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

Thibault Dardinier
Department of Computer Science
ETH Zurich, Switzerland

April 18, 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* ⇒ *'val*

type-synonym (*'var*, *'val*) *bexp* = (*'var*, *'val*) *pstate* ⇒ *bool*
type-synonym (*'var*, *'val*) *exp* = (*'var*, *'val*) *pstate* ⇒ *'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* ⇒ (*'var*, *'val*) *pstate* ⇒ (*'var*, *'val*) *pstate* ⇒ *bool*
  (⟨-, -⟩ → - [*51*,*0*] *81*)
  where
  SemSkip: ⟨*Skip*, *σ*⟩ → *σ*
| *SemAssign*: ⟨*Assign var e*, *σ*⟩ → *σ*(*var* := (*e σ*))
| *SemSeq*: ⟦ ⟨*C1*, *σ*⟩ → *σ1*; ⟨*C2*, *σ1*⟩ → *σ2* ⟧ ⟹ ⟨*Seq C1 C2*, *σ*⟩ → *σ2*
| *SemIf1*: ⟨*C1*, *σ*⟩ → *σ1* ⟹ ⟨*If C1 C2*, *σ*⟩ → *σ1*
| *SemIf2*: ⟨*C2*, *σ*⟩ → *σ2* ⟹ ⟨*If C1 C2*, *σ*⟩ → *σ2*
| *SemHavoc*: ⟨*Havoc var*, *σ*⟩ → *σ*(*var* := *v*)
| *SemAssume*: *b σ* ⟹ ⟨*Assume b*, *σ*⟩ → *σ*
| *SemWhileIter*: ⟦ ⟨*C*, *σ*⟩ → *σ'* ; ⟨*While C*, *σ'*⟩ → *σ''* ⟧ ⟹ ⟨*While C*, *σ*⟩ → *σ''*
| *SemWhileExit*: ⟨*While C*, *σ*⟩ → *σ*

inductive-cases *single-sem-Seq-elim*[*elim*!]: ⟨*Seq C1 C2*, *σ*⟩ → *σ'*

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

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**
$sem\ C\ S = \{\ (l, \sigma')\ |\sigma'\ \sigma\ l.\ (l, \sigma) \in S \wedge \langle C, \sigma\rangle \to \sigma'\ \}$

lemma *in-sem*:
$\varphi \in sem\ C\ S \longleftrightarrow (\exists \sigma.\ (fst\ \varphi, \sigma) \in S \wedge single\text{-}sem\ C\ \sigma\ (snd\ \varphi))$ (**is** *?A* $\longleftrightarrow$ *?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*:
$sem\ C\ (\bigcup x.\ f\ x) = (\bigcup x.\ sem\ C\ (f\ x))$ (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *sem-monotonic*:
assumes $S \subseteq S'$
shows $sem\ C\ S \subseteq sem\ C\ S'$
⟨*proof*⟩

lemma *subsetPairI*:
assumes $\bigwedge l\ \sigma.\ (l, \sigma) \in A \Longrightarrow (l, \sigma) \in B$
shows $A \subseteq B$

⟨*proof*⟩

lemma *sem-if*:
sem (*If C1 C2*) *S* = *sem C1 S* ∪ *sem C2 S* (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *sem-assume*:
sem (*Assume b*) *S* = { (*l*, *σ*) |*l σ*. (*l*, *σ*) ∈ *S* ∧ *b σ* } (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *while-then-reaches*:
assumes (*single-sem C*)** *σ σ''*
shows *single-sem* (*While C*) *σ σ''*
⟨*proof*⟩

lemma *in-closure-then-while*:
assumes *single-sem C' σ σ''*
shows ⋀*C*. *C'* = *While C* ⟹ (*single-sem C*)** *σ σ''*
⟨*proof*⟩

theorem *loop-equiv*:
single-sem (*While C*) *σ σ'* ⟷ (*single-sem C*)** *σ σ'*
⟨*proof*⟩

fun *iterate-sem* **where**
iterate-sem 0 - S = *S*
| *iterate-sem* (*Suc n*) *C S* = *sem C* (*iterate-sem n C S*)

lemma *in-iterate-then-in-trans*:
assumes (*l*, *σ''*) ∈ *iterate-sem n C S*
shows ∃*σ*. (*l*, *σ*) ∈ *S* ∧ (*single-sem C*)** *σ σ''*
⟨*proof*⟩

lemma *reciprocal*:
assumes (*single-sem C*)** *σ σ''*
and (*l*, *σ*) ∈ *S*
shows ∃ *n*. (*l*, *σ''*) ∈ *iterate-sem n C S*
⟨*proof*⟩

lemma *union-iterate-sem-trans*:
(*l*, *σ''*) ∈ (⋃ *n*. *iterate-sem n C S*) ⟷ (∃*σ*. (*l*, *σ*) ∈ *S* ∧ (*single-sem C*)** *σ* *σ''*) (**is** *?A* ⟷ *?B*)
⟨*proof*⟩

lemma *sem-while*:
sem (*While C*) *S* = (⋃ *n*. *iterate-sem n C S*) (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *assume-sem*:
sem (*Assume b*) *S* = *Set.filter* (*b* ∘ *snd*) *S* (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *sem-split-general*:
sem C (⋃ *x*. *F x*) = (⋃ *x*. *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* ∪ *written-vars C2*
| *written-vars* (*If C1 C2*) = *written-vars C1* ∪ *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* ∉ *written-vars C*
shows φ *x* = φ′ *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* ⇒ *bool*)

definition *entails* **where**
entails A B ⟷ (∀ *S*. *A S* ⟶ *B S*)

lemma *entails-refl*:
entails A A
⟨*proof*⟩

lemma *entailsI*:

assumes $\bigwedge S.\ A\ S \Longrightarrow B\ S$
shows *entails A B*
⟨*proof*⟩

lemma *entailsE*:
assumes *entails A B*
and *A x*
shows *B x*
⟨*proof*⟩

lemma *bientails-equal*:
assumes *entails A B*
and *entails B A*
shows $A = B$
⟨*proof*⟩

lemma *entails-trans*:
assumes *entails A B*
and *entails B C*
shows *entails A C*
⟨*proof*⟩

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

lemma *sem-assume-setify*:
sem (*Assume b*) *S* = *S* ∩ *setify-prop b* (**is** *?A* = *?B*)
⟨*proof*⟩

definition *over-approx* :: *'a set* ⇒ *'a hyperassertion* **where**
over-approx P S ⟷ *S* ⊆ *P*

definition *lower-closed* :: *'a hyperassertion* ⇒ *bool* **where**
lower-closed P ⟷ (∀ *S S'*. *P S* ∧ *S'* ⊆ *S* ⟶ *P S'*)

lemma *over-approx-lower-closed*:
lower-closed (*over-approx P*)
⟨*proof*⟩

definition *under-approx* :: *'a set* ⇒ *'a hyperassertion* **where**
under-approx P S ⟷ *P* ⊆ *S*

definition *upper-closed* :: *'a hyperassertion* ⇒ *bool* **where**
upper-closed P ⟷ (∀ *S S'*. *P S* ∧ *S* ⊆ *S'* ⟶ *P S'*)

lemma *under-approx-upper-closed*:
upper-closed (*under-approx P*)
⟨*proof*⟩

definition *closed-by-union* :: *'a hyperassertion* ⇒ *bool* **where**
closed-by-union P ⟷ (∀ *S S'*. *P S* ∧ *P S'* ⟶ *P* (*S* ∪ *S'*))

lemma *closed-by-unionI*:
assumes ⋀*a b*. *P a* ⟹ *P b* ⟹ *P* (*a* ∪ *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 ⟷ *P S* ∧ *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 ⟷ *P S* ∨ *Q S*

definition *exists* :: (*'c* ⇒ *'a hyperassertion*) ⇒ *'a hyperassertion* **where**
exists P S ⟷ (∃ *x*. *P x S*)

definition *forall* :: (*'c* ⇒ *'a hyperassertion*) ⇒ *'a hyperassertion* **where**
forall P S ⟷ (∀ *x*. *P x S*)

lemma *over-inter*:
entails (*over-approx* (*P* ∩ *Q*)) (*conj* (*over-approx P*) (*over-approx Q*))
⟨*proof*⟩

lemma *over-union*:
entails (*disj* (*over-approx P*) (*over-approx Q*)) (*over-approx* (*P* ∪ *Q*))
⟨*proof*⟩

lemma *under-union*:
entails (*under-approx* (*P* ∪ *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* ⇒ *'a hyperassertion* ⇒ *'a hyperassertion* **where**
join A B S ⟷ ($\exists$ *SA SB*. *A SA* ∧ *B SB* ∧ *S* = *SA* ∪ *SB*)

definition *general-join* :: (*'b* ⇒ *'a hyperassertion*) ⇒ *'a hyperassertion* **where**
general-join f S ⟷ ($\exists$ *F*. *S* = ($\bigcup$ *x*. *F x*) ∧ (∀ *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 $\bigotimes$ (for $x \in X$)

definition *natural-partition* **where**
natural-partition I S ⟷ ($\exists$ *F*. *S* = ($\bigcup$ *n*. *F n*) ∧ (∀ *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*) $\bigwedge$*n*. *I n* (*F n*)
⟨*proof*⟩

2.1 Rules of the Logic

Core rules from figure 2

inductive *syntactic-HHT* ::

(('lvar, 'lval, 'pvar, 'pval) state hyperassertion) ⇒ ('pvar, 'pval) stmt ⇒ (('lvar, 'lval, 'pvar, 'pval) state hyperassertion) ⇒ bool
(⊢ {-} - {-} [51,0,0] 81) **where**
RuleSkip: ⊢ {P} Skip {P}
| *RuleCons*: ⟦ entails P P' ; entails Q' Q ; ⊢ {P'} C {Q'} ⟧ ⟹ ⊢ {P} C {Q}
| *RuleSeq*: ⟦ ⊢ {P} C1 {R} ; ⊢ {R} C2 {Q} ⟧ ⟹ ⊢ {P} (Seq C1 C2) {Q}
| *RuleIf*: ⟦ ⊢ {P} C1 {Q1} ; ⊢ {P} C2 {Q2} ⟧ ⟹ ⊢ {P} (If C1 C2) {join Q1 Q2}
| *RuleWhile*: ⟦ ⋀n. ⊢ {I n} C {I (Suc n)} ⟧ ⟹ ⊢ {I 0} (While C) {natural-partition I}
| *RuleAssume*: ⊢ { (λS. P (Set.filter (b ∘ snd) S)) } (Assume b) {P}
| *RuleAssign*: ⊢ { (λS. P { (l, σ(x := e σ)) |l σ. (l, σ) ∈ S }) } (Assign x e) {P}
| *RuleHavoc*: ⊢ { (λS. P { (l, σ(x := v)) |l σ v. (l, σ) ∈ S }) } (Havoc x) {P}
| *RuleExistsSet*: ⟦⋀x::('lvar, 'lval, 'pvar, 'pval) state set. ⊢ {P x} C {Q x}⟧ ⟹ ⊢ {exists P} C {exists Q}

2.2 Soundness

Definition 5: Hyper-Triples

definition *hyper-hoare-triple* (⊨ {-} - {-} [51,0,0] 81) **where**
⊨ {P} C {Q} ⟷ (∀ S. P S ⟶ Q (sem C S))

lemma *hyper-hoare-tripleI*:
assumes ⋀S. P S ⟹ Q (sem C S)
shows ⊨ {P} C {Q}
⟨*proof*⟩

lemma *hyper-hoare-tripleE*:
assumes ⊨ {P} C {Q}
and P S
shows Q (sem C S)
⟨*proof*⟩

lemma *consequence-rule*:
assumes entails P P'
and entails Q' Q
and ⊨ {P'} C {Q'}
shows ⊨ {P} C {Q}
⟨*proof*⟩

lemma *skip-rule*:
⊨ {P} Skip {P}
⟨*proof*⟩

lemma *assume-rule*:
⊨ { (λS. P (Set.filter (b ∘ snd) S)) } (Assume b) {P}
⟨*proof*⟩

lemma *seq-rule*:

assumes $\models \{P\}\ C1\ \{R\}$
and $\models \{R\}\ C2\ \{Q\}$
shows $\models \{P\}\ Seq\ C1\ C2\ \{Q\}$
⟨*proof*⟩

lemma *if-rule*:
assumes $\models \{P\}\ C1\ \{Q1\}$
and $\models \{P\}\ C2\ \{Q2\}$
shows $\models \{P\}\ If\ C1\ C2\ \{join\ Q1\ Q2\}$
⟨*proof*⟩

lemma *sem-assign*:
$sem\ (Assign\ x\ e)\ S = \{(l, \sigma(x := e\ \sigma))\ |l\ \sigma.\ (l, \sigma) \in S\}$ (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *assign-rule*:
$\models \{\ (\lambda S.\ P\ \{\ (l, \sigma(x := e\ \sigma))\ |l\ \sigma.\ (l, \sigma) \in S\ \})\ \}\ (Assign\ x\ e)\ \{P\}$
⟨*proof*⟩

lemma *sem-havoc*:
$sem\ (Havoc\ x)\ S = \{(l, \sigma(x := v))\ |l\ \sigma\ v.\ (l, \sigma) \in S\}$ (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *havoc-rule*:
$\models \{\ (\lambda S.\ P\ \{\ (l, \sigma(x := v))\ |l\ \sigma\ v.\ (l, \sigma) \in S\ \})\ \}\ (Havoc\ x)\ \{P\}$
⟨*proof*⟩

Loops

lemma *indexed-invariant-then-power*:
assumes $\bigwedge n.$ *hyper-hoare-triple* $(I\ n)\ C\ (I\ (Suc\ n))$
and $I\ 0\ S$
shows $I\ n\ (iterate\text{-}sem\ n\ C\ S)$
⟨*proof*⟩

lemma *indexed-invariant-then-power-bounded*:
assumes $\bigwedge m.\ m < n \Longrightarrow$ *hyper-hoare-triple* $(I\ m)\ C\ (I\ (Suc\ m))$
and $I\ 0\ S$
shows $I\ n\ (iterate\text{-}sem\ n\ C\ S)$
⟨*proof*⟩

lemma *while-rule*:
assumes $\bigwedge n.$ *hyper-hoare-triple* $(I\ n)\ C\ (I\ (Suc\ n))$
shows *hyper-hoare-triple* $(I\ 0)\ (While\ C)\ (natural\text{-}partition\ I)$
⟨*proof*⟩

lemma *rule-exists*:
assumes $\bigwedge x. \models \{P\ x\}\ C\ \{Q\ x\}$
shows $\models \{exists\ P\}\ C\ \{exists\ Q\}$
⟨*proof*⟩

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

lemma *completeI*:
assumes $\models \{P\}\ C\ \{Q\} \Longrightarrow \vdash \{P\}\ C\ \{Q\}$
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* ($\lambda S'.\ \exists S.\ A\ S \wedge S' = sem\ C1\ S \cup sem\ C2\ S$) *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 $\exists R.$ *hyper-hoare-triple A C1 R* $\wedge$ *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 ⋀*R*. *complete P C1 R*
and ⋀*R*. *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*) = (λ*S*. (∃ *S'*. *S* = *sem C S'* ∧ *construct-inv P C n S'*))

lemma *iterate-sem-ind*:
assumes *construct-inv P C n S'*
shows ∃ *S*. *P S* ∧ *S'* = *iterate-sem n C S*
⟨*proof*⟩

lemma *complete-while-aux*:
assumes *hyper-hoare-triple* (λ*S*. *P S* ∧ *S* = *V*) (*While C*) *Q*
shows *entails* (*natural-partition* (*construct-inv* (λ*S*. *P S* ∧ *S* = *V*) *C*)) *Q*
⟨*proof*⟩

lemma *complete-while*:
fixes *P Q* :: ('*lvar*, '*lval*, '*pvar*, '*pval*) *state hyperassertion*
assumes ⋀*P' Q'* :: ('*lvar*, '*lval*, '*pvar*, '*pval*) *state hyperassertion*. *complete P' C Q'*
shows *complete P* (*While C*) *Q*
⟨*proof*⟩

Theorem 2

theorem *completeness*:
fixes *P Q* :: ('*lvar*, '*lval*, '*pvar*, '*pval*) *state hyperassertion*
assumes ⊨ {*P*} *C* {*Q*}
shows ⊢ {*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*)
⟨*proof*⟩

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

lemma *differ-only-byI*:
assumes $\bigwedge y.\ y \neq x \Longrightarrow a\ y = b\ y$
shows *differ-only-by a b x*
⟨*proof*⟩

lemma *diff-by-update*:
differ-only-by $(a(x := v))\ a\ x$
⟨*proof*⟩

lemma *diff-by-comm*:
differ-only-by a b x $\longleftrightarrow$ *differ-only-by b a x*
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *not-free-var-ofE*:
assumes *not-free-var-of P x*
and $\bigwedge i.\ differ\text{-}only\text{-}by\ (fst\ (states\ i))\ (fst\ (states'\ i))\ x$
and $\bigwedge i.\ snd\ (states\ i) = snd\ (states'\ i)$
and $states \in P$
shows $states' \in P$
⟨*proof*⟩

