Semantics and Data Refinement of Invariant Based Programs

Viorel Preoteasa and Ralph-Johan Back

February 23, 2021

Abstract

The invariant based programming is a technique of constructing correct programs by first identifying the basic situations (pre- and post-conditions and invariants) that can occur during the execution of the program, and then defining the transitions and proving that they preserve the invariants. Data refinement is a technique of building correct programs working on concrete datatypes as refinements of more abstract programs. In the theories presented here we formalize the predicate transformer semantics for invariant based programs and their data refinement.

Contents

1 Introduction 2

2 Preliminaries 2
  2.1 Simplification Lemmas . . . . . . . . . . . . . . . . . . . . . . 2

3 Program Statements as Predicate Transformers 3
  3.1 Assert statement . . . . . . . . . . . . . . . . . . . . . . . . . 3
  3.2 Assume statement . . . . . . . . . . . . . . . . . . . . . . . . 4
  3.3 Demonic update statement . . . . . . . . . . . . . . . . . . . . 4
  3.4 Angelic update statement . . . . . . . . . . . . . . . . . . . . 5
  3.5 The guard of a statement . . . . . . . . . . . . . . . . . . . . . 5

4 Hoare Triples 6
  4.1 Hoare rule for recursive statements . . . . . . . . . . . . . . . 7

5 Predicate Transformers Semantics of Invariant Diagrams 9

6 Data Refinement of Diagrams 15
1 Introduction

Invariant based programming [1, 2, 3, 4] is an approach to construct correct programs where we start by identifying all basic situations (pre- and post-conditions, and loop invariants) that could arise during the execution of the algorithm. These situations are determined and described before any code is written. After that, we identify the transitions between the situations, which together determine the flow of control in the program. The transitions are verified at the same time as they are constructed. The correctness of the program is thus established as part of the construction process.

These theories present the predicate transformer semantics for invariant based programs and their data refinement. The complete treatment of the semantics of invariant based programs was presented in [4]. There we introduced big and small step semantics, predicate transformer semantics, and we proved complete and correct Hoare rules for invariant based programs. These results were also formalized in the PVS theorem prover. In [6] we have studied data refinement of invariant based programs, and we outlined the steps for proving the Deutsch-Schorr-Waite marking algorithm using data refinement of invariant based programs. These theories represent a mechanical formalization of the data refinement results from [6] and some of the results from [4]. In another formalization we will show how the theory presented here can be used in the complete verification of the marking algorithm.

2 Preliminaries

theory Preliminaries
imports Main. LatticeProperties. Complete-Lattice-Prop
     LatticeProperties. Conj-Disj
begin

notation
  less-eq (infix ⊑ 50) and
  less (infix ⊏ 50) and
  inf (infixl ⊓ 70) and
  sup (infixl ⊔ 65) and
  top (∨) and
  bot (∨) and
  Inf (⋯ : [900] 900) and
  Sup (⋯ : [900] 900)

2.1 Simplification Lemmas

declare fun-upd-idem[simp]

lemma simp-eq-emptyset:
(X = {}) = (∀ x. x ∉ X)
by blast

lemma mono-comp: mono f ⊢ mono g ⊢ mono (f o g)
by (unfold mono-def) auto

Some lattice simplification rules

lemma inf-bot-bot:
(x::'a::{semilattice-inf,order-bot}) ∩ ⊥ = ⊥
apply (rule antisym)
by auto

end

3 Program Statements as Predicate Transformers

theory Statements
imports Preliminaries
begin

Program statements are modeled as predicate transformers, functions from predicates to predicates. If State is the type of program states, then a program S is a a function from State set to State set. If q ∈ State set, then the elements of S q are the initial states from which S is guarantied to terminate in a state from q.

However, most of the time we will work with an arbitrary compleate lattice, or an arbitrary boolean algebra instead of the complete boolean algebra of predicate transformers.

We will introduce in this section assert, assume, demonic choice, angelic choice, demonic update, and angelic update statements. We will prove also that these statements are monotonic.

lemma mono-top[simp]: mono top
by (simp add: mono-def top-fun-def)

lemma mono-choice[simp]: mono S ⊢ mono T ⊢ mono (S ∩ T)
apply (simp add: mono-def inf-fun-def)
apply safe
apply (rule-tac y = S x in order-trans)
apply simp-all
apply (rule-tac y = T x in order-trans)
by simp-all

3.1 Assert statement

The assert statement of a predicate p when executed from a state s fails if s ∉ p and behaves as skip otherwise.
definition
assert:: 'a::semilattice-inf \Rightarrow 'a \Rightarrow 'a ([.] [0] 1000) where

\{.p.\} q \equiv p \cap q

lemma mono-assert [simp]: mono \{.p.\}
  apply (simp add: assert-def mono-def, safe)
  apply (rule-tac y = x in order-trans)
  by simp-all

3.2 Assume statement

The assume statement of a predicate p when executed from a state s is not enabled if s \notin p and behaves as skip otherwise.

definition
assume:: 'a::boolean-algebra \Rightarrow 'a \Rightarrow 'a (\{.p.\} [0] 1000) where

\{.p.\} q \equiv \neg p \cup q

lemma mono-assume [simp]: mono (assume P)
  apply (simp add: assume-def mono-def)
  apply safe
  apply (rule-tac y = y in order-trans)
  by simp-all

