

An Under-Approximate Relational Logic

Toby Murray

February 6, 2026

Abstract

Recently, authors have proposed *under-approximate* logics for reasoning about programs [4, 2]. So far, all such logics have been confined to reasoning about individual program behaviours. Yet there exist many over-approximate *relational* logics for reasoning about pairs of programs and relating their behaviours.

We present the first under-approximate relational logic, for the simple imperative language IMP. We prove our logic is both sound and complete. Additionally, we show how reasoning in this logic can be decomposed into non-relational reasoning in an under-approximate Hoare logic, mirroring Beringer’s result for over-approximate relational logics. We illustrate the application of our logic on some small examples in which we provably demonstrate the presence of insecurity.

These proofs accompany a paper [3] that explains the results in more detail.

Contents

| | | |
|----------|--|----------|
| 1 | Under-Approximate Relational Judgement | 2 |
| 2 | Rules of the Logic | 2 |
| 3 | Simple Derived Rules | 3 |
| 4 | Soundness | 3 |
| 5 | Completeness | 5 |
| 6 | A Decomposition Principle: Proofs via Under-Approximate Hoare Logic | 7 |
| 7 | Deriving Proof Rules from Completeness | 8 |
| 8 | Examples | 8 |
| 8.1 | Some Derived Proof Rules | 8 |
| 8.2 | prog1 | 8 |

| | | |
|-----|--|----|
| 8.3 | More Derived Proof Rules for Examples | 9 |
| 8.4 | client0 | 10 |
| 8.5 | Derive a variant of the backwards variant rule | 11 |
| 8.6 | A variant of the frontier rule | 11 |
| 8.7 | client1 | 11 |
| 8.8 | client2 | 12 |

```

theory RelationalIncorrectness
  imports HOL-IMP.Big-Step
begin

```

1 Under-Approximate Relational Judgement

This is the relational analogue of O’Hearn’s [4] and de Vries & Koutavas’ [2] judgements.

Note that in our case it doesn’t really make sense to talk about “erroneous” states: the presence of an error can be seen only by the violation of a relation. Unlike O’Hearn, we cannot encode it directly into the semantics of our programs, without giving them a relational semantics. We use the standard big step semantics of IMP unchanged.

```

type-synonym rassn = state ⇒ state ⇒ bool

```

definition

```

  ir-valid :: rassn ⇒ com ⇒ com ⇒ rassn ⇒ bool

```

where

```

  ir-valid P c c' Q ≡ (∀ t t'. Q t t' ⟶ (∃ s s'. P s s' ∧ (c,s) ⇒ t ∧ (c',s') ⇒ t'))

```

2 Rules of the Logic

definition

```

  flip :: rassn ⇒ rassn

```

where

```

  flip P ≡ λs s'. P s' s

```

inductive

```

  ir-hoare :: rassn ⇒ com ⇒ com ⇒ rassn ⇒ bool

```

where

```

  ir-Skip: (∧ t t'. Q t t' ⟹ ∃ s'. P t s' ∧ (c',s') ⇒ t') ⟹

```

```

    ir-hoare P SKIP c' Q |

```

```

  ir-If-True: ir-hoare (λs s'. P s s' ∧ bval b s) c1 c' Q ⟹

```

```

    ir-hoare P (IF b THEN c1 ELSE c2) c' Q |

```

```

  ir-If-False: ir-hoare (λs s'. P s s' ∧ ¬ bval b s) c2 c' Q ⟹

```

```

    ir-hoare P (IF b THEN c1 ELSE c2) c' Q |

```

```

  ir-Seq1: ir-hoare P c c' Q ⟹ ir-hoare Q d SKIP R ⟹ ir-hoare P (Seq c d) c'

