Semantics and Data Refinement of Invariant Based Programs

Viorel Preoteasa and Ralph-Johan Back

September 19, 2015

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

\begin{verbatim}
theory Preliminaries
imports Main ../LatticeProperties/Complete-Lattice-Prop
   ../LatticeProperties/Conj-Disj
begin

notation
  less-eq (infix \sqsubseteq 50) and
  less (infix \sqsubseteq 50) and
  inf (infixl \sqcap 70) and
  sup (infixl \sqcup 65) and
  top (\top) and
  bot (\bot) and
  Inf (\cap - [900] 900) and
  Sup (\cup - [900] 900)

2.1 Simplification Lemmas
declare fun-upd-idem[simp]

lemma simp-eq-emptyset:
\end{verbatim}
(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.
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 ⇒ 'a ⇒ 'a (\[.\]-\[.\] \[0\] 1000) where \[.\] p . q ≡ - p ⊔ 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 $Q : State → Sate → bool$, 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 ⇒ 'b::ord) ⇒ 'b::ord ⇒ 'a set (\[\[\]]\[\[\]]\[0\] 1000) where \[\[\]\:
  p = \{ s . Q s ≤ p\}

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

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

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

theorem demonic-sup-inf:
  [:Q ∪ Q'] = [:Q] ∩ [:Q']
by (simp add: fun-eq-iff sup-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 ⇒ 'b::{semilattice-inf,order-bot}) ⇒ 'b ⇒ 'a set
  {Q::}{p = {s . (Q s) ∩ p ≠ ⊥}}

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::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 iniatial states from which \( S \) is enabled or fails.

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

lemma grd-choice[simp]:
  grd (S ∩ T) = (grd S) ∪ (grd T)
by (simp add: grd-def inf-fun-def)

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

lemma grd-demonic-2[simp]:
  (s ∉ grd [:Q]) = (∀ s'. s' ∉ (Q s))
by (simp add: grd-demonic)

theorem grd-angelic:
grd \{R:\} = 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 \(S : \text{State set} \to \text{State set}\) 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 Hoare :: 'a::complete-distrib-lattice ⇒ \'(b ⇒ 'a) ⇒ 'b ⇒ bool (\(\models (-)(| - |)(-)
\[0,0;900\])\) where
\(\models p \{S\} q = (p \leq (S q))\)

theorem hoare-sequential:
\(\text{mono } S \Rightarrow (\models p \{S o T\} r) = (\exists q. \models p \{S\} q \land \models q \{| T |\} r)\)
by (metis (no-types) Hoare-def monoD o-def order-refl order-trans)

theorem hoare-choice:
\(\models p \{| S \cap T\} q = (\models p \{S\} q \land \models p \{| T |\} q)\)
by (simp-all add: Hoare-def inf-fun-def)

theorem hoare-assume:
\(\models P \{| .R.\} Q = (P \cap R \leq Q)\)
apply (simp add: Hoare-def assume-def)
apply safe
apply (case-tac (inf P R) \leq (inf (sup (¬ R) Q) R))
apply (simp add: inf-sup-distrib2)
apply (simp add: le-infl1)
apply (case-tac (sup (¬ R) (inf P R)) \leq sup (¬ R) Q)
apply (simp add: sup-inf-distrib1)
by (simp add: le-supfl2)

theorem hoare-mono:
\(\text{mono } S \Rightarrow Q \leq R \Rightarrow \models P \{| S |\} Q \Rightarrow \models P \{| S |\} R\)
apply (simp add: Hoare-def)
apply (rule-tac y = $S$ $Q$ in order-trans)
by auto

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

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

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

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

lemma hoare-not-guard:
mono $(S :: (\_:\text{order-bot}) \Rightarrow \_)) \implies |p |\{S\} q = |(p \cup (\_ \_ \_ g r d S)) |\{S\} q$
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 $|p |\{S'\}q$ is true for all $p$ and $q$ such that $|p |\{S\}q$ 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-wf-induction:
mono $f \Rightarrow (\forall w . (p w) \leq f (\Sup-less p w)) \implies \Sup (\range p) \leq \lfp f$
apply (rule fp-wf-induction, simp-all)
by (drule lfp-unfold, simp)

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

lemma post-mono [simp]: mono (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 p = \top
by (simp add: post-fun-def)

lemma post-refin [simp]: mono $S \implies ((S p)::'a::bounded-lattice) \sqcap (post-fun p) x \leq S x$
apply (simp add: le-fun-def post-fun-def, safe)
by (rule-tac $f = S \in \text{monoD}$, simp-all)

