S'))$

lemma *closed-by-unionI*:
assumes $\bigwedge a b. P a \implies P b \implies P (a \cup b)$
shows *closed-by-union* P
by (*simp add: assms closed-by-union-def*)

lemma *closed-by-union-over*:
closed-by-union (*over-approx* P)
by (*simp add: closed-by-union-def over-approx-def*)

lemma *closed-by-union-under*:
closed-by-union (*under-approx* P)
by (*simp add: closed-by-union-def sup.coboundedI1 under-approx-def*)

definition *conj where*
 $conj\ P\ Q\ S \longleftrightarrow P\ S \wedge Q\ S$

lemma *entail-conj*:
assumes *entails* $A\ B$
shows *entails* A (*conj* $A\ B$)
by (*metis (full-types) assms conj-def entails-def*)

lemma *entail-conj-weaken*:
entails (*conj* $A\ B$) A
by (*simp add: conj-def entails-def*)

definition *disj where*
 $disj\ P\ Q\ S \longleftrightarrow P\ S \vee Q\ S$

definition *exists* :: ($'c \Rightarrow 'a$ hyperassertion) $\Rightarrow 'a$ hyperassertion **where**
 $exists\ P\ S \longleftrightarrow (\exists x. P\ x\ S)$

definition *forall* :: ($'c \Rightarrow 'a$ hyperassertion) $\Rightarrow 'a$ hyperassertion **where**
 $forall\ P\ S \longleftrightarrow (\forall x. P\ x\ S)$

lemma *over-inter*:
entails (*over-approx* ($P \cap Q$)) (*conj* (*over-approx* P) (*over-approx* Q))
by (*simp add: conj-def entails-def over-approx-def*)

lemma *over-union*:
entails (*disj* (*over-approx* P) (*over-approx* Q)) (*over-approx* ($P \cup Q$))
by (*metis disj-def entailsI le-supI1 le-supI2 over-approx-def*)

lemma *under-union*:
entails (*under-approx* ($P \cup Q$)) (*disj* (*under-approx* P) (*under-approx* Q))
by (*simp add: disj-def entails-def under-approx-def*)

lemma *under-inter*:
entails (*conj* (*under-approx* P) (*under-approx* Q)) (*under-approx* ($P \cap Q$))

by (*simp add: conj-def entails-def le-infI1 under-approx-def*)

Definition 6: Operator \otimes

definition *join* :: 'a hyperassertion \Rightarrow 'a hyperassertion \Rightarrow 'a hyperassertion **where**
join A B S $\longleftrightarrow (\exists SA SB. A SA \wedge B SB \wedge S = SA \cup SB)$

definition *general-join* :: ('b \Rightarrow 'a hyperassertion) \Rightarrow 'a hyperassertion **where**
general-join f S $\longleftrightarrow (\exists F. S = (\bigcup x. F x) \wedge (\forall x. f x (F x)))$

lemma *general-joinI*:

assumes $S = (\bigcup x. F x)$
and $\bigwedge x. f x (F x)$
shows *general-join* f S
using *assms(1) assms(2) general-join-def* **by** *blast*

lemma *join-closed-by-union*:

assumes *closed-by-union* Q
shows *join* Q Q = Q

proof

fix S
show *join* Q Q S \longleftrightarrow Q S
by (*metis assms closed-by-union-def join-def sup-idem*)

qed

lemma *entails-join-entails*:

assumes *entails* A1 B1
and *entails* A2 B2
shows *entails* (*join* A1 A2) (*join* B1 B2)

proof (*rule entailsI*)

fix S **assume** *join* A1 A2 S
then obtain S1 S2 **where** A1 S1 A2 S2 S = S1 \cup S2
by (*metis join-def*)
then show *join* B1 B2 S
by (*metis assms(1) assms(2) entailsE join-def*)

qed

Definition 7: Operator \otimes (for $x \in X$)

definition *natural-partition* **where**

natural-partition I S $\longleftrightarrow (\exists F. S = (\bigcup n. F n) \wedge (\forall n. I n (F n)))$

lemma *natural-partitionI*:

assumes $S = (\bigcup n. F n)$
and $\bigwedge n. I n (F n)$
shows *natural-partition* I S
using *assms(1) assms(2) natural-partition-def* **by** *blast*

lemma *natural-partitionE*:

assumes *natural-partition* I S
obtains F **where** $S = (\bigcup n. F n) \wedge \bigwedge n. I n (F n)$

by (*meson assms natural-partition-def*)

2.1 Rules of the Logic

Core rules from figure 2

inductive syntactic-HHT ::

((*'lvar, 'lval, 'pvar, 'pval*) state hyperassertion) \Rightarrow (*'pvar, 'pval*) stmt \Rightarrow ((*'lvar, 'lval, 'pvar, 'pval*) state hyperassertion) \Rightarrow bool

($\langle \vdash \{-\} - \{-\} \rangle$ [51,0,0] 81) **where**

RuleSkip: $\vdash \{P\}$ Skip $\{P\}$

| *RuleCons*: $\llbracket \text{entails } P \ P' ; \text{entails } Q' \ Q ; \vdash \{P'\} \ C \ \{Q'\} \rrbracket \Longrightarrow \vdash \{P\} \ C \ \{Q\}$

| *RuleSeq*: $\llbracket \vdash \{P\} \ C1 \ \{R\} ; \vdash \{R\} \ C2 \ \{Q\} \rrbracket \Longrightarrow \vdash \{P\} \ (\text{Seq } C1 \ C2) \ \{Q\}$

| *RuleIf*: $\llbracket \vdash \{P\} \ C1 \ \{Q1\} ; \vdash \{P\} \ C2 \ \{Q2\} \rrbracket \Longrightarrow \vdash \{P\} \ (\text{If } C1 \ C2) \ \{\text{join } Q1 \ Q2\}$

| *RuleWhile*: $\llbracket \bigwedge n. \vdash \{In\} \ C \ \{I \ (\text{Suc } n)\} \rrbracket \Longrightarrow \vdash \{I0\} \ (\text{While } C) \ \{\text{natural-partition } I\}$

| *RuleAssume*: $\vdash \{ (\lambda S. P \ (\text{Set.filter } (b \circ \text{snd}) \ S)) \} \ (\text{Assume } b) \ \{P\}$

| *RuleAssign*: $\vdash \{ (\lambda S. P \ \{ (l, \sigma(x := e \ \sigma)) \mid l \ \sigma. (l, \sigma) \in S \}) \} \ (\text{Assign } x \ e) \ \{P\}$

| *RuleHavoc*: $\vdash \{ (\lambda S. P \ \{ (l, \sigma(x := v)) \mid l \ \sigma \ v. (l, \sigma) \in S \}) \} \ (\text{Havoc } x) \ \{P\}$

| *RuleExistsSet*: $\llbracket \bigwedge x. (\text{'lvar, 'lval, 'pvar, 'pval}) \ \text{state set. } \vdash \{P \ x\} \ C \ \{Q \ x\} \rrbracket \Longrightarrow \vdash \{\text{exists } P\} \ C \ \{\text{exists } Q\}$

2.2 Soundness

Definition 5: Hyper-Triples

definition hyper-hoare-triple ($\langle \models \{-\} - \{-\} \rangle$ [51,0,0] 81) **where**
 $\models \{P\} \ C \ \{Q\} \longleftrightarrow (\forall S. P \ S \longrightarrow Q \ (\text{sem } C \ S))$

lemma hyper-hoare-tripleI:

assumes $\bigwedge S. P \ S \Longrightarrow Q \ (\text{sem } C \ S)$

shows $\models \{P\} \ C \ \{Q\}$

by (*simp add: assms hyper-hoare-triple-def*)

lemma hyper-hoare-tripleE:

assumes $\models \{P\} \ C \ \{Q\}$

and $P \ S$

shows $Q \ (\text{sem } C \ S)$

using *assms(1) assms(2) hyper-hoare-triple-def*

by *metis*

lemma consequence-rule:

assumes *entails* $P \ P'$

and *entails* $Q' \ Q$

and $\models \{P'\} \ C \ \{Q'\}$

shows $\models \{P\} \ C \ \{Q\}$

by (*metis (no-types, opaque-lifting) assms(1) assms(2) assms(3) entails-def hyper-hoare-triple-def*)

```

lemma skip-rule:
   $\models \{P\} \text{Skip} \{P\}$ 
  by (simp add: hyper-hoare-triple-def sem-skip)

lemma assume-rule:
   $\models \{ (\lambda S. P (\text{Set.filter } (b \circ \text{snd}) S)) \} (\text{Assume } b) \{P\}$ 
proof (rule hyper-hoare-tripleI)
  fix  $S$  assume  $P (\text{Set.filter } (b \circ \text{snd}) S)$ 
  then show  $P (\text{sem } (\text{Assume } b) S)$ 
    by (simp add: assume-sem)
qed

lemma seq-rule:
  assumes  $\models \{P\} C1 \{R\}$ 
  and  $\models \{R\} C2 \{Q\}$ 
  shows  $\models \{P\} \text{Seq } C1 C2 \{Q\}$ 
  using assms(1) assms(2) hyper-hoare-triple-def sem-seq
  by metis

lemma if-rule:
  assumes  $\models \{P\} C1 \{Q1\}$ 
  and  $\models \{P\} C2 \{Q2\}$ 
  shows  $\models \{P\} \text{If } C1 C2 \{\text{join } Q1 Q2\}$ 
  by (metis (full-types) assms(1) assms(2) hyper-hoare-triple-def join-def sem-if)

lemma sem-assign:
   $\text{sem } (\text{Assign } x e) S = \{(l, \sigma(x := e \sigma)) \mid l \sigma. (l, \sigma) \in S\}$  (is  $?A = ?B$ )
proof
  show  $?A \subseteq ?B$ 
  proof (rule subsetPairI)
    fix  $l \sigma'$ 
    assume  $(l, \sigma') \in \text{sem } (\text{Assign } x e) S$ 
    then obtain  $\sigma$  where  $(l, \sigma) \in S$  single-sem (Assign x e)  $\sigma \sigma'$ 
    by (metis fst-eqD in-sem snd-conv)
    then show  $(l, \sigma') \in \{(l, \sigma(x := e \sigma)) \mid l \sigma. (l, \sigma) \in S\}$ 
    by blast
  qed
  show  $?B \subseteq ?A$ 
  proof (rule subsetPairI)
    fix  $l \sigma'$ 
    assume  $(l, \sigma') \in ?B$ 
    then obtain  $\sigma$  where  $\sigma' = \sigma(x := e \sigma)$   $(l, \sigma) \in S$ 
    by blast
    then show  $(l, \sigma') \in ?A$ 
    by (metis SemAssign fst-eqD in-sem snd-conv)
  qed
qed

lemma assign-rule:

```

$\models \{ (\lambda S. P \{ (l, \sigma(x := e \sigma)) \mid l \sigma. (l, \sigma) \in S \}) \} (Assign\ x\ e) \{P\}$
proof (*rule hyper-hoare-tripleI*)
fix S **assume** $P \{ (l, \sigma(x := e \sigma)) \mid l \sigma. (l, \sigma) \in S \}$
then show $P (sem (Assign\ x\ e) S)$ **using** *sem-assign*
by *metis*
qed

lemma *sem-havoc*:
 $sem (Havoc\ x) S = \{ (l, \sigma(x := v)) \mid l \sigma v. (l, \sigma) \in S \}$ (**is** $?A = ?B$)
proof
show $?A \subseteq ?B$
proof (*rule subsetPairI*)
fix $l \sigma'$
assume $(l, \sigma') \in sem (Havoc\ x) S$
then obtain σ **where** $(l, \sigma) \in S$ *single-sem (Havoc x) $\sigma \sigma'$*
by (*metis fst-eqD in-sem snd-conv*)
then show $(l, \sigma') \in \{ (l, \sigma(x := v)) \mid l \sigma v. (l, \sigma) \in S \}$
by *blast*
qed
show $?B \subseteq ?A$
proof (*rule subsetPairI*)
fix $l \sigma'$
assume $(l, \sigma') \in ?B$
then obtain σv **where** $\sigma' = \sigma(x := v)$ $(l, \sigma) \in S$
by *blast*
then show $(l, \sigma') \in ?A$
by (*metis SemHavoc fst-eqD in-sem snd-conv*)
qed
qed

lemma *havoc-rule*:
 $\models \{ (\lambda S. P \{ (l, \sigma(x := v)) \mid l \sigma v. (l, \sigma) \in S \}) \} (Havoc\ x) \{P\}$
proof (*rule hyper-hoare-tripleI*)
fix S **assume** $P \{ (l, \sigma(x := v)) \mid l \sigma v. (l, \sigma) \in S \}$
then show $P (sem (Havoc\ x) S)$ **using** *sem-havoc* **by** *metis*
qed

Loops

lemma *indexed-invariant-then-power*:
assumes $\bigwedge n. hyper-hoare-triple (I\ n) C (I (Suc\ n))$
and $I\ 0\ S$
shows $I\ n (iterate-sem\ n\ C\ S)$
using *assms*
proof (*induct n arbitrary: S*)
next
case ($Suc\ n$)
then have $I\ n (iterate-sem\ n\ C\ S)$
by *blast*
then have $I (Suc\ n) (sem\ C (iterate-sem\ n\ C\ S))$

```

    using Suc.premis(1) hyper-hoare-tripleE by blast
  then show ?case
    by (simp add: Suc.hyps Suc.premis(1))
qed (auto)

```

lemma *indexed-invariant-then-power-bounded*:

```

  assumes  $\bigwedge m. m < n \implies \text{hyper-hoare-triple } (I\ m)\ C\ (I\ (\text{Suc } m))$ 
    and  $I\ 0\ S$ 
  shows  $I\ n\ (\text{iterate-sem } n\ C\ S)$ 
  using assms
proof (induct n arbitrary: S)
next
  case (Suc n)
  then have  $I\ n\ (\text{iterate-sem } n\ C\ S)$ 
    using less-Suc-eq by presburger
  then have  $I\ (\text{Suc } n)\ (\text{sem } C\ (\text{iterate-sem } n\ C\ S))$ 
    using Suc.premis(1) hyper-hoare-tripleE by blast
  then show ?case
    by (simp add: Suc.hyps Suc.premis(1))
qed (auto)

```

lemma *while-rule*:

```

  assumes  $\bigwedge n. \text{hyper-hoare-triple } (I\ n)\ C\ (I\ (\text{Suc } n))$ 
  shows  $\text{hyper-hoare-triple } (I\ 0)\ (\text{While } C)\ (\text{natural-partition } I)$ 
proof (rule hyper-hoare-tripleI)
  fix S assume asm0: I 0 S
  show  $\text{natural-partition } I\ (\text{sem } (\text{While } C)\ S)$ 
proof (rule natural-partitionI)
  show  $\text{sem } (\text{While } C)\ S = \bigcup (\text{range } (\lambda n. \text{iterate-sem } n\ C\ S))$ 
    by (simp add: sem-while)
  fix n show  $I\ n\ (\text{iterate-sem } n\ C\ S)$ 
    by (simp add: asm0 assms indexed-invariant-then-power)
qed
qed

```

lemma *rule-exists*:

```

  assumes  $\bigwedge x. \models \{P\ x\}\ C\ \{Q\ x\}$ 
  shows  $\models \{\text{exists } P\}\ C\ \{\text{exists } Q\}$ 
  by (metis assms exists-def hyper-hoare-triple-def)

```

Theorem 1

theorem *soundness*:

```

  assumes  $\vdash \{A\}\ C\ \{B\}$ 
  shows  $\models \{A\}\ C\ \{B\}$ 
  using assms
proof (induct rule: syntactic-HHT.induct)
  case (RuleSkip P)
  then show ?case
    using skip-rule by auto

```

```

next
  case (RuleCons P P' Q' Q C)
  then show ?case
    using consequence-rule by blast
next
  case (RuleExistsSet P C Q)
  then show ?case
    using rule-exists by blast
next
  case (RuleSeq P C1 R C2 Q)
  then show ?case
    using seq-rule by meson
next
  case (RuleIf P C1 Q1 C2 Q2)
  then show ?case
    using if-rule by blast
next
  case (RuleAssume P b)
  then show ?case
    by (auto simp add: assume-rule simp del: Set.filter-eq)
next
  case (RuleWhile I C)
  then show ?case
    using while-rule by blast
next
  case (RuleAssign x e)
  then show ?case
    by (simp add: assign-rule)
next
  case (RuleHavoc x)
  then show ?case
    using havoc-rule by fastforce
qed

```

2.3 Completeness

definition *complete*

where

$complete\ P\ C\ Q \iff (\models \{P\}\ C\ \{Q\} \longrightarrow \vdash \{P\}\ C\ \{Q\})$

lemma *completeI*:

assumes $\models \{P\}\ C\ \{Q\} \implies \vdash \{P\}\ C\ \{Q\}$

shows *complete* $P\ C\ Q$

by (*simp* add: *assms* *complete-def*)

lemma *completeE*:

assumes *complete* $P\ C\ Q$

and $\models \{P\}\ C\ \{Q\}$

shows $\vdash \{P\}\ C\ \{Q\}$

using *assms complete-def* **by** *auto*

lemma *complete-if-aux*:

assumes *hyper-hoare-triple* A (*If* $C1$ $C2$) B
shows *entails* $(\lambda S'. \exists S. A S \wedge S' = \text{sem } C1 S \cup \text{sem } C2 S) B$

proof (*rule entailsI*)

fix S' **assume** $\exists S. A S \wedge S' = \text{sem } C1 S \cup \text{sem } C2 S$

then show $B S'$

by (*metis assms hyper-hoare-tripleE sem-if*)

qed

lemma *complete-if*:

fixes $P Q :: ('lvar, 'lval, 'pvar, 'pval)$ *state hyperassertion*

assumes $\wedge P1 Q1 :: ('lvar, 'lval, 'pvar, 'pval)$ *state hyperassertion. complete* $P1$
 $C1 Q1$

and $\wedge P2 Q2 :: ('lvar, 'lval, 'pvar, 'pval)$ *state hyperassertion. complete* $P2$
 $C2 Q2$

shows *complete* P (*If* $C1$ $C2$) Q

proof (*rule completeI*)

assume *asm0*: $\models \{P\}$ *If* $C1$ $C2$ $\{Q\}$

show $\vdash \{P\}$ *stmt.If* $C1$ $C2$ $\{Q\}$

proof (*rule RuleCons*)

show $\vdash \{exists (\lambda V S. P S \wedge S = V)\}$ *stmt.If* $C1$ $C2$ $\{exists (\lambda V. join (\lambda S. S = \text{sem } C1 V \wedge P V) (\lambda S. S = \text{sem } C2 V))\}$

proof (*rule RuleExistsSet*)

fix V

show $\vdash \{(\lambda S. P S \wedge S = V)\}$ *stmt.If* $C1$ $C2$ $\{join (\lambda S. S = \text{sem } C1 V \wedge P V) (\lambda S. S = \text{sem } C2 V)\}$

proof (*rule RuleIf*)

show $\vdash \{(\lambda S. P S \wedge S = V)\}$ $C1$ $\{\lambda S. S = \text{sem } C1 V \wedge P V\}$

by (*simp add: assms(1) completeE hyper-hoare-triple-def*)

show $\vdash \{(\lambda S. P S \wedge S = V)\}$ $C2$ $\{\lambda S. S = \text{sem } C2 V\}$

by (*simp add: assms(2) completeE hyper-hoare-triple-def*)

qed

qed

show *entails* P (*exists* $(\lambda V S. P S \wedge S = V)$)

by (*simp add: entailsI exists-def*)

show *entails* (*exists* $(\lambda V. join (\lambda S. S = \text{sem } C1 V \wedge P V) (\lambda S. S = \text{sem } C2 V)))$ Q

proof (*rule entailsI*)

fix S **assume** *exists* $(\lambda V. join (\lambda S. S = \text{sem } C1 V \wedge P V) (\lambda S. S = \text{sem } C2 V))$ S

then obtain V **where** *join* $(\lambda S. S = \text{sem } C1 V \wedge P V) (\lambda S. S = \text{sem } C2 V)$ S

by (*meson exists-def*)

then obtain $S1 S2$ **where** $S = S1 \cup S2$ $S1 = \text{sem } C1 V \wedge P V$ $S2 = \text{sem } C2 V$

by (*simp add: join-def*)

then show $Q\ S$
by (*metis asm0 hyper-hoare-tripleE sem-if*)
qed
qed
qed

lemma *complete-seq-aux*:
assumes *hyper-hoare-triple* $A\ (Seq\ C1\ C2)\ B$
shows $\exists R. \text{hyper-hoare-triple } A\ C1\ R \wedge \text{hyper-hoare-triple } R\ C2\ B$
proof –
let $?R = \lambda S. \exists S'. A\ S' \wedge S = \text{sem } C1\ S'$
have *hyper-hoare-triple* $A\ C1\ ?R$
using *hyper-hoare-triple-def* **by** *blast*
moreover have *hyper-hoare-triple* $?R\ C2\ B$
proof (*rule hyper-hoare-tripleI*)
fix S **assume** $\exists S'. A\ S' \wedge S = \text{sem } C1\ S'$
then obtain S' **where** *asm0*: $A\ S'\ S = \text{sem } C1\ S'$
by *blast*
then show $B\ (\text{sem } C2\ S)$
by (*metis assms hyper-hoare-tripleE sem-seq*)
qed
ultimately show *?thesis* **by** *blast*
qed

lemma *complete-assume*:
complete $P\ (Assume\ b)\ Q$
proof (*rule completeI*)
assume *asm0*: $\models \{P\}\ Assume\ b\ \{Q\}$
show $\vdash \{P\}\ Assume\ b\ \{Q\}$
proof (*rule RuleCons*)
show $\vdash \{(\lambda S. Q\ (\text{Set.filter } (b \circ \text{snd})\ S))\}\ (Assume\ b)\ \{Q\}$
by (*simp add: RuleAssume del: Set.filter-eq*)
show *entails* $P\ (\lambda S. Q\ (\text{Set.filter } (b \circ \text{snd})\ S))$
by (*metis (mono-tags, lifting) asm0 assume-sem entails-def hyper-hoare-tripleE*)
show *entails* $Q\ Q$
by (*simp add: entailsI*)
qed
qed

lemma *complete-skip*:
complete $P\ Skip\ Q$
using *completeI RuleSkip*
by (*metis (mono-tags, lifting) entails-def hyper-hoare-triple-def sem-skip RuleCons*)

lemma *complete-assign*:
complete $P\ (Assign\ x\ e)\ Q$
proof (*rule completeI*)

```

assume asm0:  $\models \{P\} \text{Assign } x \ e \ \{Q\}$ 
show  $\vdash \{P\} \text{Assign } x \ e \ \{Q\}$ 
proof (rule RuleCons)
  show  $\vdash \{(\lambda S. Q \{(l, \sigma(x := e \ \sigma)) \mid l \ \sigma. (l, \sigma) \in S\})\} \text{Assign } x \ e \ \{Q\}$ 
    by (simp add: RuleAssign)
  show entails P  $(\lambda S. Q \{(l, \sigma(x := e \ \sigma)) \mid l \ \sigma. (l, \sigma) \in S\})$ 
  proof (rule entailsI)
    fix S assume P S
    then show  $Q \{(l, \sigma(x := e \ \sigma)) \mid l \ \sigma. (l, \sigma) \in S\}$ 
      by (metis asm0 hyper-hoare-triple-def sem-assign)
    qed
  show entails Q Q
    by (simp add: entailsI)
  qed
qed

```

```

lemma complete-havoc:
  complete P (Havoc x) Q
proof (rule completeI)
  assume asm0:  $\models \{P\} \text{Havoc } x \ \{Q\}$ 
  show  $\vdash \{P\} \text{Havoc } x \ \{Q\}$ 
  proof (rule RuleCons)
    show  $\vdash \{(\lambda S. Q \{(l, \sigma(x := v)) \mid l \ \sigma \ v. (l, \sigma) \in S\})\} (\text{Havoc } x) \ \{Q\}$ 
      using RuleHavoc by fast
    show entails P  $(\lambda S. Q \{(l, \sigma(x := v)) \mid l \ \sigma \ v. (l, \sigma) \in S\})$ 
    proof (rule entailsI)
      fix S assume P S
      then show  $Q \{(l, \sigma(x := v)) \mid l \ \sigma \ v. (l, \sigma) \in S\}$ 
        by (metis asm0 hyper-hoare-triple-def sem-havoc)
      qed
    show entails Q Q
      by (simp add: entailsI)
    qed
  qed
qed

```

```

lemma complete-seq:
  assumes  $\bigwedge R. \text{complete } P \ C1 \ R$ 
  and  $\bigwedge R. \text{complete } R \ C2 \ Q$ 
  shows complete P (Seq C1 C2) Q
  by (meson RuleSeq assms(1) assms(2) completeE completeI complete-seq-aux)

```

```

fun construct-inv
  where
    construct-inv P C 0 = P
  | construct-inv P C (Suc n) =  $(\lambda S. (\exists S'. S = \text{sem } C \ S' \wedge \text{construct-inv } P \ C \ n \ S'))$ 

```

lemma *iterate-sem-ind*:
assumes *construct-inv* $P\ C\ n\ S'$
shows $\exists S. P\ S \wedge S' = \text{iterate-sem}\ n\ C\ S$
using *assms*
by (*induct n arbitrary: S'*) (*auto*)

lemma *complete-while-aux*:
assumes *hyper-hoare-triple* $(\lambda S. P\ S \wedge S = V)\ (While\ C)\ Q$
shows *entails* (*natural-partition* (*construct-inv* $(\lambda S. P\ S \wedge S = V)\ C$)) Q
proof (*rule entailsI*)
fix S **assume** *natural-partition* (*construct-inv* $(\lambda S. P\ S \wedge S = V)\ C$) S

then obtain F **where** *asm0*: $S = (\bigcup n. F\ n) \wedge n. \text{construct-inv}\ (\lambda S. P\ S \wedge S = V)\ C\ n\ (F\ n)$
using *natural-partitionE* **by** *blast*
then have $P\ (F\ 0) \wedge F\ 0 = V$
by (*metis* (*mono-tags*, *lifting*) *construct-inv.simps(1)*)
then have $Q\ (\bigcup n. \text{iterate-sem}\ n\ C\ (F\ 0))$
using *assms* *hyper-hoare-triple-def*[*of* $\lambda S. P\ S \wedge S = V\ While\ C\ Q$] *sem-while*
by *metis*
moreover have $\bigwedge n. F\ n = \text{iterate-sem}\ n\ C\ V$
proof –
fix n
obtain S' **where** $P\ S' \wedge S' = V\ F\ n = \text{iterate-sem}\ n\ C\ S'$
using *asm0(2)* *iterate-sem-ind* **by** *blast*
then show $F\ n = \text{iterate-sem}\ n\ C\ V$
by *simp*
qed
ultimately show $Q\ S$
using *asm0(1)* **by** *auto*
qed

lemma *complete-while*:
fixes $P\ Q :: ('lvar, 'lval, 'pvar, 'pval)\ \text{state hyperassertion}$
assumes $\bigwedge P'\ Q' :: ('lvar, 'lval, 'pvar, 'pval)\ \text{state hyperassertion. complete}\ P'\ C\ Q'$
shows *complete* $P\ (While\ C)\ Q$
proof (*rule completeI*)
assume *asm0*: *hyper-hoare-triple* $P\ (While\ C)\ Q$

let $?I = \lambda V. \text{construct-inv}\ (\lambda S. P\ S \wedge S = V)\ C$

have $r: \bigwedge V. \text{syntactic-HHT}\ (?I\ V\ 0)\ (While\ C)\ (\text{natural-partition}\ (?I\ V))$
proof (*rule RuleWhile*)
fix $V\ n$ **show** *syntactic-HHT* (*construct-inv* $(\lambda S. P\ S \wedge S = V)\ C\ n$) C
(*construct-inv* $(\lambda S. P\ S \wedge S = V)\ C\ (Suc\ n)$)
by (*meson* *assms* *completeE* *construct-inv.simps(2)* *hyper-hoare-tripleI*)
qed

```

show syntactic-HHT  $P$  (While  $C$ )  $Q$ 
proof (rule RuleCons)
  show syntactic-HHT (exists  $(\lambda V. ?I V 0)$ ) (While  $C$ ) (exists  $(\lambda V. ((\textit{natural-partition}$ 
( $?I V))))$ )
    using  $r$  by (rule RuleExistsSet)
    show entails  $P$  (exists  $(\lambda V. \textit{construct-inv} (\lambda S. P S \wedge S = V) C 0)$ )
      by (simp add: entailsI exists-def)
    show entails (exists  $(\lambda V. \textit{natural-partition} (\textit{construct-inv} (\lambda S. P S \wedge S = V)$ 
 $C))$ )  $Q$ 
    proof (rule entailsI)
      fix  $S'$  assume exists  $(\lambda V. \textit{natural-partition} (\textit{construct-inv} (\lambda S. P S \wedge S =$ 
 $V) C)) S'$ 
      then obtain  $V$  where natural-partition  $(\textit{construct-inv} (\lambda S. P S \wedge S = V)$ 
 $C) S'$ 
        by (meson exists-def)
      moreover have entails  $(\textit{natural-partition} (\textit{construct-inv} (\lambda S. P S \wedge S = V)$ 
 $C)) Q$ 
        proof (rule complete-while-aux)
          show hyper-hoare-triple  $(\lambda S. P S \wedge S = V)$  (While  $C$ )  $Q$ 
            using asm0 hyper-hoare-triple-def[of  $\lambda S. P S \wedge S = V$ ]
              hyper-hoare-triple-def[of  $P$  While  $C$   $Q$ ] by auto
          qed
          ultimately show  $Q S'$ 
            by (simp add: entails-def)
        qed
      qed
    qed
  qed

```

Theorem 2

```

theorem completeness:
  fixes  $P Q :: ('lvar, 'lval, 'pvar, 'pval)$  state hyperassertion
  assumes  $\models \{P\} C \{Q\}$ 
  shows  $\vdash \{P\} C \{Q\}$ 
  using assms
proof (induct C arbitrary: P Q)
  case (Assign  $x1 x2$ )
  then show ?case
    using completeE complete-assign by fast
next
  case (Seq  $C1 C2$ )
  then show ?case
    using complete-def complete-seq by meson
next
  case (If  $C1 C2$ )
  then show ?case
    using complete-def complete-if by meson
next
  case Skip

```

```

then show ?case
  using complete-def complete-skip by meson
next
  case (Havoc x)
  then show ?case
    by (simp add: completeE complete-havoc)
next
  case (Assume b)
  then show ?case
    by (simp add: completeE complete-assume)
next
  case (While C)
  then show ?case
    using complete-def complete-while by blast
qed

```

2.4 Disproving Hyper-Triples

definition *sat* **where** $\text{sat } P \longleftrightarrow (\exists S. P \ S)$

Theorem 5

theorem *disproving-triple*:

$\neg \models \{P\} \ C \ \{Q\} \longleftrightarrow (\exists P'. \text{sat } P' \wedge \text{entails } P' \ P \wedge \models \{P'\} \ C \ \{\lambda S. \neg \ Q \ S\})$ (is
 $?A \longleftrightarrow ?B$)

proof

```

assume  $\neg \models \{P\} \ C \ \{Q\}$ 
then obtain  $S$  where  $\text{asm0}: P \ S \ \neg \ Q$  (sem C S)
  using hyper-hoare-triple-def by blast
let  $?P = \lambda S'. S = S'$ 
have  $\text{entails } ?P \ P$ 
  by (simp add: asm0(1) entails-def)
moreover have  $\models \{?P\} \ C \ \{\lambda S. \neg \ Q \ S\}$ 
  by (simp add: asm0(2) hyper-hoare-triple-def)
moreover have  $\text{sat } ?P$ 
  by (simp add: sat-def)
ultimately show  $?B$  by blast
next
assume  $\exists P'. \text{sat } P' \wedge \text{entails } P' \ P \wedge \models \{P'\} \ C \ \{\lambda S. \neg \ Q \ S\}$ 
then obtain  $P'$  where  $\text{asm0}: \text{sat } P' \ \text{entails } P' \ P \ \models \{P'\} \ C \ \{\lambda S. \neg \ Q \ S\}$ 
  by blast
then obtain  $S$  where  $P' \ S$ 
  by (meson sat-def)
then show  $?A$ 
  using asm0(2) asm0(3) entailsE hyper-hoare-tripleE
  by (metis (no-types, lifting))
qed

```

definition *differ-only-by* **where**

$\text{differ-only-by } a \ b \ x \longleftrightarrow (\forall y. y \neq x \longrightarrow a \ y = b \ y)$

lemma *differ-only-byI*:
assumes $\bigwedge y. y \neq x \implies a y = b y$
shows *differ-only-by a b x*
by (*simp add: assms differ-only-by-def*)

lemma *diff-by-update*:
differ-only-by (a(x := v)) a x
by (*simp add: differ-only-by-def*)

lemma *diff-by-comm*:
differ-only-by a b x \longleftrightarrow differ-only-by b a x
by (*metis (mono-tags, lifting) differ-only-by-def*)

lemma *diff-by-trans*:
assumes *differ-only-by a b x*
and *differ-only-by b c x*
shows *differ-only-by a c x*
by (*metis assms(1) assms(2) differ-only-by-def*)

definition *not-free-var-of where*
not-free-var-of P x \longleftrightarrow (\forall states states'
($\forall i. differ-only-by (fst (states i)) (fst (states' i)) x \wedge snd (states i) = snd (states'$
i))
 \longrightarrow (*states $\in P \longleftrightarrow$ states' $\in P$)*)

lemma *not-free-var-ofE*:
assumes *not-free-var-of P x*
and $\bigwedge i. differ-only-by (fst (states i)) (fst (states' i)) x$
and $\bigwedge i. snd (states i) = snd (states' i)$
and *states $\in P$*
shows *states' $\in P$*
using *not-free-var-of-def[of P x] assms by blast*

2.5 Synchronized Rule for Branching

definition *combine where*
combine from-nat x P1 P2 S \longleftrightarrow P1 (Set.filter ($\lambda\varphi. fst \varphi x = from-nat 1$) S)
 \wedge *P2 (Set.filter ($\lambda\varphi. fst \varphi x = from-nat 2$) S)*

lemma *combineI*:
assumes *P1 (Set.filter ($\lambda\varphi. fst \varphi x = from-nat 1$) S) \wedge P2 (Set.filter ($\lambda\varphi. fst$*
 $\varphi x = from-nat 2$) S)
shows *combine from-nat x P1 P2 S*
by (*simp add: assms combine-def del: Set.filter-eq*)

definition *modify-lvar-to where*

modify-lvar-to $x\ v\ \varphi = ((fst\ \varphi)(x := v),\ snd\ \varphi)$

lemma *logical-var-in-sem-same*:

assumes $\bigwedge\varphi. \varphi \in S \implies fst\ \varphi\ x = a$
and $\varphi' \in sem\ C\ S$
shows $fst\ \varphi'\ x = a$
by (*metis* *assms(1)* *assms(2)* *fst-conv in-sem*)

lemma *recover-after-sem*:

assumes $a \neq b$
and $\bigwedge\varphi. \varphi \in S1 \implies fst\ \varphi\ x = a$
and $\bigwedge\varphi. \varphi \in S2 \implies fst\ \varphi\ x = b$
shows $sem\ C\ S1 = Set.filter\ (\lambda\varphi. fst\ \varphi\ x = a)\ (sem\ C\ (S1 \cup S2))$ (**is** $?A = ?B$)

proof

have $r: sem\ C\ (S1 \cup S2) = sem\ C\ S1 \cup sem\ C\ S2$
by (*simp* *add: sem-union*)
moreover **have** $r1: \bigwedge\varphi'. \varphi' \in sem\ C\ S1 \implies fst\ \varphi'\ x = a$
by (*metis* *assms(2)* *fst-conv in-sem*)
moreover **have** $r2: \bigwedge\varphi'. \varphi' \in sem\ C\ S2 \implies fst\ \varphi'\ x = b$
by (*metis* *assms(3)* *fst-conv in-sem*)

show $?B \subseteq ?A$

proof (*rule* *subsetPairI*)

fix $l\ \sigma$

assume $(l, \sigma) \in Set.filter\ (\lambda\varphi. fst\ \varphi\ x = a)\ (sem\ C\ (S1 \cup S2))$

then show $(l, \sigma) \in sem\ C\ S1$

using *assms(1)* $r\ r2$ **by** *auto*

qed

show $?A \subseteq ?B$

by (*simp* *add: r r1 subsetI*)

qed

lemma *injective-then-ok*:

assumes $a \neq b$
and $S1' = (modify-lvar-to\ x\ a)\ `S1$
and $S2' = (modify-lvar-to\ x\ b)\ `S2$
shows $Set.filter\ (\lambda\varphi. fst\ \varphi\ x = a)\ (S1' \cup S2') = S1'$ (**is** $?A = ?B$)

proof

show $?B \subseteq ?A$

proof (*rule* *subsetI*)

fix y **assume** $y \in S1'$

then have $fst\ y\ x = a$ **using** *modify-lvar-to-def* *assms(2)*

by (*metis* (*mono-tags*, *lifting*) *fst-conv fun-upd-same image-iff*)

then show $y \in Set.filter\ (\lambda\varphi. fst\ \varphi\ x = a)\ (S1' \cup S2')$

by (*simp* *add: <y ∈ S1'>*)

qed

show $?A \subseteq ?B$

proof

fix y **assume** $y \in ?A$

```

then have  $y \notin S2'$ 
  using  $assms(1) assms(3)$ 
  by  $(auto simp add: modify-lvar-to-def)$ 
then show  $y \in ?B$ 
  using  $\langle y \in Set.filter (\lambda\varphi. fst \varphi x = a) (S1' \cup S2') \rangle$  by  $auto$ 
qed
qed

```

definition *not-free-var-hyper* **where**
 $not\text{-free}\text{-var}\text{-hyper} x P \longleftrightarrow (\forall S v. P S \longleftrightarrow P ((modify\text{-lvar}\text{-to} x v) ' S))$

definition *injective* **where**
 $injective f \longleftrightarrow (\forall a b. a \neq b \longrightarrow f a \neq f b)$

lemma *sem-of-modify-lvar*:
 $sem C ((modify\text{-lvar}\text{-to} r v) ' S) = (modify\text{-lvar}\text{-to} r v) ' (sem C S)$ **(is ?A = ?B)**

```

proof
  show  $?A \subseteq ?B$ 
  proof  $(rule subsetI)$ 
    fix  $y$  assume  $asm0: y \in ?A$ 
    then obtain  $x$  where  $x \in (modify\text{-lvar}\text{-to} r v) ' S$   $single\text{-sem} C (snd x) (snd y)$   $fst x = fst y$ 
      by  $(metis fst\text{-conv in}\text{-sem snd}\text{-conv})$ 
    then obtain  $xx$  where  $xx \in S$   $x = modify\text{-lvar}\text{-to} r v xx$ 
      by  $blast$ 
    then have  $(fst xx, snd y) \in sem C S$ 
      by  $(metis \langle C, snd x \rangle \rightarrow snd y \rangle fst\text{-conv in}\text{-sem modify}\text{-lvar}\text{-to}\text{-def prod.collapse snd}\text{-conv})$ 
    then show  $y \in ?B$ 
      by  $(metis \langle fst x = fst y \rangle \langle x = modify\text{-lvar}\text{-to} r v xx \rangle fst\text{-eqD modify}\text{-lvar}\text{-to}\text{-def prod.exhaust}\text{-sel rev}\text{-image}\text{-eqI snd}\text{-eqD})$ 
  qed
  show  $?B \subseteq ?A$ 
  proof  $(rule subsetI)$ 
    fix  $y$  assume  $y \in modify\text{-lvar}\text{-to} r v ' sem C S$ 
    then obtain  $yy$  where  $y = modify\text{-lvar}\text{-to} r v yy$   $yy \in sem C S$ 
      by  $blast$ 
    then obtain  $x$  where  $x \in S$   $fst x = fst yy$   $single\text{-sem} C (snd x) (snd yy)$ 
      by  $(metis fst\text{-conv in}\text{-sem snd}\text{-conv})$ 
    then have  $fst (modify\text{-lvar}\text{-to} r v x) = fst y$ 
      by  $(simp add: \langle y = modify\text{-lvar}\text{-to} r v yy \rangle modify\text{-lvar}\text{-to}\text{-def})$ 
    then show  $y \in sem C (modify\text{-lvar}\text{-to} r v ' S)$ 
      by  $(metis (mono\text{-tags, lifting}) \langle \langle C, snd x \rangle \rightarrow snd yy \rangle \langle x \in S \rangle \langle y = modify\text{-lvar}\text{-to} r v yy \rangle fst\text{-conv image}\text{-eqI in}\text{-sem modify}\text{-lvar}\text{-to}\text{-def snd}\text{-conv})$ 
  qed
qed

```

end

3 Expressivity of Hyper Hoare Logic

In this file, we define program hyperproperties (definition 8), and prove theorems 3 and 4.

3.1 Program Hyperproperties

```
theory ProgramHyperproperties
  imports Logic
begin
```

Definition 8

```
type-synonym 'a hyperproperty = ('a × 'a) set ⇒ bool
type-synonym ('pvar, 'pval) program-hyperproperty = ('pvar, 'pval) pstate hyperproperty
```

```
definition set-of-traces where
  set-of-traces C = { (σ, σ') | σ σ'. ⟨C, σ⟩ → σ' }
```

```
definition hypersat :: ('pvar, 'pval) stmt ⇒ ('pvar, 'pval) program-hyperproperty
  ⇒ bool where
  hypersat C H ⟷ H (set-of-traces C)
```

```
definition copy-p-state where
  copy-p-state to-pvar to-lval σ x = to-lval (σ (to-pvar x))
```

```
definition recover-p-state where
  recover-p-state to-pval to-lvar l x = to-pval (l (to-lvar x))
```

```
lemma injective-then-exists-inverse:
  assumes injective to-lvar
  shows ∃ to-pvar. (∀ x. to-pvar (to-lvar x) = x)
proof –
  let ?to-pvar = λy. SOME x. to-lvar x = y
  have ∧x. ?to-pvar (to-lvar x) = x
  by (metis (mono-tags, lifting) assms injective-def someI)
  then show ?thesis
  by force
qed
```

```
lemma single-step-then-in-sem:
  assumes single-sem C σ σ'
  and (l, σ) ∈ S
  shows (l, σ') ∈ sem C S
  using assms(1) assms(2) in-sem by fastforce
```

```
lemma in-set-of-traces:
  (σ, σ') ∈ set-of-traces C ⟷ ⟨C, σ⟩ → σ'
```

by (*simp add: set-of-traces-def*)

lemma *in-set-of-traces-then-in-sem:*

assumes $(\sigma, \sigma') \in \text{set-of-traces } C$

and $(l, \sigma) \in S$

shows $(l, \sigma') \in \text{sem } C S$

using *in-set-of-traces assms single-step-then-in-sem* **by** *metis*

lemma *set-of-traces-same:*

assumes $\bigwedge x. \text{to-pvar } (\text{to-lvar } x) = x$

and $\bigwedge x. \text{to-pval } (\text{to-lval } x) = x$

and $S = \{(\text{copy-p-state to-pvar to-lval } \sigma, \sigma) \mid \sigma. \text{True}\}$

shows $\{(\text{recover-p-state to-pval to-lvar } l, \sigma') \mid l \sigma'. (l, \sigma') \in \text{sem } C S\} = \text{set-of-traces } C$

(is ?A = ?B)

proof

show $?A \subseteq ?B$

proof (*rule subsetPairI*)

fix $\sigma \sigma'$ **assume** *asm0*: $(\sigma, \sigma') \in \{(\text{recover-p-state to-pval to-lvar } l, \sigma') \mid l \sigma'. (l, \sigma') \in \text{sem } C S\}$

then obtain l **where** $\sigma = \text{recover-p-state to-pval to-lvar } l$ $(l, \sigma') \in \text{sem } C S$

by *blast*

then obtain x **where** $(l, x) \in S$ $\langle C, x \rangle \rightarrow \sigma'$

by (*metis fst-conv in-sem snd-conv*)

then have $l = \text{copy-p-state to-pvar to-lval } x$

using *assms(3)* **by** *blast*

moreover have $\sigma = x$

proof (*rule ext*)

fix y **show** $\sigma y = x y$

by (*simp add: \langle \sigma = recover-p-state to-pval to-lvar l \rangle assms(1) assms(2)*)

calculation copy-p-state-def recover-p-state-def

qed

ultimately show $(\sigma, \sigma') \in \text{set-of-traces } C$

by (*simp add: \langle C, x \rangle \rightarrow \sigma' \rangle set-of-traces-def*)

qed

show $?B \subseteq ?A$

proof (*rule subsetPairI*)

fix $\sigma \sigma'$ **assume** *asm0*: $(\sigma, \sigma') \in \text{set-of-traces } C$

let $?l = \text{copy-p-state to-pvar to-lval } \sigma$

have $(?l, \sigma) \in S$

using *assms(3)* **by** *blast*

then have $(?l, \sigma') \in \text{sem } C S$

using *asm0 in-set-of-traces-then-in-sem* **by** *blast*

moreover have $\text{recover-p-state to-pval to-lvar } ?l = \sigma$

proof (*rule ext*)

fix x **show** $\text{recover-p-state to-pval to-lvar } (\text{copy-p-state to-pvar to-lval } \sigma) x = \sigma x$

by (*simp add: assms(1) assms(2) copy-p-state-def recover-p-state-def*)

qed
ultimately show $(\sigma, \sigma') \in \{(recover\text{-}p\text{-}state\ to\text{-}pval\ to\text{-}lvar\ l, \sigma') \mid l\ \sigma'. (l, \sigma') \in sem\ C\ S\}$
by force
qed
qed

Theorem 3

theorem *proving-hyperproperties:*

fixes *to-lvar* :: *'pvar* \Rightarrow *'lvar*
fixes *to-lval* :: *'pval* \Rightarrow *'lval*

assumes *injective to-lvar*
and *injective to-lval*

shows $\exists P\ Q::('lvar, 'lval, 'pvar, 'pval)\ state\ hyperassertion. (\forall C. hypersat\ C\ H \longleftrightarrow \models \{P\}\ C\ \{Q\})$

proof –

obtain *to-pval* :: *'lval* \Rightarrow *'pval* **where** *r1*: $\bigwedge x. to\text{-}pval\ (to\text{-}lval\ x) = x$
using *assms(2)* *injective-then-exists-inverse* **by** *blast*

obtain *to-pvar* :: *'lvar* \Rightarrow *'pvar* **where** *r2*: $\bigwedge x. to\text{-}pvar\ (to\text{-}lvar\ x) = x$
using *assms(1)* *injective-then-exists-inverse* **by** *blast*

let *?P* = $\lambda S. S = \{(copy\text{-}p\text{-}state\ to\text{-}pvar\ to\text{-}lval\ \sigma, \sigma) \mid \sigma. True\}$
let *?Q* = $\lambda S. H\ \{(recover\text{-}p\text{-}state\ to\text{-}pval\ to\text{-}lvar\ l, \sigma') \mid l\ \sigma'. (l, \sigma') \in S\}$

have $\bigwedge C. hypersat\ C\ H \longleftrightarrow \models \{?P\}\ C\ \{?Q\}$

proof

fix *C*

assume *hypersat C H*

show $\models \{?P\}\ C\ \{?Q\}$

proof (*rule hyper-hoare-tripleI*)

fix *S* **assume** $S = \{(copy\text{-}p\text{-}state\ to\text{-}pvar\ to\text{-}lval\ \sigma, \sigma) \mid \sigma. True\}$

have $\{(recover\text{-}p\text{-}state\ to\text{-}pval\ to\text{-}lvar\ l, \sigma') \mid l\ \sigma'. (l, \sigma') \in sem\ C\ S\}$

= *set-of-traces C*

using $\langle S = \{(copy\text{-}p\text{-}state\ to\text{-}pvar\ to\text{-}lval\ \sigma, \sigma) \mid \sigma. True\} \rangle\ set\text{-of}\text{-traces}\text{-same}[of$

to-pvar to-lvar to-pval to-lval]

r1 r2 **by** *presburger*

then show $H\ \{(recover\text{-}p\text{-}state\ to\text{-}pval\ to\text{-}lvar\ l, \sigma') \mid l\ \sigma'. (l, \sigma') \in sem\ C\ S\}$

using $\langle hypersat\ C\ H \rangle\ hypersat\text{-}def$ **by** *metis*

qed

next

fix *C*

let *?S* = $\{(copy\text{-}p\text{-}state\ to\text{-}pvar\ to\text{-}lval\ \sigma, \sigma) \mid \sigma. True\}$

assume $\models \{?P\}\ C\ \{?Q\}$

then have *?Q (sem C ?S)*

by (simp add: hyper-hoare-triple-def)
moreover have $\{(recover-p-state\ to-pval\ to-lvar\ l,\ \sigma')\ |\ l\ \sigma'.\ (l,\ \sigma') \in sem\ C\ ?S\} = set-of-traces\ C$
 using r1 r2 set-of-traces-same[of to-pvar to-lvar to-pval to-lval]
 by presburger
ultimately show hypersat C H
 by (simp add: hypersat-def)
qed
then show ?thesis
 by auto
qed

Hypersafety, hyperliveness

definition max-k where

max-k k S \longleftrightarrow finite S \wedge card S \leq k

definition hypersafety where

hypersafety P \longleftrightarrow $(\forall S. \neg P\ S \longrightarrow (\forall S'. S \subseteq S' \longrightarrow \neg P\ S'))$

definition k-hypersafety where

k-hypersafety k P \longleftrightarrow $(\forall S. \neg P\ S \longrightarrow (\exists S'. S' \subseteq S \wedge max-k\ k\ S' \wedge (\forall S''. S' \subseteq S'' \longrightarrow \neg P\ S'')))$

definition hyperliveness where

hyperliveness P \longleftrightarrow $(\forall S. \exists S'. S \subseteq S' \wedge P\ S')$

lemma k-hypersafetyI:

assumes $\bigwedge S. \neg P\ S \implies \exists S'. S' \subseteq S \wedge max-k\ k\ S' \wedge (\forall S''. S' \subseteq S'' \longrightarrow \neg P\ S'')$

shows k-hypersafety k P

by (simp add: assms k-hypersafety-def)

lemma hypersafetyI:

assumes $\bigwedge S\ S'. \neg P\ S \implies S \subseteq S' \implies \neg P\ S'$

shows hypersafety P

by (metis assms hypersafety-def)

lemma hyperlivenessI:

assumes $\bigwedge S. \exists S'. S \subseteq S' \wedge P\ S'$

shows hyperliveness P

using assms hyperliveness-def **by** blast

lemma k-hypersafe-is-hypersafe:

assumes k-hypersafety k P

shows hypersafety P

by (metis (full-types) assms dual-order.trans hypersafety-def k-hypersafety-def)

lemma *one-safety-equiv*:
assumes *sat H*
shows $k\text{-hypersafety } 1 H \longleftrightarrow (\exists P. \forall S. H S \longleftrightarrow (\forall \tau \in S. P \tau))$ (**is** $?A \longleftrightarrow ?B$)
proof
assume $?B$
then obtain P **where** $asm0: \bigwedge S. H S \longleftrightarrow (\forall \tau \in S. P \tau)$
by *auto*
show $?A$
proof (*rule k-hypersafetyI*)
fix S
assume $asm1: \neg H S$
then obtain τ **where** $\tau \in S \wedge \neg P \tau$
using $asm0$ **by** *blast*
let $?S = \{\tau\}$
have $?S \subseteq S \wedge \text{max-k } 1 ?S \wedge (\forall S''. ?S \subseteq S'' \longrightarrow \neg H S'')$
using $\langle \neg P \tau \rangle \langle \tau \in S \rangle$ $asm0$ *max-k-def* **by** *fastforce*
then show $\exists S' \subseteq S. \text{max-k } 1 S' \wedge (\forall S''. S' \subseteq S'' \longrightarrow \neg H S'')$ **by** *blast*
qed
next
assume $?A$
let $?P = \lambda \tau. H \{\tau\}$
have $\bigwedge S. H S \longleftrightarrow (\forall \tau \in S. ?P \tau)$
proof
fix S **assume** $H S$
then show $\forall \tau \in S. ?P \tau$
using $\langle k\text{-hypersafety } 1 H \rangle$ *hypersafety-def k-hypersafe-is-hypersafe* **by** *auto*
next
fix S **assume** $asm0: \forall \tau \in S. ?P \tau$
show $H S$
proof (*rule ccontr*)
assume $\neg H S$
then obtain S' **where** $S' \subseteq S \wedge \text{max-k } 1 S' \wedge (\forall S''. S' \subseteq S'' \longrightarrow \neg H S'')$
by (*metis* $\langle k\text{-hypersafety } 1 H \rangle$ *k-hypersafety-def*)
then show *False*
proof (*cases* $S' = \{\}$)
case *True*
then show $?thesis$
by (*metis* $\langle S' \subseteq S \wedge \text{max-k } 1 S' \wedge (\forall S''. S' \subseteq S'' \longrightarrow \neg H S'') \rangle$ *assms empty-subsetI sat-def*)
next
case *False*
then obtain τ **where** $\tau \in S'$
by *blast*
then have $\text{card } S' = 1$
by (*metis* *False One-nat-def Suc-leI* $\langle S' \subseteq S \wedge \text{max-k } 1 S' \wedge (\forall S''. S' \subseteq S'' \longrightarrow \neg H S'') \rangle$ *card-gt-0-iff le-antisym max-k-def*)
then have $S' = \{\tau\}$
using $\langle \tau \in S' \rangle$ *card-1-singletonE* **by** *auto*

```

    then show ?thesis
      using ⟨S' ⊆ S ∧ max-k 1 S' ∧ (∀ S''. S' ⊆ S'' ⟶ ¬ H S'')⟩ asm0 by
fastforce
    qed
  qed
  then show ?B by blast
qed

```

definition *hoarify* where

hoarify P Q S \longleftrightarrow $(\forall p \in S. \text{fst } p \in P \longrightarrow \text{snd } p \in Q)$

lemma *hoarify-hypersafety*:

hypersafety (*hoarify* P Q)

by (*metis* (*no-types*, *opaque-lifting*) *hoarify-def hypersafetyI subsetD*)

theorem *hypersafety-1-hoare-logic*:

k-hypersafety 1 (*hoarify* P Q)

proof (*rule k-hypersafetyI*)

fix S assume ¬ *hoarify* P Q S

then obtain τ where τ ∈ S fst τ ∈ P snd τ ∉ Q

using *hoarify-def* by blast

let ?S = {τ}

have ?S ⊆ S ∧ max-k 1 ?S ∧ (∀ S''. ?S ⊆ S'' ⟶ ¬ *hoarify* P Q S'')

by (*metis* *Compl-iff One-nat-def* ⟨τ ∈ S⟩ ⟨fst τ ∈ P⟩ ⟨snd τ ∉ Q⟩ *card.empty card.insert compl-le-compl-iff empty-not-insert finite.intros(1) finite.intros(2) hoarify-def insert-absorb le-numeral-extra(4) max-k-def subset-Compl-singleton*)

then show ∃ S' ⊆ S. max-k 1 S' ∧ (∀ S''. S' ⊆ S'' ⟶ ¬ *hoarify* P Q S')

by *meson*

qed

definition *incorrectnessify* where

incorrectnessify P Q S \longleftrightarrow $(\forall \sigma' \in Q. \exists \sigma \in P. (\sigma, \sigma') \in S)$

lemma *incorrectnessify-liveness*:

assumes P ≠ {}

shows *hyperliveness* (*incorrectnessify* P Q)

proof (*rule hyperlivenessI*)

fix S

obtain σ where *asm0*: σ ∈ P

using *assms* by blast

let ?S = S ∪ {(σ, σ') | σ'. σ' ∈ Q}

have *incorrectnessify* P Q ?S

using *asm0 incorrectnessify-def* by force

then show ∃ S'. S ⊆ S' ∧ *incorrectnessify* P Q S'

using *sup.cobounded1* by *blast*
qed

definition *real-incorrectnessify* **where**
real-incorrectnessify $P Q S \longleftrightarrow (\forall \sigma \in P. \exists \sigma' \in Q. (\sigma, \sigma') \in S)$

lemma *real-incorrectnessify-liveness*:
assumes $Q \neq \{\}$
shows *hyperliveness* (*real-incorrectnessify* $P Q$)
by (*metis UNIV-I assms equals0I hyperliveness-def real-incorrectnessify-def subsetI*)

Verifying GNI

definition *gni-hyperassertion* $:: 'n \Rightarrow 'n \Rightarrow ('n \Rightarrow 'v)$ *hyperassertion* **where**
gni-hyperassertion $h l S \longleftrightarrow (\forall \sigma \in S. \forall v. \exists \sigma' \in S. \sigma' h = v \wedge \sigma l = \sigma' l)$

definition *semify* **where**
semify $\Sigma S = \{ (l, \sigma') \mid \sigma' \sigma l. (l, \sigma) \in S \wedge (\sigma, \sigma') \in \Sigma \}$

definition *hyperprop-hht* **where**
hyperprop-hht $P Q \Sigma \longleftrightarrow (\forall S. P S \longrightarrow Q (\text{semify } \Sigma S))$

Theorem 4

theorem *any-hht-hyperprop*:
 $\models \{P\} C \{Q\} \longleftrightarrow \text{hypersat } C (\text{hyperprop-hht } P Q)$ (**is** $?A \longleftrightarrow ?B$)

proof

have $\bigwedge S. \text{semify } (\text{set-of-traces } C) S = \text{sem } C S$

proof –

fix S

have $\bigwedge l \sigma'. (l, \sigma') \in \text{sem } C S \longleftrightarrow (l, \sigma') \in \text{semify } (\text{set-of-traces } C) S$

proof –

fix $l \sigma'$

have $(l, \sigma') \in \text{sem } C S \longleftrightarrow (\exists \sigma. (l, \sigma) \in S \wedge \langle C, \sigma \rangle \rightarrow \sigma')$

by (*simp add: in-sem*)

also have $\dots \longleftrightarrow (\exists \sigma. (l, \sigma) \in S \wedge (\sigma, \sigma') \in \text{set-of-traces } C)$

using *set-of-traces-def* **by** *fastforce*

then show $(l, \sigma') \in \text{sem } C S \longleftrightarrow (l, \sigma') \in \text{semify } (\text{set-of-traces } C) S$

by (*simp add: calculation semify-def*)

qed

then show $\text{semify } (\text{set-of-traces } C) S = \text{sem } C S$

by *auto*

qed

show $?A \Longrightarrow ?B$

by (*simp add: $\langle \bigwedge S. \text{semify } (\text{set-of-traces } C) S = \text{sem } C S \rangle$ hyper-hoare-tripleE hyperprop-hht-def hypersat-def*)

show $?B \Longrightarrow ?A$

by (*simp add: $\langle \bigwedge S. \text{semify } (\text{set-of-traces } C) S = \text{sem } C S \rangle$ hyper-hoare-triple-def hyperprop-hht-def hypersat-def*)

qed

end

In this file, we prove most results of Appendix C: hyper-triples subsume many other triples, as well as example 3.

```
theory Expressivity
  imports ProgramHyperproperties
begin
```

3.2 Hoare Logic (HL) [7]

Definition 16 **definition** *HL* **where**

$$HL\ P\ C\ Q \longleftrightarrow (\forall \sigma\ \sigma'\ l. (l, \sigma) \in P \wedge (\langle C, \sigma \rangle \rightarrow \sigma') \longrightarrow (l, \sigma') \in Q)$$

lemma *HLI*:

```
assumes  $\bigwedge \sigma\ \sigma'\ l. (l, \sigma) \in P \implies \langle C, \sigma \rangle \rightarrow \sigma' \implies (l, \sigma') \in Q$ 
shows HL P C Q
using assms HL-def by blast
```

lemma *hoarifyI*:

```
assumes  $\bigwedge \sigma\ \sigma'. (\sigma, \sigma') \in S \implies \sigma \in P \implies \sigma' \in Q$ 
shows hoarify P Q S
by (metis assms hoarify-def prod.collapse)
```

definition *HL-hyperprop* **where**

$$HL\text{-hyperprop}\ P\ Q\ S \longleftrightarrow (\forall l. \forall p \in S. (l, fst\ p) \in P \longrightarrow (l, snd\ p) \in Q)$$

lemma *connection-HL*:

$$HL\ P\ C\ Q \longleftrightarrow HL\text{-hyperprop}\ P\ Q\ (\text{set-of-traces}\ C)\ (\text{is}\ ?A \longleftrightarrow ?B)$$

proof

```
assume ?A
then show ?B
  by (simp add: HL-def HL-hyperprop-def set-of-traces-def)
```

next

```
assume ?B
show ?A
proof (rule HLI)
  fix  $\sigma\ \sigma'\ l$  assume asm0:  $(l, \sigma) \in P \ \langle C, \sigma \rangle \rightarrow \sigma'$ 
  then have  $(\sigma, \sigma') \in \text{set-of-traces}\ C$ 
    by (simp add: set-of-traces-def)
  then show  $(l, \sigma') \in Q$ 
```

```
    using  $\langle HL\text{-hyperprop}\ P\ Q\ (\text{set-of-traces}\ C) \rangle$  asm0(1) HL-hyperprop-def by
fastforce
```

qed

qed

Proposition 1 **theorem** *HL-expresses-hyperproperties*:

$$\exists H. (\forall C. \text{hypersat}\ C\ H \longleftrightarrow HL\ P\ C\ Q) \wedge k\text{-hypersafety}\ 1\ H$$

```

proof –
  let ?H = HL-hyperprop P Q
  have  $\bigwedge C. \text{hypersat } C \text{ ?H} \longleftrightarrow \text{HL } P \ C \ Q$ 
    by (simp add: connection-HL hypersat-def)
  moreover have k-hypersafety 1 ?H
  proof (rule k-hypersafetyI)
    fix S assume asm0:  $\neg \text{HL-hyperprop } P \ Q \ S$ 
    then obtain l p where  $p \in S \ (l, \text{fst } p) \in P \ (l, \text{snd } p) \notin Q$ 
      using HL-hyperprop-def by blast
    let ?S = {p}
    have max-k 1 ?S  $\wedge (\forall S''. ?S \subseteq S'' \longrightarrow \neg \text{HL-hyperprop } P \ Q \ S'')$ 
      by (metis (no-types, lifting) One-nat-def  $\langle (l, \text{fst } p) \in P \rangle \langle (l, \text{snd } p) \notin Q \rangle$ 
        card.empty card.insert
          empty-iff finite.intros(1) finite.intros(2) le-numeral-extra(4) max-k-def
          HL-hyperprop-def singletonI subsetD)
    then show  $\exists S' \subseteq S. \text{max-k } 1 \ S' \wedge (\forall S''. S' \subseteq S'' \longrightarrow \neg \text{HL-hyperprop } P \ Q \ S'')$ 
      by (meson  $\langle p \in S \rangle$  empty-subsetI insert-subsetI)
    qed
  ultimately show ?thesis
    by blast
qed

```

Proposition 2 theorem encoding-HL:

$\text{HL } P \ C \ Q \longleftrightarrow (\text{hyper-hoare-triple } (\text{over-approx } P) \ C \ (\text{over-approx } Q)) \ (\text{is } ?A \longleftrightarrow ?B)$

proof (rule iffI)

show ?B \implies ?A

proof –

assume asm0: ?B

show ?A

proof (rule HLI)

fix $\sigma \ \sigma' \ l$

assume asm1: $(l, \sigma) \in P \ \langle C, \sigma \rangle \rightarrow \sigma'$

then have over-approx P $\{(l, \sigma)\}$

by (simp add: over-approx-def)

then have (over-approx Q) (sem C $\{(l, \sigma)\}$)

using asm0 hyper-hoare-tripleE **by** auto

then show $(l, \sigma') \in Q$

by (simp add: asm1(2) in-mono in-sem over-approx-def)

qed

qed

next

assume r: ?A

show ?B

proof (rule hyper-hoare-tripleI)

fix S **assume** asm0: over-approx P S

then have $S \subseteq P$

by (simp add: over-approx-def)

then have sem C $S \subseteq \text{sem } C \ P$

by (*simp add: sem-monotonic*)
 then have $\text{sem } C \ S \subseteq Q$
 using *r HL-def[of P C Q]*
 by (*metis (no-types, lifting) fst-conv in-mono in-sem snd-conv subrelI*)
 then show $\text{over-approx } Q \ (\text{sem } C \ S)$
 by (*simp add: over-approx-def*)
 qed
 qed

lemma *entailment-order-hoare*:
 assumes $P \subseteq P'$
 shows $\text{entails } (\text{over-approx } P) \ (\text{over-approx } P')$
 by (*simp add: assms entails-def over-approx-def subset-trans*)

3.3 Cartesian Hoare Logic (CHL) [10]

definition *k-sem where*

$k\text{-sem } C \ \text{states } \text{states}' \longleftrightarrow (\forall i. (\text{fst } (\text{states } i) = \text{fst } (\text{states}' i) \wedge \text{single-sem } C \ (\text{snd } (\text{states } i)) \ (\text{snd } (\text{states}' i))))$

lemma *k-semI*:

assumes $\bigwedge i. (\text{fst } (\text{states } i) = \text{fst } (\text{states}' i) \wedge \text{single-sem } C \ (\text{snd } (\text{states } i)) \ (\text{snd } (\text{states}' i)))$
 shows $k\text{-sem } C \ \text{states } \text{states}'$
 by (*simp add: assms k-sem-def*)

lemma *k-semE*:

assumes $k\text{-sem } C \ \text{states } \text{states}'$
 shows $\text{fst } (\text{states } i) = \text{fst } (\text{states}' i) \wedge \text{single-sem } C \ (\text{snd } (\text{states } i)) \ (\text{snd } (\text{states}' i))$
 using *assms k-sem-def* by *fastforce*

Definition 17 **definition** *CHL where*

$\text{CHL } P \ C \ Q \longleftrightarrow (\forall \text{states}. \text{states} \in P \longrightarrow (\forall \text{states}'. k\text{-sem } C \ \text{states } \text{states}' \longrightarrow \text{states}' \in Q))$

lemma *CHLI*:

assumes $\bigwedge \text{states } \text{states}'. \text{states} \in P \implies k\text{-sem } C \ \text{states } \text{states}' \implies \text{states}' \in Q$
 shows $\text{CHL } P \ C \ Q$
 by (*simp add: assms CHL-def*)

lemma *CHLE*:

assumes $\text{CHL } P \ C \ Q$
 and $\text{states} \in P$
 and $k\text{-sem } C \ \text{states } \text{states}'$
 shows $\text{states}' \in Q$
 using *assms(1) assms(2) assms(3) CHL-def* by *fast*

definition *encode-CHL where*

encode-CHL from-nat $x P S \longleftrightarrow (\forall \text{states}. (\forall i. \text{states } i \in S \wedge \text{fst } (\text{states } i) x = \text{from-nat } i) \longrightarrow \text{states} \in P)$

lemma *encode-CHLI*:

assumes $\bigwedge \text{states}. (\forall i. \text{states } i \in S \wedge \text{fst } (\text{states } i) x = \text{from-nat } i) \Longrightarrow \text{states} \in P$

shows *encode-CHL from-nat* $x P S$

using *assms(1) encode-CHL-def* **by** *force*

lemma *encode-CHLE*:

assumes *encode-CHL from-nat* $x P S$

and $\bigwedge i. \text{states } i \in S$

and $\bigwedge i. \text{fst } (\text{states } i) x = \text{from-nat } i$

shows $\text{states} \in P$

by (*metis assms(1) assms(2) assms(3) encode-CHL-def*)

lemma *equal-change-lvar*:

assumes *fst* $\varphi x = y$

shows $\varphi = ((\text{fst } \varphi)(x := y), \text{snd } \varphi)$

using *assms* **by** *fastforce*

Proposition 3 theorem *encoding-CHL*:

assumes *not-free-var-of* $P x$

and *not-free-var-of* $Q x$

and *injective from-nat*

shows *CHL* $P C Q \longleftrightarrow \models \{ \text{encode-CHL from-nat } x P \} C \{ \text{encode-CHL from-nat } x Q \}$ (**is** $?A \longleftrightarrow ?B$)

proof

assume $?A$

show $?B$

proof (*rule hyper-hoare-tripleI*)

fix S **assume** *encode-CHL from-nat* $x P S$

then obtain *asm0*: $\bigwedge \text{states } \text{states}' . (\bigwedge i. \text{states } i \in S) \Longrightarrow (\bigwedge i. \text{fst } (\text{states } i) x = \text{from-nat } i) \Longrightarrow \text{states} \in P$

by (*simp add: encode-CHLE*)

show *encode-CHL from-nat* $x Q$ (*sem* $C S$)

proof (*rule encode-CHLI*)

fix states'

assume *asm1*: $\forall i. \text{states}' i \in \text{sem } C S \wedge \text{fst } (\text{states}' i) x = \text{from-nat } i$

let $?states = \lambda i. (\text{fst } (\text{states}' i), \text{SOME } \sigma. (\text{fst } (\text{states}' i), \sigma) \in S \wedge \text{single-sem } C \sigma (\text{snd } (\text{states}' i)))$

show $\text{states}' \in Q$

using $\langle ?A \rangle$

proof (*rule CHLE*)

show $?states \in P$

```

proof (rule asm0)
  fix i
  let  $?\sigma = \text{SOME } \sigma. ((fst (states' i), \sigma) \in S \wedge \langle C, \sigma \rangle \rightarrow snd (states' i))$ 
  have  $r: (fst (states' i), ?\sigma) \in S \wedge \langle C, ?\sigma \rangle \rightarrow snd (states' i)$ 
    using someI-ex[of  $\lambda\sigma. (fst (states' i), \sigma) \in S \wedge \langle C, \sigma \rangle \rightarrow snd (states'$ 
i)] asm1 in-sem by blast
  then show  $?states\ i \in S$ 
    by blast
  show  $fst (?states\ i)\ x = from\text{-}nat\ i$ 
    by (simp add: asm1)
  qed
show k-sem C ?states states'
proof (rule k-semI)
  fix i
  let  $?\sigma = \text{SOME } \sigma. ((fst (states' i), \sigma) \in S \wedge \langle C, \sigma \rangle \rightarrow snd (states' i))$ 
  have  $r: (fst (states' i), ?\sigma) \in S \wedge \langle C, ?\sigma \rangle \rightarrow snd (states' i)$ 
    using someI-ex[of  $\lambda\sigma. (fst (states' i), \sigma) \in S \wedge \langle C, \sigma \rangle \rightarrow snd (states'$ 
i)] asm1 in-sem by blast
  then show  $fst (?states\ i) = fst (states' i) \wedge \langle C, snd (?states\ i) \rangle \rightarrow snd$ 
(states' i)
    by simp
  qed
qed
qed
qed
next
assume asm0:  $\models \{encode\text{-}CHL\ from\text{-}nat\ x\ P\} C \{encode\text{-}CHL\ from\text{-}nat\ x\ Q\}$ 

show CHL P C Q
proof (rule CHLI)
  fix states states'
  assume asm1:  $states \in P\ k\text{-}sem\ C\ states\ states'$ 

  let  $?states = \lambda i. ((fst (states\ i))(x := from\text{-}nat\ i), snd (states\ i))$ 
  let  $?states' = \lambda i. ((fst (states'\ i))(x := from\text{-}nat\ i), snd (states'\ i))$ 
  let  $?S = range\ ?states$ 

  have encode-CHL from-nat x Q (sem C ?S)
    using asm0
  proof (rule hyper-hoare-tripleE)
    show encode-CHL from-nat x P ?S
    proof (rule encode-CHLI)
      fix f assume asm2:  $\forall i. f\ i \in ?S \wedge fst (f\ i)\ x = from\text{-}nat\ i$ 
      have  $f = ?states$ 
      proof (rule ext)
        fix i
        obtain j where j-def:  $f\ i = ((fst (states\ j))(x := from\text{-}nat\ j), snd (states$ 
j))
          using asm2 by fastforce