2.5 Synchronized Rule for Branching

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

lemma *combineI*:
assumes *P1* (*Set.filter* ($\lambda\varphi$. *fst* φ *x* = *from-nat 1*) *S*) $\wedge$ *P2* (*Set.filter* ($\lambda\varphi$. *fst* φ *x* = *from-nat 2*) *S*)
shows *combine from-nat x P1 P2 S*
⟨*proof*⟩

definition *modify-lvar-to* **where**
modify-lvar-to x v φ = ((*fst* φ)(*x* := *v*), *snd* φ)

lemma *logical-var-in-sem-same*:
assumes $\bigwedge\varphi$. $\varphi \in S \Longrightarrow$ *fst* φ *x* = *a*
and $\varphi' \in$ *sem C S*
shows *fst* φ' *x* = *a*
⟨*proof*⟩

lemma *recover-after-sem*:
assumes *a* ≠ *b*
and $\bigwedge\varphi$. $\varphi \in$ *S1* $\Longrightarrow$ *fst* φ *x* = *a*
and $\bigwedge\varphi$. $\varphi \in$ *S2* $\Longrightarrow$ *fst* φ *x* = *b*
shows *sem C S1* = *Set.filter* ($\lambda\varphi$. *fst* φ *x* = *a*) (*sem C* (*S1* ∪ *S2*)) (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *injective-then-ok*:
assumes *a* ≠ *b*
and *S1'* = (*modify-lvar-to x a*) ‘ *S1*
and *S2'* = (*modify-lvar-to x b*) ‘ *S2*
shows *Set.filter* ($\lambda\varphi$. *fst* φ *x* = *a*) (*S1'* ∪ *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* ≠ *b* $\longrightarrow$ *f a* ≠ *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 $(l, \sigma') \in sem\ C\ S$
⟨*proof*⟩

lemma *set-of-traces-same*:
assumes $\bigwedge x.\ to\text{-}pvar\ (to\text{-}lvar\ x) = x$
and $\bigwedge x.\ to\text{-}pval\ (to\text{-}lval\ x) = x$
and $S = \{(copy\text{-}p\text{-}state\ to\text{-}pvar\ to\text{-}lval\ \sigma, \sigma)\ |\sigma.\ True\}$
shows $\{(recover\text{-}p\text{-}state\ to\text{-}pval\ to\text{-}lvar\ l, \sigma')\ |l\ \sigma'.\ (l, \sigma') \in sem\ C\ S\} = set\text{-}of\text{-}traces\ C$
(**is** *?A = ?B*)
⟨*proof*⟩

Theorem 3

theorem *proving-hyperproperties*:

fixes *to-lvar* :: *'pvar* ⇒ *'lvar*
fixes *to-lval* :: *'pval* ⇒ *'lval*

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

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

Hypersafety, hyperliveness

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

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

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

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

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

lemma *hypersafetyI*:
assumes $\bigwedge S\ S'.\ \neg\ P\ S \Longrightarrow S \subseteq S' \Longrightarrow \neg\ P\ S'$
shows *hypersafety P*

⟨*proof*⟩

lemma *hyperlivenessI*:
assumes $\bigwedge S.\ \exists\, S'.\ S \subseteq S' \wedge P\ S'$
shows *hyperliveness P*
⟨*proof*⟩

lemma *k-hypersafe-is-hypersafe*:
assumes *k-hypersafety k P*
shows *hypersafety P*
⟨*proof*⟩

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*)
⟨*proof*⟩

definition *hoarify* **where**
hoarify P Q S $\longleftrightarrow$ $(\forall\, p \in S.\ fst\ p \in P \longrightarrow snd\ p \in Q)$

lemma *hoarify-hypersafety*:
hypersafety (*hoarify P Q*)
⟨*proof*⟩

theorem *hypersafety-1-hoare-logic*:
k-hypersafety 1 (*hoarify P Q*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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 $\Sigma\ S = \{\ (l, \sigma')\ |\sigma'\ \sigma\ l.\ (l, \sigma) \in S \wedge (\sigma, \sigma') \in \Sigma\ \}$

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

Theorem 4

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

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 $\bigwedge \sigma\ \sigma'\ l.\ (l, \sigma) \in P \Longrightarrow \langle C, \sigma\rangle \rightarrow \sigma' \Longrightarrow (l, \sigma') \in Q$
shows *HL P C Q*
⟨*proof*⟩

lemma *hoarifyI*:
assumes $\bigwedge \sigma\ \sigma'.\ (\sigma, \sigma') \in S \Longrightarrow \sigma \in P \Longrightarrow \sigma' \in Q$
shows *hoarify P Q S*
⟨*proof*⟩

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

lemma *connection-HL*:
HL P C Q $\longleftrightarrow$ *HL-hyperprop P Q* (*set-of-traces C*) (**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

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

Proposition 2 **theorem** *encoding-HL*:
HL P C Q $\longleftrightarrow$ (*hyper-hoare-triple* (*over-approx P*) *C* (*over-approx Q*)) (**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

lemma *entailment-order-hoare*:
assumes $P \subseteq P'$
shows *entails* (*over-approx P*) (*over-approx P'*)
⟨*proof*⟩

3.3 Cartesian Hoare Logic (CHL) [10]

definition *k-sem* **where**
k-sem C states states' $\longleftrightarrow$ ($\forall$ *i*. (*fst* (*states i*) = *fst* (*states' i*) $\wedge$ *single-sem C* (*snd* (*states i*)) (*snd* (*states' i*))))

lemma *k-semI*:
assumes $\bigwedge$*i*. (*fst* (*states i*) = *fst* (*states' i*) $\wedge$ *single-sem C* (*snd* (*states i*)) (*snd* (*states' i*)))
shows *k-sem C states states'*
⟨*proof*⟩

lemma *k-semE*:
assumes *k-sem C states states'*
shows *fst* (*states i*) = *fst* (*states' i*) $\wedge$ *single-sem C* (*snd* (*states i*)) (*snd* (*states' i*))
⟨*proof*⟩

Definition 17 **definition** *CHL* **where**
CHL P C Q $\longleftrightarrow$ ($\forall$ *states*. *states* $\in$ *P* $\longrightarrow$ ($\forall$ *states'*. *k-sem C states states'* $\longrightarrow$ *states'* $\in$ *Q*))

lemma *CHLI*:
assumes $\bigwedge$*states states'*. *states* $\in$ *P* $\Longrightarrow$ *k-sem C states states'* $\Longrightarrow$ *states'* $\in$ *Q*
shows *CHL P C Q*
⟨*proof*⟩

lemma *CHLE*:
assumes *CHL P C Q*
and *states* $\in$ *P*
and *k-sem C states states'*
shows *states'* $\in$ *Q*
⟨*proof*⟩

definition *encode-CHL* **where**
encode-CHL from-nat x P S $\longleftrightarrow$ ($\forall$ *states*. ($\forall$ *i*. *states i* $\in$ *S* $\wedge$ *fst* (*states i*) *x* = *from-nat i*) $\longrightarrow$ *states* $\in$ *P*)

lemma *encode-CHLI*:
assumes $\bigwedge$*states*. ($\forall$ *i*. *states i* $\in$ *S* $\wedge$ *fst* (*states i*) *x* = *from-nat i*) $\Longrightarrow$ *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*. *states i* $\in$ *S*
and $\bigwedge$*i*. *fst* (*states i*) *x* = *from-nat i*
shows *states* $\in$ *P*
⟨*proof*⟩

lemma *equal-change-lvar*:
assumes *fst* φ *x* = *y*
shows φ = ((*fst* φ)(*x* := *y*), *snd* φ)
⟨*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 *CHL P C Q* $\longleftrightarrow$ $\models$ {*encode-CHL from-nat x P*} *C* {*encode-CHL from-nat x Q*} (**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

definition *CHL-hyperprop* **where**
CHL-hyperprop P Q S $\longleftrightarrow$ ($\forall$ *l p*. ($\forall$ *i*. *p i* $\in$ *S*) $\wedge$ (λ*i*. (*l i*, *fst* (*p i*))) $\in$ *P* $\longrightarrow$ (λ*i*. (*l i*, *snd* (*p i*))) $\in$ *Q*)

lemma *CHL-hyperpropI*:
assumes $\bigwedge$*l p*. ($\forall$ *i*. *p i* $\in$ *S*) $\wedge$ (λ*i*. (*l i*, *fst* (*p i*))) $\in$ *P* $\Longrightarrow$ (λ*i*. (*l i*, *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 (λ*i*. (*l i*, *fst* (*p i*))) $\in$ *P*
shows (λ*i*. (*l i*, *snd* (*p i*))) $\in$ *Q*
⟨*proof*⟩

Proposition 3 **theorem** *CHL-hyperproperty*:
hypersat C (*CHL-hyperprop P Q*) $\longleftrightarrow$ *CHL P C Q* (**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

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

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

Definition 18 **definition** *IL* **where**
$IL\ P\ C\ Q \longleftrightarrow Q \subseteq sem\ C\ P$

lemma *equiv-def-incorrectness*:
$IL\ P\ C\ Q \longleftrightarrow (\forall\, l\ \sigma'.\ (l, \sigma') \in Q \longrightarrow (\exists\, \sigma.\ (l, \sigma) \in P \wedge \langle C, \sigma\rangle \rightarrow \sigma'))$
⟨*proof*⟩

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

lemma *IL-hyperpropI*:
assumes $\bigwedge l\ \sigma'.\ (l, \sigma') \in Q \Longrightarrow (\exists\, \sigma.\ (l, \sigma) \in P \wedge (\sigma, \sigma') \in S)$
shows *IL-hyperprop P Q S*
⟨*proof*⟩

Proposition 5 **lemma** *IL-expresses-hyperproperties*:
$IL\ P\ C\ Q \longleftrightarrow IL\text{-}hyperprop\ P\ Q\ (set\text{-}of\text{-}traces\ C)$ (**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

lemma *IL-consequence*:
assumes *IL P C Q*
and $(l, \sigma') \in Q$
shows $\exists\, \sigma.\ (l, \sigma) \in P \wedge single\text{-}sem\ C\ \sigma\ \sigma'$
⟨*proof*⟩

Proposition 6 **theorem** *encoding-IL*:
$IL\ P\ C\ Q \longleftrightarrow (hyper\text{-}hoare\text{-}triple\ (under\text{-}approx\ P)\ C\ (under\text{-}approx\ Q))$ (**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

lemma *entailment-order-reverse-hoare*:
assumes $P \subseteq P'$
shows *entails* $(under\text{-}approx\ P')\ (under\text{-}approx\ P)$
⟨*proof*⟩

definition *incorrectify* **where**
$incorrectify\ p = under\text{-}approx\ \{\ \sigma\ |\sigma.\ p\ \sigma\}$

lemma *incorrectifyI*:
assumes $\bigwedge\sigma.\ p\ \sigma \Longrightarrow \sigma \in S$
shows *incorrectify p S*
⟨*proof*⟩

lemma *incorrectifyE*:
assumes *incorrectify p S*
and $p\ \sigma$
shows $\sigma \in S$
⟨*proof*⟩

lemma *simple-while-incorrectness*:
assumes $\bigwedge n.$ *hyper-hoare-triple* (*incorrectify* (*p n*)) *C* (*incorrectify* (*p* (*Suc n*)))
shows *hyper-hoare-triple* (*incorrectify* (*p 0*)) (*While C*) (*incorrectify* ($\lambda\sigma.\ \exists\, n.\ p$ *n* σ))
⟨*proof*⟩

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

theorem *incorrectness-hyperliveness*:
assumes $\bigwedge l.$ *sat-for-l l Q* $\Longrightarrow$ *sat-for-l l P*
shows *hyperliveness* (*IL-hyperprop P Q*)
⟨*proof*⟩

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

RIL is the old name of k-IL.

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

lemma *RILI*:
assumes $\bigwedge$*states′*. *states′* $\in$ *Q* $\Longrightarrow$ ($\exists$ *states* $\in$ *P*. *k-sem C states states′*)
shows *RIL P C Q*
⟨*proof*⟩

lemma *RILE*:
assumes *RIL P C Q*
and *states′* $\in$ *Q*
shows $\exists$ *states* $\in$ *P*. *k-sem C states states′*
⟨*proof*⟩

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

lemma *RIL-hyperpropI*:

assumes $\bigwedge l\ states'.\ (\lambda i.\ (l\ i,\ states'\ i)) \in Q \Longrightarrow (\exists\, states.\ (\lambda i.\ (l\ i,\ states\ i)) \in P \wedge (\forall\, i.\ (states\ i,\ states'\ i) \in S))$
shows *RIL-hyperprop P Q S*
⟨*proof*⟩

lemma *RIL-hyperpropE*:
assumes *RIL-hyperprop P Q S*
and $(\lambda i.\ (l\ i,\ states'\ i)) \in Q$
shows $\exists\, states.\ (\lambda i.\ (l\ i,\ states\ i)) \in P \wedge (\forall\, i.\ (states\ i,\ states'\ i) \in S)$
⟨*proof*⟩

lemma *useful*:
$states' = (\lambda i.\ ((fst \circ states')\ i,\ (snd \circ states')\ i))$
⟨*proof*⟩

Proposition 17 **theorem** *RIL-expresses-hyperproperties*:
hypersat C (*RIL-hyperprop P Q*) $\longleftrightarrow$ *RIL P C Q* (**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

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-sat-for-l l Q* $\Longrightarrow$ *k-sat-for-l l P*
shows *hyperliveness* (*RIL-hyperprop P Q*)
⟨*proof*⟩

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

lemma *strong-pre-insecI*:
assumes $\bigwedge states.\ states \in P \Longrightarrow (\forall\, i.\ fst\ (states\ i)\ x = from\text{-}nat\ i)$
$\Longrightarrow (\exists\, r.\ \forall\, i.\ ((fst\ (states\ i))(c := r),\ snd\ (states\ i)) \in S)$
and $\bigwedge states.\ (\forall\, i.\ states\ i \in S) \Longrightarrow (\forall\, i.\ fst\ (states\ i)\ x = from\text{-}nat\ i) \Longrightarrow (\forall\, i\ j.\ fst\ (states\ i)\ c = fst\ (states\ j)\ c) \Longrightarrow states \in P$
shows *strong-pre-insec from-nat x c P S*
⟨*proof*⟩

lemma *strong-pre-insecE*:
assumes *strong-pre-insec from-nat x c P S*
and $\bigwedge i.\ states\ i \in S$
and $\bigwedge i.\ fst\ (states\ i)\ x = from\text{-}nat\ i$
and $\bigwedge i\ j.\ fst\ (states\ i)\ c = fst\ (states\ j)\ c$

shows *states* ∈ *P*
⟨*proof*⟩

definition *pre-insec* **where**
pre-insec from-nat x c P S ⟷ (∀ *states* ∈ *P*.
(∀ *i*. *fst* (*states i*) *x* = *from-nat i*)
⟶ (∃ *r*. ∀ *i*. ((*fst* (*states i*))(*c* := *r*), *snd* (*states i*)) ∈ *S*))

lemma *pre-insecI*:
assumes ⋀*states*. *states* ∈ *P* ⟹ (∀ *i*. *fst* (*states i*) *x* = *from-nat i*)
⟹ (∃ *r*. ∀ *i*. ((*fst* (*states i*))(*c* := *r*), *snd* (*states i*)) ∈ *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* ∈ *P*
and ⋀*i*. *fst* (*states i*) *x* = *from-nat i*
shows ∃ *r*. ∀ *i*. ((*fst* (*states i*))(*c* := *r*), *snd* (*states i*)) ∈ *S*
⟨*proof*⟩

definition *post-insec* **where**
post-insec from-nat x c Q S ⟷ (∀ *states* ∈ *Q*. (∀ *i*. *fst* (*states i*) *x* = *from-nat i*)
⟶ (∃ *r*. (∀ *i*. ((*fst* (*states i*))(*c* := *r*), *snd* (*states i*)) ∈ *S*)))

lemma *post-insecE*:
assumes *post-insec from-nat x c Q S*
and *states* ∈ *Q*
and ⋀*i*. *fst* (*states i*) *x* = *from-nat i*
shows ∃ *r*. (∀ *i*. ((*fst* (*states i*))(*c* := *r*), *snd* (*states i*)) ∈ *S*)
⟨*proof*⟩

lemma *post-insecI*:
assumes ⋀*states*. *states* ∈ *Q* ⟹ (∀ *i*. *fst* (*states i*) *x* = *from-nat i*)
⟹ (∃ *r*. (∀ *i*. ((*fst* (*states i*))(*c* := *r*), *snd* (*states i*)) ∈ *S*))
shows *post-insec from-nat x c Q S*
⟨*proof*⟩

lemma *same-pre-post*:

pre-insec from-nat x c Q S ⟷ *post-insec from-nat x c Q S*
⟨*proof*⟩

theorem *can-be-sat*:
fixes *x* :: *'lvar*
assumes ⋀*l l' σ*. (λ*i*. (*l i*, *σ i*)) ∈ *P* ⟷ (λ*i*. (*l' i*, *σ i*)) ∈ *P*
and *injective* (*indexify* :: ((*'a* ⇒ (*'pvar* ⇒ *'pval*)) ⇒ *'lval*))
and *x* ≠ *c*
and *injective from-nat*
shows *sat* (*strong-pre-insec from-nat x c* (*P* :: (*'a* ⇒ (*'lvar* ⇒ *'lval*) × (*'pvar* ⇒ *'pval*)) *set*))
⟨*proof*⟩

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

shows *RIL P C Q* ⟷ ⊨ {*pre-insec from-nat x c P*} *C* {*post-insec from-nat x c Q*} (**is** *?A* ⟷ *?B*)
⟨*proof*⟩

Proposition 8 **theorem** *encoding-RIL*:
fixes *x* :: *'lvar*
assumes ⋀*l l' σ*. (λ*i*. (*l i*, *σ i*)) ∈ *P* ⟷ (λ*i*. (*l' i*, *σ i*)) ∈ *P*
and *injective* (*indexify* :: ((*'a* ⇒ (*'pvar* ⇒ *'pval*)) ⇒ *'lval*))
and *c* ≠ *x*
and *injective from-nat*
and *not-free-var-of* (*P* :: (*'a* ⇒ (*'lvar* ⇒ *'lval*) × (*'pvar* ⇒ *'pval*)) *set*) *x* ∧ *not-free-var-of P c*
and *not-free-var-of Q x* ∧ *not-free-var-of Q c*
shows *RIL P C Q* ⟷ ⊨ {*pre-insec from-nat x c P*} *C* {*post-insec from-nat x c Q*} (**is** *?A* ⟷ *?B*)
⟨*proof*⟩