Next theorem shows the equivalence between the validity of Hoare triples and refinement statements. This theorem together with the theorem for refinement of recursive programs will be used to prove a Hoare rule for recursive programs.

theorem hoare-refinement-post:
mono $f \implies (\{ x \} \{ f \} y) = (\{ x \} o (\text{post-fun} y) \leq f)$
apply safe
apply (simp-all add: Hoare-def)
apply (simp-all add: le-fun-def)
apply (simp add: assert-def, safe)
apply (rule-tac $y = f y \sqcap \text{post-fun} y xa \in \text{order-trans}$, simp-all)
apply (rule-tac $y = x \in \text{order-trans}$, simp-all)
apply (simp add: assert-def)
by (rule-tac $x = y \in \text{spec}$, simp)

Next theorem gives a Hoare rule for recursive programs. If we can prove correct the unfolding of the recursive definition applid to a program $f$,
\[ p \vdash w \{\{ F f \}\} y, \text{assuming that } f \text{ is correct when starting from } p \; v, \; v < w, \vdash SUP - L \; p \; w \{\{ f \}\} y, \text{then the recursive program is correct } \vdash SUP \; p \{\{ lfp F \}\} y \]

lemma assert-Sup: \[\bigsqcup \{ X::'a::complete-distrib-lattice set \}\} = \bigsqcup \{ \text{assert } ' \; X \}
by (simp add: fun-eq-iff assert-def Sup-inf)

lemma assert-Sup-range: \[\bigsqcup \{ \text{range } (p::'W \Rightarrow 'a::complete-distrib-lattice) \}\} = \bigsqcup \{ \text{range } (\text{assert } o \; p) \}
by (simp add: fun-eq-iff assert-def SUP-inf)

lemma Sup-range-comp: \[\bigsqcup \{ \text{range } p \} o S = \bigsqcup \{ \text{range } (\lambda w . ((p \; w) o S)) \}
by (simp add: fun-eq-iff)

lemma Sup-less-comp: \((\text{Sup-less } P) \; w \; S = \text{Sup-less } (\lambda w . ((p \; w) o S)) \) w
apply (simp add: Sup-less-def fun-eq-iff, safe)
apply (subgoal-tac \((\lambda f . (S x)) \) \{ y. \exists v<w. \; \forall x. y \; x = P \; v \; x)\}) = \((\lambda f . (S x)) \) \{ y. \exists v<w. \; \forall x. y \; x = P \; v \; (S x)\})
by (auto simp add: SUP-def simp del: Sup-image-eq)

lemma Sup-less-assert: \((\text{Sup-less } (\lambda w . \{ p \; w\}):'a::complete-distrib-lattice .)\) 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) \land y = \{ x.\} \})
by auto

declare mono-comp[simp]

theorem hoare-fixpoint:
  mono-mono F \implies
  (!! w f . mono f \land \models Sup p w \{ \{ f \} \} y \implies \models p w \{ \{ F f \} \} y) \implies \models (Sup (range p))\{ \{ lfp F \} \} y
apply (simp add: mono-mono-def hoare-refinement-post assert-Sup-range Sup-range-comp
del: Sup-image-eq)
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 .\} o post-fun y in spec, safe)
apply simp
by (simp add: hoare-refinement-post)

theorem (\forall t . \models (\{ s . t \in R s \}) \{ \{ S \} \} q \implies \models (\{ :)R \} p \{ \{ S \} \} q
apply (simp add: Hoare-def angelic-def subset-eq)
by auto

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: \( D : I \times I \rightarrow (\text{State set} \rightarrow \text{State set}) \), or more general \( D : I \times I \rightarrow \text{Ptran} \), 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 \((I \rightarrow \text{State set})\). If \( Q : I \rightarrow \text{State set} \) is an indexed predicate, then \( p = pt D Q i \) 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**

\[
d\text{mono} \ D = (\forall i j. \text{mono} (D ij))
\]

**lemma** \( d\text{mono-\text{mono}} \ [\text{simp}]: \text{d\text{mono} \ D \implies \text{\text{mono} (D ij)} \]

by \( (\text{simp \ add: d\text{mono-def})} \)

**theorem** \( \text{mono-step} \ [\text{simp}]: \)

\[
d\text{mono} \ D \implies \text{\text{mono} (step D)}
\]

apply \( (\text{simp \ add: d\text{mono-def \ mono-def \ le-fun-def \ step-def \ Inf-fun-def})} \)

apply auto

apply \( (\text{rule \ INF-greatest}) \)

apply auto

apply \( (\text{rule-tac y = D(xa, j) (x j) in order-trans}) \)

apply auto

apply \( (\text{rule \ INF-lower}) \)

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 \( \text{assume} \neg \text{grd (step D)} \).

