

An Under-Approximate Relational Logic

Toby Murray

March 17, 2025

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	10
7	Deriving Proof Rules from Completeness	11
8	Examples	11
8.1	Some Derived Proof Rules	11
8.2	prog1	12

8.3	More Derived Proof Rules for Examples	13
8.4	client0	14
8.5	Derive a variant of the backwards variant rule	15
8.6	A variant of the frontier rule	15
8.7	client1	16
8.8	client2	17

```

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: $(\bigwedge n. ir\text{-hoare } (\lambda s s'. P n s s' \wedge \text{bval } b s) c \text{ SKIP } (P (\text{Suc } n))) \implies$
 $ir\text{-hoare } (\lambda s s'. \exists n. P n s s') (\text{WHILE } b \text{ DO } c) c' Q \implies$
 $ir\text{-hoare } (P \ 0) (\text{WHILE } b \text{ DO } c) c' Q \mid$
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*]: $(\text{SKIP}, s') \Rightarrow t' = (s' = t')$
by *auto*

lemma *ir-pre*: $ir\text{-hoare } P c c' Q \implies (\bigwedge s s'. P s s' \implies P' s s') \implies$
 $ir\text{-hoare } P' c c' Q$
by(*erule ir-conseq, blast+*)

lemma *ir-post*: $ir\text{-hoare } P c c' Q \implies (\bigwedge s s'. Q' s s' \implies Q s s') \implies$
 $ir\text{-hoare } P c c' Q'$
by(*erule ir-conseq, blast+*)

4 Soundness

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

lemma *sym-ir-valid*[*intro*]:
 $ir\text{-valid } (\text{flip } P) c' c (\text{flip } Q) \implies ir\text{-valid } P c c' Q$
by(*fastforce simp: ir-valid-def flip-def*)

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

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

$ir\text{-valid } (\lambda a c. P a c \wedge \neg bval b a) c_2 c' Q \implies$
 $ir\text{-valid } P (IF b THEN c_1 ELSE c_2) c' Q$
by(fastforce simp: ir-valid-def)

lemma Seq1-ir-valid[intro]:
 $ir\text{-valid } P c c' Q \implies ir\text{-valid } Q d SKIP R \implies ir\text{-valid } P (c;; d) c' R$
by(fastforce simp: ir-valid-def)

lemma Seq2-ir-valid[intro]:
 $ir\text{-valid } P c SKIP Q \implies ir\text{-valid } Q d c' R \implies ir\text{-valid } P (c;; d) c' R$
by(fastforce simp: ir-valid-def)

lemma Seq-ir-valid[intro]:
 $ir\text{-valid } P c c' Q \implies ir\text{-valid } Q d d' R \implies ir\text{-valid } P (c;; d) (c';; d') R$
by(fastforce simp: ir-valid-def)

lemma Assign-blah[intro]:
 $t x = aval e (t(x := v))$
 $\implies (x ::= e, t(x := v)) \Rightarrow t$
using Assign
by (metis fun-upd-triv fun-upd-upd)

lemma Assign-ir-valid[intro]:
 $ir\text{-valid } (\lambda t t'. \exists v. P (t(x := v)) t' \wedge (t x = aval e (t(x := v)))) SKIP c' Q \implies$
 $ir\text{-valid } P (Assign x e) c' Q$
by(fastforce simp: ir-valid-def)

lemma While-False-ir-valid[intro]:
 $ir\text{-valid } (\lambda s s'. P s s' \wedge \neg bval b s) SKIP c' Q \implies$
 $ir\text{-valid } P (WHILE b DO c) c' Q$
by(fastforce simp: ir-valid-def)

lemma While-True-ir-valid[intro]:
 $ir\text{-valid } (\lambda s s'. P s s' \wedge bval b s) (Seq c (WHILE b DO c)) c' Q \implies$
 $ir\text{-valid } P (WHILE b DO c) c' Q$
by(clarsimp simp: ir-valid-def, blast)

lemma While-backwards-frontier-ir-valid':
assumes asm: $\bigwedge n. \forall t t'. P (k + Suc n) t t' \longrightarrow$
 $(\exists s. P (k + n) s t' \wedge bval b s \wedge (c, s) \Rightarrow t)$
assumes last: $\forall t t'. Q t t' \longrightarrow (\exists s s'. (\exists n. P (k + n) s s') \wedge (WHILE b DO c,$
 $s) \Rightarrow t \wedge (c', s') \Rightarrow t')$
assumes post: $Q t t'$
shows $\exists s s'. P k s s' \wedge (WHILE b DO c, s) \Rightarrow t \wedge (c', s') \Rightarrow t'$
proof –
from post last **obtain** $s s' n$ **where**
 $P (k + n) s s' (WHILE b DO c, s) \Rightarrow t (c', s') \Rightarrow t'$
by blast