3.3 Demonic update statement

The demonic update statement of a relation Q : State \rightarrow State \rightarrow bool, when executed in a state s computes nondeterministically a new state s' such Q s s' is true. In order for this statement to be correct all possible choices of s' should be correct. If there is no state s' such that Q s s', then the demonic update of Q is not enabled in s.

definition
demonic:: ('a \Rightarrow 'b::ord) \Rightarrow 'b::ord \Rightarrow 'a set ([.] [0] 1000) where

\{.Q:\} p = \{ s . Q s \leq p \}

lemma mono-demonic [simp]: mono \{.Q:\}
  apply (simp add: mono-def demonic-def)
  by auto

theorem demonic-bottom:
  [:R:] (\bot::('a::order-bot)) = \{ s . (R s) = \bot \}
  apply (unfold demonic-def, safe, simp-all)
  apply (rule antisym)
  by auto

theorem demonic-bottom-top [simp]:
  [:\top::order-bot:] = \top
by (simp add: fun-eq-iff inf-fun-def sup-fun-def demonic-def top-fun-def bot-fun-def)

theorem demonic-sup-inf:
\[ [Q \sqcup Q'] = [Q] \cap [Q'] \]
by (simp add: fun-eq-iff sup-fun-def inf-fun-def demonic-def, blast)

3.4 Angelic update statement

The angelic update statement of a relation \( Q : \text{State} \to \text{State} \to \text{bool} \) is similar to the demonic version, except that it is enough that at least for one choice \( s', Q s s' \) is correct. If there is no state \( s' \) such that \( Q s s' \), then the angelic update of \( Q \) fails in \( s \).

definition angelic :: \((a' \Rightarrow b'; \text{semilattice-inf,order-bot}) \Rightarrow a' \Rightarrow \{\text{set}\}\{0\} 1000)\) where
\{\{Q\} p = \{s . (Q s) \cap p \neq \bot\}\}

syntax -update :: patterns => patterns => logic => logic (- _ _ - 0)
translations  
-update (-patterns x xs) (-patterns y ys) t == CONST id (-abs (-pattern x xs) (-Coll (-pattern y ys) t))
-update x y t == CONST id (-abs x (-Coll y t))

term \{ y, z => x, z'. P x y z z' :\}

theorem angelic-bottom [simp]:
angelic R ⊥ = \{\}
by (simp add: angelic-def inf-bot-bot)

theorem angelic-disjunctive [simp]:
\{(R::(a' => 'b::complete-distrib-lattice)):\} ∈ Apply.Disjunctive
by (simp add: Apply.Disjunctive-def angelic-def inf-Sup, blast)

3.5 The guard of a statement

The guard of a statement \( S \) is the set of initial states from which \( S \) is enabled or fails.

definition ((grd S)::a::boolean-algebra) = - (S bot)

lemma grd-choice[simp]: grd (S \sqcap T) = (grd S) \sqcup (grd T)
by (simp add: grd-def inf-fun-def)

lemma grd-demonic: grd [:Q:] = \{s . \exists s'. s' ∈ (Q s) \}
apply (simp add: grd-def demonic-def)
by blast

lemma grd-demonic-2[simp]: (s \notin grd [:Q:]) = (\forall s'. s' \notin (Q s))
by (simp add: grd-demonic)

**Theorem** `grd-angelic`:

\[ \text{grd}\{ \cdot : R \cdot \} = \text{UNIV} \]

by (simp add: grd-def)

end

### 4 Hoare Triples

**Theory** `Hoare`

**Imports** `Statements`

**Begin**

A hoare triple for \( p, q \in \text{State set} \), and \( S : \text{State set} \rightarrow \text{State set} \) is valid, denoted \( \models p[S]q \), if every execution of \( S \) starting from state \( s \in p \) always terminates, and if it terminates in state \( s' \), then \( s' \in q \). When \( S \) is modeled as a predicate transformer, this definition is equivalent to requiring that \( p \) is a subset of the initial states from which the execution of \( S \) is guaranteed to terminate in \( q \), that is \( p \subseteq S q \).

The formal definition of a valid hoare triple only assumes that \( p \) (and also \( S q \)) ranges over a complete lattice.

**Definition**

\[ \text{Hoare} :: \text{'a::complete-distrib-lattice} \Rightarrow (\text{'b \Rightarrow 'a}) \Rightarrow \text{'b \Rightarrow bool} \] \[ (\models (-)[| |-](-)) ([0,0,900] 900) \text{ where} \]

\[ \models p [| S |] q = (p \leq (S q)) \]

**Theorem** `hoare-sequential`:

\[ \text{mono } S \Rightarrow (\models p [| S \circ T |] r) = (\exists q. (\models p [| S |] q \land \models q [| T |] r)) \]

by (metis (no-types) `Hoare-def monoD o-def order-refl order-trans`)

**Theorem** `hoare-choice`:

\[ \models p [| S \sqcap T |] q = (\models p [| S |] q \land \models p [| T |] q) \]

by (simp-all add: `Hoare-def inf-fun-def`)

**Theorem** `hoare-assume`:

\[ (\models P [| .R. |] Q) = (P \sqcap R \leq Q) \]

apply (simp add: `Hoare-def assume-def`)

apply safe

apply (case-tac (inf P R) \leq (inf (sup (\cdot - R) Q) R))

apply (simp add: inf-sup-distrib2)

apply (simp add: le-infl1)

apply (case-tac (sup (\cdot - R) (inf P R)) \leq sup (\cdot - R) Q)

apply (simp add: sup-inf-distrib1)

by (simp add: le-supl2)