3.6 Forward Underapproximation (FU)

As employed by Outcome Logic [11]

Definition 20 **definition** *FU* **where**
FU P C Q ⟷ (∀*l*. ∀*σ*. (*l*, *σ*) ∈ *P* ⟶ (∃*σ'*. *single-sem C σ σ'* ∧ (*l*, *σ'*) ∈ *Q*))

lemma *FUI*:
assumes ⋀*σ l*. (*l*, *σ*) ∈ *P* ⟹ (∃*σ'*. *single-sem C σ σ'* ∧ (*l*, *σ'*) ∈ *Q*)
shows *FU P C Q*

⟨*proof*⟩

definition *encode-FU* **where**
encode-FU P S ⟷ *P* ∩ *S* ≠ {}

Proposition 9 **theorem** *encoding-FU*:
FU P C Q ⟷ ⊨ {*encode-FU P*} *C* {*encode-FU Q*} (**is** *?A* ⟷ *?B*)
⟨*proof*⟩

definition *hyperprop-FU* **where**
hyperprop-FU P Q S ⟷ (∀ *l* σ. (*l*, σ) ∈ *P* ⟶ (∃ σ′. (*l*, σ′) ∈ *Q* ∧ (σ, σ′) ∈ *S*))

lemma *hyperprop-FUI*:
assumes ⋀*l* σ. (*l*, σ) ∈ *P* ⟹ (∃ σ′. (*l*, σ′) ∈ *Q* ∧ (σ, σ′) ∈ *S*)
shows *hyperprop-FU P Q S*
⟨*proof*⟩

lemma *hyperprop-FUE*:
assumes *hyperprop-FU P Q S*
and (*l*, σ) ∈ *P*
shows ∃ σ′. (*l*, σ′) ∈ *Q* ∧ (σ, σ′) ∈ *S*
⟨*proof*⟩

theorem *FU-expresses-hyperproperties*:
hypersat C (*hyperprop-FU P Q*) ⟷ *FU P C Q* (**is** *?A* ⟷ *?B*)
⟨*proof*⟩

theorem *hyperliveness-hyperprop-FU*:
assumes ⋀*l*. *sat-for-l l P* ⟹ *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*, σ) |σ *l*. σ *x* = *from-nat* (*0* :: *nat*) ∨ σ *x* = *from-nat 1*}
and *Q* = {(*l*, σ) |σ *l*. σ *x* = *from-nat 1*}
and *C* = *Assume* (λσ. σ *x* = *from-nat 1*)
shows *IL P C Q*
and ¬ *FU P C Q*
⟨*proof*⟩

lemma *FU-does-not-imply-incorrectness*:
assumes *P* = {(*l*, σ) |σ *l*. σ *x* = *from-nat* (*0* :: *nat*) ∨ σ *x* = *from-nat 1*}
and *Q* = {(*l*, σ) |σ *l*. σ *x* = *from-nat 1*}
assumes *injective from-nat*
shows ¬ *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-sem C states states′*)

lemma *RFUI*:
assumes $\bigwedge$*states*. *states* $\in$ *P* $\Longrightarrow$ ($\exists$ *states′* $\in$ *Q*. *k-sem C states states′*)
shows *RFU P C Q*
⟨*proof*⟩

lemma *RFUE*:
assumes *RFU P C Q*
and *states* $\in$ *P*
shows $\exists$ *states′* $\in$ *Q*. *k-sem C states states′*
⟨*proof*⟩

definition *encode-RFU* **where**
encode-RFU from-nat x P S $\longleftrightarrow$ ($\exists$ *states* $\in$ *P*. ($\forall$ *i*. *states i* $\in$ *S* $\wedge$ *fst* (*states i*) *x* = *from-nat i*))

Proposition 11 **theorem** *encode-RFU*:
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*}
(**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

definition *RFU-hyperprop* **where**
RFU-hyperprop P Q S $\longleftrightarrow$ ($\forall$ *l states*. (λ*i*. (*l i*, *states i*)) $\in$ *P*
$\longrightarrow$ ($\exists$ *states′*. (λ*i*. (*l i*, *states′ i*)) $\in$ *Q* $\wedge$ ($\forall$ *i*. (*states i*, *states′ i*) $\in$ *S*)))

lemma *RFU-hyperpropI*:
assumes $\bigwedge$*l states*. (λ*i*. (*l i*, *states i*)) $\in$ *P* $\Longrightarrow$ ($\exists$ *states′*. (λ*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*:
assumes *RFU-hyperprop P Q S*
and (λ*i*. (*l i*, *states i*)) $\in$ *P*
shows $\exists$ *states′*. (λ*i*. (*l i*, *states′ i*)) $\in$ *Q* $\wedge$ ($\forall$ *i*. (*states i*, *states′ i*) $\in$ *S*)
⟨*proof*⟩

Proposition 10 **theorem** *RFU-captures-hyperproperties*:
hypersat C (*RFU-hyperprop P Q*) $\longleftrightarrow$ *RFU P C Q* (**is** *?A* $\longleftrightarrow$ *?B*)

⟨*proof*⟩

theorem *hyperliveness-encode-RFU*:
assumes $\bigwedge l.$ *k-sat-for-l l P* $\Longrightarrow$ *k-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 $\longleftrightarrow$ $(\forall (\sigma 1, \sigma 2) \in P.\ \forall \sigma 1'.$ *k-sem C σ1 σ1′* $\longrightarrow$ $(\exists \sigma 2'.$ *k-sem C σ2 σ2′* $\wedge$ $(\sigma 1', \sigma 2') \in Q))$

lemma *RUE-I*:
assumes $\bigwedge \sigma 1\ \sigma 2\ \sigma 1'.$ $(\sigma 1, \sigma 2) \in P \Longrightarrow$ *k-sem C σ1 σ1′* $\Longrightarrow$ $(\exists \sigma 2'.$ *k-sem C σ2 σ2′* $\wedge$ $(\sigma 1', \sigma 2') \in Q)$
shows *RUE P C Q*
⟨*proof*⟩

lemma *RUE-E*:
assumes *RUE P C Q*
and $(\sigma 1, \sigma 2) \in P$
and *k-sem C σ1 σ1′*
shows $\exists \sigma 2'.$ *k-sem C σ2 σ2′* $\wedge$ $(\sigma 1', \sigma 2') \in Q$
⟨*proof*⟩

Hyperproperty **definition** *hyperprop-RUE* **where**
hyperprop-RUE P Q S $\longleftrightarrow$ $(\forall$ *l1 l2 σ1 σ2 σ1′*. $(\lambda i.\ (l1\ i, \sigma 1\ i), \lambda k.\ (l2\ k, \sigma 2\ k)) \in P \wedge$
$(\forall i.\ (\sigma 1\ i, \sigma 1'\ i) \in S) \longrightarrow (\exists \sigma 2'.\ (\forall k.\ (\sigma 2\ k, \sigma 2'\ k) \in S) \wedge (\lambda i.\ (l1\ i, \sigma 1'\ i), \lambda k.\ (l2\ k, \sigma 2'\ k)) \in Q))$

lemma *hyperprop-RUE-I*:
assumes $\bigwedge l1\ l2\ \sigma 1\ \sigma 2\ \sigma 1'.$ $(\lambda i.\ (l1\ i, \sigma 1\ i), \lambda k.\ (l2\ k, \sigma 2\ k)) \in P \Longrightarrow$
$(\forall i.\ (\sigma 1\ i, \sigma 1'\ i) \in S) \Longrightarrow (\exists \sigma 2'.\ (\forall k.\ (\sigma 2\ k, \sigma 2'\ k) \in S) \wedge (\lambda i.\ (l1\ i, \sigma 1'\ i), \lambda k.\ (l2\ k, \sigma 2'\ k)) \in Q)$
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-RUE P Q*) (**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

definition *is-type* **where**
is-type type fn x t S σ $\longleftrightarrow$ ($\forall$ *i*. σ *i* $\in$ *S* $\wedge$ *fst* (σ *i*) *t* = *type* $\wedge$ *fst* (σ *i*) *x* = *fn i*)

lemma *is-typeI*:
assumes $\bigwedge$*i*. σ *i* $\in$ *S*
and $\bigwedge$*i*. *fst* (σ *i*) *t* = *type*
and $\bigwedge$*i*. *fst* (σ *i*) *x* = *fn i*
shows *is-type type fn x t S* σ
⟨*proof*⟩

lemma *is-type-E*:
assumes *is-type type fn x t S* σ
shows σ *i* $\in$ *S* $\wedge$ *fst* (σ *i*) *t* = *type* $\wedge$ *fst* (σ *i*) *x* = *fn i*
⟨*proof*⟩

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

lemma *encode-RUE-1-I*:
assumes $\bigwedge$*k*. $\exists$ σ $\in$ *S*. *fst* σ *x* = *fn2 k* $\wedge$ *fst* σ *t* = *fn 2*
and $\bigwedge$$\sigma$ σ'. *is-type* (*fn 1*) *fn1 x t S* σ $\wedge$ *is-type* (*fn 2*) *fn2 x t S* σ'
$\Longrightarrow$ (σ, σ') $\in$ *P*
shows *encode-RUE-1 fn fn1 fn2 x t P S*
⟨*proof*⟩

lemma *encode-RUE-1-E1*:
assumes *encode-RUE-1 fn fn1 fn2 x t P S*
shows $\exists$ σ $\in$ *S*. *fst* σ *x* = *fn2 k* $\wedge$ *fst* σ *t* = *fn 2*
⟨*proof*⟩

lemma *encode-RUE-1-E2*:
assumes *encode-RUE-1 fn fn1 fn2 x t P S*
and *is-type* (*fn 1*) *fn1 x t S* σ
and *is-type* (*fn 2*) *fn2 x t S* σ'
shows (σ, σ') $\in$ *P*
⟨*proof*⟩

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

lemma *encode-RUE-2I*:
assumes $\bigwedge\sigma$. *is-type* (*fn 1*) *fn1 x t S* $\sigma \Longrightarrow (\exists\,\sigma'$. *is-type* (*fn 2*) *fn2 x t S* $\sigma' \wedge$ $(\sigma, \sigma') \in Q$)
shows *encode-RUE-2 fn fn1 fn2 x t Q S*
⟨*proof*⟩

lemma *encode-RUE-2-E*:
assumes *encode-RUE-2 fn fn1 fn2 x t Q S*
and *is-type* (*fn 1*) *fn1 x t S* σ
shows $\exists\,\sigma'$. *is-type* (*fn 2*) *fn2 x t S* $\sigma' \wedge (\sigma, \sigma') \in Q$
⟨*proof*⟩

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 $\longleftrightarrow (\forall\, i.$ *snd* $(a\ i) =$ *snd* $(b\ i) \wedge$ *differ-only-by-set vars* (*fst* $(a\ i))$ (*fst* $(b\ i)))$

lemma *differ-only-by-lsetI*:
assumes $\bigwedge i.$ *snd* $(a\ i) =$ *snd* $(b\ i)$
and $\bigwedge i.$ *differ-only-by-set vars* (*fst* $(a\ i))$ (*fst* $(b\ i))$
shows *differ-only-by-lset vars a b*
⟨*proof*⟩

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

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

Proposition 13 **theorem** *encoding-RUE*:
assumes *injective fn* $\wedge$ *injective fn1* $\wedge$ *injective fn2*
and $t \neq x$

and *injective* (*fn* :: *nat* $\Rightarrow$ *'a*)
and *injective fn1*
and *injective fn2*

and *not-in-free-vars-double* $\{x, t\}$ *P*
and *not-in-free-vars-double* $\{x, t\}$ *Q*

shows *RUE P C Q* $\longleftrightarrow$ $\models$ {*encode-RUE-1 fn fn1 fn2 x t P*} *C* {*encode-RUE-2 fn fn1 fn2 x t Q*}
(**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

3.9 Program Refinement

lemma *sem-assign-single*:
sem (*Assign x e*) {(*l*, σ)} = {(*l*, $\sigma(x := e\ \sigma)$)} (**is** *?A* = *?B*)
⟨*proof*⟩

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 (σ, σ') $\in$ *set-of-traces C*
shows ($\sigma(x := v)$, $\sigma'(x := v)$) $\in$ *set-of-traces C*
⟨*proof*⟩

lemma *not-free-in-sem-same-val*:
assumes *not-free-var-stmt x C*
and *single-sem C* $\sigma\ \sigma'$
shows $\sigma\ x = \sigma'\ x$
⟨*proof*⟩

lemma *not-free-in-sem-equiv*:
assumes *not-free-var-stmt x C*
and *single-sem C* $\sigma\ \sigma'$
shows *single-sem C* ($\sigma(x := v)$) ($\sigma'(x := v)$)
⟨*proof*⟩

Example 3 **lemma** *rewrite-if-commute*:
assumes $\models$ { *P* } *If C1 C2* {*Q*}
shows $\models$ { *P* } *If C2 C1* {*Q*}
⟨*proof*⟩

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

and *P* = (λS. *card S* = *1*)
and *Q* = (λS.

$\forall \varphi{\in}S.\ snd\ \varphi\ x = a \longrightarrow (fst\ \varphi, (snd\ \varphi)(x := b)) \in S)$
and *not-free-var-stmt x C1*
and *not-free-var-stmt x C2*
shows *refinement C1 C2* $\longleftrightarrow$
$\models$ { *P* } *If* (*Seq* (*Assign* (*x* :: *'pvar*) (λ-. *a*)) *C1*) (*Seq* (*Assign x* (λ-. *b*)) *C2*) { *Q* }
(**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

Necessary Preconditions **definition** *NC* **where**
NC P C Q $\longleftrightarrow$ $(\forall \sigma\ \sigma'\ l.\ (l, \sigma') \in Q \wedge (\langle C, \sigma\rangle \to \sigma') \longrightarrow (l, \sigma) \in P)$

lemma *NC-I*:
assumes $\bigwedge \sigma\ \sigma'\ l.\ (l, \sigma') \in Q \wedge (\langle C, \sigma\rangle \to \sigma') \Longrightarrow (l, \sigma) \in P$
shows *NC P C Q*
⟨*proof*⟩

definition *backwards-sem* **where**
backwards-sem C S' = { $(l, \sigma)\ |l\ \sigma\ \sigma'.\ (l, \sigma') \in S' \wedge \langle C, \sigma\rangle \to \sigma'$}

lemma *equiv-def-NC*:
NC P C Q $\longleftrightarrow$ *backwards-sem C Q* $\subseteq$ *P* (**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

lemma *equiv-def-FU*:
FU P C Q $\longleftrightarrow$ *P* $\subseteq$ *backwards-sem C Q* (**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

lemma *encoding-NC-in-HL-1*:
NC P C Q $\longleftrightarrow \models$ { $(\lambda S.\ S = -P)$ } *C* { $(\lambda S.\ S \cap Q = \{\}\ )$ } (**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

end
theory *Loops*
imports *Logic HOL.Wellfounded Expressivity*
begin

4 Rules for Loops

definition *lnot* **where**
lnot b σ = $(\neg b\ \sigma)$

definition *if-then-else* **where**
if-then-else b C1 C2 = *If* (*Assume b*;; *C1*) (*Assume* (*lnot b*);; *C2*)

definition *low-exp* **where**
low-exp e S = $(\forall \varphi\ \varphi'.\ \varphi \in S \wedge \varphi' \in S \longrightarrow (e\ (snd\ \varphi) = e\ (snd\ \varphi')))$

lemma *low-exp-lnot*:
low-exp b S $\longleftrightarrow$ *low-exp* (*lnot b*) *S*
⟨*proof*⟩

definition *holds-forall* **where**
holds-forall b S $\longleftrightarrow$ ($\forall \varphi \in S$. *b* (*snd* φ))

lemma *holds-forallI*:
assumes $\bigwedge \varphi$. $\varphi \in S \Longrightarrow$ *b* (*snd* φ)
shows *holds-forall b S*
⟨*proof*⟩

lemma *low-exp-two-cases*:
assumes *low-exp b S*
shows *holds-forall b S* $\vee$ *holds-forall* (*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* = *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* $\Longrightarrow$ *sem* (*if-then-else b C1 C2*) *S* = *sem C1 S*
and *holds-forall* (*lnot b*) *S* $\Longrightarrow$ *sem* (*if-then-else b C1 C2*) *S* = *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*} *if-then-else b C1 C2* {*Q*}
⟨*proof*⟩

theorem *if-synchronized*:
assumes $\models$ {*conj P* (*holds-forall b*)} *C1* {*Q*}
and $\models$ {*conj P* (*holds-forall* (*lnot b*))} *C2* {*Q*}
shows $\models$ {*conj P* (*low-exp b*)} *if-then-else b C1 C2* {*Q*}

