Semantics and Data Refinement of Invariant Based Programs

Viorel Preoteasa and Ralph-Johan Back

March 19, 2025

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 4
3.5 The guard of a statement 5

4 Hoare Triples **5**
4.1 Hoare rule for recursive statements 7

5 Predicate Transformers Semantics of Invariant Diagrams **8**

6 Data Refinement of Diagrams **12**

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 sematics for invariant based programs and their data refinement. The complete treatment of the sematics 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 invariand 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*:

$(X = \{\}) = (\forall\ x.\ x \notin X)$
⟨*proof*⟩

lemma *mono-comp*: $mono\ f \Longrightarrow mono\ g \Longrightarrow mono\ (f\ o\ g)$
⟨*proof*⟩

Some lattice simplification rules

lemma *inf-bot-bot*:
$(x::'a::\{semilattice\text{-}inf,order\text{-}bot\}) \sqcap \bot = \bot$
⟨*proof*⟩

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 $q \in State\ set$, 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*
⟨*proof*⟩

lemma *mono-choice*[*simp*]: $mono\ S \Longrightarrow mono\ T \Longrightarrow mono\ (S \sqcap T)$
⟨*proof*⟩

3.1 Assert statement

The assert statement of a predicate p when executed from a state s fails if $s \notin p$ and behaves as skip otherwise.

definition
assert$::'a::semilattice\text{-}inf \Rightarrow 'a \Rightarrow 'a$ (‹{. - .}› [*0*] *1000*) **where**
$\{.p.\}\ q \equiv\ p \sqcap q$

lemma *mono-assert* [*simp*]: *mono* $\{.p.\}$
⟨*proof*⟩

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* $\Rightarrow$ $'a \Rightarrow$ $'a$ (‹[. - .]› [0] 1000) **where**
$[.\ p\ .]\ q \equiv\ -p \sqcup q$

lemma *mono-assume* [*simp*]: *mono* (*assume P*)
⟨*proof*⟩

3.3 Demonic update statement

The demonic update statement of a relation $Q : State \to Sate \to bool$, 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*:]
⟨*proof*⟩

theorem *demonic-bottom*:
$[:R:]\ (\bot::('a::order\text{-}bot)) = \{s\ .\ (R\ s) = \bot\}$
⟨*proof*⟩

theorem *demonic-bottom-top* [*simp*]:
$[:(\bot::\text{-}::order\text{-}bot):]\ = \top$
⟨*proof*⟩

theorem *demonic-sup-inf*:
$[:Q \sqcup Q':] = [:Q:] \sqcap [:Q':]$
⟨*proof*⟩

3.4 Angelic update statement

The angelic update statement of a relation $Q : State \to State \to bool$ 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 :: ($'a \Rightarrow$ $'b$::{*semilattice-inf*,*order-bot*}) $\Rightarrow$ $'b \Rightarrow$ $'a$ *set*
(‹{: - :}› [0] 1000) **where**

{: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 ⊥ = {}
⟨*proof*⟩

theorem *angelic-disjunctive* [*simp*]:
{:(*R*::(*'a* ⇒ *'b*::*complete-distrib-lattice*)):} ∈ *Apply.Disjunctive*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *grd-demonic*: *grd* [:Q:] = {s . ∃ s' . s' ∈ (Q s) }
⟨*proof*⟩

lemma *grd-demonic-2*[*simp*]: (s ∉ *grd* [:Q:]) = (∀ s' . s' ∉ (Q s))
⟨*proof*⟩

theorem *grd-angelic*:
grd {:R:} = *UNIV*
⟨*proof*⟩

end

4 Hoare Triples

theory *Hoare*
imports *Statements*
begin

A hoare triple for $p, q \in \mathit{State\ set}$, and $S : \mathit{State\ set} \to \mathit{State\ set}$ 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* (‹⊨ (-){| - |}(-)› [*0*,*0*,*900*] *900*) **where**
⊨ *p* {|*S*|} *q* = (*p* ≤ (*S q*))

theorem *hoare-sequential*:
mono S ⟹ (⊨ *p* {| *S o T* |} *r*) = ((∃ *q*. ⊨ *p* {| *S* |} *q* ∧ ⊨ *q* {| *T* |} *r*))
⟨*proof*⟩