**Theorem** `hoare-mono`:

\[ \text{mono } S \Rightarrow Q \leq R \Rightarrow (\models p [| S |] Q \Rightarrow \models p [| S |] R) \]

6
apply (simp add: mono-def Hoare-def)
apply (rule-tac y = S Q in order-trans)
by auto

theorem hoare-pre:
\[ R \leq P \implies \vdash P \{ | S | \} Q \implies \vdash R \{ | S | \} Q \]
by (simp add: Hoare-def)

theorem hoare-Sup:
\[ (\forall p \in P . \vdash p \{ | S | \} q) = \vdash \Sup P \{ | S | \} q \]
apply (simp add: Hoare-def, safe, simp add: Sup-least)
apply (rule-tac y = \\\uplus P in order-trans, simp-all)
by (simp add: Sup-upper)

lemma hoare-magic [simp]: \[ \vdash P \{ | \top \} Q \]
by (simp add: Hoare-def top-fun-def)

lemma hoare-demonic:
\[ \vdash P \{ | [:R:] \} Q = (\forall s . s \in P \implies R s \subseteq Q) \]
apply (unfold Hoare-def demonic-def)
by auto

lemma hoare-not-guard:
mono \( S :: (\:\:\:\::\text{order-bot}) \Rightarrow \) \[ \vdash p \{ | S | \} q \]
apply (simp add: Hoare-def grd-def, safe)
apply (drule monoD)
by auto

4.1 Hoare rule for recursive statements

A statement \( S \) is refined by another statement \( S' \) if \( \vdash p\{ | S' | \} q \) is true for all \( p \) and \( q \) such that \( \vdash p\{ | S | \} q \) is true. This is equivalent to \( S \leq S' \).

Next theorem can be used to prove refinement of a recursive program. A recursive program is modeled as the least fixpoint of a monotonic mapping from predicate transformers to predicate transformers.

theorem lfp-uf-induction:
\[ \text{mono } f \Rightarrow (\forall w . (p w) \leq f (\text{Sup-less } p w)) \implies \text{Sup (range } p \text{)} \leq \text{lfp } f \]
apply (rule fp-uf-induction, simp-all)
by (drule lfp-unfold, simp)

definition post-fun \( p :: \text{a::order} \) \[ q = (\text{if } p \leq q \text{ then } \top \text{ else } \bot) \]

lemma post-mono [simp]: mono \( \text{post-fun } p :: (\:\:\:\::\text{order-bot,order-top}) \)
apply (simp add: post-fun-def mono-def, safe)
apply (subgoal-tac p \leq y, simp)
by (rule-tac y = x in order-trans, simp-all)

lemma post-top [simp]: post-fun \( p \top = \top \)
Sup-less-assert

Sup-less-comp
lemma Sup-range-comp

(⨆ assert-Sup-range
lemma assert-Sup
lemma | v < w |

Next theorem gives a Hoare rule for recursive programs. If we can prove the unfolding of the recursive definition applid to a program f,

\[ \vdash p w \{\{ f \} \} y, \text{assuming that } f \text{ is correct when starting from } p v, \]

\[ v < w, \vdash \text{SUP } L p w \{\{ f \} \} y, \text{then the recursive program is correct} \]

\[ \vdash \text{SUP } p \{\{ \text{ifp } F \} \} y \]

lemma assert-Sup: \( \bigcup \{ \text{X::'a::complete-distrib-lattice set.} \} = \bigcup \{ \text{assert } \cdot X \} \)

by (simp add: fun-eq-iff assert-def Sup-inf image-comp)

lemma assert-Sup-range: \( \bigcup \{ \text{range } (p::'W \Rightarrow 'a::complete-distrib-lattice)) \} = \bigcup \{ \text{range } (\text{assert o p}) \} \)

by (simp add: fun-eq-iff assert-def SUP-inf image-comp)

lemma Sup-range-comp: \( \bigcup \{ \text{range } p \} o S = \bigcup \{ \text{range } (\lambda w. ((p w) o S)) \} \)

by (simp add: fun-eq-iff image-comp)

lemma Sup-less-comp: (Sup-less P) w o S = Sup-less (\lambda w. ((P w) o S)) w

apply (simp add: Sup-less-def fun-eq-iff, safe)

apply (subgoal-tac \((\lambda f. f (S x)) \cdot \{y. \exists v<w. \forall x. y x = P v x \}) = ((\lambda f. f x) \cdot \{y. \exists v<w. \forall x. y x = P v (S x)\}))

apply (auto cong del: SUP-cong-simp)
done

lemma Sup-less-assert: Sup-less (\lambda w. \{. (p w)::'a::complete-distrib-lattice .\}) w = \{. Sup-less p w.\}

apply (simp add: Sup-less-def assert-Sup image-def)

apply (subgoal-tac \{y. \exists v<w. y = \{. p v .\}\} = \{y. \exists x. (\exists v<w. x = p v) \wedge y = \}