```

```

then have from-nat j = from-nat i
  by (metis asm2 fst-conv fun-upd-same)
then show  $f i = ((fst (states i))(x := from-nat i), snd (states i))$ 
  by (metis j-def assms(3) injective-def)
qed
moreover have  $?states \in P$ 
  using assms(1)
proof (rule not-free-var-ofE)
  show  $states \in P$ 
    using asm1(1) by simp
  fix  $i$ 
    show differ-only-by ( $(fst (states i)) (fst ((fst (states i))(x := from-nat i),$ 
snd (states i))) x
      by (simp add: differ-only-by-def)
    show  $snd (states i) = snd ((fst (states i))(x := from-nat i), snd (states i))$ 
      by simp
    qed
  ultimately show  $f \in P$ 
    by meson
  qed
qed
then have  $?states' \in Q$ 
proof (rule encode-CHLE)
  fix  $i$ 
    show  $(fst ((fst (states i))(x := from-nat i), snd (states' i)) x = from-nat i$ 
      by simp
    moreover have single-sem C ( $snd (?states i)$ ) ( $snd (?states' i)$ )
      using asm1(2) k-sem-def by fastforce
    ultimately show  $((fst (states i))(x := from-nat i), snd (states' i)) \in sem C$ 
       $?S$ 
      using in-sem by fastforce
    qed
  show  $states' \in Q$ 
    using assms(2)
  proof (rule not-free-var-ofE[of Q x])
    show  $?states' \in Q$ 
      by (simp add:  $\langle(\lambda i. ((fst (states i))(x := from-nat i), snd (states' i))) \in Q\rangle$ )
    fix  $i$  show differ-only-by ( $(fst ((fst (states i))(x := from-nat i), snd (states'$ 
i))) (fst (states' i)) x
      by (metis asm1(2) diff-by-update fst-conv k-sem-def)
    qed (auto)
  qed
qed

```

definition *CHL-hyperprop* **where**

$$CHL\text{-hyperprop } P \ Q \ S \iff (\forall l \ p. (\forall i. p \ i \in S) \wedge (\lambda i. (l \ i, fst (p \ i))) \in P \longrightarrow (\lambda i. (l \ i, snd (p \ i))) \in Q)$$

lemma *CHL-hyperpropI*:

assumes $\bigwedge l p. (\forall i. p \ i \in S) \wedge (\lambda i. (l \ i, \text{fst} \ (p \ i))) \in P \implies (\lambda i. (l \ i, \text{snd} \ (p \ i))) \in Q$
shows *CHL-hyperprop* $P \ Q \ S$
by (*simp add: assms CHL-hyperprop-def*)

lemma *CHL-hyperpropE*:

assumes *CHL-hyperprop* $P \ Q \ S$
and $\bigwedge i. p \ i \in S$
and $(\lambda i. (l \ i, \text{fst} \ (p \ i))) \in P$
shows $(\lambda i. (l \ i, \text{snd} \ (p \ i))) \in Q$
using *assms(1) assms(2) assms(3) CHL-hyperprop-def* **by** *blast*

Proposition 3 **theorem** *CHL-hyperproperty*:

hypersat $C \ (CHL\text{-hyperprop} \ P \ Q) \longleftrightarrow CHL \ P \ C \ Q \ (\text{is} \ ?A \longleftrightarrow ?B)$

proof

assume $?A$

show $?B$

proof (*rule CHLI*)

fix *states states'*

assume *asm0*: $\text{states} \in P \ k\text{-sem} \ C \ \text{states} \ \text{states}'$

let $?p = \lambda i. (\text{snd} \ (\text{states} \ i), \text{snd} \ (\text{states}' \ i))$

let $?l = \lambda i. \text{fst} \ (\text{states} \ i)$

have $\text{range} \ ?p \subseteq \text{set-of-traces} \ C$

proof (*rule subsetI*)

fix x **assume** $x \in \text{range} \ ?p$

then obtain i **where** $x = (\text{snd} \ (\text{states} \ i), \text{snd} \ (\text{states}' \ i))$

by *blast*

then show $x \in \text{set-of-traces} \ C$

by (*metis (mono-tags, lifting) CollectI asm0(2) k-sem-def set-of-traces-def*)

qed

have $(\lambda i. (?l \ i, \text{snd} \ (?p \ i))) \in Q$

proof (*rule CHL-hyperpropE*)

show *CHL-hyperprop* $P \ Q \ (\text{range} \ ?p)$

proof (*rule CHL-hyperpropI*)

fix $l \ p$ **assume** *asm1*: $(\forall i. p \ i \in \text{range} \ (\lambda i. (\text{snd} \ (\text{states} \ i), \text{snd} \ (\text{states}' \ i))))$

$\wedge (\lambda i. (l \ i, \text{fst} \ (p \ i))) \in P$

then show $(\lambda i. (l \ i, \text{snd} \ (p \ i))) \in Q$

using *CHL-hyperprop-def*[*of P Q set-of-traces C*] $\langle \text{hypersat} \ C \ (CHL\text{-hyperprop} \ P \ Q) \rangle$

$\langle \text{range} \ (\lambda i. (\text{snd} \ (\text{states} \ i), \text{snd} \ (\text{states}' \ i))) \subseteq \text{set-of-traces} \ C \rangle \text{hypersat-def subset-iff}$

by *blast*

qed

show $(\lambda i. (\text{fst} \ (\text{states} \ i), \text{fst} \ (\text{snd} \ (\text{states} \ i), \text{snd} \ (\text{states}' \ i)))) \in P$

by (*simp add: asm0(1)*)

fix i **show** $(\text{snd} \ (\text{states} \ i), \text{snd} \ (\text{states}' \ i)) \in \text{range} \ (\lambda i. (\text{snd} \ (\text{states} \ i), \text{snd} \ (\text{states}' \ i)))$

by *blast*

```

qed
moreover have states' = ( $\lambda i. (?l\ i, snd\ (?p\ i))$ )
proof (rule ext)
  fix i show states' i = (fst (states i), snd (snd (states i), snd (states' i)))
  by (metis asm0(2) k-sem-def prod.exhaust-sel sndI)
qed
ultimately show states'  $\in Q$ 
  by auto
qed
next
assume asm0: CHL P C Q
have CHL-hyperprop P Q (set-of-traces C)
proof (rule CHL-hyperpropI)
  fix l p assume asm1: ( $\forall i. p\ i \in \text{set-of-traces } C$ )  $\wedge$  ( $\lambda i. (l\ i, fst\ (p\ i)) \in P$ )

  show ( $\lambda i. (l\ i, snd\ (p\ i)) \in Q$ )
  using asm0
proof (rule CHLE)
  show ( $\lambda i. (l\ i, fst\ (p\ i)) \in P$ )
  by (simp add: asm1)
  show k-sem C ( $\lambda i. (l\ i, fst\ (p\ i))$ ) ( $\lambda i. (l\ i, snd\ (p\ i))$ )
proof (rule k-semI)
  fix i show fst (l i, fst (p i)) = fst (l i, snd (p i))  $\wedge$   $\langle C, snd\ (l\ i, fst\ (p\ i)) \rangle$ 
 $\rightarrow$  snd (l i, snd (p i))
  using asm1 in-set-of-traces by fastforce
qed
qed
qed
then show hypersat C (CHL-hyperprop P Q)
  by (simp add: hypersat-def)
qed

```

```

theorem k-hypersafety-HL-hyperprop:
  fixes P :: ('i  $\Rightarrow$  ('lvar, 'lval, 'pvar, 'pval) state) set
  assumes finite (UNIV :: 'i set)
  and card (UNIV :: 'i set) = k
  shows k-hypersafety k (CHL-hyperprop P Q)
proof (rule k-hypersafetyI)
  fix S
  assume  $\neg$  CHL-hyperprop P Q S
  then obtain l p where asm0:  $\forall i. p\ i \in S$  ( $\lambda i. (l\ i, fst\ (p\ i)) \in P$ )
    ( $\lambda i. (l\ i, snd\ (p\ i)) \notin Q$ )
  using CHL-hyperprop-def by blast
  let ?S = range p
  have max-k k ?S
  by (metis assms(1) assms(2) card-image-le finite-imageI max-k-def)
  moreover have  $\bigwedge S''. ?S \subseteq S'' \implies \neg$  CHL-hyperprop P Q S''

```

by (*meson asm0(2) asm0(3) CHL-hyperprop-def range-subsetD*)
ultimately show $\exists S' \subseteq S. \max-k k S' \wedge (\forall S''. S' \subseteq S'' \longrightarrow \neg \text{CHL-hyperprop } P Q S'')$
 by (*meson asm0(1) image-subsetI*)
qed

3.4 Incorrectness Logic [9] or Reverse Hoare Logic [4] (IL)

Definition 18 *definition IL where*

$IL P C Q \longleftrightarrow Q \subseteq \text{sem } C P$

lemma *equiv-def-incorrectness:*

$IL P C Q \longleftrightarrow (\forall l \sigma'. (l, \sigma') \in Q \longrightarrow (\exists \sigma. (l, \sigma) \in P \wedge \langle C, \sigma \rangle \rightarrow \sigma'))$

by (*simp add: in-sem IL-def subset-iff*)

definition *IL-hyperprop where*

$IL\text{-hyperprop } P Q S \longleftrightarrow (\forall l \sigma'. (l, \sigma') \in Q \longrightarrow (\exists \sigma. (l, \sigma) \in P \wedge (\sigma, \sigma') \in S))$

lemma *IL-hyperpropI:*

assumes $\bigwedge l \sigma'. (l, \sigma') \in Q \implies (\exists \sigma. (l, \sigma) \in P \wedge (\sigma, \sigma') \in S)$

shows $IL\text{-hyperprop } P Q S$

by (*simp add: assms IL-hyperprop-def*)

Proposition 5 *lemma IL-expresses-hyperproperties:*

$IL P C Q \longleftrightarrow IL\text{-hyperprop } P Q (\text{set-of-traces } C)$ (**is** $?A \longleftrightarrow ?B$)

proof

assume $?A$

show $?B$

proof (*rule IL-hyperpropI*)

fix $l \sigma'$ **assume** $asm0: (l, \sigma') \in Q$

then obtain σ **where** $(l, \sigma) \in P$ *single-sem C $\sigma \sigma'$*

using $\langle IL P C Q \rangle$ *equiv-def-incorrectness* **by** *blast*

then show $\exists \sigma. (l, \sigma) \in P \wedge (\sigma, \sigma') \in \text{set-of-traces } C$

using *set-of-traces-def* **by** *auto*

qed

next

assume $?B$

have $Q \subseteq \text{sem } C P$

proof (*rule subsetPairI*)

fix $l \sigma'$ **assume** $(l, \sigma') \in Q$

then obtain σ **where** $(\sigma, \sigma') \in \text{set-of-traces } C$ $(l, \sigma) \in P$

by (*meson $\langle IL\text{-hyperprop } P Q (\text{set-of-traces } C) \rangle$ IL-hyperprop-def*)

then show $(l, \sigma') \in \text{sem } C P$

using *in-set-of-traces-then-in-sem* **by** *blast*

qed

then show $?A$

by (*simp add: IL-def*)

qed

lemma *IL-consequence*:

assumes *IL P C Q*
and $(l, \sigma') \in Q$
shows $\exists \sigma. (l, \sigma) \in P \wedge \text{single-sem } C \sigma \sigma'$
using *assms(1) assms(2) equiv-def-incorrectness* **by** *blast*

Proposition 6 **theorem** *encoding-IL*:

$IL P C Q \longleftrightarrow (\text{hyper-hoare-triple } (\text{under-approx } P) C (\text{under-approx } Q)) \text{ (is } ?A \longleftrightarrow ?B)$

proof (*rule iffI*)

show $?B \implies ?A$

proof –

assume $?B$

then have *under-approx Q (sem C P)*

by (*simp add: hyper-hoare-triple-def under-approx-def*)

then show $?A$

by (*simp add: IL-def under-approx-def*)

qed

assume $?A$

then show $?B$

by (*simp add: hyper-hoare-triple-def sem-monotonic IL-def under-approx-def subset-trans*)

qed

lemma *entailment-order-reverse-hoare*:

assumes $P \subseteq P'$

shows *entails (under-approx P') (under-approx P)*

by (*simp add: assms dual-order.trans entails-def under-approx-def*)

definition *incorrectify* **where**

incorrectify p = under-approx { $\sigma \mid \sigma. p \sigma$ }

lemma *incorrectifyI*:

assumes $\bigwedge \sigma. p \sigma \implies \sigma \in S$

shows *incorrectify p S*

by (*metis assms incorrectify-def mem-Collect-eq subsetI under-approx-def*)

lemma *incorrectifyE*:

assumes *incorrectify p S*

and $p \sigma$

shows $\sigma \in S$

by (*metis CollectI assms(1) assms(2) in-mono incorrectify-def under-approx-def*)

lemma *simple-while-incorrectness*:

assumes $\bigwedge n. \text{hyper-hoare-triple } (\text{incorrectify } (p \ n)) C (\text{incorrectify } (p \ (\text{Suc } n)))$

shows *hyper-hoare-triple (incorrectify (p 0)) (While C) (incorrectify ($\lambda \sigma. \exists n. p \ n \ \sigma$))*

proof (*rule consequence-rule*)
show *entails* (*incorrectify* ($p \ 0$)) (*incorrectify* ($p \ 0$))
by (*simp add: entailsI*)
show *hyper-hoare-triple* (*incorrectify* ($p \ 0$)) (*While C*) (*natural-partition* ($\lambda n.$
incorrectify ($p \ n$)))
by (*meson assms while-rule*)

have *entails* (*incorrectify* ($\lambda\sigma. \exists n. p \ n \ \sigma$)) (*natural-partition* ($\lambda n.$ *incorrectify* ($p \ n$)))

proof (*rule entailsI*)
fix S **assume** *asm0*: *incorrectify* ($\lambda\sigma. \exists n. p \ n \ \sigma$) S
then have *under-approx* $\{ \sigma \mid \sigma \ n. p \ n \ \sigma \}$ S
by (*metis incorrectify-def*)
let $?F = \lambda n. S$
show *natural-partition* ($\lambda n.$ *incorrectify* ($p \ n$)) S
proof (*rule natural-partitionI*)
show $\bigwedge n. \text{incorrectify } (p \ n) \ (?F \ n)$
by (*metis asm0 incorrectifyE incorrectifyI*)
show $S = \bigcup (\text{range } ?F)$
by *simp*
qed
qed

show *entails* (*natural-partition* ($\lambda n.$ *incorrectify* ($p \ n$))) (*incorrectify* ($\lambda\sigma. \exists n. p \ n \ \sigma$))

proof (*rule entailsI*)
fix S **assume** *asm0*: *natural-partition* ($\lambda n.$ *incorrectify* ($p \ n$)) S
then obtain F **where** $S = (\bigcup n. F \ n) \bigwedge n. \text{incorrectify } (p \ n) \ (F \ n)$
using *natural-partitionE* **by** *blast*
show *incorrectify* ($\lambda\sigma. \exists n. p \ n \ \sigma$) S
proof (*rule incorrectifyI*)
fix σ **assume** $\exists n. p \ n \ \sigma$
then obtain n **where** $p \ n \ \sigma$
by *blast*
then have $\sigma \in F \ n$
by (*meson* $\langle \bigwedge n. \text{incorrectify } (p \ n) \ (F \ n) \rangle$ *incorrectifyE*)
then show $\sigma \in S$
using $\langle S = \bigcup (\text{range } F) \rangle$ **by** *blast*
qed
qed
qed

definition *sat-for-l* **where**

sat-for-l $l \ P \longleftrightarrow (\exists \sigma. (l, \sigma) \in P)$

theorem *incorrectness-hyperliveness*:

assumes $\bigwedge l. \text{sat-for-l } l \ Q \implies \text{sat-for-l } l \ P$

shows *hyperliveness* (*IL-hyperprop* $P \ Q$)

proof (*rule hyperlivenessI*)
fix S
let $?S = S \cup \{(\sigma, \sigma') \mid \sigma \sigma' l. (l, \sigma') \in Q \wedge (l, \sigma) \in P\}$
have *IL-hyperprop* $P Q ?S$
proof (*rule IL-hyperpropI*)
fix $l \sigma'$
assume $asm0: (l, \sigma') \in Q$
then obtain σ **where** $(l, \sigma) \in P$
by (*meson assms sat-for-l-def*)
then show $\exists \sigma. (l, \sigma) \in P \wedge (\sigma, \sigma') \in ?S$
using $asm0$ **by** *auto*
qed
then show $\exists S'. S \subseteq S' \wedge \text{IL-hyperprop } P Q S'$
by *auto*
qed

3.5 k-Incorrectness Logic [8] (k-IL)

RIL is the old name of k-IL.

Definition 19 *definition RIL where*

$$RIL P C Q \longleftrightarrow (\forall \text{states}' \in Q. \exists \text{states} \in P. k\text{-sem } C \text{ states states}')$$

lemma *RILI:*

assumes $\bigwedge \text{states}'. \text{states}' \in Q \implies (\exists \text{states} \in P. k\text{-sem } C \text{ states states}')$
shows *RIL* $P C Q$
by (*simp add: assms RIL-def*)

lemma *RILE:*

assumes *RIL* $P C Q$
and $\text{states}' \in Q$
shows $\exists \text{states} \in P. k\text{-sem } C \text{ states states}'$
using *assms(1) assms(2) RIL-def* **by** *blast*

definition *RIL-hyperprop where*

$$\begin{aligned}
 RIL\text{-hyperprop } P Q S &\longleftrightarrow (\forall l \text{states}'. (\lambda i. (l i, \text{states}' i)) \in Q \\
 &\longrightarrow (\exists \text{states}. (\lambda i. (l i, \text{states } i)) \in P \wedge (\forall i. (\text{states } i, \text{states}' i) \in S)))
 \end{aligned}$$

lemma *RIL-hyperpropI:*

assumes $\bigwedge l \text{states}'. (\lambda i. (l i, \text{states}' i)) \in Q \implies (\exists \text{states}. (\lambda i. (l i, \text{states } i)) \in P \wedge (\forall i. (\text{states } i, \text{states}' i) \in S))$
shows *RIL-hyperprop* $P Q S$
by (*simp add: assms RIL-hyperprop-def*)

lemma *RIL-hyperpropE:*

assumes *RIL-hyperprop* $P Q S$
and $(\lambda i. (l i, \text{states}' i)) \in Q$

shows $\exists \text{states}. (\lambda i. (l\ i, \text{states}\ i)) \in P \wedge (\forall i. (\text{states}\ i, \text{states}'\ i) \in S)$
using *assms(1) assms(2) RIL-hyperprop-def* **by** *blast*

lemma *useful*:

states' = $(\lambda i. ((fst \circ \text{states}')\ i, (snd \circ \text{states}')\ i))$

proof (*rule ext*)

fix *i* **show** *states'* *i* = $((fst \circ \text{states}')\ i, (snd \circ \text{states}')\ i)$

by *auto*

qed

Proposition 17 **theorem** *RIL-expresses-hyperproperties*:

hypersat C (RIL-hyperprop P Q) \longleftrightarrow RIL P C Q (is ?A \longleftrightarrow ?B)

proof

assume *?A*

show *?B*

proof (*rule RILI*)

fix *states'* **assume** *asm0*: *states'* \in *Q*

then obtain *states* **where** *asm0*: $(\lambda i. ((fst \circ \text{states}')\ i, \text{states}\ i)) \in P \wedge (\forall i. (\text{states}\ i, (snd \circ \text{states}')\ i) \in \text{set-of-traces}\ C)$

using *RIL-hyperpropE[of P Q set-of-traces C fst \circ states' snd \circ states']* *?A*

using *hypersat-def* **by** *auto*

moreover have *k-sem C* $(\lambda i. ((fst \circ \text{states}')\ i, \text{states}\ i))\ \text{states}'$

proof (*rule k-semI*)

fix *i*

have $\langle C, \text{snd}\ ((fst \circ \text{states}')\ i, \text{states}\ i) \rangle \rightarrow \text{snd}\ (\text{states}'\ i)$

using *calculation set-of-traces-def* **by** *auto*

then show $\text{fst}\ ((fst \circ \text{states}')\ i, \text{states}\ i) = \text{fst}\ (\text{states}'\ i) \wedge \langle C, \text{snd}\ ((fst \circ \text{states}')\ i, \text{states}\ i) \rangle \rightarrow \text{snd}\ (\text{states}'\ i)$

by *simp*

qed

ultimately show $\exists \text{states} \in P. \text{k-sem}\ C\ \text{states}\ \text{states}'$

by *fast*

qed

next

assume *?B*

have *RIL-hyperprop P Q (set-of-traces C)*

proof (*rule RIL-hyperpropI*)

fix *l states'*

assume *asm0*: $(\lambda i. (l\ i, \text{states}'\ i)) \in Q$

then obtain *states* **where** *states* \in *P* *k-sem C states* $(\lambda i. (l\ i, \text{states}'\ i))$

using $\langle \text{RIL}\ P\ C\ Q \rangle$ *RILE* **by** *blast*

moreover have $(\lambda i. (l\ i, (snd \circ \text{states})\ i)) = \text{states}$

proof (*rule ext*)

fix *i* **show** $(l\ i, (snd \circ \text{states})\ i) = \text{states}\ i$

by (*metis calculation(2) comp-apply fst-conv k-sem-def surjective-pairing*)

qed

moreover have $\bigwedge i. ((snd \circ \text{states})\ i, \text{states}'\ i) \in \text{set-of-traces}\ C$

by (*metis (mono-tags, lifting) calculation(2) comp-apply in-set-of-traces*)

k-sem-def snd-conv)
ultimately show $\exists \text{ states. } (\lambda i. (l\ i, \text{ states } i)) \in P \wedge (\forall i. (\text{ states } i, \text{ states}'\ i) \in \text{ set-of-traces } C)$
by force
qed
then show $?A$
using *hypersat-def* **by** *blast*
qed

definition *k-sat-for-l* **where**
k-sat-for-l $l\ P \longleftrightarrow (\exists \sigma. (\lambda i. (l\ i, \sigma\ i)) \in P)$

theorem *RIL-hyperprop-hyperlive*:
assumes $\bigwedge l. k\text{-sat-for-l } l\ Q \implies k\text{-sat-for-l } l\ P$
shows *hyperliveness* (*RIL-hyperprop* $P\ Q$)
proof (*rule hyperlivenessI*)
fix S
have *RIL-hyperprop* $P\ Q\ UNIV$
by (*meson* *assms RIL-hyperpropI iso-tuple-UNIV-I k-sat-for-l-def*)
then show $\exists S'. S \subseteq S' \wedge \text{RIL-hyperprop } P\ Q\ S'$
by *blast*
qed

definition *strong-pre-insec* **where**
strong-pre-insec $\text{from-nat } x\ c\ P\ S \longleftrightarrow (\forall \text{ states } \in P. (\forall i. \text{fst } (\text{ states } i)\ x = \text{from-nat } i) \longrightarrow (\exists r. \forall i. ((\text{fst } (\text{ states } i))(c := r), \text{snd } (\text{ states } i)) \in S)) \wedge (\forall \text{ states. } (\forall i. \text{ states } i \in S) \wedge (\forall i. \text{fst } (\text{ states } i)\ x = \text{from-nat } i) \wedge (\forall i\ j. \text{fst } (\text{ states } i)\ c = \text{fst } (\text{ states } j)\ c) \longrightarrow \text{ states } \in P)$

lemma *strong-pre-insecI*:
assumes $\bigwedge \text{ states. } \text{ states } \in P \implies (\forall i. \text{fst } (\text{ states } i)\ x = \text{from-nat } i) \implies (\exists r. \forall i. ((\text{fst } (\text{ states } i))(c := r), \text{snd } (\text{ states } i)) \in S)$
and $\bigwedge \text{ states. } (\forall i. \text{ states } i \in S) \implies (\forall i. \text{fst } (\text{ states } i)\ x = \text{from-nat } i) \implies (\forall i\ j. \text{fst } (\text{ states } i)\ c = \text{fst } (\text{ states } j)\ c) \implies \text{ states } \in P$
shows *strong-pre-insec* $\text{from-nat } x\ c\ P\ S$
by (*simp* *add: assms(1) assms(2) strong-pre-insec-def*)

lemma *strong-pre-insecE*:
assumes *strong-pre-insec* $\text{from-nat } x\ c\ P\ S$
and $\bigwedge i. \text{ states } i \in S$
and $\bigwedge i. \text{fst } (\text{ states } i)\ x = \text{from-nat } i$
and $\bigwedge i\ j. \text{fst } (\text{ states } i)\ c = \text{fst } (\text{ states } j)\ c$
shows $\text{ states } \in P$
by (*meson* *assms(1) assms(2) assms(3) assms(4) strong-pre-insec-def*)

definition *pre-insec* **where**

pre-insec from-nat x c P S \longleftrightarrow (\forall *states* \in *P*.
 $(\forall i. \text{fst } (\text{states } i) \ x = \text{from-nat } i)$
 $\longrightarrow (\exists r. \forall i. ((\text{fst } (\text{states } i))(c := r), \text{snd } (\text{states } i)) \in S)$)

lemma *pre-insecI*:

assumes \bigwedge *states*. *states* \in *P* \implies ($\forall i. \text{fst } (\text{states } i) \ x = \text{from-nat } i$)
 $\implies (\exists r. \forall i. ((\text{fst } (\text{states } i))(c := r), \text{snd } (\text{states } i)) \in S)$
shows *pre-insec from-nat x c P S*
by (*simp add: assms(1) pre-insec-def*)

lemma *strong-pre-implies-pre*:

assumes *strong-pre-insec from-nat x c P S*
shows *pre-insec from-nat x c P S*
by (*meson assms pre-insecI strong-pre-insec-def*)

lemma *pre-insecE*:

assumes *pre-insec from-nat x c P S*
and *states* \in *P*
and $\bigwedge i. \text{fst } (\text{states } i) \ x = \text{from-nat } i$
shows $\exists r. \forall i. ((\text{fst } (\text{states } i))(c := r), \text{snd } (\text{states } i)) \in S$
by (*meson assms(1) assms(2) assms(3) pre-insec-def*)

definition *post-insec where*

post-insec from-nat x c Q S \longleftrightarrow (\forall *states* \in *Q*. ($\forall i. \text{fst } (\text{states } i) \ x = \text{from-nat } i$)
 $\longrightarrow (\exists r. (\forall i. ((\text{fst } (\text{states } i))(c := r), \text{snd } (\text{states } i)) \in S))$)

lemma *post-insecE*:

assumes *post-insec from-nat x c Q S*
and *states* \in *Q*
and $\bigwedge i. \text{fst } (\text{states } i) \ x = \text{from-nat } i$
shows $\exists r. (\forall i. ((\text{fst } (\text{states } i))(c := r), \text{snd } (\text{states } i)) \in S)$
by (*meson assms(1) assms(2) assms(3) post-insec-def*)

lemma *post-insecI*:

assumes \bigwedge *states*. *states* \in *Q* \implies ($\forall i. \text{fst } (\text{states } i) \ x = \text{from-nat } i$)
 $\implies (\exists r. (\forall i. ((\text{fst } (\text{states } i))(c := r), \text{snd } (\text{states } i)) \in S))$
shows *post-insec from-nat x c Q S*
by (*simp add: assms post-insec-def*)

lemma *same-pre-post*:

pre-insec from-nat x c Q S \longleftrightarrow *post-insec from-nat x c Q S*
by (*simp add: post-insec-def pre-insec-def*)

theorem *can-be-sat*:

fixes *x* :: 'lvar

assumes $\bigwedge l l' \sigma. (\lambda i. (l i, \sigma i)) \in P \longleftrightarrow (\lambda i. (l' i, \sigma i)) \in P$
and *injective* (*indexify* :: $((a \Rightarrow (pvar \Rightarrow 'pval)) \Rightarrow 'lval)$)
and $x \neq c$
and *injective from-nat*
shows *sat* (*strong-pre-insec from-nat* $x c (P :: (a \Rightarrow (lvar \Rightarrow 'lval)) \times (pvar \Rightarrow 'pval))$ *set*)
proof –

let $?S = \bigcup states \in P. \{ ((fst (states i))(x := from-nat i))(c := indexify (\lambda i. snd (states i))), snd (states i)) \mid i. True \}$

have *strong-pre-insec from-nat* $x c P ?S$
proof (*rule strong-pre-insecI*)
fix *states*
assume *asm0*: $states \in P \forall i. fst (states i) x = from-nat i$
define *r* **where** $r = indexify (\lambda i. snd (states i))$
have $\bigwedge i. ((fst (states i))(c := r), snd (states i)) \in \{ ((fst (states i))(x := from-nat i))(c := indexify (\lambda i. snd (states i))), snd (states i)) \mid i. True \}$
proof –
fix *i*
show $((fst (states i))(c := r), snd (states i)) \in \{ ((fst (states i))(x := from-nat i))(c := indexify (\lambda i. snd (states i))), snd (states i)) \mid i. True \}$
using *asm0(2)* *r-def* **by** *fastforce*
qed
then show $\exists r. \forall i. ((fst (states i))(c := r), snd (states i)) \in ?S$
by (*meson UN-I asm0(1)*)
next
fix *states*

assume *asm0*: $\forall i. states i \in ?S \forall i. fst (states i) x = from-nat i \forall i j. fst (states i) c = fst (states j) c$

let $?P = \lambda i states'. states' \in P \wedge states i \in \{ ((fst (states' i))(x := from-nat i))(c := indexify (\lambda i. snd (states' i))), snd (states' i)) \mid i. True \}$

let $?states = \lambda i. SOME states'. ?P i states'$
have $r: \bigwedge i. ?P i (?states i)$
proof –
fix *i*
show $?P i (?states i)$
proof (*rule someI-ex[of ?P i]*)
show $\exists states'. states' \in P \wedge states i \in \{ ((fst (states' i))(x := from-nat i))(c := indexify (\lambda i. snd (states' i))), snd (states' i)) \mid i. True \}$
using *asm0(1)* **by** *fastforce*
qed
qed
moreover have $rr: \bigwedge i. fst (states i) c = indexify (\lambda j. snd (?states i j)) \wedge snd (?states i i) = snd (states i)$
proof –

```

fix  $i$ 
obtain  $j$  where  $j\text{-def}: \text{states } i = (((\text{fst } (?states\ i\ j))(x := \text{from-nat } j))(c :=$ 
 $\text{indexify } (\lambda k. \text{snd } (?states\ i\ k))), \text{snd } (?states\ i\ j))$ 
  using  $r[\text{of } i]$  by  $\text{blast}$ 
  then have  $r1: \text{snd } (?states\ i\ j) = \text{snd } (\text{states } i)$ 
  by  $(\text{metis } (\text{no-types}, \text{lifting}) \text{snd-conv})$ 
  then have  $\text{from-nat } i = \text{from-nat } j$ 
  by  $(\text{metis } (\text{no-types}, \text{lifting}) j\text{-def } \text{asm0}(2) \text{ assms}(3) \text{fst-conv } \text{fun-upd-same}$ 
 $\text{fun-upd-twist})$ 
  then have  $i = j$ 
  by  $(\text{meson } \text{assms}(4) \text{injective-def})$ 
  show  $\text{fst } (\text{states } i) c = \text{indexify } (\lambda j. \text{snd } (?states\ i\ j)) \wedge \text{snd } (?states\ i\ i) =$ 
 $\text{snd } (\text{states } i)$ 
  proof
    show  $\text{fst } (\text{states } i) c = \text{indexify } (\lambda j. \text{snd } (?states\ i\ j))$ 
    by  $(\text{metis } (\text{no-types}, \text{lifting}) j\text{-def } \text{fst-conv } \text{fun-upd-same})$ 
    show  $\text{snd } (?states\ i\ i) = \text{snd } (\text{states } i)$ 
    using  $\langle i = j \rangle r1$  by  $\text{blast}$ 
  qed
qed
moreover have  $r0: \bigwedge i j. (\lambda n. \text{snd } (?states\ i\ n)) = (\lambda n. \text{snd } (?states\ j\ n))$ 
proof  $-$ 
  fix  $i j$ 
  have  $\text{indexify } (\lambda n. \text{snd } (?states\ i\ n)) = \text{indexify } (\lambda n. \text{snd } (?states\ j\ n))$ 
  using  $\text{asm0}(3) rr$  by  $\text{fastforce}$ 
  then show  $(\lambda n. \text{snd } (?states\ i\ n)) = (\lambda n. \text{snd } (?states\ j\ n))$ 
  by  $(\text{meson } \text{assms}(2) \text{injective-def})$ 
qed
obtain  $k :: 'a$  where  $\text{True}$  by  $\text{blast}$ 
then have  $?states\ k \in P$ 
  using  $UN\text{-iff}[\text{of } - \lambda \text{states}. \{((\text{fst } (\text{states } i))(x := \text{from-nat } i, c := \text{indexify } (\lambda i.$ 
 $\text{snd } (\text{states } i))), \text{snd } (\text{states } i)) \mid i. \text{True}\} P]$ 
   $\text{asm0}(1) \text{someI-ex}[\text{of } \lambda y. y \in P \wedge \text{states } k \in \{((\text{fst } (y\ i))(x := \text{from-nat } i,$ 
 $c := \text{indexify } (\lambda i. \text{snd } (y\ i))), \text{snd } (y\ i)) \mid i. \text{True}\}]$ 
  by  $\text{fast}$ 
moreover have  $\bigwedge i. \text{snd } (?states\ k\ i) = \text{snd } (\text{states } i)$ 
proof  $-$ 
  fix  $i$ 
  have  $\text{snd } (?states\ i\ i) = \text{snd } (\text{states } i)$ 
  using  $rr$  by  $\text{blast}$ 
moreover have  $(\lambda n. \text{snd } (?states\ i\ n)) i = (\lambda n. \text{snd } (?states\ k\ n)) i$ 
  by  $(\text{metis } r0)$ 
ultimately show  $\text{snd } (?states\ k\ i) = \text{snd } (\text{states } i)$ 
  by  $\text{auto}$ 
qed
moreover have  $(\lambda i. ((\lambda i. \text{fst } (?states\ k\ i)) i, (\lambda i. \text{snd } (\text{states } i)) i)) \in P \longleftrightarrow$ 
 $(\lambda i. ((\lambda i. \text{fst } (\text{states } i)) i, (\lambda i. \text{snd } (\text{states } i)) i)) \in P$ 
  using  $\text{assms}(1)$  by  $\text{fast}$ 
moreover have  $(\lambda i. ((\lambda i. \text{fst } (\text{states } i)) i, (\lambda i. \text{snd } (\text{states } i)) i)) = \text{states}$ 

```

```

proof (rule ext)
  fix i show (fst (states i), snd (states i)) = states i
    by simp
qed
moreover have (λi. ((λi. fst (?states k i)) i, (λi. snd (states i)) i)) = ?states
k
proof (rule ext)
  fix i show (λi. ((λi. fst (?states k i)) i, (λi. snd (states i)) i)) i = ?states k i
    by (metis (no-types, lifting) calculation(4) prod.exhaust-sel)
qed
ultimately show states ∈ P by argo
qed
then show sat (strong-pre-insec from-nat x c P)
  by (meson sat-def)
qed

theorem encode-insec:
  assumes injective from-nat
    and sat (strong-pre-insec from-nat x c (P :: ('a ⇒ ('lvar ⇒ 'lval) × ('pvar
⇒ 'pval)) set))
    and not-free-var-of P x ∧ not-free-var-of P c
    and not-free-var-of Q x ∧ not-free-var-of Q c
    and c ≠ x

  shows RIL P C Q ⟷ ⊨ {pre-insec from-nat x c P} C {post-insec from-nat
x c Q} (is ?A ⟷ ?B)
proof
  assume ?A
  show ?B
  proof (rule hyper-hoare-tripleI)
    fix S assume asm0: pre-insec from-nat x c P S

    show post-insec from-nat x c Q (sem C S)
    proof (rule post-insecI)
      fix states' assume asm1: states' ∈ Q ∀i. fst (states' i) x = from-nat i
      then obtain states where states ∈ P k-sem C states states'
        using ⟨RIL P C Q⟩ RILE by blast
      then obtain r where asm2: ∧i. ((fst (states i))(c := r), snd (states i)) ∈ S
        using pre-insecE[of from-nat x c P S states]
        by (metis asm0 asm1(2) k-sem-def)
      then show ∃r. ∀i. ((fst (states' i))(c := r), snd (states' i)) ∈ sem C S
        by (metis (mono-tags, opaque-lifting) ⟨k-sem C states states'⟩ k-sem-def
single-step-then-in-sem)
    qed
  qed
next
  assume asm0: ?B
  show ?A
  proof (rule RILI)

```

```

fix states' assume asm1: states' ∈ Q
obtain S where asm2: strong-pre-insec from-nat x c P S
  by (meson assms(2) sat-def)
then have asm3: post-insec from-nat x c Q (sem C S)
  by (meson asm0 hyper-hoare-tripleE strong-pre-implies-pre)

let ?states' = λi. ((fst (states' i))(x := from-nat i), snd (states' i))
have ?states' ∈ Q
by (metis (no-types, lifting) asm1 assms(4) diff-by-update fstI not-free-var-of-def
snd-conv)
then obtain r where r-def:  $\bigwedge i. ((fst (?states' i))(c := r), snd (?states' i)) \in sem\ C\ S$ 
  using asm3 post-insecE[of from-nat x c Q] by fastforce

let ?states = λi. SOME σ. ((fst (?states' i))(c := r), σ) ∈ S ∧ single-sem C σ
(snd (?states' i))

  have asm4:  $\bigwedge i. ((fst (?states' i))(c := r), (?states\ i)) \in S \wedge single-sem\ C$ 
(?states\ i) (snd (?states' i))
  proof –
    fix i
    have  $\exists \sigma. ((fst (?states' i))(c := r), \sigma) \in S \wedge single-sem\ C\ \sigma$  (snd (?states'
i))
    by (metis r-def fst-conv in-sem snd-conv)
    then show  $((fst (?states' i))(c := r), (?states\ i)) \in S \wedge single-sem\ C$  (?states
i) (snd (?states' i))
    using someI-ex[of λσ. ((fst (?states' i))(c := r), σ) ∈ S ∧ single-sem C σ
(snd (?states' i))]
    by blast
  qed
moreover have r0:  $(\lambda i. ((fst (?states' i))(c := r), (?states\ i))) \in P$ 
  using asm2
proof (rule strong-pre-insecE)
  fix i
  show  $(\lambda i. ((fst (?states' i))(c := r), (?states\ i))) i \in S$ 
  using calculation by blast
  show fst  $((\lambda i. ((fst (?states' i))(c := r), (?states\ i))) i)$  x = from-nat i
  using assms(5) by auto
  fix j
  show fst  $((\lambda i. ((fst (?states' i))(c := r), (?states\ i))) i)$  c = fst  $((\lambda i. ((fst$ 
(?states' i))(c := r), (?states\ i))) j) c
  by fastforce
  qed
have r1:  $(\lambda i. (((fst (?states' i))(c := r))(x := fst (states' i) x), (?states\ i))) \in$ 
P
proof (rule not-free-var-ofE[of P x])
  show  $(\lambda i. ((fst (?states' i))(c := r), (?states\ i))) \in P$ 
  using r0 by fastforce
  show not-free-var-of P x

```

```

    by (simp add: assms(3))
  fix i
  show differ-only-by
    (fst ((fst ((fst (states' i))(x := from-nat i), snd (states' i)))(c := r), ?states
i))
    (fst ((fst ((fst (states' i))(x := from-nat i), snd (states' i)))(c := r, x :=
fst (states' i) x), ?states i) x
    by (metis (mono-tags, lifting) diff-by-comm diff-by-update fst-conv)
  qed (auto)
  have (λi. (((fst (?states' i))(c := r))(x := fst (states' i) x))(c := fst (states'
i) c), (?states i))) ∈ P
  proof (rule not-free-var-ofE)
    show (λi. (((fst (?states' i))(c := r))(x := fst (states' i) x), (?states i))) ∈ P
      using r1 by fastforce
    show not-free-var-of P c
      by (simp add: assms(3))
    fix i show differ-only-by
      (fst ((fst ((fst (states' i))(x := from-nat i), snd (states' i)))(c := r, x :=
fst (states' i) x), ?states i))
      (fst ((fst ((fst (states' i))(x := from-nat i), snd (states' i)))(c := r, x :=
fst (states' i) x, c := fst (states' i) c), ?states i))
      c
      by (metis (mono-tags, lifting) diff-by-comm diff-by-update fst-conv)
    qed (auto)
    moreover have (λi. (((fst (?states' i))(c := r))(x := fst (states' i) x))(c :=
fst (states' i) c), (?states i))
      = (λi. (fst (states' i), (?states i)))
    proof (rule ext)
      fix i show (λi. (((fst (?states' i))(c := r))(x := fst (states' i) x))(c := fst
(states' i) c), (?states i)) i
        = (λi. (fst (states' i), (?states i))) i
          by force
    qed
    moreover have k-sem C (λi. (fst (states' i), (?states i))) states'
  proof (rule k-semI)
    fix i
    show (fst ((λi. (fst (states' i), (?states i))) i) = fst (states' i) ∧
single-sem C (snd ((λi. (fst (states' i), (?states i))) i)) (snd (states' i)))
      using asm4 by auto
    qed
  ultimately show ∃ states∈P. k-sem C states states'
    by auto
  qed
qed

```

Proposition 8 *theorem encoding-RIL:*

```

fixes x :: 'lvar
assumes ∧l l' σ. (λi. (l i, σ i)) ∈ P ⟷ (λi. (l' i, σ i)) ∈ P
and injective (indexify :: ('a ⇒ ('pvar ⇒ 'pval)) ⇒ 'lval))

```

and $c \neq x$
and *injective from-nat*
and *not-free-var-of* ($P :: ('a \Rightarrow ('lvar \Rightarrow 'lval) \times ('pvar \Rightarrow 'pval)) \text{ set}) x \wedge$
not-free-var-of $P c$
and *not-free-var-of* $Q x \wedge$ *not-free-var-of* $Q c$
shows $RIL P C Q \longleftrightarrow \models \{pre\text{-insec from-nat } x c P\} C \{post\text{-insec from-nat } x c Q\}$ (**is** $?A \longleftrightarrow ?B$)
proof (*rule encode-insec*)
show *sat* (*strong-pre-insec from-nat* $x c (P :: ('a \Rightarrow ('lvar \Rightarrow 'lval) \times ('pvar \Rightarrow 'pval)) \text{ set})$)
proof (*rule can-be-sat*)
show *injective* (*indexify* $:: (('a \Rightarrow ('pvar \Rightarrow 'pval)) \Rightarrow 'lval)$)
by (*simp add: assms(2)*)
show $x \neq c$
using *assms(3)* **by** *auto*
qed (*auto simp add: assms*)
qed (*auto simp add: assms*)

3.6 Forward Underapproximation (FU)

As employed by Outcome Logic [11]

Definition 20 *definition FU where*

$FU P C Q \longleftrightarrow (\forall l. \forall \sigma. (l, \sigma) \in P \longrightarrow (\exists \sigma'. \text{single-sem } C \sigma \sigma' \wedge (l, \sigma') \in Q))$

lemma *FUI:*

assumes $\bigwedge \sigma l. (l, \sigma) \in P \implies (\exists \sigma'. \text{single-sem } C \sigma \sigma' \wedge (l, \sigma') \in Q)$

shows $FU P C Q$

by (*simp add: assms FU-def*)

definition *encode-FU where*

$encode\text{-FU } P S \longleftrightarrow P \cap S \neq \{\}$

Proposition 9 *theorem encoding-FU:*

$FU P C Q \longleftrightarrow \models \{encode\text{-FU } P\} C \{encode\text{-FU } Q\}$ (**is** $?A \longleftrightarrow ?B$)

proof

show $?B \implies ?A$

proof –

assume $?B$

show $?A$

proof (*rule FUI*)

fix σl

assume *asm:* $(l, \sigma) \in P$

then have $encode\text{-FU } P \{(l, \sigma)\}$

by (*simp add: encode-FU-def*)

then have $Q \cap \text{sem } C \{(l, \sigma)\} \neq \{\}$

using $\langle \models \{encode\text{-FU } P\} C \{encode\text{-FU } Q\} \rangle$ *hyper-hoare-tripleE encode-FU-def* **by** *blast*

```

    then obtain  $\varphi'$  where  $\varphi' \in Q$   $\varphi' \in \text{sem } C \{(l, \sigma)\}$ 
      by blast
    then show  $\exists \sigma'. \text{single-sem } C \sigma \sigma' \wedge (l, \sigma') \in Q$ 
      by (metis fst-conv in-sem prod.collapse singletonD snd-conv)
  qed
qed
assume ?A
show ?B
proof (rule hyper-hoare-tripleI)
  fix S assume encode-FU P S
  then obtain l  $\sigma$  where  $(l, \sigma) \in P \cap S$ 
    by (metis Expressivity.encode-FU-def ex-in-conv surj-pair)
  then obtain  $\sigma'$  where  $\text{single-sem } C \sigma \sigma' \wedge (l, \sigma') \in Q$ 
    by (meson IntD1 ‹FU P C Q› FU-def)
  then show encode-FU Q (sem C S)
    using Expressivity.encode-FU-def ‹(l,  $\sigma$ )  $\in P \cap S$ › sem-def by fastforce
  qed
qed

definition hyperprop-FU where
  hyperprop-FU P Q S  $\longleftrightarrow (\forall l \sigma. (l, \sigma) \in P \longrightarrow (\exists \sigma'. (l, \sigma') \in Q \wedge (\sigma, \sigma') \in S))$ 

lemma hyperprop-FUI:
  assumes  $\bigwedge l \sigma. (l, \sigma) \in P \implies (\exists \sigma'. (l, \sigma') \in Q \wedge (\sigma, \sigma') \in S)$ 
  shows hyperprop-FU P Q S
  by (simp add: hyperprop-FU-def assms)

lemma hyperprop-FUE:
  assumes hyperprop-FU P Q S
    and  $(l, \sigma) \in P$ 
  shows  $\exists \sigma'. (l, \sigma') \in Q \wedge (\sigma, \sigma') \in S$ 
  using hyperprop-FU-def assms(1) assms(2) by fastforce

theorem FU-expresses-hyperproperties:
  hypersat C (hyperprop-FU P Q)  $\longleftrightarrow$  FU P C Q (is ?A  $\longleftrightarrow$  ?B)
proof
  assume ?A
  show ?B
  proof (rule FUI)
    fix  $\sigma$  l assume  $(l, \sigma) \in P$ 
    then obtain  $\sigma'$  where  $\text{asm0}: (l, \sigma') \in Q \wedge (\sigma, \sigma') \in \text{set-of-traces } C$ 
      by (meson ‹hypersat C (hyperprop-FU P Q)› hyperprop-FUE hypersat-def)
    then show  $\exists \sigma'. ((C, \sigma) \rightarrow \sigma') \wedge (l, \sigma') \in Q$ 
      using in-set-of-traces by blast
  qed
next
  assume ?B
  have hyperprop-FU P Q (set-of-traces C)
  proof (rule hyperprop-FUI)

```

```

fix  $l \sigma$ 
assume  $asm0: (l, \sigma) \in P$ 
then show  $\exists \sigma'. (l, \sigma') \in Q \wedge (\sigma, \sigma') \in \text{set-of-traces } C$ 
  by (metis (mono-tags, lifting) CollectI <FU P C Q> FU-def set-of-traces-def)
qed
then show ?A
  by (simp add: hypersat-def)
qed

```

```

theorem hyperliveness-hyperprop-FU:
  assumes  $\bigwedge l. \text{sat-for-ll } P \implies \text{sat-for-ll } Q$ 
  shows hyperliveness (hyperprop-FU P Q)
proof (rule hyperlivenessI)
  fix  $S$  show  $\exists S'. S \subseteq S' \wedge \text{hyperprop-FU } P \ Q \ S'$ 
  by (meson UNIV-I hyperprop-FU-def assms sat-for-ll-def subsetI)
qed

```

No relationship between incorrectness and forward underapproximation lemma *incorrectness-does-not-imply-FU*:

```

assumes injective from-nat
assumes  $P = \{(l, \sigma) \mid \sigma \ l. \ \sigma \ x = \text{from-nat } (0 :: \text{nat}) \vee \sigma \ x = \text{from-nat } 1\}$ 
  and  $Q = \{(l, \sigma) \mid \sigma \ l. \ \sigma \ x = \text{from-nat } 1\}$ 
  and  $C = \text{Assume } (\lambda \sigma. \ \sigma \ x = \text{from-nat } 1)$ 
shows IL P C Q
  and  $\neg \text{FU } P \ C \ Q$ 
proof –
  have  $Q \subseteq \text{sem } C \ P$ 
proof (rule subsetPairI)
  fix  $l \sigma$  assume  $(l, \sigma) \in Q$ 
  then have  $\sigma \ x = \text{from-nat } 1$ 
  using assms(3) by blast
  then have  $(l, \sigma) \in P$ 
  by (simp add: assms(2))
  then show  $(l, \sigma) \in \text{sem } C \ P$ 
  by (simp add: <\sigma x = from-nat 1> assms(4) sem-assume)
qed
then show IL P C Q
  by (simp add: IL-def)
show  $\neg \text{FU } P \ C \ Q$ 
proof (rule ccontr)
  assume  $\neg \neg \text{FU } P \ C \ Q$ 
  then have  $\text{FU } P \ C \ Q$ 
  by blast
  obtain  $\sigma$  where  $\sigma \ x = \text{from-nat } 0$ 
  by simp
  then obtain  $l$  where  $(l, \sigma) \in P$ 
  using assms(2) by blast
  then obtain  $\sigma'$  where single-sem C \sigma \sigma' (l, \sigma') \in Q
  by (meson <FU P C Q> FU-def)

```

then have $\sigma' x = \text{from-nat } 0$
using $\langle \sigma x = \text{from-nat } 0 \rangle \text{ assms}(4)$ **by** *blast*
then have $\text{from-nat } 0 = \text{from-nat } 1$
using $\langle \langle C, \sigma \rangle \rightarrow \sigma' \rangle \text{ assms}(4)$ **by** *force*
then show *False*
using $\text{assms}(1)$ *injective-def[of from-nat]* **by** *auto*
qed
qed

lemma *FU-does-not-imply-incorrectness*:
assumes $P = \{(l, \sigma) \mid \sigma l. \sigma x = \text{from-nat } (0 :: \text{nat}) \vee \sigma x = \text{from-nat } 1\}$
and $Q = \{(l, \sigma) \mid \sigma l. \sigma x = \text{from-nat } 1\}$
assumes *injective from-nat*
shows $\neg \text{IL } Q \text{ Skip } P$
and $\text{FU } Q \text{ Skip } P$
proof –
show $\text{FU } Q \text{ Skip } P$
proof (*rule FUI*)
fix σl
assume $(l, \sigma) \in Q$
then show $\exists \sigma'. (\langle \text{Skip}, \sigma \rangle \rightarrow \sigma') \wedge (l, \sigma') \in P$
using *SemSkip assms(1) assms(2)* **by** *fastforce*
qed
obtain σ **where** $\sigma x = \text{from-nat } 0$
by *simp*
then obtain l **where** $(l, \sigma) \in P$
using $\text{assms}(1)$ **by** *blast*
moreover have $\sigma x \neq \text{from-nat } 1$
by (*metis* $\langle \sigma x = \text{from-nat } 0 \rangle \text{ assms}(3)$ *injective-def one-neq-zero*)
then have $(l, \sigma) \notin Q$
using $\text{assms}(2)$ **by** *blast*
ultimately show $\neg \text{IL } Q \text{ Skip } P$
using *IL-consequence* **by** *blast*
qed

3.7 k-Forward Underapproximate logic

RFU is the old name of k-FU.

Definition 21 *definition RFU where*

$$\text{RFU } P \ C \ Q \longleftrightarrow (\forall \text{states} \in P. \exists \text{states}' \in Q. \text{k-sem } C \ \text{states} \ \text{states}')$$

lemma *RFUI*:

assumes $\bigwedge \text{states}. \text{states} \in P \implies (\exists \text{states}' \in Q. \text{k-sem } C \ \text{states} \ \text{states}')$
shows $\text{RFU } P \ C \ Q$
by (*simp add: assms RFU-def*)

lemma *RFUE*:

assumes $\text{RFU } P \ C \ Q$

and $states \in P$
shows $\exists states' \in Q. k\text{-sem } C \text{ states } states'$
using $assms(1) \ assms(2) \ RFU\text{-def}$ **by** $blast$

definition $encode\text{-RFU}$ **where**

$encode\text{-RFU}$ $from\text{-nat } x \ P \ S \longleftrightarrow (\exists states \in P. (\forall i. \text{states } i \in S \wedge \text{fst } (\text{states } i) \ x = \text{from-nat } i))$

Proposition 11 **theorem** $encode\text{-RFU}$:

assumes $not\text{-free-var-of } P \ x$

and $not\text{-free-var-of } Q \ x$

and $injective \ from\text{-nat}$

shows $RFU \ P \ C \ Q \longleftrightarrow \models \{encode\text{-RFU } from\text{-nat } x \ P\} \ C \ \{encode\text{-RFU}$

$from\text{-nat } x \ Q\}$

(**is** $?A \longleftrightarrow ?B$)

proof

assume $?A$

show $?B$

proof ($rule \ hyper\text{-hoare-tripleI}$)

fix S **assume** $encode\text{-RFU } from\text{-nat } x \ P \ S$

then obtain $states$ **where** $asm0: \text{states} \in P \ \bigwedge i. \text{states } i \in S \wedge \text{fst } (\text{states } i) \ x = \text{from-nat } i$

by ($meson \ encode\text{-RFU-def}$)

then obtain $states'$ **where** $states' \in Q \ k\text{-sem } C \ \text{states } states'$

using $\langle RFU \ P \ C \ Q \rangle \ RFUE$ **by** $blast$

then have $\bigwedge i. \text{states}' i \in \text{sem } C \ S \wedge \text{fst } (\text{states}' i) \ x = \text{from-nat } i$

by ($metis \ (mono\text{-tags, lifting}) \ asm0(2) \ in\text{-sem } k\text{-sem-def } prod.\text{collapse}$)

then show $encode\text{-RFU } from\text{-nat } x \ Q \ (\text{sem } C \ S)$

by ($meson \ \langle \text{states}' \in Q \rangle \ encode\text{-RFU-def}$)

qed

next

assume $?B$

show $?A$

proof ($rule \ RFUI$)

fix $states$ **assume** $asm0: \text{states} \in P$

let $?states = \lambda i. ((\text{fst } (\text{states } i))(x := \text{from-nat } i), \text{snd } (\text{states } i))$

have $?states \in P$

using $assms(1)$

proof ($rule \ not\text{-free-var-ofE}$)

show $states \in P$ **using** $asm0$ **by** $simp$

fix i **show** $differ\text{-only-by } (\text{fst } (\text{states } i)) \ (\text{fst } ((\text{fst } (\text{states } i))(x := \text{from-nat } i), \text{snd } (\text{states } i))) \ x$

using $diff\text{-by-comm } diff\text{-by-update}$ **by** $fastforce$

qed ($auto$)

then have $encode\text{-RFU } from\text{-nat } x \ P \ (\text{range } ?states)$

using $encode\text{-RFU-def}$ **by** $fastforce$

then have $encode\text{-RFU } from\text{-nat } x \ Q \ (\text{sem } C \ (\text{range } ?states))$

using $\langle \models \{encode\text{-RFU } from\text{-nat } x \ P\} \ C \ \{encode\text{-RFU } from\text{-nat } x \ Q\} \rangle$

hyper-hoare-tripleE **by** *blast*

then obtain *states'* **where** *states'-def*: $states' \in Q \wedge i. states' i \in sem\ C$
 (*range* *?states*) $\wedge fst\ (states'\ i)\ x = from-nat\ i$
by (*meson encode-RFU-def*)

let *?states' = λi. ((fst (states' i))(x := fst (states i) x), snd (states' i))*

have *?states' ∈ Q*

using *assms(2)*

proof (*rule not-free-var-ofE*)

show $states' \in Q$ **using** $\langle states' \in Q \rangle$ **by** *simp*

fix *i* **show** *differ-only-by* ($fst\ (states'\ i)$) ($fst\ ((fst\ (states'\ i))(x := fst\ (states\ i)\ x),\ snd\ (states'\ i))$) *x*

using *diff-by-comm diff-by-update* **by** *fastforce*

qed (*auto*)

moreover obtain *to-pvar* **where** *to-pvar-def*: $\wedge i. to-pvar\ (from-nat\ i) = i$

using *assms(3) injective-then-exists-inverse* **by** *blast*

then have *inj*: $\wedge i\ j. from-nat\ i = from-nat\ j \implies i = j$

by *metis*

moreover have *k-sem C states ?states'*

proof (*rule k-semI*)

fix *i*

obtain σ **where** ($fst\ (states'\ i), \sigma \in range\ (\lambda i. ((fst\ (states\ i))(x := from-nat\ i),\ snd\ (states\ i))) \wedge \langle C, \sigma \rangle \rightarrow snd\ (states'\ i)$)

using *states'-def(2) in-sem* **by** *blast*

moreover have $fst\ (states'\ i)\ x = from-nat\ i$

by (*simp add: states'-def*)

then have *r*: ($(fst\ (states\ (inv\ ?states\ (fst\ (states'\ i), \sigma))))$)

($x := from-nat\ (inv\ ?states\ (fst\ (states'\ i), \sigma)),\ snd\ (states\ (inv\ ?states\ (fst\ (states'\ i), \sigma)))$)

$= (fst\ (states'\ i), \sigma)$

by (*metis (mono-tags, lifting) calculation f-inv-into-f*)

then have *single-sem C (snd (states i)) (snd (states' i))*

using $\langle fst\ (states'\ i)\ x = from-nat\ i \rangle$ *calculation inj* **by** *fastforce*

moreover have $fst\ (?states'\ i) = fst\ (states'\ i)$

by (*metis (mono-tags, lifting)r*) $\langle fst\ (states'\ i)\ x = from-nat\ i \rangle$ *fst-conv fun-upd-same inj*

ultimately show $fst\ (states\ i) = fst\ ((fst\ (states'\ i))(x := fst\ (states\ i)\ x),\ snd\ (states'\ i)) \wedge$

$\langle C, snd\ (states\ i) \rangle \rightarrow snd\ ((fst\ (states'\ i))(x := fst\ (states\ i)\ x),\ snd\ (states'\ i))$

by (*metis (mono-tags, lifting) fst-conv fun-upd-triv fun-upd-upd snd-conv*)

qed

ultimately show $\exists states' \in Q. k-sem\ C\ states\ states'$ **by** *blast*

qed

qed

definition *RFU-hyperprop* **where**

$RFU\text{-hyperprop } P Q S \iff (\forall l \text{ states}. (\lambda i. (l \ i, \text{states } i)) \in P$
 $\longrightarrow (\exists \text{states}' . (\lambda i. (l \ i, \text{states}' \ i)) \in Q \wedge (\forall i. (\text{states } i, \text{states}' \ i) \in S)))$

lemma *RFU-hyperpropI*:

assumes $\bigwedge l \text{ states}. (\lambda i. (l \ i, \text{states } i)) \in P \implies (\exists \text{states}' . (\lambda i. (l \ i, \text{states}' \ i)) \in Q \wedge (\forall i. (\text{states } i, \text{states}' \ i) \in S))$

shows *RFU-hyperprop* $P Q S$

by (*simp add: assms RFU-hyperprop-def*)

lemma *RFU-hyperpropE*:

assumes *RFU-hyperprop* $P Q S$

and $(\lambda i. (l \ i, \text{states } i)) \in P$

shows $\exists \text{states}' . (\lambda i. (l \ i, \text{states}' \ i)) \in Q \wedge (\forall i. (\text{states } i, \text{states}' \ i) \in S)$

using *assms(1) assms(2) RFU-hyperprop-def* **by** *blast*

Proposition 10 **theorem** *RFU-captures-hyperproperties*:

hypersat $C (RFU\text{-hyperprop } P Q) \iff RFU \ P \ C \ Q$ (**is** $?A \iff ?B$)

proof

assume $?A$

show $?B$

proof (*rule RFUI*)

fix *states* **assume** $\text{states} \in P$

moreover have $(\lambda i. ((fst \circ \text{states}) \ i, (snd \circ \text{states}) \ i)) = \text{states}$ **by** *simp*

ultimately obtain *states'* **where** *asm0*: $(\lambda i. ((fst \circ \text{states}) \ i, \text{states}' \ i)) \in Q$

$\wedge i. ((snd \circ \text{states}) \ i, \text{states}' \ i) \in \text{set-of-traces } C$

using *RFU-hyperpropE*[*of P Q set-of-traces C fst \circ states snd \circ states*]

using $\langle \text{hypersat } C (RFU\text{-hyperprop } P Q) \rangle$ *hypersat-def* **by** *auto*

moreover have *k-sem* $C \ \text{states} (\lambda i. ((fst \circ \text{states}) \ i, \text{states}' \ i))$

proof (*rule k-semI*)

fix *i*

have $\langle C, snd \ (\text{states } i) \rangle \rightarrow \text{states}' \ i$

using *calculation(2) in-set-of-traces* **by** *fastforce*

then show $fst \ (\text{states } i) = fst \ ((fst \circ \text{states}) \ i, \text{states}' \ i) \wedge \langle C, snd \ (\text{states } i) \rangle \rightarrow snd \ ((fst \circ \text{states}) \ i, \text{states}' \ i)$

by *simp*

qed

ultimately show $\exists \text{states}' \in Q. \ \text{k-sem } C \ \text{states } \text{states}'$

by *fast*

qed

next

assume $?B$

have *RFU-hyperprop* $P Q$ (*set-of-traces C*)

proof (*rule RFU-hyperpropI*)

fix *l states*

assume $(\lambda i. (l \ i, \text{states } i)) \in P$

then obtain *states'* **where** *asm0*: $\text{states}' \in Q \ \text{k-sem } C \ (\lambda i. (l \ i, \text{states } i))$

using $\langle RFU \ P \ C \ Q \rangle$ *RFUE* **by** *blast*

then have $\bigwedge i. \text{fst} (\text{states}' i) = l i$
by (*simp add: k-sem-def*)
moreover have $(\lambda i. (l i, (\text{snd} \circ \text{states}' i))) = \text{states}'$
proof (*rule ext*)
fix i **show** $(l i, (\text{snd} \circ \text{states}' i)) = \text{states}' i$
by (*metis calculation comp-apply surjective-pairing*)
qed
moreover have $\bigwedge i. (\text{states} i, (\text{snd} \circ \text{states}' i)) \in \text{set-of-traces } C$
proof –
fix i **show** $(\text{states} i, (\text{snd} \circ \text{states}' i)) \in \text{set-of-traces } C$
using *asm0(2) comp-apply[of snd states'] in-set-of-traces k-sem-def[of C lambda.*
(l i, states i) states'] snd-conv
by *fastforce*
qed
ultimately show $\exists \text{states}'. (\lambda i. (l i, \text{states}' i)) \in Q \wedge (\forall i. (\text{states} i, \text{states}' i) \in \text{set-of-traces } C)$
using *asm0(1) by force*
qed
then show ?A
by (*simp add: hypersat-def*)
qed

theorem *hyperliveness-encode-RFU*:
assumes $\bigwedge l. k\text{-sat-for-}l \ l \ P \implies k\text{-sat-for-}l \ l \ Q$
shows *hyperliveness (RFU-hyperprop P Q)*
proof (*rule hyperlivenessI*)
fix S
have *RFU-hyperprop P Q UNIV*
proof (*rule RFU-hyperpropI*)
fix l **states** **assume** *asm0: (lambda i. (l i, states i)) in P*
then obtain states' **where** $(\lambda i. (l i, \text{states}' i)) \in Q$
by (*metis assms k-sat-for-l-def*)
then show $\exists \text{states}'. (\lambda i. (l i, \text{states}' i)) \in Q \wedge (\forall i. (\text{states} i, \text{states}' i) \in \text{UNIV})$
by *blast*
qed
then show $\exists S'. S \subseteq S' \wedge \text{RFU-hyperprop } P \ Q \ S'$
by *blast*
qed

3.8 k-Universal Existential (RUE) [5]

RUE is the old name of k-UE.