⟨*proof*⟩

definition *while-cond* **where**
while-cond b C = *While* (*Assume b*;; *C*);; *Assume* (*lnot b*)

lemma *while-synchronized-rec*:
assumes ⋀*n*. ⊨ {*conj* (*I n*) (*holds-forall b*)} *Assume b*;; *C* {*conj* (*I* (*Suc n*)) (*low-exp b*)}
and *conj* (*I 0*) (*low-exp b*) *S*
shows *conj* (*I n*) (*low-exp b*) (*iterate-sem n* (*Assume b*;; *C*) *S*) ∨ *holds-forall* (*lnot b*) (*iterate-sem n* (*Assume b*;; *C*) *S*)
⟨*proof*⟩

lemma *false-then-empty-later*:
assumes *holds-forall* (*lnot b*) (*iterate-sem n* (*Assume b*;; *C*) *S*)
and *m* > *n*
shows *iterate-sem m* (*Assume b*;; *C*) *S* = {}
⟨*proof*⟩

lemma *split-union-triple*:
(⋃(*m*::*nat*). *f m*) = (⋃*m*∈{*m* |*m*. *m* < *n*}. *f m*) ∪ *f n* ∪ (⋃*m*∈{*m* |*m*. *m* > *n*}. *f m*) (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *sem-union-swap*:
sem C (⋃*x*∈*S*. *f x*) = (⋃*x*∈*S*. *sem C* (*f x*)) (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *while-synchronized-case-1*:
assumes ⋀*m*. *m* < *n* ⟹ *holds-forall b* (*iterate-sem m* (*Assume b*;; *C*) *S*)
and *holds-forall* (*lnot b*) (*iterate-sem n* (*Assume b*;; *C*) *S*)
and ⋀*n*. ⊨ {*conj* (*I n*) (*holds-forall b*)} *Assume b*;; *C* {*conj* (*I* (*Suc n*)) (*low-exp b*)}
and *conj* (*I 0*) (*low-exp b*) *S*
shows *sem* (*while-cond b C*) *S* = *iterate-sem n* (*Assume b*;; *C*) *S*
⟨*proof*⟩

lemma *while-synchronized-case-2*:
assumes ⋀*m*. *holds-forall b* (*iterate-sem m* (*Assume b*;; *C*) *S*)
and ⋀*n*. ⊨ {*conj* (*I n*) (*holds-forall b*)} *Assume b*;; *C* {*conj* (*I* (*Suc n*)) (*low-exp b*)}
and *conj* (*I 0*) (*low-exp b*) *S*
shows *sem* (*while-cond b C*) *S* = {}
⟨*proof*⟩

definition *emp* **where**
emp S $\longleftrightarrow$ *S* = {}

lemma *holds-forall-empty*:
holds-forall b {}
⟨*proof*⟩

definition *exists* **where**
exists I S $\longleftrightarrow$ (∃ *n*. *I n S*)

theorem *while-synchronized*:
assumes ⋀*n*. ⊨ {*conj* (*I n*) (*holds-forall b*)} *C* {*conj* (*I* (*Suc n*)) (*low-exp b*)}
shows ⊨ {*conj* (*I* 0) (*low-exp b*)} *while-cond b C* {*conj* (*disj* (*exists I*) *emp*) (*holds-forall* (*lnot b*))}
⟨*proof*⟩

lemma *WhileSync-simpler*:
assumes ⊨ {*conj I* (*holds-forall b*)} *C* {*conj I* (*low-exp b*)}
shows ⊨ {*conj I* (*low-exp b*)} *while-cond b C* {*conj* (*disj I emp*) (*holds-forall* (*lnot b*))}
⟨*proof*⟩

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

definition *filter-exp* **where**
filter-exp b S = *Set.filter* (*b* ∘ *snd*) *S*

lemma *filter-exp-union*:
filter-exp b (*S1* ∪ *S2*) = *filter-exp b S1* ∪ *filter-exp b S2* (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *filter-exp-union-general*:
filter-exp b (⋃*x*. *f x*) = (⋃*x*. *filter-exp b* (*f x*)) (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *filter-exp-contradict*:
filter-exp b (*filter-exp* (*lnot b*) *S*) = {}
⟨*proof*⟩

lemma *filter-exp-same*:
filter-exp b (*filter-exp b S*) = *filter-exp b S* (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *if-then-sem*:
sem (*if-then b C*) *S* = *sem C* (*filter-exp b S*) ∪ *filter-exp* (*lnot b*) *S*
⟨*proof*⟩

fun *union-up-to-n* **where**
  union-up-to-n C S 0 = iterate-sem 0 C S
| *union-up-to-n C S (Suc n) = iterate-sem (Suc n) C S ∪ union-up-to-n C S n*

lemma *union-up-to-increasing*:
  assumes *m ≤ n*
  shows *union-up-to-n C S m ⊆ union-up-to-n C S n*
  ⟨*proof*⟩

lemma *union-union-up-to-n-equiv-aux*:
  union-up-to-n C S n ⊆ (⋃ m. iterate-sem m C S)
⟨*proof*⟩

lemma *union-union-up-to-n-equiv*:
  (⋃ n. union-up-to-n C S n) = (⋃ n. iterate-sem n C S) (**is** *?A = ?B*)
⟨*proof*⟩

lemma *filter-exp-union-itself*:
  filter-exp b S ∪ S = S
  ⟨*proof*⟩

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

lemma *sem-while-with-if*:
  sem (while-cond b C) S = filter-exp (lnot b) (⋃ n. iterate-sem n (if-then b C) S)
⟨*proof*⟩

lemma *iterate-sem-assume-increasing*:
  filter-exp (lnot b) (iterate-sem n (if-then b C) S) ⊆ filter-exp (lnot b) (iterate-sem (Suc n) (if-then b C) S)
  ⟨*proof*⟩

lemma *iterate-sem-assume-increasing-union-up-to*:
  filter-exp (lnot b) (iterate-sem n (if-then b C) S) = filter-exp (lnot b) (union-up-to-n (if-then b C) S n)
⟨*proof*⟩

definition *ascending* :: *(nat ⇒ 'b set) ⇒ bool* **where**
  ascending S ⟷ (∀ n m. n ≤ m ⟶ S n ⊆ S m)

lemma *ascendingI-direct*:
  assumes *⋀n m. n ≤ m ⟹ S n ⊆ S m*

shows *ascending S*
⟨*proof*⟩

lemma *ascendingI*:
assumes ⋀*n*. *S n* ⊆ *S* (*Suc n*)
shows *ascending S*
⟨*proof*⟩

definition *upwards-closed* **where**
upwards-closed P P-inf ⟷ (∀ *S*. *ascending S* ∧ (∀ *n*. *P n* (*S n*)) ⟶ *P-inf* (⋃ *n*. *S n*))

lemma *upwards-closedI*:
assumes ⋀*S*. *ascending S* ⟹ (∀ *n*. *P n* (*S n*)) ⟹ *P-inf* (⋃ *n*. *S n*)
shows *upwards-closed P P-inf*
⟨*proof*⟩

lemma *upwards-closedE*:
assumes *upwards-closed P P-inf*
and *ascending S*
and ⋀*n*. *P n* (*S n*)
shows *P-inf* (⋃ *n*. *S n*)
⟨*proof*⟩

lemma *ascending-iterate-filter*:
ascending (λ*n*. *filter-exp* (*lnot b*) (*union-up-to-n* (*if-then b C*) *S n*))
⟨*proof*⟩

theorem *while-general*:
assumes ⋀*n*. ⊨ {*P n*} *if-then b C* {*P* (*Suc n*)}
and ⋀*n*. ⊨ {*P n*} *Assume* (*lnot b*) {*Q n*}
and *upwards-closed Q Q-inf*
shows ⊨ {*P 0*} *while-cond b C* {*conj Q-inf* (*holds-forall* (*lnot b*))}
⟨*proof*⟩

definition *while-loop-assertion-n* **where**
while-loop-assertion-n C S0 n S ⟷ (*S* = *union-up-to-n C S0 n*)

definition *while-loop-assertion-inf* **where**
while-loop-assertion-inf C S0 S ⟷ (*S* = (⋃ *n*. *union-up-to-n C S0 n*))

lemma *while-loop-assertion-upwards-closed*:
upwards-closed (*while-loop-assertion-n C S0*) (*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 $= \{x\ |x.\ \exists n.\ \forall m.\ m \geq n \longrightarrow (x \in S\ m)\}$

lemma *in-limit-sets*:
$x \in$ *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-inf $\longleftrightarrow$ $(\forall S.$ *converges-sets S* $\wedge\ (\forall n.\ P\ n\ (S\ n)) \longrightarrow$ *P-inf* (*limit-sets S*))

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

assumes *t-closed P P-inf*
shows *upwards-closed P P-inf*
⟨*proof*⟩

definition *downwards-closed* **where**
downwards-closed P-inf ⟷ (∀ *S S'*. *S* ⊆ *S'* ∧ *P-inf S'* ⟶ *P-inf S*)

definition *d-closed* **where**
d-closed P P-inf ⟷ *t-closed P P-inf* ∧ *downwards-closed P-inf*

lemma *converges-to-merged*:
assumes ⋀*x*. *x* ∈ *S-inf* ⟹ ∃ *n*. ∀ *m*. *m* ≥ *n* ⟶ (*x* ∈ *S* (*m*::*nat*))
and ⋀*x*. *x* ∉ *S-inf* ⟹ ∃ *n*. ∀ *m*. *m* ≥ *n* ⟶ (*x* ∉ *S m*)
shows *converges-sets S* ∧ *limit-sets S* = *S-inf*
⟨*proof*⟩

lemma *ascending-union-up*:
ascending (λ*n*. *union-up-to-n C S n*)
⟨*proof*⟩

lemma *converges-union*:
converges-sets (λ*n*. *union-up-to-n C S n*) ∧ *limit-sets* (λ*n*. *union-up-to-n C S n*)
= (⋃ *n*. *union-up-to-n C S n*)
⟨*proof*⟩

theorem *while-d*:
assumes ⋀*n*. ⊨ {*P n*} *if-then b C* {*P* (*Suc n*)}
and *upwards-closed P P-inf*
and ⋀*n*. *downwards-closed* (*P n*) — Satisfied by hyper-assertions that do not existentially quantify over states
shows ⊨ {*P 0*} *while-cond b C* {*conj P-inf* (*holds-forall* (*lnot b*))}
⟨*proof*⟩

lemma *in-union-up-to*:
x ∈ *union-up-to-n C S n* ⟷ (∃ *m*. *m* ≤ *n* ∧ *x* ∈ *iterate-sem m C S*)
⟨*proof*⟩

theorem *rule-while-terminates-strong*:
assumes ⋀*n*. *n* < *m* ⟹ ⊨ {*P n*} *if-then b C* {*P* (*Suc n*)}
and ⋀*S*. *P m S* ⟶ *holds-forall* (*lnot b*) *S*
shows ⊨ {*P 0*} *while-cond b C* {*P m*}
⟨*proof*⟩

lemma *false-state-in-if-then*:
assumes $\varphi \in S$
and $\neg\ b\ (snd\ \varphi)$
shows $\varphi \in sem\ (if\text{-}then\ b\ C)\ S$
⟨*proof*⟩

lemma *false-state-in-while-cond-aux*:
assumes $\varphi \in S$
and $\neg\ b\ (snd\ \varphi)$
shows $\varphi \in iterate\text{-}sem\ n\ (if\text{-}then\ b\ C)\ S$
⟨*proof*⟩

lemma *false-state-in-while-cond*:
assumes $\varphi \in S$
and $\neg\ b\ (snd\ \varphi)$
shows $\varphi \in sem\ (while\text{-}cond\ b\ C)\ S$
⟨*proof*⟩

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

lemma *sem-while-cond-union-up-to*:
$sem\ (while\text{-}cond\ b\ C)\ S = filter\text{-}exp\ (lnot\ b)\ (\bigcup n.\ union\text{-}up\text{-}to\text{-}n\ (if\text{-}then\ b\ C)\ S\ n)$
⟨*proof*⟩

lemma *iterate-sem-sum*:
$iterate\text{-}sem\ n\ C\ (iterate\text{-}sem\ m\ C\ S) = iterate\text{-}sem\ (n + m)\ C\ S$
⟨*proof*⟩

lemma *unroll-while-sem*:
$sem\ (while\text{-}cond\ b\ C)\ (iterate\text{-}sem\ n\ (if\text{-}then\ b\ C)\ S) = sem\ (while\text{-}cond\ b\ C)\ S$
⟨*proof*⟩

theorem *while-unroll*:
assumes $\bigwedge n.\ n < m \Longrightarrow \models \{P\ n\}\ if\text{-}then\ b\ C\ \{P\ (Suc\ n)\}$
and $\models \{P\ m\}\ while\text{-}cond\ b\ C\ \{Q\}$
shows $\models \{P\ 0\}\ while\text{-}cond\ b\ C\ \{Q\}$
⟨*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 $\models \{$ *natural-partition* $I\ \}$ *Assume* $(lnot\ b)\ \{\ Q\ \}$
shows $\models \{I\ 0\}$ *while-cond* $b\ C\ \{\ Q\ \}$
⟨*proof*⟩

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

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

end

5 Compositionality Rules

theory *Compositionality*
imports *Logic Expressivity Loops*
begin

In this file, we prove the soundness of all compositionality rules presented in Appendix D (figure 11).

definition *in-set* **where**
in-set $\varphi\ S \longleftrightarrow \varphi \in S$

5.1 Linking rule

proposition *rule-linking*:
assumes $\bigwedge \varphi 1\ (\varphi 2 :: ('a, 'b, 'c, 'd)\ state).\ fst\ \varphi 1 = fst\ \varphi 2 \wedge (\models \{\ (in\text{-}set\ \varphi 1 :: (('a, 'b, 'c, 'd)\ state)\ hyperassertion)\ \}\ C\ \{\ in\text{-}set\ \varphi 2\ \})$
$\Longrightarrow (\models \{\ (P\ \varphi 1 :: (('a, 'b, 'c, 'd)\ state)\ hyperassertion)\ \}\ C\ \{\ Q\ \varphi 2\ \})$
shows $\models \{\ ((\lambda S.\ \forall \varphi 1 \in S.\ P\ \varphi 1\ S) :: (('a, 'b, 'c, 'd)\ state)\ hyperassertion)\ \}\ C\ \{\ (\lambda S.\ \forall \varphi 2 \in S.\ Q\ \varphi 2\ S)\ \}$
⟨*proof*⟩

lemma *rule-linking-alt*:
assumes $\bigwedge l\ \sigma\ \sigma'.\ single\text{-}sem\ C\ \sigma\ \sigma' \Longrightarrow (\models \{\ P\ (l, \sigma)\ \}\ C\ \{\ Q\ (l, \sigma')\ \})$
shows $\models \{\ (\lambda S.\ \forall \omega \in S.\ P\ \omega\ S)\ \}\ C\ \{\ (\lambda S.\ \forall \omega' \in S.\ Q\ \omega'\ S)\ \}$
⟨*proof*⟩

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

lemma *differ-only-by-psetI*:
assumes $\bigwedge i.\ fst\ (a\ i) = fst\ (b\ i) \wedge$ *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 $\longleftrightarrow$ $(\forall \varphi 1\ \varphi 2.$ *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* $(written\text{-}vars\ C)\ b$

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 $\longleftrightarrow$ $(\forall x.\ x \notin vars \longrightarrow l1\ x = l2\ x)$

lemma *equal-outside-setI*:
assumes $\bigwedge x.\ x \notin vars \Longrightarrow l1\ x = l2\ x$

shows *equal-outside-set vars l1 l2*
⟨*proof*⟩

lemma *equal-outside-setE*:
assumes *equal-outside-set vars l1 l2*
and $x \notin vars$
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' $\longleftrightarrow$ $(\forall \omega \in S.\ \exists \omega' \in S'.$ *snd* ω = *snd* $\omega' \wedge$ *equal-outside-set vars* (*fst* ω) (*fst* ω'))

lemma *subset-mod-updatesI*:
assumes $\bigwedge \omega.\ \omega \in S \Longrightarrow (\exists \omega' \in S'.$ *snd* ω = *snd* $\omega' \wedge$ *equal-outside-set vars* (*fst* ω) (*fst* ω'))
shows *subset-mod-updates vars S S'*
⟨*proof*⟩

lemma *subset-mod-updatesE*:
assumes *subset-mod-updates vars S S'*
and $\omega \in S$
shows $\exists \omega' \in S'.$ *snd* ω = *snd* $\omega' \wedge$ *equal-outside-set vars* (*fst* ω) (*fst* ω')
⟨*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 $\bigwedge \omega.\ \omega \in S \Longrightarrow (\exists \omega' \in S'.$ *snd* ω = *snd* $\omega' \wedge$ *equal-outside-set vars* (*fst* ω) (*fst* ω'))
and $\bigwedge \omega'.\ \omega' \in S' \Longrightarrow (\exists \omega \in S.$ *snd* ω = *snd* $\omega' \wedge$ *equal-outside-set vars* (*fst* ω') (*fst* ω))
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*
⟨*proof*⟩

lemma *subset-mod-updates-trans*:
assumes *subset-mod-updates vars S1 S2*
and *subset-mod-updates vars S2 S3*
shows *subset-mod-updates vars S1 S3*
⟨*proof*⟩