8
\[
\{. x .\} \}
\]
apply (auto simp add: image-def cong del: SUP-cong-simp)
done

declare mono-comp[ simp]

theorem hoare-fixpoint:
mono-mono F ⇒
(∀ w f . mono f ∧ \# Sup-less p w { | f | } y \# ⇒ p w { | F f | } y \# ⇒ (Sup
(range p)) { | lfp F | } y
apply (simp add: mono-mono-def hoare-refinement-post assert-Sup-range Sup-range-comp)
apply (rule lfp-wf-induction)
apply auto
apply (simp add: Sup-less-comp [THEN sym])
apply (simp add: Sup-less-assert)
apply (drule-tac x = \{. Sup-less p w .\} ∘ post-fun y in spec, safe)
apply simp
by (simp add: hoare-refinement-post)

theorem (∀ t . \# (\{ s . t ∈ R s \}) { | S | } q \# ⇒ (\{:R:\} p) { | S | } q
apply (simp add: Hoare-def angelic-def subset-eq)
by auto

end

5 Predicate Transformers Semantics of Invariant Diagrams

theory Diagram
imports Hoare
begin

This theory introduces the concept of a transition diagram and proves a number of Hoare total correctness rules for these diagrams. As before the diagrams are introduced using their predicate transformer semantics.

A transition diagram \( D \) is a function from pairs of indexes to predicate transformers: \( D : I × I \to (\text{State set} \to \text{State set}) \), or more general \( D : I × I \to \text{Ptran} \), where \( \text{Ptran} \) is a complete lattice. The elements of \( I \) are called situations and intuitively a diagram is executed starting in a situation \( i \in I \) by choosing a transition \( D(i,j) \) which is enabled and continuing similarly from \( j \) if there are enabled transitions. The execution of a diagram stops when there are no more transitions enabled or when it fails.

The semantics of a transition diagram is an indexed predicate transformer \((I \to \text{State set})\). If \( Q : I \to \text{State set} \) is an indexed predicate, then \( p = \text{pt} D Q i \) is a weakest predicate such that if the execution of \( D \) starts in
a state $s \in p$ from situation $i$, then it terminates, and if it terminates in situation $j$ and state $s'$, then $s' \in Q j$.

We introduce first the indexed predicate transformer $\text{step } D$ of executing one step of diagram $D$. The predicate $\text{step } D \ Q i$ is true for those states $s$ from which the execution of one step of $D$ starting in situation $i$ ends in one of the situations $j$ such that $Q j$ is true.

**Definition**

$\text{step } D \ Q i = (\inf j . \ D (i, j) \ (Q j)) :: - :: \text{complete-lattice}$

**Definition**

$\text{dmono } D = (\forall ij . \text{mono } (D ij))$

**Lemma** $\text{dmono-mono [simp]: dmono } D \implies \text{mono } (D ij)$

by (simp add: dmono-def)

**Theorem** $\text{mono-step [simp]:}$

$\text{dmono } D \implies \text{mono } (\text{step } D)$

**Apply** (simp add: dmono-def mono-def le-fun-def step-def Inf-fun-def)

**Apply** auto

**Apply** (rule INF-greatest)

**Apply** auto

**Apply** (rule_tac $y = \text{D (xa, j) (x j)}$ in order-trans)

**Apply** auto

**Apply** (rule INF-lower)

by auto

The indexed predicate transformer of a transition diagram is defined as the least fixpoint of the unfolding of the execution of the diagram. The indexed predicate transformer $\text{dgr } D \ U$ is the choice between executing one step of $D$ followed by $U \ ((\text{step } D) \circ U)$ or skip if no transition of $D$ is enabled (assume $\neg \text{grd}(\text{step } D)$).

**Definition**

$\text{dgr } D \ U = ((\text{step } D) \circ U) \sqcap [-\text{grd}(\text{step } D)].$

**Theorem** $\text{mono-mono-dgr [simp]: dmono } D \implies \text{mono } (\text{dgr } D)$

**Apply** (simp add: mono-mono-def dgr-def)

**Apply** safe

**Apply** (simp-all add: dgr-def)

**Apply** (simp-all add: le-fun-def inf-fun-def)

**Apply** safe

**Apply** (rule_tac $y = (\text{step } D \ (x xa) \ xb)$ in order-trans)

**Apply** simp-all

**Apply** (case-tac mono (step $D$))

**Apply** (simp add: mono-def)

**Apply** (simp add: le-fun-def)

**Apply** simp

**Apply** (rule-tac $y = (\text{step } D \ (f x) \ xa)$ in order-trans)

**Apply** simp-all
apply (case-tac mono (step D))
apply (simp add: mono-def)
apply (simp-all add: le-fun-def)
apply (rule-tac y = (assume (¬ grd (step D)) x xa) in order-trans)
apply simp-all
apply (case-tac mono (assume (¬ grd (step D)))))
apply (simp add: mono-def le-fun-def)
by simp

definition pt D = lfp (dgr D)

If \( U \) is an indexed predicate transformer and if \( P, Q : I \to State set \) are
indexed predicates, then the meaning of the Hoare triple defined earlier,
\( \models P \{ U \} Q \), is that if we start \( U \) in a state \( s \) from a situation \( i \) such that
\( s \in P i \), then \( U \) terminates, and if it terminates in \( s' \) and situation \( j \), then
\( s' \in Q j \) is true.