Definition 22 *definition RUE where*

RUE P C Q $\longleftrightarrow (\forall (\sigma 1, \sigma 2) \in P. \forall \sigma 1'. k\text{-sem } C \ \sigma 1 \ \sigma 1' \longrightarrow (\exists \sigma 2'. k\text{-sem } C \ \sigma 2 \ \sigma 2' \wedge (\sigma 1', \sigma 2') \in Q))$

lemma *RUE-I*:

assumes $\bigwedge \sigma 1 \ \sigma 2 \ \sigma 1'. (\sigma 1, \sigma 2) \in P \implies k\text{-sem } C \ \sigma 1 \ \sigma 1' \implies (\exists \sigma 2'. k\text{-sem } C \ \sigma 2 \ \sigma 2' \wedge (\sigma 1', \sigma 2') \in Q)$
shows $RUE \ P \ C \ Q$
using *assms RUE-def* **by** *fastforce*

lemma *RUE-E*:

assumes $RUE \ P \ C \ Q$
and $(\sigma 1, \sigma 2) \in P$
and $k\text{-sem } C \ \sigma 1 \ \sigma 1'$
shows $\exists \sigma 2'. k\text{-sem } C \ \sigma 2 \ \sigma 2' \wedge (\sigma 1', \sigma 2') \in Q$
using *RUE-def* *assms(1)* *assms(2)* *assms(3)* **by** *blast*

Hyperproperty definition *hyperprop-RUE* **where**

$hyperprop\text{-}RUE \ P \ Q \ S \longleftrightarrow (\forall l1 \ l2 \ \sigma 1 \ \sigma 2 \ \sigma 1'. (\lambda i. (l1 \ i, \sigma 1 \ i), \lambda k. (l2 \ k, \sigma 2 \ k)) \in P \wedge$
 $(\forall i. (\sigma 1 \ i, \sigma 1' \ i) \in S) \longrightarrow (\exists \sigma 2'. (\forall k. (\sigma 2 \ k, \sigma 2' \ k) \in S) \wedge (\lambda i. (l1 \ i, \sigma 1' \ i), \lambda k. (l2 \ k, \sigma 2' \ k)) \in Q))$

lemma *hyperprop-RUE-I*:

assumes $\bigwedge l1 \ l2 \ \sigma 1 \ \sigma 2 \ \sigma 1'. (\lambda i. (l1 \ i, \sigma 1 \ i), \lambda k. (l2 \ k, \sigma 2 \ k)) \in P \implies$
 $(\forall i. (\sigma 1 \ i, \sigma 1' \ i) \in S) \implies (\exists \sigma 2'. (\forall k. (\sigma 2 \ k, \sigma 2' \ k) \in S) \wedge (\lambda i. (l1 \ i, \sigma 1' \ i), \lambda k. (l2 \ k, \sigma 2' \ k)) \in Q)$
shows $hyperprop\text{-}RUE \ P \ Q \ S$
using *assms hyperprop-RUE-def[of P Q S]*
by *force*

lemma *hyperprop-RUE-E*:

assumes $hyperprop\text{-}RUE \ P \ Q \ S$
and $(\lambda i. (l1 \ i, \sigma 1 \ i), \lambda k. (l2 \ k, \sigma 2 \ k)) \in P$
and $\bigwedge i. (\sigma 1 \ i, \sigma 1' \ i) \in S$
shows $\exists \sigma 2'. (\forall k. (\sigma 2 \ k, \sigma 2' \ k) \in S) \wedge (\lambda i. (l1 \ i, \sigma 1' \ i), \lambda k. (l2 \ k, \sigma 2' \ k)) \in Q$
using *assms(1)* *assms(2)* *assms(3)* *hyperprop-RUE-def* **by** *blast*

Proposition 12 theorem *RUE-express-hyperproperties*:

$RUE \ P \ C \ Q \longleftrightarrow hypersat \ C \ (hyperprop\text{-}RUE \ P \ Q) \ (\text{is } ?A \longleftrightarrow ?B)$

proof

assume *asm0*: $?A$
have $hyperprop\text{-}RUE \ P \ Q$ (*set-of-traces C*)
proof (*rule hyperprop-RUE-I*)
fix $l1 \ l2 \ \sigma 1 \ \sigma 2 \ \sigma 1'$
assume *asm1*: $(\lambda i. (l1 \ i, \sigma 1 \ i), \lambda k. (l2 \ k, \sigma 2 \ k)) \in P \ \forall i. (\sigma 1 \ i, \sigma 1' \ i) \in$
set-of-traces C

let $?x1 = \lambda i. (l1 \ i, \sigma 1 \ i)$
let $?x2 = \lambda k. (l2 \ k, \sigma 2 \ k)$

let $?x1' = \lambda i. (l1 \ i, \sigma 1' \ i)$

```

have  $\exists \sigma 2'. k\text{-sem } C (\lambda k. (l2\ k, \sigma 2\ k)) \sigma 2' \wedge (?x1', \sigma 2') \in Q$ 
  using asm0 asm1(1)
proof (rule RUE-E)
  show  $k\text{-sem } C (\lambda i. (l1\ i, \sigma 1\ i)) (\lambda i. (l1\ i, \sigma 1' i))$ 
  proof (rule k-semI)
    fix i
    have  $single\text{-sem } C (\sigma 1\ i) (\sigma 1' i)$  using asm1(2)
      by (simp add: set-of-traces-def)
    then show  $fst (l1\ i, \sigma 1\ i) = fst (l1\ i, \sigma 1' i) \wedge \langle C, snd (l1\ i, \sigma 1\ i) \rangle \rightarrow snd$ 
      ( $l1\ i, \sigma 1' i$ )
      by simp
    qed
  qed
then obtain  $\sigma 2'$  where asm2: k-sem C (λk. (l2 k, σ2 k)) σ2' (?x1', σ2') ∈ Q
  by blast
let  $?\sigma 2' = \lambda k. snd (\sigma 2' k)$ 

have  $\bigwedge k. (\sigma 2\ k, ?\sigma 2' k) \in set\text{-of-traces } C$ 
  by (metis (mono-tags, lifting) asm2(1) in-set-of-traces k-sem-def snd-conv)
moreover have  $\lambda k. (l2\ k, ?\sigma 2' k) = \sigma 2'$ 
proof (rule ext)
  fix k show  $(l2\ k, snd (\sigma 2' k)) = \sigma 2' k$ 
  by (metis (mono-tags, lifting) asm2(1) fst-eqD k-sem-def surjective-pairing)
qed
ultimately show  $\exists \sigma 2'. (\forall k. (\sigma 2\ k, \sigma 2' k) \in set\text{-of-traces } C) \wedge (\lambda i. (l1\ i, \sigma 1' i), \lambda k. (l2\ k, \sigma 2' k)) \in Q$ 
  using asm2(2) by fastforce
qed
then show  $?B$ 
  by (simp add: hypersat-def)
next
assume  $?B$  then have asm0: hyperprop-RUE P Q (set-of-traces C)
  by (simp add: hypersat-def)
show  $?A$ 
proof (rule RUE-I)
  fix  $\sigma 1\ \sigma 2\ \sigma 1'$ 
  assume asm1: (σ1, σ2) ∈ P k-sem C σ1 σ1'
  then have  $\bigwedge i. fst (\sigma 1\ i) = fst (\sigma 1' i)$ 
  by (simp add: k-sem-def)
  have  $\exists \sigma 2'. (\forall k. (snd (\sigma 2\ k), \sigma 2' k) \in set\text{-of-traces } C) \wedge (\lambda i. (fst (\sigma 1\ i), snd$ 
    ( $\sigma 1' i$ )),  $\lambda k. (fst (\sigma 2\ k), \sigma 2' k)) \in Q$ 
  using asm0
  proof (rule hyperprop-RUE-E[of P Q set-of-traces C λi. fst (σ1 i) λi. snd (σ1 i) λk. fst (σ2 k) λk. snd (σ2 k) λi. snd (σ1' i)])
    show  $(\lambda i. (fst (\sigma 1\ i), snd (\sigma 1\ i)), \lambda k. (fst (\sigma 2\ k), snd (\sigma 2\ k))) \in P$ 
    by (simp add: asm1(1))
    fix i show  $(snd (\sigma 1\ i), snd (\sigma 1' i)) \in set\text{-of-traces } C$ 
    by (metis (mono-tags, lifting) CollectI asm1(2) k-sem-def set-of-traces-def)
  qed

```

qed
then obtain $\sigma 2'$ **where** $asm2: \bigwedge k. (snd (\sigma 2 k), \sigma 2' k) \in set-of-traces C (\lambda i. (fst (\sigma 1 i), snd (\sigma 1' i)), \lambda k. (fst (\sigma 2 k), \sigma 2' k)) \in Q$
by *blast*
moreover have $k-sem C \sigma 2 (\lambda k. (fst (\sigma 2 k), \sigma 2' k))$
proof (*rule k-semI*)
fix i **show** $fst (\sigma 2 i) = fst (fst (\sigma 2 i), \sigma 2' i) \wedge \langle C, snd (\sigma 2 i) \rangle \rightarrow snd (fst (\sigma 2 i), \sigma 2' i)$
using *calculation(1) in-set-of-traces by auto*
qed
ultimately show $\exists \sigma 2'. k-sem C \sigma 2 \sigma 2' \wedge (\sigma 1', \sigma 2') \in Q$
using $\langle \bigwedge i. fst (\sigma 1 i) = fst (\sigma 1' i) \rangle$ **by** *auto*
qed
qed

definition *is-type* **where**

$is-type\ type\ fn\ x\ t\ S\ \sigma \longleftrightarrow (\forall i. \sigma\ i \in S \wedge fst (\sigma\ i)\ t = type \wedge fst (\sigma\ i)\ x = fn\ i)$

lemma *is-typeI*:

assumes $\bigwedge i. \sigma\ i \in S$
and $\bigwedge i. fst (\sigma\ i)\ t = type$
and $\bigwedge i. fst (\sigma\ i)\ x = fn\ i$
shows $is-type\ type\ fn\ x\ t\ S\ \sigma$
by (*simp add: assms(1) assms(2) assms(3) is-type-def*)

lemma *is-type-E*:

assumes $is-type\ type\ fn\ x\ t\ S\ \sigma$
shows $\sigma\ i \in S \wedge fst (\sigma\ i)\ t = type \wedge fst (\sigma\ i)\ x = fn\ i$
by (*meson assms is-type-def*)

definition *encode-RUE-1* **where**

$encode-RUE-1\ fn\ fn1\ fn2\ x\ t\ P\ S \longleftrightarrow (\forall k. \exists \sigma \in S. fst\ \sigma\ x = fn2\ k \wedge fst\ \sigma\ t = fn\ 2)$
 $\wedge (\forall \sigma\ \sigma'. is-type\ (fn\ 1)\ fn1\ x\ t\ S\ \sigma \wedge is-type\ (fn\ 2)\ fn2\ x\ t\ S\ \sigma' \rightarrow (\sigma, \sigma') \in P)$

lemma *encode-RUE-1-I*:

assumes $\bigwedge k. \exists \sigma \in S. fst\ \sigma\ x = fn2\ k \wedge fst\ \sigma\ t = fn\ 2$
and $\bigwedge \sigma\ \sigma'. is-type\ (fn\ 1)\ fn1\ x\ t\ S\ \sigma \wedge is-type\ (fn\ 2)\ fn2\ x\ t\ S\ \sigma' \Rightarrow (\sigma, \sigma') \in P$
shows $encode-RUE-1\ fn\ fn1\ fn2\ x\ t\ P\ S$
by (*simp add: assms(1) assms(2) encode-RUE-1-def*)

lemma *encode-RUE-1-E1*:

assumes $encode-RUE-1\ fn\ fn1\ fn2\ x\ t\ P\ S$
shows $\exists \sigma \in S. fst\ \sigma\ x = fn2\ k \wedge fst\ \sigma\ t = fn\ 2$
by (*meson assms encode-RUE-1-def*)

lemma *encode-RUE-1-E2*:
assumes *encode-RUE-1* *fn fn1 fn2 x t P S*
and *is-type (fn 1) fn1 x t S σ*
and *is-type (fn 2) fn2 x t S σ'*
shows $(\sigma, \sigma') \in P$
by (*meson assms encode-RUE-1-def*)

definition *encode-RUE-2* **where**
encode-RUE-2 *fn fn1 fn2 x t Q S* $\longleftrightarrow (\forall \sigma. \text{is-type } (fn\ 1) \text{ } fn1\ x\ t\ S\ \sigma \longrightarrow (\exists \sigma'. \text{is-type } (fn\ 2) \text{ } fn2\ x\ t\ S\ \sigma' \wedge (\sigma, \sigma') \in Q))$

lemma *encode-RUE-2I*:
assumes $\bigwedge \sigma. \text{is-type } (fn\ 1) \text{ } fn1\ x\ t\ S\ \sigma \implies (\exists \sigma'. \text{is-type } (fn\ 2) \text{ } fn2\ x\ t\ S\ \sigma' \wedge (\sigma, \sigma') \in Q)$
shows *encode-RUE-2* *fn fn1 fn2 x t Q S*
by (*simp add: assms encode-RUE-2-def*)

lemma *encode-RUE-2-E*:
assumes *encode-RUE-2* *fn fn1 fn2 x t Q S*
and *is-type (fn 1) fn1 x t S σ*
shows $\exists \sigma'. \text{is-type } (fn\ 2) \text{ } fn2\ x\ t\ S\ \sigma' \wedge (\sigma, \sigma') \in Q$
by (*meson assms(1) assms(2) encode-RUE-2-def*)

definition *differ-only-by-set* **where**
differ-only-by-set *vars a b* $\longleftrightarrow (\forall x. x \notin \text{vars} \longrightarrow a\ x = b\ x)$

definition *differ-only-by-lset* **where**
differ-only-by-lset *vars a b* $\longleftrightarrow (\forall i. \text{snd } (a\ i) = \text{snd } (b\ i) \wedge \text{differ-only-by-set } \text{vars } (\text{fst } (a\ i)) (\text{fst } (b\ i)))$

lemma *differ-only-by-lsetI*:
assumes $\bigwedge i. \text{snd } (a\ i) = \text{snd } (b\ i)$
and $\bigwedge i. \text{differ-only-by-set } \text{vars } (\text{fst } (a\ i)) (\text{fst } (b\ i))$
shows *differ-only-by-lset* *vars a b*
using *assms(1) assms(2) differ-only-by-lset-def* **by** *blast*

definition *not-in-free-vars-double* **where**
not-in-free-vars-double *vars P* $\longleftrightarrow (\forall \sigma \sigma'. \text{differ-only-by-lset } \text{vars } (\text{fst } \sigma) (\text{fst } \sigma') \wedge \text{differ-only-by-lset } \text{vars } (\text{snd } \sigma) (\text{snd } \sigma') \longrightarrow (\sigma \in P \longleftrightarrow \sigma' \in P))$

lemma *not-in-free-vars-doubleE*:
assumes *not-in-free-vars-double* *vars P*
and *differ-only-by-lset* *vars (fst σ) (fst σ')*
and *differ-only-by-lset* *vars (snd σ) (snd σ')*
and $\sigma \in P$
shows $\sigma' \in P$
by (*meson assms not-in-free-vars-double-def*)

Proposition 13 *theorem encoding-RUE:*

assumes injective fn \wedge injective fn1 \wedge injective fn2

and $t \neq x$

and injective (fn :: nat \Rightarrow 'a)

and injective fn1

and injective fn2

and not-in-free-vars-double {x, t} P

and not-in-free-vars-double {x, t} Q

*shows RUE P C Q \longleftrightarrow \models {encode-RUE-1 fn fn1 fn2 x t P} C {encode-RUE-2
fn fn1 fn2 x t Q}*

(is ?A \longleftrightarrow ?B)

proof

assume asm0: ?A

show ?B

proof (*rule hyper-hoare-tripleI*)

fix S assume asm1: encode-RUE-1 fn fn1 fn2 x t P S

show encode-RUE-2 fn fn1 fn2 x t Q (sem C S)

proof (*rule encode-RUE-2I*)

fix $\sigma 1'$ assume asm2: is-type (fn 1) fn1 x t (sem C S) $\sigma 1'$

let $?\sigma 2 = \lambda k. \text{SOME } \sigma'. \sigma' \in S \wedge \text{fst } \sigma' x = \text{fn2 } k \wedge \text{fst } \sigma' t = \text{fn } 2$

have r2: $\bigwedge k. ?\sigma 2 k \in S \wedge \text{fst } (?\sigma 2 k) x = \text{fn2 } k \wedge \text{fst } (?\sigma 2 k) t = \text{fn } 2$

proof –

fix k

show $?\sigma 2 k \in S \wedge \text{fst } (?\sigma 2 k) x = \text{fn2 } k \wedge \text{fst } (?\sigma 2 k) t = \text{fn } 2$

proof (*rule someI-ex*)

show $\exists xa. xa \in S \wedge \text{fst } xa x = \text{fn2 } k \wedge \text{fst } xa t = \text{fn } 2$

by (meson asm1 encode-RUE-1-E1)

qed

qed

let $?\sigma 1 = \lambda i. \text{SOME } \sigma. (\text{fst } (\sigma 1' i), \sigma) \in S \wedge \text{single-sem } C \sigma (\text{snd } (\sigma 1' i))$

have r1: $\bigwedge i. (\text{fst } (\sigma 1' i), ?\sigma 1 i) \in S \wedge \text{single-sem } C (?\sigma 1 i) (\text{snd } (\sigma 1' i))$

proof –

fix i

show $(\text{fst } (\sigma 1' i), ?\sigma 1 i) \in S \wedge \text{single-sem } C (?\sigma 1 i) (\text{snd } (\sigma 1' i))$

***proof** (rule someI-ex[*of* $\lambda \sigma. (\text{fst } (\sigma 1' i), \sigma) \in S \wedge \text{single-sem } C \sigma (\text{snd } (\sigma 1' i))$])*

show $\exists \sigma. (\text{fst } (\sigma 1' i), \sigma) \in S \wedge \langle C, \sigma \rangle \rightarrow \text{snd } (\sigma 1' i)$

by (meson asm2 in-sem is-type-def)

qed

qed

have $(\lambda i. (\text{fst } (\sigma 1' i), ?\sigma 1 i), ?\sigma 2) \in P$

using asm1

proof (*rule encode-RUE-1-E2*)

show is-type (fn 1) fn1 x t S $(\lambda i. (\text{fst } (\sigma 1' i), ?\sigma 1 i))$

```

      using asm2 fst-conv is-type-def[of fn 1 fn1 x t S] is-type-def[of fn 1 fn1 x
t sem C S] r1
      by force
      show is-type (fn 2) fn2 x t S ?σ2
      by (simp add: is-type-def r2)
    qed
  moreover have  $\exists \sigma 2'. k\text{-sem } C \text{ ?}\sigma 2 \sigma 2' \wedge (\sigma 1', \sigma 2') \in Q$ 
    using asm0
  proof (rule RUE-E)
    show  $(\lambda i. (\text{fst } (\sigma 1' i), \text{?}\sigma 1 i), \text{?}\sigma 2) \in P$ 
      using calculation by auto
    show  $k\text{-sem } C (\lambda i. (\text{fst } (\sigma 1' i), \text{SOME } \sigma. (\text{fst } (\sigma 1' i), \sigma) \in S \wedge \langle C, \sigma \rangle \rightarrow$ 
snd  $(\sigma 1' i))) \sigma 1'$ 
      by (simp add: k-sem-def r1)
    qed
  then obtain  $\sigma 2'$  where  $\sigma 2'\text{-def: } k\text{-sem } C \text{ ?}\sigma 2 \sigma 2' \wedge (\sigma 1', \sigma 2') \in Q$  by
blast
  then have is-type (fn 2) fn2 x t (sem C S)  $\sigma 2'$ 
    using in-sem[of - C S] k-semE[of C ?σ2 σ2']
      prod.collapse r2 is-type-def[of fn 2 fn2 x t S] is-type-def[of fn 2 fn2 x t sem
C S]
    by (metis (no-types, lifting))
  then show  $\exists \sigma 2'. \text{is-type } (fn 2) \text{ fn2 } x \text{ t } (sem C S) \sigma 2' \wedge (\sigma 1', \sigma 2') \in Q$ 
    using  $\sigma 2'\text{-def}$  by blast
  qed
next
assume asm0:  $\models \{ \text{encode-RUE-1 } fn \text{ fn1 } fn2 \text{ x t } P \} C \{ \text{encode-RUE-2 } fn \text{ fn1 } fn2$ 
 $x \text{ t } Q \}$ 
show ?A
proof (rule RUE-I)
  fix  $\sigma 1 \sigma 2 \sigma 1'$ 
  assume asm1:  $(\sigma 1, \sigma 2) \in P \text{ k-sem } C \sigma 1 \sigma 1'$ 

  let ?σ1 =  $\lambda i. (((\text{fst } (\sigma 1 i))(t := fn 1))(x := fn1 i), \text{snd } (\sigma 1 i))$ 
  let ?σ2 =  $\lambda k. (((\text{fst } (\sigma 2 k))(t := fn 2))(x := fn2 k), \text{snd } (\sigma 2 k))$ 

  let ?S1 =  $\{ \text{?}\sigma 1 i \mid i. \text{True} \}$ 
  let ?S2 =  $\{ \text{?}\sigma 2 k \mid k. \text{True} \}$ 
  let ?S =  $\text{?}S1 \cup \text{?}S2$ 

  let ?σ1' =  $\lambda i. (((\text{fst } (\sigma 1' i))(t := fn 1))(x := fn1 i), \text{snd } (\sigma 1' i))$ 

  have encode-RUE-2 fn fn1 fn2 x t Q (sem C ?S)
    using asm0
  proof (rule hyper-hoare-tripleE)
    show encode-RUE-1 fn fn1 fn2 x t P ?S
      proof (rule encode-RUE-1-I)
        fix k

```

```

let ?σ = (((fst (σ 2 k))(t := fn 2))(x := fn2 k)), snd (σ 2 k))
have ?σ ∈ ?S2
  by auto
moreover have fst ?σ x = fn2 k
  by simp
moreover have fst ?σ t = fn 2
  by (simp add: assms(2))
ultimately show ∃σ ∈ ?S1 ∪ ?S2. fst σ x = fn2 k ∧ fst σ t = fn 2
  by blast
next
fix σ σ'
  assume asm2: is-type (fn (1 :: nat)) fn1 x t (?S1 ∪ ?S2) σ ∧ is-type (fn
2) fn2 x t (?S1 ∪ ?S2) σ'
  moreover have r1: ∧i. σ i = ((fst (σ 1 i))(t := fn 1, x := fn1 i), snd (σ 1
i))
  proof -
    fix i
    have fst (σ i) t = fn 1
      by (meson calculation is-type-def)
    moreover have σ i ∈ ?S1
    proof (rule ccontr)
      assume ¬ σ i ∈ ?S1
      moreover have σ i ∈ ?S1 ∪ ?S2
        using asm2 is-type-def[of fn 1 fn1 x t]
        by (metis (no-types, lifting))
      ultimately have σ i ∈ ?S2 by simp
      then have fst (σ i) t = fn 2
        using assms(2) by auto
      then show False
        by (metis Suc-1 Suc-eq-numeral ⟨fst (σ i) t = fn 1⟩ assms(3) injective-def
numeral-One one-neq-zero pred-numeral-simps(1))
    qed
    then obtain j where σ i = ((fst (σ 1 j))(t := fn 1, x := fn1 j), snd (σ 1
j))
    by blast
    moreover have i = j
      by (metis (mono-tags, lifting) asm2 assms(4) calculation(2) fst-conv
fun-upd-same injective-def is-type-def)
    ultimately show σ i = ((fst (σ 1 i))(t := fn 1, x := fn1 i), snd (σ 1 i))
      by blast
  qed
moreover have ∧i. σ' i = ((fst (σ 2 i))(t := fn 2, x := fn2 i), snd (σ 2 i))
proof -
  fix i
  have fst (σ' i) t = fn 2
    by (meson calculation is-type-def)
  moreover have σ' i ∈ ?S2
  proof (rule ccontr)
    assume ¬ σ' i ∈ ?S2

```

```

moreover have  $\sigma' i \in ?S1 \cup ?S2$ 
  using asm2 is-type-def[of fn 2 fn2 x t]
  by (metis (no-types, lifting))
ultimately have  $\sigma' i \in ?S1$  by simp
then have  $\text{fst } (\sigma' i) t = \text{fn } 1$ 
  using assms(2) by auto
then show False
by (metis Suc-1 Suc-eq-numeral ‹fst (σ' i) t = fn 2› assms(3) injective-def
numeral-One one-neq-zero pred-numeral-simps(1))
qed
then obtain j where  $\sigma' i = ((\text{fst } (\sigma 2 j))(t := \text{fn } 2, x := \text{fn2 } j), \text{snd } (\sigma 2$ 
j))
  by blast
moreover have  $i = j$ 
  by (metis (mono-tags, lifting) asm2 assms(5) calculation(2) fst-conv
fun-upd-same injective-def is-type-def)
ultimately show  $\sigma' i = ((\text{fst } (\sigma 2 i))(t := \text{fn } 2, x := \text{fn2 } i), \text{snd } (\sigma 2 i))$ 
  by blast
qed
moreover have  $(?\sigma 1, ?\sigma 2) \in P$ 
  using assms(6)
proof (rule not-in-free-vars-doubleE)
  show  $(\sigma 1, \sigma 2) \in P$ 
    by (simp add: asm1(1))
  show differ-only-by-lset  $\{x, t\} (\text{fst } (\sigma 1, \sigma 2)) (\text{fst } (?\sigma 1, ?\sigma 2))$ 
    by (rule differ-only-by-lsetI) (simp-all add: differ-only-by-set-def)
  show differ-only-by-lset  $\{x, t\} (\text{snd } (\sigma 1, \sigma 2)) (\text{snd } (?\sigma 1, ?\sigma 2))$ 
    by (rule differ-only-by-lsetI) (simp-all add: differ-only-by-set-def)
  qed
ultimately show  $(\sigma, \sigma') \in P$ 
  by presburger
qed
qed
then have  $\exists \sigma'. \text{is-type } (\text{fn } 2) \text{ fn2 } x t (\text{sem } C ?S) \sigma' \wedge (?\sigma 1', \sigma') \in Q$ 
proof (rule encode-RUE-2-E)
  show is-type  $(\text{fn } 1) \text{ fn1 } x t (\text{sem } C ?S) ?\sigma 1'$ 
proof (rule is-typeI)
  fix i show  $\text{fst } ((\text{fst } (\sigma 1' i))(t := \text{fn } 1, x := \text{fn1 } i), \text{snd } (\sigma 1' i)) t = \text{fn } 1$ 
    by (simp add: assms(2))
  show  $((\text{fst } (\sigma 1' i))(t := \text{fn } 1, x := \text{fn1 } i), \text{snd } (\sigma 1' i)) \in \text{sem } C ?S$ 
    using UnI1[of  $?S1 ?S2$ ]
    asm1(2) k-semE[of  $C \sigma 1 \sigma 1' i$ ]
    single-step-then-in-sem[of  $C \text{snd } (\sigma 1 i) \text{snd } (\sigma 1' i) - ?S$ ]
    by force
  qed (auto)
qed
then obtain  $\sigma 2'$  where  $r: \text{is-type } (\text{fn } 2) \text{ fn2 } x t (\text{sem } C ?S) \sigma 2' \wedge (?\sigma 1', \sigma 2')$ 
 $\in Q$ 
  by blast

```

```

let ?σ2' = λk. ((fst (σ2' k))(x := fst (σ2 k) x, t := fst (σ2 k) t), snd (σ2' k))
have (σ1', ?σ2') ∈ Q
  using assms(7)
proof (rule not-in-free-vars-doubleE)
  show (?σ1', σ2') ∈ Q
    using r by blast
  show differ-only-by-lset {x, t} (fst (?σ1', σ2')) (fst (σ1', ?σ2'))
    by (rule differ-only-by-lsetI) (simp-all add: differ-only-by-set-def)
  show differ-only-by-lset {x, t} (snd (?σ1', σ2')) (snd (σ1', ?σ2'))
    by (rule differ-only-by-lsetI) (simp-all add: differ-only-by-set-def)
qed
moreover have k-sem C σ2 ?σ2'
proof (rule k-semI)
  fix i
  obtain y where y-def: y ∈ ?S fst y = fst (σ2' i) single-sem C (snd y) (snd
(σ2' i))
    using r in-sem[of σ2' i C ?S]
      is-type-E[of fn 2 fn2 x t sem C ?S σ2' i]
    by (metis (no-types, lifting) fst-conv snd-conv)
  then have fst y t = fn 2
    by (metis (no-types, lifting) is-type-def r)
  moreover have fn 1 ≠ fn 2
    by (metis Suc-1 assms(3) injective-def n-not-Suc-n)
  then have y ∉ ?S1
    using assms(2) calculation by fastforce
  then have y ∈ ?S2
    using y-def(1) by blast
  show fst (σ2 i) = fst ((fst (σ2' i))(x := fst (σ2 i) x, t := fst (σ2 i) t), snd
(σ2' i)) ∧
    ⟨C, snd (σ2 i)⟩ → snd ((fst (σ2' i))(x := fst (σ2 i) x, t := fst (σ2 i) t),
snd (σ2' i))
proof
  have r1: σ2' i ∈ sem C ?S ∧ fst (σ2' i) t = fn 2 ∧ fst (σ2' i) x = fn2 i
proof (rule is-type-E[of fn 2 fn2 x t sem C ?S σ2' i])
  show is-type (fn 2) fn2 x t (sem C ?S) σ2'
    using r by blast
qed
then obtain σ where (fst (σ2' i), σ) ∈ ?S single-sem C σ (snd (σ2' i))
  by (meson in-sem)
then have (fst (σ2' i), σ) ∈ ?S2
  using r1 ⟨fn 1 ≠ fn 2⟩ assms(2) by fastforce
then obtain k where fst (σ2' i) = (fst (σ2 k))(t := fn 2, x := fn2 k) and
σ = snd (σ2 k)
  by blast
then have k = i
  by (metis r1 assms(5) fun-upd-same injective-def)
then show ⟨C, snd (σ2 i)⟩ → snd ((fst (σ2' i))(x := fst (σ2 i) x, t := fst
(σ2 i) t), snd (σ2' i))
  using ⟨⟨C, σ⟩ → snd (σ2' i)⟩ ⟨σ = snd (σ2 k)⟩ by auto

```

show $\text{fst } (\sigma 2 \ i) = \text{fst } ((\text{fst } (\sigma 2' \ i))(x := \text{fst } (\sigma 2 \ i) \ x, t := \text{fst } (\sigma 2 \ i) \ t), \text{snd } (\sigma 2' \ i))$
by (*simp add: $\langle \text{fst } (\sigma 2' \ i) = (\text{fst } (\sigma 2 \ k))(t := \text{fn } 2, x := \text{fn } 2 \ k) \rangle \langle k = i \rangle$*)
qed
qed
ultimately show $\exists \sigma 2'. k\text{-sem } C \ \sigma 2 \ \sigma 2' \wedge (\sigma 1', \sigma 2') \in Q$
by *blast*
qed
qed

3.9 Program Refinement

lemma *sem-assign-single*:

$\text{sem } (\text{Assign } x \ e) \{(l, \sigma)\} = \{(l, \sigma(x := e \ \sigma))\}$ (**is** $?A = ?B$)

proof

show $?A \subseteq ?B$

proof (*rule subsetPairI*)

fix $la \ \sigma'$

assume $(la, \sigma') \in \text{sem } (\text{Assign } x \ e) \{(l, \sigma)\}$

then show $(la, \sigma') \in \{(l, \sigma(x := e \ \sigma))\}$

by (*metis (mono-tags, lifting) in-sem prod.sel(1) prod.sel(2) single-sem-Assign-elim singleton-iff*)

qed

show $?B \subseteq ?A$

by (*simp add: SemAssign in-sem*)

qed

definition *refinement where*

$\text{refinement } C1 \ C2 \longleftrightarrow (\text{set-of-traces } C1 \subseteq \text{set-of-traces } C2)$

definition *not-free-var-stmt where*

$\text{not-free-var-stmt } x \ C \longleftrightarrow (\forall \sigma \ \sigma' \ v. (\sigma, \sigma') \in \text{set-of-traces } C \longrightarrow (\sigma(x := v), \sigma'(x := v)) \in \text{set-of-traces } C)$
 $\wedge (\forall \sigma \ \sigma'. \text{single-sem } C \ \sigma \ \sigma' \longrightarrow \sigma \ x = \sigma' \ x)$

lemma *not-free-var-stmtE-1*:

assumes *not-free-var-stmt* $x \ C$

and $(\sigma, \sigma') \in \text{set-of-traces } C$

shows $(\sigma(x := v), \sigma'(x := v)) \in \text{set-of-traces } C$

using *assms(1) assms(2) not-free-var-stmt-def* **by** *force*

lemma *not-free-in-sem-same-val*:

assumes *not-free-var-stmt* $x \ C$

and *single-sem* $C \ \sigma \ \sigma'$

shows $\sigma \ x = \sigma' \ x$

using *assms(1) assms(2) not-free-var-stmt-def* **by** *fastforce*

lemma *not-free-in-sem-equiv*:

assumes *not-free-var-stmt* $x \ C$

and *single-sem* $C \sigma \sigma'$
shows *single-sem* $C (\sigma(x := v)) (\sigma'(x := v))$
by (*meson* *assms(1)* *assms(2)* *in-set-of-traces* *not-free-var-stmtE-1*)

Example 3 *lemma* *rewrite-if-commute*:

assumes $\models \{ P \} \text{ If } C1 \ C2 \ \{ Q \}$
shows $\models \{ P \} \text{ If } C2 \ C1 \ \{ Q \}$
by (*metis* *assms* *hyper-hoare-triple-def* *sem-if* *sup-commute*)

theorem *encoding-refinement*:

fixes $P :: (('lvar \Rightarrow 'lval) \times ('pvar \Rightarrow 'pval)) \text{ set} \Rightarrow \text{bool}$
assumes $(a :: 'pval) \neq b$

and $P = (\lambda S. \text{card } S = 1)$
and $Q = (\lambda S. \forall \varphi \in S. \text{snd } \varphi \ x = a \longrightarrow (\text{fst } \varphi, (\text{snd } \varphi)(x := b)) \in S)$
and *not-free-var-stmt* $x \ C1$
and *not-free-var-stmt* $x \ C2$
shows *refinement* $C1 \ C2 \longleftrightarrow \models \{ P \} \text{ If } (\text{Seq } (\text{Assign } (x :: 'pvar) (\lambda-. a)) \ C1) (\text{Seq } (\text{Assign } x (\lambda-. b)) \ C2) \ \{ Q \}$
(is $?A \longleftrightarrow ?B$
proof
assume $?A$
show $?B$
proof (*rule* *hyper-hoare-tripleI*)
fix S **assume** $P \ (S :: (('lvar \Rightarrow 'lval) \times ('pvar \Rightarrow 'pval)) \text{ set})$
then obtain $\sigma \ l$ **where** $\text{asm0}: S = \{(l, \sigma)\}$
by (*metis* *assms(2)* *card-1-singletonE* *surj-pair*)
let $?C = \text{If } (\text{Seq } (\text{Assign } x (\lambda-. a)) \ C1) (\text{Seq } (\text{Assign } x (\lambda-. b)) \ C2)$
let $?a = (l, \sigma(x := a))$
let $?b = (l, \sigma(x := b))$
have *if-sem*: $\text{sem } ?C \ S = \text{sem } C1 \ \{?a\} \cup \text{sem } C2 \ \{?b\}$
by (*simp* *add*: *asm0* *sem-assign-single* *sem-if* *sem-seq*)
then have $\bigwedge \varphi. \varphi \in \text{sem } ?C \ S \implies \text{snd } \varphi \ x = a \implies (\text{fst } \varphi, (\text{snd } \varphi)(x := b)) \in \text{sem } ?C \ S$
proof –
fix φ **assume** $\text{asm1}: \varphi \in \text{sem } ?C \ S \ \text{snd } \varphi \ x = a$
have $\varphi \in \text{sem } C1 \ \{?a\}$
proof (*rule* *ccontr*)
assume $\varphi \notin \text{sem } C1 \ \{(l, \sigma(x := a))\}$
then have $\varphi \in \text{sem } C2 \ \{(l, \sigma(x := b))\}$
using *if-sem* *asm1(1)* **by** *force*
then have $\text{snd } \varphi \ x = b$
using *assms(5)* *fun-upd-same* *in-sem* *not-free-in-sem-same-val*[*of* $x \ C2 \ \sigma(x := b) \ \text{snd } \varphi$] *singletonD* *snd-conv*

```

    by metis
  then show False
    using asm1(2) assms(1) by blast
qed
then have  $(\sigma(x := a), \text{snd } \varphi) \in \text{set-of-traces } C1$ 
  by (simp add: in-sem set-of-traces-def)
then have  $(\sigma(x := a), \text{snd } \varphi) \in \text{set-of-traces } C2$ 
  using ‹refinement C1 C2› refinement-def by fastforce
then have  $((\sigma(x := a))(x := b), (\text{snd } \varphi)(x := b)) \in \text{set-of-traces } C2$ 
  by (meson assms(5) not-free-var-stmtE-1)
then have single-sem C2  $(\sigma(x := b)) ((\text{snd } \varphi)(x := b))$ 
  by (simp add: set-of-traces-def)
then have  $(\text{fst } \varphi, (\text{snd } \varphi)(x := b)) \in \text{sem } C2 \{?b\}$ 
  by (metis ‹ $\varphi \in \text{sem } C1 \{(l, \sigma(x := a))\}$ › fst-eqD in-sem singleton-iff snd-eqD)
then show  $(\text{fst } \varphi, (\text{snd } \varphi)(x := b)) \in \text{sem } ?C S$ 
  by (simp add: if-sem)
qed
then show Q  $(\text{sem } ?C S)$ 
  using assms(3) by blast
qed
next
assume asm0: ?B

have set-of-traces C1  $\subseteq$  set-of-traces C2
proof (rule subsetPairI)
  fix  $\sigma \sigma'$  assume asm1:  $(\sigma, \sigma') \in \text{set-of-traces } C1$ 
  obtain l S where  $S :: ((\text{'lvar} \Rightarrow \text{'lval}) \times (\text{'pvar} \Rightarrow \text{'pval})) \text{ set} = \{(l, \sigma)\}$ 

  by simp

  let ?a =  $(l, \sigma(x := a))$ 
  let ?b =  $(l, \sigma(x := b))$ 

  let ?C = If (Seq (Assign  $(x :: \text{'pvar}) (\lambda-. a)$ ) C1) (Seq (Assign  $x (\lambda-. b)$ ) C2)
  have Q  $(\text{sem } ?C S)$ 
  proof (rule hyper-hoare-tripleE)
    show P S
      by (simp add: ‹ $S = \{(l, \sigma)\}$ › assms(2))
    show ?B using asm0 by simp
  qed
  moreover have  $(l, \sigma'(x := a)) \in \text{sem } ?C S$ 
  proof -
    have single-sem (Seq (Assign  $x (\lambda-. a)$ ) C1)  $\sigma (\sigma'(x := a))$ 
    by (meson SemAssign SemSeq asm1 assms(4) in-set-of-traces not-free-in-sem-equiv)
    then show ?thesis
      by (simp add: SemIf1 ‹ $S = \{(l, \sigma)\}$ › in-sem)
  qed
  then have  $(l, \sigma'(x := b)) \in \text{sem } ?C S$ 
    using assms(3) calculation by fastforce

```

moreover have $(l, \sigma'(x := b)) \in \text{sem } (\text{Seq } (\text{Assign } x (\lambda-. b)) C2) S$
proof (*rule ccontr*)
assume $\neg (l, \sigma'(x := b)) \in \text{sem } (\text{Seq } (\text{Assign } x (\lambda-. b)) C2) S$
then have $(l, \sigma'(x := b)) \in \text{sem } (\text{Seq } (\text{Assign } x (\lambda-. a)) C1) S$
using *calculation(2) sem-if* **by** *auto*
then have $(l, \sigma'(x := b)) \in \text{sem } C1 \{?a\}$
by (*simp add: $\langle S = \{(l, \sigma)\} \rangle$ sem-assign-single sem-seq*)
then show *False*
using *assms(1) assms(4) fun-upd-same in-sem not-free-in-sem-same-val*[of
 $x C1 \sigma(x := a) \sigma'(x := b)$] *singletonD snd-conv*
by *metis*
qed
then have *single-sem* $(\text{Seq } (\text{Assign } x (\lambda-. b)) C2) \sigma (\sigma'(x := b))$
by (*simp add: $\langle S = \{(l, \sigma)\} \rangle$ in-sem*)
then have *single-sem* $C2 (\sigma(x := b)) (\sigma'(x := b))$
by *blast*
then have $(\sigma(x := b), \sigma'(x := b)) \in \text{set-of-traces } C2$
by (*simp add: set-of-traces-def*)
then have $((\sigma(x := b))(x := \sigma x), (\sigma'(x := b))(x := \sigma x)) \in \text{set-of-traces } C2$
by (*meson assms(5) not-free-var-stmtE-1*)
then show $(\sigma, \sigma') \in \text{set-of-traces } C2$
by (*metis asm1 assms(4) fun-upd-triv fun-upd-upd in-set-of-traces not-free-in-sem-same-val*)
qed
then show $?A$
by (*simp add: refinement-def*)
qed

Necessary Preconditions definition NC where

$NC P C Q \longleftrightarrow (\forall \sigma \sigma' l. (l, \sigma') \in Q \wedge (\langle C, \sigma \rangle \rightarrow \sigma') \longrightarrow (l, \sigma) \in P)$

lemma NC-I:

assumes $\bigwedge \sigma \sigma' l. (l, \sigma') \in Q \wedge (\langle C, \sigma \rangle \rightarrow \sigma') \implies (l, \sigma) \in P$

shows $NC P C Q$

by (*simp add: NC-def assms*)

definition backwards-sem where

$\text{backwards-sem } C S' = \{ (l, \sigma) \mid l \sigma \sigma'. (l, \sigma') \in S' \wedge \langle C, \sigma \rangle \rightarrow \sigma' \}$

lemma equiv-def-NC:

$NC P C Q \longleftrightarrow \text{backwards-sem } C Q \subseteq P \text{ (is } ?A \longleftrightarrow ?B)$

proof

assume $?A$

then show $?B$

using *CollectD NC-def backwards-sem-def*[of $C Q$] *subsetI*[of *backwards-sem* $C Q P$] **by** *fastforce*

next

assume $?B$

then show $?A$

using *NC-def backwards-sem-def* **by** *fastforce*

qed

lemma *equiv-def-FU*:

$FU\ P\ C\ Q \longleftrightarrow P \subseteq \text{backwards-sem}\ C\ Q$ (is $?A \longleftrightarrow ?B$)

proof

show $?A \implies ?B$

using *FU-def*[of $P\ C\ Q$] *backwards-sem-def*[of $C\ Q$]

by *fastforce*

show $?B \implies ?A$

using *FU-def*[of $P\ C\ Q$] *backwards-sem-def*[of $C\ Q$] **by** *auto*

qed

lemma *encoding-NC-in-HL-1*:

$NC\ P\ C\ Q \longleftrightarrow \models \{ (\lambda S. S = -P) \} C \{ (\lambda S. S \cap Q = \{\}) \}$ (is $?A \longleftrightarrow ?B$)

proof

assume $?A$

show $?B$

proof (*rule hyper-hoare-tripleI*)

fix S **assume** *asm0*: $S = -P$

then show $\text{sem}\ C\ S \cap Q = \{\}$

by (*metis* (*no-types*, *lifting*) *ComplD NC-def* $\langle NC\ P\ C\ Q \rangle$ *disjoint-iff in-sem prod.collapse*)

qed

next

assume $?B$

show $?A$

proof (*rule NC-I*)

fix $\sigma\ \sigma'\ l$

assume *asm0*: $(l, \sigma') \in Q \wedge \langle C, \sigma \rangle \rightarrow \sigma'$

then have $(l, \sigma') \notin \text{sem}\ C\ (-P)$

using $\langle \models \{ (\lambda S. S = -P) \} C \{ (\lambda S. S \cap Q = \{\}) \} \rangle$ *hyper-hoare-tripleE* **by** *fastforce*

then have $(l, \sigma) \notin -P$

by (*meson* *asm0 single-step-then-in-sem*)

then show $(l, \sigma) \in P$

by *simp*

qed

qed

end

theory *Loops*

imports *Logic HOL.Wellfounded Expressivity*

begin

4 Rules for Loops

definition *lnot* **where**

$\text{lnot}\ b\ \sigma = (\neg b\ \sigma)$

definition *if-then-else where*

if-then-else b $C1$ $C2 = \text{If } (\text{Assume } b;; C1) (\text{Assume } (\text{lnot } b);; C2)$

definition *low-exp where*

low-exp e $S = (\forall \varphi \varphi'. \varphi \in S \wedge \varphi' \in S \longrightarrow (e (\text{snd } \varphi) = e (\text{snd } \varphi')))$

lemma *low-exp-lnot:*

low-exp b $S \longleftrightarrow \text{low-exp } (\text{lnot } b) S$

by (*simp add: lnot-def low-exp-def*)

definition *holds-forall where*

holds-forall b $S \longleftrightarrow (\forall \varphi \in S. b (\text{snd } \varphi))$

lemma *holds-forallI:*

assumes $\bigwedge \varphi. \varphi \in S \implies b (\text{snd } \varphi)$

shows *holds-forall* b S

using *assms holds-forall-def* **by** *blast*

lemma *low-exp-two-cases:*

assumes *low-exp* b S

shows *holds-forall* b $S \vee \text{holds-forall } (\text{lnot } b) S$

by (*metis assms holds-forall-def lnot-def low-exp-def*)

lemma *sem-assume-low-exp:*

assumes *holds-forall* b S

shows *sem* (*Assume* b) $S = S$

and *sem* (*Assume* (*lnot* b)) $S = \{\}$

using *assume-sem[of b S] assms holds-forall-def[of b S]* **apply** *fastforce*

using *assume-sem[of lnot b S] assms holds-forall-def[of b S] lnot-def[of b]*

by *fastforce*

lemma *sem-assume-low-exp-seq:*

assumes *holds-forall* b S

shows *sem* (*Assume* $b;; C$) $S = \text{sem } C S$

and *sem* (*Assume* (*lnot* b);; C) $S = \{\}$

apply (*simp add: assms sem-assume-low-exp(1) sem-seq*)

by (*metis assms empty-iff equalsOI in-sem sem-assume-low-exp(2) sem-seq*)

lemma *lnot-involution:*

lnot (*lnot* b) = b

proof (*rule ext*)

fix x **show** *lnot* (*lnot* b) $x = b x$

by (*simp add: lnot-def*)

qed

lemma *sem-if-then-else:*

shows *holds-forall* b $S \implies \text{sem } (\text{if-then-else } b C1 C2) S = \text{sem } C1 S$

and *holds-forall* (*lnot* b) $S \implies \text{sem } (\text{if-then-else } b C1 C2) S = \text{sem } C2 S$

apply (*simp add: if-then-else-def sem-assume-low-exp-seq(1) sem-assume-low-exp-seq(2) sem-if*)
by (*metis (no-types, opaque-lifting) if-then-else-def lnot-involution sem-assume-low-exp-seq(1) sem-assume-low-exp-seq(2) sem-if sup-bot-left*)

lemma *if-synchronized-aux*:

assumes $\models \{P\} C1 \{Q\}$
and $\models \{P\} C2 \{Q\}$
and *entails* P (*low-exp* b)
shows $\models \{P\}$ *if-then-else* b $C1$ $C2$ $\{Q\}$
proof (*rule hyper-hoare-tripleI*)
fix S **assume** *asm0*: P S
then have r : *low-exp* b S **using** *assms(3)* *entailsE*
by *metis*
show Q (*sem (if-then-else* b $C1$ $C2$) S)
proof (*cases holds-forall* b S)
case *True*
then show *?thesis*
by (*metis asm0 assms(1) hyper-hoare-triple-def sem-if-then-else(1)*)
next
case *False*
then show *?thesis*
by (*metis asm0 assms(2) hyper-hoare-tripleE low-exp-two-cases r sem-if-then-else(2)*)
qed
qed

theorem *if-synchronized*:

assumes $\models \{\text{conj } P \text{ (holds-forall } b)\} C1 \{Q\}$
and $\models \{\text{conj } P \text{ (holds-forall (lnot } b))\} C2 \{Q\}$
shows $\models \{\text{conj } P \text{ (low-exp } b)\}$ *if-then-else* b $C1$ $C2$ $\{Q\}$
proof (*rule hyper-hoare-tripleI*)
fix S **assume** *asm0*: $\text{conj } P$ (*low-exp* b) S
show Q (*sem (if-then-else* b $C1$ $C2$) S)
proof (*cases holds-forall* b S)
case *True*
then show *?thesis*
by (*metis asm0 assms(1) conj-def hyper-hoare-triple-def sem-if-then-else(1)*)
next
case *False*
then show *?thesis*
by (*metis asm0 assms(2) conj-def hyper-hoare-triple-def low-exp-two-cases sem-if-then-else(2)*)
qed
qed

definition *while-cond where*

while-cond b $C = \text{While (Assume } b;; C);; \text{Assume (lnot } b)$

lemma *while-synchronized-rec*:

```

assumes  $\bigwedge n. \models \{conj (I n) (holds-forall b)\} Assume\ b;;\ C \{conj (I (Suc\ n))$ 
 $(low-exp\ b)\}$ 
and  $conj (I\ 0) (low-exp\ b)\ S$ 
shows  $conj (I\ n) (low-exp\ b) (iterate-sem\ n (Assume\ b;;\ C)\ S) \vee holds-forall$ 
 $(lnot\ b) (iterate-sem\ n (Assume\ b;;\ C)\ S)$ 
using assms
proof (induct n)
case (Suc n)
then have  $r: conj (I\ n) (low-exp\ b) (iterate-sem\ n (Assume\ b;;\ C)\ S) \vee holds-forall$ 
 $(lnot\ b) (iterate-sem\ n (Assume\ b;;\ C)\ S)$ 
by blast
then show ?case
proof (cases conj (I n) (holds-forall b) (iterate-sem n (Assume b;; C) S))
case True
then show ?thesis
using Suc.prem(1) hyper-hoare-tripleE by fastforce
next
case False
then have  $holds-forall (lnot\ b) (iterate-sem\ n (Assume\ b;;\ C)\ S)$ 
by (metis conj-def low-exp-two-cases r)
then have  $iterate-sem (Suc\ n) (Assume\ b;;\ C)\ S = \{\}$ 
by (metis iterate-sem.simps(2) lnot-involution sem-assume-low-exp-seq(2))
then show ?thesis
by (simp add: holds-forall-def)
qed
qed (auto)

```

lemma *false-then-empty-later*:

```

assumes  $holds-forall (lnot\ b) (iterate-sem\ n (Assume\ b;;\ C)\ S)$ 
and  $m > n$ 
shows  $iterate-sem\ m (Assume\ b;;\ C)\ S = \{\}$ 
using assms
proof (induct m - n arbitrary: n m)
case (Suc x)
then show ?case
proof (cases x)
case 0
then show ?thesis
by (metis One-nat-def Suc.hyps(2) Suc.prem(1) Suc.prem(2) Suc-eq-plus1
 $iterate-sem.simps(2)$  le-add-diff-inverse linorder-not-le lnot-involution order.asym
 $sem-assume-low-exp-seq(2)$ )
next
case (Suc nat)
then have  $m - 1 > n$ 
using Suc.hyps(2) by auto
then have  $iterate-sem (m-1) (Assume\ b;;\ C)\ S = \{\}$ 
by (metis (no-types, lifting) Suc.hyps(1) Suc.hyps(2) Suc.prem(1) diff-Suc-1)

```

diff-commute)
then show *?thesis*
by (*metis Nat.lessE Suc.premis(1) Suc.premis(2) diff-Suc-1 iterate-sem.simps(2)*
sem-assume-low-exp-seq(2) sem-seq)
qed
qed (*simp*)

lemma *split-union-triple*:
 $(\bigcup m::nat. f m) = (\bigcup m \in \{m \mid m < n\}. f m) \cup f n \cup (\bigcup m \in \{m \mid m > n\}. f m)$ (*is* *?A = ?B*)
proof
show *?B \subseteq ?A*
by *blast*
show *?A \subseteq ?B*
proof
fix *x* **assume** *x \in ?A*
then obtain *m* **where** *x \in f m*
by *blast*
then have *m < n \vee m = n \vee m > n*
by *force*
then show *x \in ?B*
using $\langle x \in f m \rangle$ **by** *auto*
qed
qed

lemma *sem-union-swap*:
 $sem C (\bigcup x \in S. f x) = (\bigcup x \in S. sem C (f x))$ (*is* *?A = ?B*)
proof
show *?A \subseteq ?B*
proof
fix *y* **assume** *y \in ?A*
then obtain *x* **where** *x \in S y \in sem C (f x)*
using *UN-iff in-sem[of y C]* **by** *force*
then show *y \in ?B*
by *blast*
qed
show *?B \subseteq ?A*
by (*simp add: SUP-least SUP-upper sem-monotonic*)
qed

lemma *while-synchronized-case-1*:
assumes $\bigwedge m. m < n \implies holds\text{-forall } b (iterate\text{-sem } m (Assume\ b;; C) S)$
and $holds\text{-forall } (lnot\ b) (iterate\text{-sem } n (Assume\ b;; C) S)$
and $\bigwedge n. \models \{conj (I\ n) (holds\text{-forall } b)\} Assume\ b;; C \{conj (I (Suc\ n))$
 $(low\text{-exp } b)\}$
and $conj (I\ 0) (low\text{-exp } b) S$

shows $\text{sem } (\text{while-cond } b \ C) \ S = \text{iterate-sem } n \ (\text{Assume } b ;; \ C) \ S$
proof –
have $\bigwedge m. m > n \implies \text{iterate-sem } m \ (\text{Assume } b ;; \ C) \ S = \{\}$
using $\text{assms}(2)$ *false-then-empty-later* **by** *blast*
moreover have $\text{sem } (\text{While } (\text{Assume } b ;; \ C)) \ S =$
 $(\bigcup m \in \{m \mid m < n\}. \text{iterate-sem } m \ (\text{Assume } b ;; \ C) \ S) \cup \text{iterate-sem } n \ (\text{Assume } b ;; \ C) \ S \cup (\bigcup m \in \{m \mid m > n\}. \text{iterate-sem } m \ (\text{Assume } b ;; \ C) \ S)$
using *sem-while*[of *Assume* $b ;; \ C$ S] *split-union-triple* **by** *metis*
ultimately have $\text{sem } (\text{While } (\text{Assume } b ;; \ C)) \ S = (\bigcup m \in \{m \mid m < n\}. \text{iterate-sem } m \ (\text{Assume } b ;; \ C) \ S) \cup \text{iterate-sem } n \ (\text{Assume } b ;; \ C) \ S$
by *auto*
moreover have $\bigwedge m. m < n \implies \text{sem } (\text{Assume } (\text{lnot } b)) \ (\text{iterate-sem } m \ (\text{Assume } b ;; \ C) \ S) = \{\}$
using $\text{assms}(1)$ *sem-assume-low-exp(2)* **by** *blast*
then have $\text{sem } (\text{Assume } (\text{lnot } b)) \ (\bigcup m \in \{m \mid m < n\}. \text{iterate-sem } m \ (\text{Assume } b ;; \ C) \ S) = \{\}$
by (*simp add: sem-union-swap*)
then have $\text{sem } (\text{while-cond } b \ C) \ S = \text{sem } (\text{Assume } (\text{lnot } b)) \ (\text{iterate-sem } n \ (\text{Assume } b ;; \ C) \ S)$
by (*simp add: calculation sem-seq sem-union while-cond-def*)
then show *?thesis*
using $\text{assms}(2)$ *sem-assume-low-exp(1)* **by** *blast*
qed

lemma *while-synchronized-case-2:*

assumes $\bigwedge m. \text{holds-forall } b \ (\text{iterate-sem } m \ (\text{Assume } b ;; \ C) \ S)$
and $\bigwedge n. \models \{\text{conj } (I \ n) \ (\text{holds-forall } b)\} \ \text{Assume } b ;; \ C \ \{\text{conj } (I \ (\text{Suc } \ n)) \ (\text{low-exp } b)\}$
and $\text{conj } (I \ 0) \ (\text{low-exp } b) \ S$
shows $\text{sem } (\text{while-cond } b \ C) \ S = \{\}$
proof –
have $\text{sem } (\text{While } (\text{Assume } b ;; \ C)) \ S = (\bigcup n. \text{iterate-sem } n \ (\text{Assume } b ;; \ C) \ S)$
by (*simp add: sem-while*)
then have $\text{holds-forall } b \ (\text{sem } (\text{While } (\text{Assume } b ;; \ C)) \ S)$
by (*metis (no-types, lifting) UN-iff assms(1) holds-forall-def*)
then show *?thesis*
by (*simp add: sem-assume-low-exp(2) sem-seq while-cond-def*)
qed

definition *emp* **where**

$\text{emp } S \longleftrightarrow S = \{\}$

lemma *holds-forall-empty:*

$\text{holds-forall } b \ \{\}$

by (*simp add: holds-forall-def*)

definition *exists* **where**

$\text{exists } I \ S \longleftrightarrow (\exists n. I \ n \ S)$

```

theorem while-synchronized:
  assumes  $\bigwedge n. \models \{ \text{conj } (I\ n) \text{ (holds-forall } b) \} \ C \ \{ \text{conj } (I\ (\text{Suc } n)) \text{ (low-exp } b) \}$ 
  shows  $\models \{ \text{conj } (I\ 0) \text{ (low-exp } b) \} \ \text{while-cond } b \ C \ \{ \text{conj } (\text{disj } (\text{exists } I) \ \text{emp}) \}$ 
   $(\text{holds-forall } (\text{lnot } b))$ 
proof (rule hyper-hoare-tripleI)
  fix  $S$  assume  $\text{asm0}: \text{conj } (I\ 0) \text{ (low-exp } b) \ S$ 
  have  $\text{triple}: \bigwedge n. \models \{ \text{conj } (I\ n) \text{ (holds-forall } b) \} \ \text{Assume } b \ ;; \ C \ \{ \text{conj } (I\ (\text{Suc } n)) \}$ 
   $(\text{low-exp } b)$ 
proof (rule hyper-hoare-tripleI)
  fix  $n\ S$  assume  $\text{conj } (I\ n) \text{ (holds-forall } b) \ S$ 
  then have  $\text{sem } (\text{Assume } b) \ S = S$ 
  by (simp add: conj-def sem-assume-low-exp(1))
  then show  $\text{conj } (I\ (\text{Suc } n)) \text{ (low-exp } b) \ (\text{sem } (\text{Assume } b \ ;; \ C) \ S)$ 
  by (metis <conj (I n) (holds-forall b) S> assms hyper-hoare-tripleE sem-seq)
qed
show  $\text{conj } (\text{disj } (\text{exists } I) \ \text{emp}) \text{ (holds-forall } (\text{lnot } b)) \ (\text{sem } (\text{while-cond } b \ C) \ S)$ 
proof (cases  $\forall m. \text{holds-forall } b \text{ (iterate-sem } m \ (\text{Assume } b;; \ C) \ S)$ )
  case True
  then have  $\text{sem } (\text{while-cond } b \ C) \ S = \{ \}$ 
  using while-synchronized-case-2[of b C S I]
  by (metis (no-types, opaque-lifting) asm0 assms hyper-hoare-tripleI conj-def sem-assume-low-exp(1) seq-rule)
  then show ?thesis
  by (simp add: disj-def conj-def emp-def holds-forall-empty)
  next
  case False
  then have  $F: \neg (\forall m. \text{holds-forall } b \text{ (iterate-sem } m \ (\text{Assume } b;; \ C) \ S))$  by
simp
  have  $\exists n. (\forall m. m < n \longrightarrow \text{holds-forall } b \text{ (iterate-sem } m \ (\text{Assume } b;; \ C) \ S)) \wedge$ 
 $\text{holds-forall } (\text{lnot } b) \text{ (iterate-sem } n \ (\text{Assume } b;; \ C) \ S)$ 
  proof (cases  $\exists n. \neg \text{holds-forall } b \text{ (iterate-sem } n \ (\text{Assume } b;; \ C) \ S) \wedge \text{iterate-sem}$ 
 $n \ (\text{Assume } b;; \ C) \ S \neq \{ \}$ )
  case True
  then obtain  $n$  where  $\neg \text{holds-forall } b \text{ (iterate-sem } n \ (\text{Assume } b;; \ C) \ S) \wedge$ 
 $\text{iterate-sem } n \ (\text{Assume } b;; \ C) \ S \neq \{ \}$ 
  by blast
  then have  $\text{holds-forall } (\text{lnot } b) \text{ (iterate-sem } n \ (\text{Assume } b;; \ C) \ S)$ 
  by (metis asm0 conj-def low-exp-two-cases triple while-synchronized-rec)
  moreover have  $\bigwedge m. m < n \implies \text{holds-forall } b \text{ (iterate-sem } m \ (\text{Assume } b;;$ 
 $C) \ S)$ 
proof –
  fix  $m$  assume  $\text{asm1}: m < n$ 
  show  $\text{holds-forall } b \text{ (iterate-sem } m \ (\text{Assume } b;; \ C) \ S)$ 
proof (rule ccontr)
  assume  $\neg \text{holds-forall } b \text{ (iterate-sem } m \ (\text{Assume } b;; \ C) \ S)$ 
  then have  $\text{holds-forall } (\text{lnot } b) \text{ (iterate-sem } m \ (\text{Assume } b;; \ C) \ S)$ 
  by (metis asm0 conj-def low-exp-two-cases triple while-synchronized-rec)
  then have  $\text{iterate-sem } n \ (\text{Assume } b;; \ C) \ S = \{ \}$ 
  using asm1 false-then-empty-later by blast

```

```

      then show False
        using  $\langle \neg \text{holds-forall } b \text{ (iterate-sem } n \text{ (Assume } b ;; C) S) \wedge \text{iterate-sem}$ 
 $n \text{ (Assume } b ;; C) S \neq \{\} \rangle$  by fastforce
      qed
    qed
    ultimately show ?thesis
      by blast
  next
  case False
  then have  $\bigwedge n. \text{holds-forall } b \text{ (iterate-sem } n \text{ (Assume } b ;; C) S)$ 
    using holds-forall-empty by fastforce
  then show ?thesis using F by blast
  qed
  then obtain n where  $\bigwedge m. m < n \implies \text{holds-forall } b \text{ (iterate-sem } m \text{ (Assume}$ 
 $b ;; C) S)$ 
    and holds-forall (lnot b) (iterate-sem n (Assume b ;; C) S)
    by blast
  then have sem (while-cond b C) S = iterate-sem n (Assume b ;; C) S
    using triple
  proof (rule while-synchronized-case-1)
  qed (simp-all add: asm0)
  moreover have I n (iterate-sem n (Assume b ;; C) S)
  proof (cases n)
  case 0
  then show ?thesis
    by (metis asm0 iterate-sem.simps(1) conj-def)
  next
  case (Suc k)
  then have conj (I k) (low-exp b) (iterate-sem k (Assume b ;; C) S)  $\vee$ 
 $\text{holds-forall (lnot } b \text{) (iterate-sem } k \text{ (Assume } b ;; C) S)$ 
    using while-synchronized-rec[of I b C S k] asm0 triple by blast
  then show ?thesis
  proof (cases conj (I k) (low-exp b) (iterate-sem k (Assume b ;; C) S))
  case True
  then show ?thesis
    using conj-def[of - holds-forall b] conj-def[of - low-exp b] Suc
 $\langle \bigwedge m. m < n \implies \text{holds-forall } b \text{ (iterate-sem } m \text{ (Assume } b ;; C) S) \rangle$  assms
 $\text{hyper-hoare-triple-def[of ] iterate-sem.simps(2) lessI sem-assume-low-exp(1)[of}$ 
 $b \text{ iterate-sem } k \text{ (Assume } b ;; C) S]$ 
    sem-seq[of Assume b C] by metis
  next
  case False
  then show ?thesis
    by (metis F Suc  $\langle \bigwedge m. m < n \implies \text{holds-forall } b \text{ (iterate-sem } m \text{ (Assume } b ;;$ 
 $C) S) \rangle \langle \text{conj (I } k \text{) (low-exp } b \text{) (iterate-sem } k \text{ (Assume } b ;; C) S) \vee \text{holds-forall (lnot}$ 
 $b \text{) (iterate-sem } k \text{ (Assume } b ;; C) S) \rangle$  empty-iff false-then-empty-later holds-forall-def
 $\text{not-less-eq}$ )
  qed
  qed

```

ultimately show *?thesis*
by (*metis disj-def Loops.exists-def ‹holds-forall (lnot b) (iterate-sem n (Assume b ;; C) S)› conj-def*)
qed
qed

lemma *WhileSync-simpler*:
assumes $\models \{conj\ I\ (holds\text{-}forall\ b)\} C \{conj\ I\ (low\text{-}exp\ b)\}$
shows $\models \{conj\ I\ (low\text{-}exp\ b)\} while\text{-}cond\ b\ C \{conj\ (disj\ I\ emp)\ (holds\text{-}forall\ (lnot\ b))\}$
using *assms while-synchronized[of $\lambda n. I$]*
by (*simp add: disj-def Loops.exists-def conj-def hyper-hoare-triple-def*)

definition *if-then where*
if-then $b\ C = If\ (Assume\ b;;\ C)\ (Assume\ (lnot\ b))$

definition *filter-exp where*
filter-exp $b\ S = Set.filter\ (b\ o\ snd)\ S$

lemma *filter-exp-union*:
filter-exp $b\ (S1\ \cup\ S2) = filter\text{-}exp\ b\ S1\ \cup\ filter\text{-}exp\ b\ S2$ (**is** $?A = ?B$)
by (*auto simp add: filter-exp-def*)

lemma *filter-exp-union-general*:
filter-exp $b\ (\bigcup x. f\ x) = (\bigcup x. filter\text{-}exp\ b\ (f\ x))$ (**is** $?A = ?B$)
by (*auto simp add: filter-exp-def*)

lemma *filter-exp-contradict*:
filter-exp $b\ (filter\text{-}exp\ (lnot\ b)\ S) = \{\}$
by (*auto simp add: filter-exp-def lnot-def*)

lemma *filter-exp-same*:
filter-exp $b\ (filter\text{-}exp\ b\ S) = filter\text{-}exp\ b\ S$ (**is** $?A = ?B$)
by (*auto simp add: filter-exp-def*)

lemma *if-then-sem*:
sem (*if-then* $b\ C$) $S = sem\ C\ (filter\text{-}exp\ b\ S) \cup filter\text{-}exp\ (lnot\ b)\ S$
by (*simp add: assume-sem filter-exp-def if-then-def sem-if sem-seq*)

fun *union-up-to-n where*
union-up-to-n $C\ S\ 0 = iterate\text{-}sem\ 0\ C\ S$
| *union-up-to-n* $C\ S\ (Suc\ n) = iterate\text{-}sem\ (Suc\ n)\ C\ S \cup union\text{-}up\text{-}to\text{-}n\ C\ S\ n$

lemma *union-up-to-increasing*:
assumes $m \leq n$
shows *union-up-to-n* $C\ S\ m \subseteq union\text{-}up\text{-}to\text{-}n\ C\ S\ n$
using *assms*
proof (*induct n - m arbitrary: m n*)
case (*Suc x*)

then show *?case*
 by (*simp add: lift-Suc-mono-le*)
qed (*simp*)

lemma *union-union-up-to-n-equiv-aux:*
 $union-up-to-n\ C\ S\ n \subseteq (\bigcup m. iterate-sem\ m\ C\ S)$
proof (*induct n*)
case *0*
then show *?case*
 by (*metis UN-upper iso-tuple-UNIV-I union-up-to-n.simps(1)*)
next
case (*Suc n*)
show *?case*
proof
 fix *x* **assume** $x \in union-up-to-n\ C\ S\ (Suc\ n)$
then have $x \in iterate-sem\ (Suc\ n)\ C\ S \vee x \in union-up-to-n\ C\ S\ n$
 by *simp*
then show $x \in (\bigcup m. iterate-sem\ m\ C\ S)$
 using *Suc* by *blast*
qed
qed

lemma *union-union-up-to-n-equiv:*
 $(\bigcup n. union-up-to-n\ C\ S\ n) = (\bigcup n. iterate-sem\ n\ C\ S)$ (**is** *?A = ?B*)
proof
show *?B* \subseteq *?A*
 by (*metis (no-types, lifting) SUP-subset-mono UnCI subsetI union-up-to-n.elims*)
show *?A* \subseteq *?B*
 by (*simp add: SUP-le-iff union-union-up-to-n-equiv-aux*)
qed

lemma *filter-exp-union-itself:*
 $filter-exp\ b\ S \cup S = S$
 by (*auto simp add: filter-exp-def*)

lemma *iterate-sem-equiv:*
 $iterate-sem\ m\ (if-then\ b\ C)\ S$
 $= filter-exp\ (lnot\ b)\ (union-up-to-n\ (Assume\ b\ ;;\ C)\ S\ m) \cup iterate-sem\ m\ (Assume\ b\ ;;\ C)\ S$
proof (*induct m*)
case *0*
have $union-up-to-n\ (Assume\ b\ ;;\ C)\ S\ 0 = S$
 by *auto*
then show $iterate-sem\ 0\ (if-then\ b\ C)\ S = filter-exp\ (lnot\ b)\ (union-up-to-n\ (Assume\ b\ ;;\ C)\ S\ 0) \cup iterate-sem\ 0\ (Assume\ b\ ;;\ C)\ S$
 by (*auto simp add: filter-exp-def*)
next
case (*Suc m*)

let $?S = \text{iterate-sem } m \text{ (if-then } b \text{ } C) \text{ } S$
let $?SU = \text{union-up-to-}n \text{ (Assume } b \text{ ;; } C) \text{ } S \text{ } m$
let $?SN = \text{iterate-sem } m \text{ (Assume } b \text{ ;; } C) \text{ } S$
have $\text{iterate-sem } (Suc \text{ } m) \text{ (if-then } b \text{ } C) \text{ } S = \text{sem } C \text{ (filter-exp } b \text{ } ?S) \cup \text{filter-exp}$
 $(\text{lnot } b) \text{ } ?S$
by (*simp add: if-then-sem*)
also have $\dots = \text{sem } C \text{ (filter-exp } b \text{ (filter-exp } (\text{lnot } b) \text{ } ?SU)) \cup \text{sem } C \text{ (filter-exp}$
 $b \text{ } ?SN)$
 $\cup \text{filter-exp } (\text{lnot } b) \text{ (filter-exp } (\text{lnot } b) \text{ } ?SU) \cup \text{filter-exp } (\text{lnot } b) \text{ } ?SN$
by (*simp add: Suc filter-exp-union sem-union sup-assoc*)
also have $\dots = \text{sem } C \text{ (filter-exp } b \text{ } ?SN) \cup \text{filter-exp } (\text{lnot } b) \text{ } ?SU \cup \text{filter-exp}$
 $(\text{lnot } b) \text{ } ?SN$
by (*metis Un-empty-left filter-exp-contradict filter-exp-same sem-union*)
moreover have $\text{iterate-sem } (Suc \text{ } m) \text{ (Assume } b \text{ ;; } C) \text{ } S = \text{sem } C \text{ (filter-exp } b$
 $?SN)$
by (*simp add: assume-sem filter-exp-def sem-seq*)
moreover have $\text{union-up-to-}n \text{ (Assume } b \text{ ;; } C) \text{ } S \text{ (Suc } m) = \text{sem } C \text{ (filter-exp}$
 $b \text{ } ?SN) \cup ?SU$
using *calculation(3) by force*
moreover have $\text{filter-exp } (\text{lnot } b) \text{ (union-up-to-}n \text{ (Assume } b \text{ ;; } C) \text{ } S \text{ (Suc } m)) \cup$
 $\text{iterate-sem } (Suc \text{ } m) \text{ (Assume } b \text{ ;; } C) \text{ } S$
 $= \text{filter-exp } (\text{lnot } b) \text{ (sem } C \text{ (filter-exp } b \text{ } ?SN) \cup ?SU) \cup \text{sem } C \text{ (filter-exp } b$
 $?SN)$
using *calculation(3) by force*
then have $\dots = \text{filter-exp } (\text{lnot } b) \text{ } ?SU \cup \text{sem } C \text{ (filter-exp } b \text{ } ?SN)$
using *filter-exp-union-itself[of lnot b] filter-exp-union[of lnot b] Un-commute*
sup-assoc by blast
moreover have $?SN \subseteq ?SU$
by (*metis UnCI subsetI union-up-to-n.elims*)
ultimately have $\text{filter-exp } (\text{lnot } b) \text{ } ?SU \cup \text{sem } C \text{ (filter-exp } b \text{ } ?SN)$
 $= \text{sem } C \text{ (filter-exp } b \text{ } ?SN) \cup \text{filter-exp } (\text{lnot } b) \text{ } ?SU \cup \text{filter-exp } (\text{lnot } b) \text{ } ?SN$
using *filter-exp-union[of lnot b ?SU ?SN]*
using *Un-commute[of filter-exp (lnot b) ?SU sem C (filter-exp b ?SN)]*
sup.orderE sup-assoc[of sem C (filter-exp b ?SN)] by metis
then show *?case*
using $\langle \text{filter-exp } (\text{lnot } b) \text{ (sem } C \text{ (filter-exp } b \text{ (iterate-sem } m \text{ (Assume } b \text{ ;; } C)$
 $S)) \cup \text{union-up-to-}n \text{ (Assume } b \text{ ;; } C) \text{ } S \text{ } m) \cup \text{sem } C \text{ (filter-exp } b \text{ (iterate-sem } m$
 $(\text{Assume } b \text{ ;; } C) \text{ } S)) = \text{filter-exp } (\text{lnot } b) \text{ (union-up-to-}n \text{ (Assume } b \text{ ;; } C) \text{ } S \text{ } m) \cup \text{sem}$
 $C \text{ (filter-exp } b \text{ (iterate-sem } m \text{ (Assume } b \text{ ;; } C) \text{ } S)) \rangle \langle \text{iterate-sem } (Suc \text{ } m) \text{ (Assume}$
 $b \text{ ;; } C) \text{ } S = \text{sem } C \text{ (filter-exp } b \text{ (iterate-sem } m \text{ (Assume } b \text{ ;; } C) \text{ } S)) \rangle \langle \text{iterate-sem}$
 $(Suc \text{ } m) \text{ (if-then } b \text{ } C) \text{ } S = \text{sem } C \text{ (filter-exp } b \text{ (iterate-sem } m \text{ (if-then } b \text{ } C) \text{ } S)) \cup$
 $\text{filter-exp } (\text{lnot } b) \text{ (iterate-sem } m \text{ (if-then } b \text{ } C) \text{ } S) \rangle \langle \text{sem } C \text{ (filter-exp } b \text{ (filter-exp}$
 $(\text{lnot } b) \text{ (union-up-to-}n \text{ (Assume } b \text{ ;; } C) \text{ } S \text{ } m)) \cup \text{sem } C \text{ (filter-exp } b \text{ (iterate-sem } m$
 $(\text{Assume } b \text{ ;; } C) \text{ } S)) \cup \text{filter-exp } (\text{lnot } b) \text{ (filter-exp } (\text{lnot } b) \text{ (union-up-to-}n \text{ (Assume}$
 $b \text{ ;; } C) \text{ } S \text{ } m)) \cup \text{filter-exp } (\text{lnot } b) \text{ (iterate-sem } m \text{ (Assume } b \text{ ;; } C) \text{ } S) = \text{sem } C$
 $(\text{filter-exp } b \text{ (iterate-sem } m \text{ (Assume } b \text{ ;; } C) \text{ } S)) \cup \text{filter-exp } (\text{lnot } b) \text{ (union-up-to-}n$
 $(\text{Assume } b \text{ ;; } C) \text{ } S \text{ } m) \cup \text{filter-exp } (\text{lnot } b) \text{ (iterate-sem } m \text{ (Assume } b \text{ ;; } C) \text{ } S) \rangle \langle \text{sem}$
 $C \text{ (filter-exp } b \text{ (iterate-sem } m \text{ (if-then } b \text{ } C) \text{ } S)) \cup \text{filter-exp } (\text{lnot } b) \text{ (iterate-sem } m$