```

```

  R |

```

ir-Assign: $ir\text{-hoare } (\lambda t t'. \exists v. P (t(x := v)) t' \wedge (t x) = \text{aval } e (t(x := v))) \text{ SKIP } c' Q \implies$
 $ir\text{-hoare } P (\text{Assign } x e) c' Q \mid$
ir-While-False: $ir\text{-hoare } (\lambda s s'. P s s' \wedge \neg \text{bval } b s) \text{ SKIP } c' Q \implies$
 $ir\text{-hoare } P (\text{WHILE } b \text{ DO } c) c' Q \mid$
ir-While-True: $ir\text{-hoare } (\lambda s s'. P s s' \wedge \text{bval } b s) (c;; \text{WHILE } b \text{ DO } c) c' Q \implies$
 $ir\text{-hoare } P (\text{WHILE } b \text{ DO } c) c' Q \mid$
ir-While-backwards-frontier: $(\bigwedge n. ir\text{-hoare } (\lambda s s'. P n s s' \wedge \text{bval } b s) c \text{ SKIP } (P (\text{Suc } n))) \implies$
 $ir\text{-hoare } (\lambda s s'. \exists n. P n s s') (\text{WHILE } b \text{ DO } c) c' Q \implies$
 $ir\text{-hoare } (P \theta) (\text{WHILE } b \text{ DO } c) c' Q \mid$
ir-conseq: $ir\text{-hoare } P c c' Q \implies (\bigwedge s s'. P s s' \implies P' s s') \implies (\bigwedge s s'. Q' s s' \implies Q s s') \implies$
 $ir\text{-hoare } P' c c' Q' \mid$
ir-disj: $ir\text{-hoare } P_1 c c' Q_1 \implies ir\text{-hoare } P_2 c c' Q_2 \implies$
 $ir\text{-hoare } (\lambda s s'. P_1 s s' \vee P_2 s s') c c' (\lambda t t'. Q_1 t t' \vee Q_2 t t') \mid$
ir-sym: $ir\text{-hoare } (\text{flip } P) c c' (\text{flip } Q) \implies ir\text{-hoare } P c c' Q$

3 Simple Derived Rules

lemma *meh-simp[simp]*: $(\text{SKIP}, s') \Rightarrow t' = (s' = t')$
 $\langle \text{proof} \rangle$

lemma *ir-pre*: $ir\text{-hoare } P c c' Q \implies (\bigwedge s s'. P s s' \implies P' s s') \implies$
 $ir\text{-hoare } P' c c' Q$
 $\langle \text{proof} \rangle$

lemma *ir-post*: $ir\text{-hoare } P c c' Q \implies (\bigwedge s s'. Q' s s' \implies Q s s') \implies$
 $ir\text{-hoare } P c c' Q'$
 $\langle \text{proof} \rangle$

4 Soundness

lemma *Skip-ir-valid[intro]*:
 $(\bigwedge t t'. Q t t' \implies \exists s'. P t s' \wedge (c', s') \Rightarrow t') \implies ir\text{-valid } P \text{ SKIP } c' Q$
 $\langle \text{proof} \rangle$

lemma *sym-ir-valid[intro]*:
 $ir\text{-valid } (\text{flip } P) c' c (\text{flip } Q) \implies ir\text{-valid } P c c' Q$
 $\langle \text{proof} \rangle$

lemma *If-True-ir-valid[intro]*:
 $ir\text{-valid } (\lambda a c. P a c \wedge \text{bval } b a) c_1 c' Q \implies$
 $ir\text{-valid } P (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2) c' Q$
 $\langle \text{proof} \rangle$

lemma *If-False-ir-valid[intro]*:

ir-valid $(\lambda a c. P a c \wedge \neg \text{bval } b a) c_2 c' Q \implies$
ir-valid $P (IF b THEN c_1 ELSE c_2) c' Q$
 ⟨proof⟩

lemma *Seq1-ir-valid[intro]*:

ir-valid $P c c' Q \implies \text{ir-valid } Q d \text{ SKIP } R \implies \text{ir-valid } P (c;; d) c' R$
 ⟨proof⟩

lemma *Seq2-ir-valid[intro]*:

ir-valid $P c \text{ SKIP } Q \implies \text{ir-valid } Q d c' R \implies \text{ir-valid } P (c;; d) c' R$
 ⟨proof⟩

lemma *Seq-ir-valid[intro]*:

ir-valid $P c c' Q \implies \text{ir-valid } Q d d' R \implies \text{ir-valid } P (c;; d) (c';; d') R$
 ⟨proof⟩

lemma *Assign-blah[intro]*:

$t x = \text{aval } e (t(x := v))$
 $\implies (x ::= e, t(x := v)) \Rightarrow t$
 ⟨proof⟩

lemma *Assign-ir-valid[intro]*:

ir-valid $(\lambda t t'. \exists v. P (t(x := v)) t' \wedge (t x = \text{aval } e (t(x := v)))) \text{ SKIP } c' Q \implies$
ir-valid $P (\text{Assign } x e) c' Q$
 ⟨proof⟩

lemma *While-False-ir-valid[intro]*:

ir-valid $(\lambda s s'. P s s' \wedge \neg \text{bval } b s) \text{ SKIP } c' Q \implies$
ir-valid $P (\text{WHILE } b \text{ DO } c) c' Q$
 ⟨proof⟩

lemma *While-True-ir-valid[intro]*:

ir-valid $(\lambda s s'. P s s' \wedge \text{bval } b s) (\text{Seq } c (\text{WHILE } b \text{ DO } c)) c' Q \implies$
ir-valid $P (\text{WHILE } b \text{ DO } c) c' Q$
 ⟨proof⟩

lemma *While-backwards-frontier-ir-valid'*:

assumes *asm*: $\bigwedge n. \forall t t'. P (k + \text{Suc } n) t t' \longrightarrow$
 $(\exists s. P (k + n) s t' \wedge \text{bval } b s \wedge (c, s) \Rightarrow t)$
assumes *last*: $\forall t t'. Q t t' \longrightarrow (\exists s s'. (\exists n. P (k + n) s s') \wedge (\text{WHILE } b \text{ DO } c,$
 $s) \Rightarrow t \wedge (c', s') \Rightarrow t')$
assumes *post*: $Q t t'$
shows $\exists s s'. P k s s' \wedge (\text{WHILE } b \text{ DO } c, s) \Rightarrow t \wedge (c', s') \Rightarrow t'$
 ⟨proof⟩

lemma *While-backwards-frontier-ir-valid[intro]*:

$(\bigwedge n. \text{ir-valid } (\lambda s s'. P n s s' \wedge \text{bval } b s) c \text{ SKIP } (P (\text{Suc } n))) \implies$

$ir\text{-valid } (\lambda s s'. \exists n. P n s s') (WHILE\ b\ DO\ c)\ c'\ Q \implies$
 $ir\text{-valid } (P\ 0)\ (WHILE\ b\ DO\ c)\ c'\ Q$
 ⟨proof⟩

lemma *conseq-ir-valid*:

$ir\text{-valid } P\ c\ c'\ Q \implies (\bigwedge s s'. P\ s\ s' \implies P'\ s\ s') \implies (\bigwedge s s'. Q'\ s\ s' \implies Q\ s\ s')$
 \implies
 $ir\text{-valid } P'\ c\ c'\ Q'$
 ⟨proof⟩

lemma *disj-ir-valid[intro]*:

$ir\text{-valid } P_1\ c\ c'\ Q_1 \implies ir\text{-valid } P_2\ c\ c'\ Q_2 \implies$
 $ir\text{-valid } (\lambda s s'. P_1\ s\ s' \vee P_2\ s\ s')\ c\ c'\ (\lambda t t'. Q_1\ t\ t' \vee Q_2\ t\ t')$
 ⟨proof⟩

theorem *soundness*:

$ir\text{-hoare } P\ c\ c'\ Q \implies ir\text{-valid } P\ c\ c'\ Q$
 ⟨proof⟩

5 Completeness

lemma *ir-Skip-Skip[intro]*:

$ir\text{-hoare } P\ SKIP\ SKIP\ P$
 ⟨proof⟩

lemma *ir-hoare-Skip-Skip[simp]*:

$ir\text{-hoare } P\ SKIP\ SKIP\ Q = (\forall s s'. Q\ s\ s' \longrightarrow P\ s\ s')$
 ⟨proof⟩

lemma *ir-valid-Seq1*:

$ir\text{-valid } P\ (c1;;\ c2)\ c'\ Q \implies ir\text{-valid } P\ c1\ c'\ (\lambda t t'. \exists s s'. P\ s\ s' \wedge (c1, s) \Rightarrow t \wedge$
 $(c', s') \Rightarrow t' \wedge (\exists u. (c2, t) \Rightarrow u \wedge Q\ u\ t'))$
 ⟨proof⟩

lemma *ir-valid-Seq1'*:

$ir\text{-valid } P\ (c1;;\ c2)\ c'\ Q \implies ir\text{-valid } (\lambda t t'. \exists s s'. P\ s\ s' \wedge (c1, s) \Rightarrow t \wedge (c', s')$
 $\Rightarrow t' \wedge (\exists u. (c2, t) \Rightarrow u \wedge Q\ u\ t'))\ c2\ SKIP\ Q$
 ⟨proof⟩

lemma *ir-valid-track-history*:

$ir\text{-valid } P\ c\ c'\ Q \implies$
 $ir\text{-valid } P\ c\ c'\ (\lambda t t'. Q\ s\ s' \wedge (\exists s s'. P\ s\ s' \wedge (c, s) \Rightarrow t \wedge (c', s') \Rightarrow t'))$
 ⟨proof⟩

lemma *ir-valid-If*:

$ir\text{-valid } (\lambda s s'. P\ s\ s')\ (IF\ b\ THEN\ c1\ ELSE\ c2)\ c'\ Q \implies$
 $ir\text{-valid } (\lambda s s'. P\ s\ s' \wedge bval\ b\ s)\ c1\ c'\ (\lambda t t'. Q\ t\ t' \wedge (\exists s s'. P\ s\ s' \wedge (c1, s) \Rightarrow$
 $t \wedge (c', s') \Rightarrow t' \wedge bval\ b\ s)) \wedge$

$ir\text{-valid } (\lambda s s'. P s s' \wedge \neg bval b s) c2 c' (\lambda t t'. Q t t' \wedge (\exists s s'. P s s' \wedge (c2, s) \Rightarrow t \wedge (c', s') \Rightarrow t' \wedge \neg bval b s))$
 ⟨proof⟩

Inspired by the “ $p(n) = \{\sigma \mid \text{you can get back from } \sigma \text{ to some state in } p \text{ by executing } C \text{ backwards } n \text{ times}\}$ ” part of O'Hearn [4].

primrec *get-back* where

$get\text{-back } P b c 0 = (\lambda t t'. P t t') \mid$
 $get\text{-back } P b c (Suc n) = (\lambda t t'. \exists s. (c, s) \Rightarrow t \wedge bval b s \wedge get\text{-back } P b c n s t')$

lemma *ir-valid-get-back*:

$ir\text{-valid } (get\text{-back } P b c (Suc k)) (WHILE b DO c) c' Q \Longrightarrow$
 $ir\text{-valid } (get\text{-back } P b c k) (WHILE b DO c) c' Q$
 ⟨proof⟩

lemma *ir-valid-While1*:

$ir\text{-valid } (get\text{-back } P b c k) (WHILE b DO c) c' Q \Longrightarrow$
 $(ir\text{-valid } (\lambda s s'. get\text{-back } P b c k s s' \wedge bval b s) c SKIP (\lambda t t'. get\text{-back } P b c (Suc k) t t' \wedge (\exists u u'. (WHILE b DO c, t) \Rightarrow u \wedge (c', t') \Rightarrow u' \wedge Q u u')))$
 ⟨proof⟩

lemma *ir-valid-While3*:

$ir\text{-valid } (get\text{-back } P b c k) (WHILE b DO c) c' Q \Longrightarrow$
 $(ir\text{-valid } (\lambda s s'. get\text{-back } P b c k s s' \wedge bval b s) c c' (\lambda t t'. (\exists s'. (c', s') \Rightarrow t' \wedge get\text{-back } P b c (Suc k) t s' \wedge (\exists u. (WHILE b DO c, t) \Rightarrow u \wedge Q u t'))))$
 ⟨proof⟩

lemma *ir-valid-While2*:

$ir\text{-valid } P (WHILE b DO c) c' Q \Longrightarrow$
 $ir\text{-valid } (\lambda s s'. P s s' \wedge \neg bval b s) SKIP c' (\lambda t t'. Q t t' \wedge (\exists s'. (c', s') \Rightarrow t' \wedge P t s' \wedge \neg bval b t))$
 ⟨proof⟩

lemma *assign-upd-blah*:

$(\lambda a. \text{if } a = x1 \text{ then } s x1 \text{ else } (s(x1 := aval x2 s))) a = s$
 ⟨proof⟩

lemma *Assign-complete*:

assumes v : $ir\text{-valid } P (x1 ::= x2) c' Q$
assumes q : $Q t t'$
shows $\exists s'. (\exists v. P (t(x1 := v)) s' \wedge t x1 = aval x2 (t(x1 := v))) \wedge (c', s') \Rightarrow t'$
 ⟨proof⟩

lemmas $ir\text{-Skip-sym} = ir\text{-sym}[OF ir\text{-Skip}, \text{simplified flip-def}]$

theorem *completeness*:

$ir\text{-}valid\ P\ c\ c'\ Q \implies ir\text{-}hoare\ P\ c\ c'\ Q$
 ⟨proof⟩

6 A Decomposition Principle: Proofs via Under-Approximate Hoare Logic

We show the under-approximate analogue holds for Beringer's [1] decomposition principle for over-approximate relational logic.

definition

$decomp :: rassn \Rightarrow com \Rightarrow com \Rightarrow rassn \Rightarrow rassn$ **where**
 $decomp\ P\ c\ c'\ Q \equiv \lambda t\ s'. \exists s\ t'. P\ s\ s' \wedge (c,s) \Rightarrow t \wedge (c',s') \Rightarrow t' \wedge Q\ t\ t'$

lemma *ir-valid-decomp1*:

$ir\text{-}valid\ P\ c\ c'\ Q \implies ir\text{-}valid\ P\ c\ SKIP\ (decomp\ P\ c\ c'\ Q) \wedge ir\text{-}valid\ (decomp\ P\ c\ c'\ Q)\ SKIP\ c'\ Q$
 ⟨proof⟩

lemma *ir-valid-decomp2*:

$ir\text{-}valid\ P\ c\ SKIP\ R \wedge ir\text{-}valid\ R\ SKIP\ c'\ Q \implies ir\text{-}valid\ P\ c\ c'\ Q$
 ⟨proof⟩

lemma *ir-valid-decomp*:

$ir\text{-}valid\ P\ c\ c'\ Q = (ir\text{-}valid\ P\ c\ SKIP\ (decomp\ P\ c\ c'\ Q) \wedge ir\text{-}valid\ (decomp\ P\ c\ c'\ Q)\ SKIP\ c'\ Q)$
 ⟨proof⟩

Completeness with soundness means we can prove proof rules about *ir-hoare* in terms of *ir-valid*.

lemma *ir-to-Skip*:

$ir\text{-}hoare\ P\ c\ c'\ Q = (ir\text{-}hoare\ P\ c\ SKIP\ (decomp\ P\ c\ c'\ Q) \wedge ir\text{-}hoare\ (decomp\ P\ c\ c'\ Q)\ SKIP\ c'\ Q)$
 ⟨proof⟩

O'Hearn's under-approximate Hoare triple, for the "ok" case (since that is the only case we have) This is also likely the same as from the "Reverse Hoare Logic" paper (SEFM).

type-synonym $assn = state \Rightarrow bool$

definition

$ohearn :: assn \Rightarrow com \Rightarrow assn \Rightarrow bool$
where
 $ohearn\ P\ c\ Q \equiv (\forall t. Q\ t \longrightarrow (\exists s. P\ s \wedge (c,s) \Rightarrow t))$

lemma *fold-ohearn1*:

$ir\text{-}valid\ P\ c\ SKIP\ Q = (\forall t'. ohearn\ (\lambda t. P\ t\ t')\ c\ (\lambda t. Q\ t\ t'))$
 ⟨proof⟩

lemma *fold-ohearn2*:

$$ir\text{-}valid\ P\ SKIP\ c'\ Q = (\forall t. ohearn\ (P\ t)\ c'\ (Q\ t))$$

<proof>

theorem *relational-via-hoare*:

$$ir\text{-}hoare\ P\ c\ c'\ Q = ((\forall t'. ohearn\ (\lambda t. P\ t\ t')\ c\ (\lambda t. decomp\ P\ c\ c'\ Q\ t\ t')) \wedge (\forall t. ohearn\ (decomp\ P\ c\ c'\ Q\ t)\ c'\ (Q\ t)))$$

<proof>

7 Deriving Proof Rules from Completeness

Note that we can more easily derive proof rules sometimes by appealing to the corresponding properties of *ir-valid* than from the proof rules directly.