Next theorem shows that in a diagram all transitions are correct if and only
if \( step D \) is correct.

theorem hoare-step:
(\( \forall i j. \models (P i) \{ D(i,j) \} (Q j) \to (\models P \{ step D \} Q) \))
apply safe
apply (simp add: le-fun-def Hoare-def step-def)
apply safe
apply (rule INF-greatest)
apply auto
apply (simp add: le-fun-def Hoare-def step-def)
apply (erule-tac x = i in allE)
apply (rule-tac y = INF j. D(i, j) (Q j) in order-trans)
apply auto
apply (rule INF-lower)
by auto

Next theorem provides the first proof rule for total correctnes of transition
diagrams. If all transitions are correct and if a global variant decreases on
every transition then the diagram is correct and it terminates. The variant
must decrease according to a well founded and transitive relation.

theorem hoare-diagram:
dmono D \to (\( \forall i j. \models X w i \{ D(i,j) \} Sup-less X w j \to
\models (Sup (range X)) \{ pt D \} (Sup (range X) \cap (¬ grd (step D))) \))
apply (simp add: hoare-step pt-def)
apply (rule hoare-fixpoint)
apply auto
apply (simp add: dgr-def)
apply (simp add: hoare-choice)
apply safe
apply (simp add: hoare-sequential)
apply auto
apply (simp add: hoare-assume)
apply (rule le-infI)
by (rule SUP-upper, auto)

This theorem is a more general form of the more familiar form with a variant
t which must decrease. If we take \( X w i = (Y i \land t i = w) \), then the second
hypothesis of the theorem above becomes \( Y i \land t i = w \{ D(i,j) \} \) \( Y i \land t i < w \).
However, the more general form of the theorem is needed, because in
data refinements, the form \( Y i \land t i = w \) cannot be preserved.

The drawback of this theorem is that the variant must be decreased on ev-
ery transitions which may be too cumbersome for practical applications. A
similar situation occur when introducing proof rules for mutually recursive
procedures. There the straightforward generalization of the proof rule of a
recursive procedure to mutually recursive procedures suffers of a similar
problem. We would need to prove that the variant decreases before every
recursive call. Nipkow [5] has introduced a rule for mutually recursive pro-
cedures in which the variant is required to decrease only in a sequence of
recursive calls before calling again a procedure in this sequence. We intro-
duce a similar proof rule in which the variant depends also on the situation
indexes.

locale DiagramTermination =
  fixes pair:: 'a⇒'b⇒ ('c::well-founded-transitive)
begin

definition
  \( SUP-L-P X u i = (SUP v \in \{ v. \text{pair } v i < u \}. X v i :: :: \text{complete-lattice}) \)

definition
  \( SUP-LE-P X u i = (SUP v \in \{ v. \text{pair } v i \leq u \}. X v i :: :: \text{complete-lattice}) \)

lemma \( SUP-L-P-upper \):
  pair v i < u \implies P v i \leq SUP-L-P P u i
by (auto simp add: SUP-L-P-def intro: SUP-upper)

lemma \( SUP-L-P-least \):
  (!! v . pair v i < u \implies P v i \leq Q) \implies SUP-L-P P u i \leq Q
by (simp add: SUP-L-P-def, rule SUP-least, auto)

lemma \( SUP-LE-P-upper \):
  pair v i \leq u \implies P v i \leq SUP-LE-P P u i
by (auto simp add: SUP-LE-P-def intro: SUP-upper)

lemma \( SUP-LE-P-least \):
  (!! v . pair v i \leq u \implies P v i \leq Q) \implies SUP-LE-P P u i \leq Q
by (simp add: SUP-LE-P-def, rule SUP-least, auto)

lemma \( SUP-SUP-L \): \( Sup \ (range \ (SUP-LE-P X)) = Sup \ (range \ X) \)
apply (simp add: fun-eq-iff Sup-fun-def image-comp, clarify)
apply (rule antisym)
apply (rule SUP-LE-P-least)
apply (rule SUP-upper, simp)
apply (rule SUP-least)
apply (rule_tac y = SUP-LE-P X (pair xa x) x in order-trans)
by (rule SUP-upper, simp)

lemma SUP-L-SUP-LE-P [simp]: Sup-less (SUP-LE-P X) = SUP-L-P X
apply (rule antisym)
apply (subst le-fun-def, safe)
apply (rule Sup-less-least)
apply (subst le-fun-def, safe)
apply (rule SUP-LE-P-least)
apply (rule SUP-L-P-upper, simp)
apply (simp add: le-fun-def, safe)
apply (rule SUP-L-P-least)
apply (rule-tac X = SUP-LE-P X (pair v xa) xa in order-trans)
apply (rule SUP-LE-P-upper, simp)
apply (cut-tac P = SUP-LE-P X in Sup-less-upper)
by (simp, simp add: le-fun-def)

end

theorem (in DiagramTermination) hoare-diagram2:
dmono D =\Longrightarrow (\forall u i j . \uparrow X u i \{ i \ D(i, j) \} \ SUP-L-P X (pair u i) j) =\Longrightarrow
\uparrow (Sup (range X)) \ (\{ i \ pt D \} \ (Sup (range X)) \cap (- (grd (step D))))
apply (frule-tac X = SUP-LE-P X in hoare-diagram)
apply auto
apply (simp add: SUP-LE-P-def)
apply (unfold hoare-Sup [THEN sym])
apply auto
apply (rule-tac Q = SUP-L-P X (pair p i) j in hoare-mono)
apply auto
apply (rule SUP-L-P-least)
apply (rule SUP-L-P-upper)
apply (rule order-trans?)
by auto

lemma mono-pt [simp]: dmono D =\Longrightarrow mono (pt D)
apply (drule mono-mono-dgr)
by (simp add: pt-def)

theorem (in DiagramTermination) hoare-diagram3:
dmono D =\Longrightarrow
(\forall u i j . \uparrow X u i \{ i \ D(i, j) \} \ SUP-L-P X (pair u i) j) =\Longrightarrow
P \leq Sup (range X) =\Longrightarrow (\{(Sup (range X)) \cap (- (grd (step D)))\) \leq Q =\Longrightarrow
\( \vdash P \left[ | pt D | \right] Q \)

apply (rule hoare-mono)
apply auto
apply (rule hoare-pre)
apply auto
apply (rule hoare-diagram2)
by auto

The following definition introduces the concept of correct Hoare triples for diagrams.

