

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

Thibault Dardinier
Department of Computer Science
ETH Zurich, Switzerland

May 26, 2024

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	7
2.1	Rules of the Logic	10
2.2	Soundness	11
2.3	Completeness	13
2.4	Disproving Hyper-Triples	15
2.5	Synchronized Rule for Branching	16
3	Expressivity of Hyper Hoare Logic	17
3.1	Program Hyperproperties	17
3.2	Hoare Logic (HL) [7]	20
3.3	Cartesian Hoare Logic (CHL) [10]	21
3.4	Incorrectness Logic [9] or Reverse Hoare Logic [4] (IL)	23
3.5	k-Incorrectness Logic [8] (k-IL)	24
3.6	Forward Underapproximation (FU)	27
3.7	k-Forward Underapproximate logic	29
3.8	k-Universal Existential (RUE) [5]	30
3.9	Program Refinement	33
4	Rules for Loops	34
5	Compositionality Rules	43
5.1	Linking rule	43
5.2	Frame rules	44
5.3	Logical Updates	44
5.4	Filters	47
5.5	Other Compositionality Rules	48
5.6	Synchronous Reasoning (Proposition 14, Appendix H).	51
6	Syntactic Assertions	54
6.1	Preliminaries: Types, expressions, 'a assertions	54
6.2	Assume rule	56
6.2.1	Program expressions (values)	56
6.2.2	Program expressions (booleans)	57
6.2.3	Syntactic rule for assume	58
6.3	Havoc rule	58
6.3.1	Shifting variables	58
6.3.2	Expressions (Boolean and values)	60
6.3.3	Assertions	61

6.3.4	Transformation for havoc	62
6.3.5	Syntactic rule for havoc	63
6.4	Assignment rule	64
6.4.1	Program expressions	64
6.4.2	Expressions (Boolean and values)	64
6.4.3	Assertions	65
6.4.4	Syntactic rule for assignments	65
6.5	Loop rules	65
6.6	Rewrite rules for 'a assertions	69
6.7	Free variables and safe frame rule	70
7	Terminating Hyper-Triples	73
7.1	Specialize rule	74
7.2	Total version of core rules	79
8	Examples	81
8.1	Examples using the core rules.	81
8.2	Examples using the compositionality rules	81
8.3	Other examples	83
9	Summary of the Results from the Paper	83
9.1	3: Hyper Hoare Logic	83
9.1.1	3.1: Language and Semantics	83
9.1.2	3.2: Hyper-Triples, Formally	84
9.1.3	3.3: Core Rules	84
9.1.4	3.4: Soundness and Completeness	85
9.1.5	3.5: Expressivity of Hyper-Triples	85
9.2	4: Syntactic Rules	86
9.2.1	4.1: Syntactic Hyper-Assertions	86
9.2.2	4.2: Syntactic Rules for Deterministic and Non-Deterministic Assignments.	86
9.2.3	4.3: Syntactic Rules for Assume Statements	86
9.3	5: Proof Principles for Loops	87
9.4	Appendix A: Technical Definitions Omitted from the Paper	88
9.5	Appendix C: Expressing Judgments of Hoare Logics as Hyper- Triples	89
9.5.1	Appendix C.1: Overapproximate Hoare Logics	89
9.5.2	Appendix C.2: Underapproximate Hoare Logics	90
9.5.3	Appendix C.3: Beyond Over- and Underapproximation	91
9.6	Appendix D: Compositionality	92
9.6.1	Appendix D.1: Compositionality Rules	92
9.6.2	Appendix D.2: Examples	94
9.7	Appendix E: Termination-Based Reasoning	95
9.8	Appendix H: Synchronous Reasoning over Different Branches	95

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:  $\langle$ Skip,  $\sigma$  $\rangle \rightarrow \sigma$ 
```

```
| SemAssign:  $\langle$ Assign var e,  $\sigma$  $\rangle \rightarrow \sigma(\text{var} := (e \ \sigma))$ 
```

```
| SemSeq:  $\llbracket \langle C1, \sigma \rangle \rightarrow \sigma1; \langle C2, \sigma1 \rangle \rightarrow \sigma2 \rrbracket \Longrightarrow \langle$ Seq C1 C2,  $\sigma \rangle \rightarrow \sigma2$ 
```

```
| SemIf1:  $\langle C1, \sigma \rangle \rightarrow \sigma1 \Longrightarrow \langle$ If C1 C2,  $\sigma \rangle \rightarrow \sigma1$ 
```

```
| SemIf2:  $\langle C2, \sigma \rangle \rightarrow \sigma2 \Longrightarrow \langle$ If C1 C2,  $\sigma \rangle \rightarrow \sigma2$ 
```

```
| SemHavoc:  $\langle$ Havoc var,  $\sigma \rangle \rightarrow \sigma(\text{var} := v)$ 
```

```
| SemAssume:  $b \ \sigma \Longrightarrow \langle$ Assume b,  $\sigma \rangle \rightarrow \sigma$ 
```

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

```
| SemWhileExit:  $\langle$ While C,  $\sigma \rangle \rightarrow \sigma$ 
```

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

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'*) | $\sigma' \sigma \ l. (l, \sigma) \in S \wedge \langle C, \sigma \rangle \rightarrow \sigma'$ }

lemma *in-sem*:

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

Lemma 1: Useful properties of the extended semantics

lemma *sem-seq*:

sem (Seq C1 C2) S = *sem C2 (sem C1 S)* (**is** *?A = ?B*)
<proof>

lemma *sem-skip*:

sem Skip S = *S*
<proof>

lemma *sem-union*:

sem C (S1 \cup S2) = *sem C S1 \cup sem C S2* (**is** *?A = ?B*)
<proof>

lemma *sem-union-general*:

sem C ($\bigcup x. f \ x$) = ($\bigcup x. \text{sem } C \ (f \ x)$) (**is** *?A = ?B*)
<proof>

lemma *sem-monotonic*:

assumes $S \subseteq S'$
shows *sem C S* \subseteq *sem C S'*
<proof>

lemma *subsetPairI*:

assumes $\bigwedge l \ \sigma. (l, \sigma) \in A \implies (l, \sigma) \in B$
shows $A \subseteq B$

$\langle \text{proof} \rangle$

lemma *sem-if*:

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

$\langle \text{proof} \rangle$

lemma *sem-assume*:

$\text{sem } (\text{Assume } b) \ S = \{ (l, \sigma) \mid l \ \sigma. (l, \sigma) \in S \wedge b \ \sigma \}$ (**is** $?A = ?B$)

$\langle \text{proof} \rangle$

lemma *while-then-reaches*:

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

shows $\text{single-sem } (\text{While } C) \ \sigma \ \sigma''$

$\langle \text{proof} \rangle$

lemma *in-closure-then-while*:

assumes $\text{single-sem } C' \ \sigma \ \sigma''$

shows $\bigwedge C. C' = \text{While } C \implies (\text{single-sem } C)^{**} \ \sigma \ \sigma''$

$\langle \text{proof} \rangle$

theorem *loop-equiv*:

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

$\langle \text{proof} \rangle$

fun *iterate-sem where*

$\text{iterate-sem } 0 \ S = S$

$\mid \text{iterate-sem } (\text{Suc } n) \ C \ S = \text{sem } C \ (\text{iterate-sem } n \ C \ S)$

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

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

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

$\langle \text{proof} \rangle$

lemma *reciprocal*:

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

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

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

$\langle \text{proof} \rangle$

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** $?A \longleftrightarrow ?B$)

$\langle \text{proof} \rangle$

lemma *sem-while*:

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

$\langle \text{proof} \rangle$

lemma *assume-sem*:

sem (*Assume* *b*) *S* = *Set.filter* (*b* \circ *snd*) *S* (**is** ?*A* = ?*B*)
<*proof*>

lemma *sem-split-general*:

sem *C* ($\bigcup x. F x$) = ($\bigcup x. \text{sem } C (F x)$) (**is** ?*A* = ?*B*)
<*proof*>

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* φ φ'
and *x* \notin *written-vars* *C*
shows $\varphi x = \varphi' x$
<*proof*>

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*
<*proof*>

lemma *entailsI*:

assumes $\bigwedge S. A S \implies B S$
shows *entails* $A B$
 \langle *proof* \rangle

lemma *entailsE*:
assumes *entails* $A B$
and $A x$
shows $B x$
 \langle *proof* \rangle

lemma *bientails-equal*:
assumes *entails* $A B$
and *entails* $B A$
shows $A = B$
 \langle *proof* \rangle

lemma *entails-trans*:
assumes *entails* $A B$
and *entails* $B C$
shows *entails* $A C$
 \langle *proof* \rangle

definition *setify-prop* **where**
setify-prop $b = \{ (l, \sigma) \mid l \sigma. b \sigma \}$

lemma *sem-assume-setify*:
sem (*Assume* b) $S = S \cap$ *setify-prop* b (**is** $?A = ?B$)
 \langle *proof* \rangle

definition *over-approx* $:: 'a \text{ set} \Rightarrow 'a \text{ hyperassertion}$ **where**
over-approx $P S \longleftrightarrow S \subseteq P$

definition *lower-closed* $:: 'a \text{ hyperassertion} \Rightarrow \text{bool}$ **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)
 \langle *proof* \rangle

definition *under-approx* $:: 'a \text{ set} \Rightarrow 'a \text{ hyperassertion}$ **where**
under-approx $P S \longleftrightarrow P \subseteq S$

definition *upper-closed* $:: 'a \text{ hyperassertion} \Rightarrow \text{bool}$ **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)
 \langle *proof* \rangle

definition *closed-by-union* :: 'a hyperassertion \Rightarrow 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 $\bigwedge a b. P a \Longrightarrow P b \Longrightarrow P (a \cup b)$
shows *closed-by-union* P
<proof>

lemma *closed-by-union-over*:
closed-by-union (*over-approx* P)
<proof>

lemma *closed-by-union-under*:
closed-by-union (*under-approx* P)
<proof>

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

lemma *entail-conj*:
assumes *entails* $A B$
shows *entails* A (*conj* $A B$)
<proof>

lemma *entail-conj-weaken*:
entails (*conj* $A B$) A
<proof>

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))
<proof>

lemma *over-union*:
entails (*disj* (*over-approx* P) (*over-approx* Q)) (*over-approx* ($P \cup Q$))
<proof>

lemma *under-union*:
entails (*under-approx* ($P \cup Q$)) (*disj* (*under-approx* P) (*under-approx* Q))
<proof>

lemma *under-inter*:

entails (*conj* (*under-approx* P) (*under-approx* Q)) (*under-approx* ($P \cap Q$))
 ⟨*proof*⟩

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* $f S$
 ⟨*proof*⟩

lemma *join-closed-by-union*:

assumes *closed-by-union* Q
shows *join* $Q Q = Q$
 ⟨*proof*⟩

lemma *entails-join-entails*:

assumes *entails* $A1 B1$
and *entails* $A2 B2$
shows *entails* (*join* $A1 A2$) (*join* $B1 B2$)
 ⟨*proof*⟩

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* $I S$
 ⟨*proof*⟩

lemma *natural-partitionE*:

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

2.1 Rules of the Logic

Core rules from figure 2

inductive *syntactic-HHT* ::

$((\text{'lvar}, \text{'lval}, \text{'pvar}, \text{'pval}) \text{ state hyperassertion}) \Rightarrow (\text{'pvar}, \text{'pval}) \text{ stmt} \Rightarrow ((\text{'lvar}, \text{'lval}, \text{'pvar}, \text{'pval}) \text{ state hyperassertion}) \Rightarrow \text{bool}$
 $(\vdash \{-\} - \{-\} [51,0,0] 81) \text{ where}$
 $\text{RuleSkip}: \vdash \{P\} \text{ Skip } \{P\}$
 $| \text{RuleCons}: \llbracket \text{entails } P P' ; \text{entails } Q' Q ; \vdash \{P'\} C \{Q'\} \rrbracket \Longrightarrow \vdash \{P\} C \{Q\}$
 $| \text{RuleSeq}: \llbracket \vdash \{P\} C1 \{R\} ; \vdash \{R\} C2 \{Q\} \rrbracket \Longrightarrow \vdash \{P\} (\text{Seq } C1 C2) \{Q\}$
 $| \text{RuleIf}: \llbracket \vdash \{P\} C1 \{Q1\} ; \vdash \{P\} C2 \{Q2\} \rrbracket \Longrightarrow \vdash \{P\} (\text{If } C1 C2) \{\text{join } Q1 Q2\}$
 $| \text{RuleWhile}: \llbracket \bigwedge n. \vdash \{In\} C \{I (Suc n)\} \rrbracket \Longrightarrow \vdash \{I 0\} (\text{While } C) \{\text{natural-partition } I\}$
 $| \text{RuleAssume}: \vdash \{ (\lambda S. P (\text{Set.filter } (b \circ \text{snd}) S)) \} (\text{Assume } b) \{P\}$
 $| \text{RuleAssign}: \vdash \{ (\lambda S. P \{ (l, \sigma(x := e \sigma)) \mid l \sigma. (l, \sigma) \in S \}) \} (\text{Assign } x e) \{P\}$
 $| \text{RuleHavoc}: \vdash \{ (\lambda S. P \{ (l, \sigma(x := v)) \mid l \sigma v. (l, \sigma) \in S \}) \} (\text{Havoc } x) \{P\}$
 $| \text{RuleExistsSet}: \llbracket \bigwedge x::(\text{'lvar}, \text{'lval}, \text{'pvar}, \text{'pval}) \text{ state set}. \vdash \{P x\} C \{Q x\} \rrbracket \Longrightarrow \vdash \{\text{exists } P\} C \{\text{exists } Q\}$

2.2 Soundness

Definition 5: Hyper-Triples

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

lemma *hyper-hoare-tripleI*:
assumes $\bigwedge S. P S \Longrightarrow Q (\text{sem } C S)$
shows $\models \{P\} C \{Q\}$
 $\langle \text{proof} \rangle$

lemma *hyper-hoare-tripleE*:
assumes $\models \{P\} C \{Q\}$
and $P S$
shows $Q (\text{sem } C S)$
 $\langle \text{proof} \rangle$

lemma *consequence-rule*:
assumes $\text{entails } P P'$
and $\text{entails } Q' Q$
and $\models \{P'\} C \{Q'\}$
shows $\models \{P\} C \{Q\}$
 $\langle \text{proof} \rangle$

lemma *skip-rule*:
 $\models \{P\} \text{ Skip } \{P\}$
 $\langle \text{proof} \rangle$

lemma *assume-rule*:
 $\models \{ (\lambda S. P (\text{Set.filter } (b \circ \text{snd}) S)) \} (\text{Assume } b) \{P\}$
 $\langle \text{proof} \rangle$

lemma *seq-rule*:

assumes $\models \{P\} C1 \{R\}$
and $\models \{R\} C2 \{Q\}$
shows $\models \{P\} Seq C1 C2 \{Q\}$
 $\langle proof \rangle$

lemma if-rule:

assumes $\models \{P\} C1 \{Q1\}$
and $\models \{P\} C2 \{Q2\}$
shows $\models \{P\} If C1 C2 \{join Q1 Q2\}$
 $\langle proof \rangle$

lemma sem-assign:

$sem (Assign x e) S = \{(l, \sigma(x := e \sigma)) \mid l \sigma. (l, \sigma) \in S\}$ (**is** $?A = ?B$)
 $\langle proof \rangle$

lemma assign-rule:

$\models \{ (\lambda S. P \{ (l, \sigma(x := e \sigma)) \mid l \sigma. (l, \sigma) \in S \}) \} (Assign x e) \{P\}$
 $\langle proof \rangle$

lemma sem-havoc:

$sem (Havoc x) S = \{(l, \sigma(x := v)) \mid l \sigma v. (l, \sigma) \in S\}$ (**is** $?A = ?B$)
 $\langle proof \rangle$

lemma havoc-rule:

$\models \{ (\lambda S. P \{ (l, \sigma(x := v)) \mid l \sigma v. (l, \sigma) \in S \}) \} (Havoc x) \{P\}$
 $\langle proof \rangle$

Loops

lemma indexed-invariant-then-power:

assumes $\bigwedge n. hyper\text{-}hoare\text{-}triple (I n) C (I (Suc n))$
and $I 0 S$
shows $I n (iterate\text{-}sem n C S)$
 $\langle proof \rangle$

lemma indexed-invariant-then-power-bounded:

assumes $\bigwedge m. m < n \implies hyper\text{-}hoare\text{-}triple (I m) C (I (Suc m))$
and $I 0 S$
shows $I n (iterate\text{-}sem n C S)$
 $\langle proof \rangle$

lemma while-rule:

assumes $\bigwedge n. hyper\text{-}hoare\text{-}triple (I n) C (I (Suc n))$
shows $hyper\text{-}hoare\text{-}triple (I 0) (While C) (natural\text{-}partition I)$
 $\langle proof \rangle$

lemma rule-exists:

assumes $\bigwedge x. \models \{P x\} C \{Q x\}$
shows $\models \{exists P\} C \{exists Q\}$
 $\langle proof \rangle$

Theorem 1

theorem *soundness*:

assumes $\vdash \{A\} C \{B\}$
shows $\models \{A\} C \{B\}$
<proof>

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* $P C Q$
<proof>

lemma *completeE*:

assumes *complete* $P C Q$
and $\models \{P\} C \{Q\}$
shows $\vdash \{P\} C \{Q\}$
<proof>

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>

lemma *complete-if*:

fixes $P Q :: ('lvar, 'lval, 'pvar, 'pval)$ *state hyperassertion*
assumes $\bigwedge P1 Q1 :: ('lvar, 'lval, 'pvar, 'pval)$ *state hyperassertion. complete* $P1 C1 Q1$
and $\bigwedge P2 Q2 :: ('lvar, 'lval, 'pvar, 'pval)$ *state hyperassertion. complete* $P2 C2 Q2$
shows *complete* P (*If* $C1 C2$) Q
<proof>

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>

lemma *complete-assume*:

complete P (*Assume* b) Q
<proof>

lemma *complete-skip*:

complete P *Skip* Q

<proof>

lemma *complete-assign:*

complete P (Assign x e) Q

<proof>

lemma *complete-havoc:*

complete P (Havoc x) Q

<proof>

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*

<proof>

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 $\exists S. P \ S \wedge S' = \text{iterate-sem } n \ C \ S$

<proof>

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>

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>

Theorem 2

theorem *completeness:*

fixes $P \ Q :: ('lvar, 'lval, 'pvar, 'pval) \text{ state hyperassertion}$

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

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

<proof>

2.4 Disproving Hyper-Triples

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

Theorem 5

theorem *disproving-triple:*

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

definition *differ-only-by where*

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* $a b x$

$\langle proof \rangle$

lemma *diff-by-update:*

differ-only-by $(a(x := v)) a x$

$\langle proof \rangle$

lemma *diff-by-comm:*

differ-only-by $a b x \longleftrightarrow$ *differ-only-by* $b a x$

$\langle proof \rangle$

lemma *diff-by-trans:*

assumes *differ-only-by* $a b x$

and *differ-only-by* $b c x$

shows *differ-only-by* $a c x$

$\langle proof \rangle$

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 $states' \in P$

$\langle proof \rangle$

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 $combine\ from\ nat\ x\ P1\ P2\ S$

<proof>

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 $fst\ \varphi'\ x = a$

<proof>

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>

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>

definition *not-free-var-hyper where*

$not\ free\ var\ hyper\ x\ P \longleftrightarrow (\forall S\ v.\ P\ S \longleftrightarrow P\ ((modify\ lvar\ to\ x\ v)\ 'S))$

definition *injective where*

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

lemma *sem-of-modify-lvar:*

$sem\ C\ ((modify\ lvar\ to\ r\ v)\ 'S) = (modify\ lvar\ to\ r\ v)\ '(sem\ C\ S)\ (is\ ?A = ?B)$

<proof>

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⟩
```

```
lemma single-step-then-in-sem:
  assumes single-sem C σ σ'
  and (l, σ) ∈ S
  shows (l, σ') ∈ sem C S
  ⟨proof⟩
```