We see more examples of this later on when we consider examples.

lemma *ir-Seq2*:

$$ir\text{-}hoare\ P\ c\ SKIP\ Q \implies ir\text{-}hoare\ Q\ d\ c'\ R \implies ir\text{-}hoare\ P\ (Seq\ c\ d)\ c'\ R$$

<proof>

lemma *ir-Seq*:

$$ir\text{-}hoare\ P\ c\ c'\ Q \implies ir\text{-}hoare\ Q\ d\ d'\ R \implies ir\text{-}hoare\ P\ (Seq\ c\ d)\ (Seq\ c'\ d')\ R$$

<proof>

8 Examples

8.1 Some Derived Proof Rules

First derive some proof rules – here not by appealing to completeness but just using the existing rules

lemma *ir-If-True-False*:

$$ir\text{-}hoare\ (\lambda s\ s'. P\ s\ s' \wedge bval\ b\ s \wedge \neg\ bval\ b'\ s')\ c_1\ c_2'\ Q \implies ir\text{-}hoare\ P\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ (IF\ b'\ THEN\ c_1'\ ELSE\ c_2')\ Q$$

<proof>

lemma *ir-Assign-Assign*:

$$ir\text{-}hoare\ P\ (x\ ::= e)\ (x'\ ::= e')\ (\lambda t\ t'. \exists v\ v'. P\ (t(x := v))\ (t'(x' := v))) \wedge t\ x = aval\ e\ (t(x := v)) \wedge t'\ x' = aval\ e'\ (t'(x' := v)))$$

<proof>

8.2 prog1

A tiny insecure program. Note that we only need to reason on one path through this program to detect that it is insecure.

abbreviation *low-eq* :: *rassn where low-eq* $s\ s' \equiv s\ ''low'' = s'\ ''low''$

abbreviation *low-neq* :: *rassn where low-neq* $s\ s' \equiv \neg\ low\text{-}eq\ s\ s'$

definition $prog1 :: com$ **where**

$prog1 \equiv (IF (Less (N 0) (V "x")) THEN (Assign "low" (N 1)) ELSE (Assign "low" (N 0)))$

We prove that $prog1$ is definitely insecure. To do that, we need to find some non-empty post-relation that implies insecurity. The following property encodes the idea that the post-relation is non-empty, i.e. represents a feasible pair of execution paths.

definition

$nontrivial :: rassn \Rightarrow bool$

where

$nontrivial Q \equiv (\exists t t'. Q t t')$

Note the property we prove here explicitly encodes the fact that the postcondition can be anything that implies insecurity, i.e. implies $\lambda s s'. s \text{ "low" } \neq s' \text{ "low"}$. In particular we should not necessarily expect it to cover the entirety of all states that satisfy $\lambda s s'. s \text{ "low" } \neq s' \text{ "low"}$.

Also note that we also have to prove that the postcondition is non-trivial. This is necessary to make sure that the violation we find is not an infeasible path.

lemma $prog1$:

$\exists Q. ir\text{-hoare } low\text{-eq } prog1 \ prog1 \ Q \wedge (\forall s s'. Q s s' \longrightarrow low\text{-neq } s s') \wedge nontrivial Q$
 $\langle proof \rangle$

8.3 More Derived Proof Rules for Examples

definition $BEq :: aexp \Rightarrow aexp \Rightarrow bexp$ **where**

$BEq a b \equiv And (Less a (Plus b (N 1))) (Less b (Plus a (N 1)))$

lemma $BEq\text{-aval}[simp]$:

$bval (BEq a b) s = ((aval a s) = (aval b s))$
 $\langle proof \rangle$

lemma $ir\text{-If-True-True}$:

$ir\text{-hoare } (\lambda s s'. P s s' \wedge bval b s \wedge bval b' s') c_1 c_1' Q_1 \Longrightarrow$
 $ir\text{-hoare } P (IF b THEN c_1 ELSE c_2) (IF b' THEN c_1' ELSE c_2') (\lambda t t'. Q_1 t t')$
 $\langle proof \rangle$

lemma $ir\text{-If-False-False}$:

$ir\text{-hoare } (\lambda s s'. P s s' \wedge \neg bval b s \wedge \neg bval b' s') c_2 c_2' Q_2 \Longrightarrow$
 $ir\text{-hoare } P (IF b THEN c_1 ELSE c_2) (IF b' THEN c_1' ELSE c_2') (\lambda t t'. Q_2 t t')$
 $\langle proof \rangle$

lemma $ir\text{-If}'$:

$ir\text{-hoare } (\lambda s s'. P s s' \wedge bval b s \wedge bval b' s') c_1 c_1' Q_1 \Longrightarrow$
 $ir\text{-hoare } (\lambda s s'. P s s' \wedge \neg bval b s \wedge \neg bval b' s') c_2 c_2' Q_2 \Longrightarrow$