$(\text{if-then } b \ C) \ S) = \text{sem } C \ (\text{filter-exp } b \ (\text{filter-exp } (\text{lnot } b) \ (\text{union-up-to-}n \ (\text{Assume } b \ ; ; \ C) \ S \ m))) \cup \text{sem } C \ (\text{filter-exp } b \ (\text{iterate-sem } m \ (\text{Assume } b \ ; ; \ C) \ S)) \cup \text{filter-exp } (\text{lnot } b) \ (\text{filter-exp } (\text{lnot } b) \ (\text{union-up-to-}n \ (\text{Assume } b \ ; ; \ C) \ S \ m)) \cup \text{filter-exp } (\text{lnot } b) \ (\text{iterate-sem } m \ (\text{Assume } b \ ; ; \ C) \ S)$, **by auto**
qed

lemma *sem-while-with-if*:

$\text{sem } (\text{while-cond } b \ C) \ S = \text{filter-exp } (\text{lnot } b) \ (\bigcup n. \text{iterate-sem } n \ (\text{if-then } b \ C) \ S)$

proof –

have $(\bigcup n. \text{iterate-sem } n \ (\text{if-then } b \ C) \ S)$

$= (\bigcup n. \text{filter-exp } (\text{lnot } b) \ (\text{union-up-to-}n \ (\text{Assume } b ; ; \ C) \ S \ n) \cup \text{iterate-sem } n \ (\text{Assume } b ; ; \ C) \ S)$

by (*simp add: iterate-sem-equiv*)

also have $\dots = \text{filter-exp } (\text{lnot } b) \ (\bigcup n. \text{union-up-to-}n \ (\text{Assume } b ; ; \ C) \ S \ n) \cup (\bigcup n. \text{iterate-sem } n \ (\text{Assume } b ; ; \ C) \ S)$

by (*simp add: complete-lattice-class.SUP-sup-distrib filter-exp-union-general*)

also have $\dots = \text{filter-exp } (\text{lnot } b) \ (\bigcup n. \text{iterate-sem } n \ (\text{Assume } b ; ; \ C) \ S) \cup (\bigcup n. \text{iterate-sem } n \ (\text{Assume } b ; ; \ C) \ S)$

by (*simp add: union-union-up-to-n-equiv*)

also have $\dots = (\bigcup n. \text{iterate-sem } n \ (\text{Assume } b ; ; \ C) \ S)$

by (*meson filter-exp-union-itself*)

moreover have $\text{sem } (\text{while-cond } b \ C) \ S = \text{filter-exp } (\text{lnot } b) \ (\bigcup n. \text{iterate-sem } n \ (\text{Assume } b ; ; \ C) \ S)$

by (*simp add: assume-sem filter-exp-def sem-seq sem-while while-cond-def*)

ultimately show *?thesis*

by *presburger*

qed

lemma *iterate-sem-assume-increasing*:

$\text{filter-exp } (\text{lnot } b) \ (\text{iterate-sem } n \ (\text{if-then } b \ C) \ S) \subseteq \text{filter-exp } (\text{lnot } b) \ (\text{iterate-sem } (\text{Suc } n) \ (\text{if-then } b \ C) \ S)$

by (*auto simp add: filter-exp-def lnot-def if-then-sem*)

lemma *iterate-sem-assume-increasing-union-up-to*:

$\text{filter-exp } (\text{lnot } b) \ (\text{iterate-sem } n \ (\text{if-then } b \ C) \ S) = \text{filter-exp } (\text{lnot } b) \ (\text{union-up-to-}n \ (\text{if-then } b \ C) \ S \ n)$

proof (*induct n*)

case (*Suc n*)

then show *?case*

by (*metis filter-exp-union iterate-sem-assume-increasing sup.orderE union-up-to-n.simps(2)*)

qed (*simp*)

definition *ascending* :: $(\text{nat} \Rightarrow 'b \ \text{set}) \Rightarrow \text{bool}$ **where**

$\text{ascending } S \iff (\forall n \ m. \ n \leq m \longrightarrow S \ n \subseteq S \ m)$

lemma *ascendingI-direct*:

assumes $\bigwedge n \ m. \ n \leq m \implies S \ n \subseteq S \ m$

shows *ascending S*
by (*simp add: ascending-def assms*)

lemma *ascendingI*:
assumes $\bigwedge n. S\ n \subseteq S\ (Suc\ n)$
shows *ascending S*
proof (*rule ascendingI-direct*)
fix $n\ m :: nat$ **assume** *asm0*: $n \leq m$
moreover have $n \leq m \implies S\ n \subseteq S\ m$
proof (*induct m - n arbitrary: m n*)
case (*Suc x*)
then show *?case*
using *assms lift-Suc-mono-le* **by** *blast*
qed (*simp*)
ultimately show $S\ n \subseteq S\ m$
by *blast*
qed

definition *upwards-closed where*
upwards-closed P P-inf $\longleftrightarrow (\forall S. \textit{ascending}\ S \wedge (\forall n. P\ n\ (S\ n)) \longrightarrow P\text{-inf}\ (\bigcup n. S\ n))$

lemma *upwards-closedI*:
assumes $\bigwedge S. \textit{ascending}\ S \implies (\forall n. P\ n\ (S\ n)) \implies P\text{-inf}\ (\bigcup n. S\ n)$
shows *upwards-closed P P-inf*
using *assms upwards-closed-def* **by** *blast*

lemma *upwards-closedE*:
assumes *upwards-closed P P-inf*
and *ascending S*
and $\bigwedge n. P\ n\ (S\ n)$
shows $P\text{-inf}\ (\bigcup n. S\ n)$
using *assms(1) assms(2) assms(3) upwards-closed-def* **by** *blast*

lemma *ascending-iterate-filter*:
ascending $(\lambda n. \textit{filter-exp}\ (\textit{lnot}\ b)\ (\textit{union-up-to-n}\ (\textit{if-then}\ b\ C)\ S\ n))$
by (*metis ascendingI iterate-sem-assume-increasing iterate-sem-assume-increasing-union-up-to*)

theorem *while-general*:
assumes $\bigwedge n. \models \{P\ n\} \textit{if-then}\ b\ C\ \{P\ (Suc\ n)\}$
and $\bigwedge n. \models \{P\ n\} \textit{Assume}\ (\textit{lnot}\ b)\ \{Q\ n\}$
and *upwards-closed Q Q-inf*
shows $\models \{P\ 0\} \textit{while-cond}\ b\ C\ \{\textit{conj}\ Q\text{-inf}\ (\textit{holds-forall}\ (\textit{lnot}\ b))\}$
proof (*rule hyper-hoare-tripleI*)
fix S **assume** *asm0*: $P\ 0\ S$
then have $\bigwedge n. P\ n\ (\textit{iterate-sem}\ n\ (\textit{if-then}\ b\ C)\ S)$

by (*meson* *assms*(1) *indexed-invariant-then-power*)
then have $\bigwedge n. Q\ n$ (*filter-exp* (*lnot* *b*) (*union-up-to-n* (*if-then* *b* *C*) *S* *n*))
 by (*metis* *assms*(2) *assume-sem* *filter-exp-def* *hyper-hoare-triple-def* *iterate-sem-assume-increasing-union-up-*
moreover have *ascending* ($\lambda n. \text{filter-exp } (\text{lnot } b) (\text{union-up-to-n } (\text{if-then } b\ C)\ S\ n)$)
 by (*simp* *add: ascending-iterate-filter*)
ultimately have *Q-inf* (*sem* (*while-cond* *b* *C*) *S*)
 by (*metis* (*no-types*, *lifting*) *SUP-cong* *assms*(3) *filter-exp-union-general* *iterate-sem-assume-increasing-union-up-to* *sem-while-with-if* *upwards-closed-def*)
then show *Logic.conj* *Q-inf* (*holds-forall* (*lnot* *b*)) (*sem* (*while-cond* *b* *C*) *S*)
 by (*simp* *add: conj-def* *filter-exp-def* *holds-forall-def* *sem-while-with-if*)
qed

definition *while-loop-assertion-n* **where**
while-loop-assertion-n *C* *S* *0* *n* *S* $\longleftrightarrow (S = \text{union-up-to-n } C\ S\ 0\ n)$

definition *while-loop-assertion-inf* **where**
while-loop-assertion-inf *C* *S* *0* *S* $\longleftrightarrow (S = \bigcup n. \text{union-up-to-n } C\ S\ 0\ n)$

lemma *while-loop-assertion-upwards-closed*:
upwards-closed (*while-loop-assertion-n* *C* *S* *0*) (*while-loop-assertion-inf* *C* *S* *0*)
proof (*rule* *upwards-closedI*)
fix *S* **assume** *asm0*: *ascending* *S* $\forall n. \text{while-loop-assertion-n } C\ S\ 0\ n\ (S\ n)$
then have $\bigwedge n. S\ n = \text{union-up-to-n } C\ S\ 0\ n$
 by (*simp* *add: while-loop-assertion-n-def*)
then have $\bigcup (\text{range } S) = \bigcup n. \text{union-up-to-n } C\ S\ 0\ n$
 by *auto*
then show *while-loop-assertion-inf* *C* *S* *0* ($\bigcup (\text{range } S)$)
 by (*simp* *add: while-loop-assertion-inf-def*)
qed

definition *converges-sets* **where**
converges-sets *S* $\longleftrightarrow (\forall x. \exists n. (\forall m. m \geq n \longrightarrow (x \in S\ m)) \vee (\forall m. m \geq n \longrightarrow (x \notin S\ m)))$

lemma *converges-setsI*:
assumes $\bigwedge x. \exists n. (\forall m. m \geq n \longrightarrow (x \in S\ m)) \vee (\forall m. m \geq n \longrightarrow (x \notin S\ m))$
shows *converges-sets* *S*
 by (*simp* *add: assms* *converges-sets-def*)

lemma *ascending-converges*:
assumes *ascending* *S*
shows *converges-sets* *S*
proof (*rule* *converges-setsI*)
fix *x*
show $\exists n. (\forall m \geq n. x \in S\ m) \vee (\forall m \geq n. x \notin S\ m)$
proof (*cases* $x \in \bigcup n. S\ n$)

```

    case True
    then show ?thesis
      by (meson ascending-def assms in-mono)
    qed (blast)
  qed

```

definition *descending* :: (nat \Rightarrow 'b set) \Rightarrow bool **where**
descending $S \longleftrightarrow (\forall n\ m. n \geq m \longrightarrow S\ n \subseteq S\ m)$

lemma *descending-converges*:
assumes *descending* S
shows *converges-sets* S
proof (rule *converges-setsI*)
 fix x
 show $\exists n. (\forall m \geq n. x \in S\ m) \vee (\forall m \geq n. x \notin S\ m)$
proof (cases $x \in (\bigcap n. S\ n)$)
 case False
 then show ?thesis
 by (meson assms descending-def in-mono)
 qed (blast)
qed

definition *limit-sets* **where**
limit-sets $S = \{x \mid x. \exists n. \forall m. m \geq n \longrightarrow (x \in S\ m)\}$

lemma *in-limit-sets*:
 $x \in \text{limit-sets } S \longleftrightarrow (\exists n. \forall m. m \geq n \longrightarrow (x \in S\ m))$
by (simp add: *limit-sets-def*)

lemma *ascending-limits-union*:
assumes *ascending* S
shows *limit-sets* $S = (\bigcup n. S\ n)$ (is ?A = ?B)
proof
 show ?A \subseteq ?B **using** *limit-sets-def*[of S] **by** auto
 show ?B \subseteq ?A
proof
 fix x **assume** $x \in ?B$
 then obtain n **where** $x \in S\ n$
 by blast
 then have $\forall m. m \geq n \longrightarrow (x \in S\ m)$
 by (meson ascending-def assms subsetD)
 then show $x \in ?A$
 using *limit-sets-def*[of S] **by** auto
 qed
qed

lemma *descending-limits-union*:

assumes *descending S*
shows *limit-sets S = ($\bigcap n. S n$) (is ?A = ?B)*
proof
show $?B \subseteq ?A$ **using** *limit-sets-def[of S]* **by** *fastforce*
show $?A \subseteq ?B$
proof
fix *x* **assume** $x \in ?A$
then obtain *n* **where** $\forall m. m \geq n \longrightarrow (x \in S m)$
using *limit-sets-def[of S]* **by** *blast*
then have $\forall m. m < n \longrightarrow (x \in S m)$
by (*meson assms descending-def lessI less-imp-le-nat subsetD*)
then show $x \in ?B$
by (*meson INT-I $\langle \forall m \geq n. x \in S m \rangle$ linorder-not-le*)
qed
qed

definition *t-closed where*
 $t\text{-closed } P \text{ } P\text{-inf} \longleftrightarrow (\forall S. \text{converges-sets } S \wedge (\forall n. P n (S n)) \longrightarrow P\text{-inf } (\text{limit-sets } S))$

lemma *t-closed-implies-u-closed:*
assumes *t-closed P P-inf*
shows *upwards-closed P P-inf*
proof (*rule upwards-closedI*)
fix *S* **assume** *ascending S $\forall n. P n (S n)$*
then have *converges-sets S*
using *ascending-converges* **by** *blast*
then show $P\text{-inf } (\bigcup (\text{range } S))$
by (*metis $\langle \forall n. P n (S n) \rangle \langle \text{ascending } S \rangle$ ascending-limits-union assms t-closed-def*)
qed

definition *downwards-closed where*
 $downwards\text{-closed } P\text{-inf} \longleftrightarrow (\forall S S'. S \subseteq S' \wedge P\text{-inf } S' \longrightarrow P\text{-inf } S)$

definition *d-closed where*
 $d\text{-closed } P \text{ } P\text{-inf} \longleftrightarrow t\text{-closed } P \text{ } P\text{-inf} \wedge downwards\text{-closed } P\text{-inf}$

lemma *converges-to-merged:*
assumes $\bigwedge x. x \in S\text{-inf} \implies \exists n. \forall m. m \geq n \longrightarrow (x \in S (m::nat))$
and $\bigwedge x. x \notin S\text{-inf} \implies \exists n. \forall m. m \geq n \longrightarrow (x \notin S m)$
shows *converges-sets S \wedge limit-sets S = S-inf*
proof (*rule conjI*)
show *converges-sets S* **using** *converges-setsI assms* **by** *metis*
show *limit-sets S = S-inf* **(is ?A = ?B)**
proof

```

show ?B ⊆ ?A
  by (simp add: assms(1) limit-sets-def subsetI)
show ?A ⊆ ?B
proof
  fix x assume x ∈ ?A
  then obtain n where n-def: ∀ m. m ≥ n → (x ∈ S m)
    using in-limit-sets by metis
  show x ∈ ?B
  proof (rule ccontr)
    assume x ∉ S-inf
    then obtain n' where ∀ m. m ≥ n' → (x ∉ S m)
      using assms(2) by presburger
    then have x ∈ S (max n n') ∧ x ∉ S (max n n')
      using n-def by fastforce
    then show False by blast
  qed
qed
qed
qed

```

```

lemma ascending-union-up:
  ascending (λn. union-up-to-n C S n)
  by (simp add: ascending-def union-up-to-increasing)

```

```

lemma converges-union:
  converges-sets (λn. union-up-to-n C S n) ∧ limit-sets (λn. union-up-to-n C S n)
  = (⋃ n. union-up-to-n C S n)
proof (rule converges-to-merged)
  fix x
  show x ∈ ⋃ (range (union-up-to-n C S)) ⇒ ∃ n. ∀ m ≥ n. x ∈ union-up-to-n C
  S m
  by (meson UN-iff subset-eq union-up-to-increasing)
  show x ∉ ⋃ (range (union-up-to-n C S)) ⇒ ∃ n. ∀ m ≥ n. x ∉ union-up-to-n C
  S m
  by blast
qed

```

```

theorem while-d:
  assumes ∧ n. ⊨ {P n} if-then b C {P (Suc n)}
  and upwards-closed P P-inf
  and ∧ n. downwards-closed (P n) — Satisfied by hyper-assertions that do not
  existentially quantify over states
  shows ⊨ {P 0} while-cond b C {conj P-inf (holds-forall (lnot b))}
  using assms(1)
proof (rule while-general)
  show upwards-closed P P-inf
  using assms(2) by blast

```

```

fix  $n$  show  $\models \{P\ n\}$  Assume ( $\text{lnot } b$ )  $\{P\ n\}$ 
proof (rule hyper-hoare-tripleI)
  fix  $S$  assume  $P\ n\ S$ 
  moreover have sem (Assume ( $\text{lnot } b$ ))  $S \subseteq S$ 
    by (simp add: assume-sem)
  ultimately show  $P\ n$  (sem (Assume ( $\text{lnot } b$ ))  $S$ )
    by (meson assms(3) downwards-closed-def)
qed
qed

```

```

lemma in-union-up-to:
   $x \in \text{union-up-to-}n\ C\ S\ n \longleftrightarrow (\exists m. m \leq n \wedge x \in \text{iterate-sem } m\ C\ S)$ 
proof (induct n)
  case (Suc n)
  then show ?case
    by (metis UnCI UnE le-SucE le-SucI order-refl union-up-to-n.simps(2))
qed (simp)

```

```

theorem rule-while-terminates-strong:
  assumes  $\bigwedge n. n < m \implies \models \{P\ n\}$  if-then  $b\ C\ \{P\ (Suc\ n)\}$ 
    and  $\bigwedge S. P\ m\ S \longrightarrow \text{holds-forall } (\text{lnot } b)\ S$ 
  shows  $\models \{P\ 0\}$  while-cond  $b\ C\ \{P\ m\}$ 
proof (rule hyper-hoare-tripleI)
  fix  $S$  assume asm0:  $P\ 0\ S$ 
  let  $?S = \text{iterate-sem } m\ (\text{if-then } b\ C)\ S$ 
  let  $?S' = \text{iterate-sem } (Suc\ m)\ (\text{if-then } b\ C)\ S$ 
  have  $P\ m\ ?S$ 
    using asm0 assms(1) indexed-invariant-then-power-bounded by blast
  then have holds-forall ( $\text{lnot } b$ )  $?S$ 
    using assms(2) by auto
  moreover have sem (while-cond  $b\ C$ )  $S = \text{filter-exp } (\text{lnot } b)\ (\bigcup n. \text{iterate-sem } n\ (\text{Assume } b;; C)\ S)$ 
    by (simp add: assume-sem filter-exp-def sem-seq sem-while while-cond-def)

```

```

  then have  $P\ m$  (filter-exp ( $\text{lnot } b$ ) (union-up-to-}n (Assume  $b;; C$ )  $S\ m$ )  $\cup$  iterate-sem  $m$  (Assume  $b;; C$ )  $S$ )
    by (metis  $\langle P\ m$  (iterate-sem  $m$  (if-then  $b\ C$ )  $S$ )  $\rangle$  iterate-sem-equiv)

```

```

moreover have iterate-sem  $m$  (Assume  $b;; C$ )  $S \subseteq \text{filter-exp } (\text{lnot } b)\ (\text{union-up-to-}n\ (\text{Assume } b;; C)\ S\ m)$ 
proof
  fix  $x$  assume  $x \in \text{iterate-sem } m\ (\text{Assume } b;; C)\ S$ 
  then have  $x \in \text{union-up-to-}n\ (\text{Assume } b;; C)\ S\ m$ 
    by (metis UnCI union-up-to-n.elims)

```

```

then have  $x \in ?S$ 
  by (simp add:  $\langle x \in \text{iterate-}sem\ m\ (\text{Assume}\ b\ ;;\ C)\ S \rangle \text{iterate-}sem\equiv$ )
then have  $\text{lnot}\ b\ (\text{snd}\ x)$ 
  by (metis calculation(1) holds-forall-def)
then show  $x \in \text{filter-exp}\ (\text{lnot}\ b)\ (\bigcup n. \text{iterate-}sem\ n\ (\text{Assume}\ b\ ;;\ C)\ S\ m)$ 
  using  $\langle x \in \text{union-up-to-}n\ (\text{Assume}\ b\ ;;\ C)\ S\ m \rangle$ 
  by (simp add: filter-exp-def)
qed
moreover have  $\text{filter-exp}\ (\text{lnot}\ b)\ (\bigcup n. \text{iterate-}sem\ n\ (\text{Assume}\ b\ ;;\ C)\ S)$ 
 $= \text{filter-exp}\ (\text{lnot}\ b)\ (\text{union-up-to-}n\ (\text{Assume}\ b\ ;;\ C)\ S\ m)$ 
proof –
  have  $\bigwedge n. n > m \implies \text{iterate-}sem\ n\ (\text{Assume}\ b\ ;;\ C)\ S = \{\}$ 
  proof –
    fix  $n$  show  $n > m \implies \text{iterate-}sem\ n\ (\text{Assume}\ b\ ;;\ C)\ S = \{\}$ 
    proof (induct  $n - m - 1$ )
      case 0
      then show ?case
        by (metis (no-types, lifting) UnCI calculation(1) false-then-empty-later
holds-forall-def iterate-sem-equiv)
      next
      case (Suc  $x$ )
      then show ?case
        by (metis (no-types, lifting) UnCI calculation(1) false-then-empty-later
holds-forall-def iterate-sem-equiv)
    qed
  qed
  moreover have  $\text{union-up-to-}n\ (\text{Assume}\ b\ ;;\ C)\ S\ m = (\bigcup n. \text{union-up-to-}n$ 
 $(\text{Assume}\ b\ ;;\ C)\ S\ n)$  (is  $?A = ?B$ )
  proof
    show  $?B \subseteq ?A$ 
    proof
      fix  $x$  assume  $x \in ?B$ 
      then obtain  $n$  where  $x \in \text{union-up-to-}n\ (\text{Assume}\ b\ ;;\ C)\ S\ n$ 
      by blast
      then show  $x \in ?A$ 
      by (metis calculation empty-iff in-union-up-to linorder-not-le)
    qed
  qed (blast)
  then have  $(\bigcup n. \text{iterate-}sem\ n\ (\text{Assume}\ b\ ;;\ C)\ S) = \text{union-up-to-}n\ (\text{Assume}$ 
 $b\ ;;\ C)\ S\ m$ 
  by (simp add: union-union-up-to-n-equiv)
  then show ?thesis
  by auto
qed
ultimately show  $P\ m\ (\text{sem}\ (\text{while-cond}\ b\ C)\ S)$ 
  by (simp add:  $\langle \text{sem}\ (\text{while-cond}\ b\ C)\ S = \text{filter-exp}\ (\text{lnot}\ b)\ (\bigcup n. \text{iterate-}sem$ 
 $n\ (\text{Assume}\ b\ ;;\ C)\ S) \rangle \text{sup.absorb1}$ )
qed

```

lemma *false-state-in-if-then*:
assumes $\varphi \in S$
and $\neg b$ (*snd* φ)
shows $\varphi \in \text{sem}$ (*if-then* b C) S
proof –
have $\varphi \in \text{sem}$ (*Assume* (*lnot* b)) S
by (*metis SemAssume assms(1) assms(2) in-sem lnot-def prod.collapse*)
then show *?thesis*
by (*simp add: assume-sem filter-exp-def if-then-sem*)
qed

lemma *false-state-in-while-cond-aux*:
assumes $\varphi \in S$
and $\neg b$ (*snd* φ)
shows $\varphi \in \text{iterate-sem } n$ (*if-then* b C) S
proof (*induct* n)
case 0
then show *?case*
by (*simp add: assms(1)*)
next
case (*Suc* n)
then show *?case*
by (*simp add: assms(2) false-state-in-if-then*)
qed

lemma *false-state-in-while-cond*:
assumes $\varphi \in S$
and $\neg b$ (*snd* φ)
shows $\varphi \in \text{sem}$ (*while-cond* b C) S
proof –
have $\varphi \in (\bigcup n. \text{iterate-sem } n$ (*if-then* b C) S)
by (*simp add: assms(1) assms(2) false-state-in-while-cond-aux*)
then show *?thesis using sem-while-with-if*[*of* b C S] *assms(2)*
by (*simp add: filter-exp-def lnot-def*)
qed

theorem *while-exists*:
assumes $\bigwedge \varphi. \models \{ P \varphi \}$ *while-cond* b C $\{ Q \varphi \}$
shows $\models \{ (\lambda S. \exists \varphi \in S. \neg b$ (*snd* φ) $\wedge P \varphi S) \}$ *while-cond* b C $\{ (\lambda S. \exists \varphi \in S. Q \varphi S) \}$
proof (*rule hyper-hoare-tripleI*)
fix S **assume** $\exists \varphi \in S. \neg b$ (*snd* φ) $\wedge P \varphi S$
then obtain φ **where** *asm0*: $\varphi \in S$ $\neg b$ (*snd* φ) $\wedge P \varphi S$ **by** *blast*
then have $Q \varphi$ (*sem* (*while-cond* b C) S)
using *assms hyper-hoare-tripleE* **by** *blast*
then show $\exists \varphi \in \text{sem}$ (*while-cond* b C) $S. Q \varphi$ (*sem* (*while-cond* b C) S)
using *asm0(1) asm0(2) false-state-in-while-cond* **by** *blast*
qed

lemma *sem-while-cond-union-up-to*:
 $sem (while\text{-}cond\ b\ C)\ S = filter\text{-}exp (lnot\ b) (\bigcup n. union\text{-}up\text{-}to\ n (if\text{-}then\ b\ C)\ S\ n)$
by (*simp add: sem-while-with-if union-union-up-to-n-equiv*)

lemma *iterate-sem-sum*:
 $iterate\text{-}sem\ n\ C (iterate\text{-}sem\ m\ C\ S) = iterate\text{-}sem\ (n + m)\ C\ S$
by (*induct n simp-all*)

lemma *unroll-while-sem*:
 $sem (while\text{-}cond\ b\ C) (iterate\text{-}sem\ n (if\text{-}then\ b\ C)\ S) = sem (while\text{-}cond\ b\ C)\ S$
proof –
let $?S = iterate\text{-}sem\ n (if\text{-}then\ b\ C)\ S$
have $filter\text{-}exp (lnot\ b) (\bigcup m. iterate\text{-}sem\ m (if\text{-}then\ b\ C)\ S) = filter\text{-}exp (lnot\ b) (\bigcup m. iterate\text{-}sem\ (n + m) (if\text{-}then\ b\ C)\ S)$ (**is** $?A = ?B$)
proof
show $?A \subseteq ?B$
proof
fix x **assume** $x \in ?A$
then obtain m **where** $x \in iterate\text{-}sem\ m (if\text{-}then\ b\ C)\ S \neg b (snd\ x)$
by (*auto simp add: filter-exp-def lnot-def*)
then have $x \in iterate\text{-}sem\ (n + m) (if\text{-}then\ b\ C)\ S$
using *false-state-in-while-cond-aux*[*of x iterate-sem m (if-then b C) S b n C*] *iterate-sem-sum*[*of n if-then b C m S*]
by *blast*
then have $x \in (\bigcup m. iterate\text{-}sem\ (n + m) (if\text{-}then\ b\ C)\ S)$
by *blast*
then show $x \in ?B$
using $\langle x \in filter\text{-}exp (lnot\ b) (\bigcup m. iterate\text{-}sem\ m (if\text{-}then\ b\ C)\ S) \rangle$
by (*simp add: filter-exp-def*)
qed
show $?B \subseteq ?A$
proof
fix x **assume** $x \in ?B$
then obtain m **where** $x \in iterate\text{-}sem\ (n + m) (if\text{-}then\ b\ C)\ S \neg b (snd\ x)$
by (*auto simp add: filter-exp-def lnot-def*)
then show $x \in ?A$
using $\langle x \in filter\text{-}exp (lnot\ b) (\bigcup m. iterate\text{-}sem\ (n + m) (if\text{-}then\ b\ C)\ S) \rangle$
by (*auto simp add: filter-exp-def*)
qed
qed
then show *?thesis*
using *iterate-sem-sum*[*of - if-then b C n S*] *sem-while-with-if*[*of b C S*] *sem-while-with-if*[*of b C ?S*]
by (*simp add: add commute*)
qed

theorem *while-unroll*:

```

assumes  $\bigwedge n. n < m \implies \models \{P\ n\}$  if-then  $b\ C\ \{P\ (Suc\ n)\}$ 
and  $\models \{P\ m\}$  while-cond  $b\ C\ \{Q\}$ 
shows  $\models \{P\ 0\}$  while-cond  $b\ C\ \{Q\}$ 
proof (rule hyper-hoare-tripleI)
fix  $S$  assume  $P\ 0\ S$ 
let  $?S = \text{iterate-sem } m\ (\text{if-then } b\ C)\ S$ 
have  $(\forall n. n < m \longrightarrow (\models \{P\ n\}\ \text{if-then } b\ C\ \{P\ (Suc\ n)\})) \longrightarrow P\ m\ ?S$ 
proof (induct m)
case  $0$ 
then show  $?case$ 
by (simp add: <P 0 S>)
next
case  $(Suc\ m)$ 
then show  $?case$ 
by (simp add: hyper-hoare-triple-def)
qed
then have  $P\ m\ ?S$  using assms(1)
by blast
then have  $Q\ (\text{sem } (\text{while-cond } b\ C)\ ?S)$ 
using assms(2) hyper-hoare-tripleE by blast
then show  $Q\ (\text{sem } (\text{while-cond } b\ C)\ S)$ 
by (metis unroll-while-sem)
qed

```

Deriving LoopExit from NormalWhile, and ForLoop from LoopExit and Unroll

lemma *while-desugared-easy*:

```

assumes  $\bigwedge n. \models \{I\ n\}$  Assume  $b;;\ C\ \{I\ (Suc\ n)\}$ 
and  $\models \{ \text{natural-partition } I \}$  Assume  $(\text{lnot } b)\ \{ Q \}$ 
shows  $\models \{I\ 0\}$  while-cond  $b\ C\ \{ Q \}$ 
by (metis assms(1) assms(2) seq-rule while-cond-def while-rule)

```

corollary *loop-exit*:

```

assumes entails  $P\ (\text{holds-forall } (\text{lnot } b))$ 
shows  $\models \{P\}$  while-cond  $b\ C\ \{P\}$ 
proof –
have  $\models \{(if\ (0::nat) = 0\ \text{then } P\ \text{else } emp)\}$  while-cond  $b\ C\ \{P\}$ 
proof (rule while-desugared-easy[of  $\lambda(n::nat). \text{if } n = 0\ \text{then } P\ \text{else } emp\ b\ C\ P$ ])
show  $\models \{ \text{natural-partition } (\lambda(n::nat). \text{if } n = 0\ \text{then } P\ \text{else } emp) \}$  Assume  $(\text{lnot } b)\ \{P\}$ 
proof (rule hyper-hoare-tripleI)
fix  $S$  assume asm0: natural-partition  $(\lambda(n::nat). \text{if } n = 0\ \text{then } P\ \text{else } emp)\ S$ 
then obtain  $F$  where  $S = (\bigcup (n::nat). F\ n) \wedge (n::nat). (\lambda(n::nat). \text{if } n = 0\ \text{then } P\ \text{else } emp)\ n\ (F\ n)$ 
using natural-partitionE by blast
then have  $\bigwedge n. F\ (Suc\ n) = \{\}$ 

```

```

    by (metis (mono-tags, lifting) emp-def old.nat.distinct(2))
  moreover have  $S = F\ 0$ 
  proof
    show  $S \subseteq F\ 0$ 
    proof
      fix  $x$  assume  $x \in S$ 
      then obtain  $n$  where  $x \in F\ n$ 
      using  $\langle S = \bigcup (\text{range } F) \rangle$  by blast
      then show  $x \in F\ 0$ 
      by (metis calculation empty-iff gr0-implies-Suc zero-less-iff-neq-zero)
    qed
    show  $F\ 0 \subseteq S$ 
    using  $\langle S = \bigcup (\text{range } F) \rangle$  by blast
  qed
  ultimately have  $P\ S$ 
  using  $\langle \bigwedge n. (\text{if } n = 0 \text{ then } P \text{ else emp}) (F\ n) \rangle$  by presburger
  then show  $P$  (sem (Assume (lnot b))  $S$ )
  by (metis assms entailsE sem-assume-low-exp(1))
  qed
  fix  $n :: \text{nat}$ 
  show  $\models \{(\text{if } n = 0 \text{ then } P \text{ else emp})\} \text{ Assume } b ;; C \{ \text{if } \text{Suc } n = 0 \text{ then } P \text{ else emp} \}$ 
  proof (rule hyper-hoare-tripleI)
    fix  $S$  assume  $\text{asm0}: (\text{if } n = 0 \text{ then } P \text{ else emp})\ S$ 
    then show  $(\text{if } \text{Suc } n = 0 \text{ then } P \text{ else emp})$  (sem (Assume  $b ;; C$ )  $S$ )
    by (metis (mono-tags, lifting) assms emp-def entailsE holds-forall-empty
    lnot-involution nat.distinct(1) sem-assume-low-exp-seq(2))
  qed
  qed
  then show ?thesis
  by fastforce
  qed

```

```

corollary for-loop:
  assumes  $\bigwedge n. n < m \implies \models \{P\ n\} \text{ if-then } b\ C \{P\ (\text{Suc } n)\}$ 
  and entails  $(P\ m)$  (holds-forall (lnot b))
  shows  $\models \{P\ 0\} \text{ while-cond } b\ C \{P\ m\}$ 
  using assms(1)
  proof (rule while-unroll)
    show  $\models \{P\ m\} \text{ while-cond } b\ C \{P\ m\}$ 
    using assms(2) loop-exit by blast
  qed

```

end

5 Compositionality Rules

theory *Compositionality*

imports *Logic Expressivity Loops*
begin

In this file, we prove the soundness of all compositionality rules presented in Appendix D (figure 11).

definition *in-set* **where**
in-set $\varphi S \longleftrightarrow \varphi \in S$

5.1 Linking rule

proposition *rule-linking*:

assumes $\bigwedge \varphi 1 (\varphi 2 :: ('a, 'b, 'c, 'd) \text{ state}). \text{fst } \varphi 1 = \text{fst } \varphi 2 \wedge (\models \{ (\text{in-set } \varphi 1 :: ('a, 'b, 'c, 'd) \text{ state}) \text{ hyperassertion} \} C \{ \text{in-set } \varphi 2 \})$
 $\implies (\models \{ (P \varphi 1 :: ('a, 'b, 'c, 'd) \text{ state}) \text{ hyperassertion} \} C \{ Q \varphi 2 \})$
shows $\models \{ ((\lambda S. \forall \varphi 1 \in S. P \varphi 1 S) :: ('a, 'b, 'c, 'd) \text{ state}) \text{ hyperassertion} \} C \{ (\lambda S. \forall \varphi 2 \in S. Q \varphi 2 S) \}$

proof (*rule hyper-hoare-tripleI*)

fix S **assume** $\text{asm0}: \forall \varphi 1 \in S. P \varphi 1 S$

show $\forall \varphi 2 \in \text{sem } C S. Q \varphi 2 (\text{sem } C S)$

proof (*clarify*)

fix $l \sigma'$ **assume** $(l, \sigma') \in \text{sem } C S$

then obtain σ **where** $(l, \sigma) \in S \text{ single-sem } C \sigma \sigma'$

by (*metis fst-conv in-sem snd-conv*)

then show $Q (l, \sigma') (\text{sem } C S)$

by (*metis (mono-tags, opaque-lifting) asm0 assms fst-eqD hyper-hoare-triple-def in-set-def single-step-then-in-sem*)

qed

qed

lemma *rule-linking-alt*:

assumes $\bigwedge l \sigma \sigma'. \text{single-sem } C \sigma \sigma' \implies (\models \{ P (l, \sigma) \} C \{ Q (l, \sigma') \})$

shows $\models \{ (\lambda S. \forall \omega \in S. P \omega S) \} C \{ (\lambda S. \forall \omega' \in S. Q \omega' S) \}$

proof (*rule hyper-hoare-tripleI*)

fix S **assume** $\text{asm0}: \forall \omega \in S. P \omega S$

show $\forall \omega' \in \text{sem } C S. Q \omega' (\text{sem } C S)$

proof (*clarify*)

fix $l \sigma'$ **assume** $(l, \sigma') \in \text{sem } C S$

then obtain σ **where** $(l, \sigma) \in S \text{ single-sem } C \sigma \sigma'$

by (*metis fst-conv in-sem snd-conv*)

then have $\models \{ P (l, \sigma) \} C \{ Q (l, \sigma') \}$

by (*simp add: assms*)

then show $Q (l, \sigma') (\text{sem } C S)$

by (*simp add: <l, sigma> in S asm0 hyper-hoare-tripleE*)

qed

qed

5.2 Frame rules

lemma *rule-lframe*:

fixes $b :: ('a \Rightarrow ('lvar \Rightarrow 'lval)) \Rightarrow bool$
 — b takes a mapping from keys to logical states (representing the tuple), and returns a boolean

shows $\models \{ (\lambda S. \forall \varphi. (\forall k. \varphi k \in S) \longrightarrow b (fst \circ \varphi)) \} C \{ \lambda S. \forall \varphi. (\forall k. \varphi k \in S) \longrightarrow b (fst \circ \varphi) \}$

proof (*rule hyper-hoare-tripleI*)

fix $S :: ('lvar, 'lval, 'b, 'c)$ state set

assume $asm0: \forall \varphi. (\forall k. \varphi k \in S) \longrightarrow b (fst \circ \varphi)$

show $\forall \varphi. (\forall k. \varphi k \in sem\ C\ S) \longrightarrow b (fst \circ \varphi)$

proof (*clarify*)

fix $\varphi' :: 'a \Rightarrow ('lvar, 'lval, 'b, 'c)$ state

assume $asm1: \forall k. \varphi' k \in sem\ C\ S$

let $? \varphi = \lambda k. SOME\ \varphi k. fst\ \varphi k = fst\ (\varphi' k) \wedge \varphi k \in S \wedge single-sem\ C\ (snd\ \varphi k)$
 ($snd\ (\varphi' k)$)

have $r: \bigwedge k. fst\ (? \varphi k) = fst\ (\varphi' k) \wedge (? \varphi k) \in S \wedge single-sem\ C\ (snd\ (? \varphi k))$
 ($snd\ (\varphi' k)$)

proof —

fix k **show** $fst\ (? \varphi k) = fst\ (\varphi' k) \wedge (? \varphi k) \in S \wedge single-sem\ C\ (snd\ (? \varphi k))$
 ($snd\ (\varphi' k)$)

proof (*rule someI-ex*)

show $\exists x. fst\ x = fst\ (\varphi' k) \wedge x \in S \wedge \langle C, snd\ x \rangle \rightarrow snd\ (\varphi' k)$

by (*metis asm1 fst-conv in-sem snd-conv*)

qed

qed

then have $b (fst \circ ? \varphi)$

using $asm0$ **by** *presburger*

moreover have $fst \circ ? \varphi = fst \circ \varphi'$

using $ext\ r$ **by** *fastforce*

then show $b (fst \circ \varphi')$

using *calculation* **by** *presburger*

qed

qed

lemma *rule-lframe-single*:

$\models \{ (\lambda S. \forall \omega \in S. P (fst\ \omega)) \} C \{ \lambda S. \forall \omega \in S. P (fst\ \omega) \}$

proof —

let $?P = \lambda \varphi. P (\varphi ())$

have $\models \{ (\lambda S. \forall \varphi. (\forall k. \varphi k \in S) \longrightarrow ?P (fst \circ \varphi)) \} C \{ \lambda S. \forall \varphi. (\forall k. \varphi k \in S) \longrightarrow ?P (fst \circ \varphi) \}$

using *rule-lframe* **by** *fast*

moreover have $(\lambda S. \forall \omega \in S. P (fst\ \omega)) = (\lambda S. \forall \varphi. (\forall k. \varphi k \in S) \longrightarrow ?P (fst \circ \varphi))$

using ext **by** *fastforce*

ultimately show $?thesis$

using *forw-subst ssubst order-trans-rules(13) prop-subst basic-trans-rules(13)*

by *fast*

qed

definition *differ-only-by-pset where*

differ-only-by-pset vars a b $\longleftrightarrow (\forall i. \text{fst } (a \ i) = \text{fst } (b \ i) \wedge \text{differ-only-by-set vars } (snd \ (a \ i)) \ (snd \ (b \ i)))$

lemma *differ-only-by-psetI:*

assumes $\bigwedge i. \text{fst } (a \ i) = \text{fst } (b \ i) \wedge \text{differ-only-by-set vars } (snd \ (a \ i)) \ (snd \ (b \ i))$

shows *differ-only-by-pset vars a b*

by (*simp add: assms differ-only-by-pset-def*)

definition *not-in-free-pvars-prop where*

not-in-free-pvars-prop vars b $\longleftrightarrow (\forall \varphi 1 \ \varphi 2. \text{differ-only-by-pset vars } \varphi 1 \ \varphi 2 \longrightarrow (b \ \varphi 1 \longleftrightarrow b \ \varphi 2))$

proposition *rule-frame:*

fixes $b :: ('a \Rightarrow ('lvar, 'lval, 'pvar, 'pval) \text{ state}) \Rightarrow \text{bool}$

— b takes a mapping from keys to extended states (representing the tuple), and returns a boolean

assumes *not-in-free-pvars-prop (written-vars C) b*

shows $\models \{ (\lambda S. \forall \varphi. (\forall k. \varphi \ k \in S) \longrightarrow b \ \varphi) \} \ C \ \{ \lambda S. \forall \varphi. (\forall k. \varphi \ k \in S) \longrightarrow b \ \varphi \}$

proof (*rule hyper-hoare-tripleI*)

fix $S :: ('lvar, 'lval, 'pvar, 'pval) \text{ state set}$

assume $asm0: \forall \varphi. (\forall k. \varphi \ k \in S) \longrightarrow b \ \varphi$

show $\forall \varphi. (\forall k. \varphi \ k \in \text{sem } C \ S) \longrightarrow b \ \varphi$

proof (*clarify*)

fix $\varphi' :: 'a \Rightarrow ('lvar, 'lval, 'pvar, 'pval) \text{ state}$

assume $asm1: \forall k. \varphi' \ k \in \text{sem } C \ S$

let $?\varphi = \lambda k. \text{SOME } \varphi k. \text{fst } \varphi k = \text{fst } (\varphi' \ k) \wedge \varphi k \in S \wedge \text{single-sem } C \ (snd \ \varphi k)$
($snd \ (\varphi' \ k)$)

have $r: \bigwedge k. \text{fst } (? \varphi \ k) = \text{fst } (\varphi' \ k) \wedge (? \varphi \ k) \in S \wedge \text{single-sem } C \ (snd \ (? \varphi \ k))$
($snd \ (\varphi' \ k)$)

proof —

fix k **show** $\text{fst } (? \varphi \ k) = \text{fst } (\varphi' \ k) \wedge (? \varphi \ k) \in S \wedge \text{single-sem } C \ (snd \ (? \varphi \ k))$
($snd \ (\varphi' \ k)$)

proof (*rule someI-ex*)

show $\exists x. \text{fst } x = \text{fst } (\varphi' \ k) \wedge x \in S \wedge \langle C, \text{snd } x \rangle \rightarrow \text{snd } (\varphi' \ k)$

by (*metis asm1 fst-conv in-sem snd-conv*)

qed

qed

then have $b \ ? \varphi$

using $asm0$ **by** *presburger*

moreover have *differ-only-by-pset (written-vars C) ?\varphi \varphi'*

proof (*rule differ-only-by-psetI*)

```

fix  $i$  show  $\text{fst } (SOME \ \varphi k. \ \text{fst } \varphi k = \text{fst } (\varphi' \ i) \wedge \varphi k \in S \wedge \langle C, \text{snd } \varphi k \rangle \rightarrow$ 
 $\text{snd } (\varphi' \ i)) = \text{fst } (\varphi' \ i) \wedge$ 
 $\text{differ-only-by-set } (\text{written-vars } C) (\text{snd } (SOME \ \varphi k. \ \text{fst } \varphi k = \text{fst } (\varphi' \ i) \wedge \varphi k$ 
 $\in S \wedge \langle C, \text{snd } \varphi k \rangle \rightarrow \text{snd } (\varphi' \ i))) (\text{snd } (\varphi' \ i))$ 
by ( $\text{metis } (\text{mono-tags}, \text{lifting}) \text{differ-only-by-set-def } r \text{written-vars-not-modified}$ )
qed
ultimately show  $b \ \varphi'$ 
using  $\text{assms not-in-free-pvars-prop-def}$  by  $\text{blast}$ 
qed
qed

```

5.3 Logical Updates

definition *equal-outside-set* **where**

$\text{equal-outside-set vars } l1 \ l2 \longleftrightarrow (\forall x. x \notin \text{vars} \longrightarrow l1 \ x = l2 \ x)$

lemma *equal-outside-setI*:

assumes $\bigwedge x. x \notin \text{vars} \implies l1 \ x = l2 \ x$

shows *equal-outside-set vars* $l1 \ l2$

using $\text{assms equal-outside-set-def}$ **by** auto

lemma *equal-outside-setE*:

assumes *equal-outside-set vars* $l1 \ l2$

and $x \notin \text{vars}$

shows $l1 \ x = l2 \ x$

using $\text{assms equal-outside-set-def}$ **by** meson

lemma *equal-outside-sym*:

$\text{equal-outside-set vars } l \ l' \longleftrightarrow \text{equal-outside-set vars } l' \ l$

by ($\text{metis equal-outside-set-def}$)

definition *subset-mod-updates* **where**

$\text{subset-mod-updates vars } S \ S' \longleftrightarrow (\forall \omega \in S. \exists \omega' \in S'. \text{snd } \omega = \text{snd } \omega' \wedge$
 $\text{equal-outside-set vars } (\text{fst } \omega) (\text{fst } \omega'))$

lemma *subset-mod-updatesI*:

assumes $\bigwedge \omega. \omega \in S \implies (\exists \omega' \in S'. \text{snd } \omega = \text{snd } \omega' \wedge \text{equal-outside-set vars } (\text{fst } \omega) (\text{fst } \omega'))$

shows *subset-mod-updates vars* $S \ S'$

by ($\text{simp add: assms subset-mod-updates-def}$)

lemma *subset-mod-updatesE*:

assumes *subset-mod-updates vars* $S \ S'$

and $\omega \in S$

shows $\exists \omega' \in S'. \text{snd } \omega = \text{snd } \omega' \wedge \text{equal-outside-set vars } (\text{fst } \omega) (\text{fst } \omega')$

using $\text{assms}(1) \ \text{assms}(2) \ \text{subset-mod-updates-def}$ **by** blast

definition *same-mod-updates* **where**

same-mod-updates vars S S' \longleftrightarrow subset-mod-updates vars S S' \wedge subset-mod-updates vars S' S

lemma *same-mod-updatesI:*

assumes $\bigwedge \omega. \omega \in S \implies (\exists \omega' \in S'. \text{snd } \omega = \text{snd } \omega' \wedge \text{equal-outside-set vars (fst } \omega) \text{ (fst } \omega'))$

and $\bigwedge \omega'. \omega' \in S' \implies (\exists \omega \in S. \text{snd } \omega = \text{snd } \omega' \wedge \text{equal-outside-set vars (fst } \omega') \text{ (fst } \omega))$

shows *same-mod-updates vars S S'*

by (*metis assms(1) assms(2) same-mod-updates-def subset-mod-updates-def*)

lemma *same-mod-updates-sym:*

same-mod-updates vars S S' \longleftrightarrow same-mod-updates vars S' S

using *same-mod-updates-def* **by** *blast*

lemma *same-mod-updates-refl:*

same-mod-updates vars S S

by (*metis equal-outside-set-def same-mod-updates-def subset-mod-updates-def*)

lemma *equal-outside-set-trans:*

assumes *equal-outside-set vars a b*

and *equal-outside-set vars b c*

shows *equal-outside-set vars a c*

using *equal-outside-set-def[of vars a b] equal-outside-set-def[of vars b c] equal-outside-set-def[of vars a c] assms*

by *presburger*

lemma *subset-mod-updates-trans:*

assumes *subset-mod-updates vars S1 S2*

and *subset-mod-updates vars S2 S3*

shows *subset-mod-updates vars S1 S3*

proof (*rule subset-mod-updatesI*)

fix $\omega 1$ **assume** $\omega 1 \in S 1$

then obtain $\omega 2$ **where** $\omega 2 \in S 2$ $\text{snd } \omega 1 = \text{snd } \omega 2 \wedge \text{equal-outside-set vars (fst } \omega 1) \text{ (fst } \omega 2)$

using *assms(1) same-mod-updates-def subset-mod-updatesE* **by** *blast*

then show $\exists \omega' \in S 3. \text{snd } \omega 1 = \text{snd } \omega' \wedge \text{equal-outside-set vars (fst } \omega 1) \text{ (fst } \omega')$

by (*metis (no-types, lifting) assms(2) equal-outside-set-trans subset-mod-updatesE*)

qed

lemma *same-mod-updates-trans:*

assumes *same-mod-updates vars S1 S2*

and *same-mod-updates vars S2 S3*

shows *same-mod-updates vars S1 S3*

by (*meson assms(1) assms(2) same-mod-updates-def subset-mod-updates-trans*)

lemma *sem-update-commute-aux:*

assumes *subset-mod-updates vars S1 S2*

shows *subset-mod-updates vars (sem C S1) (sem C S2)*
proof (*rule subset-mod-updatesI*)
fix $\omega 1$ **assume** *asm0: $\omega 1 \in \text{sem } C \ S1$*
then obtain $l1 \ \sigma$ **where** $(l1, \sigma) \in S1$ *single-sem C σ (snd $\omega 1$) fst $\omega 1 = l1$*
by (*metis in-sem*)
then obtain $l2$ **where** $(l2, \sigma) \in S2$ *equal-outside-set vars $l1 \ l2$*
using *assms subset-mod-updatesE* **by** *fastforce*
then show $\exists \omega' \in \text{sem } C \ S2. \text{snd } \omega 1 = \text{snd } \omega' \wedge \text{equal-outside-set vars (fst } \omega 1)$
 $(\text{fst } \omega')$
by (*metis $\langle C, \sigma \rangle \rightarrow \text{snd } \omega 1 \langle \text{fst } \omega 1 = l1 \rangle \text{fst-conv in-sem snd-conv}$*)
qed

lemma *sem-update-commute:*
assumes *same-mod-updates (vars :: 'a set) S1 S2*
shows *same-mod-updates vars (sem C S1) (sem C S2)*
using *assms same-mod-updates-def sem-update-commute-aux* **by** *blast*

type-synonym $(\ 'a, \ 'b, \ 'c, \ 'd)$ *chyperassertion* = $((\ 'a \Rightarrow \ 'b) \times (\ 'c \Rightarrow \ 'd)) \ \text{set} \Rightarrow \ \text{bool}$

definition *invariant-on-updates :: 'a set $\Rightarrow (\ 'a, \ 'b, \ 'c, \ 'd)$ chyperassertion $\Rightarrow \ \text{bool}$*
where
invariant-on-updates vars P $\longleftrightarrow (\forall S \ S'. \text{same-mod-updates vars } S \ S' \longrightarrow (P \ S \longleftrightarrow P \ S'))$

lemma *invariant-on-updatesI:*
assumes $\bigwedge S \ S'. \text{same-mod-updates vars } S \ S' \Longrightarrow P \ S \Longrightarrow P \ S'$
shows *invariant-on-updates vars P*
using *assms invariant-on-updates-def same-mod-updates-sym* **by** *blast*

definition *entails-with-updates :: 'a set $\Rightarrow (\ 'a, \ 'b, \ 'c, \ 'd)$ chyperassertion $\Rightarrow (\ 'a, \ 'b, \ 'c, \ 'd)$ chyperassertion $\Rightarrow \ \text{bool}$*
where
entails-with-updates vars P Q $\longleftrightarrow (\forall S. P \ S \longrightarrow (\exists S'. \text{same-mod-updates vars } S \ S' \wedge Q \ S'))$

lemma *entails-with-updatesI:*
assumes $\bigwedge S. P \ S \Longrightarrow (\exists S'. \text{same-mod-updates vars } S \ S' \wedge Q \ S')$
shows *entails-with-updates vars P Q*
by (*simp add: assms entails-with-updates-def*)

lemma *entails-with-updatesE:*
assumes *entails-with-updates vars P Q*
and $P \ S$
shows $\exists S'. \text{same-mod-updates vars } S \ S' \wedge Q \ S'$

by (*meson* *assms(1)* *assms(2)* *entails-with-updates-def*)

proposition *rule-LUpdate*:

assumes $\models \{P'\} C \{Q\}$
and *entails-with-updates vars P P'*
and *invariant-on-updates vars Q*
shows $\models \{P\} C \{Q\}$
proof (*rule hyper-hoare-tripleI*)
fix *S* **assume** *asm0: P S*
then obtain *S'* **where** *same-mod-updates vars S S' \wedge P' S'*
using *assms(2)* *entails-with-updatesE* **by** *blast*
then have *same-mod-updates vars (sem C S) (sem C S')*
using *sem-update-commute* **by** *blast*
moreover have *Q (sem C S')*
using \langle *same-mod-updates vars S S' \wedge P' S'* \rangle *assms(1)* *hyper-hoare-tripleE* **by**
auto
ultimately show *Q (sem C S)*
using *assms(3)* *invariant-on-updates-def* **by** *blast*
qed

5.4 Filters

lemma *filter-prop-commute-aux*:

assumes $\bigwedge \omega \omega'. \text{fst } \omega = \text{fst } \omega' \implies (f \omega \longleftrightarrow f \omega')$
shows *Set.filter f (sem C S) = sem C (Set.filter f S)* (**is** $?A = ?B$)
proof
show $?A \subseteq ?B$
proof
fix ω' **assume** $\omega' \in ?A$
then obtain ω **where** $\omega \in S$ *single-sem C (snd ω) (snd ω')* $\text{fst } \omega' = \text{fst } \omega$ *f*
 ω'
by (*auto simp add: in-sem*)
then show $\omega' \in ?B$
using *assms [OF \langle fst $\omega' = \text{fst } \omega$ $\rangle]$* **by** (*auto simp add: in-sem intro: exI [of -*
 \langle *snd ω $\rangle]$ *)*)
qed
show $?B \subseteq ?A$
proof
fix x **assume** $x \in \text{sem } C (\text{Set.filter } f S)$
then show $x \in \text{Set.filter } f (\text{sem } C S)$
by *simp (metis (mono-tags, lifting) assms fst-conv in-sem mem-Collect-eq)*
qed
qed*

definition *commute-with-sem where*

commute-with-sem $f \longleftrightarrow (\forall S C. f (\text{sem } C S) = \text{sem } C (f S))$

lemma *commute-with-semI*:

assumes $\bigwedge(S :: (('a \Rightarrow 'b) \times ('c \Rightarrow 'd)) \text{ set}) C. f (\text{sem } C S) = \text{sem } C (f S)$
shows *commute-with-sem f*
by (*simp add: assms commute-with-sem-def*)

lemma *filter-prop-commute*:

assumes $\bigwedge\omega \omega'. \text{fst } \omega = \text{fst } \omega' \implies (f \omega \longleftrightarrow f \omega')$
shows *commute-with-sem (Set.filter f)*
using *assms commute-with-sem-def filter-prop-commute-aux* **by** *blast*

lemma *rule-apply*:

assumes $\models \{P\} C \{Q\}$
and *commute-with-sem f*
shows $\models \{P \circ f\} C \{Q \circ f\}$
proof (*rule hyper-hoare-tripleI*)
fix *S* **assume** $(P \circ f) S$
then show $(Q \circ f) (\text{sem } C S)$
by (*metis assms(1) assms(2) commute-with-sem-def comp-apply hyper-hoare-tripleE*)
qed

definition *apply-filter where*

apply-filter b P S $\longleftrightarrow P (\text{Set.filter } b S)$

proposition *rule-LFilter*:

assumes $\models \{P\} C \{Q\}$
shows $\models \{ P \circ (\text{Set.filter } (b \circ \text{fst})) \} C \{ Q \circ (\text{Set.filter } (b \circ \text{fst})) \}$
by (*simp add: assms filter-prop-commute rule-apply*)

definition *differ-only-by-pset-single where*

differ-only-by-pset-single vars a b $\longleftrightarrow (\text{fst } a = \text{fst } b \wedge \text{differ-only-by-set vars } (\text{snd } a) (\text{snd } b))$

definition *not-in-free-pvars-perp where*

not-in-free-pvars-perp vars b $\longleftrightarrow (\forall \varphi 1 \varphi 2. \text{differ-only-by-set vars } \varphi 1 \varphi 2 \longrightarrow (b \varphi 1 \longleftrightarrow b \varphi 2))$

lemma *single-sem-differ-by-written-vars*:

assumes *single-sem C* $\varphi \varphi'$
shows *differ-only-by-set (written-vars C)* $\varphi \varphi'$
by (*meson assms differ-only-by-set-def written-vars-not-modified*)

lemma *single-sem-not-free-vars*:

assumes *not-in-free-pvars-perp (written-vars C)* *b*
and *single-sem C* $\varphi \varphi'$
shows $b \varphi \longleftrightarrow b \varphi'$
using *assms(1) assms(2) not-in-free-pvars-perp-def single-sem-differ-by-written-vars*
by *blast*

proposition *rule-PFilter*:

```

assumes  $\models \{P\} C \{Q\}$ 
  and not-in-free-pvars-pexp (written-vars  $C$ )  $b$ 
shows  $\models \{ P \circ (\text{Set.filter } (b \circ \text{snd})) \} C \{ Q \circ (\text{Set.filter } (b \circ \text{snd})) \}$ 
proof (rule hyper-hoare-tripleI)
  fix  $S$  assume  $\text{asm0}: (P \circ \text{Set.filter } (b \circ \text{snd})) S$ 
  let  $?S = \text{Set.filter } (b \circ \text{snd}) S$ 
  have  $Q$  (sem  $C$   $?S$ )
    using  $\text{asm0}$   $\text{assms}(1)$  hyper-hoare-tripleE by auto
  moreover have sem  $C$   $?S = \text{Set.filter } (b \circ \text{snd})$  (sem  $C$   $S$ ) (is  $?A = ?B$ )
proof
  show  $?B \subseteq ?A$ 
  proof
    fix  $\varphi'$  assume  $\varphi' \in \text{Set.filter } (b \circ \text{snd})$  (sem  $C$   $S$ )
    then obtain  $\varphi$  where  $\varphi \in S$   $\text{fst } \varphi = \text{fst } \varphi'$  single-sem  $C$  ( $\text{snd } \varphi$ ) ( $\text{snd } \varphi'$ )  $b$ 
    ( $\text{snd } \varphi'$ )
    by (auto simp add: in-sem)
    then have  $b$  ( $\text{snd } \varphi$ )
    using single-sem-not-free-vars[of  $C$   $b$   $\text{snd } \varphi$   $\text{snd } \varphi'$ ]  $\text{assms}(2)$ 
    by simp
    then show  $\varphi' \in ?A$ 
    using  $\langle \langle C, \text{snd } \varphi \rangle \rightarrow \text{snd } \varphi' \rangle \langle \varphi \in S \rangle \langle \text{fst } \varphi = \text{fst } \varphi' \rangle$  [symmetric]
    by (auto simp add: in-sem intro: exI [of -  $\langle \text{snd } \varphi \rangle$ ])
  qed
show  $?A \subseteq ?B$ 
proof
    fix  $\varphi'$  assume  $\varphi' \in \text{sem } C$  ( $\text{Set.filter } (b \circ \text{snd}) S$ )
    then obtain  $\varphi$  where  $\varphi \in S$   $\text{fst } \varphi = \text{fst } \varphi'$  single-sem  $C$  ( $\text{snd } \varphi$ ) ( $\text{snd } \varphi'$ )  $b$ 
    ( $\text{snd } \varphi$ )
    by (auto simp add: in-sem)
    then have  $b$  ( $\text{snd } \varphi'$ )
    using  $\text{assms}(2)$  single-sem-not-free-vars by blast
    then show  $\varphi' \in ?B$ 
    using  $\langle \langle C, \text{snd } \varphi \rangle \rightarrow \text{snd } \varphi' \rangle \langle \varphi \in S \rangle \langle \text{fst } \varphi = \text{fst } \varphi' \rangle$  [symmetric]
    by (auto simp add: in-sem intro: exI [of -  $\langle \text{snd } \varphi \rangle$ ])
  qed
qed
ultimately show  $(Q \circ \text{Set.filter } (b \circ \text{snd}))$  (sem  $C$   $S$ )
  by auto
qed

```

5.5 Other Compositionality Rules

proposition *rule-False*:

```

hyper-hoare-triple ( $\lambda-. \text{False}$ )  $C$   $Q$ 
by (simp add: hyper-hoare-triple-def)

```

proposition *rule-True*:

```

hyper-hoare-triple  $P$   $C$  ( $\lambda-. \text{True}$ )

```

by (simp add: hyper-hoare-triple-def)

lemma *sem-inter*:

$sem\ C\ (S1\ \cap\ S2)\ \subseteq\ sem\ C\ S1\ \cap\ sem\ C\ S2$
by (simp add: sem-monotonic)

proposition *rule-Union*:

assumes $\models \{P\}\ C\ \{Q\}$
and hyper-hoare-triple $P'\ C\ Q'$
shows hyper-hoare-triple $(join\ P\ P')\ C\ (join\ Q\ Q')$
proof (rule hyper-hoare-tripleI)
fix S assume $asm0: join\ P\ P'\ S$
then obtain $S1\ S2$ where $S = S1\ \cup\ S2\ P\ S1\ P'\ S2$
by (metis join-def)
then have $sem\ C\ S = sem\ C\ S1\ \cup\ sem\ C\ S2$
using sem-union by auto
then show $join\ Q\ Q'\ (sem\ C\ S)$
by (metis $\langle P\ S1\rangle\ \langle P'\ S2\rangle\ assms(1)\ assms(2)\ hyper-hoare-tripleE\ join-def$)
qed

proposition *rule-IndexedUnion*:

assumes $\bigwedge x. \models \{P\ x\}\ C\ \{Q\ x\}$
shows hyper-hoare-triple $(general-join\ P)\ C\ (general-join\ Q)$
proof (rule hyper-hoare-tripleI)
fix S assume $general-join\ P\ S$
then obtain F where $asm0: S = (\bigcup x. F\ x)\ \bigwedge x. P\ x\ (F\ x)$
by (meson general-join-def)
have $sem\ C\ S = (\bigcup x. sem\ C\ (F\ x))$
by (simp add: $asm0(1)\ sem-split-general$)
moreover have $\bigwedge x. Q\ x\ (sem\ C\ (F\ x))$
using $asm0(2)\ assms\ hyper-hoare-tripleE$ by blast
ultimately show $general-join\ Q\ (sem\ C\ S)$
by (metis general-join-def)
qed

proposition *rule-And*:

assumes $\models \{P\}\ C\ \{Q\}$
and hyper-hoare-triple $P'\ C\ Q'$
shows hyper-hoare-triple $(conj\ P\ P')\ C\ (conj\ Q\ Q')$
proof (rule hyper-hoare-tripleI)
fix S assume $Logic.conj\ P\ P'\ S$
then show $Logic.conj\ Q\ Q'\ (sem\ C\ S)$
by (metis $assms(1)\ assms(2)\ conj-def\ hyper-hoare-tripleE$)
qed

lemma *rule-Forall*:

assumes $\bigwedge x. \models \{P\} x\ C\ \{Q\} x$
shows *hyper-hoare-triple* (forall P) C (forall Q)
by (metis assms forall-def hyper-hoare-triple-def)

lemma *rule-Or*:

assumes $\models \{P\} C\ \{Q\}$
and $\models \{P'\} C\ \{Q'\}$
shows *hyper-hoare-triple* (disj P P') C (disj Q Q')
by (metis assms(1) assms(2) disj-def hyper-hoare-triple-def)

corollary *variant-if-rule*:

assumes *hyper-hoare-triple* P C1 Q
and *hyper-hoare-triple* P C2 Q
and *closed-by-union* Q
shows *hyper-hoare-triple* P (If C1 C2) Q
by (metis assms(1) assms(2) assms(3) if-rule join-closed-by-union)