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

```
lemma in-set-of-traces-then-in-sem:
  assumes (σ, σ') ∈ set-of-traces C
  and (l, σ) ∈ S
```

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

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 $S = \{(copy-p-state \ \text{to-pvar } \ \text{to-lval } \ \sigma, \ \sigma) \mid \sigma. \ \text{True}\}$
shows $\{(recover-p-state \ \text{to-pval } \ \text{to-lvar } \ l, \ \sigma') \mid l \ \sigma'. \ (l, \ \sigma') \in \text{sem } C \ S\} =$
set-of-traces C
(is $?A = ?B)$
 ⟨proof⟩

Theorem 3

theorem *proving-hyperproperties*:

fixes $\text{to-lvar} :: 'pvar \Rightarrow 'lvar$
fixes $\text{to-lval} :: 'pval \Rightarrow 'lval$

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

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

Hypersafety, hyperliveness

definition *max-k where*

$\text{max-k } k \ S \longleftrightarrow \text{finite } S \wedge \text{card } S \leq k$

definition *hypersafety where*

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

definition *k-hypersafety where*

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

definition *hyperliveness where*

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

lemma *k-hypersafetyI*:

assumes $\bigwedge S. \ \neg P \ S \Longrightarrow \exists S'. \ S' \subseteq S \wedge \text{max-k } k \ S' \wedge (\forall S''. \ S' \subseteq S'' \longrightarrow \neg P \ S'')$
shows $k\text{-hypersafety } k \ P$
 ⟨proof⟩

lemma *hypersafetyI*:

assumes $\bigwedge S \ S'. \ \neg P \ S \Longrightarrow S \subseteq S' \Longrightarrow \neg P \ S'$
shows $\text{hypersafety } P$

$\langle \text{proof} \rangle$

lemma *hyperlivenessI*:

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

shows *hyperliveness* P

$\langle \text{proof} \rangle$

lemma *k-hypersafe-is-hypersafe*:

assumes *k-hypersafety* $k P$

shows *hypersafety* P

$\langle \text{proof} \rangle$

lemma *one-safety-equiv*:

assumes *sat* H

shows *k-hypersafety* $1 H \longleftrightarrow (\exists P. \forall S. H S \longleftrightarrow (\forall \tau \in S. P \tau))$ (**is** $?A \longleftrightarrow ?B$)

$\langle \text{proof} \rangle$

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* $P Q$)

$\langle \text{proof} \rangle$

theorem *hypersafety-1-hoare-logic*:

k-hypersafety 1 (*hoarify* $P Q$)

$\langle \text{proof} \rangle$

definition *incorrectnessify where*

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

lemma *incorrectnessify-liveness*:

assumes $P \neq \{\}$

shows *hyperliveness* (*incorrectnessify* $P Q$)

$\langle \text{proof} \rangle$

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* $P Q$)

$\langle \text{proof} \rangle$

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** $?A \longleftrightarrow ?B$)
 $\langle \text{proof} \rangle$

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* $P\ C\ Q$
 $\langle \text{proof} \rangle$

lemma *hoarifyI*:
assumes $\bigwedge \sigma\ \sigma'. (\sigma, \sigma') \in S \implies \sigma \in P \implies \sigma' \in Q$
shows *hoarify* $P\ Q\ S$
 $\langle \text{proof} \rangle$

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

lemma *connection-HL*:
 $HL\ P\ C\ Q \longleftrightarrow HL\text{-hyperprop}\ P\ Q\ (\text{set-of-traces}\ C)$ (**is** $?A \longleftrightarrow ?B$)
 $\langle \text{proof} \rangle$

Proposition 1 **theorem** *HL-expresses-hyperproperties*:
 $\exists H. (\forall C. \text{hypersat}\ C\ H \longleftrightarrow HL\ P\ C\ Q) \wedge k\text{-hypersafety}\ 1\ H$
 $\langle \text{proof} \rangle$

Proposition 2 theorem encoding-HL:

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

lemma entailment-order-hoare:

assumes $P \subseteq P'$
shows $entails\ (over\text{-}approx\ P)\ (over\text{-}approx\ P')$
 ⟨proof⟩

3.3 Cartesian Hoare Logic (CHL) [10]

definition k-sem where

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

lemma k-semI:

assumes $\bigwedge i. (fst\ (\text{states } i) = fst\ (\text{states}'\ i) \wedge single\text{-}sem\ C\ (snd\ (\text{states } i))\ (snd\ (\text{states}'\ i)))$
shows $k\text{-sem } C\ \text{states}\ \text{states}'$
 ⟨proof⟩

lemma k-semE:

assumes $k\text{-sem } C\ \text{states}\ \text{states}'$
shows $fst\ (\text{states } i) = fst\ (\text{states}'\ i) \wedge single\text{-}sem\ C\ (snd\ (\text{states } i))\ (snd\ (\text{states}'\ i))$
 ⟨proof⟩

Definition 17 definition CHL where

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

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 $CHL\ P\ C\ Q$
 ⟨proof⟩

lemma CHLE:

assumes $CHL\ P\ C\ Q$
and $\text{states} \in P$
and $k\text{-sem } C\ \text{states}\ \text{states}'$
shows $\text{states}' \in Q$
 ⟨proof⟩

definition encode-CHL where

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

lemma *encode-CHLI*:

assumes $\bigwedge \text{states}. (\forall i. \text{states } i \in S \wedge \text{fst } (\text{states } i) \ x = \text{from-nat } i) \implies \text{states} \in P$
shows *encode-CHL from-nat x P S*
<proof>

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 $\text{states} \in P$
<proof>

lemma *equal-change-lvar*:

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

Proposition 3 theorem *encoding-CHL*:

assumes *not-free-var-of P x*
and *not-free-var-of Q x*
and *injective from-nat*
shows $\text{CHL } P \ C \ Q \longleftrightarrow \models \{ \text{encode-CHL from-nat } x \ P \} \ C \ \{ \text{encode-CHL from-nat } x \ Q \}$ (**is** $?A \longleftrightarrow ?B$)
<proof>

definition *CHL-hyperprop where*

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

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*
<proof>

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 $(\lambda i. (l \ i, \text{snd } (p \ i))) \in Q$
<proof>

Proposition 3 theorem *CHL-hyperproperty*:

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

theorem *k-hypersafety-IL-hyperprop*:
fixes $P :: ('i \Rightarrow ('lvar, 'lval, 'pvar, 'pval) \text{ state}) \text{ set}$
assumes *finite* ($UNIV :: 'i \text{ set}$)
and $\text{card } (UNIV :: 'i \text{ set}) = k$
shows *k-hypersafety k (CHL-hyperprop P Q)*
 $\langle \text{proof} \rangle$

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*:

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

$\langle \text{proof} \rangle$

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*:

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

shows *IL-hyperprop P Q S*
 $\langle \text{proof} \rangle$

Proposition 5 **lemma** *IL-expresses-hyperproperties*:

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

$\langle \text{proof} \rangle$

lemma *IL-consequence*:

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

and $(l, \sigma') \in Q$
shows $\exists \sigma. (l, \sigma) \in P \wedge \text{single-sem } C\ \sigma\ \sigma'$
 $\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

lemma *entailment-order-reverse-hoare*:

$$\text{assumes } P \subseteq P'$$

shows *entails (under-approx P') (under-approx P)*
 $\langle \text{proof} \rangle$

definition *incorrectify* where

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

lemma *incorrectifyI*:
assumes $\bigwedge \sigma. p \sigma \implies \sigma \in S$
shows *incorrectify* $p S$
 $\langle \text{proof} \rangle$

lemma *incorrectifyE*:
assumes *incorrectify* $p S$
and $p \sigma$
shows $\sigma \in S$
 $\langle \text{proof} \rangle$

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* $(\text{incorrectify } (p \ 0)) \ (\text{While } C) \ (\text{incorrectify } (\lambda \sigma. \exists n. p \ n \ \sigma))$
 $\langle \text{proof} \rangle$

definition *sat-for-l where*
 $\text{sat-for-l } l \ P \longleftrightarrow (\exists \sigma. (l, \sigma) \in P)$

theorem *incorrectness-hyperliveness*:
assumes $\bigwedge l. \text{sat-for-l } l \ Q \implies \text{sat-for-l } l \ P$
shows *hyperliveness* $(\text{IL-hyperprop } P \ Q)$
 $\langle \text{proof} \rangle$

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

RIL is the old name of k-IL.

Definition 19 *definition RIL where*
 $\text{RIL } P \ C \ Q \longleftrightarrow (\forall \text{states}' \in Q. \exists \text{states} \in P. k\text{-sem } C \ \text{states} \ \text{states}')$

lemma *RILI*:
assumes $\bigwedge \text{states}'. \text{states}' \in Q \implies (\exists \text{states} \in P. k\text{-sem } C \ \text{states} \ \text{states}')$
shows *RIL* $P \ C \ Q$
 $\langle \text{proof} \rangle$

lemma *RILE*:
assumes *RIL* $P \ C \ Q$
and $\text{states}' \in Q$
shows $\exists \text{states} \in P. k\text{-sem } C \ \text{states} \ \text{states}'$
 $\langle \text{proof} \rangle$

definition *RIL-hyperprop where*
 $\text{RIL-hyperprop } P \ Q \ S \longleftrightarrow (\forall l \ \text{states}'. (\lambda i. (l \ i, \ \text{states}' \ i)) \in Q$
 $\longrightarrow (\exists \text{states}. (\lambda i. (l \ i, \ \text{states} \ i)) \in P \wedge (\forall i. (\text{states} \ i, \ \text{states}' \ i) \in S)))$

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* $P \ Q \ S$
 $\langle \text{proof} \rangle$

lemma *RIL-hyperpropE*:

assumes *RIL-hyperprop* $P \ Q \ S$
and $(\lambda i . (l \ i, \text{states}' \ i)) \in Q$
shows $\exists \text{states} . (\lambda i . (l \ i, \text{states} \ i)) \in P \wedge (\forall i . (\text{states} \ i, \text{states}' \ i) \in S)$
 $\langle \text{proof} \rangle$

lemma *useful*:

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

Proposition 17 theorem *RIL-expresses-hyperproperties*:

hypersat $C \ (RIL\text{-hyperprop} \ P \ Q) \longleftrightarrow RIL \ P \ C \ Q \ (\text{is} \ ?A \longleftrightarrow ?B)$
 $\langle \text{proof} \rangle$

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* $(RIL\text{-hyperprop} \ P \ Q)$
 $\langle \text{proof} \rangle$

definition *strong-pre-insec where*

strong-pre-insec 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 from-nat* $x \ c \ P \ S$
 $\langle \text{proof} \rangle$

lemma *strong-pre-insecE*:

assumes *strong-pre-insec 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 $states \in P$
<proof>

definition *pre-insec* **where**

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

lemma *pre-insecI*:

assumes $\bigwedge\ states. states \in P \implies (\forall\ i. fst\ (states\ i)\ x = from-nat\ i)$
 $\implies (\exists\ r. \forall\ i. ((fst\ (states\ i))(c := r), snd\ (states\ i)) \in S)$
shows *pre-insec from-nat* $x\ c\ P\ S$
<proof>

lemma *strong-pre-implies-pre*:

assumes *strong-pre-insec from-nat* $x\ c\ P\ S$
shows *pre-insec from-nat* $x\ c\ P\ S$
<proof>

lemma *pre-insecE*:

assumes *pre-insec from-nat* $x\ c\ P\ S$
and $states \in P$
and $\bigwedge\ i. fst\ (states\ i)\ x = from-nat\ i$
shows $\exists\ r. \forall\ i. ((fst\ (states\ i))(c := r), snd\ (states\ i)) \in S$
<proof>

definition *post-insec* **where**

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

lemma *post-insecE*:

assumes *post-insec from-nat* $x\ c\ Q\ S$
and $states \in Q$
and $\bigwedge\ i. fst\ (states\ i)\ x = from-nat\ i$
shows $\exists\ r. (\forall\ i. ((fst\ (states\ i))(c := r), snd\ (states\ i)) \in S)$
<proof>

lemma *post-insecI*:

assumes $\bigwedge\ states. states \in Q \implies (\forall\ i. fst\ (states\ i)\ x = from-nat\ i)$
 $\implies (\exists\ r. (\forall\ i. ((fst\ (states\ i))(c := r), snd\ (states\ i)) \in S))$
shows *post-insec from-nat* $x\ c\ Q\ S$
<proof>

lemma *same-pre-post*:

pre-insec from-nat $x c Q S \longleftrightarrow$ *post-insec from-nat* $x c Q S$
 \langle proof \rangle

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* :: $((l' a \Rightarrow ('pvar \Rightarrow 'pval)) \Rightarrow 'lval)$)
and $x \neq c$
and *injective from-nat*
shows *sat* (*strong-pre-insec from-nat* $x c (P :: (l' a \Rightarrow ('lvar \Rightarrow 'lval)) \times ('pvar \Rightarrow 'pval))$ set))
 \langle proof \rangle

theorem *encode-insec*:

assumes *injective from-nat*
and *sat* (*strong-pre-insec from-nat* $x c (P :: (l' a \Rightarrow ('lvar \Rightarrow 'lval)) \times ('pvar \Rightarrow 'pval))$ set))
and *not-free-var-of* $P x \wedge$ *not-free-var-of* $P c$
and *not-free-var-of* $Q x \wedge$ *not-free-var-of* $Q c$
and $c \neq x$

shows *RIL* $P C Q \longleftrightarrow \models \{pre-insec from-nat x c P\} C \{post-insec from-nat x c Q\}$ (is $?A \longleftrightarrow ?B$)
 \langle proof \rangle

Proposition 8 **theorem** *encoding-RIL*:

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* :: $((l' a \Rightarrow ('pvar \Rightarrow 'pval)) \Rightarrow 'lval)$)
and $c \neq x$
and *injective from-nat*
and *not-free-var-of* $(P :: (l' a \Rightarrow ('lvar \Rightarrow 'lval)) \times ('pvar \Rightarrow 'pval))$ 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* $P C Q \longleftrightarrow \models \{pre-insec from-nat x c P\} C \{post-insec from-nat x c Q\}$ (is $?A \longleftrightarrow ?B$)
 \langle proof \rangle

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* $P C Q$

<proof>

definition *encode-FU* **where**

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

Proposition 9 theorem *encoding-FU*:

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

<proof>

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 $hyperprop-FU P Q S$

<proof>

lemma *hyperprop-FUE*:

assumes $hyperprop-FU P Q S$

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

shows $\exists \sigma'. (l, \sigma') \in Q \wedge (\sigma, \sigma') \in S$

<proof>

theorem *FU-expresses-hyperproperties*:

$hypersat C (hyperprop-FU P Q) \longleftrightarrow FU P C Q$ (**is** $?A \longleftrightarrow ?B$)

<proof>

theorem *hyperliveness-hyperprop-FU*:

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

shows $hyperliveness (hyperprop-FU P Q)$

<proof>

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

assumes *injective from-nat*

assumes $P = \{(l, \sigma) \mid \sigma l. \sigma x = from-nat (0 :: nat) \vee \sigma x = from-nat 1\}$

and $Q = \{(l, \sigma) \mid \sigma l. \sigma x = from-nat 1\}$

and $C = Assume (\lambda \sigma. \sigma x = from-nat 1)$

shows $IL P C Q$

and $\neg FU P C Q$

<proof>

lemma *FU-does-not-imply-incorrectness*:

assumes $P = \{(l, \sigma) \mid \sigma l. \sigma x = from-nat (0 :: nat) \vee \sigma x = from-nat 1\}$

and $Q = \{(l, \sigma) \mid \sigma l. \sigma x = from-nat 1\}$

assumes *injective from-nat*

shows $\neg IL Q Skip P$

and $FU Q Skip P$

<proof>

3.7 k-Forward Underapproximate logic

RFU is the old name of k-FU.

Definition 21 definition *RFU* where

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

lemma *RFUI*:

$$\text{assumes } \bigwedge\ states. states \in P \implies (\exists\ states' \in Q. k\text{-sem}\ C\ states\ states')$$

shows *RFU P C Q*

<proof>

lemma *RFUE*:

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

and $states \in P$

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

<proof>

definition *encode-RFU* where

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

Proposition 11 theorem *encode-RFU*:

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

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

and *injective from-nat*

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

(is ?A \longleftrightarrow ?B)

<proof>

definition *RFU-hyperprop* where

$$RFU\text{-hyperprop } P\ Q\ S \longleftrightarrow (\forall\ l\ states. (\lambda i. (l\ i, states\ i)) \in P$$

$$\longrightarrow (\exists\ states'. (\lambda i. (l\ i, states'\ i)) \in Q \wedge (\forall\ i. (states\ i, states'\ i) \in S)))$$

lemma *RFU-hyperpropI*:

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

shows *RFU-hyperprop P Q S*

<proof>

lemma *RFU-hyperpropE*:

$$\text{assumes } RFU\text{-hyperprop } P\ Q\ S$$

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

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

<proof>

Proposition 10 theorem *RFU-captures-hyperproperties*:

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

<proof>

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>

3.8 k-Universal Existential (RUE) [5]

RUE is the old name of k-UE.