```

with asm show ?thesis
proof(induction n arbitrary: k t t')
  case 0
  then show ?case
    by (metis WhileFalse WhileTrue add.right-neutral)
  next
  case (Suc n)
  from Suc
  obtain r r' where final-iteration: P (Suc k) r r' (WHILE b DO c, r) ⇒ t (c',
r') ⇒ t'
    by (metis add-Suc-shift)
  from final-iteration(1) obtain q q' where
    P k q r' ∧ bval b q ∧ (c, q) ⇒ r
  by (metis Nat.add-0-right Suc.prem(1) plus-1-eq-Suc semiring-normalization-rules(24))
  with final-iteration show ?case by blast
qed
qed

```

lemma *While-backwards-frontier-ir-valid[intro]:*
 $(\bigwedge n. \text{ir-valid } (\lambda s s'. P n s s' \wedge \text{bval } b s) c \text{ SKIP } (P (Suc n))) \implies$
 $\text{ir-valid } (\lambda s s'. \exists n. P n s s') (\text{WHILE } b \text{ DO } c) c' Q \implies$
 $\text{ir-valid } (P 0) (\text{WHILE } b \text{ DO } c) c' Q$
by(*auto simp: meh-simp ir-valid-def intro: While-backwards-frontier-ir-valid'*)

lemma *conseq-ir-valid:*
 $\text{ir-valid } P c c' Q \implies (\bigwedge s s'. P s s' \implies P' s s') \implies (\bigwedge s s'. Q' s s' \implies Q s s')$
 \implies
 $\text{ir-valid } P' c c' Q'$
by(*clarsimp simp: ir-valid-def, blast*)

lemma *disj-ir-valid[intro]:*
 $\text{ir-valid } P_1 c c' Q_1 \implies \text{ir-valid } P_2 c c' Q_2 \implies$
 $\text{ir-valid } (\lambda s s'. P_1 s s' \vee P_2 s s') c c' (\lambda t t'. Q_1 t t' \vee Q_2 t t')$
by(*fastforce simp: ir-valid-def*)

theorem *soundness:*
 $\text{ir-hoare } P c c' Q \implies \text{ir-valid } P c c' Q$
apply(*induction rule: ir-hoare.induct*)
apply(*blast intro!: Skip-ir-valid*)
by (*blast intro: conseq-ir-valid*)+

5 Completeness

lemma *ir-Skip-Skip[intro]:*
 $\text{ir-hoare } P \text{ SKIP SKIP } P$
by(*rule ir-Skip, simp*)

lemma *ir-hoare-Skip-Skip*[simp]:
ir-hoare $P \text{ SKIP SKIP } Q = (\forall s s'. Q s s' \longrightarrow P s s')$
using *soundness ir-valid-def meh-simp ir-conseq ir-Skip by metis*

lemma *ir-valid-Seq1*:
ir-valid $P (c1;; c2) c' Q \Longrightarrow \text{ir-valid } P c1 c' (\lambda t t'. \exists s s'. P s s' \wedge (c1, s) \Rightarrow t \wedge (c', s') \Rightarrow t' \wedge (\exists u. (c2, t) \Rightarrow u \wedge Q u t'))$
by(*auto simp: ir-valid-def*)

lemma *ir-valid-Seq1'*:
ir-valid $P (c1;; c2) c' Q \Longrightarrow \text{ir-valid } (\lambda t t'. \exists s s'. P s s' \wedge (c1, s) \Rightarrow t \wedge (c', s') \Rightarrow t' \wedge (\exists u. (c2, t) \Rightarrow u \wedge Q u t')) c2 \text{ SKIP } Q$
by(*clarsimp simp: ir-valid-def, meson SeqE*)

lemma *ir-valid-track-history*:
ir-valid $P c c' Q \Longrightarrow$
ir-valid $P c c' (\lambda t t'. Q s s' \wedge (\exists s s'. P s s' \wedge (c, s) \Rightarrow t \wedge (c', s') \Rightarrow t'))$
by(*auto simp: ir-valid-def*)

lemma *ir-valid-If*:
ir-valid $(\lambda s s'. P s s') (IF b THEN c1 ELSE c2) c' Q \Longrightarrow$
ir-valid $(\lambda s s'. P s s' \wedge \text{bval } b s) c1 c' (\lambda t t'. Q t t' \wedge (\exists s s'. P s s' \wedge (c1, s) \Rightarrow t \wedge (c', s') \Rightarrow t' \wedge \text{bval } b s)) \wedge$
ir-valid $(\lambda s s'. P s s' \wedge \neg \text{bval } b s) c2 c' (\lambda t t'. Q t t' \wedge (\exists s s'. P s s' \wedge (c2, s) \Rightarrow t \wedge (c', s') \Rightarrow t' \wedge \neg \text{bval } b s))$
by(*clarsimp simp: ir-valid-def, blast*)

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

lemma *ir-valid-get-back*:
ir-valid $(\text{get-back } P b c (\text{Suc } k)) (\text{WHILE } b \text{ DO } c) c' Q \Longrightarrow$
ir-valid $(\text{get-back } P b c k) (\text{WHILE } b \text{ DO } c) c' Q$