**definition** (in DiagramTermination)

\( \text{Hoare-dgr} :: \ ('b :: \{ \text{complete-distrib-lattice, boolean-algebra} \}) \Rightarrow ('b \times 'b \Rightarrow 'u \Rightarrow 'u) \Rightarrow ('b \Rightarrow 'u) \Rightarrow \text{bool} \ \vdash (-)[| - |](\_)
\[0,0,900,900\]) where

\( \vdash P \left[ | D | \right] Q \equiv (\exists X . \forall u i j . \vdash X u i \left[ | D(i, j) | \right] SUP-L-P X (\text{pair } u i) j) \wedge P = \text{Sup (range } X) \wedge Q = (\text{((Sup (range } X) \cap \neg (\text{grad (step } D))))
\]

**definition** (in DiagramTermination)

\( \text{Hoare-dgr1} :: ('b \Rightarrow ('u :: \{ \text{complete-distrib-lattice, boolean-algebra} \}) \Rightarrow ('b \times 'b\Rightarrow 'u \Rightarrow 'u) \Rightarrow ('b \Rightarrow 'u) \Rightarrow \text{bool} \ \vdash (-)[| - |](\_)
\[0,0,900,900\]) where

\( \vdash I P \left[ | D | \right] Q \equiv (\exists X . \forall u i j . \vdash X u i \left[ | D(i, j) | \right] SUP-L-P X (\text{pair } u i) j) \wedge P \leq \text{Sup (range } X) \wedge ((\text{Sup (range } X) \cap \neg (\text{grad (step } D)))) \leq Q
\]

**theorem** (in DiagramTermination) hoare-dgr-correctness:

\( \text{dmono } D \Rightarrow (\vdash P \left[ | D | \right] Q) \Rightarrow (\vdash P \left[ | pt D | \right] Q) \)

apply (simp add: Hoare-dgr-def)
apply safe
apply (rule hoare-diagram3)
by auto

**theorem** (in DiagramTermination) hoare-dgr-correctness1:

\( \text{dmono } D \Rightarrow (\vdash I P \left[ | D | \right] Q) \Rightarrow (\vdash P \left[ | pt D | \right] Q) \)

apply (simp add: Hoare-dgr1-def)
apply safe
apply (rule hoare-diagram3)
by auto

definition

dgr-demonic \( Q \ ij = |: Q \ ij |
\)

definition

dgr-demonic-mono[simp]

dmono (dgr-demonic \( Q \))
by (simp add: dmono-def dgr-demonic-def)

definition
\[
dangelic R Q i = \text{angelic} (R i) (Q i)
\]

**lemma** \( \text{grd-dgr} \):
\[
((\text{grd} (\text{step} D) i)::('a::complete-boolean-algebra)) = \bigcup \{P . \exists j . P = \text{grd} (D(i,j))\}
\]
apply (simp add: grd-def step-def)
apply (unfold step-def uminus-Inf)
apply (case-tac (uminus `' range (\(\lambda j::'b. D (i, j) \bot)\))) = \{P::'a. \exists j::'b. P = - D (i, j \bot)\}
apply (auto cong del: SUP-cong-simp)
done

**lemma** \( \text{grd-dgr-set} \):
\[
((\text{grd} (\text{step} D) i)::('a set)) = \text{Union} \{P . \exists j . P = \text{grd} (D(i,j))\}
\]
by (simp add: grd-dgr)

**lemma** \( \text{not-grd-dgr} \) [simp]: \((a \in (\neg \text{grd} (\text{step} D) i)) = (\forall j . a \notin \text{grd} (D(i,j)))\)
apply (simp add: grd-dgr)
by auto

**lemma** \( \text{not-grd-dgr2} \) [simp]: \((a \notin \text{grd} (\text{step} D) i) = (\forall j . a \notin \text{grd} (D(i,j)))\)
apply (subst not-grd-dgr [THEN sym])
by simp

end

### 6 Data Refinement of Diagrams

**theory** DataRefinement
**imports** Diagram
**begin**

Next definition introduces the concept of data refinement of \(S_1\) to \(S_2\) using the data abstractions \(R\) and \(R'\).

**definition**
\[
\text{DataRefinement} ::= ('a::type \Rightarrow 'b::type) \Rightarrow ('b::type \Rightarrow 'c::ord) \Rightarrow ('a::type \Rightarrow 'd::type) \Rightarrow ('d::type \Rightarrow 'c::ord) \Rightarrow \text{bool where}
\]
\[
\text{DataRefinement} S1 R R' S2 = ((R o S1) \leq (S2 o R'))
\]

If demonic \(Q\) is correct with respect to \(p\) and \(q\), and \((\text{assert} p) \circ (\text{demonic} Q)\) is data refined by \(S\), then \(S\) is correct with respect to angelic \(R p\) and angelic \(R' q\).