Definition 22 **definition** *RUE* where

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

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* $P \ C \ Q$

<proof>

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

<proof>

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

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

$(\forall i. (\sigma 1 \ i, \sigma 1' \ i) \in S) \implies (\exists \sigma 2'. (\forall k. (\sigma 2 \ k, \sigma 2' \ k) \in S) \wedge (\lambda i. (l1 \ i, \sigma 1' \ i), \lambda k. (l2 \ k, \sigma 2' \ k)) \in Q))$

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-RUE* $P \ Q \ S$

<proof>

lemma *hyperprop-RUE-E*:

assumes *hyperprop-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

<proof>

Proposition 12 theorem RUE-express-hyperproperties:

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

<proof>

definition is-type where

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

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 $is\text{-}type\ type\ fn\ x\ t\ S\ \sigma$

<proof>

lemma is-type-E:

assumes $is\text{-}type\ type\ fn\ x\ t\ S\ \sigma$

shows $\sigma\ i \in S \wedge fst\ (\sigma\ i)\ t = type \wedge fst\ (\sigma\ i)\ x = fn\ i$

<proof>

definition encode-RUE-1 where

$encode\text{-}RUE\text{-}1\ fn\ fn1\ fn2\ x\ t\ P\ S \longleftrightarrow (\forall k. \exists \sigma \in S. fst\ \sigma\ x = fn2\ k \wedge fst\ \sigma\ t = fn\ 2)$

$\wedge (\forall \sigma\ \sigma'. is\text{-}type\ (fn\ 1)\ fn1\ x\ t\ S\ \sigma \wedge is\text{-}type\ (fn\ 2)\ fn2\ x\ t\ S\ \sigma'$

$\longrightarrow (\sigma, \sigma') \in P)$

lemma encode-RUE-1-I:

assumes $\bigwedge k. \exists \sigma \in S. fst\ \sigma\ x = fn2\ k \wedge fst\ \sigma\ t = fn\ 2$

and $\bigwedge \sigma\ \sigma'. is\text{-}type\ (fn\ 1)\ fn1\ x\ t\ S\ \sigma \wedge is\text{-}type\ (fn\ 2)\ fn2\ x\ t\ S\ \sigma'$

$\implies (\sigma, \sigma') \in P$

shows $encode\text{-}RUE\text{-}1\ fn\ fn1\ fn2\ x\ t\ P\ S$

<proof>

lemma encode-RUE-1-E1:

assumes $encode\text{-}RUE\text{-}1\ fn\ fn1\ fn2\ x\ t\ P\ S$

shows $\exists \sigma \in S. fst\ \sigma\ x = fn2\ k \wedge fst\ \sigma\ t = fn\ 2$

<proof>

lemma encode-RUE-1-E2:

assumes $encode\text{-}RUE\text{-}1\ fn\ fn1\ fn2\ x\ t\ P\ S$

and $is\text{-}type\ (fn\ 1)\ fn1\ x\ t\ S\ \sigma$

and $is\text{-}type\ (fn\ 2)\ fn2\ x\ t\ S\ \sigma'$

shows $(\sigma, \sigma') \in P$

<proof>

definition encode-RUE-2 where

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

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* $fn\ fn1\ fn2 \ x \ t \ Q \ S$
 $\langle \text{proof} \rangle$

lemma *encode-RUE-2-E*:

assumes *encode-RUE-2* $fn\ fn1\ fn2 \ x \ t \ Q \ S$
and *is-type* $(fn\ 1) \text{ } fn1 \ x \ t \ S \ \sigma$
shows $\exists \sigma'. \text{is-type } (fn\ 2) \text{ } fn2 \ x \ t \ S \ \sigma' \wedge (\sigma, \sigma') \in Q$
 $\langle \text{proof} \rangle$

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

differ-only-by-set $vars\ a\ b \longleftrightarrow (\forall x. x \notin vars \longrightarrow a\ x = b\ x)$

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

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

lemma *differ-only-by-lsetI*:

assumes $\bigwedge i. \text{snd } (a\ i) = \text{snd } (b\ i)$
and $\bigwedge i. \text{differ-only-by-set } vars\ (\text{fst } (a\ i))\ (\text{fst } (b\ i))$
shows *differ-only-by-lset* $vars\ a\ b$
 $\langle \text{proof} \rangle$

definition *not-in-free-vars-double* **where**

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

lemma *not-in-free-vars-doubleE*:

assumes *not-in-free-vars-double* $vars\ P$
and *differ-only-by-lset* $vars\ (\text{fst } \sigma)\ (\text{fst } \sigma')$
and *differ-only-by-lset* $vars\ (\text{snd } \sigma)\ (\text{snd } \sigma')$
and $\sigma \in P$
shows $\sigma' \in P$
 $\langle \text{proof} \rangle$

Proposition 13 **theorem** *encoding-RUE*:

assumes *injective* $fn \wedge \text{injective } fn1 \wedge \text{injective } fn2$
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 $?A \longleftrightarrow ?B$)
 $\langle proof \rangle$

3.9 Program Refinement

lemma *sem-assign-single*:

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

definition *refinement where*

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

definition *not-free-var-stmt where*

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

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

assumes $not-free-var-stmt\ x\ C$
and $(\sigma,\ \sigma') \in set-of-traces\ C$
shows $(\sigma(x := v), \sigma'(x := v)) \in set-of-traces\ C$
 $\langle proof \rangle$

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

assumes $not-free-var-stmt\ x\ C$
and $single-sem\ C\ \sigma\ \sigma'$
shows $\sigma\ x = \sigma'\ x$
 $\langle proof \rangle$

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

assumes $not-free-var-stmt\ x\ C$
and $single-sem\ C\ \sigma\ \sigma'$
shows $single-sem\ C\ (\sigma(x := v))\ (\sigma'(x := v))$
 $\langle proof \rangle$

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

assumes $\models \{P\}\ If\ C1\ C2\ \{Q\}$
shows $\models \{P\}\ If\ C2\ C1\ \{Q\}$
 $\langle proof \rangle$

theorem *encoding-refinement*:

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

and $P = (\lambda S.\ card\ S = 1)$
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* $C1 \ C2 \longleftrightarrow$
 $\models \{ P \} \text{ If } (\text{Seq } (\text{Assign } (x :: \text{'pvar'} (\lambda-. a)) \ C1) (\text{Seq } (\text{Assign } x (\lambda-. b)) \ C2)) \{$
 $Q \}$
(is $?A \longleftrightarrow ?B$
 $\langle \text{proof} \rangle$

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 $NC \ P \ C \ Q$

$\langle \text{proof} \rangle$

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*:

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

$\langle \text{proof} \rangle$

lemma *equiv-def-FU*:

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

$\langle \text{proof} \rangle$

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

$NC \ P \ C \ Q \longleftrightarrow \models \{ (\lambda S. S = \neg P) \} \ C \ \{ (\lambda S. S \cap Q = \{ \}) \} \text{ (is } ?A \longleftrightarrow ?B)$
 $\langle \text{proof} \rangle$

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**

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

definition *low-exp* **where**

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

lemma *low-exp-lnot*:

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

<proof>

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

<proof>

lemma *low-exp-two-cases*:

assumes *low-exp* b S

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

<proof>

lemma *sem-assume-low-exp*:

assumes *holds-forall* b S

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

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

<proof>

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) $S = \{\}$

<proof>

lemma *lnot-involution*:

lnot (*lnot* b) = b

<proof>

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

<proof>

lemma *if-synchronized-aux*:

assumes $\models \{P\} C1 \{Q\}$

and $\models \{P\} C2 \{Q\}$

and *entails* P (*low-exp* b)

shows $\models \{P\} \text{if-then-else } b C1 C2 \{Q\}$

<proof>

theorem *if-synchronized*:

assumes $\models \{\text{conj } P (\text{holds-forall } b)\} C1 \{Q\}$

and $\models \{\text{conj } P (\text{holds-forall } (\text{lnot } b))\} C2 \{Q\}$

shows $\models \{\text{conj } P (\text{low-exp } b)\} \text{if-then-else } b C1 C2 \{Q\}$

$\langle \text{proof} \rangle$

definition *while-cond where*

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

lemma *while-synchronized-rec:*

assumes $\bigwedge n. \models \{\text{conj } (I \ n) \ (\text{holds-forall } b)\} \ \text{Assume } b;; C \ \{\text{conj } (I \ (\text{Suc } n)) \ (\text{low-exp } b)\}$
and $\text{conj } (I \ 0) \ (\text{low-exp } b) \ S$
shows $\text{conj } (I \ n) \ (\text{low-exp } b) \ (\text{iterate-sem } n \ (\text{Assume } b;; C) \ S) \vee \text{holds-forall} \ (\text{lnot } b) \ (\text{iterate-sem } n \ (\text{Assume } b;; C) \ S)$
 $\langle \text{proof} \rangle$

lemma *false-then-empty-later:*

assumes $\text{holds-forall} \ (\text{lnot } b) \ (\text{iterate-sem } n \ (\text{Assume } b;; C) \ S)$
and $m > n$
shows $\text{iterate-sem } m \ (\text{Assume } b;; C) \ S = \{\}$
 $\langle \text{proof} \rangle$

lemma *split-union-triple:*

$(\bigcup (m::\text{nat}). f \ m) = (\bigcup m \in \{m \mid m. m < n\}. f \ m) \cup f \ n \cup (\bigcup m \in \{m \mid m. m > n\}. f \ m)$ **(is** $?A = ?B)$
 $\langle \text{proof} \rangle$

lemma *sem-union-swap:*

$\text{sem } C \ (\bigcup x \in S. f \ x) = (\bigcup x \in S. \text{sem } C \ (f \ x))$ **(is** $?A = ?B)$
 $\langle \text{proof} \rangle$

lemma *while-synchronized-case-1:*

assumes $\bigwedge m. m < n \implies \text{holds-forall } b \ (\text{iterate-sem } m \ (\text{Assume } b;; C) \ S)$
and $\text{holds-forall} \ (\text{lnot } b) \ (\text{iterate-sem } n \ (\text{Assume } b;; C) \ S)$
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 $\text{sem } (\text{while-cond } b \ C) \ S = \text{iterate-sem } n \ (\text{Assume } b;; C) \ S$
 $\langle \text{proof} \rangle$

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 $\text{sem } (\text{while-cond } b \ C) \ S = \{\}$
 $\langle \text{proof} \rangle$

definition *emp* where

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

lemma *holds-forall-empty*:

$$\text{holds-forall } b \ \{\} \\ \langle \text{proof} \rangle$$

definition *exists* where

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

theorem *while-synchronized*:

$$\text{assumes } \bigwedge n. \models \{\text{conj } (I \ n) \ (\text{holds-forall } b)\} \ C \ \{\text{conj } (I \ (\text{Suc } n)) \ (\text{low-exp } b)\} \\ \text{shows } \models \{\text{conj } (I \ 0) \ (\text{low-exp } b)\} \ \text{while-cond } b \ C \ \{\text{conj } (\text{disj } (\text{exists } I) \ \text{emp}) \\ (\text{holds-forall } (\text{lnot } b))\} \\ \langle \text{proof} \rangle$$

lemma *WhileSync-simpler*:

$$\text{assumes } \models \{\text{conj } I \ (\text{holds-forall } b)\} \ C \ \{\text{conj } I \ (\text{low-exp } b)\} \\ \text{shows } \models \{\text{conj } I \ (\text{low-exp } b)\} \ \text{while-cond } b \ C \ \{\text{conj } (\text{disj } I \ \text{emp}) \ (\text{holds-forall} \\ (\text{lnot } b))\} \\ \langle \text{proof} \rangle$$

definition *if-then* where

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

definition *filter-exp* where

$$\text{filter-exp } b \ S = \text{Set.filter } (b \circ \text{snd}) \ S$$

lemma *filter-exp-union*:

$$\text{filter-exp } b \ (S1 \cup S2) = \text{filter-exp } b \ S1 \cup \text{filter-exp } b \ S2 \ (\text{is } ?A = ?B) \\ \langle \text{proof} \rangle$$

lemma *filter-exp-union-general*:

$$\text{filter-exp } b \ (\bigcup x. \ f \ x) = (\bigcup x. \ \text{filter-exp } b \ (f \ x)) \ (\text{is } ?A = ?B) \\ \langle \text{proof} \rangle$$

lemma *filter-exp-contradict*:

$$\text{filter-exp } b \ (\text{filter-exp } (\text{lnot } b) \ S) = \{\} \\ \langle \text{proof} \rangle$$

lemma *filter-exp-same*:

$$\text{filter-exp } b \ (\text{filter-exp } b \ S) = \text{filter-exp } b \ S \ (\text{is } ?A = ?B) \\ \langle \text{proof} \rangle$$

lemma *if-then-sem*:

$$\text{sem } (\text{if-then } b \ C) \ S = \text{sem } C \ (\text{filter-exp } b \ S) \cup \text{filter-exp } (\text{lnot } b) \ S \\ \langle \text{proof} \rangle$$

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

lemma *union-up-to-increasing*:
assumes $m \leq n$
shows *union-up-to-n* $C\ S\ m \subseteq \text{union-up-to-n } C\ S\ n$
⟨*proof*⟩

lemma *union-union-up-to-n-equiv-aux*:
union-up-to-n $C\ S\ n \subseteq (\bigcup m. \text{iterate-sem } m\ C\ S)$
⟨*proof*⟩

lemma *union-union-up-to-n-equiv*:
 $(\bigcup n. \text{union-up-to-n } C\ S\ n) = (\bigcup n. \text{iterate-sem } n\ C\ S)$ (**is** $?A = ?B$)
⟨*proof*⟩

lemma *filter-exp-union-itself*:
filter-exp $b\ S \cup S = S$
⟨*proof*⟩

lemma *iterate-sem-equiv*:
iterate-sem $m\ (\text{if-then } b\ C)\ S$
= *filter-exp* $(\text{lnot } b)\ (\text{union-up-to-n } (\text{Assume } b;;\ C)\ S\ m) \cup \text{iterate-sem } m\ (\text{Assume } b;;\ C)\ S$
⟨*proof*⟩

lemma *sem-while-with-if*:
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*⟩

lemma *iterate-sem-assume-increasing*:
filter-exp $(\text{lnot } b)\ (\text{iterate-sem } n\ (\text{if-then } b\ C)\ S) \subseteq \text{filter-exp } (\text{lnot } b)\ (\text{iterate-sem } (\text{Suc } n)\ (\text{if-then } b\ C)\ S)$
⟨*proof*⟩

lemma *iterate-sem-assume-increasing-union-up-to*:
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*⟩

definition *ascending* :: $(\text{nat} \Rightarrow 'b\ \text{set}) \Rightarrow \text{bool}$ **where**
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
<proof>

lemma *ascendingI*:
assumes $\bigwedge n. S\ n \subseteq S\ (Suc\ n)$
shows *ascending* S
<proof>

definition *upwards-closed where*
upwards-closed $P\ P\text{-inf} \longleftrightarrow (\forall S. \textit{ascending}\ S \wedge (\forall n. P\ n\ (S\ n)) \longrightarrow P\text{-inf}\ (\bigcup n. S\ n))$

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* $P\ P\text{-inf}$
<proof>

lemma *upwards-closedE*:
assumes *upwards-closed* $P\ P\text{-inf}$
and *ascending* S
and $\bigwedge n. P\ n\ (S\ n)$
shows $P\text{-inf}\ (\bigcup n. S\ n)$
<proof>

lemma *ascending-iterate-filter*:
ascending $(\lambda n. \textit{filter-exp}\ (\textit{lnot}\ b)\ (\textit{union-up-to-n}\ (\textit{if-then}\ b\ C)\ S\ n))$
<proof>

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* $Q\ Q\text{-inf}$
shows $\models \{P\ 0\}\ \textit{while-cond}\ b\ C\ \{\textit{conj}\ Q\text{-inf}\ (\textit{holds-forall}\ (\textit{lnot}\ b))\}$
<proof>

definition *while-loop-assertion-n where*
while-loop-assertion-n $C\ S0\ n\ S \longleftrightarrow (S = \textit{union-up-to-n}\ C\ S0\ n)$

definition *while-loop-assertion-inf where*
while-loop-assertion-inf $C\ S0\ S \longleftrightarrow (S = (\bigcup n. \textit{union-up-to-n}\ C\ S0\ n))$

lemma *while-loop-assertion-upwards-closed*:
upwards-closed $(\textit{while-loop-assertion-n}\ C\ S0)\ (\textit{while-loop-assertion-inf}\ C\ S0)$
<proof>

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

<proof>

lemma *ascending-converges*:

assumes *ascending* S

shows *converges-sets* S

<proof>

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>

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*:

$x \in \text{limit-sets } S \longleftrightarrow (\exists n. \forall m. m \geq n \longrightarrow (x \in S\ m))$

<proof>

lemma *ascending-limits-union*:

assumes *ascending* S

shows *limit-sets* $S = (\bigcup n. S\ n)$ (**is** $?A = ?B$)

<proof>

lemma *descending-limits-union*:

assumes *descending* S

shows *limit-sets* $S = (\bigcap n. S\ n)$ (**is** $?A = ?B$)