$ir\text{-hoare } P \text{ (IF } b \text{ THEN } c_1 \text{ ELSE } c_2) \text{ (IF } b' \text{ THEN } c_1' \text{ ELSE } c_2') \text{ (}\lambda t t'. Q_1 t t' \vee Q_2 t t')$
 $\langle proof \rangle$

lemma *ir-While-triv*:

$ir\text{-hoare } (\lambda s s'. P s s' \wedge \neg bval b s \wedge \neg bval b' s') \text{ SKIP SKIP } Q_2 \implies$
 $ir\text{-hoare } P \text{ (WHILE } b \text{ DO } c) \text{ (WHILE } b' \text{ DO } c') \text{ (}\lambda s s'. (Q_2 s s')$
 $\langle proof \rangle$

lemma *ir-While'*:

$ir\text{-hoare } (\lambda s s'. P s s' \wedge bval b s \wedge bval b' s') \text{ (}c'; \text{WHILE } b \text{ DO } c) \text{ (}c'; \text{WHILE } b' \text{ DO } c') Q_1 \implies$
 $ir\text{-hoare } (\lambda s s'. P s s' \wedge \neg bval b s \wedge \neg bval b' s') \text{ SKIP SKIP } Q_2 \implies$
 $ir\text{-hoare } P \text{ (WHILE } b \text{ DO } c) \text{ (WHILE } b' \text{ DO } c') \text{ (}\lambda s s'. (Q_1 s s' \vee Q_2 s s')$
 $\langle proof \rangle$

8.4 client0

definition *low-eq-strong where*

$low\text{-eq-strong } s s' \equiv (s \text{ "high" } \neq s' \text{ "high"}) \wedge low\text{-eq } s s'$

lemma *low-eq-strong-upd[simp]*:

$var \neq \text{"high"} \wedge var \neq \text{"low"} \implies low\text{-eq-strong}(s(var := v)) (s'(var := v')) =$
 $low\text{-eq-strong } s s'$
 $\langle proof \rangle$

A variation on client0 from O'Hearn [4]: how to reason about loops via a single unfolding

definition *client0 :: com where*

$client0 \equiv (\text{Assign "x" (N 0)});;$
 $(\text{While (Less (N 0) (V "n"))$
 $((\text{Assign "x" (Plus (V "x") (V "n"))});$
 $(\text{Assign "n" (V "nondet"))});;$
 $(\text{If (BEq (V "x") (N 2000000)) (Assign "low" (V "high")) SKIP}))$

lemma *client0*:

$\exists Q. ir\text{-hoare } low\text{-eq } client0 \text{ client0 } Q \wedge (\forall s s'. Q s s' \longrightarrow low\text{-neg } s s') \wedge nontrivial$
 Q
 $\langle proof \rangle$

lemma *ir-While-backwards*:

$(\bigwedge n. ir\text{-hoare } (\lambda s s'. P n s s' \wedge bval b s) c \text{ SKIP } (P (Suc n))) \implies$
 $ir\text{-hoare } (\lambda s s'. \exists n. P n s s' \wedge \neg bval b s) \text{ SKIP } c' Q \implies$
 $ir\text{-hoare } (P 0) \text{ (WHILE } b \text{ DO } c) c' Q$
 $\langle proof \rangle$

8.5 Derive a variant of the backwards variant rule

Here we appeal to completeness again to derive this rule from the corresponding property about *ir-valid*.

8.6 A variant of the frontier rule

Again we derive this rule by appealing to completeness and the corresponding property of *ir-valid*

lemma *While-backwards-frontier-both-ir-valid'*:

assumes *asm*: $\bigwedge n. \forall t t'. P (k + \text{Suc } n) t t' \longrightarrow$
 $(\exists s s'. P (k + n) s s' \wedge \text{bval } b s \wedge \text{bval } b' s' \wedge (c, s) \Rightarrow t \wedge (c',$
 $s') \Rightarrow t')$
assumes *last*: $\forall t t'. Q t t' \longrightarrow (\exists s s'. (\exists n. P (k + n) s s') \wedge (\text{WHILE } b \text{ DO } c,$
 $s) \Rightarrow t \wedge (\text{WHILE } b' \text{ DO } c', s') \Rightarrow t')$
assumes *post*: $Q t t'$
shows $\exists s s'. P k s s' \wedge (\text{WHILE } b \text{ DO } c, s) \Rightarrow t \wedge (\text{WHILE } b' \text{ DO } c', s') \Rightarrow t'$
<proof>

lemma *While-backwards-frontier-both-ir-valid[intro]*:

$(\bigwedge n. \text{ir-valid } (\lambda s s'. P n s s' \wedge \text{bval } b s \wedge \text{bval } b' s') c c' (P (\text{Suc } n))) \Longrightarrow$
 $\text{ir-valid } (\lambda s s'. \exists n. P n s s') (\text{WHILE } b \text{ DO } c) (\text{WHILE } b' \text{ DO } c') Q \Longrightarrow$
 $\text{ir-valid } (P 0) (\text{WHILE } b \text{ DO } c) (\text{WHILE } b' \text{ DO } c') (\lambda s s'. Q s s')$
<proof>

lemma *ir-While-backwards-frontier-both*:

$\llbracket \bigwedge n. \text{ir-hoare } (\lambda s s'. P n s s' \wedge \text{bval } b s \wedge \text{bval } b' s') c c' (P (\text{Suc } n));$
 $\text{ir-hoare } (\lambda s s'. \exists n. P n s s') (\text{WHILE } b \text{ DO } c) (\text{WHILE } b' \text{ DO } c') Q \rrbracket$
 $\Longrightarrow \text{ir-hoare } (P 0) (\text{WHILE } b \text{ DO } c) (\text{WHILE } b' \text{ DO } c') (\lambda s s'. Q s s')$
<proof>

The following rule then follows easily as a special case

lemma *ir-While-backwards-both*:

$(\bigwedge n. \text{ir-hoare } (\lambda s s'. P n s s' \wedge \text{bval } b s \wedge \text{bval } b' s') c c' (P (\text{Suc } n))) \Longrightarrow$
 $\text{ir-hoare } (P 0) (\text{WHILE } b \text{ DO } c) (\text{WHILE } b' \text{ DO } c') (\lambda s s'. \exists n.$
 $P n s s' \wedge \neg \text{bval } b s \wedge \neg \text{bval } b' s')$
<proof>

8.7 client1

An example roughly equivalent to *client1* from O'Hearn [4]0

In particular we use the backwards variant rule to reason about the loop.

definition *client1* :: *com where*

client1 \equiv $(\text{Assign } "x" (N 0));;$
 $(\text{While } (\text{Less } (V "x") (V "n"))$
 $((\text{Assign } "x" (\text{Plus } (V "x") (N 1)))));;$
 $(\text{If } (\text{BEq } (V "x") (N 2000000)) (\text{Assign } "low" (V "high")) \text{SKIP}))$

lemma *client1*:

$\exists Q. \text{ir-hoare low-eq client1 client1 } Q \wedge (\forall s s'. Q s s' \longrightarrow \text{low-neq } s s') \wedge \text{nontrivial } Q$
 ⟨proof⟩

8.8 client2

An example akin to *client2* from O’Hearn [4].

Note that this example is carefully written to show use of the frontier rule first to reason up to the broken loop iteration, and then we unfold the loop at that point to reason about the broken iteration, and then use the plain backwards variant rule to reason over the remainder of the loop.

definition *client2* :: *com* **where**

```

client2 ≡ (Assign "x" (N 0));
           (While (Less (V "x") (N 4000000))
                 ((Assign "x" (Plus (V "x") (N 1))));
                 (If (BEq (V "x") (N 2000000)) (Assign "low" (V "high"))
                    SKIP))
           )

```

lemma *client2*:

$\exists Q. \text{ir-hoare low-eq client2 client2 } Q \wedge (\forall s s'. Q s s' \longrightarrow \text{low-neq } s s') \wedge \text{nontrivial } Q$
 ⟨proof⟩

end

References

- [1] L. Beringer. Relational decomposition. In *International Conference on Interactive Theorem Proving (ITP)*, pages 39–54. Springer, 2011.
- [2] E. De Vries and V. Koutavas. Reverse hoare logic. In *International Conference on Software Engineering and Formal Methods (SEFM)*, pages 155–171. Springer, 2011.
- [3] T. Murray. An under-approximate relational logic: Heralding logics of insecurity, incorrect implementation & more. arXiv eprint arXiv:2003.04791 [cs.LO], 2020. <https://arxiv.org/abs/2003.04791>.
- [4] P. W. O’Hearn. Incorrectness logic. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2019.