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*
⟨*proof*⟩

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

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

type-synonym (*'a*, *'b*, *'c*, *'d*) *chyperassertion* = ((*'a* ⇒ *'b*) × (*'c* ⇒ *'d*)) *set* ⇒ *bool*

definition *invariant-on-updates* :: *'a set* ⇒ (*'a*, *'b*, *'c*, *'d*) *chyperassertion* ⇒ *bool*
where
invariant-on-updates vars P ⟷ (∀ *S S'*. *same-mod-updates vars S S'* ⟶ (*P S* ⟷ *P S'*))

lemma *invariant-on-updatesI*:
assumes ⋀*S S'*. *same-mod-updates vars S S'* ⟹ *P S* ⟹ *P S'*
shows *invariant-on-updates vars P*
⟨*proof*⟩

definition *entails-with-updates* :: *'a set* ⇒ (*'a*, *'b*, *'c*, *'d*) *chyperassertion* ⇒ (*'a*, *'b*, *'c*, *'d*) *chyperassertion* ⇒ *bool*
where

entails-with-updates vars P Q $\longleftrightarrow$ ($\forall$ *S*. *P S* $\longrightarrow$ ($\exists$ *S'*. *same-mod-updates vars S S'* $\wedge$ *Q S'*))

lemma *entails-with-updatesI*:
assumes $\bigwedge$*S*. *P S* $\Longrightarrow$ ($\exists$ *S'*. *same-mod-updates vars S S'* $\wedge$ *Q S'*)
shows *entails-with-updates vars P Q*
⟨*proof*⟩

lemma *entails-with-updatesE*:
assumes *entails-with-updates vars P Q*
and *P S*
shows $\exists$ *S'*. *same-mod-updates vars S S'* $\wedge$ *Q S'*
⟨*proof*⟩

proposition *rule-LUpdate*:
assumes $\models$ {*P'*} *C* {*Q*}
and *entails-with-updates vars P P'*
and *invariant-on-updates vars Q*
shows $\models$ {*P*} *C* {*Q*}
⟨*proof*⟩

5.4 Filters

lemma *filter-prop-commute-aux*:
assumes $\bigwedge\omega$ ω'. *fst* ω = *fst* ω' $\Longrightarrow$ (*f* ω $\longleftrightarrow$ *f* ω')
shows *Set.filter f* (*sem C S*) = *sem C* (*Set.filter f S*) (**is** *?A* = *?B*)
⟨*proof*⟩

definition *commute-with-sem* **where**
commute-with-sem f $\longleftrightarrow$ ($\forall$ *S C*. *f* (*sem C S*) = *sem C* (*f S*))

lemma *commute-with-semI*:
assumes $\bigwedge$(*S* :: (('*a* $\Rightarrow$ '*b*) $\times$ ('*c* $\Rightarrow$ '*d*)) *set*) *C*. *f* (*sem C S*) = *sem C* (*f S*)
shows *commute-with-sem f*
⟨*proof*⟩

lemma *filter-prop-commute*:
assumes $\bigwedge\omega$ ω'. *fst* ω = *fst* ω' $\Longrightarrow$ (*f* ω $\longleftrightarrow$ *f* ω')
shows *commute-with-sem* (*Set.filter f*)
⟨*proof*⟩

lemma *rule-apply*:
assumes $\models$ {*P*} *C* {*Q*}
and *commute-with-sem f*
shows $\models$ {*P* $\circ$ *f*} *C* {*Q* $\circ$ *f*}
⟨*proof*⟩

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-pexp* **where**
not-in-free-pvars-pexp vars b $\longleftrightarrow$ ($\forall \varphi 1\ \varphi 2.$ *differ-only-by-set vars* $\varphi 1\ \varphi 2 \longrightarrow$ (*b* $\varphi 1 \longleftrightarrow$ *b* $\varphi 2$))

lemma *single-sem-differ-by-written-vars*:
assumes *single-sem C* $\varphi\ \varphi'$
shows *differ-only-by-set* (*written-vars C*) $\varphi\ \varphi'$
⟨*proof*⟩

lemma *single-sem-not-free-vars*:
assumes *not-in-free-pvars-pexp* (*written-vars C*) *b*
and *single-sem C* $\varphi\ \varphi'$
shows *b* $\varphi \longleftrightarrow$ *b* φ'
⟨*proof*⟩

proposition *rule-PFilter*:
assumes $\models \{P\}\ C\ \{Q\}$
and *not-in-free-pvars-pexp* (*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 (λ-. *False*) *C Q*
⟨*proof*⟩

proposition *rule-True*:
hyper-hoare-triple P C (λ-. *True*)
⟨*proof*⟩

lemma *sem-inter*:
sem C (*S1* $\cap$ *S2*) $\subseteq$ *sem C S1* $\cap$ *sem C S2*

⟨*proof*⟩

proposition *rule-Union*:
assumes $\models \{P\}\ C\ \{Q\}$
and *hyper-hoare-triple* $P'\ C\ Q'$
shows *hyper-hoare-triple* $(\mathit{join}\ P\ P')\ C\ (\mathit{join}\ Q\ Q')$
⟨*proof*⟩

proposition *rule-IndexedUnion*:
assumes $\bigwedge x.\ \models \{P\ x\}\ C\ \{Q\ x\}$
shows *hyper-hoare-triple* $(\mathit{general\text{-}join}\ P)\ C\ (\mathit{general\text{-}join}\ Q)$
⟨*proof*⟩

proposition *rule-And*:
assumes $\models \{P\}\ C\ \{Q\}$
and *hyper-hoare-triple* $P'\ C\ Q'$
shows *hyper-hoare-triple* $(\mathit{conj}\ P\ P')\ C\ (\mathit{conj}\ Q\ Q')$
⟨*proof*⟩

lemma *rule-Forall*:
assumes $\bigwedge x.\ \models \{P\ x\}\ C\ \{Q\ x\}$
shows *hyper-hoare-triple* $(\mathit{forall}\ P)\ C\ (\mathit{forall}\ Q)$
⟨*proof*⟩

lemma *rule-Or*:
assumes $\models \{P\}\ C\ \{Q\}$
and $\models \{P'\}\ C\ \{Q'\}$
shows *hyper-hoare-triple* $(\mathit{disj}\ P\ P')\ C\ (\mathit{disj}\ Q\ Q')$
⟨*proof*⟩

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* $(\mathit{If}\ C1\ C2)\ Q$
⟨*proof*⟩

Simplifying the rule

definition *stable-by-infinite-union* :: *'a hyperassertion* ⇒ *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 $I\ (\bigcup S \in F.\ S)$
⟨*proof*⟩

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

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

corollary *simpler-rule-while*:
assumes *hyper-hoare-triple* $I\ C\ I$
and *stable-by-infinite-union* I
shows *hyper-hoare-triple* $I\ (While\ C)\ I$
⟨*proof*⟩

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

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

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

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

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

lemma *weaker-linking-rule*:
assumes $\bigwedge l\ \sigma\ \sigma'.\ \models \{\ (\lambda S.\ (l, \sigma) \in S)\ \}\ C\ \{\ (\lambda S.\ (l, \sigma') \in S)\ \} \Longrightarrow (\ \models \{\ P\ (l, \sigma)\ \}\ C\ \{\ Q\ (l, \sigma')\ \}\ )$
shows $\models \{\ (\lambda S.\ \forall \omega \in S.\ P\ \omega\ S)\ \}\ C\ \{\ (\lambda S.\ \forall \omega' \in S.\ Q\ \omega'\ S)\ \}$
⟨*proof*⟩

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

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

⟨*proof*⟩

lemma *general-unionE*:
assumes *general-union P S*
obtains *F* **where** $S = \mathit{Union}\ F\ \bigwedge S'.\ S' \in F \Longrightarrow P\ S'$
⟨*proof*⟩

proposition *rule-BigUnion*:
fixes $P :: (((\prime a \Rightarrow \prime b) \times (\prime c \Rightarrow \prime d))\ \mathit{set} \Rightarrow \mathit{bool})$
assumes $\models \{P\}\ C\ \{Q\}$
shows $\models \{\mathit{general\text{-}union}\ P\}\ C\ \{\mathit{general\text{-}union}\ Q\}$
⟨*proof*⟩

proposition *rule-Empty*:
$\models \{\ (\lambda S.\ S = \{\})\ \}\ C\ \{\ (\lambda S.\ S = \{\})\ \}$
⟨*proof*⟩

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

lemma *has-subset-join-same*:
entails (*has-subset P*) (*join P* (λ-. *True*))
entails (*join P* (λ-. *True*)) (*has-subset P*)
⟨*proof*⟩

proposition *rule-AtLeast*:
assumes $\models \{P\}\ C\ \{Q\}$
shows $\models \{\mathit{has\text{-}subset}\ P\}\ C\ \{\mathit{has\text{-}subset}\ Q\}$
⟨*proof*⟩

definition *has-superset* **where**
$\mathit{has\text{-}superset}\ P\ S \longleftrightarrow (\exists S'.\ S \subseteq S' \wedge P\ S')$

proposition *rule-AtMost*:
assumes $\models \{P\}\ C\ \{Q\}$
shows $\models \{\mathit{has\text{-}superset}\ P\}\ C\ \{\mathit{has\text{-}superset}\ Q\}$
⟨*proof*⟩

5.6 Synchronous Reasoning (Proposition 14, Appendix H).

theorem *if-sync-rule*:
assumes $\models \{P\}\ \mathit{C1}\ \{\mathit{P1}\}$
and $\models \{P\}\ \mathit{C2}\ \{\mathit{P2}\}$

and $\models \{combine\ from\text{-}nat\ x\ P1\ P2\}\ C\ \{combine\ from\text{-}nat\ x\ R1\ R2\}$
and $\models \{R1\}\ C1'\ \{Q1\}$
and $\models \{R2\}\ C2'\ \{Q2\}$

and *not-free-var-hyper x P1*
and *not-free-var-hyper x P2*
and *from-nat 1* $\neq$ *from-nat 2*

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

shows $\models \{P\}\ If\ (Seq\ C1\ (Seq\ C\ C1'))\ (Seq\ C2\ (Seq\ C\ C2'))\ \{join\ Q1\ Q2\}$
⟨*proof*⟩

definition *update-lvar-set* **where**
update-lvar-set u e S = { ((*fst* φ')(*u* := *e* φ'), *snd* φ') |φ'. $\varphi' \in S$}

lemma *equal-outside-set-helper*:
equal-outside-set {*u*} (*fst* φ) (*fst* ((*fst* φ)(*u* := *x*), *snd* φ))
⟨*proof*⟩

lemma *same-update-lvar-set*:
same-mod-updates {*u*} *S* (*update-lvar-set u e S*)
⟨*proof*⟩

lemma *same-mod-updates-empty*:
assumes *same-mod-updates vars* {} S'
shows $S' = \{\}$
⟨*proof*⟩

definition *not-fv-hyper* **where**
not-fv-hyper t A $\longleftrightarrow$ ($\forall$ *S S'*. *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'*
⟨*proof*⟩

definition *assign-exp-to-lvar* **where**
assign-exp-to-lvar e l φ = ((*fst* φ)(*l* := *e* (*snd* φ)), *snd* φ)

definition *assign-exp-to-lvar-set* **where**
assign-exp-to-lvar-set e l S = *assign-exp-to-lvar e l* ‘ *S*

lemma *same-outside-set-lvar-assign-exp*:

snd φ = *snd* (*assign-exp-to-lvar e l* φ) $\wedge$ *equal-outside-set* $\{l\}$ (*fst* φ) (*fst* (*assign-exp-to-lvar e l* φ))
⟨*proof*⟩

lemma *assign-exp-to-lvar-set-same-mod-updates*:
same-mod-updates $\{l\}$ *S* (*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* (*assign-exp-to-lvar-set e t S*)
⟨*proof*⟩

lemma *holds-forall-same-assign-lvar*:
holds-forall b S $\longleftrightarrow$ *holds-forall b* (*assign-exp-to-lvar-set e l S*) (**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

definition *e-recorded-in-t* **where**
e-recorded-in-t e t S $\longleftrightarrow$ ($\forall \varphi \in S$. *fst* φ *t* = *e* (*snd* φ))

lemma *e-recorded-in-tI*:
assumes $\bigwedge \varphi$. $\varphi \in S \Longrightarrow$ *fst* φ *t* = *e* (*snd* φ)
shows *e-recorded-in-t e t S*
⟨*proof*⟩

definition *e-smaller-than-t* **where**
e-smaller-than-t e t lt S $\longleftrightarrow$ ($\forall \varphi \in S$. *lt* (*e* (*snd* φ)) (*fst* φ *t*))

lemma *low-expI*:
assumes $\bigwedge \varphi\ \varphi'$. $\varphi \in S \wedge \varphi' \in S \Longrightarrow$ (*e* (*snd* φ) = *e* (*snd* φ'))
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*) ⟷ *low-exp b S* (**is** *?A* ⟷ *?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* ⇒ *'a* ⇒ *'a*
type-synonym *'a comp* = *'a* ⇒ *'a* ⇒ *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 ⇒ '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* — $\forall <\varphi>.\ A$
| *AExistsState 'a assertion* — $\exists <\varphi>.\ A$
| *AForall 'a assertion* — $\forall\, y.\ A$
| *AExists 'a assertion* — $\exists\, y.\ A$
| *AOr 'a assertion 'a assertion* — $A \vee A$

| *AAnd 'a assertion 'a assertion* — $A \wedge A$

We use a list of values and a list of states to track quantified values and states, respectively.

fun *interp-exp* :: *'a list* ⇒ *'a nstate list* ⇒ *'a exp* ⇒ *'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* ⇒ *'a nstate list* ⇒ *'a assertion* ⇒ *'a nstate set* ⇒ *bool* **where**
sat-assertion vals states (*AConst b*) - ⟷ *b*
| *sat-assertion vals states* (*AComp e1 cmp e2*) - ⟷ *cmp* (*interp-exp vals states e1*) (*interp-exp vals states e2*)
| *sat-assertion vals states* (*AForallState A*) *S* ⟷ (∀ φ∈*S*. *sat-assertion vals* (φ # *states*) *A S*)
| *sat-assertion vals states* (*AExistsState A*) *S* ⟷ (∃ φ∈*S*. *sat-assertion vals* (φ # *states*) *A S*)
| *sat-assertion vals states* (*AForall A*) *S* ⟷ (∀ *v*. *sat-assertion* (*v* # *vals*) *states A S*)
| *sat-assertion vals states* (*AExists A*) *S* ⟷ (∃ *v*. *sat-assertion* (*v* # *vals*) *states A S*)
| *sat-assertion vals states* (*AAnd A B*) *S* ⟷ (*sat-assertion vals states A S* ∧ *sat-assertion vals states B S*)
| *sat-assertion vals states* (*AOr A B*) *S* ⟷ (*sat-assertion vals states A S* ∨ *sat-assertion vals states B S*)

Negation and implication are defined on top of this base language.

definition *neg-cmp* :: *'a comp* ⇒ *'a comp* **where**
neg-cmp cmp v1 v2 ⟷ ¬ (*cmp v1 v2*)

fun *ANot* **where**
ANot (*AConst b*) = *AConst* (¬ *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 ⟷ ¬(sat-assertion vals states A S)
⟨*proof*⟩

lemma *sat-assertion-Imp*:
sat-assertion vals states (AImp A B) S ⟷ (sat-assertion vals states A S ⟶ sat-assertion vals states B S)
⟨*proof*⟩

abbreviation *interp-assert* **where** *interp-assert ≡ sat-assertion [] []*

6.2 Assume rule

fun *transform-assume* :: *'a assertion ⇒ 'a assertion ⇒ '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 ⇒ ('a npstate ⇒ bool) ⇒ bool*
where
same-syn-sem bsyn bsem ⟷
(∀ states vals S. length states > 0 ⟶ bsem (snd (hd states)) = sat-assertion vals states bsyn S)

lemma *same-syn-semI*:
assumes *⋀states vals S. length states > 0 ⟹ bsem (snd (hd states)) ⟷ 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 ∘ snd) S)*
⟷ 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) = EPVar 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 =
  PBConst bool
  | PBAnd 'a pbexp 'a pbexp
  | PBOr 'a pbexp 'a pbexp
  | PBComp 'a pexp 'a comp 'a pexp

fun interp-pbexp :: 'a pbexp ⇒ 'a npstate ⇒ bool
  where
  interp-pbexp (PBConst b) - ⟷ b
| interp-pbexp (PBAnd pb1 pb2) φ ⟷ interp-pbexp pb1 φ ∧ interp-pbexp pb2 φ
| interp-pbexp (PBOr pb1 pb2) φ ⟷ interp-pbexp pb1 φ ∨ interp-pbexp pb2 φ
| interp-pbexp (PBComp p1 cmp p2) φ ⟷ cmp (interp-pexp p1 φ) (interp-pexp
p2 φ)

fun pbexp-to-assertion where
  pbexp-to-assertion - (PBConst b) = AConst b
| pbexp-to-assertion st (PBAnd pb1 pb2) = AAnd (pbexp-to-assertion st pb1) (pbexp-to-assertion
st pb2)
| pbexp-to-assertion st (PBOr pb1 pb2) = AOr (pbexp-to-assertion st pb1) (pbexp-to-assertion
st pb2)
| pbexp-to-assertion st (PBComp p1 cmp p2) = AComp (pexp-to-exp st p1) cmp
(pexp-to-exp st p2)