<proof>

definition *t-closed* **where**

t-closed $P\ P\text{-inf} \longleftrightarrow (\forall S. \text{converges-sets } S \wedge (\forall n. P\ n\ (S\ n)) \longrightarrow P\text{-inf } (\text{limit-sets } S))$

lemma *t-closed-implies-u-closed*:

assumes $t\text{-closed } P \text{ } P\text{-inf}$
shows $\text{upwards-closed } P \text{ } P\text{-inf}$
 $\langle \text{proof} \rangle$

definition $\text{downwards-closed where}$
 $\text{downwards-closed } P\text{-inf} \longleftrightarrow (\forall S S'. S \subseteq S' \wedge P\text{-inf } S' \longrightarrow P\text{-inf } S)$

definition $d\text{-closed where}$
 $d\text{-closed } P \text{ } P\text{-inf} \longleftrightarrow t\text{-closed } P \text{ } P\text{-inf} \wedge \text{downwards-closed } P\text{-inf}$

lemma $\text{converges-to-merged:}$
assumes $\bigwedge x. x \in S\text{-inf} \implies \exists n. \forall m. m \geq n \longrightarrow (x \in S (m::\text{nat}))$
and $\bigwedge x. x \notin S\text{-inf} \implies \exists n. \forall m. m \geq n \longrightarrow (x \notin S m)$
shows $\text{converges-sets } S \wedge \text{limit-sets } S = S\text{-inf}$
 $\langle \text{proof} \rangle$

lemma $\text{ascending-union-up:}$
 $\text{ascending } (\lambda n. \text{union-up-to-}n \ C \ S \ n)$
 $\langle \text{proof} \rangle$

lemma converges-union:
 $\text{converges-sets } (\lambda n. \text{union-up-to-}n \ C \ S \ n) \wedge \text{limit-sets } (\lambda n. \text{union-up-to-}n \ C \ S \ n)$
 $= (\bigcup n. \text{union-up-to-}n \ C \ S \ n)$
 $\langle \text{proof} \rangle$

theorem while-d:
assumes $\bigwedge n. \models \{P \ n\} \text{ if-then } b \ C \ \{P \ (\text{Suc } n)\}$
and $\text{upwards-closed } P \text{ } P\text{-inf}$
and $\bigwedge n. \text{downwards-closed } (P \ n)$ — Satisfied by hyper-assertions that do not existentially quantify over states
shows $\models \{P \ 0\} \text{ while-cond } b \ C \ \{\text{conj } P\text{-inf } (\text{holds-forall } (\text{lnot } b))\}$
 $\langle \text{proof} \rangle$

lemma in-union-up-to:
 $x \in \text{union-up-to-}n \ C \ S \ n \longleftrightarrow (\exists m. m \leq n \wedge x \in \text{iterate-sem } m \ C \ S)$
 $\langle \text{proof} \rangle$

theorem $\text{rule-while-terminates-strong:}$
assumes $\bigwedge n. n < m \implies \models \{P \ n\} \text{ if-then } b \ C \ \{P \ (\text{Suc } n)\}$
and $\bigwedge S. P \ m \ S \longrightarrow \text{holds-forall } (\text{lnot } b) \ S$
shows $\models \{P \ 0\} \text{ while-cond } b \ C \ \{P \ m\}$
 $\langle \text{proof} \rangle$

lemma *false-state-in-if-then*:

assumes $\varphi \in S$
 and $\neg b \text{ (snd } \varphi)$
 shows $\varphi \in \text{sem (if-then } b \ C) \ S$
<proof>

lemma *false-state-in-while-cond-aux*:

assumes $\varphi \in S$
 and $\neg b \text{ (snd } \varphi)$
 shows $\varphi \in \text{iterate-sem } n \text{ (if-then } b \ C) \ S$
<proof>

lemma *false-state-in-while-cond*:

assumes $\varphi \in S$
 and $\neg b \text{ (snd } \varphi)$
 shows $\varphi \in \text{sem (while-cond } b \ C) \ S$
<proof>

theorem *while-exists*:

assumes $\bigwedge \varphi. \models \{ P \ \varphi \} \text{ while-cond } b \ C \ \{ Q \ \varphi \}$
 shows $\models \{ (\lambda S. \exists \varphi \in S. \neg b \text{ (snd } \varphi) \wedge P \ \varphi \ S) \} \text{ while-cond } b \ C \ \{ (\lambda S. \exists \varphi \in S. Q \ \varphi \ S) \}$
<proof>

lemma *sem-while-cond-union-up-to*:

$\text{sem (while-cond } b \ C) \ S = \text{filter-exp (lnot } b) (\bigcup n. \text{union-up-to-} n \text{ (if-then } b \ C) \ S)$
<proof>

lemma *iterate-sem-sum*:

$\text{iterate-sem } n \ C \ (\text{iterate-sem } m \ C \ S) = \text{iterate-sem } (n + m) \ C \ S$
<proof>

lemma *unroll-while-sem*:

$\text{sem (while-cond } b \ C) \ (\text{iterate-sem } n \text{ (if-then } b \ C) \ S) = \text{sem (while-cond } b \ C) \ S$
<proof>

theorem *while-unroll*:

assumes $\bigwedge n. n < m \implies \models \{ P \ n \} \text{ if-then } b \ C \ \{ P \ (\text{Suc } n) \}$
 and $\models \{ P \ m \} \text{ while-cond } b \ C \ \{ Q \}$
 shows $\models \{ P \ 0 \} \text{ while-cond } b \ C \ \{ Q \}$
<proof>

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 (lnot b) { Q }*
shows $\models \{I\ 0\}$ *while-cond b C { Q }*
 $\langle \text{proof} \rangle$

corollary *loop-exit*:
assumes *entails P (holds-forall (lnot b))*
shows $\models \{P\}$ *while-cond b C {P}*
 $\langle \text{proof} \rangle$

corollary *for-loop*:
assumes $\bigwedge n. n < m \implies \models \{P\ n\}$ *if-then b C {P (Suc n)}*
and *entails (P m) (holds-forall (lnot b))*
shows $\models \{P\ 0\}$ *while-cond b C {P m}*
 $\langle \text{proof} \rangle$

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*
 $\text{in-set } \varphi\ S \longleftrightarrow \varphi \in S$

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 $\models \{ ((\lambda S. \forall \varphi 1 \in S. P\ \varphi 1\ S) :: (('a, 'b, 'c, 'd)\ \text{state})\ \text{hyperassertion}) \} C$
 $\{ (\lambda S. \forall \varphi 2 \in S. Q\ \varphi 2\ S) \}$
 $\langle \text{proof} \rangle$

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 $\models \{ (\lambda S. \forall \omega \in S. P\ \omega\ S) \} C \{ (\lambda S. \forall \omega' \in S. Q\ \omega'\ S) \}$
 $\langle \text{proof} \rangle$

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>

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>

definition *differ-only-by-pset where*

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

lemma *differ-only-by-psetI*:

assumes $\bigwedge i. fst (a i) = fst (b i) \wedge \text{differ-only-by-set vars } (snd (a i)) (snd (b i))$

shows *differ-only-by-pset vars a b*

<proof>

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 $b :: ('a \Rightarrow ('lvar, 'lval, 'pvar, 'pval) state) \Rightarrow bool$

— 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>

5.3 Logical Updates

definition *equal-outside-set where*

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*
<proof>

lemma *equal-outside-setE*:
assumes *equal-outside-set vars l1 l2*
and $x \notin \text{vars}$
shows $l1\ x = l2\ x$
<proof>

lemma *equal-outside-sym*:
equal-outside-set vars l l' \longleftrightarrow equal-outside-set vars l' l
<proof>

definition *subset-mod-updates where*
subset-mod-updates vars S S' \longleftrightarrow ($\forall \omega \in S. \exists \omega' \in S'. \text{snd } \omega = \text{snd } \omega' \wedge$
equal-outside-set vars (fst ω) (fst ω')

lemma *subset-mod-updatesI*:
assumes $\bigwedge \omega. \omega \in S \implies (\exists \omega' \in S'. \text{snd } \omega = \text{snd } \omega' \wedge \text{equal-outside-set vars (fst } \omega) \text{ (fst } \omega'))$
shows *subset-mod-updates vars S S'*
<proof>

lemma *subset-mod-updatesE*:
assumes *subset-mod-updates vars S S'*
and $\omega \in S$
shows $\exists \omega' \in S'. \text{snd } \omega = \text{snd } \omega' \wedge \text{equal-outside-set vars (fst } \omega) \text{ (fst } \omega')$
<proof>

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'*
<proof>

lemma *same-mod-updates-sym*:
same-mod-updates vars S S' \longleftrightarrow same-mod-updates vars S' S
<proof>

lemma *same-mod-updates-refl*:
same-mod-updates vars S S
<proof>

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*
 \langle *proof* \rangle

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*
 \langle *proof* \rangle

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*
 \langle *proof* \rangle

lemma *sem-update-commute-aux*:
assumes *subset-mod-updates vars S1 S2*
shows *subset-mod-updates vars (sem C S1) (sem C S2)*
 \langle *proof* \rangle

lemma *sem-update-commute*:
assumes *same-mod-updates (vars :: 'a set) S1 S2*
shows *same-mod-updates vars (sem C S1) (sem C S2)*
 \langle *proof* \rangle

type-synonym (*'a, 'b, 'c, 'd*) *chyperassertion* = $((\text{'a} \Rightarrow \text{'b}) \times (\text{'c} \Rightarrow \text{'d})) \text{ set} \Rightarrow \text{bool}$

definition *invariant-on-updates* :: $\text{'a set} \Rightarrow (\text{'a, 'b, 'c, 'd}) \text{chyperassertion} \Rightarrow \text{bool}$
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*
 \langle *proof* \rangle

definition *entails-with-updates* :: $\text{'a set} \Rightarrow (\text{'a, 'b, 'c, 'd}) \text{chyperassertion} \Rightarrow (\text{'a, 'b, 'c, 'd}) \text{chyperassertion} \Rightarrow \text{bool}$
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 $\bigwedge S. P S \implies (\exists S'. \text{same-mod-updates vars } S S' \wedge Q S')$
shows *entails-with-updates vars P Q*
 $\langle \text{proof} \rangle$

lemma *entails-with-updatesE*:

assumes *entails-with-updates vars P Q*
and $P S$
shows $\exists S'. \text{same-mod-updates vars } S S' \wedge Q S'$
 $\langle \text{proof} \rangle$

proposition *rule-LUpdate*:

assumes $\models \{P'\} C \{Q\}$
and *entails-with-updates vars P P'*
and *invariant-on-updates vars Q*
shows $\models \{P\} C \{Q\}$
 $\langle \text{proof} \rangle$

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 $\text{Set.filter } f (\text{sem } C S) = \text{sem } C (\text{Set.filter } f S)$ (**is** $?A = ?B$)
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

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)*
 $\langle \text{proof} \rangle$

lemma *rule-apply*:

assumes $\models \{P\} C \{Q\}$
and *commute-with-sem f*
shows $\models \{P \circ f\} C \{Q \circ f\}$
 $\langle \text{proof} \rangle$

definition *apply-filter where*

$apply_filter\ b\ P\ S \longleftrightarrow P\ (Set.filter\ b\ S)$

proposition *rule-LFilter:*

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

shows $\models \{ P \circ (Set.filter\ (b \circ fst)) \}\ C\ \{ Q \circ (Set.filter\ (b \circ fst)) \}$

<proof>

definition *differ-only-by-pset-single where*

$differ_only_by_pset_single\ vars\ a\ b \longleftrightarrow (fst\ a = fst\ b \wedge differ_only_by_set\ vars\ (snd\ a)\ (snd\ b))$

definition *not-in-free-pvars-pep where*

$not_in_free_pvars_pep\ vars\ b \longleftrightarrow (\forall\ \varphi1\ \varphi2. differ_only_by_set\ vars\ \varphi1\ \varphi2 \longrightarrow (b\ \varphi1 \longleftrightarrow b\ \varphi2))$

lemma *single-sem-differ-by-written-vars:*

assumes *single-sem* $C\ \varphi\ \varphi'$

shows *differ-only-by-set* $(written_vars\ C)\ \varphi\ \varphi'$

<proof>

lemma *single-sem-not-free-vars:*

assumes *not-in-free-pvars-pep* $(written_vars\ C)\ b$

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

shows $b\ \varphi \longleftrightarrow b\ \varphi'$

<proof>

proposition *rule-PFilter:*

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

and *not-in-free-pvars-pep* $(written_vars\ C)\ b$

shows $\models \{ P \circ (Set.filter\ (b \circ snd)) \}\ C\ \{ Q \circ (Set.filter\ (b \circ snd)) \}$

<proof>

5.5 Other Compositionality Rules

proposition *rule-False:*

hyper-hoare-triple $(\lambda-. False)\ C\ Q$

<proof>

proposition *rule-True:*

hyper-hoare-triple $P\ C\ (\lambda-. True)$

<proof>

lemma *sem-inter:*

$sem\ C\ (S1 \cap S2) \subseteq sem\ C\ S1 \cap sem\ C\ S2$

$\langle proof \rangle$

proposition *rule-Union:*

assumes $\models \{P\} C \{Q\}$
and *hyper-hoare-triple* $P' C Q'$
shows *hyper-hoare-triple* $(join P P') C (join Q Q')$
 $\langle proof \rangle$

proposition *rule-IndexedUnion:*

assumes $\bigwedge x. \models \{P x\} C \{Q x\}$
shows *hyper-hoare-triple* $(general-join P) C (general-join Q)$
 $\langle proof \rangle$

proposition *rule-And:*

assumes $\models \{P\} C \{Q\}$
and *hyper-hoare-triple* $P' C Q'$
shows *hyper-hoare-triple* $(conj P P') C (conj Q Q')$
 $\langle proof \rangle$

lemma *rule-Forall:*

assumes $\bigwedge x. \models \{P x\} C \{Q x\}$
shows *hyper-hoare-triple* $(forall P) C (forall Q)$
 $\langle proof \rangle$

lemma *rule-Or:*

assumes $\models \{P\} C \{Q\}$
and $\models \{P'\} C \{Q'\}$
shows *hyper-hoare-triple* $(disj P P') C (disj Q Q')$
 $\langle proof \rangle$

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* $P (If C1 C2) Q$
 $\langle proof \rangle$

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 $I (\bigcup S \in F. S)$
 $\langle proof \rangle$

lemma *stable-by-union-and-constant-then-I:*

assumes $\bigwedge n. I n = I'$
and *stable-by-infinite-union* I'
shows *natural-partition* $I = I'$
 \langle *proof* \rangle

corollary *simpler-rule-while*:
assumes *hyper-hoare-triple* $I C I$
and *stable-by-infinite-union* I
shows *hyper-hoare-triple* $I (\text{While } C) I$
 \langle *proof* \rangle

lemma *rule-and3*:
assumes $\models \{P1\} C \{Q1\}$
and $\models \{P2\} C \{Q2\}$
and $\models \{P3\} C \{Q3\}$
shows $\models \{ \text{conj } P1 (\text{conj } P2 P3) \} C \{ \text{conj } Q1 (\text{conj } Q2 Q3) \}$
 \langle *proof* \rangle

definition *not-empty where*
 $\text{not-empty } S \longleftrightarrow S \neq \{\}$

definition *finite-not-empty where*
 $\text{finite-not-empty } S \longleftrightarrow S \neq \{\} \wedge \text{finite } S$

definition *update-logical where*
 $\text{update-logical } \omega \ i \ v = ((\text{fst } \omega)(i := v), \text{snd } \omega)$

lemma *single-sem-prop*:
assumes *single-sem* $C (\text{snd } \omega) (\text{snd } \omega')$
and $\text{fst } \omega = \text{fst } \omega'$
shows $\models \{ (\lambda S. \omega \in S) \} C \{ (\lambda S. \omega' \in S) \}$
 \langle *proof* \rangle

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 $\models \{ (\lambda S. \forall \omega \in S. P \ \omega \ S) \} C \{ (\lambda S. \forall \omega' \in S. Q \ \omega' \ S) \}$
 \langle *proof* \rangle

definition *general-union :: 'a hyperassertion \implies 'a hyperassertion where*
 $\text{general-union } P \ S \longleftrightarrow (\exists F. S = \text{Union } F \wedge (\forall S' \in F. P \ S'))$

lemma *general-unionI*:
assumes $S = \text{Union } F$
and $\bigwedge S'. S' \in F \implies P \ S'$
shows *general-union* $P \ S$

<proof>

lemma *general-unionE*:

assumes *general-union P S*

obtains *F* **where** $S = \text{Union } F \wedge S'. S' \in F \implies P S'$

<proof>

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>

proposition *rule-Empty*:

$\models \{(\lambda S. S = \{\})\} C \{(\lambda S. S = \{\})\}$

<proof>

definition *has-subset where*

$\text{has-subset } P S \longleftrightarrow (\exists S'. S' \subseteq S \wedge P S')$

lemma *has-subset-join-same*:

entails $(\text{has-subset } P) (\text{join } P (\lambda-. \text{True}))$

entails $(\text{join } P (\lambda-. \text{True})) (\text{has-subset } P)$

<proof>

proposition *rule-AtLeast*:

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

shows $\models \{\text{has-subset } P\} C \{\text{has-subset } Q\}$

<proof>

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>

5.6 Synchronous Reasoning (Proposition 14, Appendix H).

theorem *if-sync-rule*:

assumes $\models \{P\} C1 \{P1\}$

and $\models \{P\} C2 \{P2\}$

and $\models \{ \text{combine from-nat } x \ P1 \ P2 \} \ C \ \{ \text{combine from-nat } x \ R1 \ R2 \}$
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* $1 \neq \text{from-nat } 2$

and *not-free-var-hyper* $x \ R1$
and *not-free-var-hyper* $x \ R2$

shows $\models \{ P \} \ \text{If } (Seq \ C1 \ (Seq \ C \ C1')) \ (Seq \ C2 \ (Seq \ C \ C2')) \ \{ \text{join } Q1 \ Q2 \}$
 $\langle \text{proof} \rangle$

definition *update-lvar-set* **where**

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 $\{u\} \ (fst \ \varphi) \ (fst \ ((fst \ \varphi)(u := x), \ snd \ \varphi))$
 $\langle \text{proof} \rangle$

lemma *same-update-lvar-set*:

same-mod-updates $\{u\} \ S \ (\text{update-lvar-set } u \ e \ S)$
 $\langle \text{proof} \rangle$

lemma *same-mod-updates-empty*:

assumes *same-mod-updates vars* $\{ \} \ S'$
shows $S' = \{ \}$
 $\langle \text{proof} \rangle$

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 $I \ S'$
 $\langle \text{proof} \rangle$

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*:

$\text{snd } \varphi = \text{snd } (\text{assign-exp-to-lvar } e \ l \ \varphi) \wedge \text{equal-outside-set } \{l\} \ (\text{fst } \varphi) \ (\text{fst } (\text{assign-exp-to-lvar } e \ l \ \varphi))$
 <proof>

lemma *assign-exp-to-lvar-set-same-mod-updates:*
same-mod-updates $\{l\} \ S \ (\text{assign-exp-to-lvar-set } e \ l \ S)$
 <proof>

lemma *holds-forall-same-mod-updates:*
assumes *same-mod-updates vars* $S \ S'$
and *holds-forall* $b \ S$
shows *holds-forall* $b \ S'$
 <proof>

lemma *not-fv-hyper-assign-exp:*
assumes *not-fv-hyper* $t \ A$
shows $A \ S \longleftrightarrow A \ (\text{assign-exp-to-lvar-set } e \ t \ S)$
 <proof>

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>

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* $e \ t \ S$
 <proof>

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* $e \ S$
 <proof>

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>

lemma *e-recorded-in-t-if-assigned:*

e-recorded-in-*t* *e t* (*assign-exp-to-lvar-set e t S*)
 ⟨*proof*⟩

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*⟩

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 $\rightarrow A \wedge A$

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 (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) - \longleftrightarrow b
 | *sat-assertion* vals states (AComp e1 cmp e2) - \longleftrightarrow cmp (*interp-exp* vals states e1) (*interp-exp* vals states e2)
 | *sat-assertion* vals states (AForallState A) S \longleftrightarrow ($\forall \varphi \in S. \text{sat-assertion vals } (\varphi \# \text{states}) A S$)
 | *sat-assertion* vals states (AExistsState A) S \longleftrightarrow ($\exists \varphi \in S. \text{sat-assertion vals } (\varphi \# \text{states}) A S$)
 | *sat-assertion* vals states (AForall A) S \longleftrightarrow ($\forall v. \text{sat-assertion } (v \# \text{vals}) \text{states } A S$)
 | *sat-assertion* vals states (AExists A) S \longleftrightarrow ($\exists v. \text{sat-assertion } (v \# \text{vals}) \text{states } A S$)
 | *sat-assertion* vals states (AAnd A B) S \longleftrightarrow (*sat-assertion* vals states A S \wedge *sat-assertion* vals states B S)
 | *sat-assertion* vals states (AOr A B) S \longleftrightarrow (*sat-assertion* vals states A S \vee *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 $\longleftrightarrow \neg$ (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*)
 ⟨*proof*⟩

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)
 ⟨*proof*⟩

abbreviation *interp-assert where* *interp-assert* \equiv *sat-assertion* [] []

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-semI*:

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*
 ⟨*proof*⟩

lemma *transform-assume-valid*:

assumes *same-syn-sem bsyn bsem*
shows *sat-assertion vals states A* (*Set.filter* (*bsem* \circ *snd*) *S*)
 \longleftrightarrow *sat-assertion vals states* (*transform-assume bsyn A*) *S*
 ⟨*proof*⟩

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 ⇒ 'a 'a pexp

```

fun *interp-pexp* :: 'a pexp ⇒ 'a npstate ⇒ 'a

where

```

  interp-pexp (PVar x) φ = φ x
| interp-pexp (PConst n) - = n
| interp-pexp (PBinop p1 op p2) φ = op (interp-pexp p1 φ) (interp-pexp p2 φ)
| interp-pexp (PFun f p) φ = f (interp-pexp p φ)

```

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⟩

```

6.2.2 Program expressions (booleans)

datatype 'a pbexp =

```

  PConst bool
| PAnd 'a pbexp 'a pbexp
| POr 'a pbexp 'a pbexp
| PComp 'a pexp 'a comp 'a pexp

```

fun *interp-pbexp* :: 'a pbexp ⇒ 'a npstate ⇒ bool

where

```

  interp-pbexp (PConst b) - ↔ b
| interp-pbexp (PAnd pb1 pb2) φ ↔ interp-pbexp pb1 φ ∧ interp-pbexp pb2 φ
| interp-pbexp (POr pb1 pb2) φ ↔ interp-pbexp pb1 φ ∨ interp-pbexp pb2 φ
| interp-pbexp (PComp p1 cmp p2) φ ↔ cmp (interp-pexp p1 φ) (interp-pexp
p2 φ)

```

fun *pbexp-to-assertion* **where**

```

  pbexp-to-assertion - (PConst b) = AConst b
| pbexp-to-assertion st (PAnd pb1 pb2) = AAnd (pbexp-to-assertion st pb1) (pbexp-to-assertion
st pb2)
| pbexp-to-assertion st (POr pb1 pb2) = AOr (pbexp-to-assertion st pb1) (pbexp-to-assertion
st pb2)
| pbexp-to-assertion st (PComp 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
 \langle *proof* \rangle

lemma *pexp-to-exp-same*:
shows *same-syn-sem* (*pexp-to-assertion* 0 *pb*) (*interp-pbexp* *pb*)
 \langle *proof* \rangle

6.2.3 Syntactic rule for assume

theorem *rule-assume-syntactic-general*:
 $\models \{ \text{sat-assertion states vals } (\text{transform-assume } (\text{pexp-to-assertion } 0 \text{ pb}) P) \}$
Assume (*interp-pbexp* *pb*) $\{ \text{sat-assertion states vals } P \}$
 \langle *proof* \rangle

theorem *rule-assume-syntactic*:
 $\models \{ \text{interp-assert } (\text{transform-assume } (\text{pexp-to-assertion } 0 \text{ pb}) P) \}$ *Assume*
(*interp-pbexp* *pb*) $\{ \text{interp-assert } P \}$
 \langle *proof* \rangle

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
 \langle *proof* \rangle

lemma *insert-at-charact-1*:
 $n \leq \text{length } l \implies k < n \implies (\text{insert-at } n \text{ x } l) ! k = l ! k$
 \langle *proof* \rangle

lemma *insert-at-charact-2*:
 $n \leq \text{length } l \implies (\text{insert-at } n \text{ x } l) ! n = x$
 \langle *proof* \rangle

lemma *insert-at-charact-3*:
 $n \leq \text{length } l \implies k \geq n \implies (\text{insert-at } n \text{ x } l) ! (\text{Suc } k) = l ! k$
 \langle *proof* \rangle

fun *shift-vars-exp where*
shift-vars-exp n (*EQVar x*) = (if $x \geq n$ 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 $\varphi \geq n$ then *EPVar* (*Suc* φ) *x* else *EPVar* φ *x*)
| *shift-states-exp* *n* (*ELVar* φ *x*) = (if $\varphi \geq n$ 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* \Rightarrow *nat* \Rightarrow 'a *exp* \Rightarrow *bool* **where**

wf-exp *nv ns* (*EPVar* *st* -) \longleftrightarrow *st* < *ns*
| *wf-exp* *nv ns* (*ELVar* *st* -) \longleftrightarrow *st* < *ns*
| *wf-exp* *nv ns* (*EQVar* *x*) \longleftrightarrow *x* < *nv*
| *wf-exp* *nv ns* (*EBinop* *e1 - e2*) \longleftrightarrow *wf-exp* *nv ns e1* \wedge *wf-exp* *nv ns e2*
| *wf-exp* *nv ns* (*EFun* *f e*) \longleftrightarrow *wf-exp* *nv ns e*
| *wf-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*)
<proof>

lemma *wf-shift-states-exp*:

assumes *wf-exp* *nv ns e*
shows *wf-exp* *nv* (*Suc* *ns*) (*shift-states-exp* *n e*)
<proof>

lemma *shift-vars-exp-charact*:

assumes $n \leq \text{length } \text{vals}$
shows *interp-exp* *vals* *states e* = *interp-exp* (*insert-at* *n v vals*) *states* (*shift-vars-exp* *n e*)
<proof>

lemma *shift-states-exp-charact*:

assumes $n \leq \text{length } \text{states}$
shows *interp-exp* *vals* *states e* = *interp-exp* *vals* (*insert-at* *n* φ *states*) (*shift-states-exp* *n e*)
<proof>

fun *shift-vars* **where**

shift-vars *n* (*AConst* *b*) = *AConst* *b*
| *shift-vars* *n* (*AComp* *e1 cmp e2*) = *AComp* (*shift-vars-exp* *n e1*) *cmp* (*shift-vars-exp*

$n\ e2$)
 $|$ $\text{shift-vars } n\ (A\text{Forall } A) = A\text{Forall } (\text{shift-vars } (\text{Suc } n)\ A)$
 $|$ $\text{shift-vars } n\ (A\text{Exists } A) = A\text{Exists } (\text{shift-vars } (\text{Suc } n)\ A)$
 $|$ $\text{shift-vars } n\ (A\text{ForallState } A) = A\text{ForallState } (\text{shift-vars } n\ A)$
 $|$ $\text{shift-vars } n\ (A\text{ExistsState } A) = A\text{ExistsState } (\text{shift-vars } n\ A)$
 $|$ $\text{shift-vars } n\ (A\text{Or } A\ B) = A\text{Or } (\text{shift-vars } n\ A)\ (\text{shift-vars } n\ B)$
 $|$ $\text{shift-vars } n\ (A\text{And } A\ B) = A\text{And } (\text{shift-vars } n\ A)\ (\text{shift-vars } n\ B)$

lemma *shift-vars-charact*:

assumes $n \leq \text{length } \text{vals}$

shows $\text{sat-assertion } \text{vals } \text{states } A\ S \longleftrightarrow \text{sat-assertion } (\text{insert-at } n\ x\ \text{vals})\ \text{states}$
 $(\text{shift-vars } n\ A)\ S$

$\langle \text{proof} \rangle$

6.3.2 Expressions (Boolean and values)

definition *update-state where*

$\text{update-state } \varphi\ x\ v = (\text{fst } \varphi, (\text{snd } \varphi)(x := v))$

fun *subst-exp-single* :: $q\text{state} \Rightarrow \text{var} \Rightarrow 'a\ \text{exp} \Rightarrow 'a\ \text{exp} \Rightarrow 'a\ \text{exp}$ **where**

$\text{subst-exp-single } \varphi\ x\ e'\ (E\text{PVar } st\ y) = (\text{if } \varphi = st \wedge x = y \text{ then } e' \text{ else } E\text{PVar } st\ y)$

$|$ $\text{subst-exp-single } \varphi\ x\ e'\ (E\text{Binop } e1\ \text{bop } e2) = E\text{Binop } (\text{subst-exp-single } \varphi\ x\ e'\ e1)\ \text{bop } (\text{subst-exp-single } \varphi\ x\ e'\ e2)$

$|$ $\text{subst-exp-single } \varphi\ x\ e'\ (E\text{Fun } f\ e) = E\text{Fun } f\ (\text{subst-exp-single } \varphi\ x\ e'\ e)$

$|$ $\text{subst-exp-single } -\ -\ -\ e = e$

lemma *wf-subst-exp*:

assumes $wf\text{-exp } nv\ ns\ e$

and $wf\text{-exp } nv\ ns\ e'$

shows $wf\text{-exp } nv\ ns\ (\text{subst-exp-single } \varphi\ x\ e'\ e)$

$\langle \text{proof} \rangle$

lemma *subst-exp-single-charact*:

assumes $\text{interp-exp } \text{vals } \text{states } e' = \text{snd } (\text{states } !\ st)\ x$

shows $\text{interp-exp } \text{vals } \text{states } (\text{subst-exp-single } st\ x\ e'\ e) = \text{interp-exp } \text{vals } \text{states } e$

$\langle \text{proof} \rangle$

definition *subst-state where*

$\text{subst-state } x\ pe\ \varphi = (\text{fst } \varphi, (\text{snd } \varphi)(x := \text{interp-pe } pe\ (\text{snd } \varphi)))$

definition *update-state-at where*

$\text{update-state-at } \text{states } n\ x\ v = \text{list-update } \text{states } n\ (\text{update-state } (\text{states } !\ n)\ x\ v)$

lemma *update-state-at-fst*:

$\text{fst } (\text{update-state-at states } n \ x \ v \ ! \ st) = \text{fst } (\text{states } ! \ st)$
 $\langle \text{proof} \rangle$

lemma *update-state-at-snd-1*:

$x \neq y \implies \text{snd } (\text{update-state-at states } n \ x \ v \ ! \ st) \ y = \text{snd } (\text{states } ! \ st) \ y$
 $\langle \text{proof} \rangle$

lemma *update-state-at-snd-2*:

$st \neq n \implies \text{snd } (\text{update-state-at states } n \ x \ v \ ! \ st) \ y = \text{snd } (\text{states } ! \ st) \ y$
 $\langle \text{proof} \rangle$

lemma *update-state-at-snd-3*:

assumes $n < \text{length states}$
shows $\text{snd } (\text{update-state-at states } n \ x \ v \ ! \ n) \ x = v$
 $\langle \text{proof} \rangle$

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 $\text{interp-exp vals states } (\text{subst-exp-single } st \ x \ e' \ e) = \text{interp-exp vals states}' \ e$
 $\langle \text{proof} \rangle$

6.3.3 Assertions

fun *subst-assertion-single* :: $qstate \Rightarrow var \Rightarrow 'a \ \text{exp} \Rightarrow 'a \ \text{assertion} \Rightarrow 'a \ \text{assertion}$
where

$\text{subst-assertion-single } st \ x \ e \ (AConst \ b) = AConst \ b$
 $| \text{subst-assertion-single } st \ x \ e \ (AComp \ e1 \ \text{cmp} \ e2) = AComp \ (\text{subst-exp-single } st \ x \ e \ e1) \ \text{cmp} \ (\text{subst-exp-single } st \ x \ e \ e2)$
 $| \text{subst-assertion-single } st \ x \ e \ (AForall \ A) = AForall \ (\text{subst-assertion-single } st \ x \ (\text{shift-vars-exp } 0 \ e) \ A)$
 $| \text{subst-assertion-single } st \ x \ e \ (AExists \ A) = AExists \ (\text{subst-assertion-single } st \ x \ (\text{shift-vars-exp } 0 \ e) \ A)$
 $| \text{subst-assertion-single } st \ x \ e \ (AOr \ A \ B) = AOr \ (\text{subst-assertion-single } st \ x \ e \ A) \ (\text{subst-assertion-single } st \ x \ e \ B)$
 $| \text{subst-assertion-single } st \ x \ e \ (AAnd \ A \ B) = AAnd \ (\text{subst-assertion-single } st \ x \ e \ A) \ (\text{subst-assertion-single } st \ x \ e \ B)$
 $| \text{subst-assertion-single } st \ x \ e \ (AForallState \ A) = AForallState \ (\text{subst-assertion-single } (\text{Suc } st) \ x \ (\text{shift-states-exp } 0 \ e) \ A)$
 $| \text{subst-assertion-single } st \ x \ e \ (AExistsState \ A) = AExistsState \ (\text{subst-assertion-single } (\text{Suc } st) \ x \ (\text{shift-states-exp } 0 \ e) \ A)$

fun *wf-assertion-aux* :: $nat \Rightarrow nat \Rightarrow 'a \ \text{assertion} \Rightarrow bool$ **where**

$\text{wf-assertion-aux } nv \ ns \ (AConst \ b) \longleftrightarrow True$
 $| \text{wf-assertion-aux } nv \ ns \ (AComp \ e1 \ \text{cmp} \ e2) \longleftrightarrow \text{wf-exp } nv \ ns \ e1 \ \wedge \ \text{wf-exp } nv \ ns \ e2$

$| \text{wf-assertion-aux } nv \ ns \ (A \text{And } A \ B) \longleftrightarrow \text{wf-assertion-aux } nv \ ns \ A \wedge \text{wf-assertion-aux } nv \ ns \ B$
 $| \text{wf-assertion-aux } nv \ ns \ (A \text{Or } A \ B) \longleftrightarrow \text{wf-assertion-aux } nv \ ns \ A \vee \text{wf-assertion-aux } nv \ ns \ B$
 $| \text{wf-assertion-aux } nv \ ns \ (A \text{Forall } A) \longleftrightarrow \text{wf-assertion-aux } (Suc \ nv) \ ns \ A$
 $| \text{wf-assertion-aux } nv \ ns \ (A \text{Exists } A) \longleftrightarrow \text{wf-assertion-aux } (Suc \ nv) \ ns \ A$
 $| \text{wf-assertion-aux } nv \ ns \ (A \text{ForallState } A) \longleftrightarrow \text{wf-assertion-aux } nv \ (Suc \ ns) \ A$
 $| \text{wf-assertion-aux } nv \ ns \ (A \text{ExistsState } A) \longleftrightarrow \text{wf-assertion-aux } nv \ (Suc \ ns) \ A$

abbreviation *wf-assertion* **where** $\text{wf-assertion} \equiv \text{wf-assertion-aux } 0 \ 0$

lemma *wf-shift-vars*:

assumes $\text{wf-assertion-aux } nv \ ns \ A$
shows $\text{wf-assertion-aux } (Suc \ nv) \ ns \ (\text{shift-vars } n \ A)$
 $\langle \text{proof} \rangle$

lemma *wf-subst-assertion*:

assumes $\text{wf-assertion-aux } nv \ ns \ A$
and $\text{wf-exp } nv \ ns \ e$
shows $\text{wf-assertion-aux } nv \ ns \ (\text{subst-assertion-single } \varphi \ x \ e \ A)$
 $\langle \text{proof} \rangle$

lemma *subst-assertion-single-charact*:

assumes $\text{interp-exp vals states } e = \text{snd } (\text{states } ! \ st) \ x$
shows $\text{sat-assertion vals states } (\text{subst-assertion-single } st \ x \ e \ A) \ S \longleftrightarrow \text{sat-assertion vals states } A \ S$
 $\langle \text{proof} \rangle$

lemma *update-state-at-cons*:

$\text{update-state-at } (\varphi \ \# \ \text{states}) \ (Suc \ n) \ x \ v = \varphi \ \# \ \text{update-state-at states } n \ x \ v$
 $\langle \text{proof} \rangle$

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 $\text{sat-assertion vals states } (\text{subst-assertion-single } st \ x \ e \ A) \ S \longleftrightarrow \text{sat-assertion vals states}' \ A \ S$
 $\langle \text{proof} \rangle$

6.3.4 Transformation for havoc

fun *transform-havoc* **where**

$\text{transform-havoc } x \ (A \text{ForallState } A) = A \text{ForallState } (A \text{Forall } (\text{subst-assertion-single}$

$0\ x\ (EQVar\ 0)\ (shift\ vars\ 0\ (transform\ havoc\ x\ A))$
 $| transform\ havoc\ x\ (AExistsState\ A) = AExistsState\ (AExists\ (subst\ assertion\ single\ 0\ x\ (EQVar\ 0)\ (shift\ vars\ 0\ (transform\ havoc\ x\ A))))$
 $| transform\ havoc\ x\ (AExists\ A) = AExists\ (transform\ havoc\ x\ A)$
 $| transform\ havoc\ x\ (AForall\ A) = AForall\ (transform\ havoc\ x\ A)$
 $| transform\ havoc\ x\ (AOr\ A\ B) = AOr\ (transform\ havoc\ x\ A)\ (transform\ havoc\ x\ B)$
 $| transform\ havoc\ x\ (AAnd\ A\ B) = AAnd\ (transform\ havoc\ x\ A)\ (transform\ havoc\ x\ B)$
 $| transform\ havoc\ x\ (AConst\ b) = AConst\ b$
 $| transform\ havoc\ x\ (AComp\ e1\ cmp\ e2) = AComp\ e1\ cmp\ e2$

lemma *sem-havoc-bis*:

$sem\ (Havoc\ x)\ S = \{(fst\ \varphi,\ (snd\ \varphi)(x := v)) \mid \varphi\ v.\ \varphi \in S\}$ **(is ?A = ?B)**
 $\langle proof \rangle$

lemma *helper-update-state*:

$(v \# vals) ! 0 = snd\ ((update\ state\ \varphi\ x\ v\ \# states) ! 0)\ x$
 $\langle proof \rangle$

lemma *helper-S-update-states*:

assumes $S' = \{ update\ state\ \varphi\ x\ v \mid \varphi\ v.\ \varphi \in S \}$
shows $(\forall \varphi \in S'. Q\ \varphi) \longleftrightarrow (\forall \varphi \in S.\ \forall v.\ Q\ (update\ state\ \varphi\ x\ v))$
 $\langle proof \rangle$

lemma *helper-S-update-states-exists*:

assumes $S' = \{ update\ state\ \varphi\ x\ v \mid \varphi\ v.\ \varphi \in S \}$
shows $(\exists \varphi \in S'. Q\ \varphi) \longleftrightarrow (\exists \varphi \in S.\ \exists v.\ Q\ (update\ state\ \varphi\ x\ v))$
 $\langle proof \rangle$

lemma *equiv-havoc-transform*:

assumes $S' = \{ update\ state\ \varphi\ x\ v \mid \varphi\ v.\ \varphi \in S \}$
shows $sat\ assertion\ vals\ states\ P\ S' \longleftrightarrow sat\ assertion\ vals\ states\ (transform\ havoc\ x\ P)\ S$
 $\langle proof \rangle$

6.3.5 Syntactic rule for havoc

theorem *rule-havoc-syntactic-general*:

$\models \{ sat\ assertion\ states\ vals\ (transform\ havoc\ x\ P) \}\ Havoc\ x\ \{ sat\ assertion\ states\ vals\ P \}$
 $\langle proof \rangle$

theorem *rule-havoc-syntactic*:

$\models \{ interp\ assert\ (transform\ havoc\ x\ P) \}\ Havoc\ x\ \{ interp\ assert\ P \}$

<proof>

6.4 Assignment rule

6.4.1 Program expressions

fun *subst-pexp* :: *var* \Rightarrow 'a *pexp* \Rightarrow 'a *pexp* \Rightarrow 'a *pexp* **where**
 subst-pexp *x e* (*PVar* *y*) = (if *x* = *y* then *e* else *PVar* *y*)
| *subst-pexp* *x e* (*PBinop* *p1 op p2*) = *PBinop* (*subst-pexp* *x e p1*) *op* (*subst-pexp* *x e p2*)
| *subst-pexp* *x e* (*PFun* *f p*) = *PFun* *f* (*subst-pexp* *x e p*)
| *subst-pexp* - - *e* = *e*

lemma *subst-pexp-charact*:

interp-pexp (*subst-pexp* *x e' e*) σ = *interp-pexp* *e* ($\sigma(x := \text{interp-pexp } e' \sigma)$)

<proof>

fun *subst-pbexp* :: *var* \Rightarrow 'a *pexp* \Rightarrow 'a *pbexp* \Rightarrow 'a *pbexp* **where**
 subst-pbexp *x e* (*PBAnd* *pb1 pb2*) = *PBAnd* (*subst-pbexp* *x e pb1*) (*subst-pbexp* *x e pb2*)
| *subst-pbexp* *x e* (*PBOr* *pb1 pb2*) = *PBOr* (*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*) $\sigma \longleftrightarrow \text{interp-pbexp } pb (\sigma(x := \text{interp-pexp } e \sigma))$

<proof>

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*)

<proof>

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*
 ⟨proof⟩

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⟩

6.4.4 Syntactic rule for assignments

theorem *rule-assign-syntactic-general:*

$\models \{ \text{sat-assertion vals states (transform-assign x pe P) } \} \text{Assign x (interp-pexp pe) } \{ \text{sat-assertion vals states P} \}$
 ⟨proof⟩

theorem *rule-assign-syntactic:*

$\models \{ \text{interp-assert (transform-assign x pe P) } \} \text{Assign x (interp-pexp pe) } \{ \text{interp-assert P} \}$
 ⟨proof⟩

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*) \longleftrightarrow *no-exists-state* *A* \wedge *no-exists-state* *B*
| *no-exists-state* (*AOr* *A B*) \longleftrightarrow *no-exists-state* *A* \wedge *no-exists-state* *B*

lemma *mono-sym-then-up-closed*:

assumes *no-exists-state* *A*
and $S \subseteq S'$
and *sat-assertion vals states* *A S'*
shows *sat-assertion vals states* *A S*
<proof>

definition *up-closed where*

up-closed *A* \longleftrightarrow ($\forall S S'$ *vals states*. $S \subseteq S' \wedge$ *sat-assertion vals states* *A S* \longrightarrow *sat-assertion vals states* *A S'*)

lemma *up-closedE*:

assumes *up-closed* *A*
and $S \subseteq S'$
and *sat-assertion vals states* *A S*
shows *sat-assertion vals states* *A S'*
<proof>

lemma *sat-assertion-aforallstateI*:

assumes $\bigwedge \varphi. \varphi \in S \implies$ *sat-assertion vals* ($\varphi \#$ *states*) *A S*
shows *sat-assertion vals states* (*AForallState* *A*) *S*
<proof>

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>

lemma *general-join-entails*:

assumes *up-closed* *A*
and $\bigwedge x.$ *sat-assertion vals states* (*AForallState* *A*) (*F x*)
shows *sat-assertion vals states* (*AForallState* *A*) ($\bigcup x.$ *F x*)
<proof>

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)
⟨proof⟩