**theorem** data-refinement:
\[
\text{mono} R \Longrightarrow \parallel p \{S\} q \Longrightarrow \text{DataRefinement} S R R' S' \Longrightarrow \]
\[
\parallel (R p) \{S'\} (R' q)
\]
apply (simp add: DataRefinement-def Hoare-def le-fun-def)
apply (drule-tac x = \(q\) in spec)
apply (rule-tac y = \(R (S q)\) in order-trans)
apply (drule-tac x = \(p\) and \(y = S q\) in monoD)
by simp-all

**Theorem: data-refinement2**

\[
\text{mono } R \implies \models p \land S \models q \implies \text{DataRefinement } (\{p\} \circ R) \circ S \models q \implies DataRefinement (\{p\} \circ R) \circ S \models q.
\]

**Proof:**

- **apply (simp add: DataRefinement-def Hoare-def le-fun-def assert-def)**
- **apply (drule-tac x = q in spec)**
- **apply (subgoal-tac p \land S = p)**
- **apply simp**
- **apply (rule antisym)**
- **by simp-all

**Theorem: data-refinement-hoare**

\[
\text{mono } S \implies \text{mono } D \implies \text{DataRefinement } (\{p\} \circ \{Q\}) \circ D \circ S = (\forall s . \models \{s' . s \in R s' \land s \in p\} \land S \models (D ((Q s):\{w:order\})).
\]

**Proof:**

- **apply (simp add: le-fun-def assert-def angelic-def demonic-def Hoare-def DataRefinement-def)**
- **apply safe**
- **apply (simp-all)**
- **apply (drule-tac x = Q s in spec)**
- **apply auto [1]**
- **apply (drule-tac x = xb in spec)**
- **apply simp**
- **apply (simp add: less-eq-set-def le-fun-def)**
- **apply (drule-tac x = xa in spec)**
- **apply (simp-all add: mono-def)**
- **by auto

**Theorem: data-refinement-choice1**

\[
\text{DataRefinement } S1 D D' \implies \text{DataRefinement } S1 D D' \implies \text{DataRefinement } (S1 \land S1') D D' (S2 \land S2') \implies \text{DataRefinement } S1 D D' (S2 \land S2')
\]

**Proof:**

- **by (simp add: DataRefinement-def hoare-choice le-fun-def inf-fun-def)**

**Theorem: data-refinement-choice2**

\[
\text{mono } D \implies \text{DataRefinement } S1 D D' \implies \text{DataRefinement } S1' D D' S2' \implies \text{DataRefinement } (S1 \land S1') D D' (S2 \land S2')
\]

**Proof:**

- **apply (simp add: DataRefinement-def inf-fun-def le-fun-def)**
- **apply safe**
- **apply (rule-tac y = D (S1 x) in order-trans)**
- **apply (drule-tac x = S1 x \land S1' x and y = S1 x in monoD)**
- **apply simp-all**
- **apply (rule-tac y = D (S1' x) in order-trans)**
- **apply (drule-tac x = S1 x \land S1' x and y = S1' x in monoD)**
- **by simp-all

**Theorem: data-refinement-top [simp]:**
DataRefinement S1 D D’ (⊥::order-top)  

by (simp add: DataRefinement-def le-fun-def top-fun-def)

definition apply-fun::('a⇒'b⇒'c)⇒('a⇒'b)⇒'a⇒'c (infixl .. 5) where  
(A .. B) = (λ x . (A x) (B x))

definition Disjunctive-fun R = (∀ i . (R i) ∈ Apply.Disjunctive)

lemma Disjunctive-Sup:  
Disjunctive-fun R ⇒ (R .. (Sup X)) = Sup {y . ∃ x ∈ X . y = (R .. x)}  
apply (subst fun-eq-iff)
apply (simp add: apply-fun-def)
apply safe
apply (subst (asm) Disjunctive-fun-def)
apply (drule-tac x = x in spec)
apply (simp add: Apply.Disjunctive-def)
apply (subgoal-tac (R x ‘ (λf. f x) ‘ X) = ((λf. f x) ‘ {y. ∃ x∈X. y = (λxa. R xa (x xa))}))
apply (auto simp add: image-image cong del: SUP-cong-simp)
done

lemma (in DiagramTermination) disjunctive-SUP-L-P:  
Disjunctive-fun R ⇒ (R .. (SUP-L-P P (pair u i))) = (SUP-L-P (λ w . (R .. (P w))) (pair u i))  
by (simp add: SUP-L-P-def apply-fun-def Disjunctive-fun-def Apply.Disjunctive-def fun-eq-iff image-comp)

lemma apply-fun-range: {y. ∃ x. y = (R .. P x)} = range (λ x . R .. P x)  
by (fact full-SetCompr-eq)

lemma [simp]: Disjunctive-fun R ⇒ mono ((R i):'a::complete-lattice⇒'b::complete-lattice)  
by (simp add: Disjunctive-fun-def)

theorem (in DiagramTermination) dgr-data-refinement-1:  
dmono D’ ⇒ Disjunctive-fun R ⇒⇒  
(∀ w i j . P w i { D(i,j) } SUP-L-P P (pair w i) j)  
⇒ (∀ w i j . DataRefinement ((assert (P w i)) o (D (i,j))) (R i) (R j) (D’ (i, j)))  
⇒⇒  
(∀ w i j . P w i { D’ } ((R .. (Sup (range P))) ∩ (- (grd (step D’)))))  
apply (simp add: Disjunctive-Sup apply-fun-range)
apply (rule hoare-diagram2)
apply simp-all
apply safe
apply (simp add: disjunctive-SUP-L-P [THEN sym])
apply (simp add: apply-fun-def)
apply (rule-tac S = D (i, j) in data-refinement2)