**definition**

\[
\text{dgr} \ D \ U = ((\text{step} \ D) \circ U) \cap [.\neg (\text{grd (step D)}).]
\]

**theorem** \( \text{mono-\text{mono-dgr} \ [\text{simp}]: \text{d\text{mono} \ D \implies \text{\text{mono \-min (dgr D})} \]

apply \( (\text{simp \ add: mono-\text{mono-def \ mono-def})} \)

apply safe

apply \( (\text{simp-all \ add: dgr-def}) \)

apply \( (\text{simp-all \ add: le-fun-def \ inf-fun-def}) \)

apply safe

apply \( (\text{rule-tac y = dgr D (xa) xb \ in order-trans}) \)

apply simp-all

apply \( (\text{case-tac mono (step D)}) \)

apply \( (\text{simp \ add: mono-def}) \)

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

apply simp

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

apply simp-all

apply \( (\text{case-tac mono (step D)}) \)
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 → 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:**
\( (\forall \ i \ j . \models (P \ i) \{ (D(i,j)) \} (Q \ j)) \) \( = \) \( (\models P \{ step D \} \ Q) \)
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 correctness 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:**
\( \text{dmono D } \implies (\forall \ w \ i \ j . \models X w i \{ (D(i,j)) \} \text{ Sup-less X w j } \implies \) \( \models (\text{Sup(range X)}) \{ \text{pt D} \} (\text{Sup(range X)} \cap -(\text{grd}(\text{step D})))) \)
apply (simp add: hoare-step pt-def del: Sup-image-eq)
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-infI1)
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 \( \Rightarrow Y \ i \land \ t \ i = w \{D(i,j)\} \ Y \ i \land \ t \ i < w \). 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 every 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 procedures in which the variant is required to decrease only in a sequence of recursive calls before calling again a procedure in this sequence. We introduce 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 \( SUP-L-P X u i \) = (\( SUP v:\{v. \ pair v i < u\} . X v i :: - :: complete-lattice\))

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

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

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

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

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

lemma \( SUP-SUP-L [simp]\): \( Sup (range (SUP-LE-P X)) = Sup (range X) \)
apply (simp add: fun-eq_iff Sup-fun-def, clarify)
apply \((\text{rule antisym})\)
apply \((\text{rule SUP-least})\)
unfolding comp-def
apply \((\text{rule SUP-LE-P-least})\)
apply \((\text{rule SUP-upper, simp})\)
apply \((\text{rule SUP-least})\)
apply \((\text{rule-tac } y = \text{SUP-LE-P } x = \text{x in order-trans})\)
apply \((\text{rule SUP-LE-P-upper, simp})\)
by \((\text{rule SUP-upper, simp})\)

lemma \(\text{SUP-L-SUP-LE-P [simp]}\): \(\text{Sup-less (SUP-LE-P } X) = \text{SUP-L-P } X\)
apply \((\text{rule antisym})\)
apply \((\text{subst le-fun-def, safe})\)
apply \((\text{rule Sup-less-least})\)
apply \((\text{subst le-fun-def, safe})\)
apply \((\text{rule SUP-LE-P-least})\)
apply \((\text{rule SUP-L-P-upper, simp})\)
apply \((\text{simp add: le-fun-def, safe})\)
apply \((\text{rule SUP-L-P-least})\)
apply \((\text{rule-tac } y = \text{SUP-LE-P } x = \text{x in order-trans})\)
apply \((\text{rule SUP-LE-P-upper, simp})\)
apply \((\text{cut-tac } P = \text{SUP-LE-P } X \text{ in SUP-less-upper})\)
by \((\text{simp, simp add: le-fun-def})\)

end

theorem \((\text{in DiagramTermination) hoare-diagram2})\:
dmono D \implies (\forall u i j . \models X u i \{ D(i, j) \} \text{SUP-L-P } X (\text{pair u i }) j) \implies
\models (\text{Sup (range X)}) \{ pt D \} (Sup (range X)) \cap (\neg (\text{grd (step D)}))\)
apply \((\text{frule-tac } X = \text{SUP-LE-P } X \text{ in hoare-diagram})\)
apply \((\text{auto simp del: Sup-image-eq})\)
apply \((\text{simp add: SUP-LE-P-def})\)
apply \((\text{unfold SUP-def hoare-Sup [THEN sym]})\)
apply auto
apply \((\text{rule-tac } Q = \text{SUP-L-P } X (\text{pair p i }) j \text{ in hoare-mono})\)
apply auto
apply \((\text{rule SUP-L-P-least})\)
apply \((\text{rule SUP-L-P-upper})\)
apply \((\text{rule order-trans3})\)
by auto