theorem *hoare-choice*:
⊨ *p* {| *S* ⊓ *T* |} *q* = (⊨ *p* {| *S* |} *q* ∧ ⊨ *p* {| *T* |} *q*)
⟨*proof*⟩

theorem *hoare-assume*:
(⊨ *P* {| [.*R*.] |} *Q*) = (*P* ⊓ *R* ≤ *Q*)
⟨*proof*⟩

theorem *hoare-mono*:
mono S ⟹ *Q* ≤ *R* ⟹ ⊨ *P* {| *S* |} *Q* ⟹ ⊨ *P* {| *S* |} *R*
⟨*proof*⟩

theorem *hoare-pre*:
R ≤ *P* ⟹ ⊨ *P* {| *S* |} *Q* ⟹ ⊨ *R* {| *S* |} *Q*
⟨*proof*⟩

theorem *hoare-Sup*:
(∀ *p* ∈ *P* . ⊨ *p* {| *S* |} *q*) = ⊨ *Sup P* {| *S* |} *q*
⟨*proof*⟩

lemma *hoare-magic* [*simp*]: ⊨ *P* {| ⊤ |} *Q*
⟨*proof*⟩

lemma *hoare-demonic*: ⊨ *P* {| [:*R*:] |} *Q* = (∀ *s* . *s* ∈ *P* ⟶ *R s* ⊆ *Q*)
⟨*proof*⟩

lemma *hoare-not-guard*:
mono (*S* :: (-::*order-bot*) ⇒ -) ⟹ ⊨ *p* {| *S* |} *q* = ⊨ (*p* ⊔ (− *grd S*)) {| *S* |} *q*
⟨*proof*⟩

4.1 Hoare rule for recursive statements

A statement S is refined by another statement S' if $\models p\{|S'|\}q$ is true for all p and q such that $\models p\{|S|\}q$ 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*:
  $mono\ f \Longrightarrow (\forall\ w\ .\ (p\ w) \leq f\ (Sup\text{-}less\ p\ w)) \Longrightarrow Sup\ (range\ p) \leq lfp\ f$
⟨*proof*⟩

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

lemma *post-mono* [*simp*]: $mono\ (post\text{-}fun\ p :: (\text{-}::\{order\text{-}bot,order\text{-}top\}))$
  ⟨*proof*⟩

lemma *post-top* [*simp*]: $post\text{-}fun\ p\ p = \top$
  ⟨*proof*⟩

lemma *post-refin* [*simp*]: $mono\ S \Longrightarrow ((S\ p)::'a::bounded\text{-}lattice) \sqcap (post\text{-}fun\ p)\ x \leq S\ x$
  ⟨*proof*⟩

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*:
  $mono\ f \Longrightarrow\ (\models x\ \{|\ f\ |\}\ y) = (\{.x.\}\ o\ (post\text{-}fun\ y) \leq f)$
  ⟨*proof*⟩

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, $\models p\ w\ \{|F\ f|\}\ y$, assumming that f is correct when starting from $p\ v$, $v < w$, $\models SUP - L\ p\ w\ \{|f|\}\ y$, then the recursive program is correct $\models SUP\ p\ \{|lfp\ F|\}\ y$

lemma *assert-Sup*: $\{.\bigsqcup\ (X::'a::complete\text{-}distrib\text{-}lattice\ set).\} = \bigsqcup\ (assert\ `\ X)$
  ⟨*proof*⟩

lemma *assert-Sup-range*: $\{.\bigsqcup\ (range\ (p::'W \Rightarrow 'a::complete\text{-}distrib\text{-}lattice)).\} = \bigsqcup\ (range\ (assert\ o\ p))$
  ⟨*proof*⟩

lemma *Sup-range-comp*: $(\bigsqcup\ range\ p)\ o\ S = \bigsqcup\ (range\ (\lambda\ w\ .\ ((p\ w)\ o\ S)))$
  ⟨*proof*⟩

lemma *Sup-less-comp*: $(Sup\text{-}less\ P)\ w\ o\ S = Sup\text{-}less\ (\lambda\ w\ .\ ((P\ w)\ o\ S))\ w$
⟨*proof*⟩