```

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)
⟨proof⟩

```

definition *shift-sequence* :: (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow (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. (shift-sequence S n) m) (is ?A = ?B)
⟨proof⟩

```

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 $P (S (\text{extract-indices-sat-P } P S 0))$

and $n > 0 \implies P (S (\text{extract-indices-sat-P } P S n))$

$\wedge \text{extract-indices-sat-P } P S n > \text{extract-indices-sat-P } P S (n - 1)$

<proof>

lemma *extract-indices-sat-P-larger:*

assumes *holds-infinitely-often* $P S$

shows *extract-indices-sat-P* $P S n \geq n$

<proof>

definition *subseq-sat where*

subseq-sat $P S n = S (\text{extract-indices-sat-P } P S n)$

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>

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 *sat-assertion vals states* $A (\bigcup n. S n)$

<proof>

lemma *upwards-closed-syn-sem-practical:*

assumes *no-forall-state-after-existential* A

shows *upwards-closed* $(\lambda n. \text{interp-assert } A)$ (*interp-assert* A)

<proof>

theorem *while-general-syntactic:*

assumes $\bigwedge n. \models \{P n\}$ *if-then* $b C \{P (Suc n)\}$

and $\bigwedge n. \models \{P n\}$ *Assume* $(\text{lnot } b) \{\text{interp-assert } A\}$

and *no-forall-state-after-existential* A

shows $\models \{P 0\}$ *while-cond* $b C \{\text{conj } (\text{interp-assert } A) (\text{holds-forall } (\text{lnot } b))\}$

<proof>

theorem *while-forall-exists-simpler:*

assumes $\models \{I\}$ *if-then* b C $\{I\}$
and $\models \{I\}$ *Assume* $(\text{lnot } b)$ $\{\text{interp-assert } Q\}$
and *no-forall-state-after-existential* A
shows $\models \{I\}$ *while-cond* b C $\{\text{conj } (\text{interp-assert } Q) (\text{holds-forall } (\text{lnot } b))\}$
<proof>

theorem *while-d-syntactic:*

assumes $\models \{\text{interp-assert } A\}$ *if-then* b C $\{\text{interp-assert } A\}$
and *no-forall-state-after-existential* A
and *no-exists-state* A
shows $\models \{\text{interp-assert } A\}$ *while-cond* b C $\{\text{conj } (\text{interp-assert } A) (\text{holds-forall } (\text{lnot } b))\}$
<proof>

lemma *downwards-closed-is-hypersafety:*

hypersafety $P \longleftrightarrow$ *downwards-closed* P
<proof>

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 $(A\text{ForallState } (A\text{Forall } A))$ $(A\text{Forall } (A\text{ForallState } A))$
<proof>

lemma *exists-commute:*

equiv $(A\text{ExistsState } (A\text{Exists } A))$ $(A\text{Exists } (A\text{ExistsState } A))$
<proof>

lemma *forall-state-and:*

equiv $(A\text{ForallState } (A\text{And } A$ $B))$ $(A\text{And } (A\text{ForallState } A) (A\text{ForallState } B))$
<proof>

lemma *exists-state-or:*

equiv $(A\text{ExistsState } (A\text{Or } A$ $B))$ $(A\text{Or } (A\text{ExistsState } A) (A\text{ExistsState } B))$
<proof>

lemma *forall-and:*

equiv $(A\text{Forall } (A\text{And } A$ $B))$ $(A\text{And } (A\text{Forall } A) (A\text{Forall } B))$
<proof>

lemma *exists-or:*

equiv (*AExists* (*AOr* *A B*)) (*AOr* (*AExists* *A*) (*AExists* *B*))
 ⟨*proof*⟩

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*⟩

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'*

⟨*proof*⟩

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*⟩

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 $\sigma \sigma' \longleftrightarrow (\forall x \in V. \sigma x = \sigma' x)$

lemma *agree-onI*:

assumes $\bigwedge x. x \in V \implies \sigma x = \sigma' x$

shows *agree-on V* $\sigma \sigma'$

⟨*proof*⟩

lemma *agree-onE*:

assumes *agree-on V* $\sigma \sigma'$

and $x \in V$

shows $\sigma x = \sigma' x$

⟨*proof*⟩

lemma *agree-on-subset*:
assumes *agree-on* $V' \sigma \sigma'$
and $V \subseteq V'$
shows *agree-on* $V \sigma \sigma'$
 \langle *proof* \rangle

lemma *agree-on-trans*:
assumes *agree-on* $V \sigma \sigma'$
and *agree-on* $V \sigma' \sigma''$
shows *agree-on* $V \sigma \sigma''$
 \langle *proof* \rangle

lemma *agree-on-sym*:
assumes *agree-on* $V \sigma \sigma'$
shows *agree-on* $V \sigma' \sigma$
 \langle *proof* \rangle

lemma *wr-charact*:
assumes *single-sem* $C \sigma \sigma'$
and $wr \ C \cap V = \{\}$
shows *agree-on* $V \sigma \sigma'$
 \langle *proof* \rangle

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 $n \leq$ *length* *states*
and *wf-exp* nv (*Suc* (*length* *states*)) e
shows *interp-exp* *vals* (*insert-at* n (l, σ) *states*) $e =$ *interp-exp* *vals* (*insert-at* n (l, σ') *states*) e
 \langle *proof* \rangle

fun *fv* **where**
fv (*AAnd* $F1 F2$) = *fv* $F1 \cup$ *fv* $F2$
| *fv* (*AOr* $F1 F2$) = *fv* $F1 \cup$ *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) = $\{\}$

| $fv (AComp\ e1\ cmp\ e2) = fv\text{-exp}\ e1 \cup fv\text{-exp}\ e2$

lemma *fv-wr-charact-aux*:

assumes *agree-on* ($fv\ F$) $\sigma\ \sigma'$
and $n \leq length\ states$
and *sat-assertion vals* (*insert-at* $n\ (l,\ \sigma)\ states$) $F\ S$
and *wf-assertion-aux* $nv\ (Suc\ (length\ states))\ F$
shows *sat-assertion vals* (*insert-at* $n\ (l,\ \sigma')\ states$) $F\ S$
<proof>

lemma *fv-wr-charact*:

assumes *agree-on* ($fv\ F$) $\sigma\ \sigma'$
and *sat-assertion vals* ($(l,\ \sigma) \# states$) $F\ S$
and *wf-assertion-aux* $nv\ (Suc\ (length\ states))\ F$
shows *sat-assertion vals* ($(l,\ \sigma') \# states$) $F\ S$
<proof>

lemma *syntactic-safe-frame-preserved*:

assumes $wr\ C \cap fv\ F = \{\}$
and *sat-assertion vals* $states\ F\ S$
and *wf-assertion-aux* $nv\ (length\ states)\ F$
and *no-exists-state* F
shows *sat-assertion vals* $states\ F\ (sem\ C\ S)$
<proof>

theorem *safe-frame-rule-syntactic*:

assumes $wr\ C \cap fv\ F = \{\}$
and *wf-assertion* F
and *no-exists-state* F
shows $\models \{interp\text{-assert}\ F\}\ C \{interp\text{-assert}\ F\}$
<proof>

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>

end

theory *TotalLogic*

imports *Loops Compositionality SyntacticAssertions*

begin

7 Terminating Hyper-Triples

definition *total-hyper-triple* ($\models_{TERM} \{-\} - \{-\}$ [51,0,0] 81) **where**
 $\models_{TERM} \{P\} C \{Q\} \longleftrightarrow (\models \{P\} C \{Q\} \wedge (\forall S. P S \longrightarrow (\forall \varphi \in S. \exists \sigma'. \text{single-sem } C (\text{snd } \varphi) \sigma')))$

lemma *total-hyper-triple-equiv*:
 $\models_{TERM} \{P\} C \{Q\} \longleftrightarrow (\models \{P\} C \{Q\} \wedge (\forall S. P S \longrightarrow (\forall \varphi \in S. \exists \sigma'. (\text{fst } \varphi, \sigma') \in \text{sem } C S \wedge \text{single-sem } C (\text{snd } \varphi) \sigma')))$
(*proof*)

lemma *total-hyper-tripleI*:
assumes $\models \{P\} C \{Q\}$
and $\bigwedge \varphi S. P S \wedge \varphi \in S \implies (\exists \sigma'. \text{single-sem } C (\text{snd } \varphi) \sigma')$
shows $\models_{TERM} \{P\} C \{Q\}$
(*proof*)

definition *terminates-in* **where**
 $\text{terminates-in } C S \longleftrightarrow (\forall \varphi \in S. \exists \sigma'. \text{single-sem } C (\text{snd } \varphi) \sigma')$

lemma *terminates-inI*:
assumes $\bigwedge \varphi. \varphi \in S \implies \exists \sigma'. \text{single-sem } C (\text{snd } \varphi) \sigma'$
shows $\text{terminates-in } C S$
(*proof*)

lemma *iterate-sem-mono*:
assumes $S \subseteq S'$
shows $\text{iterate-sem } n C S \subseteq \text{iterate-sem } n C S'$
(*proof*)

lemma *terminates-in-while-loop*:
assumes $\text{wf } P \text{ lt}$
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 $\text{terminates-in } (\text{while-cond } b C) S$
(*proof*)

lemma *total-hyper-triple-altI*:
assumes $\bigwedge S. P S \implies Q (\text{sem } C S)$
and $\bigwedge S. P S \implies \text{terminates-in } C S$
shows $\models_{TERM} \{P\} C \{Q\}$
(*proof*)

lemma *syntactic-frame-preserved*:
assumes $\text{terminates-in } C S$

and $wr\ C \cap\ fv\ F = \{\}$
and $sat\text{-}assertion\ vals\ states\ F\ S$
and $wf\text{-}assertion\text{-}aux\ nv\ (length\ states)\ F$
shows $sat\text{-}assertion\ vals\ states\ F\ (sem\ C\ S)$
 $\langle proof \rangle$

theorem *frame-rule-syntactic*:

assumes $\models_{TERM}\ \{P\}\ C\ \{Q\}$
and $wr\ C \cap\ fv\ F = \{\}$
and $wf\text{-}assertion\ F$
shows $\models_{TERM}\ \{conj\ P\ (interp\text{-}assert\ F)\}\ C\ \{conj\ Q\ (interp\text{-}assert\ F)\}$
 $\langle proof \rangle$