Simplifying the rule

definition *stable-by-infinite-union* :: 'a hyperassertion \Rightarrow bool **where**
stable-by-infinite-union I $\longleftrightarrow (\forall F. (\forall S \in F. I\ S) \longrightarrow I\ (\bigcup S \in F. S))$

lemma *stable-by-infinite-unionE*:

assumes *stable-by-infinite-union* I
and $\bigwedge S. S \in F \Longrightarrow I\ S$
shows $I\ (\bigcup S \in F. S)$
using assms(1) assms(2) *stable-by-infinite-union-def* **by** blast

lemma *stable-by-union-and-constant-then-I*:

assumes $\bigwedge n. I\ n = I'$
and *stable-by-infinite-union* I'
shows *natural-partition* I = I'

proof (rule ext)

fix S **show** *natural-partition* I S = I' S

proof

show $I'\ S \Longrightarrow$ *natural-partition* I S

proof –

assume $I'\ S$

show *natural-partition* I S

proof (rule *natural-partitionI*)

show $S = \bigcup (\text{range } (\lambda n. S))$

by simp

fix n **show** $I\ n\ S$

by (simp add: $\langle I'\ S \rangle$ assms(1))

qed

qed

assume asm0: *natural-partition* I S

then obtain F **where** $S = (\bigcup n. F\ n) \bigwedge n. I\ n\ (F\ n)$

using *natural-partitionE* **by** blast

let ?F = $\{F\ n \mid n. \text{True}\}$

have $I' (\bigcup S \in ?F. S)$
using $assms(2)$
proof (*rule stable-by-infinite-unionE*[of I'])
fix S **assume** $S \in \{F\ n \mid n. True\}$
then show $I' S$
using $\langle \bigwedge n. I\ n\ (F\ n) \rangle assms(1)$ **by force**
qed
moreover have $(\bigcup S \in ?F. S) = S$
using $\langle S = (\bigcup n. F\ n) \rangle$ **by auto**
ultimately show $I' S$ **by blast**
qed
qed

corollary *simpler-rule-while*:
assumes *hyper-hoare-triple* $I\ C\ I$
and *stable-by-infinite-union* I
shows *hyper-hoare-triple* $I\ (While\ C)\ I$
proof –
let $?I = \lambda n. I$
have *hyper-hoare-triple* $(?I\ 0)\ (While\ C)\ (natural-partition\ ?I)$
using *while-rule*[of $?I\ C$]
by (*simp add: assms(1) assms(2) stable-by-union-and-constant-then-I*)
then show *?thesis*
by (*simp add: assms(2) stable-by-union-and-constant-then-I*)
qed

lemma *rule-and3*:
assumes $\models \{P1\}\ C\ \{Q1\}$
and $\models \{P2\}\ C\ \{Q2\}$
and $\models \{P3\}\ C\ \{Q3\}$
shows $\models \{conj\ P1\ (conj\ P2\ P3)\}\ C\ \{conj\ Q1\ (conj\ Q2\ Q3)\}$
by (*simp add: assms(1) assms(2) assms(3) rule-And*)

definition *not-empty where*
 $not_empty\ S \longleftrightarrow S \neq \{\}$

definition *finite-not-empty where*
 $finite_not_empty\ S \longleftrightarrow S \neq \{\} \wedge finite\ S$

definition *update-logical where*
 $update_logical\ \omega\ i\ v = ((fst\ \omega)(i := v), snd\ \omega)$

lemma *single-sem-prop*:
assumes *single-sem* $C\ (snd\ \omega)\ (snd\ \omega')$
and $fst\ \omega = fst\ \omega'$
shows $\models \{(\lambda S. \omega \in S)\}\ C\ \{(\lambda S. \omega' \in S)\}$
by (*metis assms(1) assms(2) hyper-hoare-tripleI in-sem prod.collapse*)

lemma *weaker-linking-rule*:

assumes $\bigwedge l \sigma \sigma'. \models \{ (\lambda S. (l, \sigma) \in S) \} C \{ (\lambda S. (l, \sigma') \in S) \} \implies (\models \{ P (l, \sigma) \} C \{ Q (l, \sigma') \})$
shows $\models \{ (\lambda S. \forall \omega \in S. P \omega S) \} C \{ (\lambda S. \forall \omega' \in S. Q \omega' S) \}$
by (*simp add: assms rule-linking-alt single-sem-prop*)

definition *general-union* :: 'a hyperassertion \Rightarrow 'a hyperassertion **where**
general-union $P S \longleftrightarrow (\exists F. S = \text{Union } F \wedge (\forall S' \in F. P S'))$

lemma *general-unionI*:

assumes $S = \text{Union } F$
and $\bigwedge S'. S' \in F \implies P S'$
shows *general-union* $P S$
using *assms(1) assms(2) general-union-def by blast*

lemma *general-unionE*:

assumes *general-union* $P S$
obtains F **where** $S = \text{Union } F \bigwedge S'. S' \in F \implies P S'$
by (*metis assms general-union-def*)

proposition *rule-BigUnion*:

fixes $P :: ((('a \Rightarrow 'b) \times ('c \Rightarrow 'd)) \text{ set} \Rightarrow \text{bool})$
assumes $\models \{ P \} C \{ Q \}$
shows $\models \{ \text{general-union } P \} C \{ \text{general-union } Q \}$
proof (*rule consequence-rule*)
define $PP :: ((('a \Rightarrow 'b) \times ('c \Rightarrow 'd)) \text{ set} \Rightarrow ((('a \Rightarrow 'b) \times ('c \Rightarrow 'd)) \text{ set} \Rightarrow \text{bool})$
where
 $PP = (\lambda-. P)$
let $?P = \text{disj } (\text{general-join } PP) (\lambda S. S = \{\})$
show *entails* (*general-union* P) $?P$
proof (*rule entailsI*)
fix S **assume** *general-union* $P S$
then obtain F **where** $S = \text{Union } F \bigwedge S'. S' \in F \implies P S'$
by (*metis general-union-def*)
have *general-join* $PP S \vee S = \{\}$
proof (*cases* $S = \{\}$)
case *True*
then show *?thesis*
by *simp*
next
case *False*
then obtain S' **where** $S' \in F$
using $\langle S = \bigcup F \rangle$ **by** *blast*
let $?F = \lambda SS. \text{if } SS \in F \text{ then } SS \text{ else } S'$
have *general-join* $PP S$

```

proof (rule general-joinI)
  fix  $x$  show  $PP\ x\ (?F\ x)$ 
    by (simp add: PP-def  $\langle S' \in F \rangle \langle \bigwedge S'. S' \in F \implies P\ S' \rangle$ )
  next
    show  $S = (\bigcup x. \text{if } x \in F \text{ then } x \text{ else } S')$ 
      using  $\langle S = \bigcup F \rangle \langle S' \in F \rangle$  by force
    qed
  then show ?thesis by simp
qed
then show disj (general-join PP)  $(\lambda S. S = \{\})\ S$ 
  by (simp add: disj-def)
qed

define  $QQ :: (('a \Rightarrow 'b) \times ('c \Rightarrow 'd))\ set \Rightarrow (('a \Rightarrow 'b) \times ('c \Rightarrow 'd))\ set \Rightarrow bool$ 
where
   $QQ = (\lambda -. Q)$ 
  let  $?Q = disj\ (general-join\ QQ)\ (\lambda S. S = \{\})$ 

show  $\models \{Logic.disj\ (general-join\ PP)\ (\lambda S. S = \{\})\}\ C\ \{?Q\}$ 
proof (rule rule-Or)
  show  $\models \{(\lambda S. S = \{\})\}\ C\ \{\lambda S. S = \{\}\}$ 
    by (metis (mono-tags, lifting) empty-iff equalsOI hyper-hoare-triple-def in-sem)

  show  $\models \{general-join\ PP\}\ C\ \{general-join\ QQ\}$ 
proof (rule rule-IndexedUnion)
  fix  $x$  show  $\models \{PP\ x\}\ C\ \{QQ\ x\}$ 
    by (simp add: PP-def QQ-def assms)
  qed
qed
show entails (Logic.disj (general-join QQ)  $(\lambda S. S = \{\})$ ) (general-union Q)
proof (rule entailsI)
  fix  $S$  assume Logic.disj (general-join QQ)  $(\lambda S. S = \{\})\ S$ 
  then show general-union Q  $S$ 
proof (cases  $S = \{\}$ )
  case True
    then show ?thesis
      using general-union-def by auto
  next
    case False
      then obtain  $F$  where  $\bigwedge x. QQ\ x\ (F\ x)\ S = \bigcup (range\ F)$ 
        by (metis (mono-tags, lifting)  $\langle Logic.disj\ (general-join\ QQ)\ (\lambda S. S = \{\})\ S \rangle$  disj-def general-join-def)
      then show ?thesis
        by (metis QQ-def general-unionI rangeE)
    qed
  qed
qed
qed

```

proposition *rule-Empty*:

$\models \{ (\lambda S. S = \{\}) \} C \{ (\lambda S. S = \{\}) \}$

proof (*rule consequence-rule*)

let $?P = \text{general-union } (\lambda-. \text{False})$

show *entails* $(\lambda S. S = \{\}) ?P$

by (*metis (full-types) Union-empty empty-iff entailsI general-unionI*)

show *entails* $?P (\lambda S. S = \{\})$

using *entailsI[of ?P λS. S = {}] general-union-def Sup-bot-conv(2) by metis*

show $\models \{ \text{general-union } (\lambda-. \text{False}) \} C \{ \text{general-union } (\lambda-. \text{False}) \}$

proof (*rule rule-BigUnion*)

show $\models \{ (\lambda-. \text{False}) \} C \{ \lambda-. \text{False} \}$

using *rule-False by auto*

qed

qed

definition *has-subset where*

$\text{has-subset } P S \longleftrightarrow (\exists S'. S' \subseteq S \wedge P S')$

lemma *has-subset-join-same*:

entails (*has-subset P*) (*join P* $(\lambda-. \text{True})$)

entails (*join P* $(\lambda-. \text{True})$) (*has-subset P*)

using *entailsI[of has-subset P join P (λ-. True)] has-subset-def join-def sup.absorb-iff2*

apply *metis*

using *UnCI entails-def[of join P (λ-. True) has-subset P] has-subset-def join-def*

subset-eq

by *metis*

proposition *rule-AtLeast*:

assumes $\models \{ P \} C \{ Q \}$

shows $\models \{ \text{has-subset } P \} C \{ \text{has-subset } Q \}$

proof (*rule consequence-rule*)

let $?P = \text{join } P (\lambda-. \text{True})$

let $?Q = \text{join } Q (\lambda-. \text{True})$

show $\models \{ ?P \} C \{ ?Q \}$

by (*simp add: assms rule-True rule-Union*)

qed (*auto simp add: has-subset-join-same*)

definition *has-superset where*

$\text{has-superset } P S \longleftrightarrow (\exists S'. S \subseteq S' \wedge P S')$

proposition *rule-AtMost*:

assumes $\models \{ P \} C \{ Q \}$

shows $\models \{ \text{has-superset } P \} C \{ \text{has-superset } Q \}$

proof (*rule hyper-hoare-tripleI*)

fix $S :: (('a \Rightarrow 'b) \times ('c \Rightarrow 'd)) \text{ set}$

assume *has-superset P S*

then obtain S' **where** $S \subseteq S' P S'$
 by (*meson has-superset-def*)
then have Q (*sem C S'*)
 using *assms hyper-hoare-tripleE* **by** *blast*
then show *has-superset Q (sem C S)*
 by (*meson ⟨S ⊆ S'⟩ has-superset-def sem-monotonic*)
qed

5.6 Synchronous Reasoning (Proposition 14, Appendix H).

theorem *if-sync-rule*:

assumes $\models \{P\} C1 \{P1\}$
and $\models \{P\} C2 \{P2\}$
and $\models \{combine\ from\ nat\ x\ P1\ P2\} C \{combine\ from\ nat\ x\ R1\ R2\}$
and $\models \{R1\} C1' \{Q1\}$
and $\models \{R2\} C2' \{Q2\}$

and *not-free-var-hyper x P1*
and *not-free-var-hyper x P2*
and *from-nat 1 ≠ from-nat 2*

and *not-free-var-hyper x R1*
and *not-free-var-hyper x R2*

shows $\models \{P\} If (Seq C1 (Seq C C1')) (Seq C2 (Seq C C2')) \{join Q1 Q2\}$

proof (*rule hyper-hoare-tripleI*)

fix S **assume** *asm0: P S*
have $r0$: *sem (stmt.If (Seq C1 (Seq C C1')) (Seq C2 (Seq C C2'))) S*
 $= sem C1' (sem C (sem C1 S)) \cup sem C2' (sem C (sem C2 S))$
by (*simp add: sem-if sem-seq*)
moreover have $P1 (sem C1 S) \wedge P2 (sem C2 S)$
using *asm0 assms(1) assms(2) hyper-hoare-tripleE* **by** *blast*

let $?S1 = (modify-lvar-to\ x\ (from-nat\ 1)) \text{' } (sem\ C1\ S)$

let $?S2 = (modify-lvar-to\ x\ (from-nat\ 2)) \text{' } (sem\ C2\ S)$

let $?f1 = Set.filter (\lambda\varphi. fst\ \varphi\ x = from-nat\ 1)$

let $?f2 = Set.filter (\lambda\varphi. fst\ \varphi\ x = from-nat\ 2)$

have $P1\ ?S1 \wedge P2\ ?S2$

by (*meson ⟨P1 (sem C1 S) ∧ P2 (sem C2 S)⟩ assms(6) assms(7) not-free-var-hyper-def*)

moreover have $rr1$: *Set.filter (\lambda\varphi. fst\ \varphi\ x = from-nat\ 1) (?S1 \cup ?S2) = ?S1*

using *injective-then-ok[of from-nat 1 from-nat 2 ?S1 x]*

by (*metis (no-types, lifting) assms(8)*)

moreover have $rr2$: *Set.filter (\lambda\varphi. fst\ \varphi\ x = from-nat\ 2) (?S1 \cup ?S2) = ?S2*

using *injective-then-ok[of from-nat 1 from-nat 2 ?S2 x]*

by (*metis (no-types, lifting) assms(8) sup-commute*)

ultimately have *combine from-nat x P1 P2 (?S1 \cup ?S2)*

by (*metis combineI*)

then have *combine from-nat x R1 R2 (sem C (?S1 \cup ?S2))*

```

    using assms(3) hyper-hoare-tripleE by blast
  moreover have ?f1 (sem C (?S1 ∪ ?S2)) = sem C ?S1
    using recover-after-sem[of from-nat 1 from-nat 2 ?S1 x ?S2] assms(8) rr1 rr2
    by simp blast
  then have R1 (sem C ?S1)
    by (metis (mono-tags) calculation combine-def)
  then have R1 (sem C (sem C1 S))
    by (metis assms(9) not-free-var-hyper-def sem-of-modify-lvar)
  moreover have ?f2 (sem C (?S1 ∪ ?S2)) = sem C ?S2
    using recover-after-sem[of from-nat 2 from-nat 1 ?S2 x ?S1] assms(8) rr1 rr2
    by (simp add: ac-simps) blast

  then have R2 (sem C ?S2)
    by (metis (mono-tags) calculation(1) combine-def)
  then have R2 (sem C (sem C2 S))
    by (metis assms(10) not-free-var-hyper-def sem-of-modify-lvar)

  then show join Q1 Q2 (sem (stmt.If (Seq C1 (Seq C C1')) (Seq C2 (Seq C
C2'))) S)
    by (metis (full-types) r0 assms(4) assms(5) calculation(2) hyper-hoare-tripleE
join-def)
qed

```

definition *update-lvar-set* **where**

$$\text{update-lvar-set } u \ e \ S = \{ ((fst \varphi')(u := e \varphi'), snd \varphi') \mid \varphi'. \varphi' \in S \}$$

lemma *equal-outside-set-helper*:

```

equal-outside-set {u} (fst \varphi) (fst ((fst \varphi)(u := x), snd \varphi))
by (simp add: equal-outside-set-def)

```

lemma *same-update-lvar-set*:

$$\text{same-mod-updates } \{u\} \ S \ (\text{update-lvar-set } u \ e \ S)$$

proof (rule *same-mod-updatesI*)

```

show  $\bigwedge \omega'. \omega' \in \text{update-lvar-set } u \ e \ S \implies \exists \omega \in S. \text{snd } \omega = \text{snd } \omega' \wedge \text{equal-outside-set } \{u\} \ (\text{fst } \omega') \ (\text{fst } \omega)$ 

```

proof –

```

fix  $\omega'$  assume  $\omega' \in \text{update-lvar-set } u \ e \ S$ 

```

```

then obtain  $\varphi'$  where  $\omega' = ((fst \varphi')(u := e \varphi'), snd \varphi') \ \varphi' \in S$ 

```

```

using mem-Collect-eq update-lvar-set-def[of u e S] by auto

```

```

then show  $\exists \omega \in S. \text{snd } \omega = \text{snd } \omega' \wedge \text{equal-outside-set } \{u\} \ (\text{fst } \omega') \ (\text{fst } \omega)$ 

```

```

by (meson equal-outside-set-helper equal-outside-sym snd-eqD)

```

qed

```

show  $\bigwedge \omega. \omega \in S \implies \exists \omega' \in \text{update-lvar-set } u \ e \ S. \text{snd } \omega = \text{snd } \omega' \wedge \text{equal-outside-set } \{u\} \ (\text{fst } \omega) \ (\text{fst } \omega')$ 

```

```

by (metis (mono-tags, lifting) equal-outside-set-helper mem-Collect-eq snd-conv
update-lvar-set-def)

```

qed

lemma *same-mod-updates-empty*:
assumes *same-mod-updates vars* $\{ \} S'$
shows $S' = \{ \}$
by (*meson assms equals0D equals0I same-mod-updates-def subset-mod-updatesE*)

definition *not-fv-hyper where*
not-fv-hyper $t A \longleftrightarrow (\forall S S'. \text{same-mod-updates } \{t\} S S' \wedge A S \longrightarrow A S')$

lemma *not-fv-hyperE*:
assumes *not-fv-hyper* $e I$
and *same-mod-updates* $\{e\} S S'$
and $I S$
shows $I S'$
by (*meson assms(1) assms(2) assms(3) not-fv-hyper-def*)

definition *assign-exp-to-lvar where*
assign-exp-to-lvar $e l \varphi = ((fst \varphi)(l := e (snd \varphi)), snd \varphi)$

definition *assign-exp-to-lvar-set where*
assign-exp-to-lvar-set $e l S = \text{assign-exp-to-lvar } e l ' S$

lemma *same-outside-set-lvar-assign-exp*:
 $snd \varphi = snd (\text{assign-exp-to-lvar } e l \varphi) \wedge \text{equal-outside-set } \{l\} (fst \varphi) (fst (\text{assign-exp-to-lvar } e l \varphi))$
by (*simp add: assign-exp-to-lvar-def equal-outside-set-def*)

lemma *assign-exp-to-lvar-set-same-mod-updates*:
 $\text{same-mod-updates } \{l\} S (\text{assign-exp-to-lvar-set } e l S)$

proof (*rule same-mod-updatesI*)
show $\bigwedge \omega. \omega \in S \implies \exists \omega' \in \text{assign-exp-to-lvar-set } e l S. snd \omega = snd \omega' \wedge \text{equal-outside-set } \{l\} (fst \omega) (fst \omega')$
by (*metis assign-exp-to-lvar-set-def rev-image-eqI same-outside-set-lvar-assign-exp*)
show $\bigwedge \omega'. \omega' \in \text{assign-exp-to-lvar-set } e l S \implies \exists \omega \in S. snd \omega = snd \omega' \wedge \text{equal-outside-set } \{l\} (fst \omega') (fst \omega)$
by (*metis (mono-tags, opaque-lifting) assign-exp-to-lvar-set-def equal-outside-sym imageE same-outside-set-lvar-assign-exp*)
qed

lemma *holds-forall-same-mod-updates*:
assumes *same-mod-updates vars* $S S'$
and *holds-forall* $b S$
shows *holds-forall* $b S'$
proof (*rule holds-forallI*)
fix φ' **assume** *asm0*: $\varphi' \in S'$
then obtain φ **where** $\varphi \in S \text{ and } \varphi = snd \varphi'$
by (*metis assms(1) same-mod-updates-def subset-mod-updatesE*)

then show $b \text{ (snd } \varphi')$
by (*metis* *assms(2)* *holds-forall-def*)
qed

lemma *not-fv-hyper-assign-exp*:
assumes *not-fv-hyper* $t \ A$
shows $A \ S \longleftrightarrow A \text{ (assign-exp-to-lvar-set } e \ t \ S)$
by (*metis* *assign-exp-to-lvar-set-same-mod-updates* *assms* *not-fv-hyper-def* *same-mod-updates-sym*)

lemma *holds-forall-same-assign-lvar*:
holds-forall $b \ S \longleftrightarrow \text{holds-forall } b \text{ (assign-exp-to-lvar-set } e \ l \ S) \text{ (is } ?A \longleftrightarrow ?B)$
proof
show $?A \implies ?B$
by (*simp* *add*: *assign-exp-to-lvar-def* *assign-exp-to-lvar-set-def* *holds-forall-def*)
show $?B \implies ?A$
by (*simp* *add*: *assign-exp-to-lvar-def* *assign-exp-to-lvar-set-def* *holds-forall-def*)
qed

definition *e-recorded-in-t* **where**
e-recorded-in-t $e \ t \ S \longleftrightarrow (\forall \varphi \in S. \text{fst } \varphi \ t = e \text{ (snd } \varphi))$

lemma *e-recorded-in-tI*:
assumes $\bigwedge \varphi. \varphi \in S \implies \text{fst } \varphi \ t = e \text{ (snd } \varphi)$
shows *e-recorded-in-t* $e \ t \ S$
by (*simp* *add*: *assms* *e-recorded-in-t-def*)

definition *e-smaller-than-t* **where**
e-smaller-than-t $e \ t \ lt \ S \longleftrightarrow (\forall \varphi \in S. \text{lt } (e \text{ (snd } \varphi)) \text{ (fst } \varphi \ t))$

lemma *low-expI*:
assumes $\bigwedge \varphi \varphi'. \varphi \in S \wedge \varphi' \in S \implies (e \text{ (snd } \varphi) = e \text{ (snd } \varphi'))$
shows *low-exp* $e \ S$
using *low-exp-def* *assms* **by** *blast*

lemma *low-exp-forall-same-mod-updates*:
assumes *same-mod-updates* *vars* $S \ S'$
and *low-exp* $b \ S$
shows *low-exp* $b \ S'$
proof (*rule* *low-expI*)
fix $\varphi' \varphi''$ **assume** *asm0*: $\varphi' \in S' \wedge \varphi'' \in S'$
then obtain φ **where** $\varphi \in S \text{ snd } \varphi = \text{snd } \varphi'$
by (*metis* *assms(1)* *same-mod-updates-def* *subset-mod-updatesE*)
then show $b \text{ (snd } \varphi') = b \text{ (snd } \varphi'')$
using *asm0* *assms(1)* *assms(2)* *low-exp-def[of b S]* *same-mod-updates-def[of vars S S']* *subset-mod-updatesE*
by *metis*
qed

```

lemma e-recorded-in-t-if-assigned:
  e-recorded-in-t e t (assign-exp-to-lvar-set e t S)
  by (simp add: assign-exp-to-lvar-def assign-exp-to-lvar-set-def e-recorded-in-t-def)

lemma low-exp-commute-assign-lvar:
  low-exp b (assign-exp-to-lvar-set e t S)  $\longleftrightarrow$  low-exp b S (is ?A  $\longleftrightarrow$  ?B)
proof
  assume ?A
  then show ?B
    by (meson assign-exp-to-lvar-set-same-mod-updates low-exp-forall-same-mod-updates
same-mod-updates-sym)
  next
  assume ?B
  then show ?A
    by (meson assign-exp-to-lvar-set-same-mod-updates low-exp-forall-same-mod-updates)
qed

end

```

6 Syntactic Assertions

```

theory SyntacticAssertions
  imports Logic Loops ProgramHyperproperties Compositionality
begin

```

6.1 Preliminaries: Types, expressions, 'a assertions

```

type-synonym var = nat
type-synonym qstate = nat
type-synonym qvar = nat

type-synonym 'a nstate = (var, 'a, var, 'a) state
type-synonym 'a npstate = (var, 'a) pstate

type-synonym 'a binop = 'a  $\Rightarrow$  'a  $\Rightarrow$  'a
type-synonym 'a comp = 'a  $\Rightarrow$  'a  $\Rightarrow$  bool

```

Quantified variables and quantified states are represented as de Bruijn indices (natural numbers).

```

datatype 'a exp =
  | EPVar qstate var —  $\varphi^P(x)$ : Program variable
  | ELVar qstate var —  $\varphi^L(x)$ : Logical variable
  | EQVar qvar —  $y$ : Quantified variable
  | EConst 'a
  | EBinop 'a exp 'a binop 'a exp —  $e \oplus e$ 
  | EFun 'a  $\Rightarrow$  'a 'a exp —  $f(e)$ 

```

Quantified variables and quantified states are represented as de Bruijn indices (natural numbers). Thus, quantifiers do not have a name for the

variable or state they quantify over.

datatype 'a assertion =
 AConst bool
 | AComp 'a exp 'a comp 'a exp — $e \succeq e$
 | AForallState 'a assertion — $\forall \langle \varphi \rangle. A$
 | AExistsState 'a assertion — $\exists \langle \varphi \rangle. A$
 | AForall 'a assertion — $\forall y. A$
 | AExists 'a assertion — $\exists y. A$
 | AOr 'a assertion 'a assertion — $A \vee A$
 | AAnd 'a assertion 'a assertion — $A \wedge A$

We use a list of values and a list of states to track quantified values and states, respectively.

fun interp-exp :: 'a list \Rightarrow 'a nstate list \Rightarrow 'a exp \Rightarrow 'a **where**
 interp-exp vals states (EPCmp e1 e2) = e1 \succeq e2
 | interp-exp vals states (EPVar st x) = snd (states ! st) x
 | interp-exp vals states (ELVar st x) = fst (states ! st) x
 | interp-exp vals states (EQVar x) = vals ! x
 | interp-exp vals states (EConst v) = v
 | interp-exp vals states (EBinop e1 op e2) = op (interp-exp vals states e1) (interp-exp vals states e2)
 | interp-exp vals states (EFun f e) = f (interp-exp vals states e)

fun sat-assertion :: 'a list \Rightarrow 'a nstate list \Rightarrow 'a assertion \Rightarrow 'a nstate set \Rightarrow bool **where**
 sat-assertion vals states (AConst b) = b
 | sat-assertion vals states (AComp e1 cmp e2) = cmp (interp-exp vals states e1) (interp-exp vals states e2)
 | sat-assertion vals states (AForallState A) S = $\forall \varphi \in S. \text{sat-assertion vals } (\varphi \# \text{states}) A S$
 | sat-assertion vals states (AExistsState A) S = $\exists \varphi \in S. \text{sat-assertion vals } (\varphi \# \text{states}) A S$
 | sat-assertion vals states (AForall A) S = $\forall v. \text{sat-assertion } (v \# \text{vals}) \text{states } A S$
 | sat-assertion vals states (AExists A) S = $\exists v. \text{sat-assertion } (v \# \text{vals}) \text{states } A S$
 | sat-assertion vals states (AAnd A B) S = $(\text{sat-assertion vals states } A S \wedge \text{sat-assertion vals states } B S)$
 | sat-assertion vals states (AOr A B) S = $(\text{sat-assertion vals states } A S \vee \text{sat-assertion vals states } B S)$

Negation and implication are defined on top of this base language.

definition neg-cmp :: 'a comp \Rightarrow 'a comp **where**
 neg-cmp cmp v1 v2 = $\neg (\text{cmp } v1 v2)$

fun ANot **where**
 ANot (AConst b) = AConst ($\neg b$)
 | ANot (AComp e1 cmp e2) = AComp e1 (neg-cmp cmp) e2
 | ANot (AForallState A) = AExistsState (ANot A)

| $ANot (AExistsState A) = AForallState (ANot A)$
| $ANot (AOr A B) = AAnd (ANot A) (ANot B)$
| $ANot (AAnd A B) = AOr (ANot A) (ANot B)$
| $ANot (AForall A) = AExists (ANot A)$
| $ANot (AExists A) = AForall (ANot A)$

definition $AImp$ where

$AImp A B = AOr (ANot A) B$

lemma sat -assertion-Not:

sat -assertion vals states $(ANot A) S \longleftrightarrow \neg(sat$ -assertion vals states $A S)$
by (induct A arbitrary: vals states) (simp-all add: neg-cmp-def)

lemma sat -assertion- Imp :

sat -assertion vals states $(AImp A B) S \longleftrightarrow (sat$ -assertion vals states $A S \longrightarrow sat$ -assertion vals states $B S)$
by (simp add: $AImp$ -def sat -assertion-Not)

abbreviation $interp$ -assert where $interp$ -assert $\equiv sat$ -assertion $\square \square$

6.2 Assume rule

fun $transform$ -assume :: 'a assertion \Rightarrow 'a assertion \Rightarrow 'a assertion where

$transform$ -assume - $(AConst b) = AConst b$
| $transform$ -assume - $(AComp e1 cmp e2) = AComp e1 cmp e2$
| $transform$ -assume $b (AForallState A) = AForallState (AImp b (transform$ -assume $b A))$
| $transform$ -assume $b (AExistsState A) = AExistsState (AAnd b (transform$ -assume $b A))$
| $transform$ -assume $b (AForall A) = AForall (transform$ -assume $b A)$
| $transform$ -assume $b (AExists A) = AExists (transform$ -assume $b A)$
| $transform$ -assume $b (AAnd A B) = AAnd (transform$ -assume $b A) (transform$ -assume $b B)$
| $transform$ -assume $b (AOr A B) = AOr (transform$ -assume $b A) (transform$ -assume $b B)$

definition $same$ -syn-sem :: 'a assertion \Rightarrow ('a npstate \Rightarrow bool) \Rightarrow bool

where

$same$ -syn-sem $bsyn bsem \longleftrightarrow$
 $(\forall$ states vals $S. length$ states $> 0 \longrightarrow bsem (snd (hd$ states)) = sat -assertion vals states $bsyn S)$

lemma $same$ -syn-sem I :

assumes \bigwedge states vals $S. length$ states $> 0 \implies bsem (snd (hd$ states)) $\longleftrightarrow sat$ -assertion vals states $bsyn S$
shows $same$ -syn-sem $bsyn bsem$
by (simp add: $assms$ $same$ -syn-sem-def)

```

lemma transform-assume-valid:
  assumes same-syn-sem bsyn bsem
  shows sat-assertion vals states A (Set.filter (bsem o snd) S)
   $\longleftrightarrow$  sat-assertion vals states (transform-assume bsyn A) S
proof (induct A arbitrary: vals states)
  case (AForallState A)
  let ?S = Set.filter (bsem o snd) S
  let ?A = transform-assume bsyn A
  have sat-assertion vals states (AForallState A) ?S  $\longleftrightarrow$  ( $\forall \varphi \in ?S$ . sat-assertion
  vals ( $\varphi \#$  states) A ?S)
    by force
  also have ...  $\longleftrightarrow$  ( $\forall \varphi \in ?S$ . sat-assertion vals ( $\varphi \#$  states) ?A S)
    using AForallState by presburger
  also have ...  $\longleftrightarrow$  ( $\forall \varphi \in S$ . bsem (snd  $\varphi$ )  $\longrightarrow$  sat-assertion vals ( $\varphi \#$  states) ?A
  S)
    by fastforce
  also have ...  $\longleftrightarrow$  ( $\forall \varphi \in S$ . sat-assertion vals ( $\varphi \#$  states) bsyn S  $\longrightarrow$  sat-assertion
  vals ( $\varphi \#$  states) ?A S)
    using assms same-syn-sem-def[of bsyn bsem] by auto
  also have ...  $\longleftrightarrow$  ( $\forall \varphi \in S$ . sat-assertion vals ( $\varphi \#$  states) (AImp bsyn ?A) S)
    using sat-assertion-Imp by fast
  also have ...  $\longleftrightarrow$  sat-assertion vals states (AForallState (AImp bsyn ?A)) S
    using sat-assertion.simps(2) by force
  finally show ?case
    using transform-assume.simps(1) by fastforce
next
  case (AExistsState A)
  let ?S = Set.filter (bsem o snd) S
  let ?A = transform-assume bsyn A
  have sat-assertion vals states (AExistsState A) ?S  $\longleftrightarrow$  ( $\exists \varphi \in ?S$ . sat-assertion
  vals ( $\varphi \#$  states) A ?S)
    by force
  also have ...  $\longleftrightarrow$  ( $\exists \varphi \in ?S$ . sat-assertion vals ( $\varphi \#$  states) ?A S)
    using AExistsState by presburger
  also have ...  $\longleftrightarrow$  ( $\exists \varphi \in S$ . bsem (snd  $\varphi$ )  $\wedge$  sat-assertion vals ( $\varphi \#$  states) ?A S)
    by force
  also have ...  $\longleftrightarrow$  ( $\exists \varphi \in S$ . sat-assertion vals ( $\varphi \#$  states) bsyn S  $\wedge$  sat-assertion
  vals ( $\varphi \#$  states) ?A S)
    using assms same-syn-sem-def[of bsyn bsem] by auto
  also have ...  $\longleftrightarrow$  ( $\exists \varphi \in S$ . sat-assertion vals ( $\varphi \#$  states) (AAnd bsyn ?A) S)
    by simp
  also have ...  $\longleftrightarrow$  sat-assertion vals states (AExistsState (AAnd bsyn ?A)) S
    using sat-assertion.simps(3) by force
  then show ?case
    using calculation transform-assume.simps(2) by fastforce
next
  case (AForall A)
  let ?S = Set.filter (bsem o snd) S
  let ?A = transform-assume bsyn A

```

```

have sat-assertion vals states (AForall A) ?S  $\longleftrightarrow$   $(\forall v. \text{sat-assertion } (v \# \text{vals})$ 
states A ?S)
  by force
also have ...  $\longleftrightarrow$   $(\forall v. \text{sat-assertion } (v \# \text{vals})$  states ?A S)
  using AForall by presburger
also have ...  $\longleftrightarrow$  sat-assertion vals states (AForall ?A) S
  by simp
then show ?case
  using calculation transform-assume.simps(3) by fastforce
next
case (AExists A)
let ?S = Set.filter (bsem  $\circ$  snd) S
let ?A = transform-assume bsyn A
have sat-assertion vals states (AExists A) ?S  $\longleftrightarrow$   $(\exists v. \text{sat-assertion } (v \# \text{vals})$ 
states A ?S)
  by force
also have ...  $\longleftrightarrow$   $(\exists v. \text{sat-assertion } (v \# \text{vals})$  states ?A S)
  using AExists by presburger
also have ...  $\longleftrightarrow$  sat-assertion vals states (AExists ?A) S
  by simp
then show ?case
  using calculation transform-assume.simps(4) by fastforce
qed (simp-all)

```

6.2.1 Program expressions (values)

datatype *'a pexp =*

```

PVar var — Normal variable, like x
| PConst 'a
| PBinop 'a pexp 'a binop 'a pexp
| PFun 'a  $\Rightarrow$  'a 'a pexp

```

fun *interp-pexp :: 'a pexp \Rightarrow 'a npstate \Rightarrow 'a*

where

```

interp-pexp (PVar x)  $\varphi$  =  $\varphi$  x
| interp-pexp (PConst n) - = n
| interp-pexp (PBinop p1 op p2)  $\varphi$  = op (interp-pexp p1  $\varphi$ ) (interp-pexp p2  $\varphi$ )
| interp-pexp (PFun f p)  $\varphi$  = f (interp-pexp p  $\varphi$ )

```

fun *pexp-to-exp where*

```

pexp-to-exp st (PVar x) = EVar st x
| pexp-to-exp - (PConst n) = EConst n
| pexp-to-exp st (PBinop p1 op p2) = EBinop (pexp-to-exp st p1) op (pexp-to-exp st p2)
| pexp-to-exp st (PFun f p) = EFun f (pexp-to-exp st p)

```

lemma *same-syn-sem-exp:*

```

interp-pexp p (snd (states ! st)) = interp-exp vals states (pexp-to-exp st p)

```

```

proof (induct p)
  case (PVar x)
  then show ?case
    using hd-conv-nth by force
qed (simp-all)

```

6.2.2 Program expressions (booleans)

```

datatype 'a pbexp =
  PBConst bool
  | PBAnd 'a pbexp 'a pbexp
  | PBOr 'a pbexp 'a pbexp
  | PBComp 'a pexp 'a comp 'a pexp

```

```

fun interp-pbexp :: 'a pbexp ⇒ 'a nstate ⇒ bool
  where
    interp-pbexp (PBConst b) - ↔ b
  | interp-pbexp (PBAnd pb1 pb2) φ ↔ interp-pbexp pb1 φ ∧ interp-pbexp pb2 φ
  | interp-pbexp (PBOr pb1 pb2) φ ↔ interp-pbexp pb1 φ ∨ interp-pbexp pb2 φ
  | interp-pbexp (PBComp p1 cmp p2) φ ↔ cmp (interp-pexp p1 φ) (interp-pexp
p2 φ)

```

```

fun pbexp-to-assertion where
  pbexp-to-assertion (PBConst b) = AConst b
  | pbexp-to-assertion st (PBAnd pb1 pb2) = AAnd (pbexp-to-assertion st pb1) (pbexp-to-assertion
st pb2)
  | pbexp-to-assertion st (PBOr pb1 pb2) = AOr (pbexp-to-assertion st pb1) (pbexp-to-assertion
st pb2)
  | pbexp-to-assertion st (PBComp p1 cmp p2) = AComp (pexp-to-exp st p1) cmp
(pexp-to-exp st p2)

```

lemma same-syn-sem-assertion:

```

  interp-pbexp pb (snd (states ! st)) = sat-assertion vals states (pbexp-to-assertion
st pb) S

```

proof (induct pb)

case (PB_{Comp} x1 x2 x3)

then show ?case

```

  by (metis interp-pbexp.simps(4) pbexp-to-assertion.simps(4) same-syn-sem-exp
sat-assertion.simps(2))

```

qed (simp-all)

lemma pe_{xp}-to-exp-same:

```

  shows same-syn-sem (pbexp-to-assertion 0 pb) (interp-pbexp pb)

```

proof (rule same-syn-semI)

fix states :: 'a nstate list

fix vals S

assume 0 < length states

```

  then have sat-assertion vals states (pbexp-to-assertion 0 pb) S = sat-assertion

```

```

[] states (pbexp-to-assertion 0 pb) S

```

```

using same-syn-sem-assertion by blast
then show interp-pbexp pb (snd (hd states)) = sat-assertion vals states (pbexp-to-assertion
0 pb) S
by (metis ‹0 < length states› hd-conv-nth length-greater-0-conv same-syn-sem-assertion)
qed

```

6.2.3 Syntactic rule for assume

```

theorem rule-assume-syntactic-general:
  ⊨ { sat-assertion states vals (transform-assume (pbexp-to-assertion 0 pb) P) }
  Assume (interp-pbexp pb) {sat-assertion states vals P}
proof (rule hyper-hoare-tripleI)
  fix S assume asm0: sat-assertion states vals (transform-assume (pbexp-to-assertion
0 pb) P) S
  then have sat-assertion states vals P (Set.filter (interp-pbexp pb ∘ snd) S)
  using pbexp-to-exp-same transform-assume-valid by blast
  then show sat-assertion states vals P (sem (Assume (interp-pbexp pb)) S)
  by (simp add: assume-sem)
qed

```

```

theorem rule-assume-syntactic:
  ⊨ { interp-assert (transform-assume (pbexp-to-assertion 0 pb) P) } Assume
(interp-pbexp pb) {interp-assert P}
  by (simp add: rule-assume-syntactic-general)

```

6.3 Havoc rule

6.3.1 Shifting variables

```

fun insert-at where
  insert-at 0 x l = x # l
| insert-at (Suc n) x (t # q) = t # (insert-at n x q)
| insert-at (Suc n) x [] = [x]

```

```

lemma length-insert-at:
  length (insert-at n x l) = length l + 1
proof (induct n arbitrary: l)
  case (Suc n)
  then show ?case
  proof (cases l)
  case (Cons t q)
  then show ?thesis
  by (simp add: Suc)
  qed (simp)
qed (simp-all)

```

```

lemma insert-at-charact-1:
  n ≤ length l ⇒ k < n ⇒ (insert-at n x l) ! k = l ! k
proof (induct k arbitrary: l n)

```

```

case 0
then show ?case
  by (metis bot-nat-0.not-eq-extremum insert-at.elims le-zero-eq list.size(3) nth-Cons-0)
next
  case (Suc k)
  then obtain nn t q where n = Suc nn l = t # q
    by (metis Suc-less-eq2 le-antisym list.exhaust list.size(3) not-less-zero zero-le)
  then show ?case
    using Suc.hyps Suc.prem1 Suc.prem2 by force
qed

```

```

lemma insert-at-charact-2:
  n ≤ length l ⇒ (insert-at n x l) ! n = x
proof (induct n arbitrary: l)
  case (Suc n)
  then show ?case
    by (metis Suc-le-length-iff insert-at.simps(2) nth-Cons-Suc)
qed (simp)

```

```

lemma insert-at-charact-3:
  n ≤ length l ⇒ k ≥ n ⇒ (insert-at n x l) ! (Suc k) = l ! k
proof (induct n arbitrary: l k)
  case (Suc xa)
  then obtain t q kk where k = Suc kk l = t # q
    by (meson Suc-le-D Suc-le-length-iff)
  then show ?case
    using Suc.hyps Suc.prem1 Suc.prem2 by auto
qed (simp)

```

```

fun shift-vars-exp where
  shift-vars-exp n (EQVar x) = (if x ≥ n then EQVar (Suc x) else EQVar x)
| shift-vars-exp n (EBinop e1 op e2) = EBinop (shift-vars-exp n e1) op (shift-vars-exp n e2)
| shift-vars-exp n (EFun p e) = EFun p (shift-vars-exp n e)
| shift-vars-exp - e = e

```

```

fun shift-states-exp where
  shift-states-exp n (EPVar φ x) = (if φ ≥ n then EPVar (Suc φ) x else EPVar φ x)
| shift-states-exp n (ELVar φ x) = (if φ ≥ n then ELVar (Suc φ) x else ELVar φ x)
| shift-states-exp n (EBinop e1 op e2) = EBinop (shift-states-exp n e1) op (shift-states-exp n e2)
| shift-states-exp n (EFun p e) = EFun p (shift-states-exp n e)
| shift-states-exp - e = e

```

```

fun wf-exp :: nat ⇒ nat ⇒ 'a exp ⇒ bool where

```

$wf\text{-exp } nv \ ns \ (EPVar \ st \ -) \longleftrightarrow st < ns$
 $wf\text{-exp } nv \ ns \ (ELVar \ st \ -) \longleftrightarrow st < ns$
 $wf\text{-exp } nv \ ns \ (EQVar \ x) \longleftrightarrow x < nv$
 $wf\text{-exp } nv \ ns \ (EBinop \ e1 \ - \ e2) \longleftrightarrow wf\text{-exp } nv \ ns \ e1 \wedge wf\text{-exp } nv \ ns \ e2$
 $wf\text{-exp } nv \ ns \ (EFun \ f \ e) \longleftrightarrow wf\text{-exp } nv \ ns \ e$
 $wf\text{-exp } nv \ ns \ (EConst \ -) \longleftrightarrow True$

lemma *wf-shift-vars-exp*:
assumes *wf-exp nv ns e*
shows *wf-exp (Suc nv) ns (shift-vars-exp n e)*
using *assms*
by (*induct e*) *simp-all*

lemma *wf-shift-states-exp*:
assumes *wf-exp nv ns e*
shows *wf-exp nv (Suc ns) (shift-states-exp n e)*
using *assms*
by (*induct e*) *simp-all*

lemma *shift-vars-exp-charact*:
assumes $n \leq \text{length } vals$
shows $\text{interp-exp } vals \ \text{states } e = \text{interp-exp } (\text{insert-at } n \ v \ vals) \ \text{states } (\text{shift-vars-exp } n \ e)$
using *assms*
proof (*induct e*)
case (*EQVar x*)
then show *?case*
by (*simp add: insert-at-charact-1 insert-at-charact-3*)
qed (*simp-all*)

lemma *shift-states-exp-charact*:
assumes $n \leq \text{length } states$
shows $\text{interp-exp } vals \ \text{states } e = \text{interp-exp } vals \ (\text{insert-at } n \ \varphi \ states) \ (\text{shift-states-exp } n \ e)$
using *assms*
proof (*induct e*)
case (*EPVar x1 x2*)
then show *?case*
by (*simp add: insert-at-charact-1 insert-at-charact-3*)
next
case (*ELVar x1 x2*)
then show *?case*
by (*simp add: insert-at-charact-1 insert-at-charact-3*)
qed (*simp-all*)

fun *shift-vars* **where**
 $\text{shift-vars } n \ (AConst \ b) = AConst \ b$
 $\text{shift-vars } n \ (AComp \ e1 \ \text{cmp} \ e2) = AComp \ (\text{shift-vars-exp } n \ e1) \ \text{cmp} \ (\text{shift-vars-exp } n \ e2)$

$n \ e2)$
 $| \text{shift-vars } n \ (AForall \ A) = AForall \ (\text{shift-vars } (Suc \ n) \ A)$
 $| \text{shift-vars } n \ (AExists \ A) = AExists \ (\text{shift-vars } (Suc \ n) \ A)$
 $| \text{shift-vars } n \ (AForallState \ A) = AForallState \ (\text{shift-vars } n \ A)$
 $| \text{shift-vars } n \ (AExistsState \ A) = AExistsState \ (\text{shift-vars } n \ A)$
 $| \text{shift-vars } n \ (AOr \ A \ B) = AOr \ (\text{shift-vars } n \ A) \ (\text{shift-vars } n \ B)$
 $| \text{shift-vars } n \ (AAnd \ A \ B) = AAnd \ (\text{shift-vars } n \ A) \ (\text{shift-vars } n \ B)$

lemma *shift-vars-charact*:

assumes $n \leq \text{length } \text{vals}$

shows $\text{sat-assertion } \text{vals } \text{states } A \ S \longleftrightarrow \text{sat-assertion } (\text{insert-at } n \ x \ \text{vals}) \ \text{states } (\text{shift-vars } n \ A) \ S$

using *assms*

proof (*induct A arbitrary: vals states n*)

case (*AComp x1a x2 x3a*)

then show *?case*

using *shift-vars-exp-charact* **by** *fastforce*

next

case (*AForall A*)

have $\text{sat-assertion } \text{vals } \text{states } (AForall \ A) \ S \longleftrightarrow (\forall v. \text{sat-assertion } (v \ \# \ \text{vals}) \ \text{states } A \ S)$

by *simp*

also have $\dots \longleftrightarrow (\forall v. \text{sat-assertion } (\text{insert-at } (Suc \ n) \ x \ (v \ \# \ \text{vals})) \ \text{states } (\text{shift-vars } (Suc \ n) \ A) \ S)$

using *AForall(2) AForall(1)[of Suc n - states]* **by** *simp*

also have $\dots \longleftrightarrow (\forall v. \text{sat-assertion } (v \ \# \ \text{insert-at } n \ x \ \text{vals}) \ \text{states } (\text{shift-vars } (Suc \ n) \ A) \ S)$

by *simp*

also have $\dots \longleftrightarrow \text{sat-assertion } (\text{insert-at } n \ x \ \text{vals}) \ \text{states } (AForall \ (\text{shift-vars } (Suc \ n) \ A)) \ S$

by *simp*

then show $\text{sat-assertion } \text{vals } \text{states } (AForall \ A) \ S = \text{sat-assertion } (\text{insert-at } n \ x \ \text{vals}) \ \text{states } (\text{shift-vars } n \ (AForall \ A)) \ S$

using *calculation* **by** *simp*

next

case (*AExists A*)

have $\text{sat-assertion } \text{vals } \text{states } (AExists \ A) \ S \longleftrightarrow (\exists v. \text{sat-assertion } (v \ \# \ \text{vals}) \ \text{states } A \ S)$

by *simp*

also have $\dots \longleftrightarrow (\exists v. \text{sat-assertion } (\text{insert-at } (Suc \ n) \ x \ (v \ \# \ \text{vals})) \ \text{states } (\text{shift-vars } (Suc \ n) \ A) \ S)$

using *AExists(2) AExists(1)[of Suc n - states]* **by** *simp*

also have $\dots \longleftrightarrow (\exists v. \text{sat-assertion } (v \ \# \ \text{insert-at } n \ x \ \text{vals}) \ \text{states } (\text{shift-vars } (Suc \ n) \ A) \ S)$

by *simp*

also have $\dots \longleftrightarrow \text{sat-assertion } (\text{insert-at } n \ x \ \text{vals}) \ \text{states } (AExists \ (\text{shift-vars } (Suc \ n) \ A)) \ S$

by *simp*

then show $\text{sat-assertion } \text{vals } \text{states } (AExists \ A) \ S = \text{sat-assertion } (\text{insert-at } n$

```

x vals) states (shift-vars n (AExists A)) S
  using calculation by simp
qed (simp-all)

```

6.3.2 Expressions (Boolean and values)

definition *update-state* **where**

```

update-state  $\varphi$  x v = (fst  $\varphi$ , (snd  $\varphi$ )(x := v))

```

fun *subst-exp-single* :: *qstate* \Rightarrow *var* \Rightarrow '*a* *exp* \Rightarrow '*a* *exp* \Rightarrow '*a* *exp* **where**

```

subst-exp-single  $\varphi$  x e' (EPVar st y) = (if  $\varphi = st \wedge x = y$  then e' else EPVar st
y)
| subst-exp-single  $\varphi$  x e' (EBinop e1 bop e2) = EBinop (subst-exp-single  $\varphi$  x e' e1)
bop (subst-exp-single  $\varphi$  x e' e2)
| subst-exp-single  $\varphi$  x e' (EFun f e) = EFun f (subst-exp-single  $\varphi$  x e' e)
| subst-exp-single - - - e = e

```

lemma *wf-subst-exp*:

```

assumes wf-exp nv ns e
  and wf-exp nv ns e'
  shows wf-exp nv ns (subst-exp-single  $\varphi$  x e' e)
using assms
by (induct e) simp-all

```

lemma *subst-exp-single-charact*:

```

assumes interp-exp vals states e' = snd (states ! st) x
  shows interp-exp vals states (subst-exp-single st x e' e) = interp-exp vals states e
using assms
by (induct e) simp-all

```

definition *subst-state* **where**

```

subst-state x pe  $\varphi$  = (fst  $\varphi$ , (snd  $\varphi$ )(x := interp-pe xp pe (snd  $\varphi$ )))

```

definition *update-state-at* **where**

```

update-state-at states n x v = list-update states n (update-state (states ! n) x v)

```

lemma *update-state-at-fst*:

```

fst (update-state-at states n x v ! st) = fst (states ! st)

```

proof (cases n = st)

case *True*

then show *?thesis*

```

  by (metis fst-conv linorder-not-less list-update-beyond nth-list-update-eq up-
date-state-at-def update-state-def)

```

```

next
  case False
  then show ?thesis
    by (simp add: update-state-at-def)
qed

lemma update-state-at-snd-1:
   $x \neq y \implies \text{snd} (\text{update-state-at states } n \ x \ v \ ! \ st) \ y = \text{snd} (\text{states} \ ! \ st) \ y$ 
  apply (cases  $n = st$ )
  apply (metis fun-upd-other linorder-not-less list-update-beyond nth-list-update-eq
snd-conv update-state-at-def update-state-def)
  by (simp add: update-state-at-def)

lemma update-state-at-snd-2:
   $st \neq n \implies \text{snd} (\text{update-state-at states } n \ x \ v \ ! \ st) \ y = \text{snd} (\text{states} \ ! \ st) \ y$ 
  by (simp add: update-state-at-def)

lemma update-state-at-snd-3:
  assumes  $n < \text{length states}$ 
  shows  $\text{snd} (\text{update-state-at states } n \ x \ v \ ! \ n) \ x = v$ 
  by (simp add: assms update-state-at-def update-state-def)

lemma subst-exp-more-complex-charact:
  assumes  $\text{states}' = \text{update-state-at states } st \ x \ (\text{interp-exp vals states } e')$ 
  and  $st < \text{length states}$ 
  shows  $\text{interp-exp vals states} (\text{subst-exp-single } st \ x \ e' \ e) = \text{interp-exp vals states}' \ e$ 
  using assms
proof (induct  $e$ )
  case (EPVar  $\varphi \ y$ )
  then show ?case
    by (metis interp-exp.simps(1) subst-exp-single.simps(1) update-state-at-snd-1
update-state-at-snd-2 update-state-at-snd-3)
next
  case (ELVar  $x1 \ x2$ )
  then show ?case
    by (simp add: update-state-at-fst)
qed (simp-all)

```

6.3.3 Assertions

fun *subst-assertion-single* :: $qstate \Rightarrow var \Rightarrow 'a \ \text{exp} \Rightarrow 'a \ \text{assertion} \Rightarrow 'a \ \text{assertion}$
where

```

  subst-assertion-single  $st \ x \ e \ (AConst \ b) = AConst \ b$ 
| subst-assertion-single  $st \ x \ e \ (AComp \ e1 \ \text{cmp} \ e2) = AComp \ (\text{subst-exp-single } st \ x \ e1) \ \text{cmp} \ (\text{subst-exp-single } st \ x \ e2)$ 
| subst-assertion-single  $st \ x \ e \ (AForall \ A) = AForall \ (\text{subst-assertion-single } st \ x \ (\text{shift-vars-exp } 0 \ e) \ A)$ 
| subst-assertion-single  $st \ x \ e \ (AExists \ A) = AExists \ (\text{subst-assertion-single } st \ x$ 

```

```

(shift-vars-exp 0 e) A
| subst-assertion-single st x e (AOr A B) = AOr (subst-assertion-single st x e A)
(subst-assertion-single st x e B)
| subst-assertion-single st x e (AAnd A B) = AAnd (subst-assertion-single st x e
A) (subst-assertion-single st x e B)
| subst-assertion-single st x e (AForallState A) = AForallState (subst-assertion-single
(Suc st) x (shift-states-exp 0 e) A)
| subst-assertion-single st x e (AExistsState A) = AExistsState (subst-assertion-single
(Suc st) x (shift-states-exp 0 e) A)

```

```

fun wf-assertion-aux :: nat ⇒ nat ⇒ 'a assertion ⇒ bool where
  wf-assertion-aux nv ns (AConst b) ←→ True
| wf-assertion-aux nv ns (AComp e1 cmp e2) ←→ wf-exp nv ns e1 ∧ wf-exp nv ns
e2
| wf-assertion-aux nv ns (AAnd A B) ←→ wf-assertion-aux nv ns A ∧ wf-assertion-aux
nv ns B
| wf-assertion-aux nv ns (AOr A B) ←→ wf-assertion-aux nv ns A ∧ wf-assertion-aux
nv ns B

| wf-assertion-aux nv ns (AForall A) ←→ wf-assertion-aux (Suc nv) ns A
| wf-assertion-aux nv ns (AExists A) ←→ wf-assertion-aux (Suc nv) ns A
| wf-assertion-aux nv ns (AForallState A) ←→ wf-assertion-aux nv (Suc ns) A
| wf-assertion-aux nv ns (AExistsState A) ←→ wf-assertion-aux nv (Suc ns) A

```

abbreviation wf-assertion **where** wf-assertion ≡ wf-assertion-aux 0 0

```

lemma wf-shift-vars:
  assumes wf-assertion-aux nv ns A
  shows wf-assertion-aux (Suc nv) ns (shift-vars n A)
  using assms
  by (induct A arbitrary: n nv ns) (simp-all add: wf-shift-vars-exp)

```

```

lemma wf-subst-assertion:
  assumes wf-assertion-aux nv ns A
  and wf-exp nv ns e
  shows wf-assertion-aux nv ns (subst-assertion-single φ x e A)
  using assms
proof (induct A arbitrary: nv ns e φ)
  case (AComp x1a x2 x3a)
  then show ?case
  by (simp add: wf-subst-exp)
qed (simp-all add: wf-shift-vars-exp wf-shift-states-exp)

```

```

lemma subst-assertion-single-charact:
  assumes interp-exp vals states e = snd (states ! st) x
  shows sat-assertion vals states (subst-assertion-single st x e A) S ←→ sat-assertion
vals states A S

```

```

using assms
proof (induct A arbitrary: vals states st e)
  case (AForallState A)
    have sat-assertion vals states (AForallState A) S  $\longleftrightarrow$  ( $\forall \varphi \in S. \text{sat-assertion}$ 
vals ( $\varphi \# \text{states}$ ) A S)
    by simp
    also have ...  $\longleftrightarrow$  ( $\forall \varphi \in S. \text{sat-assertion vals ( $\varphi \# \text{states}$ ) (subst-assertion-single$ 
(Suc st) x (shift-states-exp 0 e) A) S)
    using AForallState(1)[of vals - shift-states-exp 0 e Suc st] AForallState(2)
    by (metis insert-at.simps(1) nth-Cons-Suc shift-states-exp-charact zero-le)
    finally show ?case by simp
  next
    case (AExistsState A)
    have sat-assertion vals states (AExistsState A) S  $\longleftrightarrow$  ( $\exists \varphi \in S. \text{sat-assertion}$ 
vals ( $\varphi \# \text{states}$ ) A S)
    by simp
    also have ...  $\longleftrightarrow$  ( $\exists \varphi \in S. \text{sat-assertion vals ( $\varphi \# \text{states}$ ) (subst-assertion-single$ 
(Suc st) x (shift-states-exp 0 e) A) S)
    by (metis AExistsState.hyps AExistsState.prem insert-at.simps(1) nth-Cons-Suc
shift-states-exp-charact zero-le)
    finally show ?case by simp
  next
    case (AForall A)
    have sat-assertion vals states (AForall A) S  $\longleftrightarrow$  ( $\forall v. \text{sat-assertion (v \# vals)}$ 
states A S)
    by simp
    also have ...  $\longleftrightarrow$  ( $\forall v. \text{sat-assertion (v \# vals) states (subst-assertion-single st$ 
x (shift-vars-exp 0 e) A) S)
    by (metis AForall.hyps AForall.prem insert-at.simps(1) shift-vars-exp-charact
zero-le)
    finally show ?case
    by simp
  next
    case (AExists A)
    have sat-assertion vals states (AExists A) S  $\longleftrightarrow$  ( $\exists v. \text{sat-assertion (v \# vals)}$ 
states A S)
    by simp
    also have ...  $\longleftrightarrow$  ( $\exists v. \text{sat-assertion (v \# vals) states (subst-assertion-single st$ 
x (shift-vars-exp 0 e) A) S)
    by (metis AExists.hyps AExists.prem insert-at.simps(1) shift-vars-exp-charact
zero-le)
    finally show ?case
    by simp
  next
    case (AComp e1 cmp e2)
    then show ?case
    using subst-exp-single-charact[of vals states e st x] by auto
qed (simp-all)

```

lemma *update-state-at-cons*:

update-state-at ($\varphi \# \text{states}$) (*Suc* n) $x \ v = \varphi \# \text{update-state-at states } n \ x \ v$
by (*simp add: update-state-at-def*)

lemma *subst-assertion-single-charact-better*:

assumes $\text{states}' = \text{update-state-at states } st \ x \ (\text{interp-exp vals states } e)$
and $st < \text{length states}$
shows $\text{sat-assertion vals states } (\text{subst-assertion-single } st \ x \ e \ A) \ S \longleftrightarrow \text{sat-assertion vals states}' \ A \ S$
using *assms*
proof (*induct A arbitrary: vals states states' st e*)
case (*AComp x1a x2 x3a*)
then show *?case*
by (*simp add: subst-exp-more-complex-charact*)
next
case (*AForallState A*)
have $\text{sat-assertion vals states } (\text{subst-assertion-single } st \ x \ e \ (\text{AForallState } A)) \ S$
 $\longleftrightarrow (\forall \varphi \in S. \text{sat-assertion vals } (\text{update-state-at } (\varphi \# \text{states}) \ (\text{Suc } st) \ x \ (\text{interp-exp vals } (\varphi \# \text{states}) \ (\text{shift-states-exp } 0 \ e)))) \ A \ S$
by (*simp add: AForallState.hyps AForallState.prem(2)*)
also have $\dots \longleftrightarrow (\forall \varphi \in S. \text{sat-assertion vals } (\varphi \# \text{update-state-at states } st \ x \ (\text{interp-exp vals states } e))) \ A \ S$
by (*metis update-state-at-cons insert-at.simps(1) shift-states-exp-charact zero-le*)
finally show $\text{sat-assertion vals states } (\text{subst-assertion-single } st \ x \ e \ (\text{AForallState } A)) \ S = \text{sat-assertion vals states}' \ (\text{AForallState } A) \ S$
by (*simp add: AForallState.prem(1)*)
next
case (*AExistsState A*)
have $\text{sat-assertion vals states } (\text{subst-assertion-single } st \ x \ e \ (\text{AExistsState } A)) \ S$
 $\longleftrightarrow (\exists \varphi \in S. \text{sat-assertion vals } (\text{update-state-at } (\varphi \# \text{states}) \ (\text{Suc } st) \ x \ (\text{interp-exp vals } (\varphi \# \text{states}) \ (\text{shift-states-exp } 0 \ e)))) \ A \ S$
by (*simp add: AExistsState.hyps AExistsState.prem(2)*)
also have $\dots \longleftrightarrow (\exists \varphi \in S. \text{sat-assertion vals } (\varphi \# \text{update-state-at states } st \ x \ (\text{interp-exp vals states } e))) \ A \ S$
by (*metis update-state-at-cons insert-at.simps(1) shift-states-exp-charact zero-le*)
finally show $\text{sat-assertion vals states } (\text{subst-assertion-single } st \ x \ e \ (\text{AExistsState } A)) \ S = \text{sat-assertion vals states}' \ (\text{AExistsState } A) \ S$
by (*simp add: AExistsState.prem(1)*)
next
case (*AForall A*)
have $\text{sat-assertion vals states } (\text{subst-assertion-single } st \ x \ e \ (\text{AForall } A)) \ S$
 $\longleftrightarrow (\forall v. \text{sat-assertion } (v \# \text{vals}) \ \text{states } (\text{subst-assertion-single } st \ x \ (\text{shift-vars-exp } 0 \ e) \ A) \ S)$
by (*simp add: AForall.hyps AForall.prem(2)*)
then show *?case*
by (*metis AForall.hyps AForall.prem(1) AForall.prem(2) insert-at.simps(1)*)

```

sat-assertion.simps(5) shift-vars-exp-charact zero-le)
next
  case (AExists A)
  have sat-assertion vals states (subst-assertion-single st x e (AExists A)) S
  <math>\longleftrightarrow (\exists v. \text{sat-assertion } (v \# \text{vals}) \text{ states } (\text{subst-assertion-single st } x \text{ (shift-vars-exp } 0 \text{ e) } A) S)</math>
  by (simp add: AExists.hyps AExists.prem(2))
  then show ?case
  by (metis AExists.hyps AExists.prem(1) AExists.prem(2) insert-at.simps(1)
sat-assertion.simps(6) shift-vars-exp-charact zero-le)
qed (simp-all)

```

6.3.4 Transformation for havoc

```

fun transform-havoc where
  transform-havoc x (AForallState A) = AForallState (AForall (subst-assertion-single
0 x (EQVar 0) (shift-vars 0 (transform-havoc x A))))
| transform-havoc x (AExistsState A) = AExistsState (AExists (subst-assertion-single
0 x (EQVar 0) (shift-vars 0 (transform-havoc x A))))
| transform-havoc x (AExists A) = AExists (transform-havoc x A)
| transform-havoc x (AForall A) = AForall (transform-havoc x A)
| transform-havoc x (AOr A B) = AOr (transform-havoc x A) (transform-havoc x
B)
| transform-havoc x (AAnd A B) = AAnd (transform-havoc x A) (transform-havoc
x B)
| transform-havoc x (AConst b) = AConst b
| transform-havoc x (AComp e1 cmp e2) = AComp e1 cmp e2

```

lemma *sem-havoc-bis*:

$\text{sem } (\text{Havoc } x) S = \{(fst \varphi, (snd \varphi)(x := v)) \mid \varphi v. \varphi \in S\}$ (is ?A = ?B)

proof

show ?B \subseteq ?A

using *sem-havoc* by *fastforce*

show ?A \subseteq ?B

proof

fix φ assume $\varphi \in ?A$

then obtain $l \sigma v$ where $\varphi = (l, \sigma(x := v))$ ($l, \sigma \in S$)

by (*metis in-sem prod.collapse single-sem-Havoc-elim*)

then show $\varphi \in ?B$

by *auto*

qed

qed

lemma *helper-update-state*:

$(v \# \text{vals}) ! 0 = \text{snd } ((\text{update-state } \varphi \ x \ v \ \# \ \text{states}) ! 0) \ x$

by (*simp add: update-state-def*)

lemma *helper-S-update-states*:

assumes $S' = \{ \text{update-state } \varphi \ x \ v \mid \varphi \ v. \varphi \in S \}$
shows $(\forall \varphi \in S'. Q \ \varphi) \longleftrightarrow (\forall \varphi \in S. \forall v. Q \ (\text{update-state } \varphi \ x \ v))$
proof
show $\forall \varphi \in S'. Q \ \varphi \implies \forall \varphi \in S. \forall v. Q \ (\text{update-state } \varphi \ x \ v)$
using *assms by blast*
show $\forall \varphi \in S. \forall v. Q \ (\text{update-state } \varphi \ x \ v) \implies \forall \varphi \in S'. Q \ \varphi$
using *assms by force*
qed

lemma *helper-S-update-states-exists:*
assumes $S' = \{ \text{update-state } \varphi \ x \ v \mid \varphi \ v. \varphi \in S \}$
shows $(\exists \varphi \in S'. Q \ \varphi) \longleftrightarrow (\exists \varphi \in S. \exists v. Q \ (\text{update-state } \varphi \ x \ v))$
proof
show $\exists \varphi \in S'. Q \ \varphi \implies \exists \varphi \in S. \exists v. Q \ (\text{update-state } \varphi \ x \ v)$
using *assms by force*
show $\exists \varphi \in S. \exists v. Q \ (\text{update-state } \varphi \ x \ v) \implies \exists \varphi \in S'. Q \ \varphi$
using *assms by blast*
qed

lemma *equiv-havoc-transform:*
assumes $S' = \{ \text{update-state } \varphi \ x \ v \mid \varphi \ v. \varphi \in S \}$
shows *sat-assertion vals states* $P \ S' \longleftrightarrow \text{sat-assertion vals states} \ (\text{transform-havoc } x \ P) \ S$
proof (*induct P arbitrary: vals states*)
case (*AForallState P*)

let $?PP = \text{shift-vars } 0 \ (\text{transform-havoc } x \ P)$
let $?P = \text{subst-assertion-single } 0 \ x \ (\text{EQVar } 0) \ ?PP$

have $rr: \bigwedge \varphi \ v. \text{sat-assertion} \ (v \ \# \ \text{vals}) \ (\varphi \ \# \ \text{states}) \ ?P \ S$
 $\longleftrightarrow \text{sat-assertion} \ (v \ \# \ \text{vals}) \ (\text{update-state } \varphi \ x \ v \ \# \ \text{states}) \ ?P \ S$

proof –

fix $\varphi \ v$

have *sat-assertion* $(v \ \# \ \text{vals}) \ (\text{insert-at } 0 \ \varphi \ \text{states}) \ (\text{subst-assertion-single } 0 \ x \ (\text{EQVar } 0) \ (\text{shift-vars } 0 \ (\text{transform-havoc } x \ P))) \ S =$
sat-assertion $(v \ \# \ \text{vals}) \ (\text{insert-at } 0 \ (\text{update-state } \varphi \ x \ v) \ \text{states}) \ (\text{subst-assertion-single } 0 \ x \ (\text{EQVar } 0) \ (\text{shift-vars } 0 \ (\text{transform-havoc } x \ P))) \ S$

by (*metis* (*no-types*, *lifting*) *One-nat-def helper-update-state insert-at.simps(1)*
interp-exp.simps(3) *length-insert-at list-update-code(2)* *nth-Cons-0 subst-assertion-single-charact*
subst-assertion-single-charact-better trans-less-add2 update-state-at-def zero-less-Suc)

then show *sat-assertion* $(v \ \# \ \text{vals}) \ (\varphi \ \# \ \text{states}) \ ?P \ S \longleftrightarrow \text{sat-assertion} \ (v \ \# \ \text{vals}) \ (\text{update-state } \varphi \ x \ v \ \# \ \text{states}) \ ?P \ S$

by *simp*

qed

have *sat-assertion vals states* $(\text{transform-havoc } x \ (\text{AForallState } P)) \ S \longleftrightarrow \text{sat-assertion vals states} \ (\text{AForallState} \ (\text{AForall } ?P)) \ S$

by *simp*

```

also have ...  $\longleftrightarrow$  ( $\forall \varphi \in S. \forall v. \text{sat-assertion } (v \# \text{vals}) \text{ (update-state } \varphi \ x \ v \ \# \text{states)} \ ?P \ S$ )
  using rr by simp
also have ...  $\longleftrightarrow$  ( $\forall \varphi \in S. \forall v. \text{sat-assertion } (v \# \text{vals}) \text{ (update-state } \varphi \ x \ v \ \# \text{states)} \ ?PP \ S$ )
  using rr subst-assertion-single-charact[of - - - x ?PP S] helper-update-state
  by (metis interp-exp.simps(3))
also have ...  $\longleftrightarrow$  ( $\forall \varphi \in S. \forall v. \text{sat-assertion } \text{vals} \text{ (update-state } \varphi \ x \ v \ \# \text{states)}$ )
(transform-havoc x P) S)
  by (metis insert-at.simps(1) le0 shift-vars-charact)
also have ...  $\longleftrightarrow$  ( $\forall \varphi \in S. \forall v. \text{sat-assertion } \text{vals} \text{ (update-state } \varphi \ x \ v \ \# \text{states)}$ )
P S')
  using AForallState.hyps AForallState.prems by force
also have ...  $\longleftrightarrow$  sat-assertion vals states (AForallState P) S'
  using helper-S-update-states[of S' x S  $\lambda \varphi. \text{sat-assertion } \text{vals} \text{ (}\varphi \ \# \text{states)}$ ] P S']
assms by force
  then show ?case
    using calculation by blast
next
  case (AExistsState P)

let ?PP = shift-vars 0 (transform-havoc x P)
let ?P = subst-assertion-single 0 x (EQVar 0) ?PP

have rr:  $\bigwedge \varphi \ v. \text{sat-assertion } (v \# \text{vals}) \text{ (}\varphi \ \# \text{states)}$  ?P S
 $\longleftrightarrow$  sat-assertion (v # vals) (update-state  $\varphi \ x \ v \ \# \text{states}$ ) ?P S
proof -
  fix  $\varphi \ v$ 
  have sat-assertion (v # vals) (insert-at 0  $\varphi$  states) (subst-assertion-single 0 x
(EQVar 0) (shift-vars 0 (transform-havoc x P))) S =
sat-assertion (v # vals) (insert-at 0 (update-state  $\varphi \ x \ v$ ) states) (subst-assertion-single
0 x (EQVar 0) (shift-vars 0 (transform-havoc x P))) S
  by (metis (no-types, lifting) One-nat-def helper-update-state insert-at.simps(1)
insert-at-charact-2 interp-exp.simps(3) length-insert-at list-update-code(2) subst-assertion-single-charact
subst-assertion-single-charact-better trans-less-add2 update-state-at-def zero-le zero-less-Suc)
  then show sat-assertion (v # vals) ( $\varphi \ \# \text{states}$ ) ?P S  $\longleftrightarrow$  sat-assertion (v #
vals) (update-state  $\varphi \ x \ v \ \# \text{states}$ ) ?P S
  by simp
qed
have sat-assertion vals states (transform-havoc x (AExistsState P)) S  $\longleftrightarrow$  sat-assertion
vals states (AExistsState (AExists ?P)) S
  by simp
also have ...  $\longleftrightarrow$  ( $\exists \varphi \in S. \exists v. \text{sat-assertion } (v \# \text{vals}) \text{ (update-state } \varphi \ x \ v \ \# \text{states)}$ ) ?P S)
  using rr by simp
also have ...  $\longleftrightarrow$  ( $\exists \varphi \in S. \exists v. \text{sat-assertion } (v \# \text{vals}) \text{ (update-state } \varphi \ x \ v \ \# \text{states)}$ ) ?PP S)
  by (metis helper-update-state interp-exp.simps(3) subst-assertion-single-charact)
also have ...  $\longleftrightarrow$  ( $\exists \varphi \in S. \exists v. \text{sat-assertion } \text{vals} \text{ (update-state } \varphi \ x \ v \ \# \text{states)}$ )