lemma same-syn-sem-assertion:
  interp-pbexp pb (snd (states ! st)) = sat-assertion vals states (pbexp-to-assertion
```

st pb) *S*
⟨*proof*⟩

lemma *pexp-to-exp-same*:
shows *same-syn-sem* (*pbexp-to-assertion 0 pb*) (*interp-pbexp pb*)
⟨*proof*⟩

6.2.3 Syntactic rule for assume

theorem *rule-assume-syntactic-general*:
⊨ { *sat-assertion states vals* (*transform-assume* (*pbexp-to-assertion 0 pb*) *P*) }
Assume (*interp-pbexp pb*) {*sat-assertion states vals P*}
⟨*proof*⟩

theorem *rule-assume-syntactic*:
⊨ { *interp-assert* (*transform-assume* (*pbexp-to-assertion 0 pb*) *P*) } *Assume* (*interp-pbexp pb*) {*interp-assert P*}
⟨*proof*⟩

6.3 Havoc rule

6.3.1 Shifting variables

fun *insert-at* **where**
insert-at 0 x l = *x* # *l*
| *insert-at* (*Suc n*) *x* (*t* # *q*) = *t* # (*insert-at n x q*)
| *insert-at* (*Suc n*) *x* [] = [*x*]

lemma *length-insert-at*:
length (*insert-at n x l*) = *length l* + *1*
⟨*proof*⟩

lemma *insert-at-charact-1*:
n ≤ *length l* ⟹ *k* < *n* ⟹ (*insert-at n x l*) ! *k* = *l* ! *k*
⟨*proof*⟩

lemma *insert-at-charact-2*:
n ≤ *length l* ⟹ (*insert-at n x l*) ! *n* = *x*
⟨*proof*⟩

lemma *insert-at-charact-3*:
n ≤ *length l* ⟹ *k* ≥ *n* ⟹ (*insert-at n x l*) ! (*Suc k*) = *l* ! *k*
⟨*proof*⟩

fun *shift-vars-exp* **where**
shift-vars-exp n (*EQVar x*) = (*if x* ≥ *n then EQVar* (*Suc x*) *else EQVar x*)

| *shift-vars-exp n* (*EBinop e1 op e2*) = *EBinop* (*shift-vars-exp n e1*) *op* (*shift-vars-exp n e2*)
| *shift-vars-exp n* (*EFun p e*) = *EFun p* (*shift-vars-exp n e*)
| *shift-vars-exp* - *e* = *e*

fun *shift-states-exp* **where**
shift-states-exp n (*EPVar φ x*) = (*if φ* ≥ *n then EPVar* (*Suc φ*) *x else EPVar φ x*)
| *shift-states-exp n* (*ELVar φ x*) = (*if φ* ≥ *n then ELVar* (*Suc φ*) *x else ELVar φ x*)
| *shift-states-exp n* (*EBinop e1 op e2*) = *EBinop* (*shift-states-exp n e1*) *op* (*shift-states-exp n e2*)
| *shift-states-exp n* (*EFun p e*) = *EFun p* (*shift-states-exp n e*)
| *shift-states-exp* - *e* = *e*

fun *wf-exp* :: *nat* ⇒ *nat* ⇒ *'a exp* ⇒ *bool* **where**
wf-exp nv ns (*EPVar st* -) ⟷ *st* < *ns*
| *wf-exp nv ns* (*ELVar st* -) ⟷ *st* < *ns*
| *wf-exp nv ns* (*EQVar x*) ⟷ *x* < *nv*
| *wf-exp nv ns* (*EBinop e1* - *e2*) ⟷ *wf-exp nv ns e1* ∧ *wf-exp nv ns e2*
| *wf-exp nv ns* (*EFun f e*) ⟷ *wf-exp nv ns e*
| *wf-exp nv ns* (*EConst* -) ⟷ *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* ≤ *length 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* ≤ *length 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)
| *shift-vars n* (*AForall A*) = *AForall* (*shift-vars* (*Suc n*) *A*)
| *shift-vars n* (*AExists A*) = *AExists* (*shift-vars* (*Suc n*) *A*)
| *shift-vars n* (*AForallState A*) = *AForallState* (*shift-vars n A*)
| *shift-vars n* (*AExistsState A*) = *AExistsState* (*shift-vars n A*)
| *shift-vars n* (*AOr A B*) = *AOr* (*shift-vars n A*) (*shift-vars n B*)
| *shift-vars n* (*AAnd A B*) = *AAnd* (*shift-vars n A*) (*shift-vars n B*)

lemma *shift-vars-charact*:
assumes *n* ≤ *length vals*
shows *sat-assertion vals states A S* ⟷ *sat-assertion* (*insert-at n x vals*) *states* (*shift-vars n A*) *S*
⟨*proof*⟩

6.3.2 Expressions (Boolean and values)

definition *update-state* **where**
update-state φ *x v* = (*fst* φ, (*snd* φ)(*x* := *v*))

fun *subst-exp-single* :: *qstate* ⇒ *var* ⇒ *'a exp* ⇒ *'a exp* ⇒ *'a exp* **where**
subst-exp-single φ *x e'* (*EPVar st y*) = (*if* φ = *st* ∧ *x* = *y then e' else EPVar st y*)
| *subst-exp-single* φ *x e'* (*EBinop e1 bop e2*) = *EBinop* (*subst-exp-single* φ *x e' e1*) *bop* (*subst-exp-single* φ *x e' e2*)
| *subst-exp-single* φ *x e'* (*EFun f e*) = *EFun f* (*subst-exp-single* φ *x e' e*)
| *subst-exp-single - - - e* = *e*

lemma *wf-subst-exp*:
assumes *wf-exp nv ns e*
and *wf-exp nv ns e'*
shows *wf-exp nv ns* (*subst-exp-single* φ *x e' e*)
⟨*proof*⟩

lemma *subst-exp-single-charact*:
assumes *interp-exp vals states e'* = *snd* (*states* ! *st*) *x*
shows *interp-exp vals states* (*subst-exp-single st x e' e*) = *interp-exp vals states e*
⟨*proof*⟩

definition *subst-state* **where**
subst-state x pe φ = (*fst* φ, (*snd* φ)(*x* := *interp-pexp pe* (*snd* φ)))

definition *update-state-at* **where**
update-state-at states n x v = *list-update states n* (*update-state* (*states* ! *n*) *x v*)

lemma *update-state-at-fst*:
fst (*update-state-at states n x v* ! *st*) = *fst* (*states* ! *st*)
⟨*proof*⟩

lemma *update-state-at-snd-1*:
x ≠ *y* ⟹ *snd* (*update-state-at states n x v* ! *st*) *y* = *snd* (*states* ! *st*) *y*
⟨*proof*⟩

lemma *update-state-at-snd-2*:
st ≠ *n* ⟹ *snd* (*update-state-at states n x v* ! *st*) *y* = *snd* (*states* ! *st*) *y*
⟨*proof*⟩

lemma *update-state-at-snd-3*:
assumes *n* < *length states*
shows *snd* (*update-state-at states n x v* ! *n*) *x* = *v*
⟨*proof*⟩

lemma *subst-exp-more-complex-charact*:
assumes *states'* = *update-state-at states st x* (*interp-exp vals states e'*)
and *st* < *length states*
shows *interp-exp vals states* (*subst-exp-single st x e' e*) = *interp-exp vals states'* *e*
⟨*proof*⟩

6.3.3 Assertions

fun *subst-assertion-single* :: *qstate* ⇒ *var* ⇒ *'a exp* ⇒ *'a assertion* ⇒ *'a assertion* **where**
subst-assertion-single st x e (*AConst b*) = *AConst b*
| *subst-assertion-single st x e* (*AComp e1 cmp e2*) = *AComp* (*subst-exp-single st x e e1*) *cmp* (*subst-exp-single st x e e2*)
| *subst-assertion-single st x e* (*AForall A*) = *AForall* (*subst-assertion-single st x* (*shift-vars-exp 0 e*) *A*)
| *subst-assertion-single st x e* (*AExists A*) = *AExists* (*subst-assertion-single st x* (*shift-vars-exp 0 e*) *A*)
| *subst-assertion-single st x e* (*AOr A B*) = *AOr* (*subst-assertion-single st x e A*) (*subst-assertion-single st x e B*)
| *subst-assertion-single st x e* (*AAnd A B*) = *AAnd* (*subst-assertion-single st x e A*) (*subst-assertion-single st x e B*)
| *subst-assertion-single st x e* (*AForallState A*) = *AForallState* (*subst-assertion-single* (*Suc st*) *x* (*shift-states-exp 0 e*) *A*)
| *subst-assertion-single st x e* (*AExistsState A*) = *AExistsState* (*subst-assertion-single* (*Suc st*) *x* (*shift-states-exp 0 e*) *A*)

fun *wf-assertion-aux* :: *nat* ⇒ *nat* ⇒ *'a assertion* ⇒ *bool* **where**
wf-assertion-aux nv ns (*AConst b*) ⟷ *True*
| *wf-assertion-aux nv ns* (*AComp e1 cmp e2*) ⟷ *wf-exp nv ns e1* ∧ *wf-exp nv ns e2*

| *wf-assertion-aux nv ns* (*AAnd A B*) ⟷ *wf-assertion-aux nv ns A* ∧ *wf-assertion-aux nv ns B*
| *wf-assertion-aux nv ns* (*AOr A B*) ⟷ *wf-assertion-aux nv ns A* ∧ *wf-assertion-aux nv ns B*

| *wf-assertion-aux nv ns* (*AForall A*) ⟷ *wf-assertion-aux* (*Suc nv*) *ns A*
| *wf-assertion-aux nv ns* (*AExists A*) ⟷ *wf-assertion-aux* (*Suc nv*) *ns A*
| *wf-assertion-aux nv ns* (*AForallState A*) ⟷ *wf-assertion-aux nv* (*Suc ns*) *A*
| *wf-assertion-aux nv ns* (*AExistsState A*) ⟷ *wf-assertion-aux nv* (*Suc ns*) *A*

abbreviation *wf-assertion* **where** *wf-assertion* ≡ *wf-assertion-aux 0 0*

lemma *wf-shift-vars*:
assumes *wf-assertion-aux nv ns A*
shows *wf-assertion-aux* (*Suc nv*) *ns* (*shift-vars n A*)
⟨*proof*⟩

lemma *wf-subst-assertion*:
assumes *wf-assertion-aux nv ns A*
and *wf-exp nv ns e*
shows *wf-assertion-aux nv ns* (*subst-assertion-single* φ *x e A*)
⟨*proof*⟩

lemma *subst-assertion-single-charact*:
assumes *interp-exp vals states e* = *snd* (*states* ! *st*) *x*
shows *sat-assertion vals states* (*subst-assertion-single st x e A*) *S* ⟷ *sat-assertion vals states A S*
⟨*proof*⟩

lemma *update-state-at-cons*:
update-state-at (φ # *states*) (*Suc n*) *x v* = φ # *update-state-at states n x v*
⟨*proof*⟩

lemma *subst-assertion-single-charact-better*:
assumes *states′* = *update-state-at states st x* (*interp-exp vals states e*)
and *st* < *length states*
shows *sat-assertion vals states* (*subst-assertion-single st x e A*) *S* ⟷ *sat-assertion vals states′ A S*
⟨*proof*⟩

6.3.4 Transformation for havoc

fun *transform-havoc* **where**
transform-havoc x (*AForallState A*) = *AForallState* (*AForall* (*subst-assertion-single*

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

lemma *sem-havoc-bis*:
sem (Havoc x) S = {(fst φ, (snd φ)(x := v)) |φ v. φ ∈ S} (**is** *?A = ?B*)
⟨*proof*⟩

lemma *helper-update-state*:
(v # vals) ! 0 = snd ((update-state φ x v # states) ! 0) x
⟨*proof*⟩

lemma *helper-S-update-states*:
assumes *S' = { update-state φ x v |φ v. φ ∈ S}*
shows *(∀φ ∈ S'. Q φ) ⟷ (∀φ ∈ S. ∀v. Q (update-state φ x v))*
⟨*proof*⟩

lemma *helper-S-update-states-exists*:
assumes *S' = { update-state φ x v |φ v. φ ∈ S}*
shows *(∃φ ∈ S'. Q φ) ⟷ (∃φ ∈ S. ∃v. Q (update-state φ x v))*
⟨*proof*⟩

lemma *equiv-havoc-transform*:
assumes *S' = { update-state φ x v |φ v. φ ∈ S}*
shows *sat-assertion vals states P S' ⟷ sat-assertion vals states (transform-havoc x P) S*
⟨*proof*⟩

6.3.5 Syntactic rule for havoc

theorem *rule-havoc-syntactic-general*:
⊨ { sat-assertion states vals (transform-havoc x P) } Havoc x {sat-assertion states vals P}
⟨*proof*⟩

theorem *rule-havoc-syntactic*:
⊨ { 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* ⇒ *'a pexp* ⇒ *'a pexp* ⇒ *'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* (*σ*(*x* := *interp-pexp e' σ*))
⟨*proof*⟩

fun *subst-pbexp* :: *var* ⇒ *'a pexp* ⇒ *'a pbexp* ⇒ *'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*) *σ* ⟷ *interp-pbexp pb* (*σ*(*x* := *interp-pexp e σ*))
⟨*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* ⇒ *'a pexp* ⇒ *'a exp* ⇒ *'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*:
⊨ { *sat-assertion vals states* (*transform-assign x pe P*) } *Assign x* (*interp-pexp pe*) {*sat-assertion vals states P*}
⟨*proof*⟩

theorem *rule-assign-syntactic*:
⊨ { *interp-assert* (*transform-assign x pe P*) } *Assign x* (*interp-pexp pe*) {*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*) ⟷ *no-exists-state A* ∧ *no-exists-state B*
| *no-exists-state* (*AOr A B*) ⟷ *no-exists-state A* ∧ *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 ⟷ (∀ *S S' vals states*. $S \subseteq S'$ ∧ *sat-assertion vals states A S* ⟶ *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 ⋀φ. φ ∈ *S* ⟹ *sat-assertion vals* (φ # *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* ∪ *S2*)
⟨*proof*⟩

lemma *general-join-entails*:
assumes *up-closed A*
and ⋀*x*. *sat-assertion vals states* (*AForallState A*) (*F x*)
shows *sat-assertion vals states* (*AForallState A*) (⋃ *x*. *F x*)
⟨*proof*⟩

fun *no-forall-state* :: *'a assertion* ⇒ *bool*
where
no-forall-state (*AConst* -) ⟷ *True*
| *no-forall-state* (*AComp* - - -) ⟷ *True*

| *no-forall-state* (*AForallState A*) ⟷ *False*
| *no-forall-state* (*AExistsState A*) ⟷ *no-forall-state A*
| *no-forall-state* (*AForall A*) ⟷ *no-forall-state A*
| *no-forall-state* (*AExists A*) ⟷ *no-forall-state A*
| *no-forall-state* (*AAnd A B*) ⟷ *no-forall-state A* ∧ *no-forall-state B*
| *no-forall-state* (*AOr A B*) ⟷ *no-forall-state A* ∧ *no-forall-state B*

lemma *no-forall-exists-state-not*:
no-forall-state A ≡ *no-exists-state* (*ANot A*)
⟨*proof*⟩

fun *no-forall-state-after-existential* :: *'a assertion* ⇒ *bool*
where
no-forall-state-after-existential (*AConst* -) ⟷ *True*
| *no-forall-state-after-existential* (*AComp* - - -) ⟷ *True*
| *no-forall-state-after-existential* (*AForallState A*) ⟷ *no-forall-state-after-existential A*
| *no-forall-state-after-existential* (*AForall A*) ⟷ *no-forall-state-after-existential A*
| *no-forall-state-after-existential* (*AAnd A B*) ⟷ *no-forall-state-after-existential A* ∧ *no-forall-state-after-existential B*
| *no-forall-state-after-existential* (*AOr A B*) ⟷ *no-forall-state-after-existential A* ∧ *no-forall-state-after-existential B*
| *no-forall-state-after-existential* (*AExists A*) ⟷ *no-forall-state A*
| *no-forall-state-after-existential* (*AExistsState A*) ⟷ *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* (⋃ *n*. *S n*)
⟨*proof*⟩

definition *shift-sequence* :: (*nat* ⇒ *'a*) ⇒ *nat* ⇒ (*nat* ⇒ *'a*)
where
shift-sequence S n m = *S* (*m* + *n*)

lemma *shift-sequence-properties*:
assumes *ascending S*
shows *ascending* (*shift-sequence S n*)
and (⋃ *m*. *S m*) = (⋃ *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*) ∧ *n* > *extract-indices-sat-P P S m*)

definition *holds-infinitely-often* **where**
holds-infinitely-often P S $\longleftrightarrow$ $(\forall\, m.\ \exists\, n.\ n > m \wedge P\ (S\ n))$

lemma *extract-indices-sat-P-properties*:
assumes *holds-infinitely-often P S*
shows *P* (*S* (*extract-indices-sat-P P S 0*))
and $n > 0 \Longrightarrow P\ (S\ (\textit{extract-indices-sat-P}\ P\ S\ n))$
$\wedge$ *extract-indices-sat-P P S n* > *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* (*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* (*subseq-sat P S n*)
and $(\bigcup n.\ S\ n) = (\bigcup n.$ *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 $\bigwedge n.$ *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* ($\lambda n.$ *interp-assert A*) (*interp-assert A*)
⟨*proof*⟩

theorem *while-general-syntactic*:
assumes $\bigwedge n. \models \{P\ n\}$ *if-then b C* $\{P\ (\textit{Suc}\ n)\}$
and $\bigwedge n. \models \{P\ n\}$ *Assume* (*lnot b*) {*interp-assert A*}
and *no-forall-state-after-existential A*
shows $\models \{P\ 0\}$ *while-cond b C* {*conj* (*interp-assert A*) (*holds-forall* (*lnot b*))}