7.1 Specialize rule

definition *same-syn-sem-all* :: $'a\ assertion \Rightarrow ((nat, 'a, nat, 'a)\ state \Rightarrow bool) \Rightarrow bool$

where

$same\text{-}syn\text{-}sem\text{-}all\ bsyn\ bsem \longleftrightarrow$
 $(\forall\ states\ vals\ S.\ length\ states > 0 \longrightarrow bsem\ (hd\ states) = sat\text{-}assertion\ vals\ states\ bsyn\ S)$

lemma *same-syn-sem-allI*:

assumes $\bigwedge\ states\ vals\ S.\ length\ states > 0 \implies bsem\ (hd\ states) \longleftrightarrow sat\text{-}assertion\ vals\ states\ bsyn\ S$
shows $same\text{-}syn\text{-}sem\text{-}all\ bsyn\ bsem$
 $\langle proof \rangle$

lemma *transform-assume-valid*:

assumes $same\text{-}syn\text{-}sem\text{-}all\ bsyn\ bsem$
shows $sat\text{-}assertion\ vals\ states\ A\ (Set.filter\ bsem\ S)$
 $\longleftrightarrow sat\text{-}assertion\ vals\ states\ (transform\text{-}assume\ bsyn\ A)\ S$
 $\langle proof \rangle$

fun *indep-of-set* **where**

$indep\text{-}of\text{-}set\ (AForall\ A) \longleftrightarrow indep\text{-}of\text{-}set\ A$
 $| indep\text{-}of\text{-}set\ (AExists\ A) \longleftrightarrow indep\text{-}of\text{-}set\ A$
 $| indep\text{-}of\text{-}set\ (AOr\ A\ B) \longleftrightarrow indep\text{-}of\text{-}set\ A \wedge indep\text{-}of\text{-}set\ B$
 $| indep\text{-}of\text{-}set\ (AAnd\ A\ B) \longleftrightarrow indep\text{-}of\text{-}set\ A \wedge indep\text{-}of\text{-}set\ B$
 $| indep\text{-}of\text{-}set\ (AComp\ -\ -) \longleftrightarrow True$
 $| indep\text{-}of\text{-}set\ (AConst\ -) \longleftrightarrow True$
 $| indep\text{-}of\text{-}set\ (AForallState\ -) \longleftrightarrow False$
 $| indep\text{-}of\text{-}set\ (AExistsState\ -) \longleftrightarrow False$

lemma *indep-of-set-charact*:

assumes $indep\text{-}of\text{-}set\ A$
and $sat\text{-}assertion\ vals\ states\ A\ S$

shows *sat-assertion vals states A S'*
<proof>

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*
<proof>

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*
<proof>

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 \square [φ] A $\{\}$ (is ?A \longleftrightarrow ?B)*
<proof>

definition *get-bsem where*

get-bsem bsyn $\varphi \longleftrightarrow$ sat-assertion \square [φ] 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>

lemma *get-bsem-is-bsem:*

assumes *same-syn-sem-all bsyn bsem*
shows *bsem = get-bsem bsyn*
<proof>

lemma *free-vars-syn-sem:*

assumes *same-syn-sem-all bsyn bsem*
and *fst $\varphi =$ fst φ'*
and *agree-on (fv bsyn) (snd φ) (snd φ')*
and *bsem φ*
and *wf-assertion-aux 0 (Suc 0) bsyn*
shows *bsem φ'*
<proof>

lemma *free-vars-charact*:
assumes $wr\ C \cap fv\ bsyn = \{\}$
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** $?A = ?B$)
 $\langle proof \rangle$

lemma *filter-rule-semantic*:
assumes $\models \{interp\text{-}assert\ P\}\ C\ \{interp\text{-}assert\ Q\}$
and *same-syn-sem-all* $bsyn\ bsem$
and $wr\ C \cap fv\ bsyn = \{\}$
and *wf-assertion-aux* $0\ (Suc\ 0)\ bsyn$
shows $\models \{interp\text{-}assert\ (transform\text{-}assume\ bsyn\ P)\}\ C\ \{interp\text{-}assert\ (transform\text{-}assume\ bsyn\ Q)\}$
 $\langle proof \rangle$

lemma *filter-rule-syntactic*:
assumes $\models \{interp\text{-}assert\ P\}\ C\ \{interp\text{-}assert\ Q\}$
and *indep-of-set* b
and *wf-assertion-aux* $0\ 1\ b$
and $wr\ C \cap fv\ b = \{\}$
shows $\models \{interp\text{-}assert\ (transform\text{-}assume\ b\ P)\}\ C\ \{interp\text{-}assert\ (transform\text{-}assume\ b\ Q)\}$
 $\langle proof \rangle$

definition *terminates where*
 $terminates\ C \longleftrightarrow (\forall\ \sigma. \exists\ \sigma'.\ single\text{-}sem\ C\ \sigma\ \sigma')$

lemma *terminatesI*:
assumes $\bigwedge\ \sigma. \exists\ \sigma'.\ single\text{-}sem\ C\ \sigma\ \sigma'$
shows $terminates\ C$
 $\langle proof \rangle$

lemma *terminates-implies-total*:
assumes $\models \{P\}\ C\ \{Q\}$
and $terminates\ C$
shows $\models TERM\ \{P\}\ C\ \{Q\}$
 $\langle proof \rangle$

lemma *terminates-seq*:
assumes $terminates\ C1$
and $terminates\ C2$
shows $terminates\ (C1;;\ C2)$
 $\langle proof \rangle$

lemma *terminates-assign*:
 $terminates\ (Assign\ x\ e)$

<proof>

lemma *terminates-havoc*:

terminates (Havoc c)

<proof>

lemma *terminates-if*:

assumes *terminates C1*

and *terminates C2*

shows *terminates (If C1 C2)*

<proof>

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>

lemma *terminates-if-then*:

assumes *terminates C1*

and *terminates C2*

shows *terminates (if-then-else b C1 C2)*

<proof>

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 $P (min-prop P) \wedge (\forall m. m < (min-prop P) \longrightarrow \neg P m)$

<proof>

lemma *hyper-tot-set-not-empty*:

assumes $\models_{TERM} \{P\} C \{Q\}$

and $P S$

and $S \neq \{\}$

shows $sem C S \neq \{\}$

<proof>

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'*)
 ⟨*proof*⟩

theorem *while-synchronized-tot*:

assumes *wfP lt*
and $\bigwedge n. \text{not-fv-hyper } t \ (I \ n)$
and $\bigwedge n. \models_{\text{TERM}} \{ \text{conj} \ (\text{conj} \ (I \ n) \ (\text{holds-forall } b)) \ (e\text{-recorded-in-}t \ e \ t) \} \ C$
 $\{ \text{conj} \ (\text{conj} \ (I \ (\text{Suc } n)) \ (\text{low-exp } b)) \ (e\text{-smaller-than-}t \ e \ t \ lt) \}$
shows $\models_{\text{TERM}} \{ \text{conj} \ (I \ 0) \ (\text{low-exp } b) \} \ \text{while-cond } b \ C \ \{ \text{conj} \ (\text{exists } I)$
 $(\text{holds-forall } (\text{lnot } b)) \}$
 ⟨*proof*⟩

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*⟩

theorem *WhileSyncTot*:

assumes *wfP lt*
and *not-fv-hyper t I*
and $\models_{\text{TERM}} \{ \text{conj } I \ (\lambda S. \forall \varphi \in S. b \ (\text{snd } \varphi) \wedge \text{fst } \varphi \ t = e \ (\text{snd } \varphi)) \} \ C \ \{ \text{conj}$
 $(\text{conj } I \ (\text{low-exp } b)) \ (e\text{-smaller-than-}t \ e \ t \ lt) \}$
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*⟩

lemma *total-hyper-tripleE*:

assumes $\models_{\text{TERM}} \{ P \} \ C \ \{ Q \}$
and *P S*
and $\varphi \in S$
shows $\exists \sigma'. (\text{fst } \varphi, \sigma') \in \text{sem } C \ S \wedge \text{single-sem } C \ (\text{snd } \varphi) \ \sigma'$
 ⟨*proof*⟩

theorem *normal-while-tot*:

assumes $\bigwedge n. \models \{ P \ n \} \ \text{Assume } b \ \{ Q \ n \}$
and $\bigwedge n. \models_{\text{TERM}} \{ \text{conj} \ (Q \ n) \ (e\text{-recorded-in-}t \ e \ t) \} \ C \ \{ \text{conj} \ (P \ (\text{Suc } n))$
 $(e\text{-smaller-than-}t \ e \ t \ lt) \}$
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 $\models_{TERM} \{P\ 0\}$ *while-cond* $b\ C\ \{R\}$
<proof>

definition *e-smaller-than-t-weaker* **where**

e-smaller-than-t-weaker $e\ t\ u\ lt\ S \longleftrightarrow (\forall \varphi \in S. \exists \varphi' \in S. fst\ \varphi\ u = fst\ \varphi'\ u \wedge lt\ (e\ (snd\ \varphi))\ (fst\ \varphi'\ t))$

lemma *exists-terminates-loop:*

assumes $wfP\ lt$
and $\bigwedge v. \models \{ (\lambda S. \exists \varphi \in S. e\ (snd\ \varphi) = v \wedge b\ (snd\ \varphi) \wedge P\ \varphi\ S) \}$ *if-then* $b\ C$
 $\{ (\lambda S. \exists \varphi \in S. lt\ (e\ (snd\ \varphi))\ v \wedge P\ \varphi\ S) \}$
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>

definition *t-closed* **where**

t-closed $P\ P\text{-inf} \longleftrightarrow (\forall S. \text{converges-sets}\ S \wedge (\forall n. P\ n\ (S\ n)) \longrightarrow P\text{-inf}\ (\bigcup n. S\ n))$

lemma *t-closedE:*

assumes *t-closed* $P\ P\text{-inf}$
and *converges-sets* S
and $\bigwedge n. P\ n\ (S\ n)$
shows $P\text{-inf}\ (\bigcup n. S\ n)$
<proof>

7.2 Total version of core rules

lemma *total-skip-rule:*

$\models_{TERM} \{P\}$ *Skip* $\{P\}$
<proof>

lemma *total-seq-rule:*

assumes $\models_{TERM} \{P\}\ C1\ \{R\}$
and $\models_{TERM} \{R\}\ C2\ \{Q\}$
shows $\models_{TERM} \{P\}\ Seq\ C1\ C2\ \{Q\}$
<proof>

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>

lemma *total-rule-exists:*

assumes $\bigwedge x. \models_{TERM} \{P\ x\}\ C\ \{Q\ x\}$

shows $\models_{TERM} \{exists P\} C \{exists Q\}$
 $\langle proof \rangle$

lemma *total-assign-rule*:

$\models_{TERM} \{ (\lambda S. P \{ (l, \sigma(x := e \sigma)) \mid l \sigma. (l, \sigma) \in S \}) \} (Assign x e) \{P\}$
 $\langle proof \rangle$

lemma *total-havoc-rule*:

$\models_{TERM} \{ (\lambda S. P \{ (l, \sigma(x := v)) \mid l \sigma v. (l, \sigma) \in S \}) \} (Havoc x) \{P\}$
 $\langle proof \rangle$

lemma *in-semI*:

assumes $\varphi \in S$
and $fst \varphi = fst \varphi'$
and $single-sem C (snd \varphi) (snd \varphi')$
shows $\varphi' \in sem C S$
 $\langle proof \rangle$

theorem *normal-while-tot-stronger*:

fixes $P :: nat \Rightarrow ('lvar, 'lval, 'pvar, 'pval) state set \Rightarrow bool$

assumes $\bigwedge n. \models \{P n\} Assume b \{Q n\}$
and $\bigwedge n. \models_{TERM} \{conj (Q n) (e-recorded-in-t e t)\} C \{conj (P (Suc n))$
 $(e-smaller-than-t-weaker e t u lt)\}$
and $\models \{natural-partition P\} Assume (lnot b) \{R\}$

and $wfP lt$
and $\bigwedge n. not-fv-hyper t (P n)$
and $\bigwedge n. not-fv-hyper t (Q n)$

and $\bigwedge n. not-fv-hyper u (P n)$
and $\bigwedge n. not-fv-hyper u (Q n)$

and $(tr :: 'lval) \neq fa$
and $u \neq t$

shows $\models_{TERM} \{P 0\} while-cond b C \{R\}$
 $\langle proof \rangle$

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**

$$\text{low } l \ S \longleftrightarrow (\forall \varphi 1 \ \varphi 2. \ \varphi 1 \in S \wedge \varphi 2 \in S \longrightarrow \text{snd } \varphi 1 \ l = \text{snd } \varphi 2 \ l)$$

8.1 Examples using the core rules.

definition *GNI* **where**

$$\begin{aligned} \text{GNI } l \ h \ S &\longleftrightarrow (\forall \varphi 1 \ \varphi 2. \ \varphi 1 \in S \wedge \varphi 2 \in S \\ &\longrightarrow (\exists \varphi \in S. \ \text{snd } \varphi \ h = \text{snd } \varphi 1 \ h \wedge \text{snd } \varphi \ l = \text{snd } \varphi 2 \ l)) \end{aligned}$$

lemma *GNI-I*:

$$\begin{aligned} &\text{assumes } \bigwedge \varphi 1 \ \varphi 2. \ \varphi 1 \in S \wedge \varphi 2 \in S \\ &\implies (\exists \varphi \in S. \ \text{snd } \varphi \ h = \text{snd } \varphi 1 \ h \wedge \text{snd } \varphi \ l = \text{snd } \varphi 2 \ l) \\ &\text{shows } \text{GNI } l \ h \ S \\ &\langle \text{proof} \rangle \end{aligned}$$

8.2 Examples using the compositionality rules

definition *has-minimum* $:: 'c \Rightarrow ('d \Rightarrow 'd \Rightarrow \text{bool}) \Rightarrow ('a, 'b, 'c, 'd) \text{ chyperassertion}$
where

$$\text{has-minimum } x \ \text{leq } S \longleftrightarrow (\exists \omega \in S. \ \forall \omega' \in S. \ \text{leq } (\text{snd } \omega \ x) (\text{snd } \omega' \ x))$$

lemma *has-minimumI*:

$$\begin{aligned} &\text{assumes } \omega \in S \\ &\quad \text{and } \bigwedge \omega'. \ \omega' \in S \implies \text{leq } (\text{snd } \omega \ x) (\text{snd } \omega' \ x) \\ &\text{shows } \text{has-minimum } x \ \text{leq } S \\ &\langle \text{proof} \rangle \end{aligned}$$

definition *is-monotonic* **where**

$$\text{is-monotonic } i \ x \ \text{one } \text{two} \ \text{leq } S \longleftrightarrow (\forall \omega \in S. \ \forall \omega' \in S. \ \text{fst } \omega \ i = \text{one} \wedge \text{fst } \omega' \ i = \text{two} \longrightarrow \text{leq } (\text{snd } \omega \ x) (\text{snd } \omega' \ x))$$

lemma *is-monotonicI*:

$$\begin{aligned} &\text{assumes } \bigwedge \omega \ \omega'. \ \omega \in S \implies \omega' \in S \implies \text{fst } \omega \ i = \text{one} \wedge \text{fst } \omega' \ i = \text{two} \implies \text{leq} \\ &(\text{snd } \omega \ x) (\text{snd } \omega' \ x) \\ &\text{shows } \text{is-monotonic } i \ x \ \text{one } \text{two} \ \text{leq } S \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *update-logical-equal-outside*:

$$\text{equal-outside-set } \{i\} (\text{snd } \omega) (\text{snd } (\text{update-logical } \omega \ i \ v))$$

$\langle \text{proof} \rangle$

lemma *update-logical-read*:
 $\text{fst } (\text{update-logical } \omega \ i \ v) \ i = v$
 $\langle \text{proof} \rangle$

lemma *snd-update-logical-same*:
 $\text{snd } (\text{update-logical } \omega \ i \ v) = \text{snd } \omega$
 $\langle \text{proof} \rangle$

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 $\models \{ P \} \ C1 \ ; \ C2 \ \{ \text{has-minimum } y \ \text{leq} \}$
 $\langle \text{proof} \rangle$

In this definition, we use a logical variable for h , which records the initial value of the program variable h

definition $\text{lGNI} :: 'pvar \Rightarrow 'lvar \Rightarrow (('lvar, 'lval, 'pvar, 'pval) \text{ state}) \text{ set} \Rightarrow \text{bool}$
where
 $\text{lGNI } l \ h \ S \iff (\forall \varphi1 \in S. (\forall \varphi2 \in S. \exists \varphi \in S. \text{fst } \varphi \ h = \text{fst } \varphi1 \ h \wedge \text{snd } \varphi \ l = \text{snd } \varphi2 \ l))$

Figure 13

proposition *composing-GNI-with-SNI*:

fixes $h :: 'lvar$
fixes $l :: 'pvar$

assumes $\models \{ (\text{low } l :: (('lvar, 'lval, 'pvar, 'pval) \text{ state}) \text{ hyperassertion}) \} \ C2 \ \{ \text{low } l \}$
and $\models \{ (\text{not-empty} :: (('lvar, 'lval, 'pvar, 'pval) \text{ state}) \text{ hyperassertion}) \} \ C2 \ \{ \text{not-empty} \}$
and $\models \{ (\text{low } l :: (('lvar, 'lval, 'pvar, 'pval) \text{ state}) \text{ hyperassertion}) \} \ C1 \ \{ \text{lGNI } l \ h \}$

```

shows  $\models \{ (low\ l :: ((\text{'lvar}, \text{'lval}, \text{'pvar}, \text{'pval})\ state)\ hyperassertion) \} C1;; C2$ 
 $\{ lGNI\ l\ h \}$ 
 $\langle proof \rangle$ 

```

8.3 Other examples

```

lemma program-1-sat-gni:
  assumes  $y \neq l \wedge y \neq h \wedge l \neq h$ 
  shows  $\vdash \{ low\ l \} Seq\ (Havoc\ y)\ (Assign\ l\ (\lambda\sigma.\ (\sigma\ h :: int) + \sigma\ y)) \{ GNI\ l\ h \}$ 
 $\langle proof \rangle$ 

```

```

lemma program-2-violates-gni:
  assumes  $y \neq l \wedge y \neq h \wedge l \neq h$ 
  shows  $\vdash \{ conj\ (low\ l)\ (\lambda S.\ \exists a \in S.\ \exists b \in S.\ (snd\ a\ h :: nat) \neq snd\ b\ h) \}$ 
 $Seq\ (Seq\ (Havoc\ y)\ (Assume\ (\lambda\sigma.\ \sigma\ y \geq (0 :: nat) \wedge \sigma\ y \leq (100 :: nat))))\ (Assign$ 
 $l\ (\lambda\sigma.\ \sigma\ h + \sigma\ y))$ 
 $\{ \lambda(S :: ((\text{'lvar} \Rightarrow \text{'lval}) \times (\text{'a} \Rightarrow nat))\ set).\ \neg\ GNI\ l\ h\ S \}$ 
 $\langle proof \rangle$ 

```

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 $(\text{'pvar}, \text{'pval}) \text{ pstate}$ (<- you can ctrl+click on the name *pstate* above to jump to its definition).
- Program commands (definition 1) are defined via the type $(\text{'var}, \text{'val}) \text{ stmt}$.
- 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 $(\text{'lvar}, \text{'lval}, \text{'pvar}, \text{'pval}) \text{ state}$ (file Language.thy).
- Hyper-assertions (definition 3) are defined as $(\text{'lvar}, \text{'lval}, \text{'pvar}, \text{'pval}) \text{ state hyperassertion}$ (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:

$$\begin{aligned} \text{sem } C (S1 \cup S2) &= \text{sem } C S1 \cup \text{sem } C S2 \\ S \subseteq S' &\implies \text{sem } C S \subseteq \text{sem } C S' \\ \text{sem } C (\bigcup x. f x) &= (\bigcup x. \text{sem } C (f x)) \\ \text{sem } \text{Skip } S &= S \\ \text{sem } (C1 ;; C2) S &= \text{sem } C2 (\text{sem } C1 S) \\ \text{sem } (\text{If } C1 \ C2) S &= \text{sem } C1 S \cup \text{sem } C2 S \\ \text{sem } (\text{While } C) S &= (\bigcup n. \text{iterate-sem } n \ C \ S) \\ \langle \text{proof} \rangle \end{aligned}$$