```

```

(transform-havoc x P) S)
  by (metis insert-at.simps(1) le0 shift-vars-charact)
  also have ...  $\longleftrightarrow$  ( $\exists \varphi \in S. \exists v. \text{sat-assertion vals (update-state } \varphi \ x \ v \ \# \ \text{states)}$ )
P S')
  using AExistsState.hyps AExistsState.premis by force
  also have ...  $\longleftrightarrow$  sat-assertion vals states (AExistsState P) S'
  using helper-S-update-states-exists[of S' x S  $\lambda\varphi. \text{sat-assertion vals } (\varphi \ \# \ \text{states})$ ]
P S'] assms
  by simp
  then show ?case
  using calculation by blast
qed (simp-all)

```

6.3.5 Syntactic rule for havoc

theorem rule-havoc-syntactic-general:

$\models \{ \text{sat-assertion states vals (transform-havoc } x \ P) \} \text{ Havoc } x \{ \text{sat-assertion states vals } P \}$

proof (rule hyper-hoare-tripleI)

fix S **assume** asm0: sat-assertion states vals (transform-havoc x P) S

let ?S = sem (Havoc x) S

have sat-assertion states vals P ?S \longleftrightarrow sat-assertion states vals (transform-havoc x P) S

proof (rule equiv-havoc-transform)

show sem (Havoc x) S = {update-state $\varphi \ x \ v \mid \varphi \ v. \varphi \in S$ }

by (simp add: sem-havoc-bis update-state-def)

qed

then show sat-assertion states vals P (sem (Havoc x) S)

using asm0 **by** fastforce

qed

theorem rule-havoc-syntactic:

$\models \{ \text{interp-assert (transform-havoc } x \ P) \} \text{ Havoc } x \{ \text{interp-assert } P \}$

by (simp add: rule-havoc-syntactic-general)

6.4 Assignment rule

6.4.1 Program expressions

fun subst-pepx :: var \Rightarrow 'a pexp \Rightarrow 'a pexp \Rightarrow 'a pexp **where**

subst-pepx x e (PVar y) = (if x = y then e else PVar y)

| subst-pepx x e (PBinop p1 op p2) = PBinop (subst-pepx x e p1) op (subst-pepx x e p2)

| subst-pepx x e (PFun f p) = PFun f (subst-pepx x e p)

| subst-pepx - - e = e

lemma subst-pepx-charact:

interp-pepx (subst-pepx x e' e) σ = interp-pepx e ($\sigma(x := \text{interp-pepx } e' \ \sigma)$)

proof (induct e)

```

case (PVar x)
then show ?case
  by (metis fun-upd-apply interp-pexp.simps(1) subst-pexp.simps(1))
qed (simp-all)

```

```

fun subst-pbexp :: var  $\Rightarrow$  'a pexp  $\Rightarrow$  'a pbexp  $\Rightarrow$  'a pbexp where
  subst-pbexp x e (PBAand pb1 pb2) = PBAand (subst-pbexp x e pb1) (subst-pbexp x e pb2)
| subst-pbexp x e (PBOor pb1 pb2) = PBOor (subst-pbexp x e pb1) (subst-pbexp x e pb2)
| subst-pbexp x e (PBComp p1 cmp p2) = PBComp (subst-pexp x e p1) cmp (subst-pexp x e p2)
| subst-pbexp - - (PBConst b) = PBConst b

```

lemma *subst-pbexp-charact*:

interp-pbexp (subst-pbexp x e pb) $\sigma \longleftrightarrow$ interp-pbexp pb ($\sigma(x := \text{interp-pexp } e \ \sigma)$)

proof (induct pb)

case (PBComp x1 x2 x3)

then show ?case

using *interp-pbexp.simps(4) subst-pexp-charact*

by (metis *subst-pbexp.simps(3)*)

qed (simp-all)

6.4.2 Expressions (Boolean and values)

definition *subst-all-states* **where**

subst-all-states x pe states = map (subst-state x pe) states

fun *subst-exp* :: var \Rightarrow 'a pexp \Rightarrow 'a exp \Rightarrow 'a exp **where**

subst-exp x pe (EPVar st y) = (if x = y then pexp-to-exp st pe else EPVar st y)

| *subst-exp x pe (EBinop e1 bop e2) = EBinop (subst-exp x pe e1) bop (subst-exp x pe e2)*

| *subst-exp x pe (EFun f e) = EFun f (subst-exp x pe e)*

| *subst-exp - - e = e*

lemma *subst-exp-charact-aux*:

snd (subst-state x pe (states ! st)) x = interp-exp vals states (pexp-to-exp st pe)

by (induct pe) (simp-all *add: subst-state-def*)

lemma *subst-exp-charact*:

assumes *wf-exp nv (length states) e*

shows *interp-exp vals states (subst-exp x pe e) = interp-exp vals (subst-all-states x pe states) e*

using *assms*

proof (induct e)

case (EPVar st y)

let ?states = *subst-all-states x pe states*

have *snd (subst-state x pe (states ! st)) = snd (?states ! st)*

```

    by (metis EPVar.premis(1) nth-map subst-all-states-def wf-exp.simps(1))
  show interp-exp vals states (subst-exp x pe (EPVar st y)) = interp-exp vals ?states
    (EPVar st y)
  proof (cases x = y)
    case True
      then have interp-exp vals states (subst-exp x pe (EPVar st y)) = interp-exp
        vals states (pexp-to-exp st pe)
        by simp
      moreover have interp-exp vals ?states (EPVar st y) = snd (?states ! st) y
        by simp
      moreover have ... = snd (subst-state x pe (states ! st)) y
        by (simp add: ⟨snd (subst-state x pe (states ! st)) = snd (subst-all-states x pe
          states ! st)⟩)
      moreover have snd (subst-state x pe (states ! st)) x = interp-exp vals states
        (pexp-to-exp st pe)
        by (metis subst-exp-charact-aux)
      ultimately show ?thesis
        using True by fastforce
    next
      case False
      then show ?thesis
        by (metis ⟨snd (subst-state x pe (states ! st)) = snd (subst-all-states x pe states !
          st)⟩ fun-upd-other interp-exp.simps(1) snd-conv subst-exp.simps(1) subst-state-def)
      qed
    next
      case (ELVar st y)
      let ?states = subst-all-states x pe states
      have fst (states ! st) = fst (?states ! st)
        by (metis ELVar.premis(1) fst-conv nth-map subst-all-states-def subst-state-def
          wf-exp.simps(2))
      have interp-exp vals states (subst-exp x pe (ELVar st y)) = interp-exp vals states
        (ELVar st y)
        by simp
      also have ... = fst (states ! st) y
        by simp
      also have ... = fst (?states ! st) y
        by (simp add: ⟨fst (states ! st) = fst (subst-all-states x pe states ! st)⟩)
      also have ... = interp-exp vals ?states (ELVar st y)
        by auto
      then show interp-exp vals states (subst-exp x pe (ELVar st y)) = interp-exp vals
        ?states (ELVar st y)
        using calculation by presburger
      qed (simp-all)
  
```

6.4.3 Assertions

fun transform-assign where

transform-assign x pe (AForallState A) = AForallState (subst-assertion-single 0 x (pexp-to-exp 0 pe) (transform-assign x pe A))

| *transform-assign* *x pe* (*AExistsState A*) = *AExistsState* (*subst-assertion-single 0 x (pexp-to-exp 0 pe) (transform-assign x pe A)*)
 | *transform-assign* *x pe* (*AExists A*) = *AExists* (*transform-assign x pe A*)
 | *transform-assign* *x pe* (*AForall A*) = *AForall* (*transform-assign x pe A*)
 | *transform-assign* *x pe* (*AOr A B*) = *AOr* (*transform-assign x pe A*) (*transform-assign x pe B*)
 | *transform-assign* *x pe* (*AAnd A B*) = *AAnd* (*transform-assign x pe A*) (*transform-assign x pe B*)
 | *transform-assign* *x pe* (*AConst b*) = *AConst b*
 | *transform-assign* *x pe* (*AComp e1 cmp e2*) = *AComp e1 cmp e2*

lemma *transform-assign-works*:

sat-assertion vals states (transform-assign x pe A) S = sat-assertion vals states A (subst-state x pe ' S)

proof (*induct A arbitrary: vals states*)

case (*AForallState A*)

have *sat-assertion vals states (transform-assign x pe (AForallState A)) S*

$\longleftrightarrow (\forall \varphi \in S. \text{sat-assertion vals } (\varphi \# \text{states}) (\text{subst-assertion-single } 0 x (\text{pexp-to-exp } 0 pe) (\text{transform-assign } x pe A)) S)$

by *auto*

also have ... $\longleftrightarrow (\forall \varphi \in S. \text{sat-assertion vals } (\text{update-state-at } (\varphi \# \text{states}) 0 x (\text{interp-exp vals } (\varphi \# \text{states}) (\text{pexp-to-exp } 0 pe))) (\text{transform-assign } x pe A) S)$

by (*simp add: subst-assertion-single-charact-better*)

also have ... $\longleftrightarrow (\forall \varphi \in S. \text{sat-assertion vals } (\text{update-state-at } (\varphi \# \text{states}) 0 x (\text{interp-exp vals } (\varphi \# \text{states}) (\text{pexp-to-exp } 0 pe))) A (\text{subst-state } x pe ' S))$

using *AForallState by presburger*

also have ... $\longleftrightarrow (\forall \varphi \in S. \text{sat-assertion vals } (\text{update-state } \varphi x (\text{interp-exp vals } (\varphi \# \text{states}) (\text{pexp-to-exp } 0 pe)) \# \text{states}) A (\text{subst-state } x pe ' S))$

using *update-state-at-def*

by (*metis list-update-code(2) nth-Cons-0*)

also have ... $\longleftrightarrow (\forall \varphi \in S. \text{sat-assertion vals } (\text{update-state } \varphi x (\text{interp-pexp pe (snd } \varphi)) \# \text{states}) A (\text{subst-state } x pe ' S))$

by (*metis nth-Cons-0 same-syn-sem-exp*)

finally show *sat-assertion vals states (transform-assign x pe (AForallState A))*

S = sat-assertion vals states (AForallState A) (subst-state x pe ' S)

by (*simp add: subst-state-def update-state-def*)

next

case (*AExistsState A*)

have *sat-assertion vals states (transform-assign x pe (AExistsState A)) S*

$\longleftrightarrow (\exists \varphi \in S. \text{sat-assertion vals } (\varphi \# \text{states}) (\text{subst-assertion-single } 0 x (\text{pexp-to-exp } 0 pe) (\text{transform-assign } x pe A)) S)$

by *auto*

also have ... $\longleftrightarrow (\exists \varphi \in S. \text{sat-assertion vals } (\text{update-state-at } (\varphi \# \text{states}) 0 x (\text{interp-exp vals } (\varphi \# \text{states}) (\text{pexp-to-exp } 0 pe))) (\text{transform-assign } x pe A) S)$

by (*simp add: subst-assertion-single-charact-better*)

also have ... $\longleftrightarrow (\exists \varphi \in S. \text{sat-assertion vals } (\text{update-state-at } (\varphi \# \text{states}) 0 x (\text{interp-exp vals } (\varphi \# \text{states}) (\text{pexp-to-exp } 0 pe))) A (\text{subst-state } x pe ' S))$

using *AExistsState by presburger*

also have ... $\longleftrightarrow (\exists \varphi \in S. \text{sat-assertion vals } (\text{update-state } \varphi \ x \ (\text{interp-exp vals } (\varphi \ \# \ \text{states}) \ (\text{pexp-to-exp } 0 \ \text{pe}))) \ \# \ \text{states}) \ A \ (\text{subst-state } x \ \text{pe} \ ' \ S))$
using *update-state-at-def*
by (*metis list-update-code(2) nth-Cons-0*)
also have ... $\longleftrightarrow (\exists \varphi \in S. \text{sat-assertion vals } (\text{update-state } \varphi \ x \ (\text{interp-pexp pe } (\text{snd } \varphi))) \ \# \ \text{states}) \ A \ (\text{subst-state } x \ \text{pe} \ ' \ S))$
by (*metis nth-Cons-0 same-syn-sem-exp*)
finally show *sat-assertion vals states (transform-assign x pe (AExistsState A))*
S = sat-assertion vals states (AExistsState A) (subst-state x pe ' S)
by (*simp add: subst-state-def update-state-def*)
qed (*simp-all*)

6.4.4 Syntactic rule for assignments

theorem *rule-assign-syntactic-general*:

$\models \{ \text{sat-assertion vals states } (\text{transform-assign } x \ \text{pe} \ P) \} \ \text{Assign } x \ (\text{interp-pexp } \text{pe}) \ \{ \text{sat-assertion vals states } P \}$

proof (*rule hyper-hoare-tripleI*)

fix *S* **assume** *asm0*: *sat-assertion vals states (transform-assign x pe P) S*

then have *sat-assertion vals states P (subst-state x pe ' S)*

using *transform-assign-works* **by** *blast*

moreover have $\{(l, \sigma(x := \text{interp-pexp pe } \sigma)) \mid l \ \sigma. (l, \sigma) \in S\} = \text{subst-state } x \ \text{pe} \ ' \ S$ (**is** *?A = ?B*)

proof

show *?A \subseteq ?B*

proof

fix $\varphi \in ?A$

then obtain $l \ \sigma$ **where** $\varphi = (l, \sigma(x := \text{interp-pexp pe } \sigma)) \ (l, \sigma) \in S$

by *blast*

then show $\varphi \in ?B$

by (*metis (mono-tags, lifting) fst-conv image-iff snd-conv subst-state-def*)

qed

show *?B \subseteq ?A*

using *subst-state-def* **by** *fastforce*

qed

ultimately show *sat-assertion vals states P (sem (Assign x (interp-pexp pe)) S)*

by (*simp add: sem-assign*)

qed

theorem *rule-assign-syntactic*:

$\models \{ \text{interp-assert } (\text{transform-assign } x \ \text{pe} \ P) \} \ \text{Assign } x \ (\text{interp-pexp } \text{pe}) \ \{ \text{interp-assert } P \}$

proof (*rule hyper-hoare-tripleI*)

fix *S* **assume** *asm0*: *interp-assert (transform-assign x pe P) S*

then have *sat-assertion [] [] P (subst-state x pe ' S)*

using *transform-assign-works* **by** *blast*

moreover have $\{(l, \sigma(x := \text{interp-pexp pe } \sigma)) \mid l \ \sigma. (l, \sigma) \in S\} = \text{subst-state } x$

```

pe ' S (is ?A = ?B)
proof
  show ?A ⊆ ?B
  proof
    fix φ assume φ ∈ ?A
    then obtain l σ where φ = (l, σ(x := interp-pep pe σ)) (l, σ) ∈ S
      by blast
    then show φ ∈ ?B
      by (metis (mono-tags, lifting) fst-conv image-iff snd-conv subst-state-def)
  qed
  show ?B ⊆ ?A
    using subst-state-def by fastforce
  qed
ultimately show interp-assert P (sem (Assign x (interp-pep pe)) S)
  by (simp add: sem-assign)
qed

```

6.5 Loop rules

```

fun no-exists-state :: 'a assertion ⇒ bool
  where
    no-exists-state (AConst -) ⟷ True
  | no-exists-state (AComp - -) ⟷ True
  | no-exists-state (AForallState A) ⟷ no-exists-state A
  | no-exists-state (AExistsState A) ⟷ False
  | no-exists-state (AForall A) ⟷ no-exists-state A
  | no-exists-state (AExists A) ⟷ no-exists-state A
  | no-exists-state (AAnd A B) ⟷ no-exists-state A ∧ no-exists-state B
  | no-exists-state (AOOr A B) ⟷ no-exists-state A ∧ no-exists-state B

```

lemma *mono-sym-then-up-closed*:

```

assumes no-exists-state A
  and S ⊆ S'
  and sat-assertion vals states A S'
shows sat-assertion vals states A S
using assms
proof (induct A arbitrary: vals states)
  case (AExistsState A)
  then show ?case
    by (metis no-exists-state.simps(4))
qed (auto)

```

definition *up-closed where*

```

up-closed A ⟷ (∀ S S' vals states. S ⊆ S' ∧ sat-assertion vals states A S' ⟶
sat-assertion vals states A S)

```

lemma *up-closedE*:

```

assumes up-closed A
  and S ⊆ S'

```

and *sat-assertion vals states A S*
shows *sat-assertion vals states A S'*
using *assms(1) assms(2) assms(3) up-closed-def* **by** *blast*

lemma *sat-assertion-aforallstateI*:
assumes $\bigwedge \varphi. \varphi \in S \implies \text{sat-assertion vals } (\varphi \# \text{states}) A S$
shows *sat-assertion vals states (AForallState A) S*
using *assms sat-assertion.simps(3)* **by** *blast*

lemma *join-entails*:
assumes *up-closed A*
and *sat-assertion vals states (AForallState A) S1*
and *sat-assertion vals states (AForallState A) S2*
shows *sat-assertion vals states (AForallState A) (S1 \cup S2)*
proof (*rule sat-assertion-aforallstateI*)
fix φ
assume *asm0: $\varphi \in S1 \cup S2$*
show *sat-assertion vals ($\varphi \# \text{states}$) A (S1 \cup S2)*
proof (*cases $\varphi \in S1$*)
case *True*
then have *sat-assertion vals ($\varphi \# \text{states}$) A S1*
using *assms(2) sat-assertion.simps(3)* **by** *blast*
then show *?thesis*
by (*meson assms(1) sup-ge1 up-closedE*)
next
case *False*
then show *?thesis*
by (*metis UnE asm0 assms(1) assms(3) sat-assertion.simps(3) sup-ge2 up-closedE*)
qed
qed

lemma *general-join-entails*:
assumes *up-closed A*
and $\bigwedge x. \text{sat-assertion vals states } (AForallState A) (F x)$
shows *sat-assertion vals states (AForallState A) ($\bigcup x. F x$)*
proof (*rule sat-assertion-aforallstateI*)
fix φ
assume *asm0: $\varphi \in \bigcup (\text{range } F)$*
then obtain *x where $\varphi \in F x$*
by *blast*
then have *sat-assertion vals ($\varphi \# \text{states}$) A (F x)*
using *assms(2)* **by** *force*
then show *sat-assertion vals ($\varphi \# \text{states}$) A ($\bigcup (\text{range } F)$)*
by (*meson Union-upper assms(1) range-eqI up-closedE*)

qed

```
fun no-forall-state :: 'a assertion  $\Rightarrow$  bool
  where
    no-forall-state (AConst -)  $\longleftrightarrow$  True
  | no-forall-state (AComp - -)  $\longleftrightarrow$  True
  | no-forall-state (AForallState A)  $\longleftrightarrow$  False
  | no-forall-state (AExistsState A)  $\longleftrightarrow$  no-forall-state A
  | no-forall-state (AForall A)  $\longleftrightarrow$  no-forall-state A
  | no-forall-state (AExists A)  $\longleftrightarrow$  no-forall-state A
  | no-forall-state (AAnd A B)  $\longleftrightarrow$  no-forall-state A  $\wedge$  no-forall-state B
  | no-forall-state (AOr A B)  $\longleftrightarrow$  no-forall-state A  $\wedge$  no-forall-state B
```

```
lemma no-forall-exists-state-not:
  no-forall-state A  $\equiv$  no-exists-state (ANot A)
by (induct A) auto
```

```
fun no-forall-state-after-existential :: 'a assertion  $\Rightarrow$  bool
  where
    no-forall-state-after-existential (AConst -)  $\longleftrightarrow$  True
  | no-forall-state-after-existential (AComp - -)  $\longleftrightarrow$  True
  | no-forall-state-after-existential (AForallState A)  $\longleftrightarrow$  no-forall-state-after-existential
    A
  | no-forall-state-after-existential (AForall A)  $\longleftrightarrow$  no-forall-state-after-existential A
  | no-forall-state-after-existential (AAnd A B)  $\longleftrightarrow$  no-forall-state-after-existential
    A  $\wedge$  no-forall-state-after-existential B
  | no-forall-state-after-existential (AOr A B)  $\longleftrightarrow$  no-forall-state-after-existential A
     $\wedge$  no-forall-state-after-existential B
  | no-forall-state-after-existential (AExists A)  $\longleftrightarrow$  no-forall-state A
  | no-forall-state-after-existential (AExistsState A)  $\longleftrightarrow$  no-forall-state A
```

```
lemma up-closed-from-no-exists-state-false:
  assumes no-forall-state A
    and sat-assertion vals states A (S n)
    shows sat-assertion vals states A ( $\bigcup$  n. S n)
  using assms
proof (induct A)
  case (AExistsState A)
  then show ?case
    by (meson UN-upper no-forall-exists-state-not iso-tuple-UNIV-I mono-sym-then-up-closed
    sat-assertion-Not)
next
  case (AForall A)
  then show ?case
```

```

  by (meson UN-upper no-forall-exists-state-not iso-tuple-UNIV-I mono-sym-then-up-closed
sat-assertion-Not)
next
  case (AExists A)
  then show ?case
  by (meson UNIV-I no-forall-exists-state-not UN-upper mono-sym-then-up-closed
sat-assertion-Not)
qed (force+)

```

definition *shift-sequence* :: $(\text{nat} \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow (\text{nat} \Rightarrow 'a)$
where
shift-sequence $S\ n\ m = S\ (m + n)$

lemma *shift-sequence-properties*:

```

assumes ascending S
shows ascending (shift-sequence S n)
and  $(\bigcup m. S\ m) = (\bigcup m. (\text{shift-sequence}\ S\ n)\ m)$  (is ?A = ?B)
apply (metis (mono-tags, lifting) Suc-n-not-le-n add-Suc ascendingI ascend-
ing-def assms nle-le shift-sequence-def)
proof
show ?B  $\subseteq$  ?A
by (simp add: SUP-le-iff UN-upper shift-sequence-def)
show ?A  $\subseteq$  ?B
by (metis SUP-mono' ascending-def assms le-add1 shift-sequence-def)
qed

```

fun *extract-indices-sat-P* **where**

```

  extract-indices-sat-P P S 0 = (SOME n. P (S n))
| extract-indices-sat-P P S (Suc m) = (SOME n. P (S n)  $\wedge$  n > extract-indices-sat-P
P S m)

```

definition *holds-infinitely-often* **where**

holds-infinitely-often $P\ S \longleftrightarrow (\forall m. \exists n. n > m \wedge P\ (S\ n))$

lemma *extract-indices-sat-P-properties*:

```

assumes holds-infinitely-often P S
shows P (S (extract-indices-sat-P P S 0))
and  $n > 0 \implies P\ (S\ (\text{extract-indices-sat-P}\ P\ S\ n))$ 
 $\wedge$  extract-indices-sat-P P S n > extract-indices-sat-P P S (n - 1)
apply (metis assms extract-indices-sat-P.simps(1) holds-infinitely-often-def someI-ex)
using assms
proof (induct n)
case (Suc n)
then have P (S (extract-indices-sat-P P S n))
by (metis bot-nat-0.not-eq-extremum extract-indices-sat-P.simps(1) holds-infinitely-often-def
someI-ex)
let ?P =  $\lambda m. P\ (S\ m) \wedge m > \text{extract-indices-sat-P}\ P\ S\ n$ 
have ?P (SOME m. ?P m)

```

```

proof (rule someI-ex)
  show  $\exists x. P (S x) \wedge \text{extract-indices-sat-P } P S n < x$ 
    by (meson Suc.prem(2) holds-infinitely-often-def)
qed
then show ?case
  by simp
qed (simp)

lemma extract-indices-sat-P-larger:
  assumes holds-infinitely-often P S
  shows extract-indices-sat-P P S n  $\geq$  n
  using assms
proof (induct n)
  case (Suc n)
  then show ?case
    by (metis Suc-leI diff-Suc-1 diff-diff-cancel extract-indices-sat-P-properties(2)
less-imp-diff-less zero-less-Suc)
qed (simp)

definition subseq-sat where
  subseq-sat P S n = S (extract-indices-sat-P P S n)

lemma subseq-sat-properties:
  assumes holds-infinitely-often P S
  and ascending S
  shows ascending (subseq-sat P S)
  and  $\bigwedge n. P (\text{subseq-sat } P S n)$ 
  and  $(\bigcup n. S n) = (\bigcup n. \text{subseq-sat } P S n)$  (is ?A = ?B)
proof (rule ascendingI)
  fix n
  have extract-indices-sat-P P S (Suc n)  $>$  extract-indices-sat-P P S n
    by (metis assms(1) diff-Suc-1 extract-indices-sat-P-properties(2) zero-less-Suc)
  then show subseq-sat P S n  $\subseteq$  subseq-sat P S (Suc n)
    by (metis ascending-def assms(2) less-imp-le-nat subseq-sat-def)
  fix n show P (subseq-sat P S n)
    by (metis assms(1) extract-indices-sat-P-properties(1) extract-indices-sat-P-properties(2)
grOI subseq-sat-def)
next
  show ?A = ?B
  proof
  show ?B  $\subseteq$  ?A
    by (simp add: SUP-le-iff UN-upper subseq-sat-def)
  show ?A  $\subseteq$  ?B
  proof
  fix x assume  $x \in \bigcup (\text{range } S)$ 
  then obtain n where  $x \in S n$  by blast
  then have  $x \in \text{subseq-sat } P S n$ 
    by (metis ascending-def assms(1) assms(2) extract-indices-sat-P-larger
subseq-sat-def subsetD)

```

```

    then show  $x \in ?B$  by blast
  qed
qed
qed

lemma no-forall-state-after-existential-sem:
  assumes no-forall-state-after-existential A
    and ascending S
    and  $\bigwedge n. \text{sat-assertion vals states } A (S n)$ 
  shows  $\text{sat-assertion vals states } A (\bigcup n. S n)$ 
  using assms
proof (induct A arbitrary: vals states S)
  case (AForallState A)
  show ?case
  proof (rule sat-assertion-aforallstateI)
    fix  $\varphi$ 
    assume  $\varphi \in \bigcup (\text{range } S)$ 
    then obtain  $n$  where  $\varphi \in S n$  by blast
    then have  $\bigwedge m. m \geq n \implies \text{sat-assertion vals } (\varphi \# \text{states}) A (S m)$ 
    by (meson AForallState.prem(2) AForallState.prem(3) ascending-def sat-assertion.simps(3)
subsetD)
    let ?S = shift-sequence S n
    have  $\text{sat-assertion vals } (\varphi \# \text{states}) A (\bigcup (\text{range } ?S))$ 
    proof (rule AForallState(1))
      show no-forall-state-after-existential A
        using AForallState.prem(1) by auto
      show ascending ?S
        by (simp add: AForallState.prem(2) shift-sequence-properties(1))
      fix  $m$  show  $\text{sat-assertion vals } (\varphi \# \text{states}) A (\text{shift-sequence } S n m)$ 
        by (simp add:  $\langle \bigwedge m. n \leq m \implies \text{sat-assertion vals } (\varphi \# \text{states}) A (S m) \rangle$ 
shift-sequence-def)
    qed
    then show  $\text{sat-assertion vals } (\varphi \# \text{states}) A (\bigcup (\text{range } S))$ 
    by (metis AForallState.prem(2) shift-sequence-properties(2))
  qed
qed
next
  case (AExistsState A)
  then show ?case
    by (meson no-forall-state.simps(4) no-forall-state-after-existential.simps(8)
up-closed-from-no-exists-state-false)
next
  case (AExists A)
  then show ?case
    by (meson no-forall-state.simps(6) no-forall-state-after-existential.simps(7)
up-closed-from-no-exists-state-false)
next
  case (AOr A1 A2)

```

```

show ?case
proof (cases holds-infinitely-often (sat-assertion vals states A1) S)
  case True
    then have sat-assertion vals states A1 (⋃ (range (subseq-sat (sat-assertion
      vals states A1) S)))
      using AOr.hyps(1) AOr.prem(1) AOr.prem(2) subseq-sat-properties(1)
      subseq-sat-properties(2) no-forall-state-after-existential.simps(6) by blast
    then show ?thesis
      using AOr.prem(2) True subseq-sat-properties(3) by fastforce
  next
    case False
      then have holds-infinitely-often (sat-assertion vals states A2) S
        using AOr.prem(3) sat-assertion.simps(8) holds-infinitely-often-def
        by (metis gt-ex max-less-iff-conj)
      then have sat-assertion vals states A2 (⋃ (range (subseq-sat (sat-assertion
        vals states A2) S)))
        using AOr.hyps(2) AOr.prem(1) AOr.prem(2) subseq-sat-properties(1)
        subseq-sat-properties(2) no-forall-state-after-existential.simps(6) by blast
      then show ?thesis
        using AOr.prem(2) ⟨holds-infinitely-often (sat-assertion vals states A2) S⟩
        subseq-sat-properties(3) by fastforce
    qed
  next
    case (AAnd A1 A2)
      then show ?case
        using sat-assertion.simps(7) no-forall-state-after-existential.simps(5) by blast
    qed (simp-all)

```

lemma *upwards-closed-syn-sem-practical*:

```

assumes no-forall-state-after-existential A
shows upwards-closed (λn. interp-assert A) (interp-assert A)
by (simp add: asms no-forall-state-after-existential-sem upwards-closed-def)

```

theorem *while-general-syntactic*:

```

assumes ∧n. ⊨ {P n} if-then b C {P (Suc n)}
  and ∧n. ⊨ {P n} Assume (lnot b) {interp-assert A}
  and no-forall-state-after-existential A
shows ⊨ {P 0} while-cond b C {conj (interp-assert A) (holds-forall (lnot b))}
by (metis asms(1) asms(2) asms(3) upwards-closed-syn-sem-practical while-general)

```

theorem *while-forall-exists-simpler*:

```

assumes ⊨ {I} if-then b C {I}
  and ⊨ {I} Assume (lnot b) {interp-assert Q}
  and no-forall-state-after-existential Q
shows ⊨ {I} while-cond b C {conj (interp-assert Q) (holds-forall (lnot b))}
by (metis asms(1) asms(2) asms(3) upwards-closed-syn-sem-practical while-general)

```

theorem *while-d-syntactic*:
assumes $\models \{ \text{interp-assert } A \} \text{ if-then } b \ C \ \{ \text{interp-assert } A \}$
and *no-forall-state-after-existential* A
and *no-exists-state* A
shows $\models \{ \text{interp-assert } A \} \text{ while-cond } b \ C \ \{ \text{conj } (\text{interp-assert } A) \ (\text{holds-forall } (\text{not } b)) \}$
using *assms(1)*
proof (*rule while-d*)
show *upwards-closed* $(\lambda a. \text{interp-assert } A) \ (\text{interp-assert } A)$
by (*simp add: assms(2) no-forall-state-after-existential-sem upwards-closed-def*)
fix n **show** *downwards-closed* $(\text{interp-assert } A)$
using *assms(3) downwards-closed-def mono-sym-then-up-closed* **by** *fastforce*
qed

lemma *downwards-closed-is-hypersafety*:
hypersafety $P \longleftrightarrow \text{downwards-closed } P$
using *downwards-closed-def hypersafety-def* **by** *metis*

6.6 Rewrite rules for 'a assertions

definition *equiv where*
equiv $A \ B \longleftrightarrow (\forall \text{vals states } S. \text{sat-assertion vals states } A \ S \longleftrightarrow \text{sat-assertion vals states } B \ S)$

lemma *forall-commute*:
equiv $(A\text{ForallState } (A\text{Forall } A)) \ (A\text{Forall } (A\text{ForallState } A))$
using *equiv-def* **by** *force*

lemma *exists-commute*:
equiv $(A\text{ExistsState } (A\text{Exists } A)) \ (A\text{Exists } (A\text{ExistsState } A))$
using *equiv-def* **by** *force*

lemma *forall-state-and*:
equiv $(A\text{ForallState } (A\text{And } A \ B)) \ (A\text{And } (A\text{ForallState } A) \ (A\text{ForallState } B))$
using *equiv-def* **by** *force*

lemma *exists-state-or*:
equiv $(A\text{ExistsState } (A\text{Or } A \ B)) \ (A\text{Or } (A\text{ExistsState } A) \ (A\text{ExistsState } B))$
using *equiv-def* **by** *force*

lemma *forall-and*:
equiv $(A\text{Forall } (A\text{And } A \ B)) \ (A\text{And } (A\text{Forall } A) \ (A\text{Forall } B))$
using *equiv-def* **by** *force*

lemma *exists-or*:
equiv $(A\text{Exists } (A\text{Or } A \ B)) \ (A\text{Or } (A\text{Exists } A) \ (A\text{Exists } B))$

using *equiv-def* by *force*

lemma *entailment-natural-partition*:

assumes *no-forall-state P*

shows *entails* (*natural-partition* ($\lambda(n::nat). \text{interp-assert } (AForallState P)$)) (*interp-assert* (*AForallState P*))

proof (*rule entailsI*)

fix *S* :: ((*nat* \Rightarrow 'a) \times (*nat* \Rightarrow 'a)) *set*

assume *asm0*: *natural-partition* ($\lambda(n::nat). \text{interp-assert } (AForallState P)$) *S*

then obtain *F* **where** *S* = ($\bigcup(n::nat). F n$) $\wedge n. \text{interp-assert } (AForallState P) (F n)$

using *natural-partitionE*[of $\lambda n. \text{interp-assert } (AForallState P) S$] by *blast*

then have $\bigwedge \varphi. \varphi \in S \Longrightarrow \text{sat-assertion } [] [\varphi] P S$

by (*meson UN-iff assms sat-assertion.simps(3) up-closed-from-no-exists-state-false*)

then show *interp-assert* (*AForallState P*) *S*

by *simp*

qed

lemma *no-forall-state-mono*:

assumes *no-forall-state A*

and *sat-assertion vals states A S*

and $S \subseteq S'$

shows *sat-assertion vals states A S'*

by (*metis assms(1) assms(2) assms(3) mono-sym-then-up-closed no-forall-exists-state-not sat-assertion-Not*)

lemma *entailment-loop-join*:

assumes *no-forall-state P*

shows *entails* (*join* (*interp-assert* (*AForallState P*)) (*interp-assert* (*AForallState P*))) (*interp-assert* (*AForallState P*))

proof (*rule entailsI*)

fix *S* :: ((*nat* \Rightarrow 'a) \times (*nat* \Rightarrow 'a)) *set*

assume *asm0*: *join* (*interp-assert* (*AForallState P*)) (*interp-assert* (*AForallState P*)) *S*

then obtain *S1 S2* **where** $r: S = S1 \cup S2$ *interp-assert* (*AForallState P*) *S1* *interp-assert* (*AForallState P*) *S2*

by (*metis join-def*)

have $\bigwedge \varphi. \varphi \in S \Longrightarrow \text{sat-assertion } [] [\varphi] P S$

proof –

fix φ **assume** *asm1*: $\varphi \in S$

then show *sat-assertion* $[] [\varphi] P S$

by (*metis Un-iff r assms no-forall-state-mono sat-assertion.simps(3) sup-ge1 sup-ge2*)

qed

then show *interp-assert* (*AForallState P*) *S*

by *simp*
qed

6.7 Free variables and safe frame rule

fun *wr* :: (nat, nat) *stmt* \Rightarrow nat *set* **where**
 | *wr* *Skip* = {}
 | *wr* (*Assign* *x* *-*) = {*x*}
 | *wr* (*Havoc* *x*) = {*x*}
 | *wr* (*Assume* *b*) = {}
 | *wr* (*C1* ;; *C2*) = *wr* *C1* \cup *wr* *C2*
 | *wr* (*If* *C1* *C2*) = *wr* *C1* \cup *wr* *C2*
 | *wr* (*While* *C*) = *wr* *C*

definition *agree-on* **where**
agree-on *V* σ σ' \longleftrightarrow ($\forall x \in V. \sigma x = \sigma' x$)

lemma *agree-onI*:
assumes $\bigwedge x. x \in V \implies \sigma x = \sigma' x$
shows *agree-on* *V* σ σ'
using *agree-on-def* *assms* **by** *blast*

lemma *agree-onE*:
assumes *agree-on* *V* σ σ'
and $x \in V$
shows $\sigma x = \sigma' x$
by (*meson* *agree-on-def* *assms*(1) *assms*(2))

lemma *agree-on-subset*:
assumes *agree-on* *V'* σ σ'
and $V \subseteq V'$
shows *agree-on* *V* σ σ'
by (*meson* *agree-on-def* *assms*(1) *assms*(2) *in-mono*)

lemma *agree-on-trans*:
assumes *agree-on* *V* σ σ'
and *agree-on* *V* σ' σ''
shows *agree-on* *V* σ σ''
proof (*rule* *agree-onI*)
fix *x* **assume** $x \in V$
then show $\sigma x = \sigma'' x$
by (*metis* *agree-on-def* *assms*(1) *assms*(2))
qed

lemma *agree-on-sym*:
assumes *agree-on* *V* σ σ'
shows *agree-on* *V* σ' σ
by (*metis* *agree-on-def* *assms*)

```

lemma wr-charact:
  assumes single-sem  $C \ \sigma \ \sigma'$ 
    and  $wr \ C \cap \ V = \ \{\}$ 
  shows agree-on  $V \ \sigma \ \sigma'$ 
  using assms
proof (induct rule: single-sem.induct)
  case (SemSeq  $C1 \ \sigma \ \sigma1 \ C2 \ \sigma2$ )
  then show ?case
    by (metis Un-empty agree-on-trans inf-sup-distrib2 wr.simps(5))
qed (auto simp add: agree-on-def)

fun fv-exp :: 'a exp  $\Rightarrow$  var set where
  fv-exp (EBinop  $e1 \ - \ e2$ ) = fv-exp  $e1 \ \cup \ fv-exp \ e2$ 
| fv-exp (EPVar  $- \ x$ ) =  $\{x\}$ 
| fv-exp (EFun  $- \ e$ ) = fv-exp  $e$ 
| fv-exp  $-$  =  $\{\}$ 

lemma fv-wr-charact-exp:

  assumes agree-on (fv-exp  $e$ )  $\sigma \ \sigma'$ 
    and  $n \leq \text{length} \ \text{states}$ 
    and wf-exp  $nv \ (\text{Suc} \ (\text{length} \ \text{states})) \ e$ 
  shows interp-exp vals (insert-at  $n \ (l, \ \sigma) \ \text{states}$ )  $e = \text{interp-exp vals} \ (\text{insert-at} \ n \ (l, \ \sigma') \ \text{states}) \ e$ 
  using assms
proof (induct e)
  case (EPVar  $st \ x$ )
  then show ?case
  proof (cases st < n)
    case True
    then show ?thesis
    by (simp add: assms(2) insert-at-charact-1)
  next
  case False
  then have  $st \geq n$  by simp
  then show ?thesis
  proof (cases st = n)
    case True
    then have interp-exp vals (insert-at  $n \ (l, \ \sigma) \ \text{states}$ ) (EPVar  $st \ x$ ) =  $\sigma \ x$ 
    by (simp add: assms(2) insert-at-charact-2)
    also have  $\dots = \sigma' \ x$ 
    by (metis EPVar.prem(1) agree-on-def fv-exp.simps(2) insertCI)
    then show ?thesis
    by (simp add: True assms(2) insert-at-charact-2)
  next
  case False
  then have interp-exp vals (insert-at  $n \ (l, \ \sigma) \ \text{states}$ ) (EPVar  $st \ x$ ) = snd
  (insert-at  $n \ (l, \ \sigma) \ \text{states}$ ) !  $st$ )  $x$ 

```

```

    by simp
    also have ... = snd (states ! (st - 1)) x
    by (metis False One-nat-def ⟨n ≤ st⟩ add-diff-inverse-nat assms(2) dual-order.antisym
insert-at-character-3 le-less-Suc-eq le-zero-eq not-less-eq-eq plus-1-eq-Suc)
    then show ?thesis
    by (metis False One-nat-def ⟨n ≤ st⟩ add-diff-inverse-nat assms(2) dual-order.antisym
insert-at-character-3 interp-exp.simps(1) le-less-Suc-eq le-zero-eq not-less-eq-eq plus-1-eq-Suc)
  qed
qed
next
case (ELVar st x)
then show ?case
proof (cases st < n)
  case True
  then show ?thesis
  by (simp add: assms(2) insert-at-character-1)
next
case False
then have st ≥ n by simp
then show ?thesis
proof (cases st = n)
  case True
  then have interp-exp vals (insert-at n (l, σ) states) (ELVar st x) = l x
  by (simp add: assms(2) insert-at-character-2)
  then show ?thesis
  by (simp add: True assms(2) insert-at-character-2)
next
case False
  then have interp-exp vals (insert-at n (l, σ) states) (ELVar st x) = fst
((insert-at n (l, σ) states) ! st) x
  by simp
  also have ... = fst (states ! (st - 1)) x
  by (metis False One-nat-def ⟨n ≤ st⟩ add-diff-inverse-nat assms(2) dual-order.antisym
insert-at-character-3 le-less-Suc-eq le-zero-eq not-less-eq-eq plus-1-eq-Suc)
  then show ?thesis
  by (metis False One-nat-def ⟨n ≤ st⟩ add-diff-inverse-nat assms(2) dual-order.antisym
insert-at-character-3 interp-exp.simps(2) le-less-Suc-eq le-zero-eq not-less-eq-eq plus-1-eq-Suc)
  qed
qed
next
case (EBinop e1 x2 e2)
then show ?case
  by (simp add: agree-on-def)
qed (simp-all)

```

```

fun fv where
  fv (AAnd F1 F2) = fv F1 ∪ fv F2

```

```

| fv (AOr F1 F2) = fv F1 ∪ fv F2
| fv (AForall F) = fv F
| fv (AExists F) = fv F
| fv (AForallState F) = fv F
| fv (AExistsState F) = fv F
| fv (AConst b) = {}
| fv (AComp e1 cmp e2) = fv-exp e1 ∪ fv-exp e2

```

lemma *fv-wr-charact-aux*:

```

assumes agree-on (fv F) σ σ'
and n ≤ length states
and sat-assertion vals (insert-at n (l, σ) states) F S
and wf-assertion-aux nv (Suc (length states)) F
shows sat-assertion vals (insert-at n (l, σ') states) F S
using assms
proof (induct F arbitrary: vals states n nv)
case (AExists F)
then show ?case
by (metis fv.simps(4) sat-assertion.simps(6) wf-assertion-aux.simps(6))
next
case (AComp e1 cmp e2)
then have interp-exp vals (insert-at n (l, σ) states) e1 = interp-exp vals (insert-at
n (l, σ') states) e1
using fv-wr-charact-exp[of e1 σ σ' n states nv vals l] agree-on-subset by fastforce
moreover have interp-exp vals (insert-at n (l, σ) states) e2 = interp-exp vals
(insert-at n (l, σ') states) e2
using fv-wr-charact-exp[of e2 σ σ' n states nv vals l] agree-on-subset AComp
by auto
ultimately show sat-assertion vals (insert-at n (l, σ') states) (AComp e1 cmp
e2) S
using AComp.prems(3) by auto
next
case (AForallState F)
then have  $\bigwedge \varphi. \varphi \in S \implies$  sat-assertion vals (insert-at (Suc n) (l, σ) (ϕ #
states)) F S
by simp
then have  $\bigwedge \varphi. \varphi \in S \implies$  sat-assertion vals (insert-at (Suc n) (l, σ') (ϕ #
states)) F S
by (metis AForallState.hyps AForallState.prems(1) AForallState.prems(2) AForall-
State.prems(4) Suc-le-mono fv.simps(5) length-Cons wf-assertion-aux.simps(7))
then show ?case by simp
next
case (AExistsState F)
then obtain ϕ where asm0: ϕ ∈ S sat-assertion vals (insert-at (Suc n) (l, σ)
(ϕ # states)) F S

```

by auto
then have *sat-assertion vals* (*insert-at* (*Suc n*) (*l, σ'*) (*φ # states*)) *F S*
by (*metis AExistsState.hyps AExistsState.premis(1) AExistsState.premis(2) AExistsState.premis(4) Suc-le-mono fv.simps(6) length-Cons wf-assertion-aux.simps(8)*)
then show *?case using asm0 by auto*
qed (*auto simp add: agree-on-def*)

lemma *fv-wr-charact:*

assumes *agree-on* (*fv F*) *σ σ'*
and *sat-assertion vals* (*(l, σ) # states*) *F S*
and *wf-assertion-aux nv* (*Suc (length states)*) *F*
shows *sat-assertion vals* (*(l, σ') # states*) *F S*
proof –
have *sat-assertion vals* (*insert-at 0 (l, σ') states*) *F S*
using *assms fv-wr-charact-aux by fastforce*
then show *?thesis*
by simp
qed

lemma *syntactic-safe-frame-preserved:*

assumes *wr C ∩ fv F = {}*
and *sat-assertion vals states* *F S*
and *wf-assertion-aux nv* (*length states*) *F*
and *no-exists-state F*
shows *sat-assertion vals states* *F (sem C S)*
using *assms*
proof (*induct F arbitrary: vals states nv*)
case (*AForallState F*)
then have $\bigwedge \varphi. \varphi \in \text{sem } C \ S \implies \text{sat-assertion vals } (\varphi \# \text{states}) \ F \ (\text{sem } C \ S)$
proof –
fix *φ* **assume** *asm0: φ ∈ sem C S*
then obtain *σ* **where** *single-sem C σ (snd φ) (fst φ, σ) ∈ S*
using *in-sem by blast*
then have *sat-assertion vals* (*(fst φ, σ) # states*) *F (sem C S)*
using *AForallState.hyps AForallState.premis assms(1) by auto*
moreover have *agree-on* (*fv F*) *σ (snd φ)*
using *AForallState.premis(1) <<(C::(nat, nat) stmt), σ::nat ⇒ nat> → snd (φ::(nat ⇒ nat) × (nat ⇒ nat))> wr-charact by auto*
moreover have *wf-assertion-aux nv* (*Suc (length states)*) *F*
using *AForallState.premis(3) by auto*
ultimately have *sat-assertion vals* (*(fst φ, snd φ) # states*) *F (sem C S)*
using *fv-wr-charact[of F σ snd φ vals fst φ states sem C S]*
by fast
then show *sat-assertion vals* (*φ # states*) *F (sem C S)*
by simp
qed
then show *?case*

using *AForallState.hyps AForallState.premis(2) assms(1)* **by** *auto*
qed (*fastforce+*)

theorem *safe-frame-rule-syntactic*:

assumes $wr\ C \cap\ fv\ F = \{\}$
and *wf-assertion F*
and *no-exists-state F*
shows $\models \{interp\text{-}assert\ F\}\ C\ \{interp\text{-}assert\ F\}$
by (*metis assms(1) assms(2) assms(3) hyper-hoare-tripleI list.size(3) syntac-*
tic-safe-frame-preserved)

theorem *LUpdateS*:

assumes $\models \{(\lambda S.\ P\ S \wedge\ e\text{-recorded-in-t}\ e\ t\ S)\}\ C\ \{Q\}$
and *not-fv-hyper t P*
and *not-fv-hyper t Q*
shows $\models \{P\}\ C\ \{Q\}$
proof (*rule hyper-hoare-tripleI*)
fix *S* **assume** *asm: P S*
let *?S = assign-exp-to-lvar-set e t S*
have *Q (sem C ?S)*
using *asm assms(1) assms(2) e-recorded-in-t-if-assigned hyper-hoare-tripleE*
not-fv-hyper-assign-exp **by** *fastforce*
then show *Q (sem C S)*
by (*meson assign-exp-to-lvar-set-same-mod-updates assms(3) not-fv-hyper-def*
same-mod-updates-sym sem-update-commute)
qed

end

theory *TotalLogic*

imports *Loops Compositionality SyntacticAssertions*

begin

7 Terminating Hyper-Triples

definition *total-hyper-triple* ($\langle \models_{TERM} \{-\} - \{-\} \rangle$ [51,0,0] 81) **where**

$\models_{TERM} \{P\}\ C\ \{Q\} \longleftrightarrow (\models \{P\}\ C\ \{Q\} \wedge (\forall S.\ P\ S \longrightarrow (\forall \varphi \in S.\ \exists \sigma'.\ single\text{-}sem\ C\ (snd\ \varphi)\ \sigma')))$

lemma *total-hyper-triple-equiv*:

$\models_{TERM} \{P\}\ C\ \{Q\} \longleftrightarrow (\models \{P\}\ C\ \{Q\} \wedge (\forall S.\ P\ S \longrightarrow (\forall \varphi \in S.\ \exists \sigma'.\ (fst\ \varphi,\ \sigma') \in sem\ C\ S \wedge single\text{-}sem\ C\ (snd\ \varphi)\ \sigma')))$

by (*metis prod.collapse single-step-then-in-sem total-hyper-triple-def*)

lemma *total-hyper-tripleI*:

assumes $\models \{P\}\ C\ \{Q\}$

and $\bigwedge \varphi\ S.\ P\ S \wedge\ \varphi \in S \implies (\exists \sigma'.\ single\text{-}sem\ C\ (snd\ \varphi)\ \sigma')$

shows $\models_{TERM} \{P\}\ C\ \{Q\}$

by (*simp add: assms(1) assms(2) total-hyper-triple-def*)

definition *terminates-in where*

terminates-in $C\ S \longleftrightarrow (\forall \varphi \in S. \exists \sigma'. \text{single-sem } C\ (\text{snd } \varphi)\ \sigma')$

lemma *terminates-inI:*

assumes $\bigwedge \varphi. \varphi \in S \implies \exists \sigma'. \text{single-sem } C\ (\text{snd } \varphi)\ \sigma'$

shows *terminates-in* $C\ S$

using *assms terminates-in-def by blast*

lemma *iterate-sem-mono:*

assumes $S \subseteq S'$

shows *iterate-sem* $n\ C\ S \subseteq \text{iterate-sem } n\ C\ S'$

using *assms*

by (*induct n arbitrary: S S' (simp-all add: sem-monotonic)*)

lemma *terminates-in-while-loop:*

assumes *wfP lt*

and $\bigwedge \varphi\ n. \varphi \in \text{iterate-sem } n\ (\text{Assume } b;;\ C)\ S \wedge b\ (\text{snd } \varphi) \implies (\exists \sigma'. \text{single-sem } C\ (\text{snd } \varphi)\ \sigma' \wedge (\neg b\ \sigma' \vee \text{lt } (e\ \sigma')\ (e\ (\text{snd } \varphi))))$

shows *terminates-in* (*while-cond* $b\ C)\ S$

proof (*rule terminates-inI*)

fix φ **assume** *asm0*: $\varphi \in S$

let $?S = \{\varphi\}$

let $?R = \{(x, y). \text{lt } x\ y\}$

let $?Q = \{e\ (\text{snd } \varphi) \mid \varphi\ n. b\ (\text{snd } \varphi) \wedge \varphi \in \text{iterate-sem } n\ (\text{Assume } b;;\ C)\ ?S\}$

show $\exists \sigma'. \langle \text{while-cond } b\ C, \text{snd } \varphi \rangle \rightarrow \sigma'$

proof (*cases* $b\ (\text{snd } \varphi)$)

case *False*

then show *?thesis*

by (*metis SemAssume SemSeq SemWhileExit lnot-def while-cond-def*)

next

case *True*

show *?thesis*

proof (*rule wfE-min*)

show *wf ?R*

using *assms(1) wfp-def by blast*

show $e\ (\text{snd } \varphi) \in ?Q$

using *True asm0 iterate-sem.simps(1) by fastforce*

fix z **assume** *asm1*: $z \in ?Q\ (\bigwedge y. (y, z) \in \{(x, y). \text{lt } x\ y\} \implies y \notin ?Q)$

then obtain $\varphi' n$ **where** $z = e\ (\text{snd } \varphi')\ b\ (\text{snd } \varphi')\ \varphi' \in \text{iterate-sem } n\ (\text{Assume } b;;\ C)\ ?S$

by *blast*

then obtain σ' **where** $\text{single-sem } C\ (\text{snd } \varphi')\ \sigma' \wedge (\neg b\ \sigma' \vee \text{lt } (e\ \sigma')\ (e\ (\text{snd } \varphi')))$

using *assms(2) iterate-sem-mono[of ?S S n Assume b;; C]*

by (*meson asm0 empty-subsetI in-mono insert-subsetI*)

then have $\neg b\ \sigma' \vee e\ \sigma' \notin ?Q$

using $\langle z = e\ (\text{snd } \varphi') \rangle$ *asm1(2) by blast*

moreover have $(fst \ \varphi', \sigma') \in sem \ (Assume \ b;; \ C) \ (iterate-sem \ n \ (Assume \ b;; \ C) \ ?S)$
by $(metis \ (no-types, \ opaque-lifting) \ SemAssume \ SemSeq \ \langle(\langle C, \ snd \ \varphi' \rangle \rightarrow \sigma') \wedge (\neg \ b \ \sigma' \vee lt \ (e \ \sigma') \ (e \ (snd \ \varphi')))\rangle \ \langle\varphi' \in iterate-sem \ n \ (Assume \ b;; \ C) \ ?S\rangle \ \langle b \ (snd \ \varphi') \rangle \ single-step-then-in-sem \ surjective-pairing)$
ultimately have $\neg \ b \ \sigma'$
using $iterate-sem.simps(2)[of \ n \ Assume \ b;; \ C \ \{\varphi\}] \ mem-Collect-eq \ snd-conv$
by $(metis \ (mono-tags, \ lifting))$
then have $(fst \ \varphi', \sigma') \in iterate-sem \ (Suc \ n) \ (Assume \ b;; \ C) \ ?S$
by $(simp \ add: \ \langle fst \ \varphi', \sigma' \rangle \in sem \ (Assume \ b;; \ C) \ (iterate-sem \ n \ (Assume \ b;; \ C) \ ?S))$
then have $(fst \ \varphi', \sigma') \in (\bigcup n. \ iterate-sem \ n \ (Assume \ b;; \ C) \ ?S)$ **by** *blast*
then have $(fst \ \varphi', \sigma') \in filter-exp \ (lnot \ b) \ (\bigcup n. \ iterate-sem \ n \ (Assume \ b;; \ C) \ ?S)$
using $filter-exp-def[of \ lnot \ b] \ lnot-def[of \ b \ \sigma'] \ \langle \neg \ b \ \sigma' \rangle$ **by** *force*
then have $(fst \ \varphi', \sigma') \in sem \ (while-cond \ b \ C) \ ?S$
by $(simp \ add: \ assume-sem \ filter-exp-def \ sem-seq \ sem-while \ while-cond-def)$
then show $\exists \sigma'. \ \langle while-cond \ b \ C, \ snd \ \varphi \rangle \rightarrow \sigma'$
by $(metis \ in-sem \ singletonD \ snd-conv)$
qed
qed
qed

lemma *total-hyper-triple-altI*:
assumes $\bigwedge S. \ P \ S \implies Q \ (sem \ C \ S)$
and $\bigwedge S. \ P \ S \implies terminates-in \ C \ S$
shows $\models_{TERM} \ \{P\} \ C \ \{Q\}$
by $(metis \ assms(1) \ assms(2) \ hyper-hoare-tripleI \ terminates-in-def \ total-hyper-triple-def)$

lemma *syntactic-frame-preserved*:
assumes $terminates-in \ C \ S$
and $wr \ C \cap \ fv \ F = \{\}$
and $sat-assertion \ vals \ states \ F \ S$
and $wf-assertion-aux \ nv \ (length \ states) \ F$
shows $sat-assertion \ vals \ states \ F \ (sem \ C \ S)$
using *assms*
proof $(induct \ F \ arbitrary: \ vals \ states \ nv)$
case $(AForallState \ F)$
then have $\bigwedge \varphi. \ \varphi \in sem \ C \ S \implies sat-assertion \ vals \ (\varphi \ \# \ states) \ F \ (sem \ C \ S)$
proof $-$
fix φ **assume** $asm0: \ \varphi \in sem \ C \ S$
then obtain σ **where** $single-sem \ C \ \sigma \ (snd \ \varphi) \ (fst \ \varphi, \ \sigma) \in S$
using *in-sem* **by** *blast*
then have $sat-assertion \ vals \ ((fst \ \varphi, \ \sigma) \ \# \ states) \ F \ (sem \ C \ S)$
using $AForallState.hyps \ AForallState.prem1 \ assms(1)$ **by** *auto*
moreover have $agree-on \ (fv \ F) \ \sigma \ (snd \ \varphi)$
using $AForallState.prem2 \ \langle(\langle C::(nat, \ nat) \ stmt), \ \sigma::nat \Rightarrow nat \rangle \rightarrow snd$

```

( $\varphi :: (\text{nat} \Rightarrow \text{nat}) \times (\text{nat} \Rightarrow \text{nat})$ ) $\rangle$  wr-charact by auto
  moreover have wf-assertion-aux nv (Suc (length states)) F
    using AForallState.premis(4) by auto
  ultimately have sat-assertion vals ((fst  $\varphi$ , snd  $\varphi$ ) # states) F (sem C S)
    using fv-wr-charact[of F  $\sigma$  snd  $\varphi$  vals fst  $\varphi$  states sem C S]
    by fast
  then show sat-assertion vals ( $\varphi$  # states) F (sem C S)
    by simp
qed
then show ?case
  using AForallState.hyps AForallState.premis(2) assms(1) by auto
next
case (AExistsState F)
then obtain  $\varphi$  where asm0:  $\varphi \in S$  sat-assertion vals ( $\varphi$  # states) F S
  by auto
then obtain  $\sigma'$  where single-sem C (snd  $\varphi$ )  $\sigma'$ 
  using assms(1) terminates-in-def by blast
then have sat-assertion vals ((fst  $\varphi$ ,  $\sigma'$ ) # states) F S
  by (metis AExistsState.premis(2) AExistsState.premis(4) asm0(2) fv.simps(6)
fv-wr-charact prod.collapse wf-assertion-aux.simps(8) wr-charact)
then show ?case
  by (metis AExistsState.hyps AExistsState.premis(2) AExistsState.premis(4)  $\langle C$ ,
snd  $\varphi \rangle \rightarrow \sigma'$  asm0(1) assms(1) fv.simps(6) length-Cons prod.collapse sat-assertion.simps(4)
single-step-then-in-sem wf-assertion-aux.simps(8))
qed (fastforce+)

```

```

theorem frame-rule-syntactic:
  assumes  $\models_{\text{TERM}} \{P\} C \{Q\}$ 
    and wr C  $\cap$  fv F = {}
    and wf-assertion F
  shows  $\models_{\text{TERM}} \{\text{conj } P \text{ (interp-assert } F)\} C \{\text{conj } Q \text{ (interp-assert } F)\}$ 
proof (rule total-hyper-tripleI)
  let ?F = interp-assert F
  show  $\bigwedge \varphi S. \text{Logic.conj } P \text{ ?F } S \wedge \varphi \in S \implies \exists \sigma'. \langle C, \text{snd } \varphi \rangle \rightarrow \sigma'$ 
    by (metis assms(1) conj-def total-hyper-triple-def)
  show  $\models \{\text{Logic.conj } P \text{ ?F}\} C \{\text{Logic.conj } Q \text{ ?F}\}$ 
  proof (rule hyper-hoare-tripleI)
    fix S assume asm0: Logic.conj P ?F S
    then have terminates-in C S
    by (simp add:  $\langle \bigwedge \varphi S. \text{Logic.conj } P \text{ ?F } S \wedge \varphi \in S \implies \exists \sigma'. \langle C, \text{snd } \varphi \rangle \rightarrow \sigma' \rangle$ 
terminates-in-def)
    moreover have ?F (sem C S)
    by (metis asm0 assms(2) assms(3) calculation conj-def list.size(3) syntac-
tic-frame-preserved)
    ultimately show Logic.conj Q ?F (sem C S)
    by (metis asm0 assms(1) conj-def hyper-hoare-tripleE total-hyper-triple-def)
qed
qed

```

7.1 Specialize rule

definition *same-syn-sem-all* :: 'a assertion \Rightarrow ((nat, 'a, nat, 'a) state \Rightarrow bool) \Rightarrow bool

where

same-syn-sem-all bsyn bsem \longleftrightarrow
 $(\forall$ states vals S . length states $> 0 \longrightarrow$ bsem (hd states) = sat-assertion vals states bsyn $S)$

lemma *same-syn-sem-allI*:

assumes \bigwedge states vals S . length states $> 0 \implies$ bsem (hd states) \longleftrightarrow sat-assertion vals states bsyn S

shows *same-syn-sem-all* bsyn bsem

by (simp add: assms same-syn-sem-all-def)

lemma *transform-assume-valid*:

assumes *same-syn-sem-all* bsyn bsem

shows sat-assertion vals states A (Set.filter bsem S)

\longleftrightarrow sat-assertion vals states (transform-assume bsyn A) S

proof (induct A arbitrary: vals states)

case (AForallState A)

let $?S =$ Set.filter (bsem) S

let $?A =$ transform-assume bsyn A

have sat-assertion vals states (AForallState A) $?S \longleftrightarrow$ $(\forall \varphi \in ?S$. sat-assertion vals $(\varphi \#$ states) A $?S)$

by force

also have ... \longleftrightarrow $(\forall \varphi \in ?S$. sat-assertion vals $(\varphi \#$ states) $?A$ $S)$

using AForallState **by** presburger

also have ... \longleftrightarrow $(\forall \varphi \in S$. bsem $\varphi \longrightarrow$ sat-assertion vals $(\varphi \#$ states) $?A$ $S)$

by fastforce

also have ... \longleftrightarrow $(\forall \varphi \in S$. sat-assertion vals $(\varphi \#$ states) bsyn $S \longrightarrow$ sat-assertion vals $(\varphi \#$ states) $?A$ $S)$

using assms same-syn-sem-all-def[of bsyn bsem] **by** auto

also have ... \longleftrightarrow $(\forall \varphi \in S$. sat-assertion vals $(\varphi \#$ states) (AImp bsyn $?A$) $S)$

using sat-assertion-Imp **by** fast

also have ... \longleftrightarrow sat-assertion vals states (AForallState (AImp bsyn $?A$)) S

using sat-assertion.simps(2) **by** force

then show $?case$

using calculation transform-assume.simps(1) **by** fastforce

next

case (AExistsState A)

let $?S =$ Set.filter (bsem) S

let $?A =$ transform-assume bsyn A

have sat-assertion vals states (AExistsState A) $?S \longleftrightarrow$ $(\exists \varphi \in ?S$. sat-assertion vals $(\varphi \#$ states) A $?S)$

by force

also have ... \longleftrightarrow $(\exists \varphi \in ?S$. sat-assertion vals $(\varphi \#$ states) $?A$ $S)$

using AExistsState **by** presburger

also have ... \longleftrightarrow $(\exists \varphi \in S$. bsem $\varphi \wedge$ sat-assertion vals $(\varphi \#$ states) $?A$ $S)$

by force

```

also have ...  $\longleftrightarrow (\exists \varphi \in S. \text{sat-assertion vals } (\varphi \# \text{states}) \text{ bsyn } S \wedge \text{sat-assertion}$ 
 $\text{vals } (\varphi \# \text{states}) \text{ ?A } S)$ 
  using assms same-syn-sem-all-def[of bsyn bsem] by auto
also have ...  $\longleftrightarrow (\exists \varphi \in S. \text{sat-assertion vals } (\varphi \# \text{states}) (A \text{And } \text{bsyn } \text{?A}) S)$ 
  by simp
also have ...  $\longleftrightarrow \text{sat-assertion vals states } (A \text{ExistsState } (A \text{And } \text{bsyn } \text{?A})) S$ 
  using sat-assertion.simps(3) by force
then show ?case
  using calculation transform-assume.simps(2) by fastforce
next
  case (AForall A)
  let ?S = Set.filter (bsem) S
  let ?A = transform-assume bsyn A
  have sat-assertion vals states (AForall A) ?S  $\longleftrightarrow (\forall v. \text{sat-assertion } (v \# \text{vals})$ 
 $\text{states } A \text{ ?S})$ 
  by force
also have ...  $\longleftrightarrow (\forall v. \text{sat-assertion } (v \# \text{vals}) \text{states } \text{?A } S)$ 
  using AForall by presburger
also have ...  $\longleftrightarrow \text{sat-assertion vals states } (A \text{Forall } \text{?A}) S$ 
  by simp
then show ?case
  using calculation transform-assume.simps(3) by fastforce
next
  case (AExists A)
  let ?S = Set.filter (bsem) S
  let ?A = transform-assume bsyn A
  have sat-assertion vals states (AExists A) ?S  $\longleftrightarrow (\exists v. \text{sat-assertion } (v \# \text{vals})$ 
 $\text{states } A \text{ ?S})$ 
  by force
also have ...  $\longleftrightarrow (\exists v. \text{sat-assertion } (v \# \text{vals}) \text{states } \text{?A } S)$ 
  using AExists by presburger
also have ...  $\longleftrightarrow \text{sat-assertion vals states } (A \text{Exists } \text{?A}) S$ 
  by simp
then show ?case
  using calculation transform-assume.simps(4) by fastforce
qed (simp-all)

```

fun *indep-of-set* **where**

```

  indep-of-set (AForall A)  $\longleftrightarrow \text{indep-of-set } A$ 
| indep-of-set (AExists A)  $\longleftrightarrow \text{indep-of-set } A$ 
| indep-of-set (AOr A B)  $\longleftrightarrow \text{indep-of-set } A \wedge \text{indep-of-set } B$ 
| indep-of-set (AAnd A B)  $\longleftrightarrow \text{indep-of-set } A \wedge \text{indep-of-set } B$ 
| indep-of-set (AComp - -)  $\longleftrightarrow \text{True}$ 
| indep-of-set (AConst -)  $\longleftrightarrow \text{True}$ 
| indep-of-set (AForallState -)  $\longleftrightarrow \text{False}$ 
| indep-of-set (AExistsState -)  $\longleftrightarrow \text{False}$ 

```

lemma *indep-of-set-charact*:

```

assumes indep-of-set A
  and sat-assertion vals states A S
  shows sat-assertion vals states A S'
using assms
by (induct A arbitrary: vals states) (auto)

lemma wf-exp-take:
  assumes wf-exp nv ns e
  shows interp-exp vals states e = interp-exp (take nv vals) (take ns states) e
  using assms
proof (induct e arbitrary: nv ns vals states)
  case (EQVar x)
  then show ?case
    by force
next
  case (EBinop e1 x2 e2)
  then show ?case
    by (metis interp-exp.simps(5) wf-exp.simps(4))
next
  case (EFun f e)
  then show ?case
    by (metis interp-exp.simps(6) wf-exp.simps(5))
qed (simp-all)

lemma wf-assertion-aux-take:
  assumes wf-assertion-aux nv ns A
  shows sat-assertion vals states A S  $\longleftrightarrow$  sat-assertion (take nv vals) (take ns states) A S
  using assms
proof (induct A arbitrary: nv ns vals states)
  case (AConst x)
  then show ?case
    by simp
next
  case (AComp x1a x2 x3a)
  then show ?case
    by (metis sat-assertion.simps(2) wf-assertion-aux.simps(2) wf-exp-take)
next
  case (AForallState A)
  then show ?case using AForallState.hyps[of nv Suc ns vals]
    by (metis sat-assertion.simps(3) take-Suc-Cons wf-assertion-aux.simps(7))
next
  case (AExistsState A)
  then show ?case using AExistsState.hyps[of nv Suc ns vals]
    by (metis sat-assertion.simps(4) take-Suc-Cons wf-assertion-aux.simps(8))
next
  case (AForall A)
  then show ?case using AForall.hyps[of Suc nv ns - states]
    by (metis sat-assertion.simps(5) take-Suc-Cons wf-assertion-aux.simps(5))

```

```

next
  case (AExists A)
  then show ?case using AExists.hyps[of Suc nv ns - states]
    by (metis sat-assertion.simps(6) take-Suc-Cons wf-assertion-aux.simps(6))
next
  case (AOr A1 A2)
  then show ?case
    by (metis sat-assertion.simps(8) wf-assertion-aux.simps(4))
next
  case (AAnd A1 A2)
  then show ?case
    by (metis sat-assertion.simps(7) wf-assertion-aux.simps(3))
qed

```

lemma *syntactic-charact-for-equivalence*:

```

  assumes indep-of-set A
    and wf-assertion-aux (0::nat) (1::nat) A
  shows sat-assertion vals ( $\varphi \#$  states) A S  $\longleftrightarrow$  sat-assertion [] [ $\varphi$ ] A {} (is ?A
 $\longleftrightarrow$  ?B)
proof -
  have ?A  $\longleftrightarrow$  sat-assertion vals ( $\varphi \#$  states) A {}
    using assms(1) indep-of-set-charact by blast
  also have ...  $\longleftrightarrow$  sat-assertion (take (0::nat) vals) (take (1::nat) ( $\varphi \#$  states))
  A {}
    using wf-assertion-aux-take[of 0 1 A vals  $\varphi \#$  states {}] assms(2)
    by blast
  ultimately show ?thesis by simp
qed