lemma *Sup-less-assert*: $Sup\text{-}less\ (\lambda w.\ \{.\ (p\ w){::}'a{::}complete\text{-}distrib\text{-}lattice\ .\})\ w = \{.Sup\text{-}less\ p\ w.\}$
⟨*proof*⟩

declare *mono-comp*[*simp*]

theorem *hoare-fixpoint*:
$mono\text{-}mono\ F \Longrightarrow$
$(!!\ w\ f\ .\ mono\ f \wedge \models Sup\text{-}less\ p\ w\ \{|\ f\ |\}\ y \Longrightarrow \models p\ w\ \{|\ F\ f\ |\}\ y) \Longrightarrow \models (Sup\ (range\ p))\ \{|\ lfp\ F\ |\}\ y$
⟨*proof*⟩

theorem $(\forall\ t\ .\ \models (\{s\ .\ t \in R\ s\})\ \{|S|\}\ q) \Longrightarrow \models (\{:R:\}\ p)\ \{|\ S\ |\}\ q$
⟨*proof*⟩

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 corectness 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: $D : I \times I \to (State\ set \to State\ set)$, or more general $D : I \times I \to Ptran$, where $Ptran$ 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 trasitions. 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 $(I \to State\ set)$. If $Q : I \to State\ set$ is an indexed predicate, then $p = pt\ D\ Q\ i$ is a weakest predicate such that if the executution 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 $step\ D$ of executing one step of diagram D. The predicate $step\ D\ Q\ i$ 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
step D Q i = (*INF j . D* (*i*, *j*) (*Q j*) :: - :: *complete-lattice*)

definition
dmono D = (∀ *ij . mono* (*D ij*))

lemma *dmono-mono* [*simp*]: *dmono D* ⟹ *mono* (*D ij*)
⟨*proof*⟩

theorem *mono-step* [*simp*]:
dmono D ⟹ *mono* (*step D*)
⟨*proof*⟩

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 *dgr D U* is the choice between executing one step of D follwed by U ((*step D*) ∘ U) or skip if no transion of D is enabled (*assume* ¬*grd*(*step D*)).

definition
dgr D U = ((*step D*) *o U*) ⊓ [.−(*grd* (*step D*)).]

theorem *mono-mono-dgr* [*simp*]: *dmono D* ⟹ *mono-mono* (*dgr D*)
⟨*proof*⟩

definition
pt D = *lfp* (*dgr D*)

If U is an indexed predicate transformer and if $P, Q : I \to \mathit{State\ set}$ 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*:
(∀ *i j* . ⊨ (*P i*) {| *D*(*i*,*j*) |} (*Q j*)) = (⊨ *P* {| *step D* |} *Q*)
⟨*proof*⟩

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 ⟹ (∀ *w i j* . ⊨ *X w i* {| *D*(*i*,*j*) |} *Sup-less X w j*) ⟹
⊨ (*Sup* (*range X*)) {| *pt D* |} (*Sup*(*range X*) ⊓ −(*grd* (*step D*)))

⟨*proof*⟩

This theorem is a more general form of the more familiar form with a variant t which must decrease. If we take $X\ w\ i = (Y\ i \wedge t\ i = w)$, then the second hypothesis of the theorem above becomes $\models Y\ i \wedge t\ i = w\{|D(i,j)|\}Y\ i \wedge t\ i < w$. However, the more general form of the theorem is needed, because in data refinements, the form $Y\ i \wedge t\ i = w$ 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* ≤ *u*}. *X v i* :: - :: *complete-lattice*)

lemma *SUP-L-P-upper*:
pair v i < *u* ⟹ *P v i* ≤ *SUP-L-P P u i*
⟨*proof*⟩

lemma *SUP-L-P-least*:
(!! *v* . *pair v i* < *u* ⟹ *P v i* ≤ *Q*) ⟹ *SUP-L-P P u i* ≤ *Q*
⟨*proof*⟩

lemma *SUP-LE-P-upper*:
pair v i ≤ *u* ⟹ *P v i* ≤ *SUP-LE-P P u i*
⟨*proof*⟩

lemma *SUP-LE-P-least*:
(!! *v* . *pair v i* ≤ *u* ⟹ *P v i* ≤ *Q*) ⟹ *SUP-LE-P P u i* ≤ *Q*
⟨*proof*⟩