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 $\models \{P\} C \{Q\}$
 $\langle proof \rangle$

Theorem 2: Completeness

theorem *thm2-completeness*:
assumes $\models \{P\} C \{Q\}$
shows $\vdash \{P\} C \{Q\}$
 $\langle proof \rangle$

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 $\exists P Q :: (\text{'lvar}, \text{'lval}, \text{'pvar}, \text{'pval}) \text{ state hyperassertion. } (\forall C. \text{hypersat } C \text{ } H \iff \models \{P\} C \{Q\})$
 $\langle proof \rangle$

Theorem 4: Expressing hyper-triples as hyperproperties

theorem *thm4-expressing-hyper-triples-as-hyperproperties*:
 $\models \{P\} C \{Q\} \iff \text{hypersat } C (\text{hyperprop-hht } P \ Q)$
 $\langle proof \rangle$

Theorem 5: Disproving hyper-triples

theorem *thm5-disproving-triples*:
 $\neg \models \{P\} C \{Q\} \iff (\exists P'. \text{sat } P' \wedge \text{entails } P' \ P \wedge \models \{P'\} C \{\lambda S. \neg Q \ S\})$
 $\langle proof \rangle$

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*:

$$\vdash \{ \text{interp-assert } (\text{transform-assign } x \ e \ P) \} \text{Assign } x \ (\text{interp-pexp } e) \{ \text{interp-assert } P \}$$

<proof>

Rule HavocS from figure 3

proposition *HavocS*:

$$\vdash \{ \text{interp-assert } (\text{transform-havoc } x \ P) \} \text{Havoc } x \ { \text{interp-assert } P }$$

<proof>

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*:

$$\vdash \{ \text{interp-assert } (\text{transform-assume } (\text{pbexp-to-assertion } 0 \ b) \ P) \} \text{Assume } (\text{interp-pbexp } b) \ { \text{interp-assert } P }$$

<proof>

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 *pbexp-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 $\vdash \{I\ 0\} \text{ while-cond } b\ C \{ Q \}$
 $\langle \text{proof} \rangle$

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 $\vdash \{ \text{conj } I\ (\text{low-exp } b) \} \text{ while-cond } b\ C \{ \text{conj } (\text{disj } I\ \text{emp})\ (\text{holds-forall } (\text{lnot } b)) \}$
 $\langle \text{proof} \rangle$

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 $\text{entails } P \text{ (low-exp } b)$
and $\vdash \{ \text{conj } P \text{ (holds-forall } b) \} C1 \{ Q \}$
and $\vdash \{ \text{conj } P \text{ (holds-forall (lnot } b)) \} C2 \{ Q \}$
shows $\vdash \{ P \} \text{if-then-else } b \ C1 \ C2 \{ Q \}$
 $\langle \text{proof} \rangle$

This result uses the following construct:

- $\text{if-then-else } b \ C1 \ C2$ is syntactic sugar for $\text{stmt.If (Assume } b \ ; \ ; \ C1) \ (\text{Assume (lnot } b) \ ; \ ; \ C2)$.

Rule $\text{While-}\forall*\exists*$ from figure 5 (presented in subsection 5.2)

theorem $\text{while-forall-exists}$:

assumes $\vdash \{ I \} \text{if-then } b \ C \{ I \}$
and $\vdash \{ I \} \text{Assume (lnot } b) \{ \text{interp-assert } Q \}$
and $\text{no-forall-state-after-existential } Q$
shows $\vdash \{ I \} \text{while-cond } b \ C \{ \text{interp-assert } Q \}$
 $\langle \text{proof} \rangle$

This result uses the following constructs:

- $\text{if-then } b \ C$ is syntactic sugar for $\text{stmt.If (Assume } b \ ; \ ; \ C) \ (\text{Assume (lnot } b))$.
- $\text{no-forall-state-after-existential } Q$ holds iff there is no universal state quantifier $\forall \langle - \rangle$ after any \exists in Q .

Rule $\text{While-}\exists$ 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 $\text{wfp } \text{lt}$
shows $\vdash \{ (\lambda S. \exists \varphi \in S. P \ \varphi \ S) \} \text{while-cond } b \ C \{ (\lambda S. \exists \varphi \in S. Q \ \varphi \ S) \}$
 $\langle \text{proof} \rangle$

$\text{wfp } \text{lt}$ 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*:

$\exists H. (\forall C. \text{hypersat } C\ H \longleftrightarrow HL\ P\ C\ Q)$
<proof>

Proposition 2: Expressing HL in Hyper Hoare Logic

proposition *prop-2-expressing-HL-in-HHL*:

$HL\ P\ C\ Q \longleftrightarrow (\text{hyper-hoare-triple } (\text{over-approx } P)\ C\ (\text{over-approx } Q))$
<proof>

Proposition 3: CHL triples express hyperproperties

proposition *prop-3-CHL-is-hyperproperty*:

$\text{hypersat } C\ (\text{CHL-hyperprop } P\ Q) \longleftrightarrow CHL\ P\ C\ Q$
<proof>

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

<proof>

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*:

$IL\ P\ C\ Q \longleftrightarrow IL\text{-hyperprop}\ P\ Q\ (\text{set-of-traces}\ C)$
<proof>

Proposition 6: Expressing IL in Hyper Hoare Logic

proposition *prop-6-expressing-IL-in-HHL*:

$IL\ P\ C\ Q \longleftrightarrow (\text{hyper-hoare-triple}\ (\text{under-approx}\ P)\ C\ (\text{under-approx}\ Q))$
<proof>

Proposition 7: k - IL triples express hyperproperties

proposition *prop-7-kIL-hyperproperties*:

$\text{hypersat}\ C\ (RIL\text{-hyperprop}\ P\ Q) \longleftrightarrow RIL\ P\ C\ Q$
<proof>

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* ($P :: ('a \Rightarrow ('lvar \Rightarrow 'lval) \times ('pvar \Rightarrow 'pval))\ \text{set})\ x \wedge$
not-free-var-of $P\ c$

and *not-free-var-of* $Q\ x \wedge$ *not-free-var-of* $Q\ c$
shows $RIL\ P\ C\ Q \longleftrightarrow \models \{pre-insec\ from-nat\ x\ c\ P\}\ C\ \{post-insec\ from-nat\ x\ c\ Q\}$
<proof>

proposition *FU-hyperproperties:*
hypersat $C\ (hyperprop-FU\ P\ Q) \longleftrightarrow FU\ P\ C\ Q$
<proof>

Proposition 9: Expressing FU in Hyper Hoare Logic

proposition *prop-9-expressing-FU-in-HHL:*
 $FU\ P\ C\ Q \longleftrightarrow \models \{encode-FU\ P\}\ C\ \{encode-FU\ Q\}$
<proof>

Proposition 10: k-FU triples express hyperproperties

proposition *prop-10-kFU-expresses-hyperproperties:*
hypersat $C\ (RFU-hyperprop\ P\ Q) \longleftrightarrow RFU\ P\ C\ Q$
<proof>

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 $RFU\ P\ C\ Q \longleftrightarrow \models \{encode-RFU\ from-nat\ x\ P\}\ C\ \{encode-RFU\ from-nat\ x\ Q\}$
<proof>

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:*
 $RUE\ P\ C\ Q \longleftrightarrow hypersat\ C\ (hyperprop-RUE\ P\ Q)$
<proof>

Proposition 13: Expressing k-UE in Hyper Hoare Logic

proposition *prop-13-expressing-kUE-in-HHL:*
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 $RUE P C Q \longleftrightarrow \models \{encode-RUE-1\} fn\ fn1\ fn2\ x\ t\ P\} C \{encode-RUE-2\} fn\ fn1\ fn2\ x\ t\ Q\}$
<proof>

Example 3

proposition *proving-refinement*:

fixes $P :: (('lvar \Rightarrow 'lval) \times ('pvar \Rightarrow 'pval))\ set \Rightarrow bool$
and $t :: 'pvar$
assumes $(one :: 'pval) \neq two$ — We assume two distinct program values *one* and *two*, to represent 1 and 2.
and $P = (\lambda S. card\ S = 1)$
and $Q = (\lambda S. \forall \varphi \in S. snd\ \varphi\ t = two \longrightarrow (fst\ \varphi, (snd\ \varphi)(t := one)) \in S)$
and *not-free-var-stmt* $t\ C1$
and *not-free-var-stmt* $t\ C2$
shows *refinement* $C2\ C1 \longleftrightarrow$
 $\models \{ P \} If\ (Seq\ (Assign\ t\ (\lambda-. one))\ C1)\ (Seq\ (Assign\ t\ (\lambda-. two))\ C2)\ \{ Q \}$
<proof>

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 $\bigwedge \varphi1\ (\varphi2 :: ('a, 'b, 'c, 'd)\ state). fst\ \varphi1 = fst\ \varphi2 \wedge (\vdash \{ (in-set\ \varphi1 :: (('a, 'b, 'c, 'd)\ state)\ hyperassertion) \} C \{ in-set\ \varphi2 \})$
 $\implies (\vdash \{ (P\ \varphi1 :: (('a, 'b, 'c, 'd)\ state)\ hyperassertion) \} C \{ Q\ \varphi2 \})$
shows $\vdash \{ ((\lambda S. \forall \varphi1 \in S. P\ \varphi1\ S) :: (('a, 'b, 'c, 'd)\ state)\ hyperassertion) \} C \{ (\lambda S. \forall \varphi2 \in S. Q\ \varphi2\ S) \}$
<proof>

proposition *rule-And*:

assumes $\vdash \{ P \} C \{ Q \}$
and $\vdash \{ P' \} C \{ Q' \}$
shows $\vdash \{ conj\ P\ P' \} C \{ conj\ Q\ Q' \}$
<proof>

proposition *rule-Or*:

assumes $\vdash \{ P \} C \{ Q \}$
and $\vdash \{ P' \} C \{ Q' \}$
shows $\vdash \{ disj\ P\ P' \} C \{ disj\ Q\ Q' \}$
<proof>

proposition *rule-FrameSafe*:

assumes $wr\ C \cap fv\ F = \{ \}$

and *wf-assertion* F
and *no-exists-state* F
shows $\vdash \{ \text{interp-assert } F \} C \{ \text{interp-assert } F \}$
 $\langle \text{proof} \rangle$

proposition *rule-Forall*:
assumes $\bigwedge x. \vdash \{ P \ x \} C \{ Q \ x \}$
shows $\vdash \{ \text{forall } P \} C \{ \text{forall } Q \}$
 $\langle \text{proof} \rangle$

proposition *rule-IndexedUnion*:
assumes $\bigwedge x. \vdash \{ P \ x \} C \{ Q \ x \}$
shows $\vdash \{ \text{general-join } P \} C \{ \text{general-join } Q \}$
 $\langle \text{proof} \rangle$

proposition *rule-Union*:
assumes $\vdash \{ P \} C \{ Q \}$
and $\vdash \{ P' \} C \{ Q' \}$
shows $\vdash \{ \text{join } P \ P' \} C \{ \text{join } Q \ Q' \}$
 $\langle \text{proof} \rangle$

proposition *rule-BigUnion*:
fixes $P :: ((('a \Rightarrow 'b) \times ('c \Rightarrow 'd)) \text{ set} \Rightarrow \text{bool})$
assumes $\vdash \{ P \} C \{ Q \}$
shows $\vdash \{ \text{general-union } P \} C \{ \text{general-union } Q \}$
 $\langle \text{proof} \rangle$

proposition *rule-Specialize*:
assumes $\vdash \{ \text{interp-assert } P \} C \{ \text{interp-assert } Q \}$
and *indep-of-set* b
and *wf-assertion-aux* $0 \ 1 \ b$
and $wr \ C \cap \text{fv } b = \{ \}$
shows $\vdash \{ \text{interp-assert } (\text{transform-assume } b \ P) \} C \{ \text{interp-assert } (\text{transform-assume } b \ Q) \}$
 $\langle \text{proof} \rangle$

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 $\vdash \{ P \} C \{ Q \}$
 $\langle \text{proof} \rangle$

proposition *rule-LUpdateSyntactic*:
assumes $\vdash \{ (\lambda S. P \ S \wedge \text{e-recorded-in-t } e \ t \ S) \} C \{ Q \}$

and *not-fv-hyper* $t P$
and *not-fv-hyper* $t Q$
shows $\vdash \{ P \} C \{ Q \}$
 $\langle proof \rangle$

proposition *rule-AtMost*:
assumes $\vdash \{ P \} C \{ Q \}$
shows $\vdash \{ has-superset P \} C \{ has-superset Q \}$
 $\langle proof \rangle$

proposition *rule-AtLeast*:
assumes $\vdash \{ P \} C \{ Q \}$
shows $\vdash \{ has-subset P \} C \{ has-subset Q \}$
 $\langle proof \rangle$

proposition *rule-True*:
 $\vdash \{ P \} C \{ \lambda-. True \}$
 $\langle proof \rangle$

proposition *rule-False*:
 $\vdash \{ (\lambda-. False) \} C \{ Q \}$
 $\langle proof \rangle$

proposition *rule-Empty*:
 $\vdash \{ (\lambda S. S = \{\}) \} C \{ (\lambda S. S = \{\}) \}$
 $\langle proof \rangle$

9.6.2 Appendix D.2: Examples

Example shown in figure 12. To see the actual proof, ctrl+click on $\llbracket = \{ ?P \} ?C1.0 \{ has-minimum ?x ?leq \}; \models \{ is-monotonic ?i ?x ?one ?two ?leq \} ?C2.0 \{ is-monotonic ?i ?y ?one ?two ?leq \}; \models \{ is-singleton \} ?C2.0 \{ is-singleton \}; ?one \neq ?two; \wedge x. ?leq x x \rrbracket \implies \models \{ ?P \} ?C1.0 ;; ?C2.0 \{ has-minimum ?y ?leq \}$.

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 $\vdash \{ P \} C1 \{ has-minimum x leq \}$
and $\vdash \{ is-monotonic i x one two leq \} C2 \{ is-monotonic i y one two leq \}$
and $\vdash \{ (is-singleton :: ((('a \Rightarrow 'b) \times ('c \Rightarrow 'd)) set \Rightarrow bool)) \} C2 \{ is-singleton \}$
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 $\vdash \{ P \} C1 ;; C2 \{ \text{has-minimum } y \text{ leq} \}$
 $\langle \text{proof} \rangle$

Example shown in figure 13. To see the actual proof, ctrl+click on $\llbracket \models \{ \text{low } ?l \} ?C2.0 \{ \text{low } ?l \}; \models \{ \text{not-empty} \} ?C2.0 \{ \text{not-empty} \}; \models \{ \text{low } ?l \} ?C1.0 \{ \text{lGNI } ?l \ ?h \} \rrbracket \implies \models \{ \text{low } ?l \} ?C1.0 ;; ?C2.0 \{ \text{lGNI } ?l \ ?h \}$.

proposition *fig-13-composing-GNI-with-SNI*:

fixes $h :: 'lvar$
fixes $l :: 'pvar$
assumes $\vdash \{ (\text{low } l :: ((\text{'lvar}, \text{'lval}, \text{'pvar}, \text{'pval}) \text{ state}) \text{ hyperassertion}) \} C2 \{ \text{low } l \}$
and $\vdash \{ (\text{not-empty} :: ((\text{'lvar}, \text{'lval}, \text{'pvar}, \text{'pval}) \text{ state}) \text{ hyperassertion}) \} C2 \{ \text{not-empty} \}$
and $\vdash \{ (\text{low } l :: ((\text{'lvar}, \text{'lval}, \text{'pvar}, \text{'pval}) \text{ state}) \text{ hyperassertion}) \} C1 \{ \text{lGNI } l \ h \}$
shows $\vdash \{ (\text{low } l :: ((\text{'lvar}, \text{'lval}, \text{'pvar}, \text{'pval}) \text{ state}) \text{ hyperassertion}) \} C1 ;; C2 \{ \text{lGNI } l \ h \}$
 $\langle \text{proof} \rangle$

9.7 Appendix E: Termination-Based Reasoning

Terminating hyper-triples (definition 24) are defined as *total-hyper-triple*, and usually written $\models \text{TERM} \{ P \} C \{ Q \}$.

theorem *rule-Frame*:

assumes $\models \text{TERM} \{ P \} C \{ Q \}$
and $wr \ C \cap fv \ F = \{ \}$
and *wf-assertion* F
shows $\models \text{TERM} \{ \text{conj } P \ (\text{interp-assert } F) \} C \{ \text{conj } Q \ (\text{interp-assert } F) \}$
 $\langle \text{proof} \rangle$

theorem *rule-WhileSyncTerm*:

assumes $\models \text{TERM} \{ \text{conj } I \ (\lambda S. \forall \varphi \in S. b \ (\text{snd } \varphi) \wedge \text{fst } \varphi \ t = e \ (\text{snd } \varphi)) \} C \{ \text{conj } (\text{conj } I \ (\text{low-exp } b)) \ (\text{e-smaller-than-t } e \ t \ lt) \}$
and *wfP* lt
and *not-fv-hyper* $t \ I$
shows $\models \text{TERM} \{ \text{conj } I \ (\text{low-exp } b) \} \text{ while-cond } b \ C \{ \text{conj } I \ (\text{holds-forall } (\text{not } b)) \}$
 $\langle \text{proof} \rangle$

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 $\models \{ \text{combine from-nat } x \ P1 \ P2 \} C \{ \text{combine from-nat } x \ R1 \ R2 \}$

```

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  $\neq$  from-nat 2 — We can represent 1 and 2 as distinct logical
values.
and not-free-var-hyper  $x R1$ 
and not-free-var-hyper  $x R2$ 
shows  $\models \{P\} \text{If } (Seq C1 (Seq C C1')) (Seq C2 (Seq C C2')) \{join Q1 Q2\}$ 
 $\langle proof \rangle$ 

```

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.
- [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.
- [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).