```

definition *get-bsem where*

```

  get-bsem bsyn  $\varphi \longleftrightarrow$  sat-assertion [] [ $\varphi$ ] bsyn {}

```

lemma *syntactic-charact-for-bsem*:

```

  assumes indep-of-set A
    and wf-assertion-aux (0::nat) (1::nat) A
  shows same-syn-sem-all A (get-bsem A)
proof (rule same-syn-sem-allI)
  fix states :: ((nat  $\Rightarrow$  'a)  $\times$  (nat  $\Rightarrow$  'a)) list
  fix vals S
  assume asm0: 0 < length states
  then show get-bsem A (hd states) = sat-assertion vals states A S
    by (metis assms(1) assms(2) get-bsem-def length-greater-0-conv list.collapse
  syntactic-charact-for-equivalence)
qed

```

lemma *get-bsem-is-bsem*:

```

assumes same-syn-sem-all bsyn bsem
shows bsem = get-bsem bsyn
proof (rule ext)
  fix x
  have bsem (hd [x]) = sat-assertion [] [x] bsyn {}
    using assms same-syn-sem-all-def by fastforce
  then show bsem x = get-bsem bsyn x
    by (simp add: get-bsem-def)
qed

```

lemma free-vars-syn-sem:

```

assumes same-syn-sem-all bsyn bsem
  and fst  $\varphi = \text{fst } \varphi'$ 
  and agree-on (fv bsyn) (snd  $\varphi$ ) (snd  $\varphi'$ )
  and bsem  $\varphi$ 
  and wf-assertion-aux 0 (Suc 0) bsyn
shows bsem  $\varphi'$ 
proof –
  have sat-assertion [] (insert-at 0 (fst  $\varphi$ , snd  $\varphi'$ ) []) bsyn {}
    using assms(3)
  proof (rule fv-wr-charact-aux)
    show sat-assertion [] (insert-at 0 (fst  $\varphi$ , snd  $\varphi$ ) []) bsyn {}
      by (metis assms(1) assms(4) get-bsem-def get-bsem-is-bsem insert-at.simps(1)
        prod.collapse)
    show wf-assertion-aux 0 (Suc (length [])) bsyn
      by (simp add: assms(5))
  qed (simp)
  then show ?thesis
    by (metis assms(1) assms(2) get-bsem-def get-bsem-is-bsem insert-at.simps(1)
      prod.collapse)
qed

```

lemma free-vars-charact:

```

assumes wr  $C \cap \text{fv } \text{bsyn} = \{\}$ 
  and same-syn-sem-all bsyn bsem
  and wf-assertion-aux 0 (Suc 0) bsyn
shows sem C (Set.filter bsem S) = Set.filter bsem (sem C S) (is ?A = ?B)
proof
  show ?A  $\subseteq$  ?B
  proof
    fix x assume asm0:  $x \in \text{sem } C (\text{Set.filter bsem } S)$ 
    then obtain  $\sigma$  where  $(\text{fst } x, \sigma) \in \text{Set.filter bsem } S$  single-sem C  $\sigma$  (snd x)
      by (meson in-sem)
    then have agree-on (fv bsyn)  $\sigma$  (snd x)
      using assms(1) wr-charact by blast
    then show  $x \in \text{Set.filter bsem } (\text{sem } C S)$ 
      using  $\langle (\text{fst } x, \sigma) \in \text{Set.filter bsem } S \rangle \langle \langle C, \sigma \rangle \rightarrow \text{snd } x \rangle$  assms(2) assms(3)
  qed

```

```

    free-vars-syn-sem [of bsyn bsem]
  by (cases x) (auto intro: single-step-then-in-sem)
qed
show ?B  $\subseteq$  ?A
proof
  fix x assume asm0: x  $\in$  ?B
  then obtain  $\sigma$  where bsem x (fst x,  $\sigma$ )  $\in$  S single-sem C  $\sigma$  (snd x)
  by (auto simp add: in-sem)
  then have agree-on (fv bsyn)  $\sigma$  (snd x)
  using assms(1) wr-charact by blast
  then have bsem (fst x,  $\sigma$ )
  using  $\langle$ bsem x $\rangle$  agree-on-sym assms(2) assms(3) free-vars-syn-sem by fastforce
  then show x  $\in$  ?A
  using  $\langle$ (fst x,  $\sigma$ )  $\in$  S $\rangle$   $\langle$ C,  $\sigma$  $\rangle$   $\rightarrow$  snd x $\rangle$  by (auto simp add: in-sem)
qed
qed

```

```

lemma filter-rule-semantic:
  assumes  $\models$  {interp-assert P} C {interp-assert Q}
    and same-syn-sem-all bsyn bsem
    and wr C  $\cap$  fv bsyn = {}
    and wf-assertion-aux 0 (Suc 0) bsyn
  shows  $\models$  { interp-assert (transform-assume bsyn P) } C { interp-assert
(transform-assume bsyn Q) }
proof (rule hyper-hoare-tripleI)
  fix S assume asm0: interp-assert (transform-assume bsyn P) S
  then have sat-assertion [] [] P (Set.filter bsem S)
  using TotalLogic.transform-assume-valid assms(2) by blast
  then have sat-assertion [] [] Q (sem C (Set.filter bsem S))
  using assms(1) hyper-hoare-tripleE by blast
  moreover have sem C (Set.filter bsem S) = Set.filter bsem (sem C S)
  using assms(2) assms(3) assms(4) free-vars-charact by presburger
  ultimately show interp-assert (transform-assume bsyn Q) (sem C S)
  using TotalLogic.transform-assume-valid assms(2) by fastforce
qed

```

```

lemma filter-rule-syntactic:
  assumes  $\models$  {interp-assert P} C {interp-assert Q}
    and indep-of-set b
    and wf-assertion-aux 0 1 b
    and wr C  $\cap$  fv b = {}
  shows  $\models$  { interp-assert (transform-assume b P) } C { interp-assert (transform-assume
b Q) }
  using assms(1) filter-rule-semantic
  by (metis One-nat-def assms(2) assms(3) assms(4) syntactic-charact-for-bsem)

```

definition terminates where

$terminates\ C \iff (\forall \sigma. \exists \sigma'. single-sem\ C\ \sigma\ \sigma')$

lemma *terminatesI*:
 assumes $\bigwedge \sigma. \exists \sigma'. single-sem\ C\ \sigma\ \sigma'$
 shows *terminates C*
 using *terminates-def* **assms** **by** *auto*

lemma *terminates-implies-total*:
 assumes $\models \{P\}\ C\ \{Q\}$
 and *terminates C*
 shows $\models_{TERM} \{P\}\ C\ \{Q\}$
 using *assms(1)*
proof (*rule total-hyper-tripleI*)
 fix $\varphi\ S$ **assume** *asm0*: $P\ S \wedge \varphi \in S$
 then show $\exists \sigma'. \langle C, snd\ \varphi \rangle \rightarrow \sigma'$
 by (*metis* *assms(2)* *terminates-def*)
qed

lemma *terminates-seq*:
 assumes *terminates C1*
 and *terminates C2*
 shows *terminates (C1;; C2)*
 by (*meson* *SemSeq* *assms(1)* *assms(2)* *terminates-def*)

lemma *terminates-assign*:
 terminates (Assign x e)
 by (*meson* *SemAssign* *terminates-def*)

lemma *terminates-havoc*:
 terminates (Havoc c)
 by (*meson* *SemHavoc* *terminates-def*)

lemma *terminates-if*:
 assumes *terminates C1*
 and *terminates C2*
 shows *terminates (If C1 C2)*
 by (*meson* *SemIf2* *assms(2)* *terminates-def*)

lemma *rule-lframe-exist*:
 fixes $b :: ('a \Rightarrow ('lvar \Rightarrow 'lval)) \Rightarrow bool$
 — b takes a mapping from keys to logical states (representing the tuple), and
 returns a boolean

assumes $\models_{TERM} \{P\}\ C\ \{Q\}$
 shows $\models_{TERM} \{ conj\ P\ (\lambda S. \exists \varphi. (\forall k. \varphi\ k \in S) \wedge b\ (fst\ \circ\ \varphi))\ }\ C\ \{ conj\ Q\ (\lambda S. \exists \varphi. (\forall k. \varphi\ k \in S) \wedge b\ (fst\ \circ\ \varphi))\ }\}$

proof (*rule total-hyper-tripleI*)

```

fix  $\varphi$   $S$ 
show Logic.conj P ( $\lambda S. \exists \varphi. (\forall k. \varphi k \in S) \wedge b (fst \circ \varphi)$ )  $S \wedge \varphi \in S \implies \exists \sigma'$ .
 $\langle C, snd \varphi \rangle \rightarrow \sigma'$ 
  by (metis (mono-tags, lifting) assms conj-def total-hyper-triple-equiv)
next
show  $\models \{Logic.conj P (\lambda S. \exists \varphi. (\forall k. \varphi k \in S) \wedge b (fst \circ \varphi))\} C \{Logic.conj Q$ 
 $(\lambda S. \exists \varphi. (\forall k. \varphi k \in S) \wedge b (fst \circ \varphi))\}$ 
proof (rule hyper-hoare-tripleI)
  fix  $S :: ('lvar, 'lval, 'b, 'c)$  state set
  assume asm0: Logic.conj P ( $\lambda S. \exists \varphi. (\forall k. \varphi k \in S) \wedge b (fst \circ \varphi)$ )  $S$ 
  then obtain  $\varphi$  where  $(\forall k. \varphi k \in S) \wedge b (fst \circ \varphi)$ 
  using conj-def[of P  $\lambda S. \exists \varphi. (\forall k. \varphi k \in S) \wedge b (fst \circ \varphi)$  S] by blast
  let  $?\sigma = \lambda k. SOME \sigma'. (fst (\varphi k), \sigma') \in sem C S \wedge single-sem C (snd (\varphi k))$ 
 $\sigma'$ 
  let  $?\varphi = \lambda k. (fst (\varphi k), ?\sigma k)$ 
  have  $r: \bigwedge k. (fst (\varphi k), ?\sigma k) \in sem C S \wedge single-sem C (snd (\varphi k)) (?\sigma k)$ 
proof -
  fix  $k$  show  $(fst (\varphi k), ?\sigma k) \in sem C S \wedge single-sem C (snd (\varphi k)) (?\sigma k)$ 
proof (rule someI-ex[of  $\lambda \sigma'. (fst (\varphi k), \sigma') \in sem C S \wedge single-sem C (snd$ 
 $(\varphi k)) \sigma'$ ])
  show  $\exists \sigma'. (fst (\varphi k), \sigma') \in sem C S \wedge \langle C, snd (\varphi k) \rangle \rightarrow \sigma'$ 
  by (metis (mono-tags, lifting)  $\langle (\forall k. \varphi k \in S) \wedge b (fst \circ \varphi) \rangle$  asm0 assms
conj-def total-hyper-triple-equiv)
  qed
qed
moreover have  $fst \circ \varphi = fst \circ ?\varphi$  by (rule ext) simp
ultimately have  $\exists \varphi. (\forall k. \varphi k \in sem C S) \wedge b (fst \circ \varphi)$ 
  using  $\langle (\forall k. \varphi k \in S) \wedge b (fst \circ \varphi) \rangle$  by force
moreover have  $Q (sem C S)$ 
  by (metis (mono-tags, lifting) asm0 assms conj-def hyper-hoare-tripleE total-hyper-triple-def)
ultimately show Logic.conj Q ( $\lambda S. \exists \varphi. (\forall k. \varphi k \in S) \wedge b (fst \circ \varphi)$ )  $(sem C$ 
 $S)$ 
  using conj-def[of Q  $\lambda S. \exists \varphi. (\forall k. \varphi k \in S) \wedge b (fst \circ \varphi)$ ] by simp
qed
qed

lemma terminates-if-then:
  assumes terminates C1
  and terminates C2
  shows terminates (if-then-else b C1 C2)
proof (rule terminatesI)
  fix  $\sigma$ 
  show  $\exists \sigma'. \langle if-then-else b C1 C2, \sigma \rangle \rightarrow \sigma'$ 
proof (cases b  $\sigma$ )
  case True
  then show ?thesis
  by (metis SemAssume SemIf1 SemSeq assms(1) if-then-else-def terminates-def)
next

```

```

    case False
  then show ?thesis
    by (metis (no-types, opaque-lifting) SemAssume SemIf2 SemSeq assms(2)
if-then-else-def lnot-def terminates-def)
  qed
qed

```

definition *min-prop* :: (nat \Rightarrow bool) \Rightarrow nat **where**
min-prop P = (SOME n. P n \wedge (\forall m. m < n \longrightarrow \neg P m))

lemma *min-prop-charact*:
 assumes P n
 shows P (min-prop P) \wedge (\forall m. m < (min-prop P) \longrightarrow \neg P m)
 unfolding min-prop-def
 using min-prop-def[of P] assms exists-least-iff[of P] someI[of λ n. P n \wedge (\forall m.
 m < n \longrightarrow \neg P m)]
 by blast

lemma *hyper-tot-set-not-empty*:
 assumes \models TERM {P} C {Q}
 and P S
 and S \neq {}
 shows sem C S \neq {}
 by (metis assms(1) assms(2) assms(3) ex-in-conv total-hyper-triple-equiv)

lemma *iterate-sem-mod-updates-same*:
 assumes same-mod-updates vars S S'
 shows same-mod-updates vars (iterate-sem n C S) (iterate-sem n C S')
 using assms
proof (induct n)
 case 0
 then show ?case by simp
next
 case (Suc n)
 then show ?case
 by (simp add: sem-update-commute)
qed

theorem *while-synchronized-tot*:
 assumes wfP lt
 and \bigwedge n. not-fv-hyper t (I n)
 and \bigwedge n. \models TERM {conj (conj (I n) (holds-forall b)) (e-recorded-in-t e t)} C
 {conj (conj (I (Suc n)) (low-exp b)) (e-smaller-than-t e t lt)}
 shows \models TERM {conj (I 0) (low-exp b)} while-cond b C {conj (exists I)
 (holds-forall (lnot b))}
proof (rule total-hyper-triple-altI)

```

fix  $S$  assume  $asm0$ :  $conj (I 0) (low-exp\ b) S$ 
let  $?S = \lambda n. assign-exp-to-lvar-set\ e\ t\ (iterate-sem\ n\ (Assume\ b;;\ C)\ S)$ 

have  $S$ -same:  $\bigwedge n. same-mod-updates\ \{t\}\ (?S\ n)\ (iterate-sem\ n\ (Assume\ b;;\ C)\ S)$ 
by (simp add: assign-exp-to-lvar-set-same-mod-updates same-mod-updates-sym)

have  $triple$ :  $\bigwedge n. \models \{conj\ (I\ n)\ (holds-forall\ b)\}\ Assume\ b\ ;;\ C\ \{conj\ (I\ (Suc\ n))\ (low-exp\ b)\}$ 
proof (rule hyper-hoare-tripleI)
  fix  $n\ S$  assume  $conj (I\ n) (holds-forall\ b) S$ 
  let  $?S = assign-exp-to-lvar-set\ e\ t\ S$ 
  have  $conj (I\ n) (holds-forall\ b) ?S$ 
  by (metis (Logic.conj (I\ n) (holds-forall\ b) S) assms(2) conj-def holds-forall-same-assign-lvar not-fv-hyper-assign-exp)
  then have  $conj (conj (I\ n) (holds-forall\ b)) (e-recorded-in-t\ e\ t)\ ?S$ 
  by (simp add: conj-def e-recorded-in-t-if-assigned)
  then have  $Logic.conj (I\ (Suc\ n)) (low-exp\ b) (sem\ (Assume\ b\ ;;\ C)\ ?S)$ 
  by (metis (mono-tags, lifting) assms(3) conj-def hyper-hoare-tripleE sem-assume-low-exp-seq(1) total-hyper-triple-def)
  moreover have  $same-mod-updates\ \{t\}\ (sem\ (Assume\ b\ ;;\ C)\ ?S)\ (sem\ (Assume\ b\ ;;\ C)\ S)$ 
  by (simp add: assign-exp-to-lvar-set-same-mod-updates same-mod-updates-sym sem-update-commute)
  ultimately show  $conj (I\ (Suc\ n)) (low-exp\ b) (sem\ (Assume\ b\ ;;\ C)\ S)$ 
  by (metis assms(2) conj-def low-exp-forall-same-mod-updates not-fv-hyperE)
qed

have  $\bigwedge n. iterate-sem\ n\ (Assume\ b\ ;;\ C)\ S \neq \{\}\ \Longrightarrow\ conj (I\ n) (low-exp\ b) (iterate-sem\ n\ (Assume\ b\ ;;\ C)\ S)$ 
proof –
  fix  $n$ 
  show  $iterate-sem\ n\ (Assume\ b\ ;;\ C)\ S \neq \{\}\ \Longrightarrow\ conj (I\ n) (low-exp\ b) (iterate-sem\ n\ (Assume\ b\ ;;\ C)\ S)$ 
  proof (induct n)
    case  $0$ 
    then show  $?case$ 
    by (simp add: asm0)
  next
    case  $(Suc\ n)$ 
    then show  $?case$ 
    by (metis (full-types) conj-def false-then-empty-later holds-forall-empty hyper-hoare-tripleE iterate-sem.simps(2) lessI low-exp-two-cases triple)
  qed
qed

have  $terminates$ :  $\exists n. iterate-sem\ n\ (Assume\ b;;\ C)\ S = \{\}$ 
proof (rule ccontr)

```

```

assume asm0:  $\neg (\exists n. \text{iterate-sem } n \text{ (Assume } b ;; C) S = \{\})$ 

let ?R =  $\{(x, y). \text{lt } x \ y\}$ 
let ?Q =  $\{e \text{ (snd } \varphi) \mid \varphi n. \varphi \in ?S \ n\}$ 

obtain  $\varphi 0$  where  $\varphi 0 \in ?S \ 0$ 
by (metis all-not-in-conv asm0 false-then-empty-later holds-forall-empty holds-forall-same-assign-lvar lessI)

show False
proof (rule wfE-min)
  show wf ?R
    using assms(1) wfp-def by blast
  show  $e \text{ (snd } \varphi 0) \in ?Q$ 
    using  $\langle \varphi 0 \in \text{assign-exp-to-lvar-set } e \ t \text{ (iterate-sem } 0 \text{ (Assume } b ;; C) S) \rangle$ 
by blast
  fix z assume asm1:  $z \in ?Q \ (\bigwedge y. (y, z) \in \{(x, y). \text{lt } x \ y\} \implies y \notin ?Q)$ 
  then obtain  $\varphi \ n$  where  $z = e \text{ (snd } \varphi) \ \varphi \in ?S \ n$  by blast
  moreover have conj (I n) (holds-forall b) (iterate-sem n (Assume b ;; C) S)
  proof –
    have conj (I n) (low-exp b) (iterate-sem n (Assume b ;; C) S)
      using  $\langle \bigwedge n. \text{iterate-sem } n \text{ (Assume } b ;; C) S \neq \{\} \implies \text{Logic.conj (I n)}$ 
      (low-exp b) (iterate-sem n (Assume b ;; C) S) \rangle asm0 by presburger
    moreover have sem (Assume b ;; C) (iterate-sem n (Assume b ;; C) S)  $\neq$ 
     $\{\}$ 
      using asm0 iterate-sem.simps(2) by blast
    ultimately show ?thesis
      by (metis asm0 conj-def false-then-empty-later lessI low-exp-two-cases)
  qed
  then have conj (conj (I n) (holds-forall b)) (e-recorded-in-t e t) (?S n)
    by (metis (mono-tags, lifting) assms(2) conj-def e-recorded-in-t-if-assigned holds-forall-same-assign-lvar not-fv-hyper-assign-exp)
  then have conj (conj (I (Suc n)) (low-exp b)) (e-smaller-than-t e t lt) (sem
  C (?S n))
    using assms(3) hyper-hoare-tripleE total-hyper-triple-def by blast
  moreover obtain  $\sigma'$  where  $\langle C, \text{snd } \varphi \rangle \rightarrow \sigma'$ 
    by (meson  $\langle \text{Logic.conj (Logic.conj (I n) (holds-forall b)) (e-recorded-in-t e$ 
     $t) \text{ (assign-exp-to-lvar-set } e \ t \text{ (iterate-sem } n \text{ (Assume } b ;; C) S) \rangle$  assms(3) calcula-
    tion(2) total-hyper-triple-def)
  then have  $(\text{fst } \varphi, \sigma') \in \text{sem (Assume } b ;; C) (?S \ n)$ 
    by (metis  $\langle \text{Logic.conj (Logic.conj (I n) (holds-forall b)) (e-recorded-in-t e$ 
     $t) \text{ (assign-exp-to-lvar-set } e \ t \text{ (iterate-sem } n \text{ (Assume } b ;; C) S) \rangle$  calculation(2)
    conj-def prod.collapse sem-assume-low-exp-seq(1) single-step-then-in-sem)
  then have lt (e  $\sigma'$ ) z
    by (metis (no-types, lifting)  $\langle \text{Logic.conj (Logic.conj (I n) (holds-forall$ 
     $b)) (e-recorded-in-t e t) \text{ (assign-exp-to-lvar-set } e \ t \text{ (iterate-sem } n \text{ (Assume } b ;;$ 
     $C) S) \rangle$  calculation(1) calculation(2) calculation(3) conj-def e-recorded-in-t-def
    e-smaller-than-t-def fst-conv sem-assume-low-exp-seq(1) snd-conv)

```

```

moreover obtain  $\sigma$  where  $(fst\ \varphi, \sigma) \in ?S\ n\ single\text{-}sem\ (Assume\ b\ ;;\ C)\ \sigma\ \sigma'$ 
  by  $(metis\ \langle fst\ \varphi, \sigma' \rangle \in sem\ (Assume\ b\ ;;\ C)\ (assign\text{-}exp\text{-}to\text{-}lvar\text{-}set\ e\ t$ 
 $(iterate\text{-}sem\ n\ (Assume\ b\ ;;\ C)\ S))\ \rangle\ fst\text{-}conv\ in\text{-}sem\ snd\text{-}conv)$ 
  then obtain  $l$  where  $(l, \sigma) \in iterate\text{-}sem\ n\ (Assume\ b\ ;;\ C)\ S$ 
  using  $assign\text{-}exp\text{-}to\text{-}lvar\text{-}def[of\ e\ t]\ assign\text{-}exp\text{-}to\text{-}lvar\text{-}set\text{-}def[of\ e\ t]\ image\text{-}iff$ 
 $prod.collapse\ snd\text{-}conv$ 
  by  $fastforce$ 
  then have  $(l, \sigma') \in iterate\text{-}sem\ (Suc\ n)\ (Assume\ b\ ;;\ C)\ S$ 
  by  $(metis\ \langle Assume\ b\ ;;\ C, \sigma \rangle \rightarrow \sigma')\ iterate\text{-}sem.simps(2)\ single\text{-}step\text{-}then\text{-}in\text{-}sem)$ 
  then have  $assign\text{-}exp\text{-}to\text{-}lvar\ e\ t\ (l, \sigma') \in ?S\ (Suc\ n)$ 
  by  $(simp\ add:\ assign\text{-}exp\text{-}to\text{-}lvar\text{-}set\text{-}def)$ 
  then have  $e\ \sigma' \in ?Q$ 
  by  $(metis\ (mono\text{-}tags,\ lifting)\ CollectI\ same\text{-}outside\text{-}set\text{-}lvar\text{-}assign\text{-}exp$ 
 $snd\text{-}conv)$ 
  ultimately show  $False$ 
  using  $asm1(2)$  by  $blast$ 
qed
qed

show  $conj\ (exists\ I)\ (holds\text{-}forall\ (lnot\ b))\ (sem\ (while\text{-}cond\ b\ C)\ S)$ 
proof  $-$ 
  let  $?n\text{-}emp = min\text{-}prop\ (\lambda n.\ iterate\text{-}sem\ n\ (Assume\ b\ ;;\ C)\ S = \{\})$ 
  have  $rr:\ iterate\text{-}sem\ ?n\text{-}emp\ (Assume\ b\ ;;\ C)\ S = \{\} \wedge (\forall m < ?n\text{-}emp.\ iterate\text{-}sem$ 
 $m\ (Assume\ b\ ;;\ C)\ S \neq \{\})$ 
  using  $min\text{-}prop\text{-}charact\ terminates\ by\ fast$ 
  show  $?thesis$ 
proof  $(cases\ ?n\text{-}emp)$ 
  case  $0$ 
  then have  $S = \{\}$ 
  using  $rr$  by  $auto$ 
  then show  $?thesis$ 
  by  $(metis\ Loops.exists\text{-}def\ asm0\ conj\text{-}def\ holds\text{-}forall\text{-}empty\ sem\text{-}assume\text{-}low\text{-}exp\text{-}seq(2))$ 
sem\text{-}seq)
next
  case  $(Suc\ k)$ 
  then have  $iterate\text{-}sem\ k\ (Assume\ b\ ;;\ C)\ S \neq \{\}$ 
  using  $lessI\ rr$  by  $presburger$ 
  then have  $conj\ (I\ k)\ (low\text{-}exp\ b)\ (iterate\text{-}sem\ k\ (Assume\ b\ ;;\ C)\ S)$ 
  by  $(simp\ add:\ \langle \bigwedge n.\ iterate\text{-}sem\ n\ (Assume\ b\ ;;\ C)\ S \neq \{\} \implies Logic.conj\ (I$ 
 $n)\ (low\text{-}exp\ b)\ (iterate\text{-}sem\ n\ (Assume\ b\ ;;\ C)\ S) \rangle)$ 
  moreover have  $holds\text{-}forall\ (lnot\ b)\ (iterate\text{-}sem\ k\ (Assume\ b\ ;;\ C)\ S)$ 
  proof  $(rule\ ccontr)$ 
  assume  $asm2:\ \neg\ holds\text{-}forall\ (lnot\ b)\ (iterate\text{-}sem\ k\ (Assume\ b\ ;;\ C)\ S)$ 
  then have  $holds\text{-}forall\ b\ (iterate\text{-}sem\ k\ (Assume\ b\ ;;\ C)\ S)$ 
  by  $(metis\ calculation\ conj\text{-}def\ low\text{-}exp\text{-}two\text{-}cases)$ 
  let  $?S = assign\text{-}exp\text{-}to\text{-}lvar\text{-}set\ e\ t\ (iterate\text{-}sem\ k\ (Assume\ b\ ;;\ C)\ S)$ 
  have  $conj\ (conj\ (I\ k)\ (holds\text{-}forall\ b))\ (e\text{-}recorded\text{-}in\text{-}t\ e\ t)\ ?S$ 
  by  $(metis\ (no\text{-}types,\ lifting)\ \langle holds\text{-}forall\ b\ (iterate\text{-}sem\ k\ (Assume\ b$ 

```

```

;; C) S)› assign-exp-to-lvar-set-same-mod-updates assms(2) calculation conj-def
e-recorded-in-t-if-assigned holds-forall-same-mod-updates not-fv-hyperE)
  moreover have sem (Assume b) ?S = ?S
    by (metis calculation conj-def sem-assume-low-exp(1))
  ultimately have sem (Assume b;; C) ?S ≠ {}
    by (metis asm2 assms(3) holds-forall-empty holds-forall-same-assign-lvar
hyper-tot-set-not-empty sem-seq)

  moreover have same-mod-updates {t} (sem (Assume b;; C) ?S) (assign-exp-to-lvar-set
e t (iterate-sem ?n-emp (Assume b;; C) S))
    by (metis Suc assign-exp-to-lvar-set-def assign-exp-to-lvar-set-same-mod-updates
image-empty iterate-sem.simps(2) rr same-mod-updates-sym sem-update-commute)
  ultimately show False
    by (metis assign-exp-to-lvar-set-def image-empty rr same-mod-updates-empty
same-mod-updates-sym)
  qed
  then have ∃ n. holds-forall (lnot b) (iterate-sem n (Assume b;; C) S) ∧
iterate-sem n (Assume b;; C) S ≠ {}
    ∧ (∀ m. m < n → ¬ (holds-forall (lnot b) (iterate-sem m (Assume b;; C) S)
∧ iterate-sem m (Assume b;; C) S ≠ {}))
    using exists-least-iff[of λn. holds-forall (lnot b) (iterate-sem n (Assume b;;
C) S) ∧ iterate-sem n (Assume b;; C) S ≠ {}]
    using ‹iterate-sem k (Assume b;; C) S ≠ {}› by blast
  then obtain n where def-n: holds-forall (lnot b) (iterate-sem n (Assume b;;
C) S) iterate-sem n (Assume b;; C) S ≠ {}
    ∧ m. m < n ⇒ ¬ (holds-forall (lnot b) (iterate-sem m (Assume b;; C) S)
∧ iterate-sem m (Assume b;; C) S ≠ {}) by blast
  moreover have (∀ m. m < n → holds-forall b (iterate-sem m (Assume b;;
C) S)) ∧ holds-forall (lnot b) (iterate-sem n (Assume b;; C) S)

    by (metis ‹∧ n. iterate-sem n (Assume b;; C) S ≠ {} ⇒ Logic.conj
(I n) (low-exp b) (iterate-sem n (Assume b;; C) S)› conj-def def-n(1) def-n(2)
false-then-empty-later holds-forall-empty low-exp-two-cases)
  then have sem (while-cond b C) S = iterate-sem n (Assume b;; C) S
    using triple while-synchronized-case-1 asm0 by blast
  ultimately show ?thesis
    by (metis Loops.exists-def ‹∧ n. iterate-sem n (Assume b;; C) S ≠ {} ⇒
Logic.conj (I n) (low-exp b) (iterate-sem n (Assume b;; C) S)› conj-def)
  qed
qed

show terminates-in (while-cond b C) S
proof (rule terminates-in-while-loop)
  show wfP lt
    by (simp add: assms(1))
  fix φ n
  assume asm1: φ ∈ iterate-sem n (Assume b;; C) S ∧ b (snd φ)

```

```

let ?S = iterate-sem n (Assume b ;; C) S
have conj (I n) (low-exp b) ?S
  using ⟨ $\bigwedge n. \text{iterate-sem } n \text{ (Assume } b ;; C) S \neq \{\}$   $\implies$  Logic.conj (I n) (low-exp
b) (iterate-sem n (Assume b ;; C) S)⟩ asm1 by blast
  then have conj (I n) (holds-forall b) ?S
    by (metis asm1 conj-def holds-forall-def lnot-def low-exp-two-cases)
  let ?SS = assign-exp-to-lvar-set e t ?S
  have conj (conj (I n) (holds-forall b)) (e-recorded-in-t e t) ?SS
    by (metis (no-types, lifting) ⟨Logic.conj (I n) (holds-forall b) (iterate-sem
n (Assume b ;; C) S)⟩ assign-exp-to-lvar-set-same-mod-updates assms(2) conj-def
e-recorded-in-t-if-assigned holds-forall-same-mod-updates not-fv-hyperE)
  then have conj (conj (I (Suc n)) (low-exp b)) (e-smaller-than-t e t lt) (sem C
?SS)
    using assms(3) hyper-hoare-tripleE total-hyper-triple-def by blast
  moreover have assign-exp-to-lvar e t  $\varphi \in$  ?SS
    by (simp add: asm1 assign-exp-to-lvar-set-def)
  then obtain  $\sigma'$  where ⟨C, snd (assign-exp-to-lvar e t  $\varphi$ )⟩  $\rightarrow$   $\sigma'$ 
    by (meson ⟨Logic.conj (Logic.conj (I n) (holds-forall b)) (e-recorded-in-t e
t) (assign-exp-to-lvar-set e t (iterate-sem n (Assume b ;; C) S))⟩ assms(3) total-hyper-triple-def)
  then have (⟨C, snd  $\varphi$ ⟩  $\rightarrow$   $\sigma'$ )  $\wedge$  (fst (assign-exp-to-lvar e t  $\varphi$ ),  $\sigma'$ )  $\in$  sem C
?SS
    by (metis ⟨assign-exp-to-lvar e t  $\varphi \in$  assign-exp-to-lvar-set e t (iterate-sem
n (Assume b ;; C) S)⟩ assign-exp-to-lvar-def prod.collapse single-step-then-in-sem
snd-conv)
  then have lt (e  $\sigma'$ ) (fst (fst (assign-exp-to-lvar e t  $\varphi$ ),  $\sigma'$ ) t)
    by (metis calculation conj-def e-smaller-than-t-def snd-conv)
  then show  $\exists \sigma'. (\langle C, \text{snd } \varphi \rangle \rightarrow \sigma') \wedge (\neg b \sigma' \vee \text{lt } (e \sigma') (e (\text{snd } \varphi)))$ 
    by (metis ⟨(⟨C, snd  $\varphi$ ⟩  $\rightarrow$   $\sigma'$ )  $\wedge$  (fst (assign-exp-to-lvar e t  $\varphi$ ),  $\sigma'$ )  $\in$  sem C
(assign-exp-to-lvar-set e t (iterate-sem n (Assume b ;; C) S))⟩ assign-exp-to-lvar-def
fst-conv fun-upd-same)
  qed
qed

```

lemma *total-consequence-rule*:

```

assumes entails P P'
and entails Q' Q
  and  $\models_{\text{TERM}} \{P'\} C \{Q'\}$ 
  shows  $\models_{\text{TERM}} \{P\} C \{Q\}$ 
proof (rule total-hyper-tripleI)
  show  $\models \{P\} C \{Q\}$ 
    using assms(1) assms(2) assms(3) consequence-rule total-hyper-triple-def by
blast
  fix  $\varphi$  S show P S  $\wedge \varphi \in S \implies \exists \sigma'. \langle C, \text{snd } \varphi \rangle \rightarrow \sigma'$ 
    by (meson assms(1) assms(3) entailsE total-hyper-triple-def)
qed

```

theorem *WhileSyncTot*:

```

assumes wfP lt

```

and *not-fv-hyper* t I
and $\models_{\text{TERM}} \{ \text{conj } I (\lambda S. \forall \varphi \in S. b (\text{snd } \varphi) \wedge \text{fst } \varphi t = e (\text{snd } \varphi)) \} C \{ \text{conj} \\ (\text{conj } I (\text{low-exp } b)) (\text{e-smaller-than-t } e t \text{ lt}) \}$
shows $\models_{\text{TERM}} \{ \text{conj } I (\text{low-exp } b) \} \text{while-cond } b C \{ \text{conj } I (\text{holds-forall } (\text{lnot} \\ b)) \}$
proof –
define I' **where** $I' = (\lambda(n::\text{nat}). I)$
have $\models_{\text{TERM}} \{ \text{conj } (\text{conj } I (\text{holds-forall } b)) (\text{e-recorded-in-t } e t) \} C \{ \text{conj } (\text{conj} \\ I (\text{low-exp } b)) (\text{e-smaller-than-t } e t \text{ lt}) \}$
proof (*rule total-consequence-rule*)
show $\models_{\text{TERM}} \{ \text{conj } I (\lambda S. \forall \varphi \in S. b (\text{snd } \varphi) \wedge \text{fst } \varphi t = e (\text{snd } \varphi)) \} C \{ \text{conj} \\ (\text{conj } I (\text{low-exp } b)) (\text{e-smaller-than-t } e t \text{ lt}) \}$
using *assms(3)* **by** *blast*
show *entails* (*Logic.conj* (*Logic.conj* $I (\text{holds-forall } b)) (\text{e-recorded-in-t } e t))$
(*Logic.conj* $I (\lambda S. \forall \varphi \in S. b (\text{snd } \varphi) \wedge \text{fst } \varphi t = e (\text{snd } \varphi))$)
by (*simp add: conj-def e-recorded-in-t-def entails-def holds-forall-def*)
qed (*simp add: entails-refl*)
then have $\models_{\text{TERM}} \{ \text{Logic.conj } (I' 0) (\text{low-exp } b) \} \text{while-cond } b C \{ \text{Logic.conj} \\ (\text{Loops.exists } I') (\text{holds-forall } (\text{lnot } b)) \}$
using *while-synchronized-tot[of lt t I' b e C]* I' -*def* **assms** **by** *blast*
then show *?thesis* **using** I' -*def*
by (*simp add: Loops.exists-def conj-def hyper-hoare-triple-def total-hyper-triple-def*)
qed

lemma *total-hyper-tripleE*:
assumes $\models_{\text{TERM}} \{ P \} C \{ Q \}$
and $P S$
and $\varphi \in S$
shows $\exists \sigma'. (\text{fst } \varphi, \sigma') \in \text{sem } C S \wedge \text{single-sem } C (\text{snd } \varphi) \sigma'$
by (*meson assms(1) assms(2) assms(3) total-hyper-triple-equiv*)

theorem *normal-while-tot*:
assumes $\bigwedge n. \models \{ P n \} \text{Assume } b \{ Q n \}$
and $\bigwedge n. \models_{\text{TERM}} \{ \text{conj } (Q n) (\text{e-recorded-in-t } e t) \} C \{ \text{conj } (P (\text{Suc } n)) \\ (\text{e-smaller-than-t } e t \text{ lt}) \}$
and $\models \{ \text{natural-partition } P \} \text{Assume } (\text{lnot } b) \{ R \}$

and *wfP* lt
and $\bigwedge n. \text{not-fv-hyper } t (P n)$
and $\bigwedge n. \text{not-fv-hyper } t (Q n)$

shows $\models_{\text{TERM}} \{ P 0 \} \text{while-cond } b C \{ R \}$
proof (*rule total-hyper-triple-altI*)
fix S **assume** *asm0*: $P 0 S$
have $\bigwedge n. P n (\text{iterate-sem } n (\text{Assume } b;; C) S)$
proof (*rule indexed-invariant-then-power*)

```

fix n
have  $\models \{Q\ n\}\ C\ \{P\ (Suc\ n)\}$ 
proof (rule hyper-hoare-tripleI)
  fix S assume asm1: Q n S
  let ?S = assign-exp-to-lvar-set e t S
  have conj (Q n) (e-recorded-in-t e t) ?S
  by (metis asm1 assms(6) conj-def e-recorded-in-t-if-assigned not-fv-hyper-assign-exp)
  then have conj (P (Suc n)) (e-smaller-than-t e t lt) (sem C ?S)
    using assms(2) hyper-hoare-tripleE total-hyper-triple-def by blast
  then show P (Suc n) (sem C S)
    using assign-exp-to-lvar-set-same-mod-updates[of t S e] assms(5) conj-def
    not-fv-hyperE same-mod-updates-sym[of {t}] sem-update-commute[of {t}]
    by metis
  qed
  then show  $\models \{P\ n\}\ Assume\ b\ ;;\ C\ \{P\ (Suc\ n)\}$ 
    using assms(1) seq-rule total-hyper-triple-def by blast
  qed (simp add: asm0)
  then have natural-partition P ( $\bigcup n.$  iterate-sem n (Assume b ;; C) S)
    by (simp add: natural-partitionI)
  then show R (sem (while-cond b C) S)
    by (metis (no-types, lifting) SUP-cong assms(3) hyper-hoare-triple-def sem-seq
    sem-while while-cond-def)

show terminates-in (while-cond b C) S
proof (rule terminates-in-while-loop)
  show wfP lt
    by (simp add: assms(4))
  fix  $\varphi$  n
  assume asm1:  $\varphi \in$  iterate-sem n (Assume b ;; C) S  $\wedge$  b (snd  $\varphi$ )

  let ?S = assign-exp-to-lvar-set e t (iterate-sem n (Assume b ;; C) S)
  let ? $\varphi$  = assign-exp-to-lvar e t  $\varphi$ 
  have conj (P n) (e-recorded-in-t e t) ?S
    by (metis  $\langle \bigwedge n.$  P n (iterate-sem n (Assume b ;; C) S)  $\rangle$  assms(5) conj-def
    e-recorded-in-t-if-assigned not-fv-hyper-assign-exp)
  then have conj (Q n) (e-recorded-in-t e t) (sem (Assume b) ?S)
    using assms(1)
  by (metis (mono-tags, lifting) IntE conj-def e-recorded-in-t-def hyper-hoare-tripleE
    sem-assume-setify)
  then have conj (P (Suc n)) (e-smaller-than-t e t lt) (sem C (sem (Assume b)
  ?S))
    using assms(2) hyper-hoare-tripleE total-hyper-triple-def by blast
  moreover have ? $\varphi \in$  (sem (Assume b) ?S)
    using asm1
  by (simp add: assign-exp-to-lvar-set-def assume-sem same-outside-set-lvar-assign-exp
  [of  $\varphi$  e t])
  then obtain  $\sigma'$  where (fst ? $\varphi$ ,  $\sigma'$ )  $\in$  sem C (sem (Assume b) ?S)  $\wedge$   $\langle C$ , snd
  ? $\varphi$   $\rangle \rightarrow \sigma'$ 
    using total-hyper-tripleE assms(2)[of n]

```

using $\langle \text{Logic.conj } (Q \ n) \ (e\text{-recorded-in-}t \ e \ t) \ (\text{sem } (\text{Assume } b) \ (\text{assign-exp-to-lvar-set } e \ t \ (\text{iterate-sem } n \ (\text{Assume } b \ ;; \ C) \ S))) \rangle$ **by** *blast*
then have $lt \ (e \ (\text{snd } (\text{fst } ?\varphi, \sigma'))) \ (\text{fst } (\text{fst } ?\varphi, \sigma') \ t)$
using *calculation*
by $(\text{auto simp add: conj-def e-smaller-than-t-def})$
ultimately show $\exists \sigma'. \langle C, \text{snd } \varphi \rangle \rightarrow \sigma' \wedge (\neg b \ \sigma' \vee lt \ (e \ \sigma') \ (e \ (\text{snd } \varphi)))$
by $(\text{metis } \langle \text{fst } (\text{assign-exp-to-lvar } e \ t \ \varphi), \sigma' \rangle \in \text{sem } C \ (\text{sem } (\text{Assume } b) \ (\text{assign-exp-to-lvar-set } e \ t \ (\text{iterate-sem } n \ (\text{Assume } b \ ;; \ C) \ S))) \wedge \langle C, \text{snd } (\text{assign-exp-to-lvar } e \ t \ \varphi) \rangle \rightarrow \sigma')$ *assign-exp-to-lvar-def fst-conv fun-upd-same snd-conv*
qed
qed

definition *e-smaller-than-t-weaker* **where**

e-smaller-than-t-weaker $e \ t \ u \ lt \ S \longleftrightarrow (\forall \varphi \in S. \exists \varphi' \in S. \text{fst } \varphi \ u = \text{fst } \varphi' \ u \wedge lt \ (e \ (\text{snd } \varphi)) \ (\text{fst } \varphi' \ t))$

lemma *exists-terminates-loop:*

assumes *wfP lt*
and $\bigwedge v. \models \{ (\lambda S. \exists \varphi \in S. e \ (\text{snd } \varphi) = v \wedge b \ (\text{snd } \varphi) \wedge P \ \varphi \ S) \}$ *if-then* $b \ C$
 $\{ (\lambda S. \exists \varphi \in S. lt \ (e \ (\text{snd } \varphi)) \ v \wedge P \ \varphi \ S) \}$
and $\bigwedge \varphi. \models \{ P \ \varphi \}$ *while-cond* $b \ C \ \{ Q \ \varphi \}$
shows $\models \{ (\lambda S. \exists \varphi \in S. P \ \varphi \ S) \}$ *while-cond* $b \ C \ \{ (\lambda S. \exists \varphi \in S. Q \ \varphi \ S) \}$
proof $(\text{rule hyper-hoare-tripleI})$
fix S **assume** *asm0*: $\exists \varphi \in S. P \ \varphi \ S$
then obtain φ **where** $\varphi \in S \ P \ \varphi \ S$ **by** *blast*
show $\exists \varphi \in \text{sem } (\text{while-cond } b \ C) \ S. Q \ \varphi \ (\text{sem } (\text{while-cond } b \ C) \ S)$
proof $(\text{cases } b \ (\text{snd } \varphi))$
case *True*

let $?R = \{(x, y). lt \ x \ y\}$

let $?Q = \{ e \ (\text{snd } \varphi') \mid \varphi' \ n. \varphi' \in \text{iterate-sem } n \ (\text{if-then } b \ C) \ S \wedge b \ (\text{snd } \varphi') \wedge P \ \varphi' \ (\text{iterate-sem } n \ (\text{if-then } b \ C) \ S) \}$

have *main-res*: $\exists n \ \varphi'. \varphi' \in \text{iterate-sem } n \ (\text{if-then } b \ C) \ S \wedge \neg b \ (\text{snd } \varphi') \wedge P \ \varphi' \ (\text{iterate-sem } n \ (\text{if-then } b \ C) \ S)$

proof (rule wfE-min)

show *wf ?R*

using *assms(1) wfp-def* **by** *blast*

show $e \ (\text{snd } \varphi) \in ?Q$

using *True* $\langle P \ \varphi \ S \rangle \langle \varphi \in S \rangle$ *iterate-sem.simps(1)* **by** *fastforce*

fix z

assume *asm1*: $z \in ?Q \wedge y. (y, z) \in ?R \implies y \notin ?Q$

then obtain $n \ \varphi'$ **where** $\varphi' \in \text{iterate-sem } n \ (\text{if-then } b \ C) \ S \wedge b \ (\text{snd } \varphi') \wedge P \ \varphi' \ (\text{iterate-sem } n \ (\text{if-then } b \ C) \ S) \wedge z = e \ (\text{snd } \varphi')$

by *blast*

then have $(\lambda S. \exists \varphi \in S. (b \ (\text{snd } \varphi) \longrightarrow lt \ (e \ (\text{snd } \varphi)) \ z) \wedge P \ \varphi \ S) \ (\text{sem } (\text{if-then } b \ C) \ ((\text{iterate-sem } n \ (\text{if-then } b \ C) \ S)))$

using *assms(2) hyper-hoare-tripleE* **by** *blast*
then obtain φ'' **where** $(b \text{ (snd } \varphi'') \longrightarrow \text{lt } (e \text{ (snd } \varphi'')) z) \wedge P \varphi''$ (*sem (if-then b C) ((iterate-sem n (if-then b C) S))*)
 $\varphi'' \in \text{sem (if-then b C) ((iterate-sem n (if-then b C) S))}$
by *blast*
then have $\neg b \text{ (snd } \varphi')$
by (*metis (mono-tags, lifting) asm1(2) case-prodI iterate-sem.simps(2) mem-Collect-eq*)
then show $\exists n \varphi'. \varphi' \in \text{iterate-sem } n \text{ (if-then b C) } S \wedge \neg b \text{ (snd } \varphi') \wedge P \varphi'$ (*iterate-sem n (if-then b C) S*)
by (*metis (b (snd } \varphi'') \longrightarrow \text{lt } (e \text{ (snd } \varphi'')) z) \wedge P \varphi'' (*sem (if-then b C) (iterate-sem n (if-then b C) S)*) $\rangle \langle \varphi'' \in \text{sem (if-then b C) (iterate-sem n (if-then b C) S) \rangle$ *iterate-sem.simps(2)*)
qed
then obtain $n \varphi'$ **where** $\varphi' \in \text{iterate-sem } n \text{ (if-then b C) } S \wedge \neg b \text{ (snd } \varphi') \wedge P \varphi'$ (*iterate-sem n (if-then b C) S*)
by *blast*
then have $\exists \varphi \in \text{sem (while-cond b C) (iterate-sem n (if-then b C) S). } Q \varphi$ (*sem (while-cond b C) (iterate-sem n (if-then b C) S)*)
using *while-exists[of P b C Q] assms(3) hyper-hoare-tripleE* **by** *blast*
then show $\exists \varphi \in \text{sem (while-cond b C) } S. Q \varphi$ (*sem (while-cond b C) S*)
by (*simp add: unroll-while-sem*)
next
case *False*
then show *?thesis*
using *while-exists[of P b C Q] assms(3) hyper-hoare-tripleE* $\langle P \varphi S \rangle \langle \varphi \in S \rangle$
by *blast*
qed
qed*

definition *t-closed* **where**

$t\text{-closed } P \text{ } P\text{-inf} \longleftrightarrow (\forall S. \text{converges-sets } S \wedge (\forall n. P \ n \ (S \ n)) \longrightarrow P\text{-inf } (\bigcup n. S \ n))$

lemma *t-closedE*:

assumes *t-closed* $P \text{ } P\text{-inf}$
and *converges-sets* S
and $\bigwedge n. P \ n \ (S \ n)$
shows $P\text{-inf } (\bigcup n. S \ n)$
using *TotalLogic.t-closed-def* *assms(1) assms(2) assms(3)* **by** *blast*

7.2 Total version of core rules

lemma *total-skip-rule*:

$\models_{\text{TERM}} \{P\} \text{Skip } \{P\}$
by (*meson SemSkip skip-rule total-hyper-triple-def*)

lemma *total-seq-rule*:

assumes $\models_{\text{TERM}} \{P\} \text{C1 } \{R\}$

and $\models_{TERM} \{R\} C2 \{Q\}$
shows $\models_{TERM} \{P\} Seq C1 C2 \{Q\}$
proof (*rule total-hyper-tripleI*)
show $\models \{P\} C1 ;; C2 \{Q\}$
using *assms(1) assms(2) seq-rule total-hyper-triple-def* **by** *blast*
fix φS **assume** $P S \wedge \varphi \in S$
then obtain σ' **where** $\langle C1, snd \varphi \rangle \rightarrow \sigma' (fst \varphi, \sigma') \in sem C1 S$
using *assms(1) total-hyper-tripleE* **by** *blast*
then obtain σ'' **where** $\langle C2, \sigma' \rangle \rightarrow \sigma''$
using *assms*
by (*metis (full-types) $\langle P S \wedge \varphi \in S \rangle$ hyper-hoare-tripleE snd-conv total-hyper-triple-def*)
then show $\exists \sigma''. \langle C1 ;; C2, snd \varphi \rangle \rightarrow \sigma''$
using *SemSeq $\langle \langle C1, snd \varphi \rangle \rightarrow \sigma' \rangle$* **by** *blast*
qed

lemma *total-if-rule:*

assumes $\models_{TERM} \{P\} C1 \{Q1\}$
and $\models_{TERM} \{P\} C2 \{Q2\}$
shows $\models_{TERM} \{P\} If C1 C2 \{join Q1 Q2\}$
proof (*rule total-hyper-tripleI*)
show $\models \{P\} stmt.If C1 C2 \{join Q1 Q2\}$
using *assms(1) assms(2) if-rule total-hyper-triple-equiv* **by** *blast*
fix φS **assume** $P S \wedge \varphi \in S$
then show $\exists \sigma'. \langle stmt.If C1 C2, snd \varphi \rangle \rightarrow \sigma'$
using *SemIf1 assms(1) total-hyper-tripleE* **by** *blast*
qed

lemma *total-rule-exists:*

assumes $\bigwedge x. \models_{TERM} \{P x\} C \{Q x\}$
shows $\models_{TERM} \{exists P\} C \{exists Q\}$
using *total-hyper-tripleI[of exists P C exists Q]*
by (*metis (mono-tags, lifting) Loops.exists-def assms hyper-hoare-triple-def total-hyper-triple-def*)

lemma *total-assign-rule:*

$\models_{TERM} \{ (\lambda S. P \{ (l, \sigma(x := e \sigma)) \mid l \sigma. (l, \sigma) \in S \}) \} (Assign x e) \{P\}$
using *total-hyper-tripleI[of - Assign x e P]*
using *SemAssign assign-rule* **by** *fastforce*

lemma *total-havoc-rule:*

$\models_{TERM} \{ (\lambda S. P \{ (l, \sigma(x := v)) \mid l \sigma v. (l, \sigma) \in S \}) \} (Havoc x) \{P\}$
using *total-hyper-tripleI[of - Havoc x P]*
using *SemHavoc havoc-rule* **by** *fastforce*

lemma *in-semI:*

assumes $\varphi \in S$
and $fst \varphi = fst \varphi'$

```

    and single-sem C (snd  $\varphi$ ) (snd  $\varphi'$ )
    shows  $\varphi' \in \text{sem } C S$ 
    using assms
    by (metis (no-types, lifting) prod.collapse single-step-then-in-sem)

theorem normal-while-tot-stronger:
  fixes  $P :: \text{nat} \Rightarrow ('lvar, 'lval, 'pvar, 'pval) \text{ state set} \Rightarrow \text{bool}$ 

  assumes  $\bigwedge n. \models \{P\ n\} \text{ Assume } b \{Q\ n\}$ 
    and  $\bigwedge n. \models \text{TERM } \{conj\ (Q\ n)\ (e\text{-recorded-in-}t\ e\ t)\} C \{conj\ (P\ (Suc\ n))\}$ 
    (e-smaller-than-t-weaker e t u lt)
    and  $\models \{natural\text{-partition } P\} \text{ Assume } (\text{lnot } b) \{R\}$ 

    and wfP lt
    and  $\bigwedge n. \text{not-fv-hyper } t (P\ n)$ 
    and  $\bigwedge n. \text{not-fv-hyper } t (Q\ n)$ 

    and  $\bigwedge n. \text{not-fv-hyper } u (P\ n)$ 
    and  $\bigwedge n. \text{not-fv-hyper } u (Q\ n)$ 

    and (tr :: 'lval)  $\neq \text{fa}$ 
    and  $u \neq t$ 

  shows  $\models \text{TERM } \{P\ 0\} \text{ while-cond } b C \{R\}$ 
proof (rule total-hyper-triple-altI)
  fix  $S :: ('lvar, 'lval, 'pvar, 'pval) \text{ state set}$ 
  assume asm0:  $P\ 0\ S$ 
  have  $\bigwedge n. P\ n\ (\text{iterate-sem } n\ (\text{Assume } b;;\ C)\ S)$ 
  proof (rule indexed-invariant-then-power)
  fix  $n$ 
  have  $\models \{Q\ n\} C \{P\ (Suc\ n)\}$ 
  proof (rule hyper-hoare-tripleI)
  fix  $S$  assume asm1:  $Q\ n\ S$ 
  let  $?S = \text{assign-exp-to-lvar-set } e\ t\ S$ 
  have  $conj\ (Q\ n)\ (e\text{-recorded-in-}t\ e\ t)\ ?S$ 
  by (metis asm1 assms(6) conj-def e-recorded-in-t-if-assigned not-fv-hyper-assign-exp)
  then have  $conj\ (P\ (Suc\ n))\ (e\text{-smaller-than-t-weaker } e\ t\ u\ lt)\ (\text{sem } C\ ?S)$ 
  using assms(2) hyper-hoare-tripleE total-hyper-triple-def by blast
  then show  $P\ (Suc\ n)\ (\text{sem } C\ S)$ 
  using assign-exp-to-lvar-set-same-mod-updates[of t] assms(5) conj-def not-fv-hyperE[of
   $t\ P\ (Suc\ n)] \text{ same-mod-updates-sym[of } \{t\}] \text{ sem-update-commute[of } \{t\}]$ 
  by metis
  qed
  then show  $\models \{P\ n\} \text{ Assume } b;;\ C \{P\ (Suc\ n)\}$ 
  using assms(1) seq-rule total-hyper-triple-def by blast
qed (simp add: asm0)
  then have natural-partition  $P\ (\bigcup n. \text{iterate-sem } n\ (\text{Assume } b;;\ C)\ S)$ 
  by (simp add: natural-partitionI)

```

```

then show  $R$  (sem (while-cond b C) S)
  using SUP-cong assms(3) hyper-hoare-triple-def[of natural-partition P Assume
(not b) R] sem-seq sem-while while-cond-def
  by metis

show terminates-in (while-cond b C) S
proof (rule terminates-in-while-loop)
  show wfP lt
    by (simp add: assms(4))
  fix  $\varphi$  n
  assume asm1:  $\varphi \in \text{iterate-sem } n$  (Assume b ;; C) S  $\wedge$  b (snd  $\varphi$ )

  let ?S = assign-exp-to-lvar-set e t (iterate-sem n (Assume b ;; C) S)
  let ? $\varphi$  = assign-exp-to-lvar e t  $\varphi$ 

  let ?SS = update-lvar-set u ( $\lambda\varphi'$ . if  $\varphi' = ?\varphi$  then tr else fa) ?S

  have SS-def: ?SS = { ((fst  $\varphi'$ )(u := if  $\varphi' = ?\varphi$  then tr else fa), snd  $\varphi'$ ) |  $\varphi' \in ?S$  }
    by (simp add: update-lvar-set-def)

  have same: same-mod-updates {u} ?S ?SS
    by (meson same-update-lvar-set)

  let ? $\varphi'$  = ((fst ? $\varphi$ )(u := tr), snd ? $\varphi$ )

  have conj (P n) (e-recorded-in-t e t) ?SS
    by (metis  $\langle \wedge n. P n$  (iterate-sem n (Assume b ;; C) S)  $\rangle$  assms(5) conj-def
e-recorded-in-t-if-assigned not-fv-hyper-assign-exp)
  moreover have e-recorded-in-t e t ?SS
  proof (rule e-recorded-in-tI)
    fix  $\varphi'$  assume  $\varphi' \in ?SS$ 
    then show fst  $\varphi'$  t = e (snd  $\varphi'$ ) unfolding SS-def
      using assms(10) e-recorded-in-t-def e-recorded-in-t-if-assigned by fastforce
    qed

  then have conj (P n) (e-recorded-in-t e t) ?SS
    using assms(7) calculation conj-def not-fv-hyperE[of u P n] same
    by metis
  then have conj (Q n) (e-recorded-in-t e t) (sem (Assume b) ?SS)
    using assms(1) assume-sem[of b] e-recorded-in-t-def[of e t] hyper-hoare-tripleE[of
P n Assume b Q n
  update-lvar-set u ( $\lambda\varphi'$ . if  $\varphi' = \text{assign-exp-to-lvar } e \ t \ \varphi$  then tr else fa)
(assign-exp-to-lvar-set e t (iterate-sem n (Assume b ;; C) S))]
    by (simp add: conj-def sem-assume-setify)
  then have conj (P (Suc n)) (e-smaller-than-t-weaker e t u lt) (sem C (sem
(Assume b) ?SS))

```

using *assms(2) hyper-hoare-tripleE total-hyper-triple-def* **by** *blast*
moreover have $?φ \in (sem\ (Assume\ b)\ ?S)$
using *asm1 same-outside-set-lvar-assign-exp [of φ e t]*
by *(auto simp add: assign-exp-to-lvar-set-def assume-sem)*
moreover have $?φ' \in (sem\ (Assume\ b)\ ?SS)$
unfolding *SS-def*
proof *(rule in-semI)*
show $((fst\ ?φ)(u := if\ ?φ = assign-exp-to-lvar\ e\ t\ φ\ then\ tr\ else\ fa),\ snd\ ?φ) \in \{((fst\ φ^{\wedge})(u := if\ φ' = assign-exp-to-lvar\ e\ t\ φ\ then\ tr\ else\ fa),\ snd\ φ^{\wedge}) \mid φ'.\ φ' \in assign-exp-to-lvar-set\ e\ t\ (iterate-sem\ n\ (Assume\ b\ ;;\ C)\ S)\}$
using *asm1 assign-exp-to-lvar-set-def* **by** *fastforce*
show $fst\ ((fst\ (assign-exp-to-lvar\ e\ t\ φ))(u := if\ assign-exp-to-lvar\ e\ t\ φ = assign-exp-to-lvar\ e\ t\ φ\ then\ tr\ else\ fa),\ snd\ (assign-exp-to-lvar\ e\ t\ φ)) = fst\ ((fst\ (assign-exp-to-lvar\ e\ t\ φ))(u := tr),\ snd\ (assign-exp-to-lvar\ e\ t\ φ))$
by *presburger*
show $\langle Assume\ b,\ snd\ ((fst\ (assign-exp-to-lvar\ e\ t\ φ))(u := if\ assign-exp-to-lvar\ e\ t\ φ = assign-exp-to-lvar\ e\ t\ φ\ then\ tr\ else\ fa),\ snd\ (assign-exp-to-lvar\ e\ t\ φ)) \rangle \rightarrow snd\ ((fst\ (assign-exp-to-lvar\ e\ t\ φ))(u := tr),\ snd\ (assign-exp-to-lvar\ e\ t\ φ))$
by *(metis SemAssume asm1 same-outside-set-lvar-assign-exp snd-conv)*
qed
then obtain $σ'$ **where** $(fst\ ?φ',\ σ') \in sem\ C\ (sem\ (Assume\ b)\ ?SS) \wedge \langle C,\ snd\ ?φ' \rangle \rightarrow σ'$
using *total-hyper-tripleE assms(2)[of n]*
using $\langle Logic.conj\ (Q\ n)\ (e-recorded-in-t\ e\ t)\ (sem\ (Assume\ b)\ (update-lvar-set\ u\ (\lambda φ'.\ if\ φ' = assign-exp-to-lvar\ e\ t\ φ\ then\ tr\ else\ fa)\ (assign-exp-to-lvar-set\ e\ t\ (iterate-sem\ n\ (Assume\ b\ ;;\ C)\ S)))) \rangle$ **by** *blast*
moreover obtain $φ'$ **where** $φ' \in sem\ C\ (sem\ (Assume\ b)\ ?SS)\ fst\ (fst\ ?φ',\ σ')\ u = fst\ φ'\ u \wedge lt\ (e\ (snd\ (fst\ ?φ',\ σ')))\ (fst\ φ'\ t)$
using *calculation conj-def[of P (Suc n) e-smaller-than-t-weaker e t u lt]*
e-smaller-than-t-weaker-def[of e t u lt]
by *meson*
then have $fst\ φ'\ u = tr$
by *simp*
moreover have $fst\ ?φ\ t = fst\ φ'\ t \wedge single-sem\ C\ (snd\ φ)\ (snd\ φ')$
proof –
obtain $φ0$ **where** $φ0 \in sem\ (Assume\ b)\ ?SS\ fst\ φ0 = fst\ φ'\ single-sem\ C\ (snd\ φ0)\ (snd\ φ')$
by *(metis (no-types, lifting) φ' \in sem\ C\ (sem\ (Assume\ b)\ (update-lvar-set\ u\ (\lambda φ'.\ if\ φ' = assign-exp-to-lvar\ e\ t\ φ\ then\ tr\ else\ fa)\ (assign-exp-to-lvar-set\ e\ t\ (iterate-sem\ n\ (Assume\ b\ ;;\ C)\ S)))) \rangle fst-conv in-sem snd-conv)*
then have $φ0 \in ?SS$
using *assume-sem* **by** *auto*
then obtain $φ0'$ **where**
 $φ0' \in (assign-exp-to-lvar-set\ e\ t\ (iterate-sem\ n\ (Assume\ b\ ;;\ C)\ S))$
 $φ0 = ((fst\ φ0')(u := if\ φ0' = assign-exp-to-lvar\ e\ t\ φ\ then\ tr\ else\ fa),\ snd\ φ0')$
using *SS-def* **by** *auto*

```

then have  $\varphi 0 = ((fst \ ?\varphi)(u := tr), snd \ ?\varphi)$ 
by (metis (full-types) ‹fst  $\varphi 0 = fst \ \varphi'$ › assms(9) calculation(5) fst-conv
fun-upd-same)
then show ?thesis
by (metis ‹⟨C, snd  $\varphi 0$ ⟩  $\rightarrow$  snd  $\varphi'$ › ‹fst  $\varphi 0 = fst \ \varphi'$ › assms(10) fst-conv
fun-upd-other same-outside-set-lvar-assign-exp snd-conv)
qed
ultimately show  $\exists \sigma'. ((C, snd \ \varphi) \rightarrow \sigma') \wedge (\neg b \ \sigma' \vee lt \ (e \ \sigma') \ (e \ (snd \ \varphi)))$ 
by (metis (no-types, lifting) ‹fst (fst ((fst (assign-exp-to-lvar e t  $\varphi$ ))(u
:= tr), snd (assign-exp-to-lvar e t  $\varphi$ ),  $\sigma'$ ) u = fst  $\varphi'$  u  $\wedge$  lt (e (snd (fst ((fst
(assign-exp-to-lvar e t  $\varphi$ ))(u := tr), snd (assign-exp-to-lvar e t  $\varphi$ ),  $\sigma'$ ))) (fst  $\varphi'$ 
t)› assign-exp-to-lvar-def fst-conv fun-upd-same snd-conv)
qed
qed

```

end

8 Examples

In this file, we prove the correctness of the two compositionality proofs presented in Appendix D.2.

```

theory ExamplesCompositionality
imports Logic Compositionality
begin

```

definition *low* **where**

$$low \ l \ S \longleftrightarrow (\forall \varphi 1 \ \varphi 2. \varphi 1 \in S \wedge \varphi 2 \in S \longrightarrow snd \ \varphi 1 \ l = snd \ \varphi 2 \ l)$$

8.1 Examples using the core rules.

definition *GNI* **where**

$$GNI \ l \ h \ S \longleftrightarrow (\forall \varphi 1 \ \varphi 2. \varphi 1 \in S \wedge \varphi 2 \in S \\ \longrightarrow (\exists \varphi \in S. snd \ \varphi \ h = snd \ \varphi 1 \ h \wedge snd \ \varphi \ l = snd \ \varphi 2 \ l))$$

lemma *GNI-I*:

```

assumes  $\bigwedge \varphi 1 \ \varphi 2. \varphi 1 \in S \wedge \varphi 2 \in S$ 
 $\implies (\exists \varphi \in S. snd \ \varphi \ h = snd \ \varphi 1 \ h \wedge snd \ \varphi \ l = snd \ \varphi 2 \ l)$ 
shows GNI l h S
by (simp add: GNI-def assms)

```

8.2 Examples using the compositionality rules

definition *has-minimum* $:: 'c \Rightarrow ('d \Rightarrow 'd \Rightarrow bool) \Rightarrow ('a, 'b, 'c, 'd) \text{ chyperassertion}$
where

$$has-minimum \ x \ leq \ S \longleftrightarrow (\exists \omega \in S. \forall \omega' \in S. leq \ (snd \ \omega \ x) \ (snd \ \omega' \ x))$$

lemma *has-minimumI*:

assumes $\omega \in S$
and $\bigwedge \omega'. \omega' \in S \implies \text{leq } (\text{snd } \omega \ x) \ (\text{snd } \omega' \ x)$
shows *has-minimum* $x \ \text{leq } S$
by (*metis* *assms(1)* *assms(2)* *has-minimum-def*)

definition *is-monotonic* **where**

is-monotonic $i \ x \ \text{one} \ \text{two} \ \text{leq } S \longleftrightarrow (\forall \omega \in S. \forall \omega' \in S. \text{fst } \omega \ i = \text{one} \wedge \text{fst } \omega' \ i = \text{two} \longrightarrow \text{leq } (\text{snd } \omega \ x) \ (\text{snd } \omega' \ x))$

lemma *is-monotonicI*:

assumes $\bigwedge \omega \ \omega'. \omega \in S \implies \omega' \in S \implies \text{fst } \omega \ i = \text{one} \wedge \text{fst } \omega' \ i = \text{two} \implies \text{leq } (\text{snd } \omega \ x) \ (\text{snd } \omega' \ x)$
shows *is-monotonic* $i \ x \ \text{one} \ \text{two} \ \text{leq } S$
by (*simp* *add: assms is-monotonic-def*)

lemma *update-logical-equal-outside*:

equal-outside-set $\{i\} \ (\text{snd } \omega) \ (\text{snd } (\text{update-logical } \omega \ i \ v))$
by (*simp* *add: equal-outside-set-def update-logical-def*)

lemma *update-logical-read*:

fst $(\text{update-logical } \omega \ i \ v) \ i = v$
by (*simp* *add: update-logical-def*)

lemma *snd-update-logical-same*:

snd $(\text{update-logical } \omega \ i \ v) = \text{snd } \omega$
by (*simp* *add: update-logical-def*)