lemma *SUP-SUP-L* [*simp*]: *Sup* (*range* (*SUP-LE-P X*)) = *Sup* (*range X*)
⟨*proof*⟩

lemma *SUP-L-SUP-LE-P* [*simp*]: *Sup-less* (*SUP-LE-P X*) = *SUP-L-P X*
⟨*proof*⟩

end

theorem (**in** *DiagramTermination*) *hoare-diagram2*:
dmono D ⟹ (∀ *u i j* . ⊨ *X u i* {| *D*(*i*, *j*) |} *SUP-L-P X* (*pair u i*) *j*) ⟹
⊨ (*Sup* (*range X*)) {| *pt D* |} ((*Sup* (*range X*)) ⊓ (−(*grd* (*step D*))))
⟨*proof*⟩

lemma *mono-pt* [*simp*]: *dmono D* ⟹ *mono* (*pt D*)
⟨*proof*⟩

theorem (**in** *DiagramTermination*) *hoare-diagram3*:
dmono D ⟹
(∀ *u i j* . ⊨ *X u i* {| *D*(*i*, *j*) |} *SUP-L-P X* (*pair u i*) *j*) ⟹
P ≤ *Sup* (*range X*) ⟹ ((*Sup* (*range X*)) ⊓ (−(*grd* (*step D*)))) ≤ *Q* ⟹
⊨ *P* {| *pt D* |} *Q*
⟨*proof*⟩

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*) ⇒ (′*b* ⇒ ′*u*) ⇒ *bool* (‹⊢ (-){| - |}(-) ›
[*0*,*0*,*900*] *900*) **where**
⊢ *P* {| *D* |} *Q* ≡ (∃ *X* . (∀ *u i j* . ⊨ *X u i* {| *D*(*i*, *j*) |} *SUP-L-P X* (*pair 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*) ⇒ (′*b* ⇒ ′*u*) ⇒ *bool* (‹⊢*1* (-){| - |}(-) ›
[*0*,*0*,*900*] *900*) **where**
⊢*1* *P* {| *D* |} *Q* ≡ (∃ *X* . (∀ *u i j* . ⊨ *X u i* {| *D*(*i*, *j*) |} *SUP-L-P X* (*pair 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*)
⟨*proof*⟩

theorem (**in** *DiagramTermination*) *hoare-dgr-correctness1*:
dmono D ⟹ (⊢*1* *P* {| *D* |} *Q*) ⟹ (⊨ *P* {| *pt D* |} *Q*)
⟨*proof*⟩

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

theorem *dgr-demonic-mono*[*simp*]:
$dmono\ (dgr\text{-}demonic\ Q)$
⟨*proof*⟩

definition
$dangelic\ R\ Q\ i = angelic\ (R\ i)\ (Q\ i)$

lemma *grd-dgr*:
$((grd\ (step\ D)\ i)::('a::complete\text{-}boolean\text{-}algebra)) = \bigsqcup\ \{P\ .\ \exists\ j\ .\ P = grd\ (D(i,j))\}$
⟨*proof*⟩

lemma *grd-dgr-set*:
$((grd\ (step\ D)\ i)::('a\ set)) = Union\ \{P\ .\ \exists\ j\ .\ P = grd\ (D(i,j))\}$
⟨*proof*⟩

lemma *not-grd-dgr* [*simp*]: $(a \in (-\ grd\ (step\ D)\ i)) = (\forall\ j\ .\ a \notin grd\ (D(i,j)))$
⟨*proof*⟩

lemma *not-grd-dgr2* [*simp*]: $a \notin (grd\ (step\ D)\ i) = (\forall\ j\ .\ a \notin grd\ (D(i,j)))$
⟨*proof*⟩
end

6 Data Refinement of Diagrams

theory *DataRefinement*
imports *Diagram*
begin

Next definition introduces the concept of data refinement of $S1$ to $S2$ using the data abstractions R and R'.