17
by auto

definition
DgrDataRefinement1 D R D' = (∀ i j. DataRefinement (D i j) (R i) (R j) (D' (i, j)))

definition
DgrDataRefinement2 P D R D' = (∀ i j. DataRefinement ({.P i.} o D i j) (R i) (R j) (D' (i, j)))

theorem DataRefinement-mono:
T ≤ S ⇒ mono R ⇒ DataRefinement S R R' S' ⇒ DataRefinement T R R' S'
apply (simp add: DataRefinement-def mono-def)
apply (subst le-fun-def)
apply (simp add: le-fun-def)
apply safe
apply (rule-tac y = R (S x) in order-trans)
by simp-all

definition
mono-fun R = (∀ i. mono (R i))

theorem DgrDataRefinement-mono:
Q ≤ P ⇒ mono-fun R ⇒ DgrDataRefinement2 P D R D' ⇒ DgrDataRefinement2 Q D R D'
apply (simp add: DgrDataRefinement2-def)
apply auto
apply (rule-tac S = {.P i.} o D(i, j) in DataRefinement-mono)
apply (simp-all add: le-fun-def assert-def)
apply safe
apply (rule-tac y = Q i in order-trans)
by (simp-all add: mono-fun-def)

Next theorem is the diagram version of the data refinement theorem. If the diagram demonic choice T is correct, and it is refined by D, then D is also correct. One important point in this theorem is that if the diagram demonic choice T terminates, then D also terminates.

theorem (in DiagramTermination) Diagram-DataRefinement1:
dmono D ⇒ Disjunctive-fun R ⇒ ⊢ P {|| D ||} Q ⇒ DgrDataRefinement1 D R D'⇒
⊢ (R .. P) {|| D' ||} ((R .. P) ∩ (¬(grd (step D'))))
apply (unfold Hoare-dgr-def DgrDataRefinement1-def dgr-demonic-def)
apply safe
apply (rule-tac x=λ w . R .. (X w) in exl)
apply safe
apply (unfold disjunctive-SUP-L-P [THEN sym])
apply (simp add: apply-fun-def)
apply (rule-tac S = D(i, j) and R = R i and R' = R j in data-refinement)
by \((\text{simp-all add: } \text{Disjunctive-Sup apply-fun-range})\)

\textbf{lemma} \(\text{comp-left-mono [simp]: } S \leq S' \implies S \circ T \leq S' \circ T\)

by \((\text{simp add: le-fun-def})\)

\textbf{lemma} \(\text{assert-pred-mono [simp]: } p \leq q \implies \{.p.\} \leq \{.q.\}\)

apply \((\text{simp add: le-fun-def assert-def})\)
apply safe
apply \((\text{rule-tac } y = p \text{ in order-trans})\)
by \text{simp-all}

\textbf{theorem} \((\text{in DiagramTermination}) \text{Diagram-DataRefinement2:}\)
\(\text{dmono } D \implies \text{Disjunctive-fun } R \implies \vdash P \{\| D \|\} Q \implies \text{DgrDataRefinement2 } P\)
\(D \circ R \circ D' \implies \)
\(\vdash (R \.. P) \{\| D' \|\} ((R \.. P) \cap (\neg (\text{grd } (\text{step } D'))))\)
apply \((\text{unfold Hoare-dgr-def DgrDataRefinement2-def dgr-demonic-def})\)
apply auto
apply \((\text{rule-tac } x = \lambda w . R \.. (X w) \text{ in exI})\)
apply safe
apply \((\text{unfold disjunctive-SUP-L-P [THEN sym])})\)
apply \((\text{simp add: apply-fun-def})\)
apply \((\text{rule-tac } S = D (i,j) \text{ and } R = R i \text{ and } R' = R j \text{ in data-refinement2})\)
apply \((\text{simp-all add: Disjunctive-Sup})\)
apply \((\text{rule-tac } S = \{\text{Sup } (\text{range } X) i.\} \circ D (i, j) \text{ in DataRefinement-mono})\)
apply \((\text{rule comp-left-mono})\)
apply \((\text{rule assert-pred-mono})\)
apply \((\text{simp add: Sup-fun-def comp-def})\)
apply \((\text{rule SUP-upper})\)
apply \((\text{auto simp add: apply-fun-def apply-fun-range image-image fun-eq-iff})\)
apply \((\text{auto intro: arg-cong [where } f = \text{Sup} ] \text{arg-cong2 [where } f = \text{inf}]\))
done

\textbf{lemma} \((R':a::\text{complete-lattice} \implies 'b::\text{complete-lattice}) \in \text{Apply,Disjunctive} \implies \)
\(\text{DataRefinement } S \circ R' \circ S' \implies R (\neg \text{grd } S) \leq \neg \text{grd } S'\)
apply \((\text{simp add: DataRefinement-def grd-def le-fun-def})\)
apply \((\text{drule-tac } x = \perp \text{ in spec})\)
by \text{simp}

end

References