Figure 12

proposition *composing-monotonicity-and-minimum*:

fixes $P :: ((('a \Rightarrow 'b) \times ('c \Rightarrow 'd)) \text{ set} \Rightarrow \text{bool})$
fixes $i :: 'a$
fixes $x :: 'c$
fixes $y :: 'c$
fixes $\text{leq} :: 'd \Rightarrow 'd \Rightarrow \text{bool}$
fixes $\text{one} :: 'b$
fixes $\text{two} :: 'b$

assumes $\models \{ P \} \ C1 \ \{ \text{has-minimum } x \ \text{leq} \}$
and $\models \{ \text{is-monotonic } i \ x \ \text{one} \ \text{two} \ \text{leq} \} \ C2 \ \{ \text{is-monotonic } i \ y \ \text{one} \ \text{two} \ \text{leq} \}$
and $\models \{ (\text{is-singleton} :: ((('a \Rightarrow 'b) \times ('c \Rightarrow 'd)) \text{ set} \Rightarrow \text{bool})) \} \ C2 \ \{ \text{is-singleton} \}$
and $\text{one} \neq \text{two}$

and $\bigwedge x. \text{leq } x \ x$ — reflexivity

```

shows  $\models \{ P \} C1 ;; C2 \{ \text{has-minimum } y \text{ leq} \}$ 
using assms(1)
proof (rule seq-rule)

let  $?P1 = \text{is-singleton} \circ (\text{Set.filter } (\lambda\omega. \text{fst } \omega \text{ } i = \text{one}))$ 
let  $?P2 = \text{is-monotonic } i \text{ } x \text{ } \text{one } \text{two } \text{leq}$ 
let  $?P3 = \lambda S. \forall \omega \in S. \text{fst } \omega \text{ } i = \text{one} \vee \text{fst } \omega \text{ } i = \text{two}$ 

let  $?P = \text{conj } ?P1 (\text{conj } ?P2 ?P3)$ 

let  $?Q1 = \text{is-singleton} \circ (\text{Set.filter } (\lambda\omega. \text{fst } \omega \text{ } i = \text{one}))$ 
let  $?Q2 = \text{is-monotonic } i \text{ } y \text{ } \text{one } \text{two } \text{leq}$ 
let  $?Q3 = \lambda S. \forall \omega \in S. \text{fst } \omega \text{ } i = \text{one} \vee \text{fst } \omega \text{ } i = \text{two}$ 

let  $?Q = \text{conj } ?Q1 (\text{conj } ?Q2 ?Q3)$ 

show  $\models \{ (\text{has-minimum } x \text{ leq} :: (((\text{'a} \Rightarrow \text{'b}) \times (\text{'c} \Rightarrow \text{'d})) \text{ set} \Rightarrow \text{bool})) \} C2 \{ \text{has-minimum } y \text{ leq} \}$ 
proof (rule rule-LUpdate)

show entails-with-updates  $\{i\} (\text{has-minimum } x \text{ leq}) ?P$ 
proof (rule entails-with-updatesI)
  fix  $S :: ((\text{'a} \Rightarrow \text{'b}) \times (\text{'c} \Rightarrow \text{'d})) \text{ set}$ 
  assume asm0: has-minimum  $x \text{ leq } S$ 
  then obtain  $\omega$  where minimum:  $\omega \in S \wedge \omega'. \omega' \in S \implies \text{leq } (\text{snd } \omega \text{ } x) (\text{snd } \omega' \text{ } x)$ 
    by (metis has-minimum-def)
    let  $?\omega = \text{update-logical } \omega \text{ } i \text{ } \text{one}$ 
    let  $?S = \{ \text{update-logical } \omega' \text{ } i \text{ } \text{two} \mid \omega'. \omega' \in S \} \cup \{?\omega\}$ 
    have same-mod-updates  $\{i\} S ?S$ 
    proof (rule same-mod-updatesI)
      fix  $\omega'$  assume asm1:  $\omega' \in S$ 
      then have update-logical  $\omega' \text{ } i \text{ } \text{two} \in ?S$ 
      by blast
      moreover have snd  $\omega' = \text{snd } (\text{update-logical } \omega' \text{ } i \text{ } \text{two}) \wedge \text{equal-outside-set } \{i\} (\text{fst } \omega') (\text{fst } (\text{update-logical } \omega' \text{ } i \text{ } \text{two}))$ 
      by (metis equal-outside-setI fst-eqD fun-upd-other singletonI snd-update-logical-same update-logical-def)
      ultimately show  $\exists \omega'' \in ?S. \text{snd } \omega' = \text{snd } \omega'' \wedge \text{equal-outside-set } \{i\} (\text{fst } \omega') (\text{fst } \omega'')$ 
      by blast
    next
    fix  $\omega'$ 
    assume asm1:  $\omega' \in \{ \text{update-logical } \omega' \text{ } i \text{ } \text{two} \mid \omega'. \omega' \in S \} \cup \{ \text{update-logical } \omega \text{ } i \text{ } \text{one} \}$ 
    show  $\exists \omega \in S. \text{snd } \omega = \text{snd } \omega' \wedge \text{equal-outside-set } \{i\} (\text{fst } \omega') (\text{fst } \omega)$ 
    proof (cases  $\omega' \in \{ \text{update-logical } \omega' \text{ } i \text{ } \text{two} \mid \omega'. \omega' \in S \}$ )
      case True

```

then obtain ω'' **where** $\omega' = \text{update-logical } \omega'' \text{ i two } \omega'' \in S$
by *blast*
then show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *equal-outside-set-def fst-conv fun-upd-other insertCI snd-update-logical-same update-logical-def*)
next
case *False*
then show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *UnE asm1 equal-outside-setI fst-eqD fun-upd-other minimum(1) singletonD singletonI snd-update-logical-same update-logical-def*)
qed
qed
moreover have *is-singleton* (*Set.filter* ($\lambda\omega. \text{fst } \omega \text{ i} = \text{one}$) *?S*)
proof –
have *Set.filter* ($\lambda\omega. \text{fst } \omega \text{ i} = \text{one}$) *?S* \subseteq $\{\omega\}$
proof
fix ω **assume** $\omega \in \text{Set.filter } (\lambda\omega. \text{fst } \omega \text{ i} = \text{one}) \text{ ?S}$
then have $\omega \in \text{?S} \wedge \text{fst } \omega \text{ i} = \text{one}$
by *simp*
then show $\omega \in \{\omega\}$
using *assms(4) update-logical-read* **by** *force*
qed
moreover have $\{\omega\} \subseteq \text{Set.filter } (\lambda\omega. \text{fst } \omega \text{ i} = \text{one}) \text{ ?S}$
by (*simp add: update-logical-read*)
ultimately show *?thesis*
by (*simp add: is-singleton-def order-antisym-conv*)
qed
moreover have *is-monotonic* *i x one two leq ?S*
proof (*rule is-monotonicI*)
fix $\omega' \omega''$ **assume** *asm1*: $\omega' \in \text{?S } \omega'' \in \text{?S } \text{fst } \omega' \text{ i} = \text{one} \wedge \text{fst } \omega'' \text{ i} = \text{two}$
then have $\omega' \in \{\text{update-logical } \omega' \text{ i one}\} \wedge \omega'' \in \{\text{update-logical } \omega'' \text{ i two}\}$
using *assms(4) update-logical-read* **by** *fastforce*
then show *leq* (*snd* $\omega' \text{ x}$) (*snd* $\omega'' \text{ x}$)
by (*metis* (*no-types*, *lifting*) *asm1(2) calculation(1) minimum(2) same-mod-updates-def singletonD snd-update-logical-same subset-mod-updatesE*)
qed
moreover have $\bigwedge \omega'. \omega' \in \text{?S} \implies \text{fst } \omega' \text{ i} = \text{one} \vee \text{fst } \omega' \text{ i} = \text{two}$
using *update-logical-read* **by** *fastforce*
ultimately have *same-mod-updates* $\{i\} S \text{ ?S} \wedge \text{?P } \text{?S}$
using *comp-eq-dest-lhs[of is-singleton Set.filter (\lambda\omega. fst \omega i = one)] conj-def[of ?P1 conj ?P2 ?P3] conj-def[of ?P2 ?P3]*
by *simp*
then show $\exists S'. \text{same-mod-updates } \{i\} S S' \wedge \text{?P } S'$
by *blast*
qed
show *invariant-on-updates* $\{i\}$ (*has-minimum* *y leq* :: $((('a \implies 'b) \times ('c \implies 'd)) \text{ set} \implies \text{bool})$)

```

proof (rule invariant-on-updatesI)
  fix S :: (('a ⇒ 'b) × ('c ⇒ 'd)) set
  fix S'
  assume asm0: same-mod-updates {i} S S' has-minimum y leq S
  then show has-minimum y leq S'
  using has-minimum-def[of y leq S'] has-minimum-def[of y leq S] same-mod-updates-def[of
{i} S S'] subset-mod-updatesE[of {i}]
    by metis
qed

show ⊨ { ?P } C2 { has-minimum y leq }
proof (rule consequence-rule)
  show entails ?Q (has-minimum y leq)
  proof (rule entailsI)
    fix S :: (('a ⇒ 'b) × ('c ⇒ 'd)) set
    assume ?Q S
    then obtain asm0: is-singleton (Set.filter (λω. fst ω i = one) S) is-monotonic
i y one two leq S
      ∧ω'. ω' ∈ S ⇒ fst ω' i = one ∨ fst ω' i = two
      by (simp add: conj-def)
    then obtain ω where Set.filter (λω. fst ω i = one) S = {ω}
      by (metis is-singleton-def)
    then have ω ∈ S by auto
    then show has-minimum y leq S
    proof (rule has-minimumI)
      fix ω'
      assume asm1: ω' ∈ S
      show leq (snd ω y) (snd ω' y)
      proof (cases fst ω' i = one)
        case True
          then show ?thesis
            using ⟨Set.filter (λω. fst ω i = one) S = {ω}⟩ asm1 assms(5) by
fastforce
        case False
          then have fst ω' i = two
            using asm0(3) asm1 by blast
          then show ?thesis
            using ⟨Set.filter (λω. fst ω i = one) S = {ω}⟩ asm0(2) asm1
is-monotonic-def[of i y one two leq S]
            by auto
      qed
    qed
  qed
show ⊨ { ?P } C2 { ?Q }
proof (rule rule-and3)
  show ⊨ { is-singleton ∘ Set.filter (λω. fst ω i = one) } C2 { is-singleton ∘
Set.filter (λω. fst ω i = one) }
  proof (rule rule-apply)

```

```

show  $\models$  { (is-singleton :: (((a  $\Rightarrow$  b)  $\times$  (c  $\Rightarrow$  d)) set  $\Rightarrow$  bool)) } C2
{is-singleton}
using assms(3) by blast
show commute-with-sem (Set.filter ( $\lambda\omega. \text{fst } \omega \text{ } i = \text{one}$ ))
by (simp add: filter-prop-commute)
qed
show  $\models$  {is-monotonic i x one two leq} C2 {is-monotonic i y one two leq}
by (simp add: assms(2))

show  $\models$  {( $\lambda S. \forall \omega \in S. \text{fst } \omega \text{ } i = \text{one} \vee \text{fst } \omega \text{ } i = \text{two}$ )} C2 { $\lambda S. \forall \omega \in S. \text{fst } \omega \text{ } i = \text{one} \vee \text{fst } \omega \text{ } i = \text{two}$ }
using rule-lframe-single by fast
qed
qed (auto simp add: entails-refl)
qed
qed

```

In this definition, we use a logical variable for *h*, which records the initial value of the program variable *h*

definition *lGNI* :: '*pvar* \Rightarrow '*lvar* \Rightarrow (('lvar, 'lval, 'pvar, 'pval) *state*) *set* \Rightarrow *bool*
where
lGNI l h S \longleftrightarrow ($\forall \varphi 1 \in S. (\forall \varphi 2 \in S. \exists \varphi \in S. \text{fst } \varphi \text{ } h = \text{fst } \varphi 1 \text{ } h \wedge \text{snd } \varphi \text{ } l = \text{snd } \varphi 2 \text{ } l)$)

Figure 13

proposition *composing-GNI-with-SNI*:

fixes *h* :: '*lvar*
fixes *l* :: '*pvar*

```

assumes  $\models$  { (low l :: (('lvar, 'lval, 'pvar, 'pval) state) hyperassertion) } C2 {
low l }
and  $\models$  { (not-empty :: (('lvar, 'lval, 'pvar, 'pval) state) hyperassertion) } C2
{ not-empty }
and  $\models$  { (low l :: (('lvar, 'lval, 'pvar, 'pval) state) hyperassertion) } C1 {
lGNI l h }
shows  $\models$  { (low l :: (('lvar, 'lval, 'pvar, 'pval) state) hyperassertion) } C1; C2
{ lGNI l h }
using assms(3)
proof (rule seq-rule)
show  $\models$  { (lGNI l h :: (('lvar, 'lval, 'pvar, 'pval) state) hyperassertion) } C2
{ lGNI l h }
unfolding lGNI-def
proof (rule rule-linking)
fix  $\varphi 1 \varphi 1' ::$  ('lvar, 'lval, 'pvar, 'pval) state
assume asm0: fst  $\varphi 1 = \text{fst } \varphi 1' \wedge \models$  { (in-set  $\varphi 1$  :: ('lvar, 'lval, 'pvar, 'pval)
state hyperassertion) } C2 {in-set  $\varphi 1'$ }
show  $\models$  {( $\lambda S. \forall \varphi 2 \in S. \exists \varphi \in S. \text{fst } \varphi \text{ } h = \text{fst } \varphi 1 \text{ } h \wedge \text{snd } \varphi \text{ } l = \text{snd } \varphi 2 \text{ } l$ )} C2
{ $\lambda S. \forall \varphi 2 \in S. \exists \varphi \in S. \text{fst } \varphi \text{ } h = \text{fst } \varphi 1' \text{ } h \wedge \text{snd } \varphi \text{ } l = \text{snd } \varphi 2 \text{ } l$ }
proof (rule consequence-rule)

```

```

let ?ex = λS. ∃φ ∈ S. fst φ h = fst φ1 h
let ?P = general-union (conj (low l) ?ex)
show ⊨ { (?P :: (('lvar, 'lval, 'pvar, 'pval) state) hyperassertion) } C2 {?P}
proof (rule rule-BigUnion; rule rule-And)
  show ⊨ {(low l :: (('lvar, 'lval, 'pvar, 'pval) state) hyperassertion)} C2 {low
l}
  using assms(1) by blast
  show ⊨ { ?ex } C2 { ?ex }
  proof (rule consequence-rule)
    let ?b = λφ0. φ0 h = fst φ1 h
    show ⊨ {not-empty ∘ Set.filter (?b ∘ fst)} C2 { not-empty ∘ Set.filter (?b
∘ fst)}
    using assms(2) rule-LFilter by auto
    show entails (λS. ∃φ ∈ S. fst φ h = fst φ1 h) (not-empty ∘ Set.filter ((λφ0.
φ0 h = fst φ1 h) ∘ fst))
    using CollectI comp-apply empty-iff not-empty-def
      entailsI[of (λS. ∃φ ∈ S. fst φ h = fst φ1 h) (not-empty ∘ Set.filter
((λφ0. φ0 h = fst φ1 h) ∘ fst))]
    by fastforce
    show entails (not-empty ∘ Set.filter ((λφ0. φ0 h = fst φ1 h) ∘ fst)) (λS.
∃φ ∈ S. fst φ h = fst φ1 h)
    proof (rule entailsI)
      fix S assume (not-empty ∘ Set.filter ((λφ0. φ0 h = fst φ1 h) ∘ fst)) S
      then obtain φ where φ ∈ Set.filter ((λφ0. φ0 h = fst φ1 h) ∘ fst) S
      by (metis comp-apply equals0I not-empty-def)
      then show ∃φ ∈ S. fst φ h = fst φ1 h
      by auto
    qed
  qed
  show entails (λS. ∀φ2 ∈ S. ∃φ ∈ S. fst φ h = fst φ1 h ∧ snd φ l = snd φ2 l)
?P
proof (rule entailsI)
  fix S assume asm0: ∀φ2 ∈ S. ∃φ ∈ S. fst φ h = fst φ1 h ∧ snd φ l = snd
φ2 l
  thm general-unionI
  let ?F = { {φ, φ2} | φ φ2. φ ∈ S ∧ φ2 ∈ S ∧ snd φ l = snd φ2 l ∧ fst φ
h = fst φ1 h }
  show general-union (Logic.conj (low l) (λS. ∃φ ∈ S. fst φ h = fst φ1 h)) S
  proof (rule general-unionI)
    show S = ⋃ ?F
    proof
      show S ⊆ ⋃ ?F
      proof
        fix φ2 assume φ2 ∈ S
        then obtain φ where φ ∈ S fst φ h = fst φ1 h ∧ snd φ l = snd φ2 l
        using asm0 by blast
        then have {φ, φ2} ∈ ?F
        using ⟨φ2 ∈ S⟩ by blast
      qed
    qed
  qed

```

```

      then show  $\varphi 2 \in \bigcup ?F$  by blast
    qed
    show  $\bigcup ?F \subseteq S$  by blast
  qed
  fix  $S'$  assume asm1:  $S' \in \{\{\varphi, \varphi 2\} \mid \varphi \varphi 2. \varphi \in S \wedge \varphi 2 \in S \wedge \text{snd } \varphi l = \text{snd } \varphi 2 l \wedge \text{fst } \varphi h = \text{fst } \varphi 1 h\}$ 
  then obtain  $\varphi \varphi 2$  where  $S' = \{\varphi, \varphi 2\}$   $\varphi \in S \wedge \varphi 2 \in S \wedge \text{snd } \varphi l = \text{snd } \varphi 2 l \wedge \text{fst } \varphi h = \text{fst } \varphi 1 h$ 
  by blast
  then show Logic.conj (low l) ( $\lambda S. \exists \varphi \in S. \text{fst } \varphi h = \text{fst } \varphi 1 h$ )  $S'$ 
  using conj-def[of low l  $\lambda S. \exists \varphi \in S. \text{fst } \varphi h = \text{fst } \varphi 1 h$ ] insert-iff low-def singletonD
  by fastforce
  qed
  qed
  show entails  $?P$  ( $\lambda S. \forall \varphi 2 \in S. \exists \varphi \in S. \text{fst } \varphi h = \text{fst } \varphi 1' h \wedge \text{snd } \varphi l = \text{snd } \varphi 2 l$ )
  proof (rule entailsI)
    fix  $S$  assume general-union (Logic.conj (low l) ( $\lambda S. \exists \varphi \in S. \text{fst } \varphi h = \text{fst } \varphi 1 h$ ))  $S$ 
    then obtain  $F$  where  $S = \bigcup F \wedge S'. S' \in F \implies \text{low l } S' \wedge (\exists \varphi \in S'. \text{fst } \varphi h = \text{fst } \varphi 1 h)$ 
    using conj-def[of low l  $\lambda S. \exists \varphi \in S. \text{fst } \varphi h = \text{fst } \varphi 1 h$ ]
      general-unionE[of Logic.conj (low l) ( $\lambda S. \exists \varphi \in S. \text{fst } \varphi h = \text{fst } \varphi 1 h$ )  $S$ ]
    by (metis (mono-tags, lifting))
    then show  $\forall \varphi 2 \in S. \exists \varphi \in S. \text{fst } \varphi h = \text{fst } \varphi 1' h \wedge \text{snd } \varphi l = \text{snd } \varphi 2 l$ 
    by (metis (mono-tags, lifting) Union-iff asm0 low-def)
  qed
  qed
  qed
  qed

```

8.3 Other examples

lemma *program-1-sat-gni*:

assumes $y \neq l \wedge y \neq h \wedge l \neq h$

shows $\vdash \{ \text{low } l \} \text{Seq} (\text{Havoc } y) (\text{Assign } l (\lambda \sigma. (\sigma h :: \text{int}) + \sigma y)) \{ \text{GNI } l h \}$

proof (rule *RuleSeq*)

let $?R = \lambda S. \forall \varphi 1 \varphi 2. \varphi 1 \in S \wedge \varphi 2 \in S$

$\longrightarrow (\exists \varphi \in S. (\text{snd } \varphi h :: \text{int}) = \text{snd } \varphi 1 h \wedge \text{snd } \varphi h + \text{snd } \varphi y = \text{snd } \varphi 2 h + \text{snd } \varphi 2 y)$

show $\vdash \{ \text{low } l \} \text{Havoc } y \{ ?R \}$

proof (rule *RuleCons*)

show $\vdash \{ (\lambda S. ?R \{(l, \sigma(y := v)) \mid l \sigma v. (l, \sigma) \in S\}) \} \text{Havoc } y \{ ?R \}$

using *RuleHavoc*[of $?R$] by *blast*

show *entails* (*low l*) ($\lambda S. ?R \{(l, \sigma(y := v)) \mid l \sigma (v :: \text{int}). (l, \sigma) \in S\}$)

proof (rule *entailsI*)

fix S

```

show ?R  $\{(l, \sigma(y := v)) \mid l \sigma (v :: \text{int}). (l, \sigma) \in S\}$ 
proof (clarify)
  fix a b aa ba l la  $\sigma \sigma' v va$ 
  assume asm0:  $(l, \sigma) \in S (la, \sigma') \in S$ 
  let ?v =  $(\sigma'(y := va)) h + (\sigma'(y := va)) y + - \sigma h$ 
  let ? $\varphi$  =  $(l, \sigma(y := v))$ 
  have  $\text{snd } ?\varphi h = \text{snd } (l, \sigma(y := v)) h \wedge \text{snd } ?\varphi h + \text{snd } ?\varphi y = \text{snd } (la,$ 
 $\sigma'(y := va)) h + \text{snd } (la, \sigma'(y := va)) y$ 
  using assms by force
  then show  $\exists \varphi \in \{(l, \sigma(y := v)) \mid l \sigma v. (l, \sigma) \in S\}.$ 
 $\text{snd } \varphi h = \text{snd } (l, \sigma(y := v)) h \wedge \text{snd } \varphi h + \text{snd } \varphi y = \text{snd } (la, \sigma'(y :=$ 
 $va)) h + \text{snd } (la, \sigma'(y := va)) y$ 
  using asm0(1) by blast
  qed
qed
show entails ?R ?R
  by (meson entailsI)
qed
show  $\vdash \{?R\} (\text{Assign } l (\lambda \sigma. \sigma h + \sigma y)) \{GNI \ l \ h\}$ 
proof (rule RuleCons)
  show  $\vdash \{(\lambda S. GNI \ l \ h \ \{(la, \sigma(l := \sigma h + \sigma y)) \mid la \ \sigma. (la, \sigma) \in S\}) \text{Assign } l$ 
 $(\lambda \sigma. \sigma h + \sigma y) \{GNI \ l \ h\}$ 
  using RuleAssign[of GNI l h l  $\lambda \sigma. \sigma h + \sigma y$ ] by blast
  show entails (GNI l h) (GNI l h)
  by (simp add: entails-def)
  show entails ?R  $(\lambda S. GNI \ l \ h \ \{(la, \sigma(l := \sigma h + \sigma y)) \mid la \ \sigma. (la, \sigma) \in S\})$ 
proof (rule entailsI)
  fix S
  assume asm0:  $\forall \varphi1 \ \varphi2. \varphi1 \in S \wedge \varphi2 \in S \longrightarrow (\exists \varphi \in S. \text{snd } \varphi h = \text{snd } \varphi1$ 
 $h \wedge \text{snd } \varphi h + \text{snd } \varphi y = \text{snd } \varphi2 h + \text{snd } \varphi2 y)$ 
  show GNI l h  $\{(la, \sigma(l := \sigma h + \sigma y)) \mid la \ \sigma. (la, \sigma) \in S\}$ 
proof (rule GNI-I)
  fix  $\varphi1 \ \varphi2$ 
  assume asm1:  $\varphi1 \in \{(la, \sigma(l := \sigma h + \sigma y)) \mid la \ \sigma. (la, \sigma) \in S\} \wedge \varphi2 \in$ 
 $\{(la, \sigma(l := \sigma h + \sigma y)) \mid la \ \sigma. (la, \sigma) \in S\}$ 
  then obtain  $la \ \sigma \ la' \ \sigma'$  where  $(la, \sigma) \in S (la', \sigma') \in S \varphi1 = (la, \sigma(l :=$ 
 $\sigma h + \sigma y)) \varphi2 = (la', \sigma'(l := \sigma' h + \sigma' y))$ 
  by blast
  then obtain  $\varphi$  where  $\varphi \in S \text{snd } \varphi h = \sigma h \text{snd } \varphi h + \text{snd } \varphi y = \sigma' h +$ 
 $\sigma' y$ 
  using asm0 snd-conv by force
  let ? $\varphi$  =  $(\text{fst } \varphi, (\text{snd } \varphi)(l := \text{snd } \varphi h + \text{snd } \varphi y))$ 
  have  $\text{snd } ?\varphi h = \text{snd } \varphi1 h \wedge \text{snd } ?\varphi l = \text{snd } \varphi2 l$ 
  using  $\langle \varphi1 = (la, \sigma(l := \sigma h + \sigma y)) \rangle \langle \varphi2 = (la', \sigma'(l := \sigma' h + \sigma' y)) \rangle$ 
 $\langle \text{snd } \varphi h + \text{snd } \varphi y = \sigma' h + \sigma' y \rangle \langle \text{snd } \varphi h = \sigma h \rangle$  assms by force
  then show  $\exists \varphi \in \{(la, \sigma(l := \sigma h + \sigma y)) \mid la \ \sigma. (la, \sigma) \in S\}. \text{snd } \varphi h = \text{snd}$ 
 $\varphi1 h \wedge \text{snd } \varphi l = \text{snd } \varphi2 l$ 
  using  $\langle \varphi \in S \rangle$  mem-Collect-eq[of ? $\varphi$ ]
  by (metis (mono-tags, lifting) prod.collapse)

```

qed
 qed
 qed
 qed

lemma *program-2-violates-gni*:

assumes $y \neq l \wedge y \neq h \wedge l \neq h$

shows $\vdash \{ \text{conj } (\text{low } l) (\lambda S. \exists a \in S. \exists b \in S. (\text{snd } a \ h :: \text{nat}) \neq \text{snd } b \ h) \}$

$\text{Seq } (\text{Seq } (\text{Havoc } y) (\text{Assume } (\lambda \sigma. \sigma \ y \geq (0 :: \text{nat}) \wedge \sigma \ y \leq (100 :: \text{nat})))) (\text{Assign}$

$l (\lambda \sigma. \sigma \ h + \sigma \ y))$

$\{ \lambda(S :: (('lvar \Rightarrow 'lval) \times ('a \Rightarrow \text{nat})) \text{ set}). \neg \text{GNI } l \ h \ S \}$

proof (*rule RuleSeq*)

let $?R0 = \lambda(S :: (('lvar \Rightarrow 'lval) \times ('a \Rightarrow \text{nat})) \text{ set}).$

$(\exists a \in S. \exists b \in S. \text{snd } b \ h > \text{snd } a \ h \wedge \text{snd } a \ y \geq (0 :: \text{nat}) \wedge \text{snd } a \ y \leq 100$

$\wedge \text{snd } b \ y = 100)$

let $?R1 = \lambda(S :: (('lvar \Rightarrow 'lval) \times ('a \Rightarrow \text{nat})) \text{ set}).$

$(\exists a \in S. \exists b \in S. \text{snd } b \ h > \text{snd } a \ h \wedge \text{snd } b \ y = 100) \wedge (\forall c \in S. \text{snd } c \ y \leq$

$100)$

let $?R2 = \lambda(S :: (('lvar \Rightarrow 'lval) \times ('a \Rightarrow \text{nat})) \text{ set}).$

$\exists a \in S. \exists b \in S. \forall c \in S. \text{snd } c \ h = \text{snd } a \ h \longrightarrow \text{snd } c \ h + \text{snd } c \ y = \text{snd } b \ h$

$+ \text{snd } b \ y$

show $\vdash \{ \text{conj } (\text{low } l) (\lambda S. \exists a \in S. \exists b \in S. \text{snd } a \ h \neq \text{snd } b \ h) \} \text{Seq } (\text{Havoc } y)$

$(\text{Assume } (\lambda \sigma. 0 \leq \sigma \ y \wedge \sigma \ y \leq (100 :: \text{nat}))) \{ ?R1 \}$

proof (*rule RuleSeq*)

show $\vdash \{ \text{conj } (\text{low } l) (\lambda S. \exists a \in S. \exists b \in S. \text{snd } a \ h \neq \text{snd } b \ h) \} \text{Havoc } y \{ ?R0 \}$

proof (*rule RuleCons*)

show $\vdash \{ (\lambda S. ?R0 \{ (l, \sigma(y := v)) \mid l \ \sigma \ v. (l, \sigma) \in S \}) \} \text{Havoc } y \{ ?R0 \}$

using *RuleHavoc[of - y]* **by** *fast*

show *entails* $?R0 \ ?R0$

by (*simp add: entailsI*)

show *entails* $(\text{conj } (\text{low } l) (\lambda S. \exists a \in S. \exists b \in S. \text{snd } a \ h \neq \text{snd } b \ h)) (\lambda S. ?R0$

$\{ (l, \sigma(y := v)) \mid l \ \sigma \ v. (l, \sigma) \in S \})$

proof (*rule entailsI*)

fix $S :: (('lvar \Rightarrow 'lval) \times ('a \Rightarrow \text{nat})) \text{ set}$

assume $\text{conj } (\text{low } l) (\lambda S. \exists a \in S. \exists b \in S. \text{snd } a \ h \neq \text{snd } b \ h) \ S$

then have $\exists a \in S. \exists b \in S. \text{snd } a \ h \neq \text{snd } b \ h$ **using** *conj-def*[of *low l* $\lambda S. \exists a \in S. \exists b \in S. \text{snd } a \ h \neq \text{snd } b \ h$]

by *blast*

then obtain $a \ b$ **where** $a \in S \ b \in S \ \text{snd } b \ h > \text{snd } a \ h$

by (*meson linorder-neq-iff*)

let $?a = (\text{fst } a, (\text{snd } a)(y := 100))$

let $?b = (\text{fst } b, (\text{snd } b)(y := 100))$

have $?a \in \{ (l, \sigma(y := v)) \mid l \ \sigma \ v. (l, \sigma) \in S \} \wedge ?b \in \{ (l, \sigma(y := v)) \mid l \ \sigma \ v. (l, \sigma) \in S \}$

using $\langle a \in S \rangle \langle b \in S \rangle$ **by** *fastforce*

moreover have $\text{snd } ?b \ h > \text{snd } ?a \ h \wedge \text{snd } ?a \ y \geq (0 :: \text{nat}) \wedge \text{snd } ?a \ y$

```

≤ 100 ∧ snd ?b y = 100
  using ⟨snd a h < snd b h⟩ assms by force
  ultimately show ?R0 {(l, σ(y := v)) | l σ v. (l, σ) ∈ S} by blast
qed
qed
show ⊢ {?R0} Assume (λσ. 0 ≤ σ y ∧ σ y ≤ 100) {?R1}
proof (rule RuleCons)
  show ⊢ {(λS. ?R1 (Set.filter ((λσ. 0 ≤ σ y ∧ σ y ≤ 100) ∘ snd)
    S))} Assume (λσ. 0 ≤ σ y ∧ σ y ≤ 100) {?R1}
  using RuleAssume[of - λσ. 0 ≤ σ y ∧ σ y ≤ 100]
  by fast
  show entails ?R1 ?R1
  by (simp add: entailsI)
  show entails ?R0 (λS. ?R1 (Set.filter ((λσ. 0 ≤ σ y ∧ σ y ≤ 100) ∘ snd)
S))
  proof (rule entailsI)
    fix S :: ('lvar ⇒ 'lval) × ('a ⇒ nat)) set
    assume asm0: ?R0 S
    then obtain a b where a∈S b∈S snd a h < snd b h ∧ 0 ≤ snd a y ∧ snd
a y ≤ (100 :: nat) ∧ snd b y = 100
      by blast
    then have a ∈ Set.filter ((λσ. 0 ≤ σ y ∧ σ y ≤ 100) ∘ snd) S ∧ b ∈
Set.filter ((λσ. 0 ≤ σ y ∧ σ y ≤ 100) ∘ snd) S
      by (simp add: ⟨a ∈ S⟩ ⟨b ∈ S⟩)
    then show ?R1 (Set.filter ((λσ. 0 ≤ σ y ∧ σ y ≤ 100) ∘ snd) S)
      using ⟨snd a h < snd b h ∧ 0 ≤ snd a y ∧ snd a y ≤ 100 ∧ snd b y =
100⟩
      by (cases a; cases b) fastforce
    qed
  qed
qed
show ⊢ { ?R1 } Assign l (λσ. σ h + σ y) {λS. ¬ GNI l h S}
proof (rule RuleCons)
  show ⊢ {(λS. ¬ GNI l h S {(la, σ(l := σ h + σ y)) | la σ. (la, σ) ∈ S})} Assign
l (λσ. σ h + σ y) {λS. ¬ GNI l h S}
  using RuleAssign[of λS. ¬ GNI l h S l λσ. σ h + σ y]
  by blast
  show entails (λS. ¬ GNI l h S) (λS. ¬ GNI l h S)
  by (simp add: entails-def)
  show entails (λS. (∃ a∈S. ∃ b∈S. snd a h < snd b h ∧ snd b y = 100) ∧ (∀ c∈S.
snd c y ≤ 100))
    (λ(S :: ('lvar ⇒ 'lval) × ('a ⇒ nat)) set). ¬ GNI l h {(la, σ(l := σ h + σ
y)) | la σ. (la, σ) ∈ S})
  proof (rule entailsI)
    fix S :: ('lvar ⇒ 'lval) × ('a ⇒ nat)) set
    assume asm0: (∃ a∈S. ∃ b∈S. snd a h < snd b h ∧ snd b y = 100) ∧ (∀ c∈S.
snd c y ≤ 100)
    then obtain a b where asm1: a∈S b∈S snd a h < snd b h ∧ snd b y = 100
by blast

```

```

    let ?a = (fst a, (snd a)(l := snd a h + snd a y))
    let ?b = (fst b, (snd b)(l := snd b h + snd b y))
    have  $\bigwedge la \sigma. (la, \sigma) \in S \implies (\sigma(l := \sigma h + \sigma y)) h = \text{snd } ?a h \implies (\sigma(l := \sigma h + \sigma y)) l \neq \text{snd } ?b l$ 
      using asm0 asm1(3) assms by fastforce
    moreover have  $r: ?a \in \{(la, \sigma(l := \sigma h + \sigma y)) \mid la \sigma. (la, \sigma) \in S\} \wedge ?b \in \{(la, \sigma(l := \sigma h + \sigma y)) \mid la \sigma. (la, \sigma) \in S\}$ 
      using asm1(1) asm1(2) by fastforce
    show  $\neg \text{GNI } l h \{(la, \sigma(l := \sigma h + \sigma y)) \mid la \sigma. (la, \sigma) \in S\}$ 
      proof (rule ccontr)
        assume  $\neg \neg \text{GNI } l h \{(la, \sigma(l := \sigma h + \sigma y)) \mid la \sigma. (la, \sigma) \in S\}$ 
        then have  $\text{GNI } l h \{(la, \sigma(l := \sigma h + \sigma y)) \mid la \sigma. (la, \sigma) \in S\}$ 
          by blast
        then obtain  $\varphi$  where  $\varphi \in \{(la, \sigma(l := \sigma h + \sigma y)) \mid la \sigma. (la, \sigma) \in S\}$  snd
 $\varphi h = \text{snd } ?a h$  snd  $\varphi l = \text{snd } ?b l$ 
          using GNI-def[of l h \{(la, \sigma(l := \sigma h + \sigma y)) \mid la \sigma. (la, \sigma) \in S\}] r
          by meson
        then show False
          using calculation by auto
      qed
    qed
  qed
end

```

```

end
theory PaperResults
  imports Loops SyntacticAssertions Compositionality TotalLogic ExamplesCompositional
begin

```

9 Summary of the Results from the Paper

This file contains the formal results mentioned the paper. It is organized in the same order and with the same structure as the paper.

- You can use the panel "Sidekick" on the right to see and navigate the structure of the file, via sections and subsections.
- You can ctrl+click on terms to jump to their definition.
- After jumping to another location, you can come back to the previous location by clicking the green left arrow, on the right side of the menu above.

9.1 3: Hyper Hoare Logic

9.1.1 3.1: Language and Semantics

The programming language is defined in the file `Language.thy`:

- The type of program state (definition 1) is $(\text{'pvar}, \text{'pval}) \textit{pstate}$ (<– you can ctrl+click on the name *pstate* above to jump to its definition).
- Program commands (definition 1) are defined via the type $(\text{'var}, \text{'val}) \textit{stmt}$.
- The big-step semantics (figure 9) is defined as *single-sem*. We also use the notation $\langle C, \sigma \rangle \rightarrow \sigma'$.

9.1.2 3.2: Hyper-Triples, Formally

- Extended states (definition 2) are defined as $(\text{'lvar}, \text{'lval}, \text{'pvar}, \text{'pval}) \textit{state}$ (file `Language.thy`).
- Hyper-assertions (definition 3) are defined as $(\text{'lvar}, \text{'lval}, \text{'pvar}, \text{'pval}) \textit{state hyperassertion}$ (file `Logic.thy`).
- The extended semantics (definition 4) is defined as *sem* (file `Language.thy`).
- Lemma 1 is shown and proven below.
- Hyper-triples (definition 5) are defined as *hyper-hoare-triple* (file `Logic.thy`). We also use the notation $\models \{P\} C \{Q\}$.

lemma *lemma1*:

$\textit{sem} C (S1 \cup S2) = \textit{sem} C S1 \cup \textit{sem} C S2$

$S \subseteq S' \implies \textit{sem} C S \subseteq \textit{sem} C S'$

$\textit{sem} C (\bigcup x. f x) = (\bigcup x. \textit{sem} C (f x))$

$\textit{sem} \textit{Skip} S = S$

$\textit{sem} (C1 ;; C2) S = \textit{sem} C2 (\textit{sem} C1 S)$

$\textit{sem} (\textit{If} C1 C2) S = \textit{sem} C1 S \cup \textit{sem} C2 S$

$\textit{sem} (\textit{While} C) S = (\bigcup n. \textit{iterate-sem} n C S)$

using *sem-if sem-seq sem-union sem-skip sem-union-general sem-monotonic sem-while*
by *metis+*

9.1.3 3.3: Core Rules

The core rules (from figure 2) are defined in the file `Logic.thy` as *syntactic-HHT*. We also use the notation $\vdash \{P\} C \{Q\}$. Operators \otimes (definition 6) and \otimes (definition 7) are defined as *join* and *natural-partition*, respectively.

9.1.4 3.4: Soundness and Completeness

Theorem 1: Soundness

theorem *thm1-soundness*:
assumes $\vdash \{P\} C \{Q\}$
shows $\models \{P\} C \{Q\}$
using *assms soundness by auto*

Theorem 2: Completeness

theorem *thm2-completeness*:
assumes $\models \{P\} C \{Q\}$
shows $\vdash \{P\} C \{Q\}$
using *assms completeness by auto*

9.1.5 3.5: Expressivity of Hyper-Triples

Program hyperproperties (definition 8) are defined in the file ProgramHyperproperties as the type $(\text{'pvar}, \text{'pval}) \text{ program-hyperproperty}$, which is syntactic sugar for the type $((\text{'pvar}, \text{'pval}) \text{ pstate} \times (\text{'pvar}, \text{'pval}) \text{ pstate}) \text{ set} \Rightarrow \text{bool}$. As written in the paper (after the definition), this type is equivalent to the type $((\text{'pvar}, \text{'pval}) \text{ pstate} \times (\text{'pvar}, \text{'pval}) \text{ pstate}) \text{ set set}$. The satisfiability of program hyperproperties is defined via the function *hypersat*.

Theorem 3: Expressing hyperproperties as hyper-triples

theorem *thm3-expressing-hyperproperties-as-hyper-triples*:
fixes *to-lvar* $:: \text{'pvar} \Rightarrow \text{'lvar}$
fixes *to-lval* $:: \text{'pval} \Rightarrow \text{'lval}$
fixes *H* $:: (\text{'pvar}, \text{'pval}) \text{ program-hyperproperty}$
assumes *injective to-lvar* — The cardinality of *'lvar* is at least the cardinality of *'pvar*.
and *injective to-lval* — The cardinality of *'lval* is at least the cardinality of *'pval*.
shows $\exists P Q :: (\text{'lvar}, \text{'lval}, \text{'pvar}, \text{'pval}) \text{ state hyperassertion. } (\forall C. \text{hypersat } C \text{ } H \longleftrightarrow \models \{P\} C \{Q\})$
using *assms proving-hyperproperties*
by *blast*

Theorem 4: Expressing hyper-triples as hyperproperties

theorem *thm4-expressing-hyper-triples-as-hyperproperties*:
 $\models \{P\} C \{Q\} \longleftrightarrow \text{hypersat } C (\text{hyperprop-hht } P \ Q)$
by (*simp add: any-hht-hyperprop*)

Theorem 5: Disproving hyper-triples

theorem *thm5-disproving-triples*:
 $\neg \models \{P\} C \{Q\} \longleftrightarrow (\exists P'. \text{sat } P' \wedge \text{entails } P' \ P \wedge \models \{P\} C \{\lambda S. \neg Q \ S\})$
using *disproving-triple by auto*

9.2 4: Syntactic Rules

9.2.1 4.1: Syntactic Hyper-Assertions

Syntactic hyper-expressions and hyper-assertions (definition 9) are defined in the file `SyntacticAssertions.thy` as `'val SyntacticAssertions.exp` and `'val assertion` respectively, where `'val` is the type of both logical and program values. Note that we use de Bruijn indices (i.e, natural numbers) for states and variables bound by quantifiers.

9.2.2 4.2: Syntactic Rules for Deterministic and Non-Deterministic Assignments.

We prove semantic versions of the syntactic rules from subsection 4 (figure 3). We use *interp-assert* to convert a syntactic hyper-assertion into a semantic one, because our hyper-triples require semantic hyper-assertions. Similarly, we use *interp-pexp* to convert a syntactic program expression into a semantic one. *transform-assign* $x e P$ and *transform-havoc* $x P$ correspond to A^e_x and H_x from definition 10.

Rule AssignS from figure 3

proposition *AssignS*:

$\vdash \{ \textit{interp-assert} (\textit{transform-assign} x e P) \} \textit{Assign} x (\textit{interp-pexp} e) \{ \textit{interp-assert} P \}$
using *completeness rule-assign-syntactic* **by** *blast*

Rule HavocS from figure 3

proposition *HavocS*:

$\vdash \{ \textit{interp-assert} (\textit{transform-havoc} x P) \} \textit{Havoc} x \{ \textit{interp-assert} P \}$
using *completeness rule-havoc-syntactic* **by** *blast*

9.2.3 4.3: Syntactic Rules for Assume Statements

transform-assume corresponds to Π_b (definition 11).

Rule AssumeS from figure 3

proposition *AssumeS*:

$\vdash \{ \textit{interp-assert} (\textit{transform-assume} (\textit{pbexp-to-assertion} 0 b) P) \} \textit{Assume} (\textit{interp-pbexp} b) \{ \textit{interp-assert} P \}$
using *completeness rule-assume-syntactic* **by** *blast*

As before, we use *interp-pbexp* to convert the syntactic program Boolean expression b into a semantic one. Similarly, *pbexp-to-assertion* $0 b$ converts the syntactic program Boolean expression p into a syntactic hyper-assertion. The number 0 is a de Bruijn index, which corresponds to the closest quantified state. For example, the hyper-assertion $\forall \langle \varphi \rangle. \varphi(a)=\varphi(b) \wedge (\exists \langle \varphi' \rangle.$

$\varphi(x) \succeq \varphi'(y)$) would be written as $\forall. 0(a)=0(b) \wedge (\exists. 1(x) \succeq 0(y))$ with de Bruijn indices. Thus, one can think of *pbeap-to-assertion* $0\ b$ as $b(\varphi)$, where φ is simply the innermost quantified state.

9.3 5: Proof Principles for Loops

We show in the following our proof rules for loops, presented in figure 5.

Rule WhileDesugared from figure 5

theorem *while-desugared*:

assumes $\wedge n. \vdash \{I\ n\} \text{ Assume } b;; C \{I\ (Suc\ n)\}$
and $\vdash \{ \text{natural-partition } I \} \text{ Assume } (\text{lnot } b) \{ Q \}$
shows $\vdash \{I\ 0\} \text{ while-cond } b\ C \{ Q \}$
by (*metis completeness soundness assms(1) assms(2) seq-rule while-cond-def while-rule*)

This result uses the following constructs:

- *natural-partition* I corresponds to the \otimes operator from definition 7.
- *lnot* b negates b .
- *while-cond* $b\ C$ is defined as $\text{While } (\text{Assume } b ;; C) ;; \text{Assume } (\text{lnot } b)$.

Rule WhileSync from figure 5 (presented in subsection 5.1)

lemma *WhileSync*:

assumes *entails* $I\ (\text{low-exp } b)$
and $\vdash \{ \text{conj } I\ (\text{holds-forall } b) \} C \{ I \}$
shows $\vdash \{ \text{conj } I\ (\text{low-exp } b) \} \text{ while-cond } b\ C \{ \text{conj } (\text{disj } I\ \text{emp})\ (\text{holds-forall } (\text{lnot } b)) \}$
using *WhileSync-simpler entails-conj completeness soundness assms(1) assms(2) entails-refl*
consequence-rule[of conj I (holds-forall b) conj I (holds-forall b) I conj I (low-exp b)] **by** *blast*

This result uses the following constructs:

- *Logic.conj* $A\ B$ corresponds to the hyper-assertion $A \wedge B$.
- *holds-forall* b corresponds to $\text{box}(b)$.
- *low-exp* b corresponds to $\text{low}(b)$.
- *Logic.disj* $A\ B$ corresponds to the hyper-assertion $A \vee B$.
- *emp* checks whether the set of states is empty.

Rule IfSync from figure 5 (presented in subsection 5.1)

theorem *IfSync*:

assumes *entails* P (*low-exp* b)
and $\vdash \{ \text{conj } P \text{ (holds-forall } b) \} C1 \{ Q \}$
and $\vdash \{ \text{conj } P \text{ (holds-forall (lnot } b)) \} C2 \{ Q \}$
shows $\vdash \{ P \} \text{if-then-else } b C1 C2 \{ Q \}$
using *completeness soundness consequence-rule*[of - *conj* P (*low-exp* b) Q] *assms*(1)
entail-conj entails-refl assms if-synchronized **by** *metis*

This result uses the following construct:

- *if-then-else* $b C1 C2$ is syntactic sugar for *stmt.If* (*Assume* b ;; $C1$) (*Assume* (*lnot* b) ;; $C2$).

Rule *While*– $\forall * \exists *$ from figure 5 (presented in subsection 5.2)

theorem *while-forall-exists*:

assumes $\vdash \{ I \} \text{if-then } b C \{ I \}$
and $\vdash \{ I \} \text{Assume (lnot } b) \{ \text{interp-assert } Q \}$
and *no-forall-state-after-existential* Q
shows $\vdash \{ I \} \text{while-cond } b C \{ \text{interp-assert } Q \}$
using *consequence-rule*[of $I I \text{conj (interp-assert } Q) \text{(holds-forall (lnot } b)) \text{interp-assert } Q$]
using *completeness soundness while-forall-exists-simpler assms entail-conj-weaken entails-refl* **by** *metis*

This result uses the following constructs:

- *if-then* $b C$ is syntactic sugar for *stmt.If* (*Assume* b ;; C) (*Assume* (*lnot* b)).
- *no-forall-state-after-existential* Q holds iff there is no universal state quantifier $\forall \langle - \rangle$ after any \exists in Q .

Rule *While*– \exists from figure 5 (presented in subsection 5.3)

theorem *while-loop-exists*:

assumes $\bigwedge v. \vdash \{ (\lambda S. \exists \varphi \in S. e \text{ (snd } \varphi) = v \wedge b \text{ (snd } \varphi) \wedge P \varphi S) \} \text{if-then } b$
 $C \{ (\lambda S. \exists \varphi \in S. \text{lt } (e \text{ (snd } \varphi)) v \wedge P \varphi S) \}$
and $\bigwedge \varphi. \vdash \{ P \varphi \} \text{while-cond } b C \{ Q \varphi \}$
and *wfP* lt
shows $\vdash \{ (\lambda S. \exists \varphi \in S. P \varphi S) \} \text{while-cond } b C \{ (\lambda S. \exists \varphi \in S. Q \varphi S) \}$
using *completeness soundness exists-terminates-loop assms* **by** *blast*

wfP lt in this result ensures that the binary operator lt is well-founded. e is a function of a program state, which must decrease after each iteration.

9.4 Appendix A: Technical Definitions Omitted from the Paper

The big-step semantics (figure 9) is defined as *single-sem*. We also use the notation $\langle C, \sigma \rangle \rightarrow \sigma'$. The following definitions are formalized in the file `SyntacticAssertions.thy`:

- Evaluation of hyper-expressions (definition 12): *interp-exp*.
- Satisfiability of hyper-assertions (definition 12): *sat-assertion*.
- Syntactic transformation for deterministic assignments (definition 13): *transform-assign*.
- Syntactic transformation for non-deterministic assignments (definition 14): *transform-havoc*.
- Syntactic transformation for assume statements. (definition 15): *transform-assume*.

9.5 Appendix C: Expressing Judgments of Hoare Logics as Hyper-Triples

9.5.1 Appendix C.1: Overapproximate Hoare Logics

The following judgments are defined in the file `Expressivity.thy` as follows:

- Definition 16 (Hoare Logic): $HL\ P\ C\ Q$.
- Definition 17 (Cartesian Hoare Logic): $CHL\ P\ C\ Q$.

Proposition 1: HL triples express hyperproperties

proposition *prop-1-HL-expresses-hyperproperties*:

$\exists H. (\forall C. \text{hypersat}\ C\ H \longleftrightarrow HL\ P\ C\ Q)$

using *HL-expresses-hyperproperties* **by** *blast*

Proposition 2: Expressing HL in Hyper Hoare Logic

proposition *prop-2-expressing-HL-in-HHL*:

$HL\ P\ C\ Q \longleftrightarrow (\text{hyper-hoare-triple}\ (\text{over-approx}\ P)\ C\ (\text{over-approx}\ Q))$

using *encoding-HL* **by** *auto*

Proposition 3: CHL triples express hyperproperties

proposition *prop-3-CHL-is-hyperproperty*:

$\text{hypersat}\ C\ (\text{CHL-hyperprop}\ P\ Q) \longleftrightarrow CHL\ P\ C\ Q$

using *CHL-hyperproperty* **by** *fast*

Proposition 4: Expressing CHL in Hyper Hoare Logic

proposition *prop-4-encoding-CHL-in-HHL:*

assumes *not-free-var-of P x*
and *not-free-var-of Q x*
and *injective from-nat*
shows $CHL\ P\ C\ Q \longleftrightarrow \models \{encode-CHL\ from-nat\ x\ P\}\ C\ \{encode-CHL\ from-nat\ x\ Q\}$
using *encoding-CHL assms by fast*

The function *from-nat* gives us a way to encode numbers from 1 to k as logical values. Moreover, note that we represent k-tuples implicitly, as mappings of type $'a \Rightarrow 'b$: When the type $'a$ has k elements, a function of type $'a \Rightarrow 'b$ corresponds to a k-tuple of elements of type 'b. This representation is more convenient to work with, and more general, since it also captures infinite sequences.

9.5.2 Appendix C.2: Underapproximate Hoare Logics

The following judgments are defined in the file Expressivity.thy as follows:

- Definition 18 (Incorrectness Logic): $IL\ P\ C\ Q$.
- Definition 19 (k-Incorrectness Logic): $RIL\ P\ C\ Q$.
- Definition 20 (Forward Underapproximation): $FU\ P\ C\ Q$.
- Definition 21 (k-Forward Underapproximation): $RFU\ P\ C\ Q$.

RIL is the old name of k-IL, and RFU is the old name of k-FU.

Proposition 5: IL triples express hyperproperties

proposition *prop-5-IL-hyperproperties:*

$IL\ P\ C\ Q \longleftrightarrow IL-hyperprop\ P\ Q\ (set-of-traces\ C)$
using *IL-expresses-hyperproperties by fast*

Proposition 6: Expressing IL in Hyper Hoare Logic

proposition *prop-6-expressing-IL-in-HHL:*

$IL\ P\ C\ Q \longleftrightarrow (hyper-hoare-triple\ (under-approx\ P)\ C\ (under-approx\ Q))$
using *encoding-IL by fast*

Proposition 7: k-IL triples express hyperproperties

proposition *prop-7-kIL-hyperproperties:*

$hypersat\ C\ (RIL-hyperprop\ P\ Q) \longleftrightarrow RIL\ P\ C\ Q$
using *RIL-expresses-hyperproperties by fast*

Proposition 8: Expressing k-IL in Hyper Hoare Logic

proposition *prop-8-expressing-kIL-in-HHL*:

fixes $x :: 'lvar$
assumes $\bigwedge l l' \sigma. (\lambda i. (l i, \sigma i)) \in P \longleftrightarrow (\lambda i. (l' i, \sigma i)) \in P$
and *injective* (*indexify* $:: (('a \Rightarrow ('pvar \Rightarrow 'pval)) \Rightarrow 'lval)$)
and $c \neq x$
and *injective from-nat*
and *not-free-var-of* ($P :: ('a \Rightarrow ('lvar \Rightarrow 'lval)) \times ('pvar \Rightarrow 'pval)$) *set*) $x \wedge$
not-free-var-of $P c$
and *not-free-var-of* $Q x \wedge$ *not-free-var-of* $Q c$
shows $RIL P C Q \longleftrightarrow \models \{pre-insec\ from-nat\ x\ c\ P\} C \{post-insec\ from-nat\ x\ c\ Q\}$
using *assms(1) assms(2) assms(3) assms(4) assms(5) assms(6)* **by** (*rule encoding-RIL*)

proposition *FU-hyperproperties*:

hypersat C (*hyperprop-FU* $P Q$) $\longleftrightarrow FU P C Q$
by (*rule FU-expresses-hyperproperties*)

Proposition 9: Expressing FU in Hyper Hoare Logic

proposition *prop-9-expressing-FU-in-HHL*:

$FU P C Q \longleftrightarrow \models \{encode-FU P\} C \{encode-FU Q\}$
by (*rule encoding-FU*)

Proposition 10: k-FU triples express hyperproperties

proposition *prop-10-kFU-expresses-hyperproperties*:

hypersat C (*RFU-hyperprop* $P Q$) $\longleftrightarrow RFU P C Q$
by (*rule RFU-captures-hyperproperties*)

Proposition 11: Expressing k-FU in Hyper Hoare Logic

proposition *prop-11-encode-kFU-in-HHL*:

assumes *not-free-var-of* $P x$
and *not-free-var-of* $Q x$
and *injective from-nat*
shows $RFU P C Q \longleftrightarrow \models \{encode-RFU\ from-nat\ x\ P\} C \{encode-RFU\ from-nat\ x\ Q\}$
using *assms*
by (*rule encode-RFU*)

9.5.3 Appendix C.3: Beyond Over- and Underapproximation

The following judgment is defined in the file *Expressivity.thy* as follows:

- Definition 22 (k-Universal Existential): $RUE P C Q$. Note that RUE is the old name of k-UE.

Proposition 12: k-UE triples express hyperproperties

proposition *prop-12-kUE-expresses-hyperproperty*:

$RUE\ P\ C\ Q \longleftrightarrow \text{hypersat } C\ (\text{hyperprop-RUE } P\ Q)$
by (rule RUE-express-hyperproperties)

Proposition 13: Expressing k-UE in Hyper Hoare Logic

proposition *prop-13-expressing-kUE-in-HHL*:

assumes $\text{injective } fn \wedge \text{injective } fn1 \wedge \text{injective } fn2$
and $t \neq x$
and $\text{injective } (fn :: \text{nat} \Rightarrow 'a)$
and $\text{injective } fn1$
and $\text{injective } fn2$
and $\text{not-in-free-vars-double } \{x, t\} P$
and $\text{not-in-free-vars-double } \{x, t\} Q$
shows $RUE\ P\ C\ Q \longleftrightarrow \models \{ \text{encode-RUE-1 } fn\ fn1\ fn2\ x\ t\ P \}\ C\ \{ \text{encode-RUE-2 } fn\ fn1\ fn2\ x\ t\ Q \}$
using *assms* **by** (rule encoding-RUE)

Example 3

proposition *proving-refinement*:

fixes $P :: (('lvar \Rightarrow 'lval) \times ('pvar \Rightarrow 'pval))\ \text{set} \Rightarrow \text{bool}$
and $t :: 'pvar$
assumes ($\text{one} :: 'pval$) \neq two — We assume two distinct program values *one* and *two*, to represent 1 and 2.
and $P = (\lambda S. \text{card } S = 1)$
and $Q = (\lambda S. \forall \varphi \in S. \text{snd } \varphi\ t = \text{two} \longrightarrow (\text{fst } \varphi, (\text{snd } \varphi)(t := \text{one})) \in S)$
and $\text{not-free-var-stmt } t\ C1$
and $\text{not-free-var-stmt } t\ C2$
shows $\text{refinement } C2\ C1 \longleftrightarrow$
 $\models \{ P \}\ \text{If } (\text{Seq } (\text{Assign } t\ (\lambda-. \text{one}))\ C1)\ (\text{Seq } (\text{Assign } t\ (\lambda-. \text{two}))\ C2)\ \{ Q \}$
proof –
have $\text{refinement } C2\ C1 \longleftrightarrow \models \{ P \}\ \text{If } (\text{Seq } (\text{Assign } t\ (\lambda-. \text{two}))\ C2)\ (\text{Seq } (\text{Assign } t\ (\lambda-. \text{one}))\ C1)\ \{ Q \}$
using *encoding-refinement*[of *two one P Q t C2 C1*] *assms* **by** *blast*
then show *?thesis* **using** *rewrite-if-commute* **by** *blast*
qed

9.6 Appendix D: Compositionality

9.6.1 Appendix D.1: Compositionality Rules

In the following, we show the rules from figure 11, in the order in which they appear.

proposition *rule-Linking*:

assumes $\bigwedge \varphi1\ (\varphi2 :: ('a, 'b, 'c, 'd)\ \text{state}). \text{fst } \varphi1 = \text{fst } \varphi2 \wedge (\vdash \{ (\text{in-set } \varphi1 :: ('a, 'b, 'c, 'd)\ \text{state})\ \text{hyperassertion} \})\ C\ \{ \text{in-set } \varphi2 \}$
 $\implies (\vdash \{ (P\ \varphi1 :: ('a, 'b, 'c, 'd)\ \text{state})\ \text{hyperassertion} \})\ C\ \{ Q\ \varphi2 \}$
shows $\vdash \{ ((\lambda S. \forall \varphi1 \in S. P\ \varphi1\ S) :: ('a, 'b, 'c, 'd)\ \text{state})\ \text{hyperassertion} \}\ C\ \{ (\lambda S. \forall \varphi2 \in S. Q\ \varphi2\ S) \}$
using *assms* *soundness completeness rule-linking* **by** *blast*

proposition *rule-And*:

assumes $\vdash \{P\} C \{Q\}$
and $\vdash \{P'\} C \{Q'\}$
shows $\vdash \{conj P P'\} C \{conj Q Q'\}$
using *assms soundness completeness rule-And* **by** *metis*

proposition *rule-Or*:

assumes $\vdash \{P\} C \{Q\}$
and $\vdash \{P'\} C \{Q'\}$
shows $\vdash \{disj P P'\} C \{disj Q Q'\}$
using *assms soundness completeness rule-Or* **by** *metis*

proposition *rule-FrameSafe*:

assumes *wr* $C \cap fv F = \{\}$
and *wf-assertion* F
and *no-exists-state* F
shows $\vdash \{interp-assert F\} C \{interp-assert F\}$
using *safe-frame-rule-syntactic assms completeness* **by** *metis*

proposition *rule-Forall*:

assumes $\bigwedge x. \vdash \{P x\} C \{Q x\}$
shows $\vdash \{forall P\} C \{forall Q\}$
using *assms soundness completeness rule-Forall* **by** *metis*

proposition *rule-IndexedUnion*:

assumes $\bigwedge x. \vdash \{P x\} C \{Q x\}$
shows $\vdash \{general-join P\} C \{general-join Q\}$
using *assms soundness completeness rule-IndexedUnion* **by** *metis*

proposition *rule-Union*:

assumes $\vdash \{P\} C \{Q\}$
and $\vdash \{P'\} C \{Q'\}$
shows $\vdash \{join P P'\} C \{join Q Q'\}$
using *assms soundness completeness rule-Union* **by** *metis*

proposition *rule-BigUnion*:

fixes $P :: ((('a \Rightarrow 'b) \times ('c \Rightarrow 'd)) \text{ set} \Rightarrow \text{bool})$
assumes $\vdash \{P\} C \{Q\}$
shows $\vdash \{general-union P\} C \{general-union Q\}$
using *assms soundness completeness rule-BigUnion* **by** *blast*

proposition *rule-Specialize*:

assumes $\vdash \{interp-assert P\} C \{interp-assert Q\}$
and *indep-of-set* b
and *wf-assertion-aux* $0\ 1\ b$
and *wr* $C \cap fv b = \{\}$
shows $\vdash \{interp-assert (transform-assume b P)\} C \{interp-assert (transform-assume b Q)\}$

using *filter-rule-syntactic assms soundness completeness* **by** *blast*

In the following, *entails-with-updates vars P P'* and *invariant-on-updates vars Q* respectively correspond to the notions of entailments modulo logical variables and invariance with respect to logical updates, as described in definition 23.

proposition *rule-LUpdate*:

assumes $\vdash \{P'\} C \{Q\}$
and *entails-with-updates vars P P'*
and *invariant-on-updates vars Q*
shows $\vdash \{P\} C \{Q\}$
using *assms soundness completeness rule-LUpdate* **by** *blast*

proposition *rule-LUpdateSyntactic*:

assumes $\vdash \{(\lambda S. P S \wedge e\text{-recorded-in-}t\ e\ t\ S)\} C \{Q\}$
and *not-fv-hyper t P*
and *not-fv-hyper t Q*
shows $\vdash \{P\} C \{Q\}$
using *LUpdateS soundness completeness assms* **by** *fast*

proposition *rule-AtMost*:

assumes $\vdash \{P\} C \{Q\}$
shows $\vdash \{has\text{-superset } P\} C \{has\text{-superset } Q\}$
using *assms soundness completeness rule-AtMost* **by** *blast*

proposition *rule-AtLeast*:

assumes $\vdash \{P\} C \{Q\}$
shows $\vdash \{has\text{-subset } P\} C \{has\text{-subset } Q\}$
using *assms soundness completeness rule-AtLeast* **by** *blast*

proposition *rule-True*:

$\vdash \{P\} C \{\lambda-. True\}$
using *rule-True completeness* **by** *blast*

proposition *rule-False*:

$\vdash \{(\lambda-. False)\} C \{Q\}$
using *rule-False completeness* **by** *blast*

proposition *rule-Empty*:

$\vdash \{(\lambda S. S = \{\})\} C \{(\lambda S. S = \{\})\}$
using *completeness rule-Empty* **by** *blast*

9.6.2 Appendix D.2: Examples

Example shown in figure 12. To see the actual proof, ctrl+click on \models
 $\{?P\} ?C1.0 \{has\text{-minimum } ?x ?leq\}; \models \{is\text{-monotonic } ?i ?x ?one ?two ?leq\} ?C2.0 \{is\text{-monotonic } ?i ?y ?one ?two ?leq\}; \models \{is\text{-singleton}\} ?C2.0$

$\{is\text{-singleton}\}; ?one \neq ?two; \wedge x. ?leq\ x\ x \implies \models \{?P\} ?C1.0 \;; ?C2.0$
 $\{has\text{-minimum}\ ?y\ ?leq\}$.

proposition *fig-12-composing-monotonicity-and-minimum*:

fixes $P :: ((('a \Rightarrow 'b) \times ('c \Rightarrow 'd))\ set \Rightarrow bool)$
fixes $i :: 'a$
fixes $x\ y :: 'c$
fixes $leq :: 'd \Rightarrow 'd \Rightarrow bool$
fixes $one\ two :: 'b$
assumes $\vdash \{ P \} C1 \{ has\text{-minimum}\ x\ leq \}$
and $\vdash \{ is\text{-monotonic}\ i\ x\ one\ two\ leq \} C2 \{ is\text{-monotonic}\ i\ y\ one\ two\ leq \}$
and $\vdash \{ (is\text{-singleton} :: ((('a \Rightarrow 'b) \times ('c \Rightarrow 'd))\ set \Rightarrow bool)) \} C2 \{ is\text{-singleton} \}$
and $one \neq two$ — We use distinct logical values *one* and *two* to represent 1 and 2.
and $\wedge x. leq\ x\ x$ — We assume that *leq* is a partial order, and thus that it satisfies reflexivity.
shows $\vdash \{ P \} C1 \;; C2 \{ has\text{-minimum}\ y\ leq \}$
using *assms soundness completeness composing-monotonicity-and-minimum by metis*

Example shown in figure 13. To see the actual proof, ctrl+click on $\models \{low\ ?l\} ?C2.0 \{low\ ?l\}; \models \{not\text{-empty}\} ?C2.0 \{not\text{-empty}\}; \models \{low\ ?l\} ?C1.0 \{lGNI\ ?l\ ?h\} \implies \models \{low\ ?l\} ?C1.0 \;; ?C2.0 \{lGNI\ ?l\ ?h\}$.

proposition *fig-13-composing-GNI-with-SNI*:

fixes $h :: 'lvar$
fixes $l :: 'pvar$
assumes $\vdash \{ (low\ l :: (('lvar, 'lval, 'pvar, 'pval)\ state)\ hyperassertion) \} C2 \{ low\ l \}$
and $\vdash \{ (not\text{-empty} :: (('lvar, 'lval, 'pvar, 'pval)\ state)\ hyperassertion) \} C2 \{ not\text{-empty} \}$
and $\vdash \{ (low\ l :: (('lvar, 'lval, 'pvar, 'pval)\ state)\ hyperassertion) \} C1 \{ lGNI\ l\ h \}$
shows $\vdash \{ (low\ l :: (('lvar, 'lval, 'pvar, 'pval)\ state)\ hyperassertion) \} C1 \;; C2 \{ lGNI\ l\ h \}$
using *assms soundness completeness composing-GNI-with-SNI by metis*

9.7 Appendix E: Termination-Based Reasoning

Terminating hyper-triples (definition 24) are defined as *total-hyper-triple*, and usually written $\models_{TERM} \{P\} C \{Q\}$.

theorem *rule-Frame*:

assumes $\models_{TERM} \{P\} C \{Q\}$
and $wr\ C \cap fv\ F = \{ \}$
and *wf-assertion* F
shows $\models_{TERM} \{conj\ P\ (interp\text{-assert}\ F)\} C \{conj\ Q\ (interp\text{-assert}\ F)\}$
by (*simp add: assms(1) assms(2) assms(3) frame-rule-syntactic*)

theorem *rule-WhileSyncTerm*:

assumes $\models_{TERM} \{conj\ I\ (\lambda S. \forall \varphi \in S. b\ (snd\ \varphi) \wedge fst\ \varphi\ t = e\ (snd\ \varphi))\}\ C$
 $\{conj\ (conj\ I\ (low\text{-}exp\ b))\ (e\text{-smaller}\text{-}than\text{-}t\ e\ t\ lt)\}$
and $wfP\ lt$
and $not\text{-}fv\text{-}hyper\ t\ I$
shows $\models_{TERM} \{conj\ I\ (low\text{-}exp\ b)\}\ while\text{-}cond\ b\ C\ \{conj\ I\ (holds\text{-}forall\ (lnot\ b))\}$
by $(meson\ WhileSyncTot\ assms(1)\ assms(2)\ assms(3))$

9.8 Appendix H: Synchronous Reasoning over Different Branches

Proposition 14: Synchronous if rule

proposition *prop-14-synchronized-if-rule:*

assumes $\models \{P\}\ C1\ \{P1\}$
and $\models \{P\}\ C2\ \{P2\}$
and $\models \{combine\ from\text{-}nat\ x\ P1\ P2\}\ C\ \{combine\ from\text{-}nat\ x\ R1\ R2\}$
and $\models \{R1\}\ C1'\ \{Q1\}$
and $\models \{R2\}\ C2'\ \{Q2\}$
and $not\text{-}free\text{-}var\text{-}hyper\ x\ P1$
and $not\text{-}free\text{-}var\text{-}hyper\ x\ P2$
and $from\text{-}nat\ 1 \neq from\text{-}nat\ 2$ — We can represent 1 and 2 as distinct logical values.
and $not\text{-}free\text{-}var\text{-}hyper\ x\ R1$
and $not\text{-}free\text{-}var\text{-}hyper\ x\ R2$
shows $\models \{P\}\ If\ (Seq\ C1\ (Seq\ C\ C1'))\ (Seq\ C2\ (Seq\ C\ C2'))\ \{join\ Q1\ Q2\}$
using *if-sync-rule assms by meson*

end

References

- [1] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *21st IEEE Computer Security Foundations Symposium*, pages 51–65, 2008. doi:10.1109/CSF.2008.7.
- [2] Thibault Dardinier and Peter Müller. Hyper Hoare Logic: (dis-)proving program hyperproperties (extended version). *arXiv preprint arXiv:2301.10037*, 2023. doi:10.48550/arXiv.2301.10037.
- [3] Thibault Dardinier and Peter Müller. Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties. *Proc. ACM Program. Lang.*, 8(PLDI), jun 2024. doi:10.1145/3656437.
- [4] Edsko de Vries and Vasileios Koutavas. Reverse Hoare Logic. In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *Software Engineering and Formal Methods*, pages 155–171, 2011.

- [5] Robert Dickerson, Qianchuan Ye, Michael K. Zhang, and Benjamin Delaware. RHLE: Modular deductive verification of relational $\forall\exists$ properties. In *Programming Languages and Systems: 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings*, page 6787, 2022. doi:[10.1007/978-3-031-21037-2_4](https://doi.org/10.1007/978-3-031-21037-2_4).
- [6] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium in Applied Mathematics*, pages 19–32, 1967.
- [7] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576580, oct 1969. doi:[10.1145/363235.363259](https://doi.org/10.1145/363235.363259).
- [8] Toby Murray. An under-approximate relational logic: Heralding logics of insecurity, incorrect implementation and more. *arXiv preprint arXiv:2003.04791*, 2020. URL: <https://arxiv.org/abs/2003.04791>, doi: [10.48550/ARXIV.2003.04791](https://doi.org/10.48550/ARXIV.2003.04791).
- [9] Peter W. O’Hearn. Incorrectness Logic. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi:[10.1145/3371078](https://doi.org/10.1145/3371078).
- [10] Marcelo Sousa and Isil Dillig. Cartesian Hoare Logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’16*, page 5769, New York, NY, USA, 2016. Association for Computing Machinery. doi:[10.1145/2908080.2908092](https://doi.org/10.1145/2908080.2908092).
- [11] Noam Zilberstein, Derek Dreyer, and Alexandra Silva. Outcome Logic: A unifying foundation for correctness and incorrectness reasoning. *Proc. ACM Program. Lang.*, 7(OOPSLA1), april 2023. doi: [10.1145/3586045](https://doi.org/10.1145/3586045).