⟨*proof*⟩

theorem *while-forall-exists-simpler*:
assumes $\models \{I\}$ *if-then b C* $\{I\}$
and $\models \{I\}$ *Assume* (*lnot b*) $\{$*interp-assert Q*$\}$
and *no-forall-state-after-existential Q*
shows $\models \{I\}$ *while-cond b C* $\{$*conj* (*interp-assert Q*) (*holds-forall* (*lnot b*))$\}$
⟨*proof*⟩

theorem *while-d-syntactic*:
assumes $\models \{$ *interp-assert A* $\}$ *if-then b C* $\{$ *interp-assert A* $\}$
and *no-forall-state-after-existential A*
and *no-exists-state A*
shows $\models \{$ *interp-assert A* $\}$ *while-cond b C* $\{$*conj* (*interp-assert A*) (*holds-forall* (*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 $\longleftrightarrow$ ($\forall$ *vals states S. sat-assertion vals states A S* $\longleftrightarrow$ *sat-assertion vals states B S*)

lemma *forall-commute*:
equiv (*AForallState* (*AForall A*)) (*AForall* (*AForallState A*))
⟨*proof*⟩

lemma *exists-commute*:
equiv (*AExistsState* (*AExists A*)) (*AExists* (*AExistsState A*))
⟨*proof*⟩

lemma *forall-state-and*:
equiv (*AForallState* (*AAnd A B*)) (*AAnd* (*AForallState A*) (*AForallState B*))
⟨*proof*⟩

lemma *exists-state-or*:
equiv (*AExistsState* (*AOr A B*)) (*AOr* (*AExistsState A*) (*AExistsState B*))
⟨*proof*⟩

lemma *forall-and*:
equiv (*AForall* (*AAnd A B*)) (*AAnd* (*AForall A*) (*AForall 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* (λ(*n*::*nat*). *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* ⇒ *nat set* **where**
wr Skip = {}
| *wr* (*Assign x* -) = {*x*}
| *wr* (*Havoc x*) = {*x*}
| *wr* (*Assume b*) = {}
| *wr* (*C1*;; *C2*) = *wr C1* ∪ *wr C2*
| *wr* (*If C1 C2*) = *wr C1* ∪ *wr C2*
| *wr* (*While C*) = *wr C*

definition *agree-on* **where**
agree-on V σ σ′ ⟷ (∀ *x*∈*V*. σ *x* = σ′ *x*)

lemma *agree-onI*:
assumes ⋀*x*. *x*∈*V* ⟹ σ *x* = σ′ *x*
shows *agree-on V* σ σ′
⟨*proof*⟩

lemma *agree-onE*:
assumes *agree-on V* σ σ′
and *x* ∈ *V*
shows σ *x* = σ′ *x*
⟨*proof*⟩

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

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

lemma *agree-on-sym*:
assumes *agree-on* $V\ \sigma\ \sigma'$
shows *agree-on* $V\ \sigma'\ \sigma$
⟨*proof*⟩

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

fun *fv-exp* :: $'a$ *exp* ⇒ *var set* **where**
fv-exp (*EBinop e1 - e2*) = *fv-exp e1* ∪ *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*
⟨*proof*⟩

fun *fv* **where**
fv (*AAnd F1 F2*) = *fv F1* ∪ *fv F2*
| *fv* (*AOr F1 F2*) = *fv F1* ∪ *fv F2*
| *fv* (*AForall F*) = *fv F*
| *fv* (*AExists F*) = *fv F*
| *fv* (*AForallState F*) = *fv F*
| *fv* (*AExistsState F*) = *fv F*
| *fv* (*AConst b*) = {}

| *fv* (*AComp e1 cmp e2*) = *fv-exp e1* ∪ *fv-exp e2*

lemma *fv-wr-charact-aux*:

assumes *agree-on* (*fv F*) σ σ′
and $n \leq$ *length states*
and *sat-assertion vals* (*insert-at n* (*l*, σ) *states*) *F S*
and *wf-assertion-aux nv* (*Suc* (*length states*)) *F*
shows *sat-assertion vals* (*insert-at n* (*l*, σ′) *states*) *F S*
⟨*proof*⟩

lemma *fv-wr-charact*:
assumes *agree-on* (*fv F*) σ σ′
and *sat-assertion vals* ((*l*, σ) # *states*) *F S*
and *wf-assertion-aux nv* (*Suc* (*length states*)) *F*
shows *sat-assertion vals* ((*l*, σ′) # *states*) *F S*
⟨*proof*⟩

lemma *syntactic-safe-frame-preserved*:
assumes *wr C* ∩ *fv F* = {}
and *sat-assertion vals states F S*
and *wf-assertion-aux nv* (*length states*) *F*
and *no-exists-state F*
shows *sat-assertion vals states F* (*sem C S*)
⟨*proof*⟩

theorem *safe-frame-rule-syntactic*:
assumes *wr C* ∩ *fv F* = {}
and *wf-assertion F*
and *no-exists-state F*
shows ⊨ {*interp-assert F*} *C* {*interp-assert F*}
⟨*proof*⟩

theorem *LUpdateS*:
assumes ⊨ { (λ*S*. *P S* ∧ *e-recorded-in-t e t S*) } *C* { *Q* }
and *not-fv-hyper t P*
and *not-fv-hyper t Q*
shows ⊨ { *P* } *C* { *Q* }
⟨*proof*⟩

end
theory *TotalLogic*
imports *Loops Compositionality SyntacticAssertions*

begin

7 Terminating Hyper-Triples

definition *total-hyper-triple* (⊨*TERM* {-} - {-} [*51*,*0*,*0*] *81*) **where**
⊨*TERM* {*P*} *C* {*Q*} ⟷ (⊨ {*P*} *C* {*Q*} ∧ (∀ *S*. *P S* ⟶ (∀ φ ∈ *S*. ∃ σ′. *single-sem C* (*snd* φ) σ′)))

lemma *total-hyper-triple-equiv*:
⊨*TERM* {*P*} *C* {*Q*} ⟷ (⊨ {*P*} *C* {*Q*} ∧ (∀ *S*. *P S* ⟶ (∀ φ ∈ *S*. ∃ σ′. (*fst* φ, σ′) ∈ *sem C S* ∧ *single-sem C* (*snd* φ) σ′)))
⟨*proof*⟩

lemma *total-hyper-tripleI*:
assumes ⊨ {*P*} *C* {*Q*}
and ⋀φ *S*. *P S* ∧ φ ∈ *S* ⟹ (∃ σ′. *single-sem C* (*snd* φ) σ′)
shows ⊨*TERM* {*P*} *C* {*Q*}
⟨*proof*⟩

definition *terminates-in* **where**
terminates-in C S ⟷ (∀ φ ∈ *S*. ∃ σ′. *single-sem C* (*snd* φ) σ′)

lemma *terminates-inI*:
assumes ⋀φ. φ ∈ *S* ⟹ ∃ σ′. *single-sem C* (*snd* φ) σ′
shows *terminates-in C S*
⟨*proof*⟩

lemma *iterate-sem-mono*:
assumes *S* ⊆ *S*′
shows *iterate-sem n C S* ⊆ *iterate-sem n C S*′
⟨*proof*⟩

lemma *terminates-in-while-loop*:
assumes *wfP lt*
and ⋀φ *n*. φ ∈ *iterate-sem n* (*Assume b*;; *C*) *S* ∧ *b* (*snd* φ) ⟹ (∃ σ′. *single-sem C* (*snd* φ) σ′ ∧ (¬ *b* σ′ ∨ *lt* (*e* σ′) (*e* (*snd* φ))))
shows *terminates-in* (*while-cond b C*) *S*
⟨*proof*⟩

lemma *total-hyper-triple-altI*:
assumes ⋀*S*. *P S* ⟹ *Q* (*sem C S*)
and ⋀*S*. *P S* ⟹ *terminates-in C S*
shows ⊨*TERM* {*P*} *C* {*Q*}
⟨*proof*⟩

lemma *syntactic-frame-preserved*:
assumes *terminates-in C S*

and *wr C ∩ fv F = {}*
and *sat-assertion vals states F S*
and *wf-assertion-aux nv (length states) F*
shows *sat-assertion vals states F (sem C S)*
⟨*proof*⟩

theorem *frame-rule-syntactic*:
assumes ⊨*TERM {P} C {Q}*
and *wr C ∩ fv F = {}*
and *wf-assertion F*
shows ⊨*TERM {conj P (interp-assert F)} C {conj Q (interp-assert F)}*
⟨*proof*⟩

7.1 Specialize rule

definition *same-syn-sem-all* :: *'a assertion* ⇒ *((nat, 'a, nat, 'a) state* ⇒ *bool)* ⇒ *bool*
where
same-syn-sem-all bsyn bsem ⟷
(∀ *states vals S. length states > 0* ⟶ *bsem (hd states) = sat-assertion vals states bsyn S)*

lemma *same-syn-sem-allI*:
assumes ⋀*states vals S. length states > 0* ⟹ *bsem (hd states)* ⟷ *sat-assertion vals states bsyn S*
shows *same-syn-sem-all bsyn bsem*
⟨*proof*⟩

lemma *transform-assume-valid*:
assumes *same-syn-sem-all bsyn bsem*
shows *sat-assertion vals states A (Set.filter bsem S)*
⟷ *sat-assertion vals states (transform-assume bsyn A) S*
⟨*proof*⟩

fun *indep-of-set* **where**
indep-of-set (AForall A) ⟷ *indep-of-set A*
| *indep-of-set (AExists A)* ⟷ *indep-of-set A*
| *indep-of-set (AOr A B)* ⟷ *indep-of-set A* ∧ *indep-of-set B*
| *indep-of-set (AAnd A B)* ⟷ *indep-of-set A* ∧ *indep-of-set B*
| *indep-of-set (AComp - - -)* ⟷ *True*
| *indep-of-set (AConst -)* ⟷ *True*
| *indep-of-set (AForallState -)* ⟷ *False*
| *indep-of-set (AExistsState -)* ⟷ *False*

lemma *indep-of-set-charact*:
assumes *indep-of-set A*
and *sat-assertion vals states A S*

shows *sat-assertion vals states A S′*
⟨*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* ⟷ *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* (*φ* # *states*) *A S* ⟷ *sat-assertion* [] [*φ*] *A* {} (**is** *?A* ⟷ *?B*)
⟨*proof*⟩

definition *get-bsem* **where**
get-bsem bsyn φ ⟷ *sat-assertion* [] [*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 φ* = *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* ∩ *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*)
⟨*proof*⟩

lemma *filter-rule-semantic*:
assumes ⊨ {*interp-assert P*} *C* {*interp-assert Q*}
and *same-syn-sem-all bsyn bsem*
and *wr C* ∩ *fv bsyn* = {}
and *wf-assertion-aux 0* (*Suc 0*) *bsyn*
shows ⊨ { *interp-assert* (*transform-assume bsyn P*) } *C* { *interp-assert* (*transform-assume bsyn Q*) }
⟨*proof*⟩

lemma *filter-rule-syntactic*:
assumes ⊨ {*interp-assert P*} *C* {*interp-assert Q*}
and *indep-of-set b*
and *wf-assertion-aux 0 1 b*
and *wr C* ∩ *fv b* = {}
shows ⊨ { *interp-assert* (*transform-assume b P*) } *C* { *interp-assert* (*transform-assume b Q*) }
⟨*proof*⟩

definition *terminates* **where**
terminates C ⟷ (∀ σ. ∃ σ′. *single-sem C* σ σ′)

lemma *terminatesI*:
assumes ⋀σ. ∃ σ′. *single-sem C* σ σ′
shows *terminates C*
⟨*proof*⟩

lemma *terminates-implies-total*:
assumes ⊨ {*P*} *C* {*Q*}
and *terminates C*
shows ⊨*TERM* {*P*} *C* {*Q*}
⟨*proof*⟩

lemma *terminates-seq*:
assumes *terminates C1*
and *terminates C2*
shows *terminates* (*C1*;; *C2*)
⟨*proof*⟩

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**
$min\text{-}prop\ P = (SOME\ n.\ P\ n \wedge (\forall m.\ m < n \longrightarrow \neg\ P\ m))$

lemma *min-prop-charact*:
assumes *P n*
shows $P\ (min\text{-}prop\ P) \wedge (\forall m.\ m < (min\text{-}prop\ P) \longrightarrow \neg\ P\ m)$
⟨*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 ⋀*n*. *not-fv-hyper t* (*I n*)
and ⋀*n*. ⊨*TERM* {*conj* (*conj* (*I n*) (*holds-forall b*)) (*e-recorded-in-t e t*)} *C* {*conj* (*conj* (*I* (*Suc n*)) (*low-exp b*)) (*e-smaller-than-t e t lt*)}
shows ⊨*TERM* {*conj* (*I 0*) (*low-exp b*)} *while-cond b C* {*conj* (*exists I*) (*holds-forall* (*lnot b*))}
⟨*proof*⟩

lemma *total-consequence-rule*:
assumes *entails P P'*
and *entails Q' Q*
and ⊨*TERM* {*P'*} *C* {*Q'*}
shows ⊨*TERM* {*P*} *C* {*Q*}
⟨*proof*⟩

theorem *WhileSyncTot*:
assumes *wfP lt*
and *not-fv-hyper t I*
and ⊨*TERM* {*conj I* (λ*S*. ∀ φ∈*S*. *b* (*snd* φ) ∧ *fst* φ *t* = *e* (*snd* φ))} *C* {*conj* (*conj I* (*low-exp b*)) (*e-smaller-than-t e t lt*)}
shows ⊨*TERM* {*conj I* (*low-exp b*)} *while-cond b C* {*conj I* (*holds-forall* (*lnot b*))}
⟨*proof*⟩

lemma *total-hyper-tripleE*:
assumes ⊨*TERM* {*P*} *C* {*Q*}
and *P S*
and φ ∈ *S*
shows ∃σ'. (*fst* φ, σ') ∈ *sem C S* ∧ *single-sem C* (*snd* φ) σ'
⟨*proof*⟩

theorem *normal-while-tot*:
assumes ⋀*n*. ⊨ {*P n*} *Assume b* {*Q n*}
and ⋀*n*. ⊨*TERM* {*conj* (*Q n*) (*e-recorded-in-t e t*)} *C* {*conj* (*P* (*Suc n*)) (*e-smaller-than-t e t lt*)}
and ⊨ {*natural-partition P*} *Assume* (*lnot b*) {*R*}

and *wfP lt*
and ⋀*n*. *not-fv-hyper t* (*P n*)
and ⋀*n*. *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-inf* $\longleftrightarrow (\forall S.\ converges\text{-}sets\ S \wedge (\forall n.\ P\ n\ (S\ n)) \longrightarrow P\text{-}inf\ (\bigcup n.\ S\ n))$

lemma *t-closedE*:
assumes *t-closed P P-inf*
and *converges-sets S*
and $\bigwedge n.\ P\ n\ (S\ n)$
shows *P-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 $\models TERM\ \{exists\ P\}\ C\ \{exists\ Q\}$
⟨*proof*⟩

lemma *total-assign-rule*:
$\models TERM\ \{\ (\lambda S.\ P\ \{\ (l,\ \sigma(x := e\ \sigma))\ |l\ \sigma.\ (l,\ \sigma) \in S\ \})\ \}\ (Assign\ x\ e)\ \{P\}$
⟨*proof*⟩

lemma *total-havoc-rule*:
$\models TERM\ \{\ (\lambda S.\ P\ \{\ (l,\ \sigma(x := v))\ |l\ \sigma\ v.\ (l,\ \sigma) \in S\ \})\ \}\ (Havoc\ x)\ \{P\}$
⟨*proof*⟩

lemma *in-semI*:
assumes $\varphi \in S$
and $fst\ \varphi = fst\ \varphi'$
and *single-sem* C $(snd\ \varphi)$ $(snd\ \varphi')$
shows $\varphi' \in sem\ C\ S$
⟨*proof*⟩

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

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* $b\ C\ \{R\}$
⟨*proof*⟩

end

8 Examples

In this file, we prove the correctness of the two compositionality proofs presented in Appendix D.2.

theory *ExamplesCompositionality*
imports *Logic Compositionality*
begin

definition *low* **where**
low l S $\longleftrightarrow$ ($\forall$ *φ1 φ2*. *φ1* $\in$ *S* $\wedge$ *φ2* $\in$ *S* $\longrightarrow$ *snd φ1 l* = *snd φ2 l*)

8.1 Examples using the core rules.

definition *GNI* **where**
GNI l h S $\longleftrightarrow$ ($\forall$ *φ1 φ2*. *φ1* $\in$ *S* $\wedge$ *φ2* $\in$ *S*
$\longrightarrow$ ($\exists$ *φ* $\in$ *S*. *snd φ h* = *snd φ1 h* $\wedge$ *snd φ l* = *snd φ2 l*))

lemma *GNI-I*:
assumes $\bigwedge$*φ1 φ2*. *φ1* $\in$ *S* $\wedge$ *φ2* $\in$ *S*
$\Longrightarrow$ ($\exists$ *φ* $\in$ *S*. *snd φ h* = *snd φ1 h* $\wedge$ *snd φ l* = *snd φ2 l*)
shows *GNI l h S*
⟨*proof*⟩

8.2 Examples using the compositionality rules

definition *has-minimum* :: *'c* $\Rightarrow$ (*'d* $\Rightarrow$ *'d* $\Rightarrow$ *bool*) $\Rightarrow$ (*'a*, *'b*, *'c*, *'d*) *chyperassertion* **where**
has-minimum x leq S $\longleftrightarrow$ ($\exists\omega{\in}S$. $\forall\omega'{\in}S$. *leq* (*snd ω x*) (*snd ω' x*))

lemma *has-minimumI*:
assumes *ω* $\in$ *S*
and $\bigwedge\omega'$. *ω'* $\in$ *S* $\Longrightarrow$ *leq* (*snd ω x*) (*snd ω' x*)
shows *has-minimum x leq S*
⟨*proof*⟩

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

lemma *is-monotonicI*:
assumes $\bigwedge\omega\ \omega'$. *ω* $\in$ *S* $\Longrightarrow$ *ω'* $\in$ *S* $\Longrightarrow$ *fst ω i* = *one* $\wedge$ *fst ω' i* = *two* $\Longrightarrow$ *leq* (*snd ω x*) (*snd ω' x*)
shows *is-monotonic i x one two leq S*
⟨*proof*⟩

lemma *update-logical-equal-outside*:
equal-outside-set {*i*} (*snd ω*) (*snd* (*update-logical ω i v*))

⟨*proof*⟩

lemma *update-logical-read*:
fst (*update-logical* ω *i v*) *i* = *v*
⟨*proof*⟩

lemma *snd-update-logical-same*:
snd (*update-logical* ω *i v*) = *snd* ω
⟨*proof*⟩

Figure 12

proposition *composing-monotonicity-and-minimum*:
fixes *P* :: ((('*a* ⇒ '*b*) × ('*c* ⇒ '*d*)) *set* ⇒ *bool*)
fixes *i* :: '*a*
fixes *x* :: '*c*
fixes *y* :: '*c*
fixes *leq* :: '*d* ⇒ '*d* ⇒ *bool*
fixes *one* :: '*b*
fixes *two* :: '*b*

assumes ⊨ { *P* } *C1* { *has-minimum x leq* }
and ⊨ { *is-monotonic i x one two leq* } *C2* { *is-monotonic i y one two leq* }
and ⊨ { (*is-singleton* :: ((('*a* ⇒ '*b*) × ('*c* ⇒ '*d*)) *set* ⇒ *bool*)) } *C2* { *is-singleton* }
and *one* ≠ *two*