lemma mono-pt [simp]: dmono D \implies mono (pt D)
apply \((\text{drule mono-mono-dgr})\)
by \((\text{simp add: pt-def})\)

theorem \((\text{in DiagramTermination) hoare-diagram3})\:
dmono D \implies
(\forall u i j . \models X u i \{ D(i, j) \} \text{SUP-L-P } X (\text{pair u i }) j) \implies
P \leq \text{Sup (range X)} \implies ((\text{Sup (range X)}) \cap (\neg (\text{grd (step D)})) \leq Q \implies
apply (rule hoare-mono)
apply auto
apply (rule hoare-pre)
apply (auto simp add: SUP-def simp del: Sup-image-eq)
apply (rule hoare-diagram2)
by auto

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

definition (in DiagramTermination)
Hoare-dgr :: ('b ⇒ ('u::{complete-distrib-lattice, boolean-algebra})) ⇒ ('b × 'b ⇒ 'u ⇒ 'u) ⇒ bool (⊢ (| - |)(-)
[0,0,900] 900) where
⊢ P {| D |} Q ≡ (∃ X . (∀ u i j . ⊨ D(u,i) j) ∧ P = Sup (range X) ∧ Q = ((Sup (range X)) ∩ (-(grd (step D)))))

definition (in DiagramTermination)
Hoare-dgr1 :: ('b ⇒ ('u::{complete-distrib-lattice, boolean-algebra})) ⇒ ('b × 'b ⇒ 'u ⇒ 'u) ⇒ bool (⊢1 (-)(| - |)(-)
[0,0,900] 900) where
⊢1 P {| D |} Q ≡ (∃ X . (∀ u i j . ⊨ D(u,i) j) ∧ P ≤ Sup (range X) ∧ ((Sup (range X)) ∩ (-(grd (step D))))) ≤ Q

theorem (in DiagramTermination) hoare-dgr-correctness:
dmono D ⇒ (⊢ P {| D |} Q) ⇒ (⊢ P {| pt D |} Q)
apply (simp add: Hoare-dgr-def)
apply safe
apply (rule hoare-diagram3)
by auto

theorem (in DiagramTermination) hoare-dgr-correctness1:
dmono D ⇒ (⊢1 P {| D |} Q) ⇒ (⊢ P {| pt D |} Q)
apply (simp add: Hoare-dgr1-def)
apply safe
apply (rule hoare-diagram3)
by auto

definition
dgr-demonic Q ij = [:Q ij:]

theorem dgr-demonic-mono[simp]:
dmono (dgr-demonic Q)
by (simp add: dmono-def dgr-demonic-def)

definition
dangelic \( R \) \( Q \) \( i = \) 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 INF-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
done

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

**Lemma** \( \text{not-grd-dgr} \) [simp]: \(a \in (- \text{grd} \ (\text{step} \ D) \ i)) = (\forall \ j . a \notin \text{grd} (D(i,j)))\)
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
\[
\text{DataRefinement} \ S_1 \ R \ R' \ S_2 = ((R \circ S_1) \leq (S_2 \circ R'))
\]

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:
\[
\text{mono} \ R \implies \models \ p \ [\ | \ S \ | ] q \implies \text{DataRefinement} \ S \ R \ R' \ S' \implies \\
\models \ (R \ p) \ [\ | \ S' \ | ] (R' \ q)
\]
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 \( \text{monoD} \))
by simp-all

theorem data-refinement2:
\[
\text{mono } R \implies \models p \{ \| S \| \} q \implies \text{DataRefinement} (\{p\} \circ S) R R' S' \implies \models (R p) \{ \| S' \| \} (R' q)
\]
apply (simp add: DataRefinement-def Hoare-def le-fun-def assert-def)
apply (drule-tac \( x = q \) in spec)
apply (subgoal-tac \( p \cap S q = p \))
apply simp
apply (rule antisym)
by simp-all