definition
$DataRefinement :: ('a::type \Rightarrow 'b::type)$
$\Rightarrow ('b::type \Rightarrow 'c::ord) \Rightarrow ('a::type \Rightarrow 'd::type)$
$\Rightarrow ('d::type \Rightarrow 'c::ord) \Rightarrow bool$ **where**
$DataRefinement\ S1\ R\ R'\ S2 = ((R\ o\ S1) \le (S2\ o\ R'))$

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

theorem *data-refinement*:
$mono\ R \Longrightarrow \models p\ \{|\ S\ |\}\ q \Longrightarrow\ DataRefinement\ S\ R\ R'\ S' \Longrightarrow$
$\models (R\ p)\ \{|\ S'\ |\}\ (R'\ q)$
⟨*proof*⟩

theorem *data-refinement2*:
$mono\ R \Longrightarrow \models p\ \{|\ S\ |\}\ q \Longrightarrow\ DataRefinement\ (\{.p.\}\ o\ S)\ R\ R'\ S' \Longrightarrow$
$\models (R\ p)\ \{|\ S'\ |\}\ (R'\ q)$

⟨*proof*⟩

theorem *data-refinement-hoare*:
$mono\ S \Longrightarrow mono\ D \Longrightarrow DataRefinement\ (\{.p.\}\ o\ [:Q:])\ \{:R:\}\ D\ S =$
$(\forall\ s\ .\ \models \{s'\ .\ s \in R\ s' \wedge s \in p\}\ \{|\ S\ |\}\ (D\ ((Q\ s)::'a::order)))$
⟨*proof*⟩

theorem *data-refinement-choice1*:
$DataRefinement\ S1\ D\ D'\ S2 \Longrightarrow DataRefinement\ S1\ D\ D'\ S2' \Longrightarrow DataRefinement\ S1\ D\ D'\ (\ S2 \sqcap S2')$
⟨*proof*⟩

theorem *data-refinement-choice2*:
$mono\ D \Longrightarrow DataRefinement\ S1\ D\ D'\ S2 \Longrightarrow DataRefinement\ S1'\ D\ D'\ S2' \Longrightarrow$

$DataRefinement\ (S1 \sqcap S1')\ D\ D'\ (S2 \sqcap S2')$
⟨*proof*⟩

theorem *data-refinement-top* [*simp*]:
$DataRefinement\ S1\ D\ D'\ (\top::\text{-}::order\text{-}top)$
⟨*proof*⟩

definition $apply\text{-}fun::('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ (**infixl** ‹..› *5*) **where**
$(A\ ..\ B) = (\lambda\ x\ .\ (A\ x)\ (B\ x))$

definition
$Disjunctive\text{-}fun\ R = (\forall\ i\ .\ (R\ i) \in Apply.Disjunctive)$

lemma *Disjunctive-Sup*:
$Disjunctive\text{-}fun\ R \Longrightarrow (R\ ..\ (Sup\ X)) = Sup\ \{y\ .\ \exists\ x \in X\ .\ y = (R\ ..\ x)\}$
⟨*proof*⟩

lemma (**in** *DiagramTermination*) *disjunctive-SUP-L-P*:
$Disjunctive\text{-}fun\ R \Longrightarrow (R\ ..\ (SUP\text{-}L\text{-}P\ P\ (pair\ u\ i))) = (SUP\text{-}L\text{-}P\ (\lambda\ w\ .\ (R\ ..\ (P\ w))))\ (pair\ u\ i)$
⟨*proof*⟩

lemma *apply-fun-range*: $\{y.\ \exists x.\ y = (R\ ..\ P\ x)\} = range\ (\lambda\ x\ .\ R\ ..\ P\ x)$
⟨*proof*⟩

lemma [*simp*]: $Disjunctive\text{-}fun\ R \Longrightarrow mono\ ((R\ i)::'a::complete\text{-}lattice \Rightarrow 'b::complete\text{-}lattice)$
⟨*proof*⟩

theorem (**in** *DiagramTermination*) *dgr-data-refinement-1*:
$dmono\ D' \Longrightarrow Disjunctive\text{-}fun\ R \Longrightarrow$
$(\forall\ w\ i\ j\ .\ \models P\ w\ i\ \ \{|\ D(i,j)\ |\}\ SUP\text{-}L\text{-}P\ P\ (pair\ w\ i)\ j) \Longrightarrow$
$(\forall\ w\ i\ j\ .\ DataRefinement\ ((assert\ (P\ w\ i))\ o\ (D\ (i,j)))\ (R\ i)\ (R\ j)\ (D'\ (i,\ j)))$