and ⋀*x*. *leq x x* — reflexivity

shows ⊨ { *P* } *C1* ;; *C2* { *has-minimum y leq* }
⟨*proof*⟩

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

definition *lGNI* :: '*pvar* ⇒ '*lvar* ⇒ (('*lvar*, '*lval*, '*pvar*, '*pval*) *state*) *set* ⇒ *bool*
where
lGNI l h S ⟷ (∀ φ*1* ∈ *S*. (∀ φ*2* ∈ *S*. ∃ φ ∈ *S*. *fst* φ *h* = *fst* φ*1 h* ∧ *snd* φ *l* = *snd* φ*2 l*))

Figure 13

proposition *composing-GNI-with-SNI*:
fixes *h* :: '*lvar*
fixes *l* :: '*pvar*

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

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

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* (λσ. (σ *h* :: *int*) + σ *y*)) { *GNI l h* }
⟨*proof*⟩

lemma *program-2-violates-gni*:
assumes $y \neq l \wedge y \neq h \wedge l \neq h$
shows $\vdash$ { *conj* (*low l*) (λ*S*. ∃ *a* ∈ *S*. ∃ *b* ∈ *S*. (*snd a h* :: *nat*) ≠ *snd b h*) }
Seq (*Seq* (*Havoc y*) (*Assume* (λσ. σ *y* ≥ (*0* :: *nat*) ∧ σ *y* ≤ (*100* :: *nat*)))) (*Assign l* (λσ. σ *h* + σ *y*))
{λ(*S* :: (('*lvar* ⇒ '*lval*) × ('*a* ⇒ *nat*)) *set*). ¬ *GNI l h S*}
⟨*proof*⟩

end
theory *PaperResults*
imports *Loops SyntacticAssertions Compositionality TotalLogic ExamplesCompositionality*
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 (*'pvar*, *'pval*) *pstate* (<– you can ctrl+click on the name *pstate* above to jump to its definition).
- Program commands (definition 1) are defined via the type (*'var*, *'val*) *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 (*'lvar*, *'lval*, *'pvar*, *'pval*) *state* (file Language.thy).
- Hyper-assertions (definition 3) are defined as (*'lvar*, *'lval*, *'pvar*, *'pval*) *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*:
sem C (S1 ∪ S2) = sem C S1 ∪ sem C S2
S ⊆ S' ⟹ sem C S ⊆ sem C S'
sem C (⋃x. f x) = (⋃x. sem C (f x))
sem Skip S = S
sem (C1;; C2) S = sem C2 (sem C1 S)
sem (If C1 C2) S = sem C1 S ∪ sem C2 S
sem (While C) S = (⋃n. iterate-sem n C S)
⟨*proof*⟩

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 $\bigotimes$ (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 $\models \{P\}\ C\ \{Q\}$
⟨*proof*⟩

Theorem 2: Completeness

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

9.1.5 3.5: Expressivity of Hyper-Triples

Program hyperproperties (definition 8) are defined in the file ProgramHyperproperties as the type (*'pvar*, *'pval*) *program-hyperproperty*, which is syntactic sugar for the type ((*'pvar*, *'pval*) *pstate* × (*'pvar*, *'pval*) *pstate*) *set* ⇒ *bool*. As written in the paper (after the definition), this type is equivalent to the type ((*'pvar*, *'pval*) *pstate* × (*'pvar*, *'pval*) *pstate*) *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* :: *'pvar* ⇒ *'lvar*
fixes *to-lval* :: *'pval* ⇒ *'lval*
fixes *H* :: (*'pvar*, *'pval*) *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 ∃ *P Q*::(*'lvar*, *'lval*, *'pvar*, *'pval*) *state hyperassertion*. (∀ *C*. *hypersat C H* ⟷ ⊨ {*P*} *C* {*Q*})
⟨*proof*⟩

Theorem 4: Expressing hyper-triples as hyperproperties

theorem *thm4-expressing-hyper-triples-as-hyperproperties*:
$\models \{P\}\ C\ \{Q\} \longleftrightarrow$ *hypersat C* (*hyperprop-hht P Q*)
⟨*proof*⟩

Theorem 5: Disproving hyper-triples

theorem *thm5-disproving-triples*:
$\neg \models \{P\}\ C\ \{Q\} \longleftrightarrow (\exists P'.\ sat\ P' \wedge entails\ P'\ P \wedge \models \{P'\}\ C\ \{\lambda S.\ \neg\ Q\ S\})$
⟨*proof*⟩

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*:
⊢ { *interp-assert* (*transform-assign x e P*) } *Assign x* (*interp-pexp e*) {*interp-assert P*}
⟨*proof*⟩

Rule HavocS from figure 3

proposition *HavocS*:
⊢ { *interp-assert* (*transform-havoc x P*) } *Havoc x* {*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*:
⊢ { *interp-assert* (*transform-assume* (*pbexp-to-assertion 0 b*) *P*) } *Assume* (*interp-pbexp b*) {*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 $\forall <\varphi>.\ \varphi(a)=\varphi(b) \wedge (\exists <\varphi'>.$

$\varphi(x) \succeq \varphi'(y)$) 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 $\bigwedge n. \vdash \{I\ n\}$ *Assume b*;; C $\{I$ (*Suc n*)$\}$
and $\vdash$ { *natural-partition I* } *Assume* (*lnot b*) { Q }
shows $\vdash \{I\ 0\}$ *while-cond b C* { Q }
⟨*proof*⟩

This result uses the following constructs:

- *natural-partition I* corresponds to the $\bigotimes$ operator from definition 7.
- *lnot b* negates b.
- *while-cond b C* is defined as *While* (*Assume b* ;; *C*) ;; *Assume* (*lnot b*).

Rule WhileSync from figure 5 (presented in subsubsection 5.1)

lemma *WhileSync*:
assumes *entails I* (*low-exp b*)
and $\vdash \{$*conj I* (*holds-forall b*)$\}$ *C* $\{I\}$
shows $\vdash \{$*conj I* (*low-exp b*)$\}$ *while-cond b C* $\{$*conj* (*disj I emp*) (*holds-forall* (*lnot b*))$\}$
⟨*proof*⟩

This result uses the following constructs:

- *Logic.conj A B* corresponds to the hyper-assertion $A \wedge B$.
- *holds-forall b* corresponds to $box(b)$.
- *low-exp b* corresponds to $low(b)$.
- *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 subsubsection 5.1)

theorem *IfSync*:

assumes *entails P* (*low-exp b*)
and ⊢ {*conj P* (*holds-forall b*)} *C1* {*Q*}
and ⊢ {*conj P* (*holds-forall* (*lnot b*))} *C2* {*Q*}
shows ⊢ {*P*} *if-then-else b C1 C2* {*Q*}
⟨*proof*⟩

This result uses the following construct:

- *if-then-else b C1 C2* is syntactic sugar for *stmt.If* (*Assume b* ;; *C1*) (*Assume* (*lnot b*) ;; *C2*).

Rule *While*−∀∗∃∗ from figure 5 (presented in subsubsection 5.2)

theorem *while-forall-exists*:
assumes ⊢ {*I*} *if-then b C* {*I*}
and ⊢ {*I*} *Assume* (*lnot b*) {*interp-assert Q*}
and *no-forall-state-after-existential Q*
shows ⊢ {*I*} *while-cond b C* {*interp-assert Q*}
⟨*proof*⟩

This result uses the following constructs:

- *if-then b C* is syntactic sugar for *stmt.If* (*Assume b* ;; *C*) (*Assume* (*lnot b*)).
- *no-forall-state-after-existential Q* holds iff there is no universal state quantifier ∀⟨-⟩ after any ∃ in Q.

Rule *While*−∃ from figure 5 (presented in subsubsection 5.3)

theorem *while-loop-exists*:
assumes ⋀*v*. ⊢ { (λ*S*. ∃φ∈*S*. *e* (*snd* φ) = *v* ∧ *b* (*snd* φ) ∧ *P* φ *S*) } *if-then b C* { (λ*S*. ∃φ∈*S*. *lt* (*e* (*snd* φ)) *v* ∧ *P* φ *S*) }
and ⋀φ. ⊢ { *P* φ } *while-cond b C* { *Q* φ }
and *wfP lt*
shows ⊢ { (λ*S*. ∃φ∈*S*. *P* φ *S*) } *while-cond b C* { (λ*S*. ∃φ∈*S*. *Q* φ *S*)}
⟨*proof*⟩

wfp lt in this result ensures that the binary operator *lt* is well-founded. *e* is a function of a program state, which must decrease after each iteration.

9.4 Appendix A: Technical Definitions Omitted from the Paper

The big-step semantics (figure 9) is defined as *single-sem*. We also use the notation ⟨*C*, σ⟩ → σ′. 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*:
$\exists H.\ (\forall C.\ hypersat\ C\ H \longleftrightarrow HL\ P\ C\ Q)$
⟨*proof*⟩

Proposition 2: Expressing HL in Hyper Hoare Logic

proposition *prop-2-expressing-HL-in-HHL*:
$HL\ P\ C\ Q \longleftrightarrow (hyper\text{-}hoare\text{-}triple\ (over\text{-}approx\ P)\ C\ (over\text{-}approx\ Q))$
⟨*proof*⟩

Proposition 3: CHL triples express hyperproperties

proposition *prop-3-CHL-is-hyperproperty*:
$hypersat\ C\ (CHL\text{-}hyperprop\ P\ Q) \longleftrightarrow CHL\ P\ C\ Q$
⟨*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 $CHL\ P\ C\ Q \longleftrightarrow \models \{encode\text{-}CHL\ from\text{-}nat\ x\ P\}\ C\ \{encode\text{-}CHL\ from\text{-}nat\ x\ Q\}$

⟨*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-hyperprop P Q* (*set-of-traces C*)
⟨*proof*⟩

Proposition 6: Expressing IL in Hyper Hoare Logic

proposition *prop-6-expressing-IL-in-HHL*:
IL P C Q $\longleftrightarrow$ (*hyper-hoare-triple* (*under-approx P*) *C* (*under-approx Q*))
⟨*proof*⟩

Proposition 7: k-IL triples express hyperproperties

proposition *prop-7-kIL-hyperproperties*:
hypersat C (*RIL-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 :: $('a \Rightarrow ('lvar \Rightarrow 'lval) \times ('pvar \Rightarrow 'pval))$ *set*) $x \wedge$ *not-free-var-of P c*

and *not-free-var-of Q x* ∧ *not-free-var-of Q c*
shows *RIL P C Q* ⟷ ⊨ {*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*) ⟷ *FU P C Q*
⟨*proof*⟩

Proposition 9: Expressing FU in Hyper Hoare Logic

proposition *prop-9-expressing-FU-in-HHL*:
FU P C Q ⟷ ⊨ {*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*) ⟷ *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* ⟷ ⊨ {*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 ⟷ *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* ∧ *injective fn1* ∧ *injective fn2*
and *t* ≠ *x*
and *injective* (*fn* :: *nat* ⇒ *'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* (λ-. *one*)) *C1*) (*Seq* (*Assign t* (λ-. *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 $\bigwedge \varphi 1\ (\varphi 2 :: ('a, 'b, 'c, 'd)\ state).\ fst\ \varphi 1 = fst\ \varphi 2 \wedge (\vdash \{\ (in\text{-}set\ \varphi 1 :: (('a, 'b, 'c, 'd)\ state)\ hyperassertion)\ \}\ C\ \{\ in\text{-}set\ \varphi 2\ \})$
$\Longrightarrow (\vdash \{\ (P\ \varphi 1 :: (('a, 'b, 'c, 'd)\ state)\ hyperassertion)\ \}\ C\ \{\ Q\ \varphi 2\ \})$
shows $\vdash \{\ ((\lambda S.\ \forall \varphi 1 \in S.\ P\ \varphi 1\ S) :: (('a, 'b, 'c, 'd)\ state)\ hyperassertion)\ \}\ C\ \{\ (\lambda S.\ \forall \varphi 2 \in S.\ Q\ \varphi 2\ S)\ \}$
⟨*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 $\vdash$ {*interp-assert F*} *C* {*interp-assert F*}
⟨*proof*⟩

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

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

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

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

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

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 $\vdash \{P\}\ C\ \{Q\}$
⟨*proof*⟩

proposition *rule-LUpdateSyntactic*:
assumes $\vdash$ { ($\lambda S.\ P\ S \wedge$ *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* }
⟨*proof*⟩

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

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

proposition *rule-True*:
$\vdash$ {*P*} *C* {λ-. *True*}
⟨*proof*⟩

proposition *rule-False*:
$\vdash$ { (λ-. *False*) } *C* {*Q*}
⟨*proof*⟩

proposition *rule-Empty*:
$\vdash$ { (λ*S*. *S* = {}) } *C* { (λ*S*. *S* = {}) }
⟨*proof*⟩

9.6.2 Appendix D.2: Examples

Example shown in figure 12. To see the actual proof, ctrl+click on ⟦$\models$ {*?P*} *?C1.0* {*has-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* ≠ *?two*; ⋀*x*. *?leq x x*⟧ ⟹ $\models$ {*?P*} *?C1.0* ;; *?C2.0* {*has-minimum ?y ?leq*}.

proposition *fig-12-composing-monotonicity-and-minimum*:
fixes *P* :: (((′*a* ⇒ ′*b*) × (′*c* ⇒ ′*d*)) *set* ⇒ *bool*)
fixes *i* :: ′*a*
fixes *x y* :: ′*c*
fixes *leq* :: ′*d* ⇒ ′*d* ⇒ *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* ⇒ ′*b*) × (′*c* ⇒ ′*d*)) *set* ⇒ *bool*)) } *C2* { *is-singleton* }
and *one* ≠ *two* — We use distinct logical values *one* and *two* to represent 1 and 2.
and ⋀*x*. *leq x x* — We assume that *leq* is a partial order, and thus that it

satisfies reflexivity.
shows $\vdash$ { *P* } *C1* ;; *C2* { *has-minimum y leq* }
⟨*proof*⟩

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

proposition *fig-13-composing-GNI-with-SNI*:
fixes *h* :: *'lvar*
fixes *l* :: *'pvar*
assumes $\vdash$ { (*low l* :: ((*'lvar*, *'lval*, *'pvar*, *'pval*) *state*) *hyperassertion*) } *C2* { *low l* }
and $\vdash$ { (*not-empty* :: ((*'lvar*, *'lval*, *'pvar*, *'pval*) *state*) *hyperassertion*) } *C2* { *not-empty* }
and $\vdash$ { (*low l* :: ((*'lvar*, *'lval*, *'pvar*, *'pval*) *state*) *hyperassertion*) } *C1* { *lGNI l h* }
shows $\vdash$ { (*low l* :: ((*'lvar*, *'lval*, *'pvar*, *'pval*) *state*) *hyperassertion*) } *C1*;; *C2* { *lGNI l h* }
⟨*proof*⟩

9.7 Appendix E: Termination-Based Reasoning

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

theorem *rule-Frame*:
assumes $\models$*TERM* {*P*} *C* {*Q*}
and *wr C* ∩ *fv F* = {}
and *wf-assertion F*
shows $\models$*TERM* {*conj P* (*interp-assert F*)} *C* {*conj Q* (*interp-assert F*)}
⟨*proof*⟩

theorem *rule-WhileSyncTerm*:
assumes $\models$*TERM* {*conj I* (λ*S*. ∀φ∈*S*. *b* (*snd* φ) ∧ *fst* φ *t* = *e* (*snd* φ))} *C* {*conj* (*conj I* (*low-exp b*)) (*e-smaller-than-t e t lt*)}
and *wfP lt*
and *not-fv-hyper t I*
shows $\models$*TERM* {*conj I* (*low-exp b*)} *while-cond b C* {*conj I* (*holds-forall* (*lnot b*))}
⟨*proof*⟩

9.8 Appendix H: Synchronous Reasoning over Different Branches

Proposition 14: Synchronous if rule

proposition *prop-14-synchronized-if-rule*:
assumes $\models$ {*P*} *C1* {*P1*}
and $\models$ {*P*} *C2* {*P2*}
and $\models$ {*combine from-nat x P1 P2*} *C* {*combine from-nat x R1 R2*}

and $\models \{R1\}\ C1'\ \{Q1\}$
and $\models \{R2\}\ C2'\ \{Q2\}$
and *not-free-var-hyper x P1*
and *not-free-var-hyper x P2*
and *from-nat 1* $\neq$ *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 $\models \{P\}$ *If* (*Seq C1* (*Seq C C1'*)) (*Seq C2* (*Seq C C2'*)) $\{$*join Q1 Q2*$\}$
⟨*proof*⟩

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 ofărelational ∀∃ 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.

[9] Peter W. O'Hearn. Incorrectness Logic. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi: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.

[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.