proof(*induct k*)
case 0
then show ?case **by**(*clarsimp simp: ir-valid-def, blast*)
next
case (*Suc k*)
then show ?case **using** *WhileTrue get-back.simps(2) ir-valid-def by (smt (verit))*
qed

lemma *ir-valid-While1*:
ir-valid $(\text{get-back } P b c k) (\text{WHILE } b \text{ DO } c) c' Q \Longrightarrow$

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

proof (*subst ir-valid-def, clarsimp*)

fix $t \ t' \ s \ u \ u'$

assume *ir-valid* (*get-back* $P \ b \ c \ k$) (*WHILE* $b \ \text{DO } c$) $c' \ Q$

(*WHILE* $b \ \text{DO } c, t$) $\Rightarrow u$

$(c, s) \Rightarrow t$

$(c', t') \Rightarrow u'$

$Q \ u \ u'$

bval $b \ s$

get-back $P \ b \ c \ k \ s \ t'$

thus $\exists s. \text{get-back } P \ b \ c \ k \ s \ t' \wedge \text{bval } b \ s \wedge (c, s) \Rightarrow t$

proof(*induction k arbitrary: t t' s u u'*)

case 0

then show *?case*

by *auto*

next

case (*Suc* k)

show *?case*

using *Suc.prem*s(3) *Suc.prem*s(6) *Suc.prem*s(7) **by** *blast*

qed

qed

lemma *ir-valid-While3*:

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

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

apply(*subst ir-valid-def, clarsimp*)

proof –

fix $t \ t' \ s' \ s \ u$

assume *ir-valid* (*get-back* $P \ b \ c \ k$) (*WHILE* $b \ \text{DO } c$) $c' \ Q$

(*WHILE* $b \ \text{DO } c, t$) $\Rightarrow u$

$(c, s) \Rightarrow t$

$(c', s') \Rightarrow t'$

$Q \ u \ t'$

bval $b \ s$

get-back $P \ b \ c \ k \ s \ s'$

thus $\exists s s'. \text{get-back } P \ b \ c \ k \ s \ s' \wedge \text{bval } b \ s \wedge (c, s) \Rightarrow t \wedge (c', s') \Rightarrow t'$

proof(*induction k arbitrary: t t' s' s u*)

case 0

then show *?case*

by *auto*

next

case (*Suc* k)

show *?case*

using *Suc.prem*s(3) *Suc.prem*s(4) *Suc.prem*s(6) *Suc.prem*s(7) **by** *blast*

qed

qed

lemma *ir-valid-While2*:

ir-valid P (*WHILE* b *DO* c) $c' Q \implies$
ir-valid $(\lambda s s'. P s s' \wedge \neg \text{bval } b s)$ *SKIP* $c' (\lambda t t'. Q t t' \wedge (\exists s'. (c', s') \Rightarrow t' \wedge P t s' \wedge \neg \text{bval } b t))$
by(*clarsimp simp: ir-valid-def, blast*)

lemma *assign-upd-blah*:

$(\lambda a. \text{if } a = x1 \text{ then } s x1 \text{ else } (s(x1 := \text{aval } x2 s))) a = s$
by(*rule ext, auto*)

lemma *Assign-complete*:

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

proof –

from v **and** q **obtain** $s s'$ **where** a : $P s s' \wedge (x1 ::= x2, s) \Rightarrow t \wedge (c', s') \Rightarrow t'$
using *ir-valid-def* **by** (*smt (verit)*)
hence $P (\lambda a. \text{if } a = x1 \text{ then } s x1 \text{ else } (s(x1 := \text{aval } x2 s))) a$ $s' \wedge \text{aval } x2 s = \text{aval } x2 (s(x1 := s x1))$
using *assign-upd-blah*
by *simp*
thus *?thesis*
using *assign-upd-blah a*
by (*metis AssignE fun-upd-same fun-upd-triv fun-upd-upd*)

qed

lemmas *ir-Skip-sym = ir-sym[OF ir-Skip, simplified flip-def]*

theorem *completeness*:

ir-valid $P c c' Q \implies \text{ir-hoare } P c c' Q$
proof(*induct c arbitrary: P c' Q*)
case *SKIP*
show *?case*
apply(*rule ir-Skip*)
using *SKIP* **by**(*fastforce simp: ir-valid-def*)
next
case (*Assign* $x1 x2$)
show *?case*
apply(*rule ir-Assign*)
apply(*rule ir-Skip*)
using *Assign-complete Assign* **by** *blast*
next
case (*Seq* $c1 c2$)
have a : *ir-hoare* $P c1 c' (\lambda t t'. \exists s s'. P s s' \wedge (c1, s) \Rightarrow t \wedge (c', s') \Rightarrow t' \wedge (\exists u. (c2, t) \Rightarrow u \wedge Q u t'))$
using *ir-valid-Seq1 Seq* **by** *blast*
show *?case*