$\Longrightarrow$

$\models (R\ ..\ (Sup\ (range\ P)))\ \{|\ pt\ D'\ |\}\ ((R\ ..\ (Sup\ (range\ P)))\ \sqcap\ (-(grd\ (step\ D'))))$
⟨*proof*⟩

definition
$DgrDataRefinement1\ D\ R\ D' = (\forall\ i\ j\ .\ DataRefinement\ (D\ (i\ ,\ j))\ (R\ i)\ (R\ j)\ (D'\ (i, j)))$

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

theorem *DataRefinement-mono*:
$T \le S \Longrightarrow mono\ R \Longrightarrow DataRefinement\ S\ R\ R'\ S' \Longrightarrow DataRefinement\ T\ R\ R'\ S'$
⟨*proof*⟩

definition
$mono\text{-}fun\ R = (\forall\ i\ .\ mono\ (R\ i))$

theorem *DgrDataRefinement-mono*:
$Q \le P \Longrightarrow mono\text{-}fun\ R \Longrightarrow DgrDataRefinement2\ P\ D\ R\ D' \Longrightarrow DgrDataRefinement2\ Q\ D\ R\ D'$
⟨*proof*⟩

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*:
$dmono\ D \Longrightarrow Disjunctive\text{-}fun\ R \Longrightarrow \vdash P\ \{|\ D\ |\}\ Q \Longrightarrow DgrDataRefinement1\ D\ R\ D' \Longrightarrow$
$\vdash (R\ ..\ P)\ \{|\ D'\ |\}\ ((R\ ..\ P)\ \sqcap\ (-(grd\ (step\ D'))))$
⟨*proof*⟩

lemma *comp-left-mono* [*simp*]: $S \le S' \Longrightarrow S\ o\ T \le S'\ o\ T$
⟨*proof*⟩

lemma *assert-pred-mono* [*simp*]: $p \le q \Longrightarrow \{.p.\} \le \{.q.\}$
⟨*proof*⟩

theorem (**in** *DiagramTermination*) *Diagram-DataRefinement2*:
$dmono\ D \Longrightarrow Disjunctive\text{-}fun\ R \Longrightarrow \vdash P\ \{|\ D\ |\}\ Q \Longrightarrow DgrDataRefinement2\ P\ D\ R\ D' \Longrightarrow$
$\vdash (R\ ..\ P)\ \{|\ D'\ |\}\ ((R\ ..\ P)\ \sqcap\ (-(grd\ (step\ D'))))$
⟨*proof*⟩

lemma $(R'::'a::\mathit{complete\text{-}lattice} \Rightarrow 'b::\mathit{complete\text{-}lattice}) \in \mathit{Apply.Disjunctive} \Longrightarrow$
$\mathit{DataRefinement}\ S\ R\ R'\ S' \Longrightarrow R\ (-\ \mathit{grd}\ S) \leq -\ \mathit{grd}\ S'$
⟨*proof*⟩

end

References

[1] R.-J. Back. Semantic correctness of invariant based programs. In *International Workshop on Program Construction*, Chateau de Bonas, France, 1980.

[2] R.-J. Back. Invariant based programs and their correctness. In W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, pages 223–242. MacMillan Publishing Company, 1983.

[3] R.-J. Back. Invariant based programming: Basic approach and teaching experience. *Formal Aspects of Computing*, 2008.

[4] R.-J. Back and V. Preoteasa. Semantics and proof rules of invariant based programs. Technical Report 903, TUCS, Jul 2008.

[5] T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In *CSL '02: Proceedings of the 16th International Workshop and 11th Annual Conference of the EACSL on Computer Science Logic*, pages 103–119, London, UK, 2002. Springer-Verlag.

[6] V. Preoteasa and R.-J. Back. Data refinement of invariant based programs. *Electronic Notes in Theoretical Computer Science*, 259:143 – 163, 2009. Proceedings of the 14th BCS-FACS Refinement Workshop (REFINE 2009).