theorem data-refinement-hoare:
\[
\text{mono } S \implies \text{mono } D \implies \text{DataRefinement} (\{p\} \circ \{R\}) \circ D S = \big( \forall s . \models (s'. s \in R s' \land s \in p) \{ \| S \| \} (D ((Q s) :: 'a :: order)) \big)
\]
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:
\[
\text{DataRefinement } S1 D D' S2 \implies \text{DataRefinement } S1 D D' S2' \implies \text{DataRefinement } S1 D D' (S2 \cap S2')
\]
by (simp add: DataRefinement-def hoare-choice le-fun-def inf-fun-def)

theorem data-refinement-choice2:
\[
\text{mono } D \implies \text{DataRefinement } S1 D D' S2 \implies \text{DataRefinement } S1' D D' S2'
\]
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 \( x = S1 x \cap S1' x \) and \( y = S1 x \) in \( \text{monoD} \))
apply simp-all
apply (rule-tac \( y = D (S1' x) \) in order-trans)
apply (drule-tac \( x = S1 x \cap S1' x \) and \( y = S1' x \) in \( \text{monoD} \))
by simp-all

theorem data-refinement-top [simp]:
DataRefinement S1 D D' (\leadsto::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 \imp (R .. (Sup X)) = Sup {x .. y ∈ 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) =\{y .. ∃ x∈X . y = (λxa. R xa (x xa)))\})
by (auto simp add: SUP-def simp del: Sup-image-eq)

lemma (in DiagramTermination) disjunctive-SUP-L-P:
Disjunctive-fun R \imp (R .. (SUP-L-P P (pair u i))) = (SUP-L-P (λ w . (R .. (P w)))) (pair u i)
apply (subst fun-eq-iff)
apply (simp add: SUP-L-P-def apply-fun-def Disjunctive-fun-def Apply.Disjunctive-def, safe)
apply (subgoal_tac (R x · (λv. P v x) · \{v .. pair v x ⊆ pair u i\}) =
       ((λv. R x (P v x)) · \{v .. pair v x ⊆ pair u i\}))
by (auto simp add: SUP-def simp del: Sup-image-eq)

lemma apply-fun-range: \{y .. ∃ x . y = (R .. P x)\} = range (λ x .. R .. P x)
by auto

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

theorem (in DiagramTermination) dgr-data-refinement-1:
dmono D' \imp Disjunctive-fun R \imp
(∀ w i j . \{ w i \} \{ D(i,j) \} SUP-L-P P (pair w i) j \imp
(∀ w i j . DataRefinement ((assert (P w i)) o (D (i,j))) (R i) (R j) (D' (i, j))))) \imp
\{ (R .. (Sup (range P))) \} \{ pt D' \} ((R .. (Sup (range P)))) \cap (-{grd (step D')}))
apply (simp add: Disjunctive-Sup apply-fun-range del: Sup-image-eq)
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)
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 exI)
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 (simp-all add: Disjunctive-Sup apply-fun-range del: Sup-image-eq)

lemma comp-left-mono [simp]: S ≤ S' ⇒ S o T ≤ S' o T
by (simp add: le-fun-def)

lemma assert-pred-mono [simp]: p ≤ q ⇒ {.p.} ≤ {.q.}
apply (simp add: le-fun-def assert-def)
apply safe
apply (rule-tac y = p in order-trans)
by simp-all

theorem (in DiagramTermination) Diagram-DataRefinement2:
dmono D ⇒ Disjunctive-fun R ⇒ P || D || Q ⇒ DgrDataRefinement2 P D R D' ⇒
  ⊢ (R .. P) || D' || ((R .. P) ∩ (¬(grd (step D'))))
apply (unfold Hoare-dgr-def DgrDataRefinement2-def dgr-demonic-def)
apply auto
apply (rule-tac x=λ w . R .. (X w) in exI)
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-refinement2)
apply (simp-all add: Disjunctive-Sup)
apply (rule-tac S = {.Sup (range X) i.} o D (i, j) in DataRefinement-mono)
apply (rule comp-left-mono)
apply (rule assert-pred-mono)
apply (simp add: Sup-fun-def comp-def)
apply (rule SUP-upper)
apply (auto simp add: apply-fun-def apply-fun-range SUP-def image-image fun-eq-iff simp del: Sup-image-eq)
apply (metis Apply.DisjunctiveD Disjunctive-fun-def range-composition)+
done

lemma (R':·a::complete-lattice ⇒ ·b::complete-lattice) ∈ Apply.Disjunctive ⇒
  DataRefinement S R R' S' ⇒⇒ R (¬ grd S) ≤ ¬ grd S' 
apply (simp add: DataRefinement-def grd-def le-fun-def)
apply (drule-tac x = ⊥ in spec)
by simp

end
References