```

apply(rule ir-Seq1)
apply (blast intro: a)
apply(rule ir-Skip-sym)
by(metis SeqE ir-valid-def Seq)
next
case (If x1 c1 c2)
have t: ir-hoare ( $\lambda s s'. P s s' \wedge \text{bval } x1 s$ ) c1 c'
  ( $\lambda t t'. Q t t' \wedge (\exists s s'. P s s' \wedge (c1, s) \Rightarrow t \wedge (c', s') \Rightarrow t' \wedge \text{bval } x1 s)$ ) and
  f: ir-hoare ( $\lambda s s'. P s s' \wedge \neg \text{bval } x1 s$ ) c2 c'
  ( $\lambda t t'. Q t t' \wedge (\exists s s'. P s s' \wedge (c2, s) \Rightarrow t \wedge (c', s') \Rightarrow t' \wedge \neg \text{bval } x1 s)$ )
using ir-valid-If If by blast+
show ?case

apply(rule ir-conseq)
apply(rule ir-disj)
apply(rule ir-If-True,fastforce intro: t)
apply(rule ir-If-False, fastforce intro: f)
apply blast
by (smt (verit) IfE ir-valid-def If)
next
case (While x1 c)
have a:  $\bigwedge n. \text{ir-hoare } (\lambda s s'. \text{get-back } P x1 c n s s' \wedge \text{bval } x1 s) c \text{ SKIP } (\text{get-back } P x1 c (Suc n))$ 
using ir-valid-While1 While
by (smt (verit, ccfv-threshold) get-back.simps(2) ir-Skip-sym)
have b: ir-hoare ( $\lambda s s'. P s s' \wedge \text{bval } x1 s$ ) c c'
  ( $\lambda t t'. \exists s s'. (c', s') \Rightarrow t' \wedge (\exists s. (c, s) \Rightarrow t \wedge \text{bval } x1 s \wedge P s s') \wedge$ 
  ( $\exists u. (\text{WHILE } x1 \text{ DO } c, t) \Rightarrow u \wedge Q u t')$ )
using ir-valid-While3[where k=0,simplified] While by blast
have gb:  $\bigwedge t t'. Q t t' \wedge (\exists s s'. (c', s') \Rightarrow t' \wedge P t s' \wedge \neg \text{bval } x1 t) \Longrightarrow$ 
   $\exists s s'. ((\exists n. \text{get-back } P x1 c n t s') \wedge \neg \text{bval } x1 t) \wedge (c', s') \Rightarrow t'$ 
by (meson get-back.simps(1))

show ?case

apply(rule ir-conseq)
apply(rule-tac P=get-back P x1 c in ir-While-backwards-frontier)
apply(blast intro: a)

apply(rule ir-conseq)
apply(rule ir-disj)
apply(rule-tac P= $\lambda s s'. \exists n. \text{get-back } P x1 c n s s'$  and  $Q=(\lambda t t'. Q t t' \wedge$ 
  ( $\exists s s'. (c', s') \Rightarrow t' \wedge P t s' \wedge \neg \text{bval } x1 t)$ ) in ir-While-False)
apply(rule ir-Skip, blast intro: gb)
apply(rule ir-While-True)
apply(rule ir-Seq1[OF b])
apply(rule ir-Skip-sym)
apply(fastforce)
apply (metis get-back.simps(1))

```

```

    apply assumption
    apply simp
    by (metis While.prem1 WhileE ir-valid-def)
qed

```

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 :: rasn => com => com => rasn => rasn where
decomp P c c' Q ≡ λt s'. ∃ s t'. P s s' ∧ (c,s) => t ∧ (c',s') => t' ∧ Q t t'

```

lemma *ir-valid-decomp1*:

```

ir-valid P c c' Q => ir-valid P c SKIP (decomp P c c' Q) ∧ ir-valid (decomp P
c c' Q) SKIP c' Q
by(fastforce simp: ir-valid-def meh-simp decomp-def)

```

lemma *ir-valid-decomp2*:

```

ir-valid P c SKIP R ∧ ir-valid R SKIP c' Q => ir-valid P c c' Q
by(fastforce simp: ir-valid-def meh-simp decomp-def)

```

lemma *ir-valid-decomp*:

```

ir-valid P c c' Q = (ir-valid P c SKIP (decomp P c c' Q) ∧ ir-valid (decomp P
c c' Q) SKIP c' Q)
using ir-valid-decomp1 ir-valid-decomp2 by blast

```

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

lemma *ir-to-Skip*:

```

ir-hoare P c c' Q = (ir-hoare P c SKIP (decomp P c c' Q) ∧ ir-hoare (decomp P
c c' Q) SKIP c' Q)
using soundness completeness ir-valid-decomp
by meson

```

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* => *bool*

definition

```

ohearn :: assn => com => assn => bool
where
ohearn P c Q ≡ (∀ t. Q t → (∃ s. P s ∧ (c,s) => t))

```

lemma *fold-ohearn1*:

ir-valid $P\ c\ SKIP\ Q = (\forall t'.\ ohearn\ (\lambda t.\ P\ t\ t')\ c\ (\lambda t.\ Q\ t\ t'))$
by(*fastforce simp add: ir-valid-def ohearn-def*)

lemma *fold-ohearn2*:

ir-valid $P\ SKIP\ c'\ Q = (\forall t.\ ohearn\ (P\ t)\ c'\ (Q\ t))$
by(*simp add: ir-valid-def ohearn-def*)

theorem *relational-via-hoare*:

ir-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 –

have $a:$ $\bigwedge P\ c\ c'\ Q.\ ir-hoare\ P\ c\ c'\ Q = ir-valid\ P\ c\ c'\ Q$

using *soundness completeness by auto*

show *?thesis*

using *ir-to-Skip a fold-ohearn1 fold-ohearn2 by metis*

qed

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-hoare $P\ c\ SKIP\ Q \implies ir-hoare\ Q\ d\ c'\ R \implies ir-hoare\ P\ (Seq\ c\ d)\ c'\ R$
using *soundness completeness Seq2-ir-valid by metis*

lemma *ir-Seq*:

ir-hoare $P\ c\ c'\ Q \implies ir-hoare\ Q\ d\ d'\ R \implies ir-hoare\ P\ (Seq\ c\ d)\ (Seq\ c'\ d')\ R$
using *soundness completeness Seq-ir-valid by metis*

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-hoare $(\lambda s\ s'.\ P\ s\ s' \wedge bval\ b\ s \wedge \neg\ bval\ b'\ s')\ c_1\ c_2'\ Q \implies$
 $ir-hoare\ P\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ (IF\ b'\ THEN\ c_1'\ ELSE\ c_2')\ Q$
apply(*rule ir-If-True*)
apply(*rule ir-sym*)
apply(*rule ir-If-False*)
apply(*rule ir-sym*)
by(*simp add: flip-def*)

lemma *ir-Assign-Assign*:

$ir\text{-hoare } P (x ::= e) (x' ::= e') (\lambda t t'. \exists v v'. P (t(x := v)) (t'(x' := v')) \wedge t x =$
 $aval e (t(x := v)) \wedge t' x' = aval e' (t'(x' := v'))))$
apply(rule *ir-Assign*)
apply(rule *ir-sym*)
apply(rule *ir-Assign*)
by(simp add: *flip-def, auto*)

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'* $\equiv s \text{ ''low''} = s' \text{ ''low''}$

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

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

prog1 $\equiv (IF (Less (N 0) (V \text{ ''x''})) THEN (Assign \text{ ''low''} (N 1)) ELSE (Assign \text{ ''low''} (N 0)))$

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*:

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

apply(rule *exI*)

apply(rule *conjI*)

apply(simp add: *prog1-def*)

apply(rule *ir-If-True-False*)

apply(rule *ir-Assign-Assign*)

apply(rule *conjI*)

apply *auto*[1]

apply(*clarsimp simp: nontrivial-def*)

apply(rule-tac *x=λv. 1 in exI*)

apply *simp*

apply(rule-tac *x=λv. 0 in exI*)

by *auto*

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

$$bval\ (BEq\ a\ b)\ s = ((aval\ a\ s) = (aval\ b\ s))$$

by(*auto simp add: BEq-def*)

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

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

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

by(*simp add: ir-If-True ir-sym flip-def*)

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

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

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

by(*simp add: ir-If-False ir-sym flip-def*)

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

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

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

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

apply(*rule ir-pre*)

apply(*rule ir-disj*)

apply(*rule ir-If-True-True*)

apply *assumption*

apply(*rule ir-If-False-False*)

apply *assumption*

apply *blast*

done

lemma $ir\text{-While-triv}$:

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

$$ir\text{-hoare}\ P\ (WHILE\ b\ DO\ c)\ (WHILE\ b'\ DO\ c')\ (\lambda s\ s'.\ (Q_2\ s\ s'))$$

by(*simp add: ir-While-False ir-sym flip-def*)

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

$$ir\text{-hoare}\ (\lambda s\ s'.\ P\ s\ s' \wedge bval\ b\ s \wedge bval\ b'\ s')\ (c;;\ WHILE\ b\ DO\ c)\ (c';;\ WHILE\ b'\ DO\ c')\ Q_1 \Longrightarrow$$

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

$$ir\text{-hoare}\ P\ (WHILE\ b\ DO\ c)\ (WHILE\ b'\ DO\ c')\ (\lambda s\ s'.\ (Q_1\ s\ s' \vee Q_2\ s\ s'))$$

apply(*rule ir-pre*)

apply(*rule ir-disj*)

apply(*rule ir-While-True*)

apply(*rule ir-sym*)

apply(*simp add: flip-def*)

apply(*rule ir-While-True*)

```

  apply(rule ir-sym)
  apply(simp add: flip-def)
  apply(rule ir-While-triv)
  apply assumption
  apply simp
done

```

8.4 client0

definition *low-eq-strong where*

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

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

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

by(*auto simp: low-eq-strong-def*)

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

definition *client0 :: com where*

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

lemma *client0:*

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

apply(*rule exI, rule conjI, simp add: client0-def*)

apply(*rule-tac P=low-eq-strong in ir-pre*)

apply(*rule ir-Seq*)

apply(*rule ir-Seq*)

apply(*rule ir-Assign-Assign*)

apply *clarsimp*

apply(*rule ir-While'*)

apply *clarsimp*

apply(*rule ir-Seq*)

apply(*rule ir-Seq*)

apply(*rule ir-Assign-Assign*)

apply(*rule ir-Assign-Assign*)

apply *clarsimp*

apply(*rule ir-While-triv*)

apply *clarsimp*

apply *assumption*

apply *clarsimp*
apply *assumption*

apply(*rule ir-If-True-True*)
apply(*rule ir-Assign-Assign*)
apply(*fastforce simp: low-eq-strong-def*)
apply(*rule conjI*)
apply(*clarsimp simp: low-eq-strong-def split: if-splits*)

apply(*clarsimp simp: low-eq-strong-def nontrivial-def*)
apply(*rule-tac x=λv. if v = "x" then 2000000 else if v = "high" then 1 else if v = "n" then -1 else if v = "nondet" then -1 else if v = "low" then 1 else undefined in exI*)
apply(*rule-tac x=λv. if v = "x" then 2000000 else if v = "high" then 0 else if v = "n" then -1 else if v = "nondet" then -1 else if v = "low" then 0 else undefined in exI*)
apply *clarsimp*
done

lemma *ir-While-backwards*:

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

apply(*rule ir-While-backwards-frontier*)
apply *assumption*
apply(*rule ir-While-False*)
apply *auto*
done

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*: $\bigwedge n. \forall t t'. P (k + Suc n) t t' \longrightarrow$
 $(\exists s s'. P (k + n) s s' \wedge \text{bval } b s \wedge \text{bval } b' s' \wedge (c, s) \Rightarrow t \wedge (c', s') \Rightarrow t')$

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 –
from *post last* **obtain** $s s' n$ **where**
 $P (k + n) s s' (\text{WHILE } b \text{ DO } c, s) \Rightarrow t (\text{WHILE } b' \text{ DO } c', s') \Rightarrow t'$
by *blast*
with *asm show ?thesis*
proof(*induction n arbitrary: k t t'*)
case 0
then show *?case*
by (*metis WhileFalse WhileTrue add.right-neutral*)
next
case (*Suc n*)
from *Suc*
obtain $r r'$ **where** *final-iteration*: $P (\text{Suc } k) r r' (\text{WHILE } b \text{ DO } c, r) \Rightarrow t$
 $(\text{WHILE } b' \text{ DO } c', r') \Rightarrow t'$
by (*metis add-Suc-shift*)
from *final-iteration(1)* **obtain** $q q'$ **where**
 $P k q q' \wedge \text{bval } b \text{ } q \wedge \text{bval } b' \text{ } q' \wedge (c, q) \Rightarrow r \wedge (c', q') \Rightarrow r'$
by (*metis Nat.add-0-right Suc.prem(1) plus-1-eq-Suc semiring-normalization-rules(24)*)
with *final-iteration* **show** *?case* **by** *blast*
qed
qed

lemma *While-backwards-frontier-both-ir-valid[intro]*:
 $(\bigwedge n. \text{ir-valid } (\lambda s s'. P n s s' \wedge \text{bval } b \text{ } s \wedge \text{bval } b' \text{ } s') c c' (P (\text{Suc } n))) \Longrightarrow$
 $\text{ir-valid } (\lambda s s'. \exists n. P n s s') (\text{WHILE } b \text{ DO } c) (\text{WHILE } b' \text{ DO } c') Q \Longrightarrow$
 $\text{ir-valid } (P 0) (\text{WHILE } b \text{ DO } c) (\text{WHILE } b' \text{ DO } c') (\lambda s s'. Q s s')$
by(*auto simp: ir-valid-def intro: While-backwards-frontier-both-ir-valid'*)

lemma *ir-While-backwards-frontier-both*:
 $\llbracket \bigwedge n. \text{ir-hoare } (\lambda s s'. P n s s' \wedge \text{bval } b \text{ } s \wedge \text{bval } b' \text{ } s') c c' (P (\text{Suc } n));$
 $\text{ir-hoare } (\lambda s s'. \exists n. P n s s') (\text{WHILE } b \text{ DO } c) (\text{WHILE } b' \text{ DO } c') Q \rrbracket$
 $\Longrightarrow \text{ir-hoare } (P 0) (\text{WHILE } b \text{ DO } c) (\text{WHILE } b' \text{ DO } c') (\lambda s s'. Q s s')$
using *soundness completeness While-backwards-frontier-both-ir-valid* **by** *auto*

The following rule then follows easily as a special case

lemma *ir-While-backwards-both*:
 $(\bigwedge n. \text{ir-hoare } (\lambda s s'. P n s s' \wedge \text{bval } b \text{ } s \wedge \text{bval } b' \text{ } s') c c' (P (\text{Suc } n))) \Longrightarrow$
 $\text{ir-hoare } (P 0) (\text{WHILE } b \text{ DO } c) (\text{WHILE } b' \text{ DO } c') (\lambda s s'. \exists n. P n s s' \wedge \neg \text{bval } b \text{ } s \wedge \neg \text{bval } b' \text{ } s')$
apply(*rule ir-While-backwards-frontier-both*)
apply *blast*
by(*simp add: ir-While-False ir-sym flip-def*)

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**

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

lemma *client1*:

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

apply(*rule exI*, *rule conjI*, *simp add: client1-def*)
apply(*rule-tac P=low-eq-strong in ir-pre*)
apply(*rule ir-Seq*)
apply(*rule ir-Seq*)
apply(*rule ir-Assign-Assign*)
apply *clarsimp*

apply(*rule ir-pre*)
apply(*rule ir-While-backwards-both*[**where** $P = \lambda n s s'. \text{low-eq-strong } s s' \wedge s "x" = \text{int } n \wedge s' "x" = \text{int } n \wedge \text{int } n \leq s "n" \wedge \text{int } n \leq s' "n"$])
apply(*rule ir-post*)
apply(*rule ir-Assign-Assign*)
apply *clarsimp*

apply *clarsimp*

apply(*rule ir-If-True-True*)
apply(*rule ir-Assign-Assign*)
apply(*fastforce simp: low-eq-strong-def*)
apply(*rule conjI*)
apply(*clarsimp simp: low-eq-strong-def split: if-splits*)

apply *clarsimp*

apply(*clarsimp simp: low-eq-strong-def nontrivial-def*)
apply(*rule-tac x=λv. if v = "x" then 2000000 else if v = "high" then 1 else if v = "n" then 2000000 else if v = "nondet" then -1 else if v = "low" then 1 else undefined in exI*)
apply(*rule-tac x=λv. if v = "x" then 2000000 else if v = "high" then 0 else if v = "n" then 2000000 else if v = "nondet" then -1 else if v = "low" then 0 else undefined in exI*)
apply *clarsimp*
done

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*:

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

```

apply(rule exI, rule conjI, simp add: client2-def)
apply(rule-tac P=low-eq-strong in ir-pre)
apply(rule ir-Seq)
apply(rule ir-Assign-Assign)
apply clarsimp

```

```

apply(rule ir-pre)

```

```

apply(rule ir-While-backwards-frontier-both[where P= $\lambda n s s'. \text{low-eq-strong } s s' \wedge s \text{"x"} = \text{int } n \wedge s' \text{"x"} = \text{int } n \wedge s \text{"x"} \geq 0 \wedge s \text{"x"} \leq 2000000 - 1 \wedge s' \text{"x"} \geq 0 \wedge s' \text{"x"} \leq 2000000 - 1$ ])

```

```

apply(rule ir-Seq)
apply(rule ir-Assign-Assign)
apply clarsimp
apply(rule ir-post)
apply(rule ir-If')
apply(rule ir-Assign-Assign)
apply(rule ir-Skip-Skip)
apply clarsimp

```

```

apply clarsimp

```

```

apply(rule ir-While')
apply clarsimp
apply(rule ir-Seq)
apply(rule ir-Seq)
apply(rule ir-Assign-Assign)
apply(rule ir-If-True-True)
apply(rule ir-Assign-Assign)
apply clarsimp

```

```

apply(rule ir-pre)

```

```

apply(rule ir-While-backwards-both[where P= $\lambda n s s'. s \text{"x"} = 2000000 + \text{int } n \wedge s' \text{"x"} = 2000000 + \text{int } n \wedge s \text{"x"} \geq 2000000 \wedge s \text{"x"} \leq 4000000 \wedge s' \text{"x"} \geq 2000000 \wedge s' \text{"x"} \leq 4000000 \wedge s \text{"low"} = s \text{"high"} \wedge s' \text{"low"} = s' \text{"high"} \wedge s \text{"high"} \neq s' \text{"high"}$ ])

```

```

apply(rule ir-Seq)
apply(rule ir-Assign-Assign)
apply(rule ir-If-False-False)
apply fastforce
apply (fastforce simp: low-eq-strong-def)
apply fastforce
apply(fastforce simp: low-eq-strong-def)
apply(fastforce simp: low-eq-strong-def)

apply(rule conjI)
apply(clarsimp simp: low-eq-strong-def split: if-splits)

apply clarsimp

apply(clarsimp simp: low-eq-strong-def nontrivial-def)
apply(rule-tac x= $\lambda v$ . if v = "x" then 4000000 else if v = "high" then 1 else if
v = "n" then 2000000 else if v = "nondet" then -1 else if v = "low" then 1 else
undefined in exI)
apply(rule-tac x= $\lambda v$ . if v = "x" then 4000000 else if v = "high" then 0 else if
v = "n" then 2000000 else if v = "nondet" then -1 else if v = "low" then 0 else
undefined in exI)
apply clarsimp
done

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.